IBM® Information Management Software

Smarter Modeling of IBM InfoSphere Master Data Management Solutions

- Overcome the challenges in master data management integration
- Understand the importance of a methodology
- Gain insights into consistent and comprehensive modelling

Jan-Bernd Bracht
Joerg Rehr
Markus Siebert
Rouven Thimm

Redbooks

ibm.com/redbooks

International Technical Support Organization

Smarter Modeling of IBM InfoSphere Master Data Management Solutions

July 2012

SG24-7956-00

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (July 2012)

This edition applies to IBM InfoSphere Data Architect Version 7.5.2, IBM Rational Software Architect Version 7.5.5.3, and IBM Rational Team Concert Version 2.0.0.2 iFix 5.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

Preface

The concept of master data management is reflected in different manifestations. Master data management can provide an aggregated view of master data maintained in existing systems as a less complex implementation style and then again as a highly complex approach as a transaction hub being a vital component for any integrated use cases and business processes.

This IBM® Redbooks® publication primarily focuses on the complex characteristics in combination with transaction hub implementations and details associated with business modelling challenges, appropriate methods, and tooling to support design and implementation requirements for that type of master data management project style.

For any other implementation style of master data management, such as master data management registry matching and linking facilities or for the small-scale projects of any implementation style, the specified methodology will potentially pose too much overhead respecting the reduced scope and complexity that would be faced under those conditions.

Master data management is the combining of a set of processes and tools that define and manage the non-transactional data entities of an organization. Master data management can provide processes for collecting, consolidating, persisting, and distributing this data throughout an organization.

IBM InfoSphere® Master Data Management Server (InfoSphere MDM Server) creates trusted views of master data that can improve applications and business processes. InfoSphere MDM Server helps you gain control over business information by enabling you to manage and maintain a complete and accurate view of master data. InfoSphere MDM Server enables you to extract maximum value from master data by centralizing multiple data domains and providing a comprehensive set of prebuilt business services that support a full range of master data management functionality.

This publication presents a development approach for master data management projects and for projects based on InfoSphere MDM Server in particular. It focuses on the characteristics of transaction hub implementations and the associated business modeling challenges. It discusses the appropriate methods and tooling that can support design and implementation requirements for master data management in a transaction hub implementation.

This book also discusses the principles of domain-driven design (DDD) and, more generally speaking, model-driven development (MDD). It also describes preferred practices for large and small projects. It is meant as a guide for any type of data-centric solution that needs to be embedded into a service-oriented architecture (SOA) and is not limited to master data management solutions. This book focuses on the design phase of master data management projects. Although it discusses other phases, it does not provide details of the implementation and code generation phases.

The target audience for this book includes enterprise architects, information, integration and solution architects and designers, developers, and product managers.

The team who wrote this book

This book was produced by a team of specialists from IBM Software Group Services and IBM Global Business Services® in IBM Germany.

Jan-Bernd Bracht has more than 10 years of experience as an IT architect. He joined IBM Software Group Services in 2008 and is currently supporting customers in Europe with the integration of master and reference data management systems into their system landscape. He is an accredited Architect for Information Architecture. Prior to joining IBM, he worked as a J2EE and Enterprise Application Integration Architect in various projects in the finance and health industries. For more than four years, he was responsible for the overall architecture of the portal platform of a major German bank. He also has several years of experience in the field of Microsoft technologies.

Joerg Rehr has almost 20 years of experience in the strategic use of emerging technologies across various industries. He is a certified Architect for Information Architectures and MDM Business Architect. Since 2007, Joerg has worked for IBM Software Group Services in Germany supporting IBM InfoSphere Master Data Management Server customers with their integration efforts. Prior to that, he worked in the field of Enterprise Application Integration with a focus on the integration of existing systems. He has extensive experience in J2EE and .NET development environments MDD modeling techniques.

Markus Siebert is an accredited IT Architect for Information Architecture who joined IBM Software Group Services in 2006. Since then, he has worked in projects establishing and integrating master data management solutions for customers in industries such as health care, general stores, and banking. Before joining IBM, he worked as a J2EE Developer at a company developing software for customer care and mobile phone operators.

Rouven Thimm joined IBM in 2007 after graduating with a degree in business information systems. He is currently working as an Application Consultant and Information Architect with IBM Global Business Services. He focuses on information systems, knowledge management, and master data management. As a member of the Business Analytics and Optimization service line, he has broad insight into projects of different sizes. His experience ranges from requirements engineering to actual system implementation using a variety of development environments and working across different industries, including aviation, banking, and defense.

Other contributors

Thank you to the following people who contributed to this IBM Redbooks publication in the form of advice, written content, and project support.

From IBM locations worldwide

- Our management team for making it all possible and allowing us to dedicate time to writing this book, especially considering the difficult economic times in which we currently live:
  - Franz-Josef Arzdorf
  - Peter Starting
  - Guido Düx
  - Reinhard Buchholz
- Our teams, GBS BAO and MDM Services Germany, for being supportive and for providing additional insights gained from many client engagements over many years. We would like to especially thank the following people:
  - Thomas Hoering for coordinating our efforts
  - Corina Mühlig and Sascha Slomka for lending us their expertise in data mapping
  - Michael Sebald who pioneered with us before pursuing other challenges
- Special thanks to Michael Schmut, a member of the MDM Services team, who helped us with the editorial work.
- Special mention for the lab teams in Toronto, Canada, and Hursley, UK, whose experienced engineers supported us by providing deep insights and by reviewing the content that we created. On behalf of these labs, we extend our appreciation to the following people:
  - Lena Woolf
  - Karen Chouinard
  - Catherine S. Griffin
  - Mike Cobbett

- While writing this book, we were also working on customer assignments and gathered inspiration and feedback from a large number of people. Some of our greatest influences are the following people:
  - Michael Klischat
  - Thorsten Gau
  - Peter Münster

From the International Technical Support Organization, San Jose, CA

- Chuck Ballard, ITSO Project Manager
- Mary Comianos, Publications Management
- Emma Jacobs, Graphics Support
- Ann Lund, Residency Administration
- Debbie Willmschen, Technical Writer

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks publications

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

Part 1

Master data management and model-driven development

The chapters in this part build the foundation for the later chapters. This part explains why master data management is an ideal approach for resolving the complexities of master data management implementations. This part includes the following chapters:

- Chapter 1, “Introduction” on page 3

  Chapter 1, “Introduction” on page 3, provides an introduction to the concept of master data management and discusses the methodology and the outline of this book. It also describes the case study and use cases that are used for discussion and examples.

- Chapter 2, “Master data management” on page 25

  Chapter 2, “Master data management” on page 25, builds the foundation for the later chapters in the book. It defines master data and discusses the challenges related to using master data management. It also elaborates on the architectural considerations and the components and functions that are required for support of a master data management project.

- Chapter 3, “Model-driven development” on page 75

  Chapter 3, “Model-driven development” on page 75, draws conclusions from the previous chapter, and then focuses on model-driven environments and adds the business perspective.

1

Introduction

This chapter provides an introduction to the concepts of master data management, model-driven projects, and transaction hub implementations. It also describes the target audience for this book and how to use this book to its full advantage. The chapter introduces a detailed case study and the use cases that were used in the development of this book.

1.1 Basic master data management modeling concepts

This section introduces basic concepts used in master data management and model-driven projects and introduces the methodology used in the development of this book.

1.1.1 Master data management

Often, solutions in the field of master data management tend to be complex in nature. The complexity of a solution depends on many factors, including the implementation style chosen.

Reference implementations are typically less complex when compared to transaction hub implementations. Generally, the complexity of a transactions hub style implementation comes from the master data management systems that are placed at the core of the business and that require an appropriate level of integration. Contrary to other business-related systems, master data management provides core data to a single consumer and, ideally, to all systems on which the business depends that make use of the same data. This approach enables the customer to achieve the envisioned complete view of the business entities.

Master data management can either provide an aggregated view of master data without actually replacing the existing systems (reference-style master data management) or serve as a transaction hub and be a vital component for any integrated use cases and business processes.

In this publication we primarily focus on the transaction hub implementations and details on associated business modelling challenges, appropriate methods, and tooling to support design and implementation requirements for that kind of master data management project style.

For any other implementation style of master data management, such as master data management registry matching and linking facilities or for the small-scale projects of any implementation style, the specified methodology will potentially, but not necessarily, pose too much overhead respecting the reduced scope and complexity that you are facing under those conditions.

InfoSphere MDM Server is a major software component to realize any master data management manifestation, and InfoSphere MDM Server was the product that we implemented in many projects whereby the methodology and tooling can be adopted to other products in a similar fashion.

When integrating with other systems, all of which must continuously fulfil their business purpose, all systems must now operate on a single source of truth (which is the one true information source). The integration with other systems propels the need to set up projects to get data in and out of the master data management system, either in batch mode or through an online synchronization. In addition, other systems, such as back-office user interfaces, will access and change data directly on demand.

Thus, the focus is on data as well as system integration. Achieving both reliable core data and painless integration implies thorough design and precise service contracts. Throughout the course of an integration project, the following activities need to be completed:

- Identify the master data elements and map those elements to a data model of the master data management system, such as InfoSphere MDM Server. Also, build additional data elements if necessary.
- Map data between source systems and the target master data management system.
- Define system behavior.
- Create business services where the services of a master data management system do not suffice.
- Describe and implement the logic for those business services.

1.1.2 Model-driven projects

While there are many ways achieving the goals described, this book focuses on the model-driven approach. Projects need to progress quickly and smoothly. You can use manual design and implementation processes, but an integrated design and implementation approach can save time after you create the initial models, provide reproducibility, and improve quality.

In addition to these distinguishing factors, model-driven projects are easier to maintain, because the model acts as the base for the source code and represents accurate and up-to-date documentation. Having worked on a number of model-driven projects, we noticed that the same patterns resurface. We also have experience with a large-scale project where developers had to be phased out of the project ahead of the scheduled time due to lack of work. The lack of work was a result of better-than-expected quality, which resulted in fewer bugs identified by the test teams and, consequently, fewer defects to be fixed.

To begin, establish the model at the center of the project and have it adopted by everyone, which can be a challenge. Next, train the people who will be working on the different parts of the overall model. A traditional implementation project requires more developers than designers. However, a model-driven project requires more skilled designers than modelers, because source code is generated directly from the models as well.

Establishing a model-driven methodology and setting the project up can be cumbersome. It can be difficult to prove the effectiveness of this methodology when compared to a more traditional implementation methodology. Project team members might challenge the speed at which models are delivered. They might point out that whereas you are still creating models, they have crafted portions of the required source code.

Although these points are valid and although model-driven projects do require some ramp up time, the project will gain momentum later. A model-driven project is better positioned to handle change because you can determine the impact that a change might have in advance. You can evaluate the model, determine any side effects, and thus reduce possible implementation errors.

1.1.3 The methodology used in this book

During the course of writing this book, we had the chance to visit client environments and work through the various aspects covered in the book. Many clients struggle with similar issues. Regardless of the tool that is used, the ideas that we outline in this book provide a head start when working with a client engagement. Even if you have to determine the details, adopt the tools, and look for certain features, with this book in hand you will understand where to start, which points are most critical, and which fundamentals to lay out.

Beyond these guidelines, clients also shared with us information about their previous attempts at model-driven projects, their goals, and their issues. The concepts that we were able to introduce into their models, such as stereotypes, separation into analysis and implementation models, issue and change tracking within the respective models, and glossary aspect, were greatly appreciated.

Figure 1-1 provides a high-level overview of the methodology presented in this book.

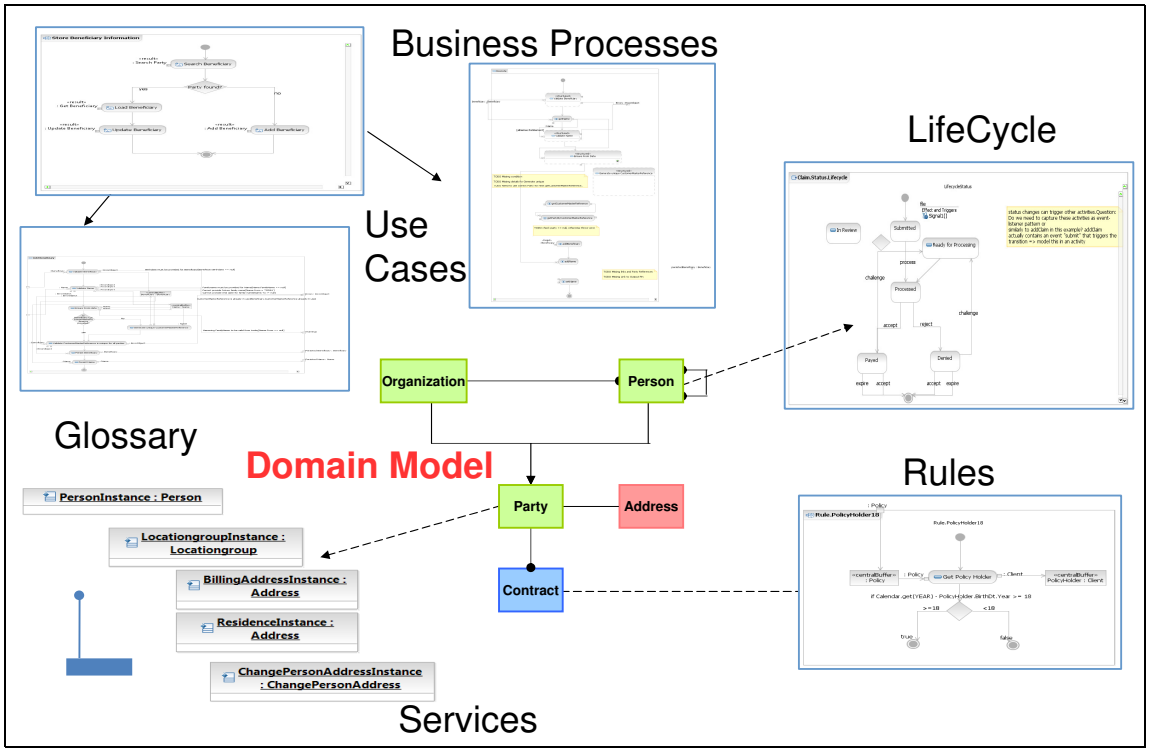

Figure 1-1 Connections and dependencies between models

Figure 1-1 illustrates an integrated set of models that provides a distinguishing differentiator from other methodologies. The business-centric model is at the heart of this methodology, as manifested in the *domain model*. It describes the business in conjunction with the *glossary*. *Business processes* and *use cases* effectively manipulate the data in the domain model and, therefore, are tightly coupled with it. Similarly, the *lifecycle* and *rules* are directly related to the domain model and, therefore, are directly attached to the domain objects.

This scenario provides a functional service integration layer against the master data management system, and describes the services that form such a layer. Again, these services work on domain objects but are also tied with use case descriptions so that large pieces of service control code can be generated eventually through the established models.

The goal is to pave the way for a model-driven master data management solution design and implementation methodology. Because master data management projects revolve around business domains, these types of projects are predestined for a domain-driven design (DDD) approach. This approach focuses on the business domain, the issues and behavior that revolve around that domain, and how this approach leads to a functioning system.

The approach taken in this book

This book incorporates the principles of domain-driven design and, more generically speaking, model-driven development (MDD). It is based on experiences gained from large master data management solution projects and, therefore, describes preferred practices for both large and small projects. Although following this methodology is optional in smaller projects, larger projects can experience benefits in terms of speed, accuracy, and quality. This book is meant to serve as a guide for any type of data-centric solution that needs to be embedded into a service-oriented architecture (SOA) landscape and is not limited exclusively to master data management solutions.

This book describes the application of IBM Rational® Software Architect and IBM InfoSphere Data Architect in a model-driven implementation of InfoSphere MDM Server. It describes how to use these products to apply the model-driven design approach using a continuous example. The book also describes approaches to specific problems originating from real-life project experience using Rational Software Architect, InfoSphere Data Architect, and InfoSphere MDM Server.

The examples in this book use the following software configurations:

- IBM InfoSphere Data Architect V7.5.2
- IBM Rational Software Architect V7.5.5.3
- IBM Rational Team Concert™ V2.0.0.2 Interim Fix 5

Version note: Using InfoSphere Data Architect V7.5.3, instead of InfoSphere Data Architect V7.5.2 in combination with Rational Software Architect V7.5.5.3, produced unexpected behavior, issues with diagrams failing to draw, and a few other minor deficiencies.

Also, the scenarios in this book do not use Rational Software Architect V8, because at the time of writing there was no InfoSphere Data Architect equivalent. Rational Software Architect V8 and InfoSphere Data Architect V7.5 are incompatible.

1.2 Target audience

In most cases, you will not want to read this entire book but will want to skip directly to the information that you find most useful for your circumstances. The content of this book is intended for the following roles:

- Enterprise architects

  Enterprise Architects can gain insights into the complexities of integrating master data management systems in an enterprise. This book provides information about how to determine the impact that any decision regarding integration patterns will have on the solution that you to govern.

  Although this book does not focus on enterprise architecture modeling, the role of the enterprise architect is vital to ensuring success in the model-driven project. This role has more influence on the overall outcome of the solution than any other. Therefore, you will find useful information in all the chapters.

- Information, integration, and solution architects and designers

  People in these types of roles might be interested in the discussion about MDD and DDD in Chapter 2, “Master data management” on page 25, and Chapter 3, “Model-driven development” on page 75.

  In addition, you can learn about this methodology and its theoretical aspects as presented in a comprehensive model in the following chapters included in Part 2, “Smarter modeling” on page 129:

  - Chapter 4, “Smarter modeling in practice” on page 131
  - Chapter 5, “Sources” on page 245
  - Chapter 6, “The modeling disciplines” on page 261
  - Chapter 7, “The models” on page 281
  - Chapter 8, “Mappings and transformations” on page 411
  - Chapter 9, “Deliverables” on page 461

  This book does not focus on any single model. Instead, it emphasizes how to make all individual models work in unison towards one goal, forming a single, consistent model that describes the master data management system.

- Developers

  With an MDD approach, code artifacts are generated all the way to the code representing the service flow logic. Most other approaches stop at the generation of static classes and wrappers that are derived from appropriate class and protocol models. The approach that we describe in this book generates the service flow logic code, which represents a dynamic aspect.

Although this book does not provide details about code generation, it does discuss important aspects that are related to the design of dynamic code generation. It includes information regarding suggestions for stereotypes, profile extensions, and use of UML element properties. These features describe all relevant details of the problem domain and are available when you add true code generation at a later point in time.

For information about the modeling approach, read the chapters in Part 2, “Smarter modeling” on page 129.

- Project managers

  Project managers will find the information in Chapter 2, “Master data management” on page 25, and Chapter 3, “Model-driven development” on page 75, most useful to gain insights into the challenges of master data management system implementations. In particular, Chapter 3, “Model-driven development” on page 75, includes a discussion on using domain insights for project managers.

  Chapter 10, “Planning for your master data management project” on page 491, also provides a supporting discussion about the toolset based on a number of Rational products.

1.3 How to read this book

This book is organized into the following parts:

- Part 1, “Master data management and model-driven development” on page 1

  The chapters in this part build the foundation for the later chapters in this book and explain why an MDD solution is an ideal approach for tackling the complexities of master data management implementations. This part includes the following chapters:

  – Chapter 1, “Introduction” on page 3

    This is the chapter that you are reading now. It provides an introduction to the concept of master data management and discusses the methodology and the outline of this book. It also describes the case study and use cases that the book uses.

  – Chapter 2, “Master data management” on page 25

    This chapter builds the foundation for the later chapters in this book. It defines master data and discusses the challenges related to using master data management that need to be overcome. It also elaborates on the architectural considerations and the components and functions that are required in support of a master data management project.

- Chapter 3, “Model-driven development” on page 75

  This chapter draws conclusions from the previous chapter. It focuses generically on model-driven environments and adds the business perspective.

- Part 2, “Smarter modeling” on page 129

  The chapters in this part represent the practical guidelines for building a set of models that are interrelated and interconnected and that work together toward a single common goal. This part includes the following chapters:

  - Chapter 4, “Smarter modeling in practice” on page 131

    This chapter brings together the information presented in Part 1, “Master data management and model-driven development” on page 1. It acknowledges the variety of systems, data models, and their transformations, as well as the mappings and service design, while also establishing one consistent and highly integrated model. The goal is to provide the foundational models that can be used for automated code generation at a later time.

  - Chapter 5, “Sources” on page 245

    This chapter discusses the InfoSphere MDM Server sources.

  - Chapter 6, “The modeling disciplines” on page 261

    This chapter describes the modeling methodology, which is based on a metamodel. The metamodel describes the artifacts that are required and how these artifacts relate.

  - Chapter 7, “The models” on page 281

    This chapter discusses models and, in particular, the categories into which the models fit.

  - Chapter 8, “Mappings and transformations” on page 411

    This chapter discusses mappings and transformations. Mappings are one of the core deliverables of any project that involves InfoSphere MDM Server.

  - Chapter 9, “Deliverables” on page 461

    This chapter describes everything related to the final deliverable, including reporting and accompanying activities.

- Part 3, “Planning considerations” on page 489

  This part discusses considerations as you begin planning your master data management project. It discusses challenges and implementation strategy. It also provides information about setting up and executing a project and discusses the tool selection. This part includes one chapter, Chapter 10, “Planning for your master data management project” on page 491.

1.4 Case study: A fictitious insurance company

The examples in this book use a single example case study so that the discussion can center on the models that are required and so that we can provide sample models. The sample models provide images that are used throughout the chapters of this book. These excerpts describe modeling concepts and, more importantly, demonstrate the interrelated and interconnected nature of the various models. These examples illustrate the modeling approach that we describe in this book in the context of building master data management solutions.

The case study uses a fictitious insurance company. Although this case study is simplified, it includes numerous aspects of real-life master data management implementations. The limited scope of the case study allows us to focus on the most relevant aspects.

1.4.1 Business scenario

The fictitious insurance company has decided to embark on a master data management strategy. It is planning to implement a master data management hub to achieve a *complete view* (or a 360-degree view) of its customers so that it can pursue cross-selling and up-selling opportunities. The complete view provides a complete picture of all data related to persons and contracts from a single trusted source, instead of multiple sources with redundant data that deviates in respect to correctness and accuracy.

Currently, the fictitious insurance company has its customer data distributed over two systems. *System A* is an older system, but it effectively stores accounts. *System B* was introduced by the fictitious insurance company to provide basic CRM functionality. System B stores individuals as the main entity. The fictitious insurance company operates a number of applications that access system A, system B, or both systems. However, none of the applications can provide a complete and consistent view. Data is available only from either the viewpoint of the account or from the viewpoint of the individual.

Figure 1-2 shows a customer view from an account perspective.

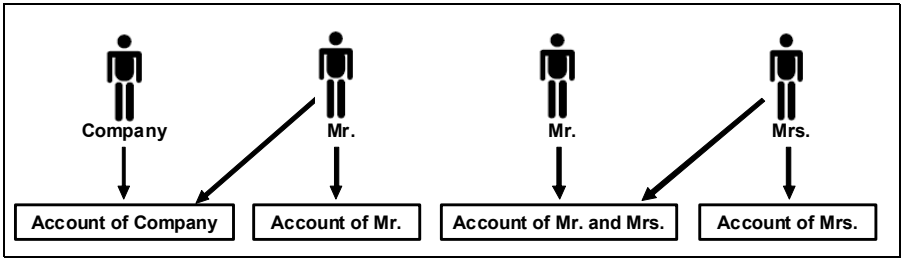

Figure 1-2 View of the customers from an account perspective

Using the account perspective of system A provides no easy way of knowing that the owner of “Account of Mr.” also plays a role in respect to “Account of Company” and “Account of Mr. and Mrs.” Being aware of this relationship, however, is crucial to perform a proper risk assessment or to identify whether the company can cross-sell or up-sell to this individual.

From a person perspective as provided by system B, because the account system stores each person separately, there is no easy way to know the relationship between the various accounts that Mr. and Mrs. maintain. Thus, the fictitious insurance company’s ability to assess the overall risk and to realize potential cross-selling or up-selling opportunity identification is impacted negatively.

1.4.2 Envisioned solution

To achieve a complete view, the fictitious insurance company introduced a master data management system, as illustrated in Figure 1-3. This system consolidates the data from systems A and B, removes redundancies, ensures integrity, and effectively becomes a source of trusted information for the fictitious insurance company.

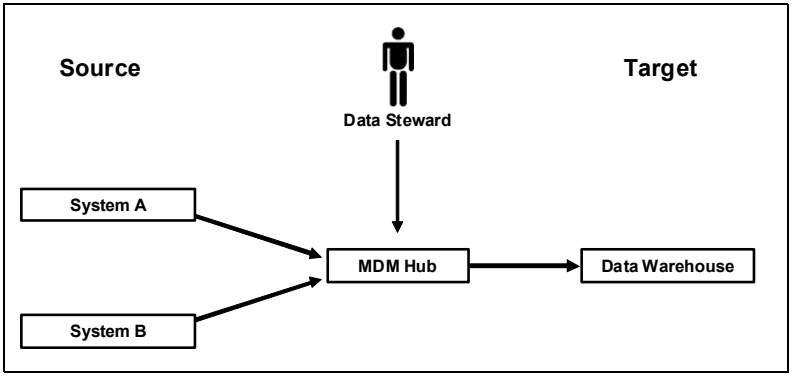

Figure 1-3 Sample system context

The master data management system synchronizes the original source systems outbound in a complex implementation style. The respective downstream systems can then continue to operate against the original source systems for the initial phase of the master data management integration, prior to switching system interfaces in a second phase of the master data management program. Data stewards ensure that the new system adheres to well-defined data governance rules and procedures. The consolidated data will then be used to provision the data warehouse.

This system achieves the goal of providing a single system that maintains a single view on each person and that correlates that single view with all of the person's accounts. The scenario of the same person involved in many accounts now looks entirely different, as depicted in Figure 1-4.

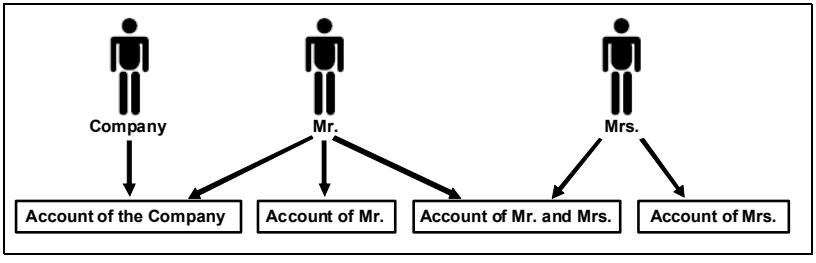

Figure 1-4 360-degree view of a customer

Now that the person of Mr. is de-duplicated, the risk becomes clear, because all of the accounts, including the company's and the joint account with the person of Mrs., come into the equation.

1.4.3 Description of the case study models

The fictitious insurance company has a number of models in place that describe their current systems. We refer to these current systems as the *customer source models*. Based on these source models, the fictitious insurance company can design and implement a solution.

The fictitious insurance company provides the following source models:

- Domain model
- Business processes
- Use cases

Domain model

The *domain model* describes the logical data from a business perspective and is independent of its physical implementation, as illustrated in Figure 1-5.

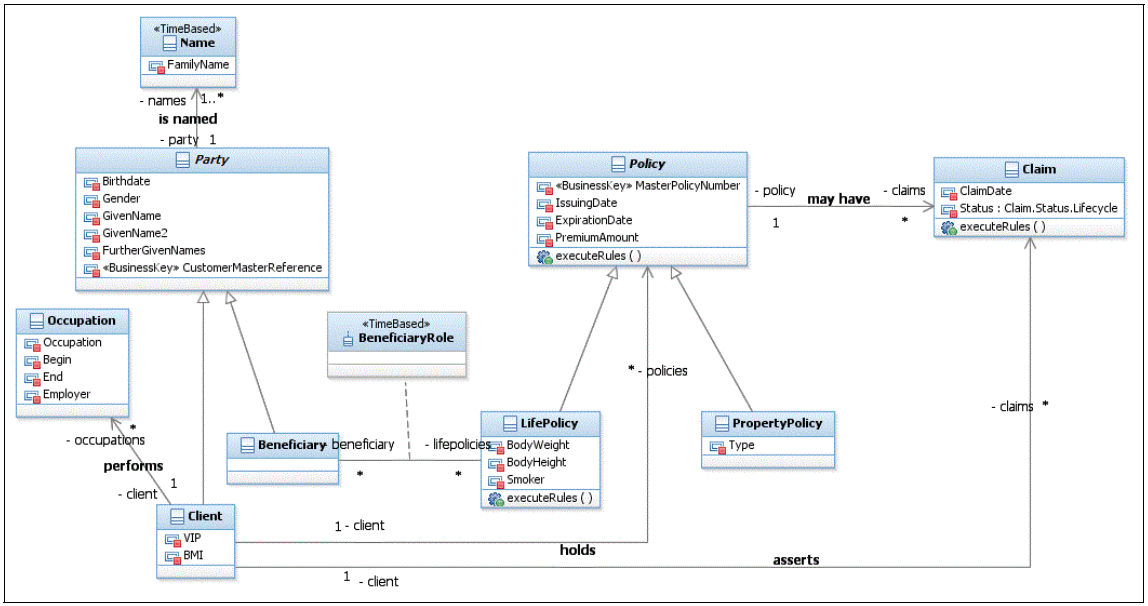

Figure 1-5 Sample domain model

The descriptions of the individual entities that make up the domain model provide the following business-related information about the domain:

- Party

  A party is an individual who is somehow related to the fictitious insurance company's business. The party is one of the core entities of the system, and it has associations to almost every other entity. The party has the following specializations:

  - Client
  - Beneficiary

  The party itself is abstract.

- Party name

  A party can have different names over its life span, for example, because of a marriage or some other reason for a name change of an individual. However, it is important to know at which point in time a certain party has had a certain name. The validity information needs to be stored on that entity, making it a *time-based* entity. Although a party can have many names over time, you can apply limits by allowing only one name at a given point in time.

- Client

  A client is a party who holds a policy. In most cases, the client is the insured person or the individual who owns the property that is covered by the property insurance. Other types of clients are possible, but this scenario limits the client to one of these two options.

- Occupation

  The occupation describes job-related information about a certain client. A client can have one or multiple occupations and possibly can have more than one occupation at the same time. The scenario does not have a requirement to store the occupation as a time-based object.

- Beneficiary

  A beneficiary is a party who benefits from a life insurance policy in case the insured person passes away before the life insurance policy is due. The beneficiary is different from the client because the beneficiary has different associations to other objects. In addition, less information is stored about the beneficiary. A beneficiary can benefit from more than one life policy.

- Policy

  A policy is a certain insurance contract between the fictitious insurance company and one of its clients. The policy itself is an abstract with two (non-abstract) specializations:

  – Life policy
  – Property policy

  Each policy carries a master policy number that uniquely identifies a certain policy within the fictitious insurance company.

- Life policy

  A life policy is an insurance policy for the life of a certain individual. It has associations to the beneficiary and to the client.

- Property policy

  A property policy insures a certain property. A property policy does not have an association to a beneficiary.

- Claim

  A claim is asserted by a client for a certain policy.

- Beneficiary role

  The beneficiary role associates the beneficiary with a certain life policy. A beneficiary can benefit from multiple life policies, and a life policy can have numerous beneficiaries. The beneficiary role can change over time, and this entity is time-based.

Business processes

This section outlines one representative business process. It does not include a complete business process model, because that type of model does not add value with regard to the modeling methodology.

Figure 1-6 depicts one exemplary business process.

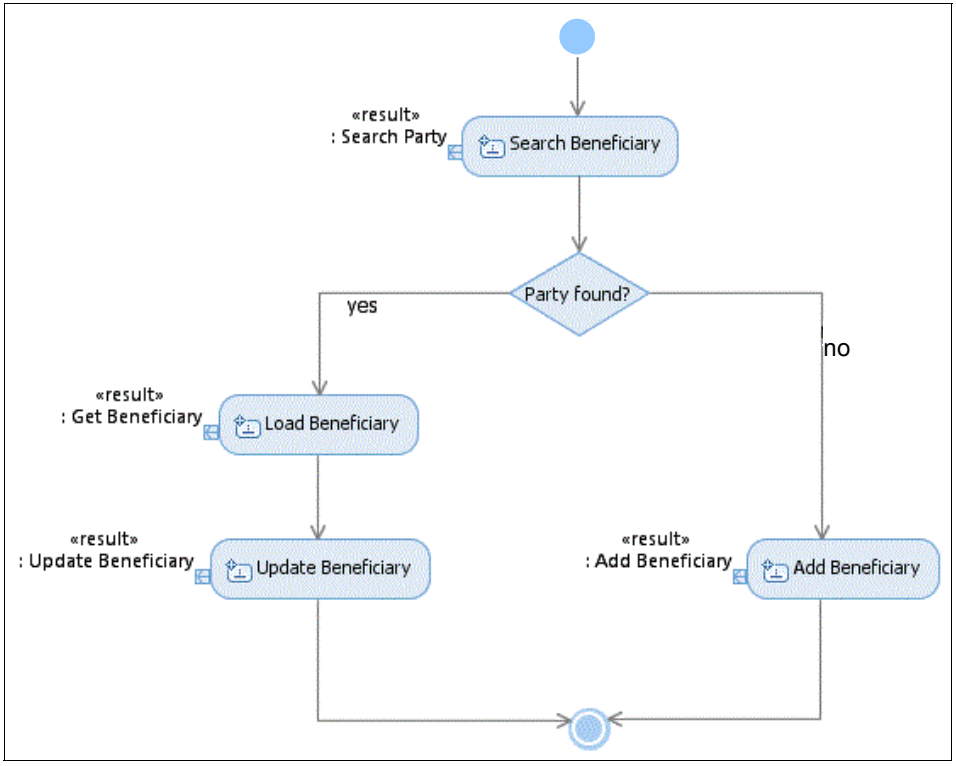

Figure 1-6 Sample business process - store beneficiary information

This process describes the way that the user first searches for a party who is or will be stored as a beneficiary. If the party is found, the existing beneficiary is loaded before being updated. If the party is not found, a new beneficiary is added to the system.

Real-life scenarios include additional details to such a process. For example, the process might include a manual selection after the search and manual user interaction through the user interface, which creates or modifies the desired beneficiary data prior to the invocation of the add or update use cases.

Use cases

This section outlines a few exemplary use cases. Due to the number of use cases identified for this case study, the use cases are group into these domains:

- Party
- Policy
- Claim

Party

The party use cases provide a description for all information that is related to client or beneficiary interaction. Figure 1-7 illustrates a model of this use case from a high-level point of view.

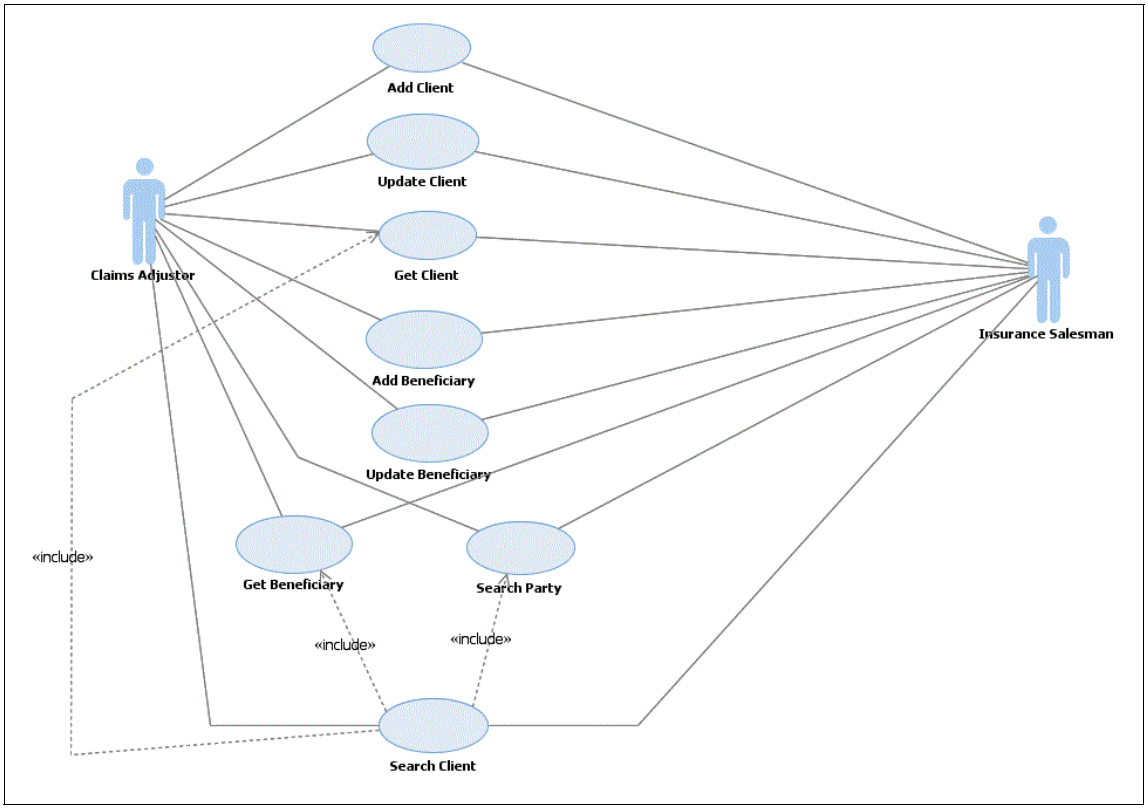

Figure 1-7 Sample use case - subdomain party

This use case diagram provides a description for the interaction of the actors with the use cases. Most notably, meaningful use cases are provisioned and are close to how expectations for the services will be offered. Notice that this use case has includes, which is a normal approach to structuring use cases and providing for some input towards reuse on that level. Also, it uses an abstract *search party* use case that handles both clients and beneficiaries.

The following is an exemplary representation of one of the use cases, which is also invoked from the business process that was outlined earlier:

- Use case

  Add beneficiary.

- Primary actor

  Claims adjustor, insurance salesman, or system performing corresponding function.

- Supporting actors

  None.

- Stakeholder and interests

  None.

- Input

  Beneficiary, Name (1..1).

- Output

  Beneficiary, Name (1..1).

- Pre-conditions

  None.

- Post-conditions

  – Success: The specified beneficiary and the related objects are persisted and returned with possible warnings.

  – Failure: An adequate error message is returned, and no modifications are made to the system.

- Trigger

  A new party is introduced to the fictitious insurance company in the role of a beneficiary. The required data is collected, entered into the system, and sent for processing.

  The main success scenario is as follows:

  a. Validate the beneficiary.

     If any of the following attributes are not provided, return an error:

     “{fieldname} must be provided for beneficiary**”:** Birthdate**,** GivenName

b. Validate the name.

   i. If no Name object is provided or if `FamilyName` is not provided, return an error:

      "`Familyname is required to register party`**"**

   ii. if From is provided and >Today, return an error:

      "`Cannot provide future family name`**"**

   iii. if To is provided, return an error:

      "`Cannot provide end date for family name`**"**

c. If From is not provided for name, set to Today and return a warning:

   "`Assuming family name to be valid from today`**"**

d. If `CustomerMasterReference` is not provided, generate by combining today's date (*yymmdd*), the party's birth date (*yymmdd*), the first letter from `GivenName`, the first letter from `FamilyName`, and a 4-digit numeric code starting with `0000`, which is reset daily and incremented per party.

e. Validate that CustomerMasterReference is unique.

   If the number identifier is not unique, return an error:

   "`CustomerMasterReference is already in use`**"**

f. Persist the beneficiary and name.

g. Return the beneficiary and name.

- Extensions

  None.

- Variations

  See subordinate lists in the main success scenario. For example, "1.a)" represents a variation of the success flow described by "1".

- Special requirements

  None.

Figure 1-8 shows the visual representation of the use case low logic.

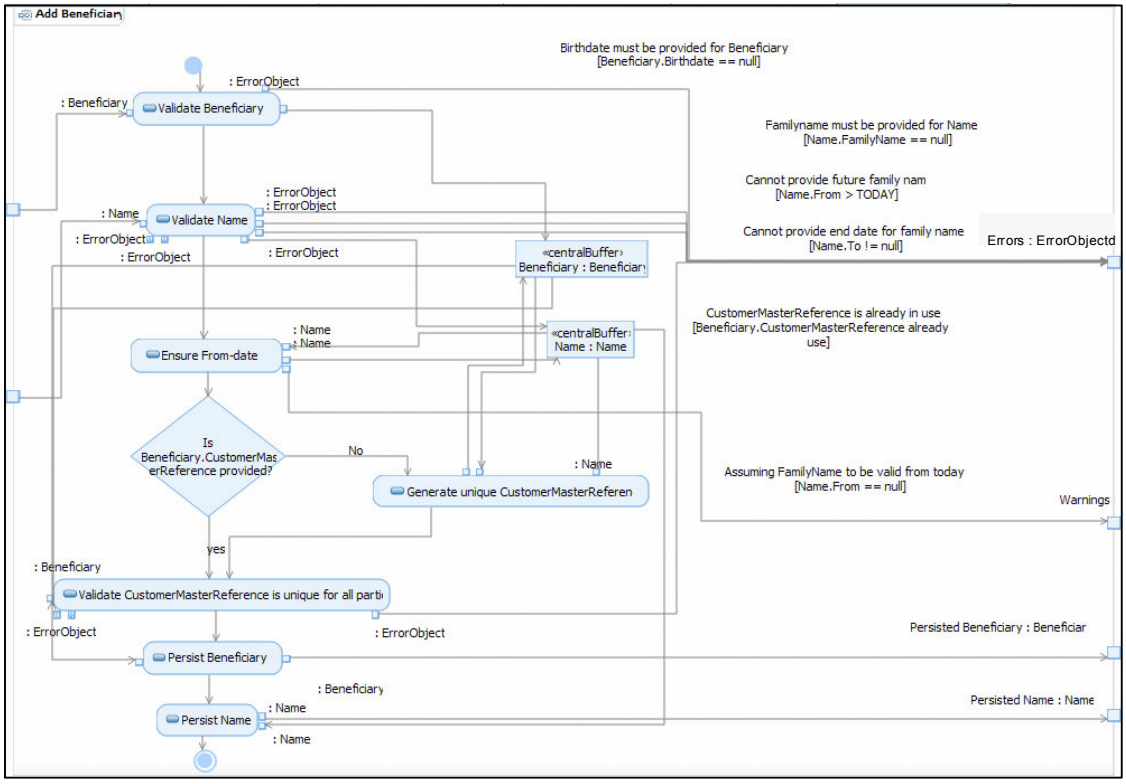

Figure 1-8 Sample use case - add beneficiary

It is here that we divert from the usual way of modeling use cases. Typically, you find use cases only in written form supported by the high-level representation that describes the interaction between actors and use cases. We have provided the flow logic in an activity diagram, which later derives and generates further models from it.

Policy

The policy use cases provide a description for all policy-related interaction. Figure 1-9 illustrates a model for this use case from a high-level point of view.

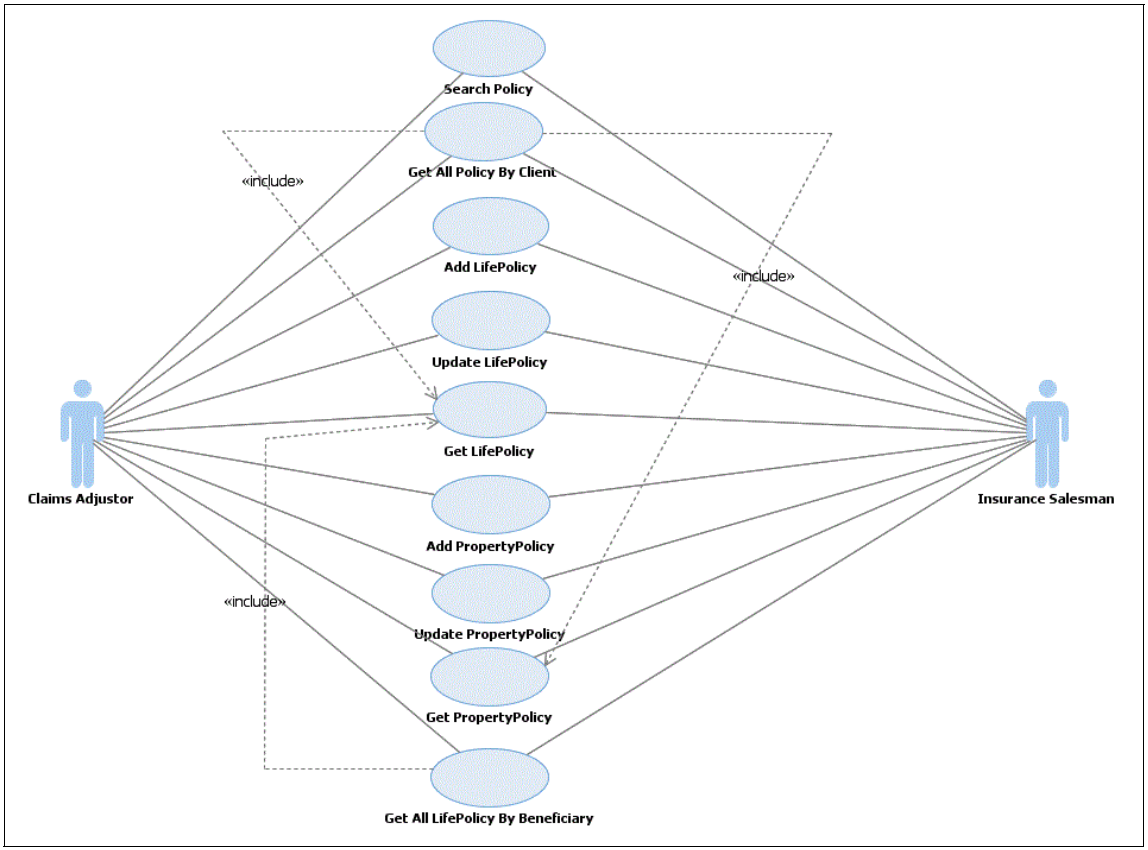

Figure 1-9 Sample use cases - subdomain policy

The descriptions and other details for this use case are similar to the party-related use cases.

Claim

The claim use cases provide a description for all claim-related interaction. Figure 1-10 shows a model of this use case from a high-level point of view.

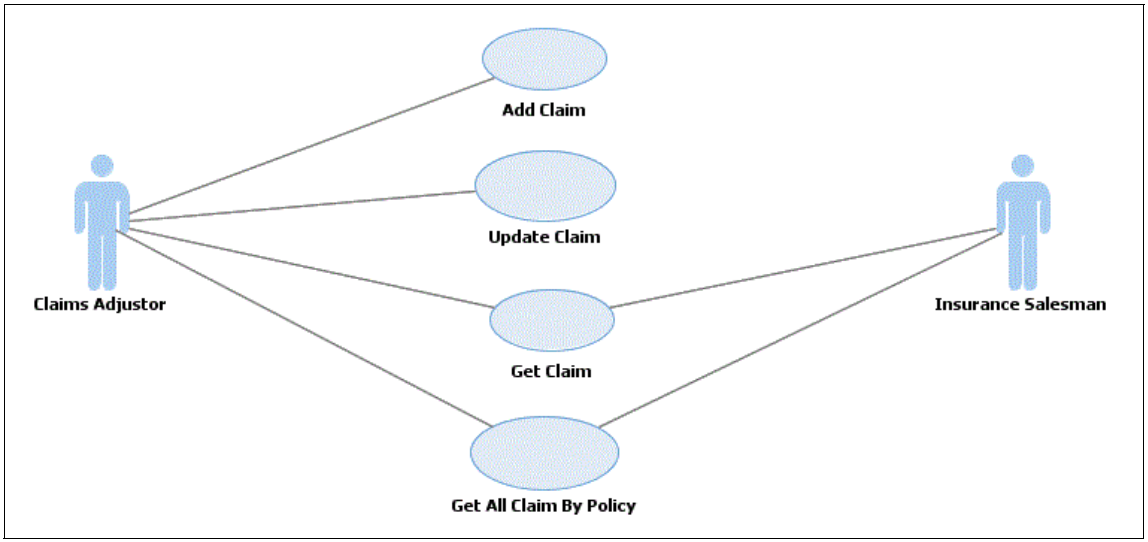

Figure 1-10 Sample use cases - subdomain claim

The descriptions and other details are similar to the party-related use cases.

2

Master data management

Master data management is a complex topic. It is often mistaken for a product or a project. However, master data management cannot be represented by a single technology, activity, or person. Instead, it is a *holistic approach*, a strategic program that is deeply linked to the way an enterprise does business. A master data management implementation also poses specific challenges, including challenges that revolve around the following components:

- Data
- Processes
- Organization
- Solutions

This chapter discusses the considerations when implementing a master data management solution. It also provides an overview of the concept of master data management and explains its place in the overall architecture.

2.1 Considerations for implementing a master data management solution

The core feature in any master data management implementation is the data itself. Thus, when implementing a master data management solution, you need to consider a development approach that is suitable for describing such data manipulation. You need an approach that describes all relevant aspects about the data, including mapping and transforming data between systems and creating a business services external system to interact with the master data management system.

In addition, this data manipulation must use a consistent method that is easy to understand and maintain. This consistent method is especially important when working on a large-scale project, which can often be incomprehensible to any single person. This book focuses on this aspect of implementation and provides insights and reasoning as to why model-driven development (MDD) can be beneficial.

Master data management is not only about the data. It is also about the business processes that manipulate this data. Without a master data management system, data manipulation often happens in an inconsistent way. Thus, you also need to consider the business processes and use cases and introduce models that make it easier to describe system behavior in a consistent and reusable way. These models need to be sustainable for continued development of a new system that sometimes replaces or, more often, works in unison with many other systems. In many cases, you will have to modify the surrounding systems to be able to integrate with the new master data management system. The models presented in this book take this integration into consideration.

When implementing a master data management solution, also consider the organizational measures that need to be addressed with the introduction of this new system. The introduction of data governance is a preferred practice. Although this book does not focus on data governance, it does discuss the aspects of data governance that are relevant to the modeling approach described in the included examples.

In addition, you need to define the business vocabulary from the beginning of a master data management project. As discussed in Part 2, “Smarter modeling” on page 129, you need to introduce a glossary and support it through the introduction of a business domain model. Organizational measures start from the top in understanding the relevance and importance of defining the strategy that drives a master data management project. These measures also include measures to constantly monitor and improve on the quality of the data. You also need to consider the inclusion of data analysis, preferably before starting the master data management endeavor, so that you are certain that you are building a system that contains accurate data.

It is important to select the right products to implement such a complex system, which usually operates at the heart of a business and, consequently, which will need to be integrated with other existing systems in the current system landscape.

This book and the models that are described and contained within aim to provide interoperability between all products and systems. Keep in mind, however, that no one-system-fits-all offering is available. Each vendor focuses on certain industries, organizational structures, or business problems. Consider carefully the product that is best suited for your business.

After you consider these factors, you can begin the project implementation for a master data management solution. The examples and models in this book depict master data management as a discipline that comprises many products, includes organizational adoption and system integration, and, most importantly, provides a strategic vision.

2.2 The definition of master data

Before you look into modeling aspects, you first need to understand the concept of master data and what it really represents. Understanding the true meaning of master data unlocks a common understanding in regard to its complexity and possible solutions for it.

This book uses the following formal definition for master data:

> Master data management encompasses the business processes and the technical and data integration architecture to create and consistently maintain a “system of record” for core business entities across disparate applications in the enterprise.

Thus, master data is defined as the facts that describe the following core business entities:

- Customers
- Suppliers
- Partners
- Products
- Materials
- Bill of materials
- Parts
- Chart of accounts
- Locations
- Employees

Master data management consistently defines and manages the operational business data entities of an organization that are not considered business transactions.

Master data represents the key information that an enterprise has. It spans many domains, such as customers, products, accounts, locations, and so forth. Implementations that are focused solely on customers are known as *Customer Data Integration* (CDI). *Product information management* (PIM) represents the corresponding discipline that focuses solely on the product domain. Because most master data management systems provide multi-domain capabilities and focus on more than just a single domain, these specializations do not play a significant role in the market or in the remainder of the discussion in this book. The remaining discussions in the book regarding master data management always include CDI and PIM.

An issue with master data is that prior to the introduction of a dedicated master data management system the master data is not always gathered in a single system. Instead, it is spread throughout many currently existing systems within an organization, as illustrated in Figure 2-1.

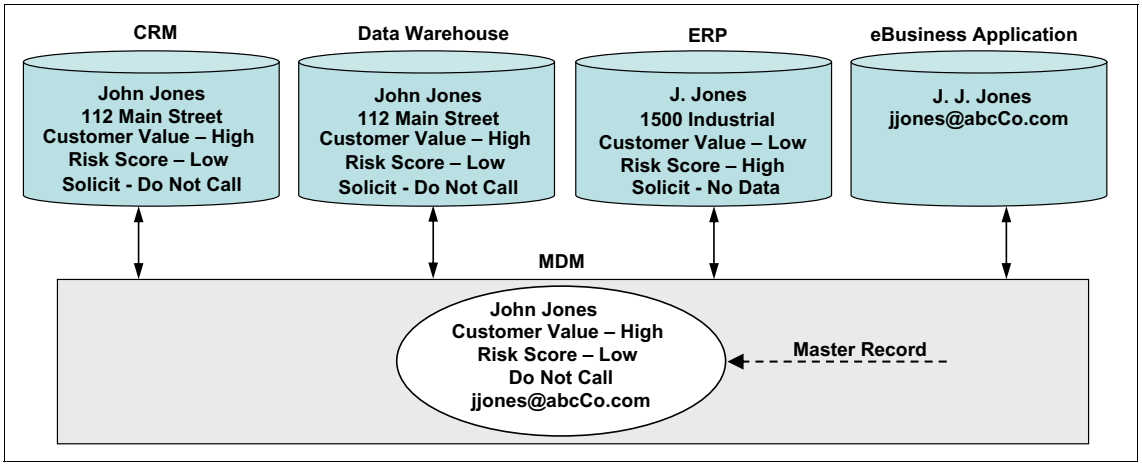

Figure 2-1 Definition of master data and the master record

In this example, the data is spread across a *customer relationship management* (CRM) system, an *enterprise resource planning* (ERP) system, a data warehouse, and a custom e-business application. Most entities, for example, the customer name and addresses, are spread across disparate systems. Some attributes that make up an entity can also be distributed, as illustrated with the customer's address, where private addresses are held in one system and business addresses are held in another system. To make matters worse, even where the same entities and attributes are used, each system uses its own business processes that manipulate this data in different ways. As a result, different attribute values are in different systems based on the non-standard processing within each system.

Many business decisions are based upon master data, which is why master data is used widely throughout numerous business processes. This use of master data is key to your investigation when considering a master data management implementation. Its use is not limited to a single business unit or a single system within the enterprise. Instead, large enterprises evolve over time, as the number of systems that support the business evolve. The real issue with master data is that each of these systems implements business processes that differ from other systems. It is these differences that cause inconsistencies to the master data.

Examples of these types of these differing processes include:

- Using different data standardization procedures
- Varying business rules
- Applying new semantics to the same attribute

Even the slightest differences in standardization procedures can cause different values to be calculated and persisted. For example, one system might standardize street names to “St.”, whereas another system uses the value “Street”. Even though the same street name is entered in both systems, the different standardization rules cause a deviation in that data right away. You can observe a similar affect with varying business rules. Although these rules do not actively change or manipulate data, they still can prompt an exception in one system, whereas the other system is liberal enough to accept this data.

As is often the case with existing systems, existing attributes are reused or misused in a different context. Supported by some other flag that indicates that this attribute has a different meaning than originally intended, you can experience semantically different data creeping into the range of attribute values.

These types of variations give a single attribute a completely different meaning or cause a different value to be processed. The result is an incredibly difficult and complex data repository that does not allow for easy application of consistency rules. Instead, the data requires an odd exception to be embedded to address any of the factors that can cause inconsistencies as outlined previously.

Figure 2-2 illustrates the business process at the root of the problem for inconsistent data values across different systems, even if it is the same attribute stored in those systems.

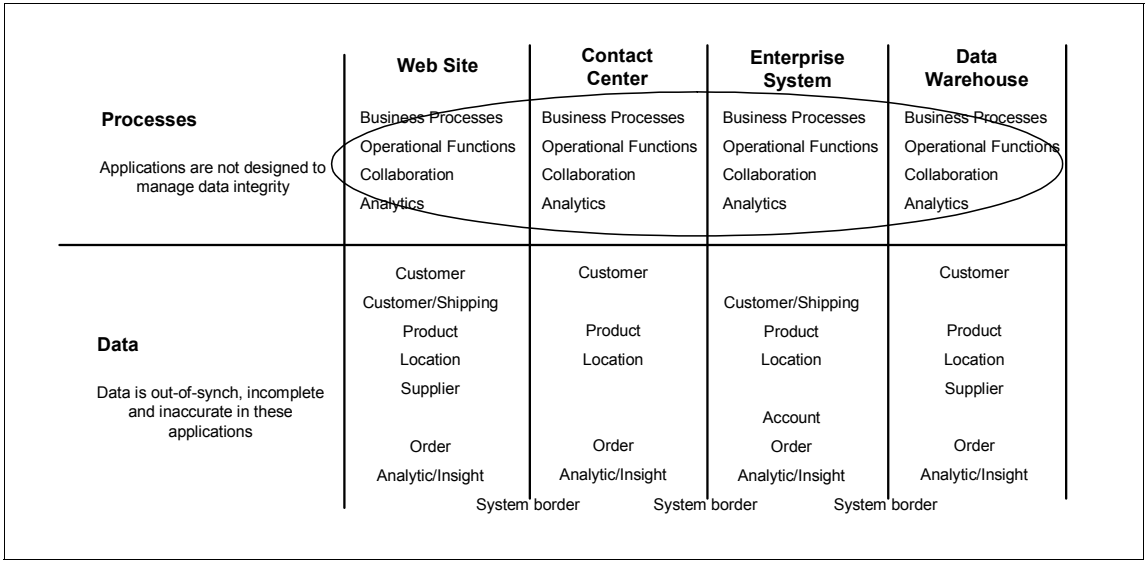

Figure 2-2 Business processes and operational functions as cause for inaccurate data

The consequences that arise from these inconsistencies are now easy to see, although it is more difficult to judge the impact that these inconsistencies might have on the enterprise. The impact can be as simple as two reports showing different data, but imagine what happens if one report is used to validate a critical compliance report. The penalties for producing inconsistent data in regulatory compliance reports can be high. Another impact is in maintaining relationships with enterprise customers where inconsistent data might leave the customer confused, unhappy, or convinced that the data is not being properly and thoroughly handled. Both examples represent reputational risk.

Other complex and costly issues can arise when a certain business process changes. In effect, all other business processes that deal with the same data must also be changed to guarantee consistent attribute values across all processes and systems. If the business processes are not changed, you are again faced with a deviation of the master data across those systems.

These types of inconsistencies are where master data and the respective master data management system come into the equation. Such common data issues can be taken care of by a single system, providing common processes. Then changes are applied only once, and as a result all processing systems rely on and operate on the same accurate data.

Transactional data, in comparison, describes any business that anyone has done with the enterprise, such as logging money on to an account and ordering goods or supplies. Because there are dedicated systems for exactly such processing, the master data management system provides the contacts data, for example, from the *supply chain management* (SCM) system, on which it then executes the processing of the supply order. Following such an approach stresses the fact that the introduction of a master data management system does not replace any single system that deals with master and transactional data at the same time. Systems dedicated to transactional processing, such as order processing, are still required. However, instead of storing their own customer data to provide names and addresses or product definitions for an order, these systems now rely on the provisioning of such data from a dedicated master data management system.

A master data management system will not handle customer transaction data and should never do so. The benefit of the master data management system is that if master data changes, it is changed only once in a central location, the master data management system. Other systems potentially interested in that data can receive notification about the nature of the change and trigger appropriate actions from it. Business rules that are common to all systems dependent on this data can then be implemented on the master data management system itself, and all other logic should continue to remain outside of the master data management system and the systems to which the logic is most relevant.

2.3 Master data management and its place in the overall architecture

Determining the fit of a new master data management system into your existing system landscape is one of the first challenges that you encounter when you decide on the implementation of a master data management system. Further questions arise in quick succession. Will other systems have to be discontinued? How does the current *customer relationship management* (CRM) or *enterprise resource planning* (ERP) system fit? How can we come up with a robust, reliable, flexible, and adoptable master data management system architecture that allows for scaling with the increasing requirements over time? These and many more questions can be the cause for many sleepless nights.

We are looking beyond the master data management boundaries and into your entire system landscape to respond to these questions. We need to be sure that we understand the purpose of the new master data management system that we are integrating and also its relationships to other systems in the organization before building it. Figure 2-3 demonstrates an exemplary system landscape that helps us in this discussion.

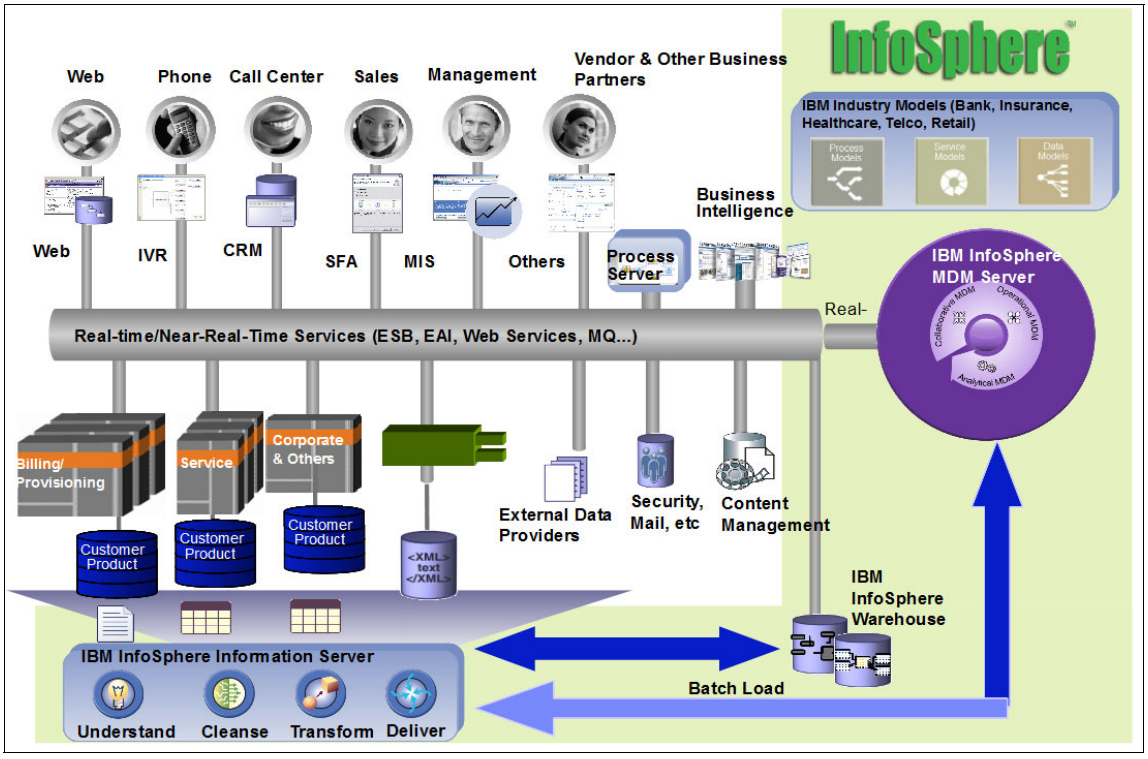

Figure 2-3 Architectural overview and system landscape

The architectural overview in Figure 2-3 shows a number of consumers on top and that are attached to a bus in the center of the diagram. The bottom right of the picture depicts some of the reference systems that we have to include to resolve functionality, such as authorization or even connections to external data providers. Most important to the continuing discussion is the systems that are attached to the bus on the bottom left. These are all managing master data of some kind. Based on that, the task is to extract the master data and related business processes from these sources and add them to the new master data management system. The methodology applied follows the sequence understand, cleanse, transform, and deliver.

It is virtually impossible to address all concerns related to complex system landscapes in the context of this book, but we have to set the scene for the key part of this book, the modeling for a master data management system. We can only do that by looking into the most feasible approaches from a high level and drilling down into a little lower level of architectural design. Keep in mind that we can only consider the most common approaches, and what we describe here is not the ultimate and only solution regarding the placement for a master data management system. Use the following perspectives on which to base your informed decision:

- High-level enterprise architecture
- Building blocks of enterprise architectures
- SOA reference architecture
- Master data management reference architecture
- Master data management implementation styles
- End-to-end integration patterns
- Communication and transformation
- Object models and their locations
- Adaptive service interface
- Conclusions

2.3.1 High-level enterprise architecture

Discussions of the architectural layers in an Information Management System on a high level include the following layers, illustrated in Figure 2-4:

- Data management and content management
- Information integration and management and warehousing
- Business intelligence and performance management

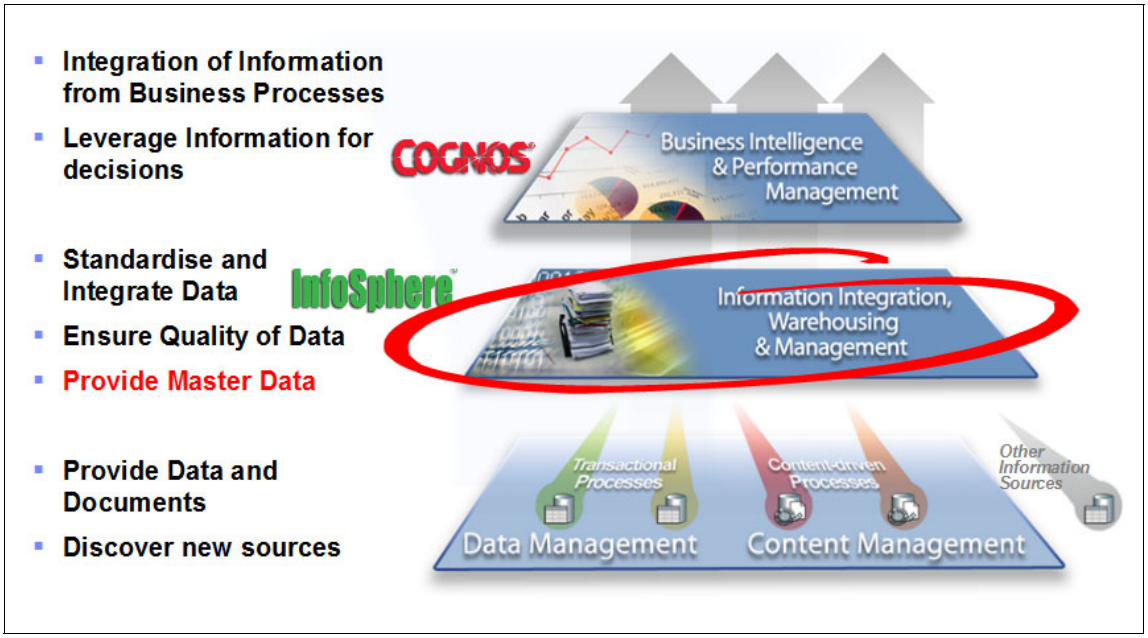

Figure 2-4 Layering in information management

These layers represent functional building blocks on an enterprise level and provide some logical structuring. The layers depend and rely on each other. This layering concept reflects a data-centric or information-centric view of the enterprise architecture.

Within the layers, data management and content management provide the foundation of this information architecture. All data is stored within the database management systems as represented by IBM DB2®. The data to be stored includes meta, transactional, operational, or analytical data. Although we are so far referring mostly to structured data, we need to consider an appropriate system for unstructured data too for which we use the content management system. Content represents documents, images, and other artifacts.

Content management systems (CMS), therefore, include dedicated systems, such as *document management systems* (DMS), and thus represent a higher level in the ontology for the description of content. The kind of content, however, is not exactly relevant in respect to master data management solutions. Important to consider is only the fact that content could be referred to and stored to complement the master data itself.

Information integration and management and warehousing then is the next logical layer on top of the data storage layer. It is dedicated to the integration of data with other systems. This includes data quality and data standardization, as well as metadata management. We also find the data warehouse here. The reason that we find the data warehouse on this layer is that it provides additional functionality on top of the pure data storage.

The master data management system is also on this layer. This layering concept shows that master data management is more than pure storage of common data across the enterprise. It can provide additional functionality such as historization, matching, de-duplicatio,n and much more.

Business intelligence (BI) and performance management then is located at the topmost layer. Business analysis based on bad data or data with a lesser quality will likely not result in the best answers and solutions. Using high-quality data as provided through the master data management system of the middle layer takes away the uncertainty and avoidable faults resulting from bad data. Your predictions and the reports generated from high-quality data are much more accurate, and business decisions based upon them are much more likely to be the right decisions. Whether it is a master data management system or “just” data quality-related components such as for data standardization or data quality, the intermediate middle layer should be used and built up before attempting the roll-out of a business intelligence solution.

2.3.2 Building blocks of enterprise architectures

Coming from a high-level, information-centric layering concept, we are now going to take a different look at an enterprise architecture. The components of an enterprise architecture can be grouped as follows:

- The first group encompasses any kind of infrastructure component, such as enterprise service buses (ESBs), persistence layer, and access services, as well as any other means of communication and integration of services.
- The second group consists of data-related components, where data can mean anything from relational, non-aggregated data to highly enriched information or content.

- The third group is represented by the actual business-related services, and components, such as workflow management services and rules services, are part of the fourth group. User interface (UI) related functionality can be found on the presentation logic layer. Systems management services and development services fall into a group hat we define as supporting infrastructure services. This latter group differs from actual infrastructure base services in that they do not participate directly in business transactions.

This grouping by no means represents a formal classification, but it helps to identify similarities between architectural views. This perspective can be applied to the enterprise architecture, but also product architectures or business line architectures. We use this grouping concept to position some of the building blocks defined.

As other popular layering concepts, this approach is based on the idea that we can distinguish between the following layers:

- Presentation
- Application
- Service
- Data

Before dwelling on these layers in greater detail and attempting to assign building blocks to each, first have a closer look at reference architectures and how they relate to the information-centric view of with master data management.

2.3.3 SOA reference architecture

The service-oriented architecture (SOA) reference architecture describes the relationships of typical SOA components. Because InfoSphere MDM Server claims to be a perfect fit for an SOA, we want to understand better why that is the case. Figure 2-5 describes an SOA logically from a middleware standpoint.

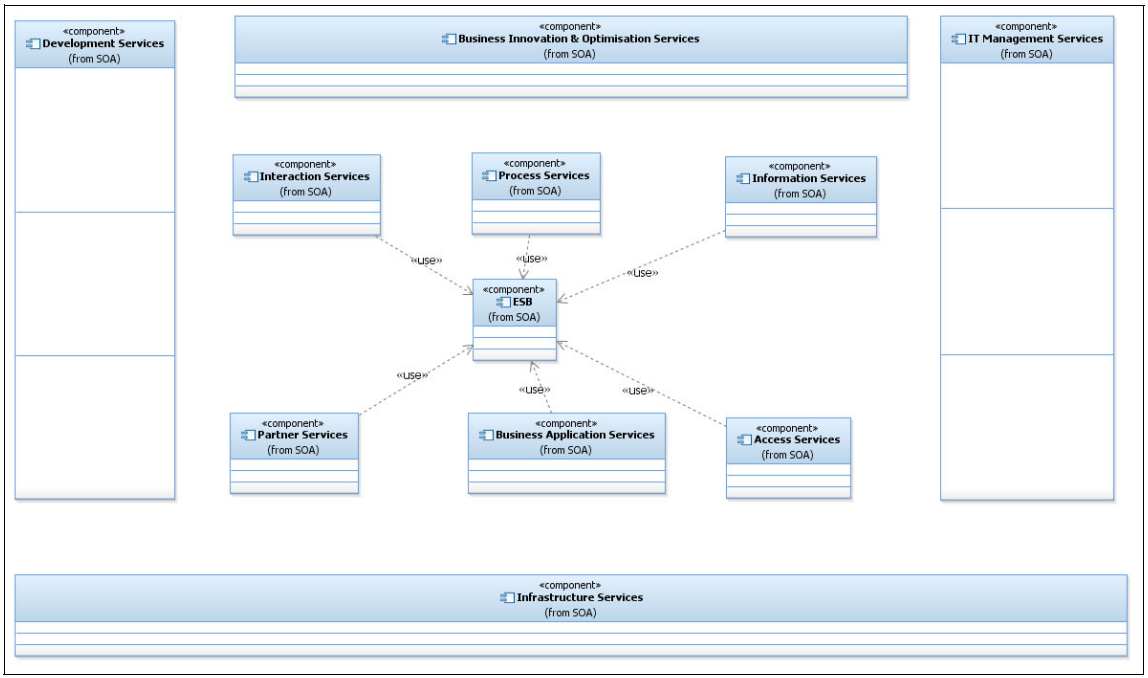

Figure 2-5 SOA reference architecture

It places an ESB as the messaging component in the middle. Whether or not an ESB is used, or another type of integration, plays an important factor when deciding on the technical integration of the master data management system. Here, for now, the six main service components communicate solely through it. Master data management in this architecture is merely a sub-component that belongs to the information services component.

The following components related to the ESB in the SOA reference architecture are available:

- Interaction services
- Process services
- Partner services
- Business application services

- Access services
- Information services

Detailed component descriptions for the SOA reference architecture are not included in this book. However, for further information about that topic, refer to the documents in , “Related publications” on page 559. In the context of this book, the information services component provides a unified view on the following data-related services:

- Analysis services
- Content services
- Data services
- Discovery services
- Information integration services
- Master data management services
- Metadata services

Master data management services correlate directly with the data services exposed through the other sub-components and, in this reference architecture, are exposed through the ESB to other components, such as the process services. We are also going to stress how closely related all these sub-components are. They are all interrelated and support each other. Metadata services, for example, are an essential ingredient when setting up a master data management system. They also support all other sub-components and ensure that a consistent view of metadata is provided to all of them.

Other prime examples are the analysis services. In this case, they do not represent the business intelligence (BI) layer, which is located in the business innovation and optimization services top-level component. Instead, here the analysis services refer to capabilities, such as identifying relationships between entities, for example, households, where all people are grouped together because of a shared address. This also includes checks against blacklists, such as would be done when trying to prevent fraud. In one example, the master data management system could first call upon some data services to get a name or address cleansed and standardized.

We could also take a different approach and store cleansed data only, which would allow us to skip the additional steps. However, we must also keep in mind that users can query the system looking to retrieve certain records. As search criteria they are likely to provide the data similar to the way it was originally entered into the system, possibly a lot different from the standardized form. We thus consider it good practice to store the original data values to maximize retrieval. We can also store both values, non-standardized for retrieval purposes as well as standardized data, which can be fed directly into the analysis services. The results in return can either enrich the master data or cause other actions to be triggered.

2.3.4 Master data management reference architecture

The IBM conceptual master data management reference architecture uses similar components as the SOA reference architecture. The difference is in putting more emphasis on the master data management related functions and services, as Figure 2-6 shows.

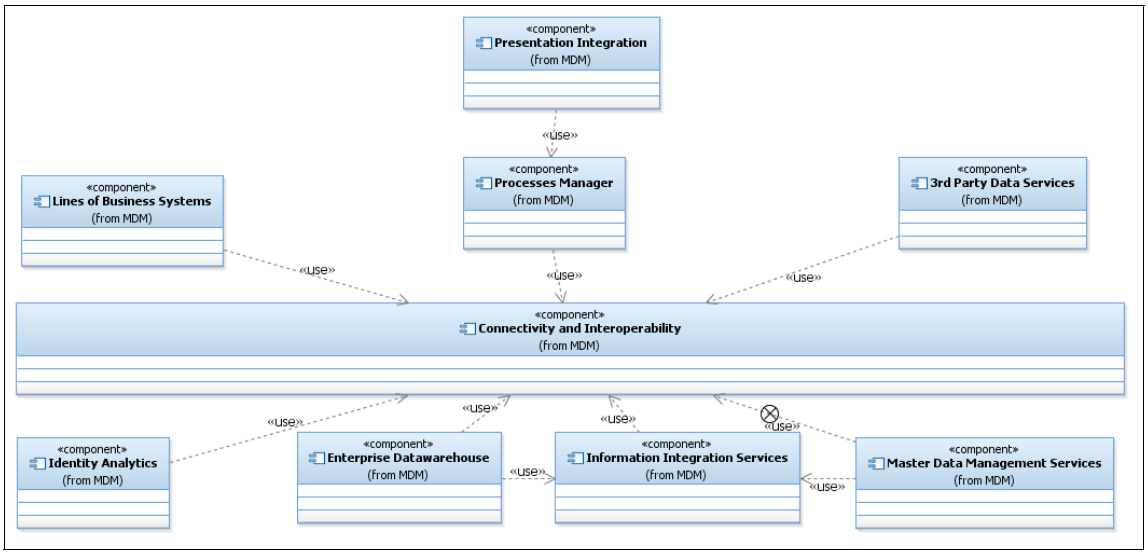

Figure 2-6 Conceptual master data management reference architecture

Because of its relevance to this book, we explore each of these services to a greater extent than we did for the logical SOA components. These core services envisaged in the master data management reference architecture are available:

- Third-party services

  Third-party services represents external providers such as Dun and Bradstreet and ACXIOM. They can be included in the overall architecture to enrich the data about organizations, people, and other master data related aspects, including addresses within their own master data management system. In addition, there are also external databases maintained, for example, for politically exposed persons (PEP) and other whitelist and blacklist type of databases. These are often destined to support your *know your customer* (KYC) efforts.

- Connectivity and interoperability services

  The connectivity and interoperability services provide a means of access against internal and external systems in respect to the enterprise. It is here that we can place batch processing and standardized transport (such as FTP or email) capabilities. It is optional to use the capabilities of this service and not a requirement to gain access to the master data management system.

- Enterprise data warehouse

  The enterprise data warehouse represents the analytical counterpart to the transaction master data. It is here that we are able to analyze the data stored. Of course, we can already analyze the data directly from the database of the master data management system, though good practice provisions for a separation of concerns while also acknowledging the layering concept outlined earlier in this chapter.

- Identity analytics

  Identity analytics allows for a multitude of additional capabilities in respect to a master data management system. One aspect includes the identification and verification of an individual party. These services allow you to obtain a higher degree of certainty that you are indeed working with the person who you think you are.

  Another aspect is related to the resolution of relationships between entities. For one, you can identify a possible obvious or non-obvious relationship of a party with any other party that might or might not have been stored already. If we have stored that other party, we could now enrich the data with the newly identified relationship and thus persist more information than we originally had at our disposal.

  Based on the same principle, you can also identify groups of people or households, as well as social media influencers or opponents, always keeping in mind that you have the data that is necessary to perform such a task.

- Information integration services

  Information integration services provide real-time interaction support with master data management services, and support the initial and incremental load of large volumes of data to target systems such as the enterprise data warehouse and master data management system. They also provide profiling and analysis services to understand data that will be aggregated and merged for loading into a target database and services to standardize and cleanse information.

- Lines of business systems

  The lines of business (LOB) systems provide the core enterprise capabilities using master data. These can be CRM, ERP, and other systems to name but a few.

- Presentation integration

  Presentation integration is also required with master data management systems as with most other systems. This includes a user interface for the business administrator to maintain code tables, product data, and other data elements, as well as a user interface or the data steward, which play a major role in a data governance organization. This service also includes custom user interfaces that are specifically tailored to the customized system, as well as portals into which this might plug.

- Process manager

  The enterprise can use a process manager to choreograph and manage long-running processes that span multiple systems and involve human tasks. Industry-specific process models or custom-developed process models could be used to define the activities and tasks that would be automated by a process manager. The process manager would provide the automation to implement the flow of the tasks and activities for these processes.

2.3.5 Master data management implementation styles

The SOA and master data management reference architectures only provide a conceptual view of the components that play a role in a master data management system. They do not explain how these components are best used in specific scenarios. This is why appropriate patterns are required. Such patterns describe the interaction between specific components in a specific scenario. Different patterns have to be adopted depending on the functional and non-functional requirements imposed on the system. These patterns depend to a large degree on the implementation style adopted.

We define four categories into which any master data management system can be placed. The key is to select the right category for the functional requirements imposed, but first consider the following styles:

- External reference style
- Registry style
- Coexistence style
- Centralized style

The simplest implementation styles are named the external reference and the registry style. Strategically, the implementation of a master data management system starts as a small coherent system that enables the merging of master data from many disparate systems into a single master data repository. The master system only holds references to the data that still resides in the old systems. Accumulating all references to all master data into a single system will enable the identification of master data and their appropriate correlation across existing system boundaries.

To display the aggregated master data, these references will be resolved and the original data is retrieved. Thus, de-duplication is of a virtual nature only and does not eliminate true duplicates within the existing systems. Based on this approach there are already capabilities available that we could not have been provided previously, such as unified compliance reports identifying everything related to an individual. For this reason, the reference and registry style implementations provide a solid basis for further functions aiming towards coexistence and centralized styles.

This leads us to the coexistence style that, instead of simply storing references to the master data, stores the master data itself in its own data model. The responsibility of maintaining the data, however, remains with the existing systems that continue to act as primaries. The coexistence master data hub therefore operates as a secondary to those primaries. The requirement for a synchronization process immediately becomes obvious. Whether we synchronize through batch or real-time processes depends largely on the non-functional requirements that outline data accuracy.

Similar to the reference style implementation, the master data can already be used to create added value, for example, in compliance reporting by providing the ability to de-duplicate person data and aggregate all known information about a single person view. The usage scenario is a read-only one though, because of the old systems still acting as the primaries in the enterprise. Any changes to the coexistence master data will therefore be subject to overwriting with the next synchronization cycle.

In a centralized implementation style, the master data management system that also acts as the primary can eventually be used in read and write operations. Transactional capabilities have now been transferred to the master data management system. We stress that transactional capabilities only refer to the maintenance of the master data itself and not to the recording of any interactions that an individual might have with the enterprise, such as lodging money onto an account.

Based on this aspect, you now have to implement and provide for all processes related to this master data and all systems that depend on it. An important decision to be made in conjunction with the centralized style is whether to keep the existing systems in operation to supply information to their surrounding secondary systems or whether all these surrounding systems can be switched and integrated with the master data management system. In any case, this approach caters to switching the integration for all systems at the same time for successive a replacement.

Considering a step-by-step approach to implementing the styles, we are faced with the reference and registry style as the simplest ones by solely integrating identification data of reference keys and meta information. The coexistence style in comparison adds more complexity to the transformation requirement loading the system because of the increased amount of entities to be populated. The complexity of the centralized style is much higher because of the requirement that the processes maintain the master data, which is not required in pure referential (secondary) scenarios.

Another large part that contributes to complexity is the possible requirement for synchronization with any existing system that cannot switch off transitioning to the new master data management system. Following a step-by-step approach can minimize the risk involved in embarking on a large-scale centralized style type of project right from the beginning. However, it is vital to always bear the strategy set in mind when developing a road map that follows a step-by-step approach, as discussed based on the implementation styles.

2.3.6 End-to-end integration patterns

Applying the reference architecture and the implementation styles leads to the definition of integration patterns. Here we restrict ourselves to the description of the most common patterns only. Also, the patterns presented are still rather coarse grained. The reason for this is that in the end-to-end view we apply these patterns. As enterprise architects, break these patterns down into smaller granular pieces. With the smaller patterns, you can assemble the different possible end-to-end scenarios.

This section provides insights into building your own repository of patterns and focuses on four typical end-to-end patterns.

One-off initial and delta batch load and read-only downstream access

This most simple integration pattern can be used to replace an existing master data system. The extract, transform, and load (ETL) process is simple because it is required only for a single initial load and can be supported by an accompanying delta load to populate the InfoSphere MDM Server database with master data originating from the existing source systems. After this load process has been completed, the existing primary systems will be switched off and the InfoSphere MDM Server replaces the previous primaries as the new primary system.

Only if the data maintenance processes can be kept simple will master data be additionally maintained by data stewards through the default data stewardship user interface provided by InfoSphere MDM Server. However, the data is maintained mostly through online channels connecting to master data management.

Downstream systems only access read-only business services provided by InfoSphere MDM Server through either web services or *remote messaging interface* (RMI). The pattern is shown in Figure 2-7.

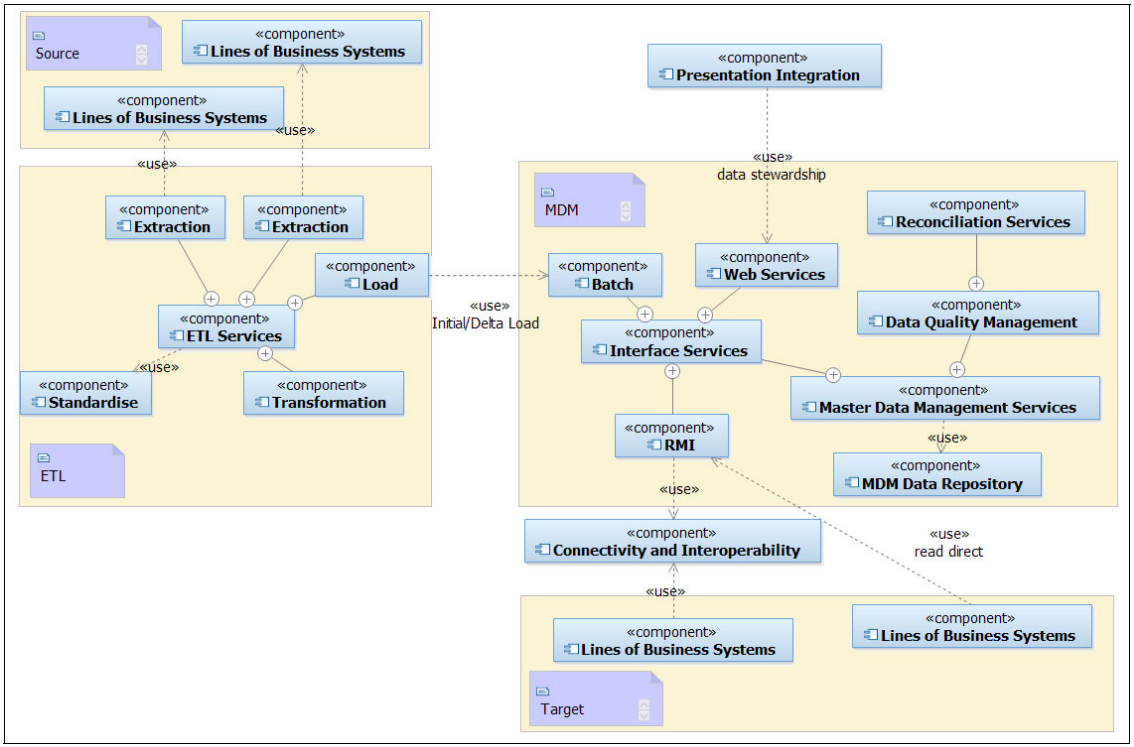

Figure 2-7 One-off initial or delta batch load and read-only downstream access

The general idea is to keep the integration pattern simple and efficient, which is why we recommend integration through RMI in most scenarios. In addition to the default party information, there can be additional benefits through enhancement and maintenance of appropriate hierarchical or group relationships. Data quality of the master data can be maintained through default data deduplication processes.

Furthermore, this scenario can support reference business keys to contracts or accounts that a party might have with the enterprise. This enables a true 360-degree view of a person across all the persons accounts.

Batch synchronization and read-only downstream access

This second most simple integration pattern can be used to consolidate the existing master data from the existing systems that will remain acting primaries. The ETL process is a little more complex and requires additional services due to the requirement of incremental loads for all subsequent batches. So, instead of a single batch load we are now faced with frequent batch loads to keep the master data current based on the actual data contained in the existing systems that remain primary. Typically in this scenario, the incremental batch loads are executed once daily, weekly, or in any other short-term frequency. This pattern is shown in Figure 2-8.

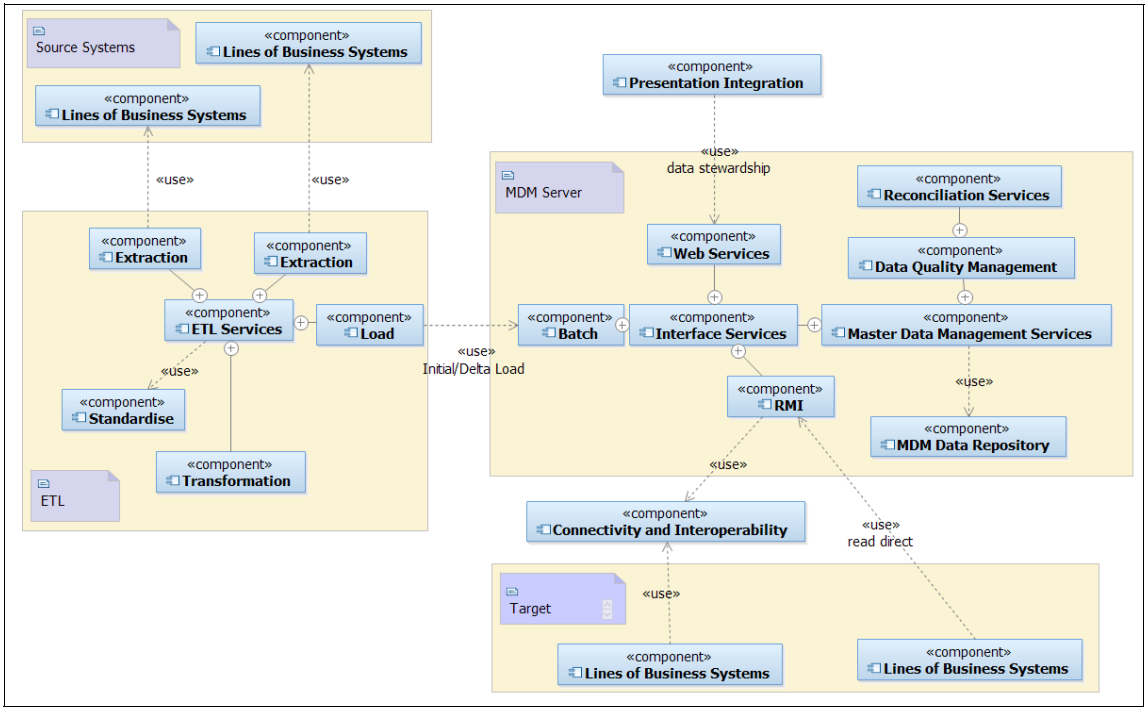

Figure 2-8 Batch synchronization and read-only downstream access

Continuous synchronization and read-only downstream access

You can use this integration pattern to consolidate master data from existing systems that will remain acting primaries. The major difference to the batch integration pattern lies in the real-time synchronization between the existing source and the master data management systems. Real-time synchronization can invoke the corresponding master data management services directly or using a messaging infrastructure. It is important to keep the right sequence of messages to maintain consistency in the data in case an an error occurs when an update follows the corresponding add. Thus, the requirement is to prevent parallel execution and to keep the correct sequence of events for all services that can manipulate the same instance of any data element.

The ETL process is a little more complex and requires additional services for incremental loads in all subsequent batches. So, instead of a single batch load we are now faced with frequent batch loads to keep the master data current based on the actual data contained in the existing systems that remain primary. Typically, in this scenario, the incremental batch loads are executed once daily, weekly, or in any other short-term frequency. This pattern is shown in Figure 2-9.

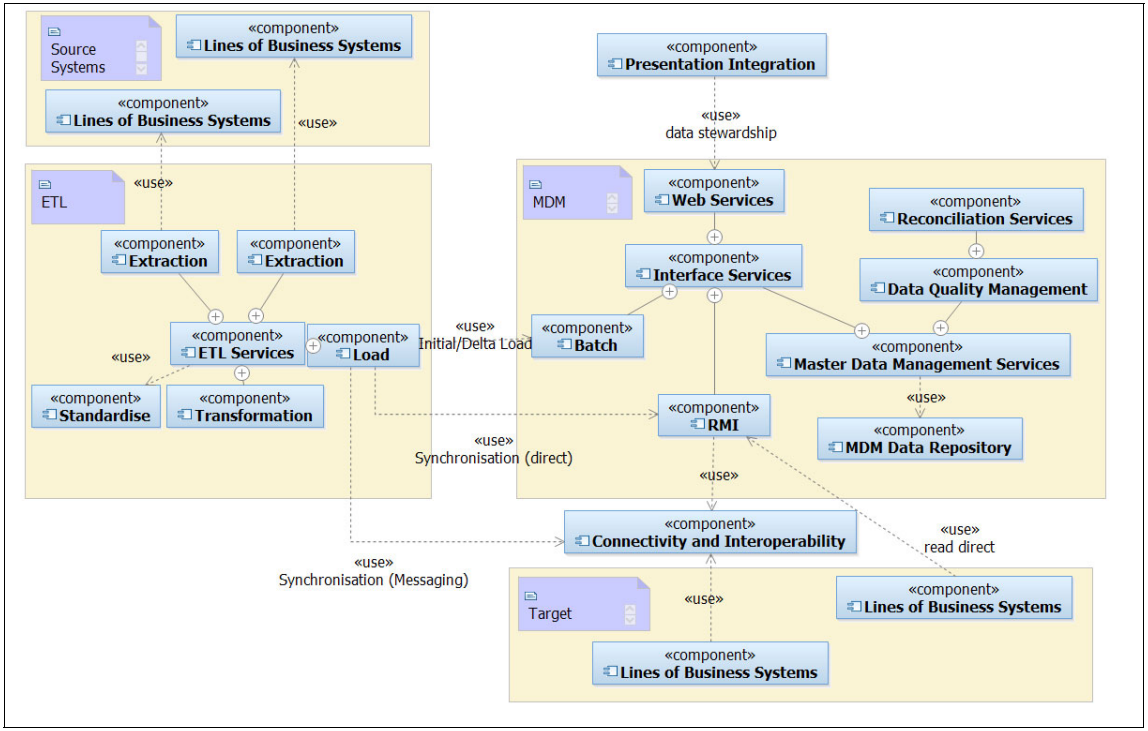

Figure 2-9 Continuous synchronization and read-only downstream access

Continuous synchronization with ESB for up and downstream

One key component that also happens to be at the center of both, the SOA as well as the master data management reference architecture, is the ESB. Whether we want to use it depends on a strategic decision having distributed components communicate using a central bus. Sometimes point-to-point interaction is the preferred way of integration. We want to discourage you from the latter due to maintenance of a variety of interfaces as opposed to the possibility of using a single normalized protocol towards the ESB.

The ESB provides mediation services to other interface structures. Standardization or identity analytics do not necessarily have to be invoked through the ESB. Instead, we prefer attaching these components directly to the master data management system to optimize the interaction between both components.

2.3.7 Communication and transformation

The discussion about patterns has already brought up the communication topic. You might have noticed disclaimers and a few attempts to define exceptions in the pattern discussion. Because we consider the communication aspect important, we want to expand on our views.

Using an ESB makes a lot of sense in many scenarios, and we already outlined the requirement for transformation. These transformations are required between systems that operate on different data models. Implementing the object structure from one system in another one to be able to directly integrate with it is an approach known in Enterprise Application Integration (EAI) projects.

Understand that this method breaks a fundamental rule of SOA, loose coupling. For an SOA, you need to avoid hard-wiring logic from different systems. As a result, the previously mentioned transformation of data models between systems is required, which is also depicted in a slightly different architectural view, as shown in Figure 2-10.

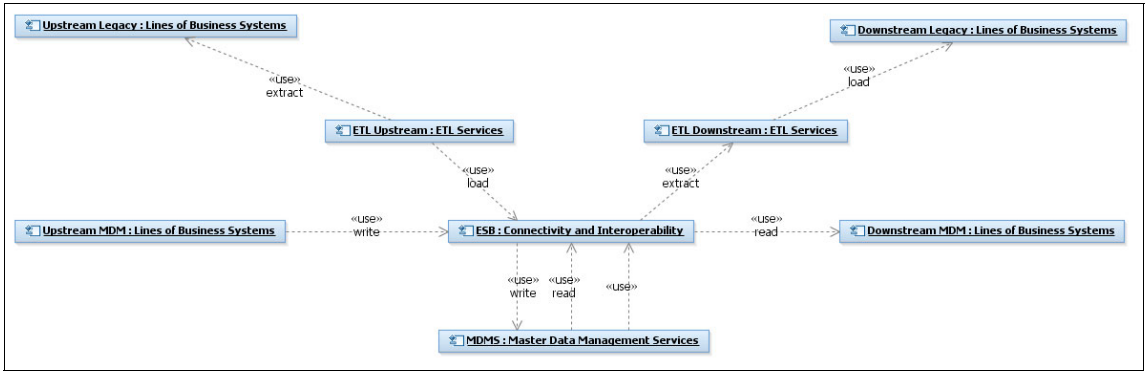

Figure 2-10 Continuous integration with ESB for up and downstream

Because this is only one possibility of many, we are showing the same end-to-end integration aspect again, but this time without the ESB in the middle in Figure 2-11.

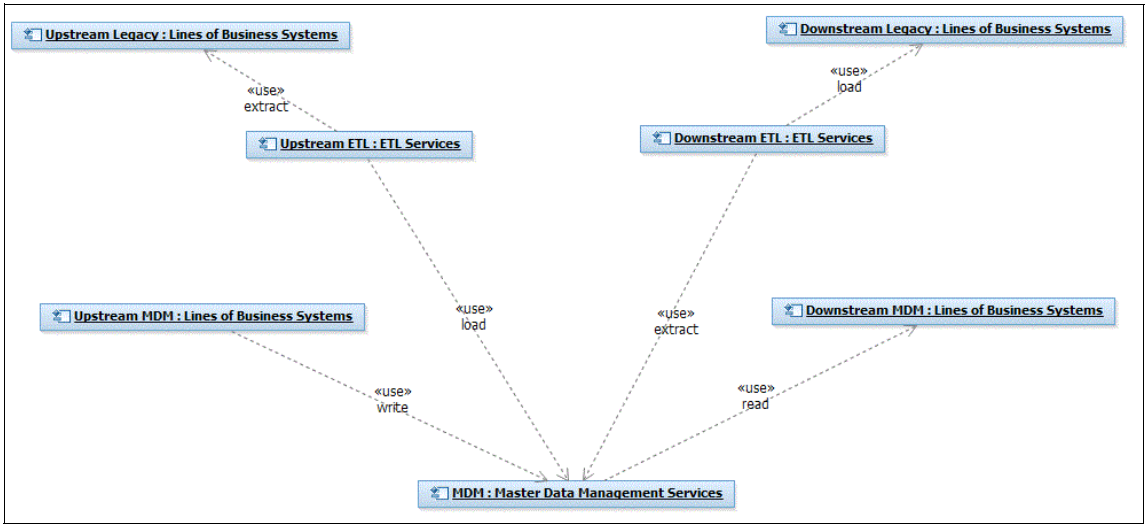

Figure 2-11 Continuous integration without ESB for up and downstream

Based on these diagrams, you can look closer at the transformation requirement and the fitting of an ESB. Considering that you have to integrate a number of different existing systems and that each of these systems provides its own data model, the ESB can also be the enabler between such object structures and can carry out the required transformation.

To help you determine whether to use an ESB, keep in mind the following key points:

- The amount of business logic required
- The need to perform normalization and denormalization between distinct data models
- Non-functional requirements that demand high throughput
- The requirement to carry out initial and delta load

We suggest transformations only through the ESB in systems that do not have to carry out transformations with a lot of business logic involved. Neither do we suggest using an ESB when the required transformation includes a large degree of normalization or denormalization. The amount of processing and the likelihood of accumulation of additional data is too high. In particular, the aspect of normalization or denormalization is often not given the consideration that it deserves.

Often in these cases we have to gather much more data to populate a denormalized from a normalized model, such as the data model provided by InfoSphere MDM Server. Instead, a dedicated ETL tool can perform a much better job. It deals with the processing much more efficiently than an ESB would because the ETL tool is purpose built for such tasks in comparison to the transformation capabilities of an ESB that comes with pure necessity when having to integrate with numerous different types of interfaces.

The increased performance also makes the ETL component suitable for transformations of large volumes of data as happens during batch processing, for example. We cannot recommend piping a large volume of data coming from batch processes through an ESB. In such cases, consider bypassing the transport mechanism of the ESB and using an ETL tools extract and load capabilities instead.

Another advantage of ETL tools such as IBM InfoSphere DataStage®, which is integrated in the IBM InfoSphere Information Server, is their ability to carry the major task in initial and delta load scenarios. Regardless of the implementation style or system retirement and replacement option selected, such capabilities are always required to load all relevant master data content into the new master data management system. The ETL tool helps to extract the data from the existing systems, transform them from that data model into the data model required by the target system (in this case the master data management system), and either directly load the master data management database or invoke the services exposed by the master data management system. Using an ESB during mass processing of an initial or delta load causes too much overhead and should be avoided.

After an ETL tool is in place due to the aforementioned necessities of high throughput and initial load requirements, consider using it for the in-synchronization tasks as well. In such scenarios, the ETL and ESB would coexist with the ESB, predominantly routing the messages from the ETL tool to the master data management system while the ETL tool is responsible for the transformation of the data models.

2.3.8 Object models and their locations

Up to this point we have only dealt with the communication with the master data management system. The communication, or rather the object structures that we are transporting, we are going to investigate next. This includes the question of how the data model can look and where these objects and their respective models reside. The existing systems often bring along a set of models, each of them somewhat different. InfoSphere MDM Server provides its own model. For many integration scenarios this completely satisfies the requirements. The ESB mediates and translates from any existing model to the InfoSphere MDM Server model.

Of course, we cannot, or in most cases do not want to, change the models for any of these systems. However, we do have an influence on and a choice for the location of the object model that we are going to use with the new master data management system that we are going to build. We propose the definition of a so-called business domain model in large and complex master data management integration scenarios. We expect many benefits in respect to its introduction that we will explain in the next part of this book. Its placement within the architectural context is critical. The following paragraphs are dedicated to outlining the options that we have for placing it with the corresponding advantages and disadvantages that each placement provides.

First, we look at the general application architecture. Picking up on the definitions in the discussion about the building blocks, the general application pattern follows this four-tier layering model comprising the following layers:

- Presentation
- Application
- Services
- Data

At the bottom there is the *data integration layer*. It is here where the data is persisted and where we find the lowest level access to the data in its physical and logical form. This means that we do not simply allocate the physical database tables here, but also the data objects that encapsulate them.

The middle layer represents the *services layer*. This includes the business services as well as the services through which we execute the business logic and the transactions that coordinate these services. Services on this layer are usually context-free and independent and serve to orchestrate the behavior of several domain objects.

The *application layer* choreographs services and adds logic specific to the application. It contains activity or task-like services that can be re-used and treated as any other SOA-compliant service. Sometimes these services are too specific to be reusable. Keep in mind that the main difference between the application and the services layer is the level of granularity and the level of ubiquity. This discussion about the services model is equally applicable to the application layer when it comes to application-specific logic.

Actually, certain applications might not expose any kind of application-independent service. In those cases, you need not include a services layer in the models. The topmost *presentation layer* is dedicated to the user and includes the UI that is required to interact with the user and that invokes the services on the application layer.

In respect to an SOA in general, Figure 2-12 places the components of an SOA into their appropriate layers.

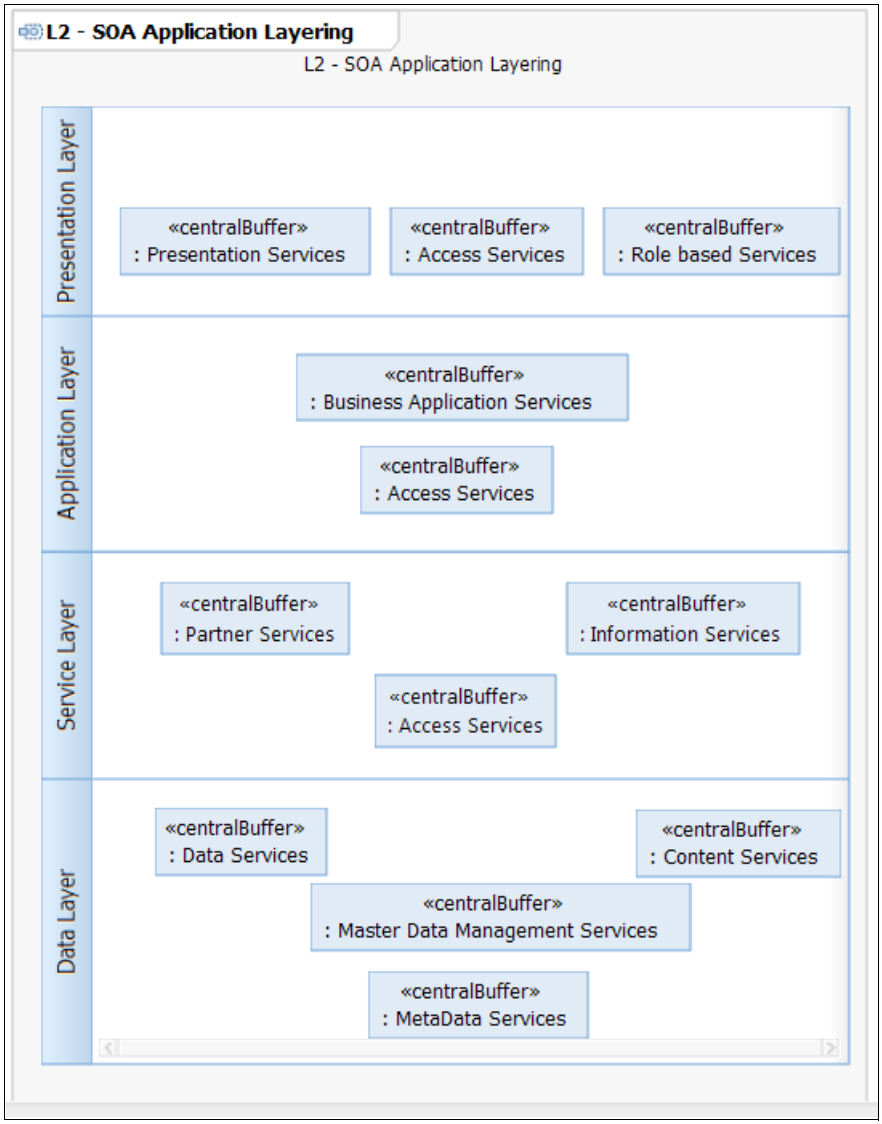

Figure 2-12 SOA application layering

The presentation, access, and role-based services all fall within the presentation layer.

Business applications and their corresponding access services belong to the application layer, and services independent of specific applications reside in the service layer. These services include generic information services and partner services.

All data-related services, including the data that they represent, are part of the data layer. Placing the individual master data management related components into the general application layers, we achieve the layering shown in Figure 2-13.

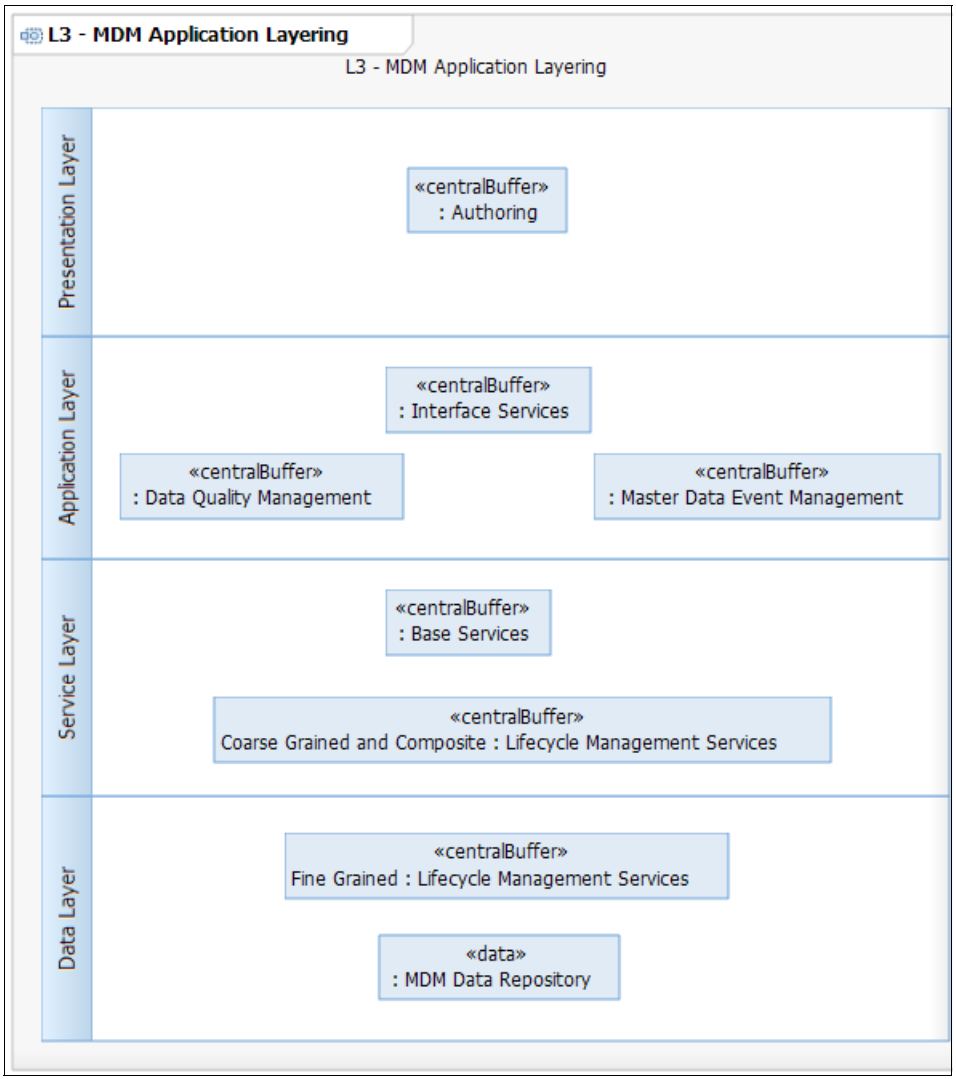

Figure 2-13 Master data management application layering

The authoring services, including hierarchy management and everything represented by a user interface, such as business administration, product administration, and data stewardship, is placed in the presentation layer.

All logic-related components, including the services that expose the business logic, are placed in the application layer and the services layer, respectively. On the service layer these are the base services and the custom composite services. Use of these services takes place on the application layer where custom domain-specific tasks such as quality management and de-duplication take place.

The master data management data repository and the fine-grained services and data objects make up the data layer. There is one argument that could see the fine-grained services also in the services layer, because they are directly accessible by the application or any other application layer in an EAI. Also, these services contain a considerable amount of business logic. However, because this component also provides the business objects that represent the data objects of the repository, we agree with placing that component here.

Now that you understand the service component placement in respect to a standardized SOA reference model and the InfoSphere MDM Server model, you can embed the aforementioned business domain model. Figure 2-14 shows the business domain model architecture in respect to a master data management system in general.

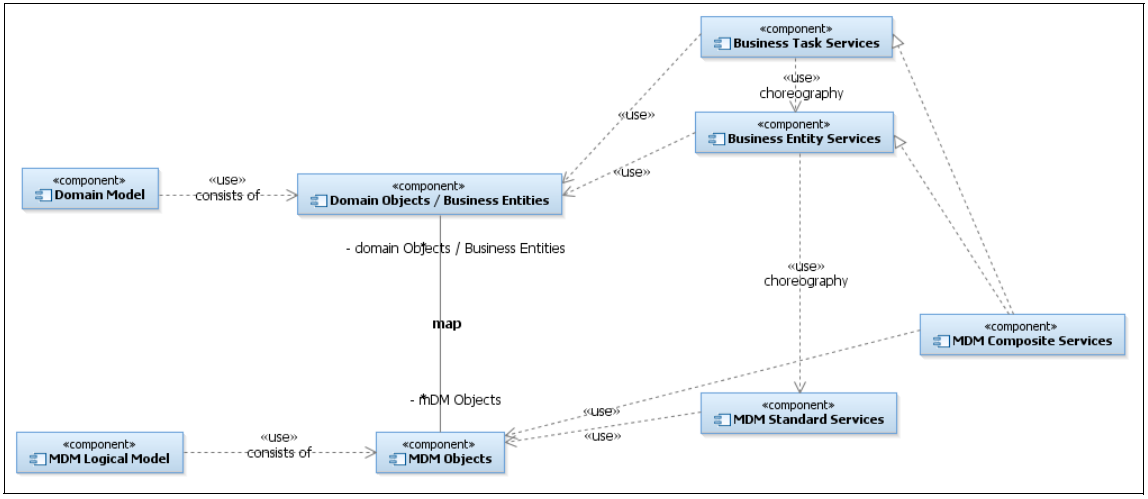

Figure 2-14 The domain model and the composition of custom domain services

The domain model, as any other object model, is based on its domain objects. These objects with their corresponding attributes are mapped from or to master data management objects. This mapping requires transformation, but more importantly it represents an abstraction layer. Instead of exposing a master data management specific language to the layers communicating with InfoSphere MDM Server, we are hiding this logic and instead expose a custom domain model. Communication should be much easier for other systems that we need to connect to the master data management system. Such an approach is required when a standardized domain model is already established in the enterprise. We also recommend introducing one when there are too many systems involved where no one can agree on which data model to adopt and the InfoSphere MDM Server provided data model is just seen as one of the many arbitrary models embedded in the many surrounding systems.

After we have decided on the introduction of a domain model, we are faced with several tasks. First, we have to map between data models. Based on that mapping we are required to develop a transformation that can be executed during run time. Third, we need to define a service choreography, mapping from business task or business entity services to the standard master data management services provided by InfoSphere MDM Server. Both business task and business entity services can be built as master data management custom composite services.

Now that we have discussed the implications of a domain model, let us see where we insert it into the layering concept. Figure 2-15 describes the application layering concept in respect to applications that are based on business domain models.

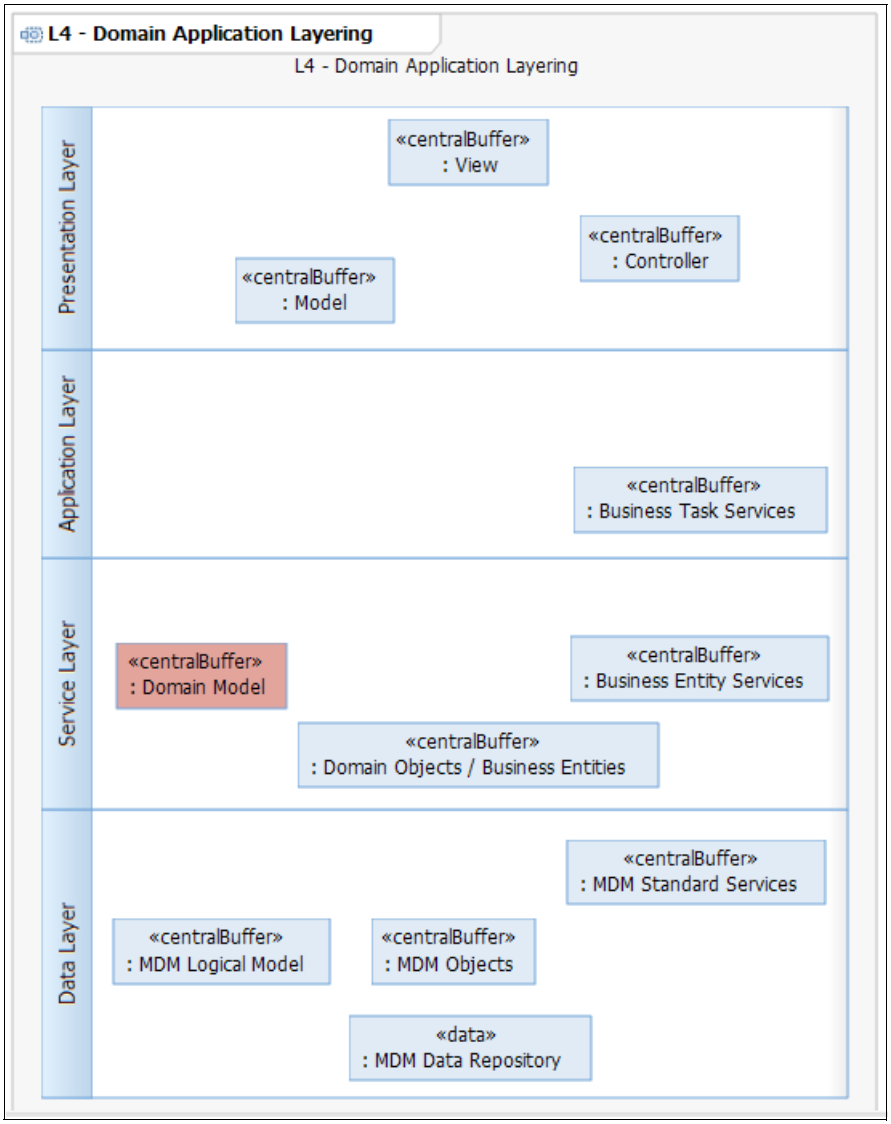

Figure 2-15 Domain application layering

The application layer shows a typical model, view, and controller pattern. Although any other pattern is equally possible, remember that the business domain model is now in the services and domain layer. This pattern is predestined to be used to reflect on the model also in conjunction with the presentation services. Note that a model can be transformed to some view-specific transformation before actually being displayed in the UI.

The business domain model, as the core model of the application, resides in the services layer. As mentioned previously, a business model can also partly or completely pertain to the application model. We discuss this later in greater detail. Right now you might be wondering why the domain model resides inside a “procedural” layer. It does so for various reasons, but predominantly because it contains a huge chunk of the business logic required not just in the model itself, but more importantly also in the lifecycle and rules that accompany the model. The model can get accessed through a set of services dedicated to maintaining the model, the business entity services. They are *create, read, update, and delete* services, and they allow fine-grained access to the business domain model. The model should be embedded such that all business logic is directly developed in the language of the business domain model. Hence, the remaining pieces of business logic are ingrained in the business task services that also represent in more or less parts the use case flow logic.

Keeping the same argument as previously made in relation to the master data management services layering, the master data management standard services remain in the data layer. The master data management logical model and all its objects are in this layer. Finally, those objects are consumed by the standard services through which master data management applications gain access to the master data management data.

However, all the previous layering concepts were reasonably simple, because they focused only on a single application. In the system setup though, we need to consider the existence of multiple systems on both sides, up and downstream, that either feed into or extract data from the master data management system. We are, therefore, selecting a simple system setup that represents an end-to-end data flow, as can be seen in Figure 2-16. Based on that we discuss the placement of the domain model and where we see the required transformation from and to it happening.

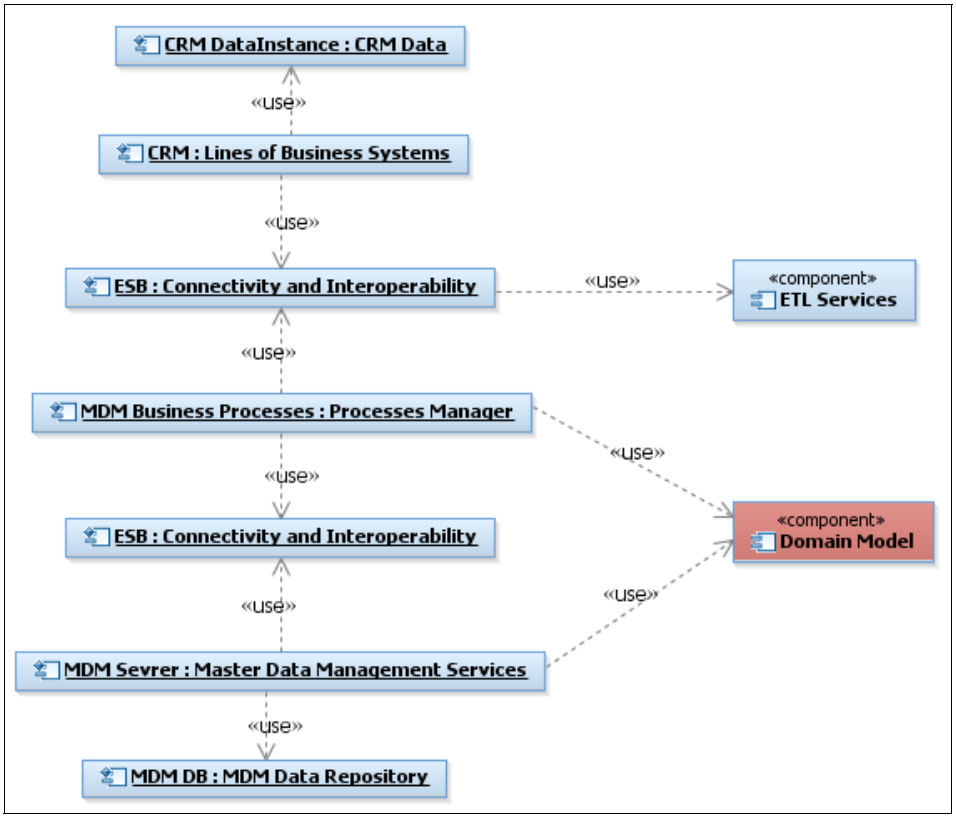

Figure 2-16 L5-master data management end-to-end layering

The general approach that we outlined for the placement of the business domain model still applies. It resides on top of the master data management services. But where exactly? There are now many other components using it, including the ETL tool typically required to interact with existing upstream or downstream systems.

Let us work through the logical choices. We could place the business domain model either in the ESB in the sense of a normalized model used for transformations (note that the ESB is in the picture multiple times due to the multiple communication steps) or down toward the master data management services. When working with business domain models we usually expect a significant deviation from any other data model currently in place in the system landscape. It is because of that and the fact that the ESB should only take care of simple mediations that we advise against the placement of the business domain model inside the ESB. Instead we suggest using ETL services to transform the data model of the upstream or downstream system to the business domain model. Using ETL versus ESB mediation depends on other factors too, as we have already described while discussing the transport mechanisms.

Another key factor for placing the ETL and business domain model is to see where the process implementation is done. All business logic is ideally written in the language of the business domain model. Taking it to an extreme, we would even create a *domain-specific language* based on the business domain model, but that is out of scope of this book. Still, the business logic is described in terms of the business domain model to ensure consistency between business and implementation models. In this particular example, the business logic is implemented through master data management processes that are external to InfoSphere MDM Server. It is attached to the ESB. Because the ESB is coordinating messages from many systems, we need to ensure that it is transforming the master data management system's own data model into the business domain model that is conformed with ours prior to feeding the data into the business process that is orchestrating the business task services exposed by the system. We invoke ETL services from the ESB to achieve this transformation.

The business domain model itself is therefore placed either inside or directly on top of the master data management services. This is necessary because there are typically multiple access paths into InfoSphere MDM Server. Some use ETL, whereas others do not, perhaps because they are already based on exchanging the business domain objects. The principle of gaining access to the master data through means of business task services remains. In large and complex models we also suggest that these business task services carry out a choreography against another set of business domain related services, the business entity services, which are directly related to the entities that make up the business domain model. In smaller models, the business task services can also carry out a choreography directly against the InfoSphere MDM Server standard services.

2.3.9 Adaptive service interface

In respect to the transformation that is required between a custom domain model and the InfoSphere MDM Server built-in model, InfoSphere MDM Server V10 introduces a feature called the *Adaptive Service Interface* (ASI), which provides the following support:

- Support for Enterprise Information Model language

  A master data management solution needs to adapt InfoSphere MDM Server messages according to Enterprise Information Model defined by the client.

- Support for industry-standard message format

  A master data management solution needs to adapt InfoSphere MDM Server messages according to industry standards, such as ACCORD, NIEM, HL7, and so forth.

- Web service tailoring

  WSDLs shipped with InfoSphere MDM Server in some cases might be more than are required for a specific client scenario.

In the context of what we are addressing with the complex integration scenarios that are typically also based on a custom domain model, we can use this feature such that it provides the required domain abstraction for us. However, we believe that it should predominantly be used for the tailoring of the web services provided by InfoSphere MDM Server, as well as simple communication protocol normalization.

You should reduce its use to the scenarios, where you do not require additional business logic such as consistency rules, because an implementation using the ASI would require distribution of domain abstraction from additional logic. In case this additional logic must participate in the context of the transaction, we would be required to implement the transaction handling in the business logic layer, as opposed to using InfoSphere MDM Server. Figure 2-17 shows the general setup.

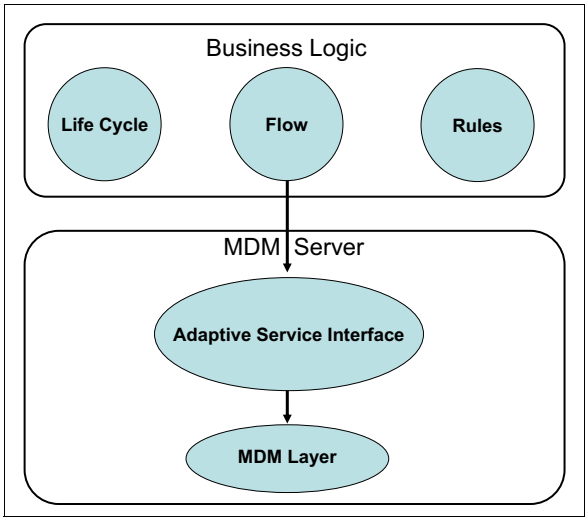

Figure 2-17 Implementation of business logic using the InfoSphere MDM Server ASI

Initially, the domain model is captured in InfoSphere MDM Server Workbench, allowing this model to be available for further use. Now determine and define the mapping between the domain model and the InfoSphere MDM Server model that is predefined and available. Based on this mapping, the ASI performs the translation between these models during run time. As represented in Figure 2-17, the ASI still resides within master data management and exposes the custom domain model exactly as needed. However, because it is not intended that this interface hosts much additional business logic, you need to provision a business logic layer that holds the logic that ensures consistency within the context of the domain model through rules and lifecycle in addition to the usual business logic execution flow.

For these reasons, build your domain abstraction and couple it with the rules and lifecycle that go with it in case of custom domain implementations that go beyond simple protocol normalization. Figure 2-18 depicts the implementation of business logic using a domain abstraction layer within InfoSphere MDM Server.

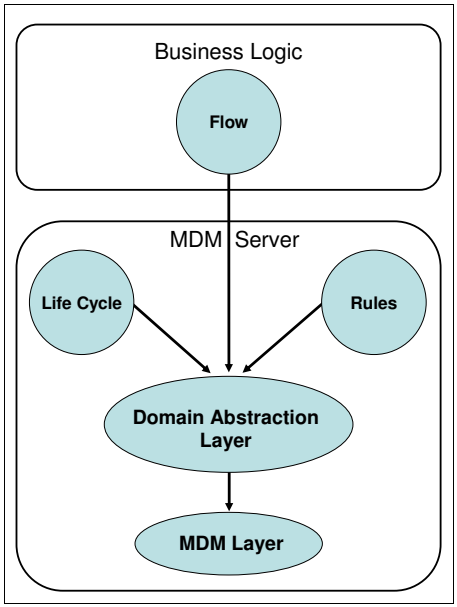

Figure 2-18 Implementation of business logic using a domain abstraction

To achieve this architecture, follow the modeling steps described in the remainder of this book. The result will be a domain abstraction layer embedded in InfoSphere MDM Server much as that provided through the ASI. However, in this case we are also coupling the rules and lifecycle management with it, which we believe is advisable due to transaction management. We now have all logic that needs to be executed embedded in InfoSphere MDM Server and can thus manage the transaction context from within, and therefore also use further features provided by the product, such as sending out notifications to downstream systems.

2.3.10 Conclusions

In a real-world environment there might be project or other business factors to consider besides an ideal architecture. The following guidelines help to find a good approach. Note that we assume the master data management system of choice to be InfoSphere MDM Server.

Start by looking at the models provided by the product:

- The InfoSphere MDM Server data model is generic, as opposed to a specific domain model made to fit a specific enterprise.
- The InfoSphere MDM Server logical data model is a 1:1 representation of the generic data model.

Now look at your business:

- Your business has been speaking its own “language” for a long time already, and adapting to a language introduced by an external product represents a lengthy process.
- Your business rules are likely described in business terminology. To apply them to a different data model they would have to be rewritten to match the host model.
- You have a lot of IT infrastructure that already brings along a set of logical models.

You can now weigh the options. If most of your enterprise IT systems use independent models that require mappings and transformations anyway, the master data management project can be a good opportunity to introduce a new domain model as a corporate standard and gradually transition to mapping against this model only. In such an environment you would gradually build the equivalent of the enterprise conceptual model, but with much greater detail. Modeling every data aspect for all systems in a single model is not feasible. Consequently, you produce one model for each system introduced into the enterprise. With each extension to the domain model you need to ensure consistency across the boundaries to the extensions of the domain model.

On the other far end, if some of your systems already use a model that fits the master data management system, the new product should fit in seamlessly without any necessity to define models. Finally, if a domain model is already established in your organization, it is feasible to establish a mapping to the master data management product only.

With so many systems and corresponding models in place, there is a need for frequent model transformations. Establishing a central domain model brings along positive aspects, in particular when it comes to designing and implementing the new master data management system. Provided that you use InfoSphere MDM Server and that you do not add an intermediary data model, you will always face the challenge of mapping the data model from each surrounding system to the model that is provided by InfoSphere MDM Server to achieve interoperability.

It is here where we can use the proposed approach of establishing an abstraction layer in the form of the domain model provided by the InfoSphere MDM Server component. Both designers and developers can continue to work within their

respective realms. Designers continue to work using business language only, and the developers focus on the core InfoSphere MDM Server data model. Only a few people are dedicated to mapping between both worlds. The ability to focus on the business language alone will improve the quality of the system and prevent mistakes that arise from not knowing InfoSphere MDM Server well enough or misinterpreting the mapping. At the same time specialists with in-depth product knowledge can fine-tune the implementation without having to worry about losing business functionality hidden in the data mappings or service designs.

2.4 The context of master data management components

The architecture, design, and implementation of a master data management solution strongly depend on the context in which the system operates. In addition, you need to consider organizational functions that also impact the system design. Incorporating information governance into the master data management strategy requires further additional aspects to be considered. This chapter outlines the most important components and functions to consider in master data management solutions.

2.4.1 Functions

We have already outlined how a master data management system is related to the business processes manipulating the master data. These processes often involve people within the organization fulfilling a certain role. Such a role could involve approval or four-eye (two individuals) principles. They also include the data steward's responsibility regarding maintenance of high quality of the master data. On the technical level, this includes databases and the master data management system that manages the data consistently across all business processes that manipulate or use master data.

Such a high-level perspective is much too simple. There are many more aspects to be considered. So, before discussing the components, we first look at the following supporting functions:

- Master data management strategy

  Your company's overall strategy drives the business and defines its business goals. You need to ensure that there is a part of it dedicated to the master data management strategy. It needs to define how master data management fits into the organization, what its goals are, and why we are aiming to implement a master data management system. Because any strategy requires the people or parts in the organization supporting it, you need to include the stakeholders and business units sponsoring it. It is vital, and we dedicate more space to it in the next chapter where we outline what makes a successful strategy and how to turn that into successful projects.

- Data governance

  Data governance is an important ingredient in a successful master data management solution. The likelihood for a successful master data management implementation increases with the existence of data or information governance. This is largely due to the fact of shifting ownership. Where previously the organization that owned the data silo also owned its data, there is now no clearly defined ownership anymore because everyone owns the data in equal shares. To avoid issues resulting from that shared ownership, establish a central function dedicated to the management of the master data. Such a function is supported by processes established and the technology supporting them.

- Data quality

  Data quality is another important aspect of master data. It is affected by your data governance initiative. An increase of data quality should more or less automatically be the outcome from the successful establishment of data governance. Data quality can be increased at three different points in time related to a master data management implementation (prior, during, or after loading the master data management system). Putting data quality processes in place early benefits everyone using the data later. It is not always possible to dedicate the extra effort up front due to tight deadlines for the master data management solution. However, doing so pays off and should be considered.

- Architecture and tools

  Architecture and tools cover the technological functions. This is mainly about how the master data management system is integrated into the larger corporate architecture, as we have already discussed. It addresses concerns such as where to place the master data management system, how to operate it, and what its interfaces are. It also covers the methodology with which to drive the solution to a success and the tool selection. As you will see in later parts and chapters of this book, we suggest a model-driven development (MDD) approach. This decision rules out a number of tools and narrows down tool selection to the ones capable of building consistent models.

2.4.2 Components

So far we have discussed the organizational context of a master data management system. Another area of interest is the functional context. Master data management comprises many different aspects and therefore is difficult to cover by a single product. We believe that will remain the case for some time to come. Figure 2-19 depicts the functional components and also shows how the organizational functions discussed earlier relate to these.

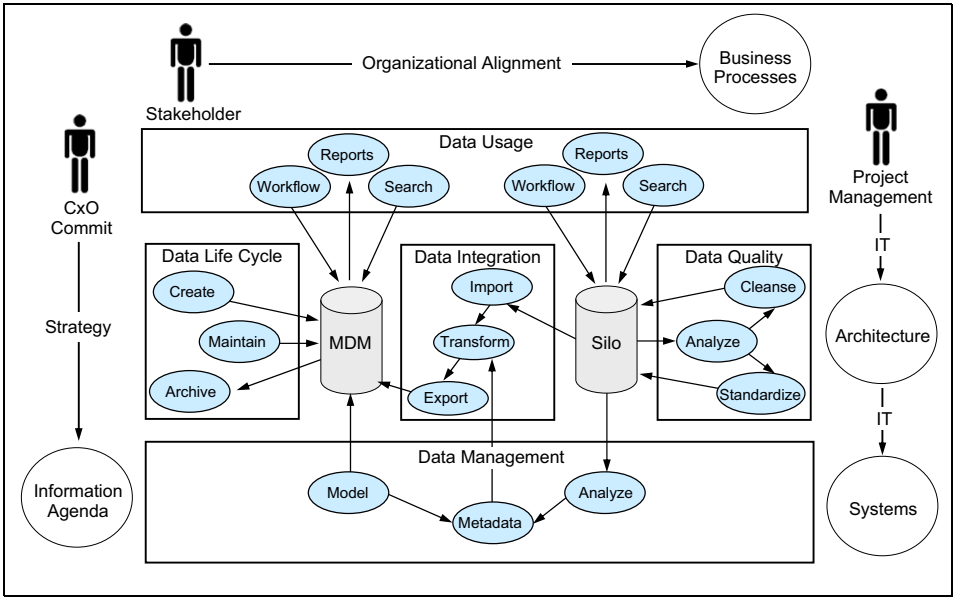

Figure 2-19 The context of master data management components

Figure 2-19 on page 67 describes the different data sources, existing silos, and master data management in its center. These are managed by the surrounding components that support distinct processes. On the outskirts of this diagram we show bringing in the supporting functions.

Multiple, if not all, components involved in the system architecture are concerned by the functions. Strategy impacts and defines the overall system architecture and components selected. The organizational alignment and its processes impact data quality, data usage, and data lifecycle. Architecture and tools define the best possible solution and ways the system integration will take place.

Despite this diagram only outlining high-level components, there is a lot to be read out of it in relation to the possible setup of a master data management system. The key to understanding the concept of a master data management solution lies in a good understanding of these components and how their functional aspects can be applied to support the master data management strategy defined. The definition of these components gets rather tricky for two reasons:

- Point in time when a component is used
- Interdependencies between components

Each of the components can be called upon at different points in time. This in turn causes different results and side effects. The data quality component, for example, can be called upon prior to, during, or after loading the master data management system. Each point in time has a different impact on the system.

An even more difficult aspect resides in the interdependencies that these components have with each other. One example for this is the possibility of the data integration component directly invoking data quality services to standardize the data to be moved into the master data management system. Another example is the creation of the necessary metadata prior to the modeling and creation of the master data management system, while still drawing upon this data during run time to drive aspects with respect to the data usage. The interdependencies come for one from the different points in time that a component can be invoked but also are due to a master data management system carrying out many functions to provide the service expected of it.

We describe the individual components represented in the diagram in the sequence that represents the preferred definition of a master data management solution. Because this ideal sequence cannot always be applied we also describe what happens if you alter the sequence. This approach provides insight into the vision you have for your master data management solution, and you can see the many side effects that almost any decision can have on a master data management system and how it is embedded within the organization.

The following main components are included:

- Data silos

  Data silos are the beginning of it all and ultimately the driver of why we consider the transition to a single master data management system. They represent the individual systems that an enterprise operates. Sometimes they integrate with other systems. They maintain their own business processes, which in turn results in data inconsistencies. As soon as two processes operate on the same data type that is stored in different IT systems, this silo effect occurs and the need for master data management is likely to arise. However, a master data management system does not necessarily replace an existing silo in its entirety. Instead, it eliminates the silo effect in respect to the data, but the true business functionality will still have to be maintained outside master data management.

  Because of that we would rather see a master data management system complementing silos in order for them to operate more efficiently and effectively based on higher data quality. There is the persistent argument that says that you do not require a master data management system when you already have a CRM or ERP system in place.

  Although we acknowledge that these can be used to carry out master data management functionality, you have to ask yourself whether you really would want all other systems that require master data to integrate against your CRM or ERP system. We do not think that those systems were made to support complex integration scenarios.

  That said, even CRM and ERP systems can perform better based on higher quality data when provided through a master data management. In addition to that, the data between both systems will be reliably consistent, and thus instills more trust in the information that they contain and will be derived from through further analytical processing in a data warehouse.

  In the context of master data management, the master data management system will not hold every data attribute that those systems require, but will focus only on the data elements that are required across many systems and that are master data. The remaining attributes will still be maintained in their dedicated systems and enriched by the master data.

- Master data

  Master data represents the new domain to which all other system requiring or manipulating that data will have to be connected. There is an interesting topic to it. Although it is obvious to use ETL when loading the data for the first time, it is a little less obvious what is going to happen after that point in time. This brings us to the discussion about continuous synchronization. There is a need for continued ETL support even after the initial load. One example for that is the required synchronization with an existing system that cannot be immediately retired. When you are building a new master data management system there are a few components, such as the front-end system, that you will also build new. For these components you can use the data structure of the master data management system. ETL is not required. Another common usage scenario, however, is when you attach an old downstream system directly to the new master data management system. Such a downstream system typically operates on a data structure that is different from the data structure coming out of the new master data management system. Then ETL is required to transform between these structures.

- Data quality

  Data quality is the component most related to the data in those aforementioned silos. It is based on thorough data analysis. We achieve that through profiling of the existing data to obtain insight into the current data quality. The result of such analysis is a plain report outlining the issues, but a list of data that requires cleansing can also be produced.

  There are two options that we can apply:

  – Cleansing
  – Standardization

  Cleansing relates replacement of null values or providing default values to fields that are not populated. Such cleansing allows the master data management system to implement specific business rules that do not have to address any different semantics that result from otherwise inaccurate data.

  The other aspect is represented by standardization of data. Here we ensure that names or addresses, for example, are always spelled in the same way. Instead of having names such as “St.” or “Street”, we always standardize on “Street”. Standardization of names of a person and her identifiers are also common. All types of standardization allow the master data management system to match more quickly and with better results. These matches are used in data de-duplication processes. A higher quality of person data and the corresponding 360-degree view is the result of cleansed data.

You can also perform both steps at the same time, standardizing the data while cleansing it. To be able to do this, you also have to carry out the required analysis prior to loading. You do not save much time, apart from the fact that you can test best on the existing source data instead of waiting for tests against cleansed source data.

Now that we have laid out the basic considerations about the data quality component, we also want to discuss the effect that it is having depending on the point in time when it is invoked. The three options for us are as follows:

- Prior to loading data
- While loading data
- After having loaded data

Putting a focus on this topic prior to a master data management implementation allows you to build up the best possible master data management repository with the highest quality of data and results in the lowest number of duplicate or redundant data records. This is due to the fact that the master data management typical data de-duplication mechanisms depend on a clean set of data to work to their full extent.

Because data quality measures cannot always be initiated prior to a master data management implementation, the next best option is to address data quality while loading the data into the master data management system. One scenario where this certainly makes sense is when a system integration does not see immediate system replacement achieved through a single load into the master data management system, but rather a constant synchronization stream between those systems over a long period of time before the old data silo eventually gets switched off. Embedding cleansing and standardization in this process, therefore, allows for such a stream of data with a constant and predictable quality.

If there is only a single load followed by immediate deactivation of the silo and the project has failed to include data quality in advance, then that is where the third and last option comes in, cleansing within the new master data management system itself. This could be achieved through running data quality processes frequently or carrying out corrections as one works with the data throughout its lifecycle.

Although you might want to set up such process in the realms of data governance, we advise against this approach for load processes. The reason for this lies in the impact on functional aspects, including the complex topic of business rules. Bad quality data requires business rules to be defined in a way different from their original intent to cater to the variations that we have to allow for with bad data. Consequently, this can lead to continued bad data in the system, which we cannot recover from during the lifecycle of the system. Such an effect reduces the benefits that the introduction of a master data management system typically provides.

- Data management

  Data management is the start of the new master data management system. It is here that further data analysis regarding its semantics and other properties as simple as checking for field lengths and data types takes place. This metadata should be stored in a separate repository for other parts of the systems to be able to reuse it during design and runtime phases.

  One example is the need to model the master data management repository from such data. Another example is for data stewards of the new data governance organization to be able to get insight into all master data management related data from a single repository instead of having to reach out to multiple places.

  One such metadata repository is provided through the IBM InfoSphere Information Server. The advantage of using this product is that the same metadata can be used when building ETL jobs for DataStage as required for the data integration, as well as data cleansing and standardization jobs using IBM InfoSphere QualityStage® as required for data quality component.

- Data integration

  The data integration component is not required when looking at initial loads only. We require it for online synchronization interfaces. It always comes into the equation when two data structures of two disparate systems, of which the new master data management system is one, have to communicate with each other. It applies to both upstream and downstream system connections. It is here that the typical ETL capabilities are provided. This component can integrate directly with the data quality component to optimize the process of cleansing and standardization. The mapping between the disparate systems should come from metadata information stored in the central repository for data management.

- Data lifecycle

  Any system working with data needs to maintain that data throughout its full data lifecycle. The lifecycle starts with the creation of the data. After the data is created it needs to be maintained. Records are being updated and new information added while some other gets deleted. This also includes changing of roles and building or removing relationships between different entities. Last but not least, the data gets archived.

  The provisioning of an archiving solution to a master data management system should be planned for and incorporated, but this is typically a task related to the infrastructure and does not affect the functionality of the master data management system. It is directly related to the database that is underlying the master data management system. The data is being worked with either through loading using batch or online synchronization or indeed any front-end system that allows for online input.

- Data usage

  Eventually we would like to make sense of the master data and use it to its full extent to achieve the business goals manifested in the strategy. Data usage includes all activities included in the data lifecycle component where data gets maintained. In addition, we manage the processes that deal with the data through their own workflows. There is a second quality to the usage of data though, which is not related to the manipulation or maintenance of data, but to deriving further information from it. Similarly to the workflow, it includes searches to the data. On top of that, however, are reports that are extracted from the data to gain insights and drive business decisions.

3

Model-driven development

Now we consider traditional modeling techniques that document requirements through use cases. Usually, these diagrams come with a written use case document that states all the conditions and errors that need to be addressed. In a more ambitious scenario, designers might come up with activity diagrams and timing diagrams. Then, instead of using a written requirements document, developers implement the program flow according to these diagrams. Now when a feature request is implemented, a designer changes the model and then provides it to the developer, who then extends or changes the code accordingly. If everything works well, this process guarantees that the design and the code are synchronized.

However, this approach includes the following shortcomings:

- There are many possibilities for inconsistencies. In some cases, the model might not be implemented directly because it lacks information or contains element relationships that cannot be implemented due to programming language restrictions. Multiple inheritances are just one example for this source of inconsistency.

- The design seldom reflects each and every detail of what needs to be implemented. For example, if it consists of nested activity diagrams, it does not prescribe the structure of the classes and packages and does not contain technical aspects, such as exception handling, object instantiation and finalization, handling of collections, and so forth. In general, it can provide a valid description of the functional flow, but does not say a word about the technical architecture. To the contrary, it might even run counter to architectural goals because it relies more on procedural flows than on re-usable, object-oriented, and component-oriented design.

The first chapters of this book discussed master data management in general, including facts about master data, the challenges (such as a solution imposes on any enterprise attempting to implement a master data management solution), an architectural discussion regarding the best possible integration of a master data management, the context of master data management related components, and a discussion about the strategy leading to a road map and eventually the master data management solution implementation.

A *model-driven development methodology* for the implementation of a master data management system addresses these issues, including:

- Integration of business processes and data
- Integration of multiple systems, components, and functions
- The scale of the solution in a complex implementation style
- Consistency and traceability

This chapter describes and discusses model-driven development (MDD).

3.1 Introduction to model-driven development

A common misconception regarding master data management is that people consider it almost exclusively as an extension to their database management system (DBMS). This point of view ignores the business processes that manipulate the data. It is true that downstream systems are responsible for the implementation of their own business processes. However, some process elements are applicable to the master data too and thus have to be considered.

The goal is to describe these master data related processes separately to guarantee consistency and completeness. Each of these processes represents a model. More importantly, these business process models are tightly coupled with the data models on which they operate. Business processes are often only verbally described. However, to consistently describe the manipulating effect that those processes have on the data, they need to incorporate the data models.

Thus, projects should maintain both models in a single integrated tool for the purpose of consistency and traceability.

The same holds true in respect to the multiple systems, components, and functions that a master data management solution represents. All of these parts must integrate with each other while acknowledging the differences in their respective data models. The need for transformation between these data models arises. We best describe such transformation against the data models that we have in the modeling environment, and describe the transformation directly against those data models. It is here where we can describe the dependencies or differences between these systems most clearly and accurately.

The scale of a complex master data management solution grows quite quickly into proportions that are difficult to handle by any individual. Using a multitude of different tools that are not interconnected does not make it any easier and is reason enough to opt for a model-driven methodology. It is difficult to describe and comprehend the system behavior, interactions, data models, their mappings and services in separate text documents, spreadsheets, and non-formal flow diagrams.

Such tool choice is feasible for smaller system designs, but can negatively affect productivity and maintainability in large-scale system design. In our experience it is incredibly difficult to keep many single documents consistent all while keeping them simple and comprehensible enough to be useful to the different teams and groups of people involved in the implementation of a master data management system. Even if projects do not consider comprehensive documentation as important as it might be in agile environments, MDD provides you with documentation that matches the generated source code, and consequently project members and businesses benefit from it. One might even hold that MDD facilitates the use of agile techniques in a more formalized context.

Master data management systems are placed in the core of an enterprise. Any change imposed on the master data management system potentially affects many other systems due to service reuse and the common master data related business processes established there. It is difficult to oversee such impacts and side-effects at the coding level. Keeping track of these changes on the design level instead helps to enforce consistency and traceability of these changes. MDD addresses this issue. The result is much earlier identification of impact and is a form of risk mitigation.

Before deciding on the introduction of an MDD methodology, be aware of the following key challenges that you are most likely to encounter:

- *Acceptance* of the effectiveness of an MDD methodology is difficult to prove. Many arguments are aimed at keeping control in developers' hands. We believe that a properly adopted MDD does not imply loss of control to anyone. Instead, it allows everyone to focus on what they know best and address the business challenges that we are here to solve. Developers are still completely responsible for their own code and can continue to be as creative as ever. The only difference now is that they will have to accept a more formal approach to turning a design into implementation source code by providing the required transformations.
- *Skills* are another issue. Even though many years ago model-driven methodologies were already introduced, they have not been fully adopted. Perhaps that methods in the past were not mature enough to deliver the expected results and we are still going through this phase of neglect. Naturally then, we do not have the skills available when there are not enough people working with it. Instead, we tend to live with the status quo. The status quo means to stick with more or less traditional methodologies that everyone still knows best. That also means that we are living with a natural break between design and implementation. Consequently, we need to address the skills shortage through education.
- *Adoption* represents the remaining issue even when you have successfully resolved the previous two issues. This issue is a serious one, because it is not aimed at any single individual, team, or role. It spans almost every team, including project management. In fact, project management needs to support the project in successfully adopting this methodology. Only then can all teams from design, to development, and through to test apply their skills and successfully drive the solution models to its goals.

The following sections describe MDD and domain-driven design (DDD) in more detail. If you are already familiar with these methodologies, you can skip these sections or use them as a refresher.

3.2 Understanding model-driven development

The primary goal of MDD and model-driven architecture (MDA) is to place the model into the center of system development. A *model* represents the abstraction and a sufficient description of the problem domain for which you are creating your solution. Source code and documentation are generated automatically from models. In addition, further models are generated.

An important aspect of a model-driven methodology is the implementation of the application that is based on a model. You can also create a model that is used only for design and documentation of the business requirements after the fact. Benefits of a model from a model-driven approach are often overlooked. You are working from a model-based approach if you create the model beforehand. A model-driven approach, in comparison, includes deriving source code for the implementation from the model.

Another benefit of a model-driven methodology is that it supports rapid change through *design-based impact analysis*. After you have established the impact of the change, you can roll it out in a way that is consistent within the model. Others who are working on the model can pick up and incorporate the change into their tasks. The model also documents the change, and it serves as input for the changes in source code. Consequently, MDD supports the full cycle of an implementation project.

This section provides an overview of the common aspects of model-driven methodologies:

- Introduction
- Objectives
- Realizations
- Processes
- Patterns and guidelines
- Tools

See “Related publications” on page 559 if you need further details that are not described here.

3.2.1 Introduction

Before going into any great detail about MDD, this section introduces the following main concepts:

- MDD layering model
- MDA layering model

MDD layering model

MDD is sometimes also referred to as *model-driven software development* (MDSD). This is a development approach that aims to develop software using different models that depend on each other. MDD is a general-purpose methodology and does not implement specific standards. It serves primarily to raise the degree of abstraction and to bridge the gap between functional requirements and technical implementation. This approach enables business analysts, technical architects, and developers to focus on their particular skills.

Figure 3-1 illustrates this model layering concept on a high level.

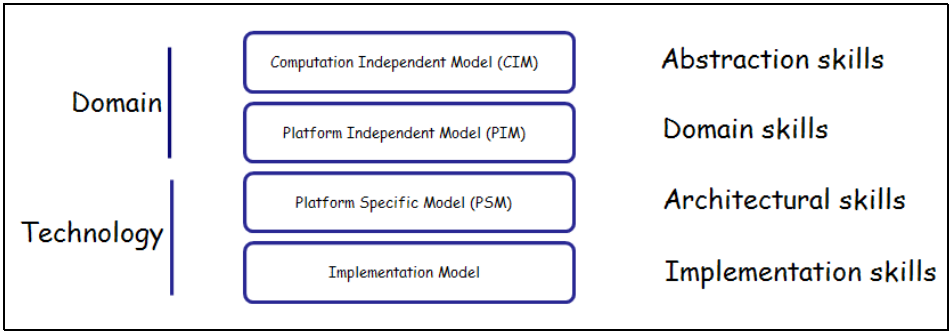

Figure 3-1 High-level model layering concept

This layering approach includes the following distinct layers:

- Computation independent model (CIM)

  The CIM represents the non-standardized functional description of the system. It is the highest-level of abstraction, and includes neither the platform nor the description standard dependent specifics. This means that we solely and generically describe the purpose of the system that we are going to build. This is typically done by means of documents such as requirements and use cases.

- Platform-independent model (PIM)

  The PIM turns the description from the CIM into a standardized notation such as Unified Modeling Language (UML). It is still completely independent of specific programming languages, infrastructure components, and any other artifact of the target platform. It serves as a basis for code generation because of the many domain-related specifics that it captures. Consequently, you find, for example, UML representations of the use cases, but it is also enriched by class diagrams representing the business domain model.

- Platform-specific model (PSM)

  The PDM contains additional information for the PIM. It includes specifics of a programming and execution environment and acknowledges products that have been selected for the implementation and as constraints or options that are enabled by the selected programming environment.

- Implementation model

  The Implementation model layer represents the actual sources. Remember that MDD does not necessarily aim to generate the entire source code. Although the vision is to create executable UML models, MDD focuses on routine tasks and repetitive code. In the context of this book, the term *sources* encompasses the following components:

  - Java source code
  - Documentation
  - Test cases and scripts
  - Build and deployment scripts
  - SQL scripts and DDLs

Regardless of the extent of code generation, moving from one layer to the next lower layer implies transforming a source representation to a target representation. The preparation of the required transformations poses a challenge and demands sufficient ramp-up time in the project. Transformation is thus another key aspect of MDD, which is discussed further throughout this book.

MDA layering model

MDA is a standardized framework for software development. It has been defined and introduced by the Object Management Group (OMG). MDA, as opposed to MDD, focuses on generating infrastructure code, based on standardized techniques. Also, MDA concentrates on the separation between design and IT architecture. MDA defines the layers and components as depicted in Figure 3-2.

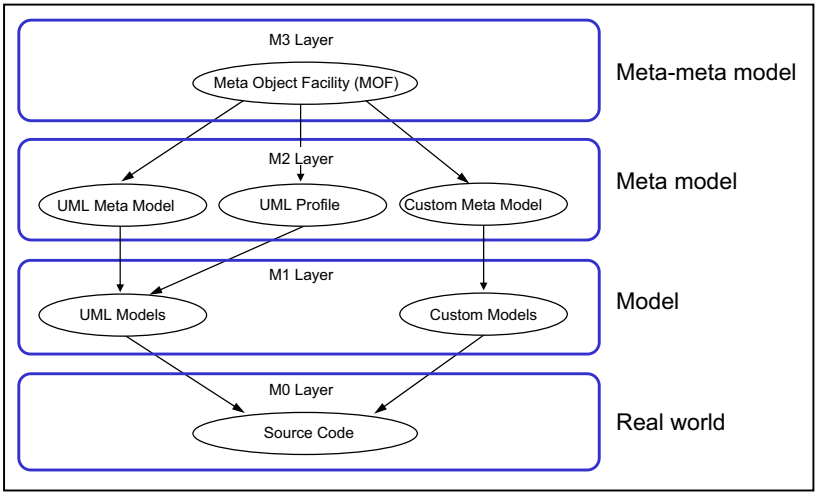

Figure 3-2 Layers defined by MDA

The following layers are defined, where each layer represents an instance of its parent:

- Meta-metamodel layer (M3)

  At the M3 layer, the Meta Object Facility (MOF) defines an abstract language and framework for specifying metamodels. MOF was introduced by OMG and is the foundation for defining any modeling language, such as UML or the MOF itself.

- Metamodel layer (M2)

  The M2 layer encompasses all standard and custom metamodels defined by MOF. Most importantly, the UML metamodel is defined at this layer, as are UML profiles and custom metamodels, which provide a means of extending UML in a specific context. UML profiles consist of UML stereotypes and tagged values that allow you to decorate a UML application with additional meta information and without changing the MOF representation of the UML metamodel. Custom metamodels expand the metamodel by means of the MOF, which provides for more flexibility, but also requires significantly more effort. UML profiles, therefore, are considered *lightweight*.

- Model layer (M1)

  The M1 layer contains the actual models, which include the models that describe the reality that you are implementing. For example, you can find a domain model of a master data management system in this layer.

- Real world layer (M0)

  The M0 layer includes components from the *real world* or, in terms of software development, the source code.

The OMG has defined an interchange standard, XML Metadata Interchange (XMI), that provides a transformation between the model layers, the MOF, and the XML-based metamodels and models.

Terminology note: MDA layering is similar to layering used for MDD. Thus, MDD terms are sometimes used in the context of MDA. This book uses MDA techniques and specifications.

3.2.2 Objectives

The main objectives of MDD are to lower the overall cost of building large applications and to reduce the time that is required to develop applications. You can realize the shorter duration only after MDD has been established and after the required transformations are in place. In respect to master data management, the intention is to reduce the time that is required for otherwise lengthy development cycles. At the same time, MDD positions your solution to be able to handle required changes at any point in time.

In addition, MDD helps projects to achieve a higher level of consistency and accuracy in the design and development phases. It does so by minimizing the gap between design and code artifacts, breaking down the artificial barrier. Consistency also contributes to time savings, but more importantly it drives to higher quality from which the time savings result. A good example of these savings is change requests that can be directly applied to the analysis model and then transformed to the design and source view rather than manually being applied to each and every layer. Minimizing the number of manual changes and bridging between different levels of abstraction also lowers the implementation risk associated with changes like programming errors, unexpected side-effects, and mismatches between design and implementation.

Another important aspect of MDD is the separation of technology and business. Using MDD provides an easy means for separating the business architecture from the technical framework. Both business and technical models are still related through their respective transformations. We thereby achieved the separation of concerns while keeping the advantages of a consistent and interrelated set of models with regard to the framework. MDD makes it much easier to follow best practices simply by applying the corresponding design and implementation patterns to the generated code. In addition, parts of the technology stack can be replaced without having any impact on the business functionality. In reality, these aspects cannot be entirely decoupled, but technological changes such as version updates and changed infrastructures should not have any side-effects on the business functionality in place. That is, because the business model is platform-independent and depends only on the business domain, it is not outdated by new technologies.

Instead of relying on technology-specific skills, MDD enables companies to focus more on business-related skills. In fact, it enables every individual to focus on what she knows best. This objective goes along with the simplification of developing new applications and maintaining existing ones. New applications benefit from the existing software and framework stack and the business model in place.

MDD also uses existing implementations by reverse-engineering the existing code and making it part of the model. The model can then be used to generate wrappers or migrate the code to a new platform.

Finally, MDD enables you to maintain and generate consistent documentation for both design and source artifacts. Here, the model serves as the one and only source of documentation and is generally easier to read and understand, especially by business users. This consistent documentation also helps ease communication between designers, implementers, and stakeholders.

3.2.3 Realizations

MDD centers on descriptive models and transformations between them. You can realize each of the following aspects in different ways:

- Description languages
- Transformations

Description languages

Description languages serve to express the key aspects of a particular domain or problem area. They have the capability to formally describe the domain and are based on an abstract syntax. This definition of the syntax is in itself again a metamodel. The resulting concrete syntax is referred to as the notation. The semantics of the resulting description must be well-defined.

Description languages that comply with these requirements are also known as *domain-specific languages* (DSL), which for the purposes of this discussion include both internal and external domain-specific languages. An *internal* domain-specific language extends an existing language, and an *external* domain-specific language defines an entirely new language based on the domain definitions. Examples of internal domain-specific languages are UML profiles, as discussed previously. Regular expressions and custom languages based on a custom metamodel are examples of external domain-specific languages.

This book focuses on UML profiles but does not use them exclusively. If your requirements include an external domain-specific language, you can adopt the approach outlined here to add a domain-specific language and continue describing other artifacts through that syntax.

Transformations

Traversing between the metamodel layers requires thoroughly designed transformations. MDD turns transformations into first-class citizens. Transformations should always operate on the level of a common metamodel shared by the source and the target representation. If it does not, we create a dependency of the target on the source, which is not desirable, because non-structural changes of the source would then require a modification of the transformation routine.

For example, rather than transforming the syntax of a domain-specific language directly to the code representation, we first parse it into a common abstract model. With regard to programming languages, this common representation is known as the *abstract syntax tree* (AST). The actual transformation is then applied on this level before the target representation is transformed into the target model.

Transformations can be categorized according to their source and target types, as follows:

- Model-to-model (M2M)
- Model-to-text (M2T)
- Text-to-model (T2M)

Apart from this categorization, transformations can also be distinguished by the tool set to be used. This aspect is discussed later in the discussion on UML-based approach.

3.2.4 Processes

Establishing an MDD-based process is a demanding task. You must decide on the components of the solution that you want to generate automatically, the components that you want to become a part of a technical framework, and the components that you prefer to implement manually. You also need to consider the challenges regarding acceptance, skills, and adoption.

The following steps outline how to set up MDD projects. These steps apply whether you are working on projects related to master data management or any other type of project.

Process note: Be advised that these steps do *not* represent a strictly sequential flow, but rather are an incremental list of work items.

The MDD process includes the following main steps:

1. Assess whether you want to use MDD.

   All projects do not need to be based on MDD. The first step is for you to determine whether MDD is the best solution for you. Master data management projects can benefit from MDD for the following reasons:

   – IBM InfoSphere Master Data Management Server (referred to as *InfoSphere MDM Server*) makes use of code generation techniques. Thus, it is consistent to extend this approach to other artifacts as well.

   – The complexity of the master data management system context and the impact that operational MDD has on other applications require the use of a highly structured and reproducible approach.

2. Obtain consent from the key players.

   Because introducing and maintaining an MDD environment requires notable effort, both business and IT must agree to adopting it.

3. Define the overall architecture.

   The architectural blueprint enables architects and designers to identify and elaborate on the details of the MDD environment. It is vital for everyone to understand which model resides where and the components that are responsible for realizing the individual models. In addition, everyone must be able to see how each model contributes to the entire solution.

4. Identify the impact of MDD on your environment.

   Some components cannot be generated automatically, and others should be part of a technical framework rather than being generated. This framework component can then be used by generator components.

5. Define and plan how the MDD environment will be established.

   Planning for MDD means considering efforts related to the MDD environment in any project plan and ensuring that the MDD environment is developed in accordance with the business applications. Plan for ramp-up time for both skills and creation of transformations. Avoid ivory towers. Include several iterations in the plan.

6. Define a UML profile or a domain-specific language.

   A domain-specific language can cover all aspects that are related to your MDD environment. The domain-specific language itself is one of the most important parts of the MDD environment. Consider whether you want to define a domain-specific language or base efforts on UML profiles only. Just like any other source code involved, either must be developed incrementally and must reflect both technical and business needs that evolve over time.

7. Define clear modeling guidelines.

   It is not sufficient to define a domain-specific language or a UML profile. Designers using the domain-specific language need clear instructions about how to use it within your environment. Without detailed guidelines, designs might be readable and understandable but might not work with the code generators or might not lead to the expected results.

8. Develop a reference implementation.

   Usually, rather than developing a reference implementation from scratch, architects start with a prototype that extends over time. Hence, the prototype fulfills two purposes. First, it serves as a template for future implementations. Second, it provides immediate feedback to those implementing the MDD platform and allowing for corrective measures to be implemented.

9. Define transformation rules.

   After you have defined the target and source representations of your models you can create the corresponding transformation rules. As mentioned previously, transformations play a key role in the entire MDD approach. You require sufficient time to define and create these transformations and most likely are going through several iterations before getting them right.

10. Provide tool support.

    The usability and applicability of MDD depend heavily on the quality and the integration of the tool chain.

11. Define clear programming guidelines.

    This item represents the counterpart to the definition of comprehensive design guidelines. The fact that you are generating code does not mean that you can ignore coding guidelines. To the contrary, the more involved the MDD environment becomes, the more important precise guidelines become. Transformations should adhere to guidelines. The generated code has to be maintained and extended in the future and needs to be readable. The integration of your manually developed source code with the generated artifacts should be seamless, and you have to avoid collisions.

12. Train designers and developers.

    This step ties into the point regarding the skills issue. You need to provide sufficient education to the teams in a sophisticated development process. The success of your MDD project depends on both the acceptance and the understanding of the process as a whole.

13. Apply the MDD environment to a real-life project.

    Generate code as defined in the environment, and develop customized code wherever needed.

14. Test the partially generated application.

    Technical tests should consider the fact that parts of the codes are generated automatically. If any modifications have been applied to the code generation framework, you need to test the entire application for regressions. Note that generating code might also imply the generation of test cases. The framework itself must be tested independently without the aid of generators. Apart from dynamic testing, quality checks for static code can also be applied to generated code, but usually require customized quality checks.

15. Start a new iteration.

    As discussed previously, developing an MDD environment typically requires several iterations. This is your opportunity to refactor and adjust the MDD environment according to changed or extended requirements. After the framework is used for productive applications, project teams need to take into account compatibility and regression issues, because any change to existing applications might also go with the regeneration of source code.

All these tasks are highly correlated with each other, and their execution requires a good understanding of the system context and the business requirements.

Now that you are aware of the main steps in the overall process, the next sections take a closer look at the step that generates the code in the process. Some of the activities related to code generation are specific to master data management projects, because InfoSphere MDM Server already provides tools and capabilities. You can use these tools and capabilities in the MDD setup so that you can reuse provided assets.

This discussion about code generation distinguishes between the following distinct natures of the model aspects from which you want to generate code:

- Generating code from static model aspects
- Generating code from dynamic model aspects
- Dynamic model aspects

Generating code from static model aspects

Generating Java classes from a domain model is the most prominent example of *static model aspects*. The generation consists of mapping the attributes, stereotypes, and relationships to the Java representation. The classes that are created do not depend on the context or other more dynamic constraints and, therefore, their generation is straightforward, as is the generation of accompanying artifacts, such as schema definitions. In respect to UML-based models, static model aspects are the generation approach most frequently applied.

This definition considers only the transformation between the M1 and M0 layers of the MDA or the respective platform-independent model (PIM) and implementation model of the MDD. It is the most important consideration, because it is here where source code is generated for the real-world M0 or implementation model layer, respectively. To run through the entire layering stack, you also have to acknowledge the transformation requirement on higher layers. Even though they are merely describing a model-to-model transition, the M1 model that you are transforming into the source code has to correspond to the business point of view. This model in turn is embedded on the PIM layer of the MDD that describes the domain specifics. It also requires a previous transformation from the PIM domain model into a PDM object model.

Apart from class models, you can also generate structural data representations from physical and even from logical data models, as long as the structure, dependencies, cardinality, and so forth do, not change.

Most master data management systems use code generators at this level. Data additions and extensions that a programmer develops for InfoSphere MDM Server, for example, are designed through a model editor in InfoSphere MDM Server Workbench. The resulting Ecore metamodel model is then transformed along with the GenModel into Java code, SQL code, and XML Schema Definition (XSD). The transformation is defined by JET and XSL templates. Apart from the structural code, the generated artifacts also encompass create, read, update, and delete services to create, read, update, and delete the underlying data objects.

Eclipse plug-ins can close the gap between this Ecore-based approach and UML. A plug-in can take a UML class model and turn it into a master data management extension project. A plug-in also can provide stereotypes for these data objects and can contain stereotypes for master data management rules, services, and so forth. Eventually, a plug-in will be based on an IBM Rational Software Architect transformation. The user can run the transformation configuration, which can return a master data management hub project including data classes, code types, error codes, and so forth.

Generating code from dynamic model aspects

Dynamic aspects are rarely covered by generation tools, because it is hard to match service descriptions with the corresponding code representation. Service flows frequently contain much more ambiguity and must be modeled thoroughly to provide a sufficient basis for code generators. You require much more detailed design guidelines and strict adherence to them in addition to well-defined metamodels that guide you through the transformations between the design and development models. The metamodels are required as a means of abstraction and also are used in an attempt to avoid putting the burden for source code definitions onto a designer.

A good starting point for generation of dynamic model aspects is the generation of service descriptions and interfaces. As a first step, we advise you to simply generate empty, annotated services that must be implemented manually. Next, include the call dependencies between several services in the generation process. This step does not yet require the use of dynamic flow diagrams, because it could be modeled as object dependencies. Finally, the objective of the next step is the complete generation of the service implementations from dynamic diagrams, mainly activities. This represents the greatest challenge and a thorough design model.

Master data management systems, such as InfoSphere MDM Server, offer limited support for service generation because the product itself already provides services against the core entities accompanied by detailed documentation. Another reason resides in the capabilities provided by master data management development tools, such as InfoSphere MDM Server Workbench, which generates simple create, read, update, and delete services automatically that go along with every entity. With InfoSphere MDM Server, you do not need these generators to provide dynamic code generation. It is always meant to be left to further customization.

3.2.5 Patterns and guidelines

MDD suggests the use of guidelines. Our own experience also shows the relevance and importance in defining and applying guidelines. These guidelines do not apply to the design or the process, architecture, or coding.

This section discusses the following guidelines that are related to MDD:

- Process guidelines
- Architecture and domain design guidelines

Process guidelines

To achieve the goals of MDD it is crucial to strictly adhere to the model-driven process. Most importantly, code that is generated automatically must be fully regenerated within the build process. That is, generated code must be strictly separated from sources that are manually developed, because it is not desirable to re-apply manual changes to automatically generated code after each build. This type of process jeopardizes the intent to rely on a consistent model only. Another important constraint in respect to MDD is related to reverse-engineering of models. This activity creates round-trips in the models, which counteracts the consistency and traceability of the model.

Developing an MDD infrastructure is an iterative process. Although some framework and transformation-related parts, such as the required guidelines and a few metamodels, can be defined in advance, most of the MDD platform is aligned with the development of the target application, which enables the infrastructure team to respond to feedback immediately. Rather than developing the MDD platform separately, it should be extracted during application development.

For you to benefit most from an MDD, define a rich description language and a rich implementation model. *Rich* in this case refers to the number of models that you define and the number of descriptive elements that they contain. The more specific the models are, the more code that can be generated automatically. Do not be overly ambitious, however, and start with the richest models possible. Instead, start with a simple platform in the first iterations that might enable only the development team to generate glue code automatically. After you have mastered this level of code generation, you can establish a more sophisticated generation based on additional information that you capture in the models and that drives your automatic generation.

Architecture and domain design guidelines

Consistency and accuracy of the metamodel is a key aspect of MDD. The model must be easily understandable and observable. Designers and developers should be able to understand the process behind the tool chain quickly. Acceptance is at risk if they find it cumbersome to adopt the MDD. After the transformation from model to source code, the generated code should be easily readable too. While developers should not modify the code directly, they should still be able to understand it in the same way that they read through and understand manually written code. Only if the development process adheres to such principles will you reach the goal of an easily maintainable code.

It is important to keep the technical domain models strictly separate from the business domain models in order to work on the same domain model in parallel. Although you still add new business subdomains to the business domain model, you can enrich the technical domain model with the required technical details. A similar differentiation applies on a technical level with respect to the separation of generated and non-generated code.

Based on this guiding principle, ensure that the technical framework provides design patterns that support this separation of concerns, for example, factories and dependency injection. It is difficult to provide one domain-specific language for the entire problem area, especially in a large and complex master data management solution that spans multiple subdomains. Consider the definition of multiple different domain-specific languages or UML profiles for each distinct subdomain. Separate technical subdomains in a similar fashion, because different parts of the solution might require different domain-specific languages that are based on different technical approaches. A prominent example of unique subdomains is cross-cutting concerns and systems integration.

Another means of separating concerns is the use of aspect-oriented programming (AOP). In object-oriented programming languages, *aspects* are related to cross-cutting concerns of an application. These concerns refer to programming logic that spans several domains. AOP focuses on technical functions, such as logging, exception handling auditing, and so forth. Consider adopting AOP for business-related logic such as the handling of rules.

Due to the various layers that you have to support, find ways to an aspect-oriented description in all modeling layers. Using aspect-oriented techniques enables designers to separate these aspects from the business logic flow in the design phase already. Using aspects in the models provides the opportunity either to use a standard aspect weaver to generate the aspect-related source code or to implement this functionality in the generator, which might be more straightforward in smaller projects.

3.2.6 Tools

MDD is based on automatic generation of source code from models. For this automatic generation, you need to identify tools that achieve this goal. Although many tools provide these capabilities, we base the approach that we describe in this book on Eclipse, IBM Rational Software Architect, and IBM InfoSphere Data Architect, respectively. This section describes the foundation that the tools support. Support by the tools is required for the following components:

- Technical modeling components
- Transformations
- Rational specifics

Technical modeling components

Using the Eclipse-based tools, which both IBM Rational Software Architect and IBM InfoSphere Data Architect represent, dictates to a certain degree what components the generation process is going to use. It makes use of a subset of the MOF used within Eclipse, named *Ecore*. To be more precise, the Ecore model is the Eclipse implementation of MOF, the *essential* version of MOF.

The Eclipse Modeling Framework (EMF) is another cornerstone of the code generation process in Eclipse. EMF serves as a foundation of the UML and data models and also supports the transformation of the models into code. It provides a platform for all modeling frameworks and extensions in Eclipse. One of these modeling frameworks is the UML implementation that comes with the integrated development environment (IDE), UML2, and EMF Object Constraint Language (OCL), the implementation of the OCL standard.

Figure 3-3 illustrates an overview of the Eclipse metamodel.

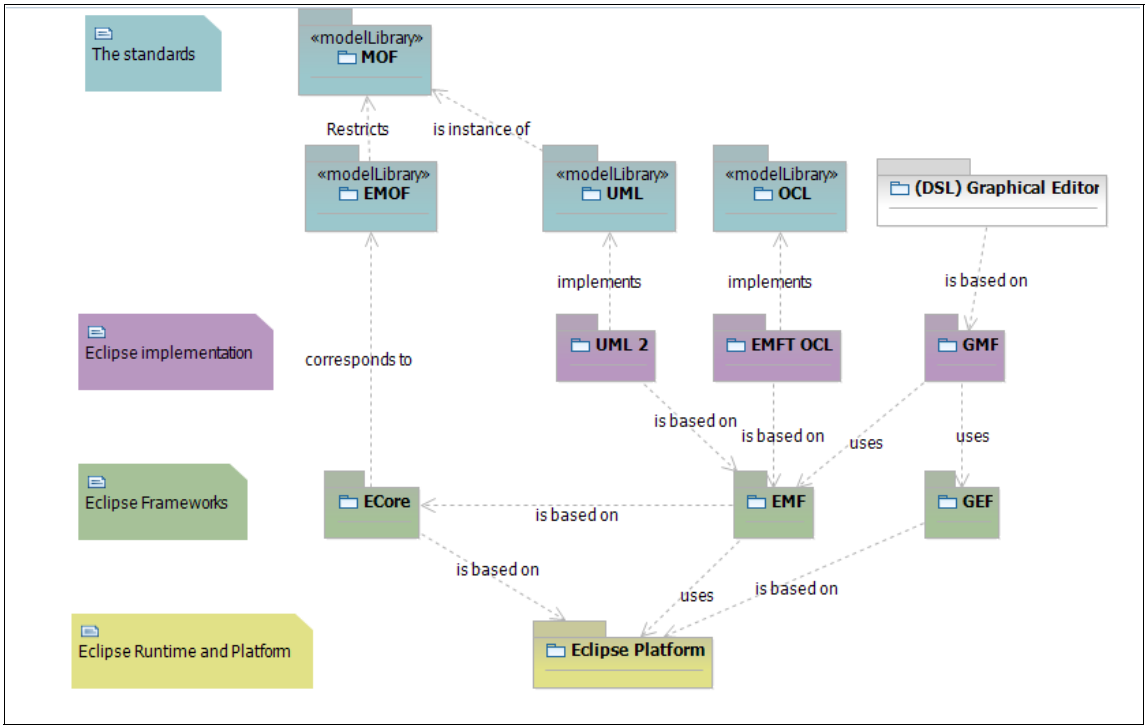

Figure 3-3 Eclipse metamodel

The Graphical Editing Framework (GEF) and its modeling extension, the Graphical Modeling Framework (GMF), serve as a platform for modeling tools. More specifically, they serve as the base for graphical modeling editors. Both are based on the EMF. These editors are frequently used in the context of domain-specific languages and ease the burden of writing specifications in the domain-specific language manually.

Transformations

Support for this important aspect in MDD is also generally provided by Eclipse. Eclipse relies on a template-based approach. The Java Emitter Template (JET) uses a syntax similar to JavaServer Pages (JSP) and can contain dynamic expressions, Java scriplets, and directives. The JET provides a mechanism to transform models to text representations. To use the JET, you can either enable the JET within existing projects or create new JET transformation projects.

Figure 3-4 shows the generation process with JET in some more detail.

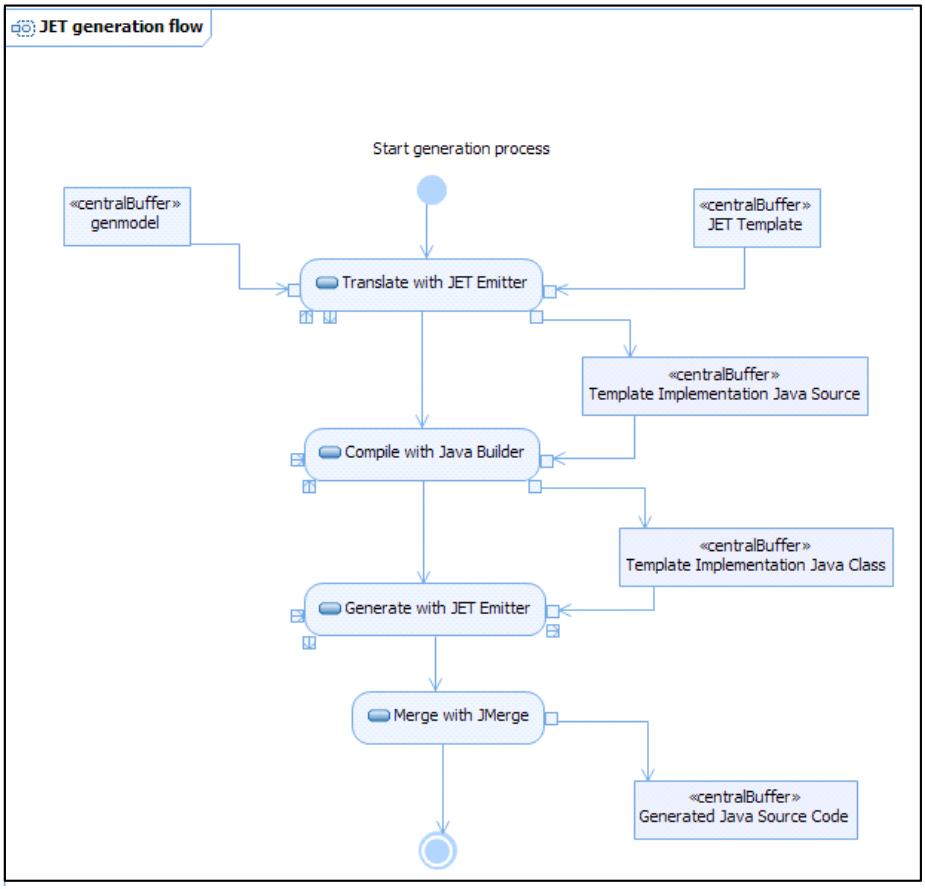

Figure 3-4 JET generation flow

The JET takes XML or EMF models as input, which is, more precisely, the underlying Ecore metamodel. The templates translate these models into Java template classes and finally into the actual implementation classes. You can apply a JET to models with a high degree of abstraction directly, which makes the required JET unnecessarily complex. The recommendation is to create an intermediate model explicitly. In our approach, the intermediate model is represented by the implementation model.

After the JET is applied to the corresponding Eclipse project, the translation happens automatically. Another artifact that is required for code generation is the GenModel. This file contains generator properties and will be synchronized with the Ecore model automatically. Logically, the GenModel belongs to the group of platform-specific models.

JavaMerge, yet another component of the framework, provides a means of integrating manually written code and generated code by annotating Java classes. Source methods will be regenerated and overwritten or not when the JET transformation is run again, depending on these annotations. This component also formats the generated output.

Rational specifics

The IBM Rational tools (for which we also include IBM InfoSphere Data Architect) provide several enhancements and extensions to Eclipse. Most importantly, they offer a transformation API that encompasses support for model-to-model and model-to-text transformations, as well as
predefined transformations.

Figure 3-5 shows the layers of the transformation components.

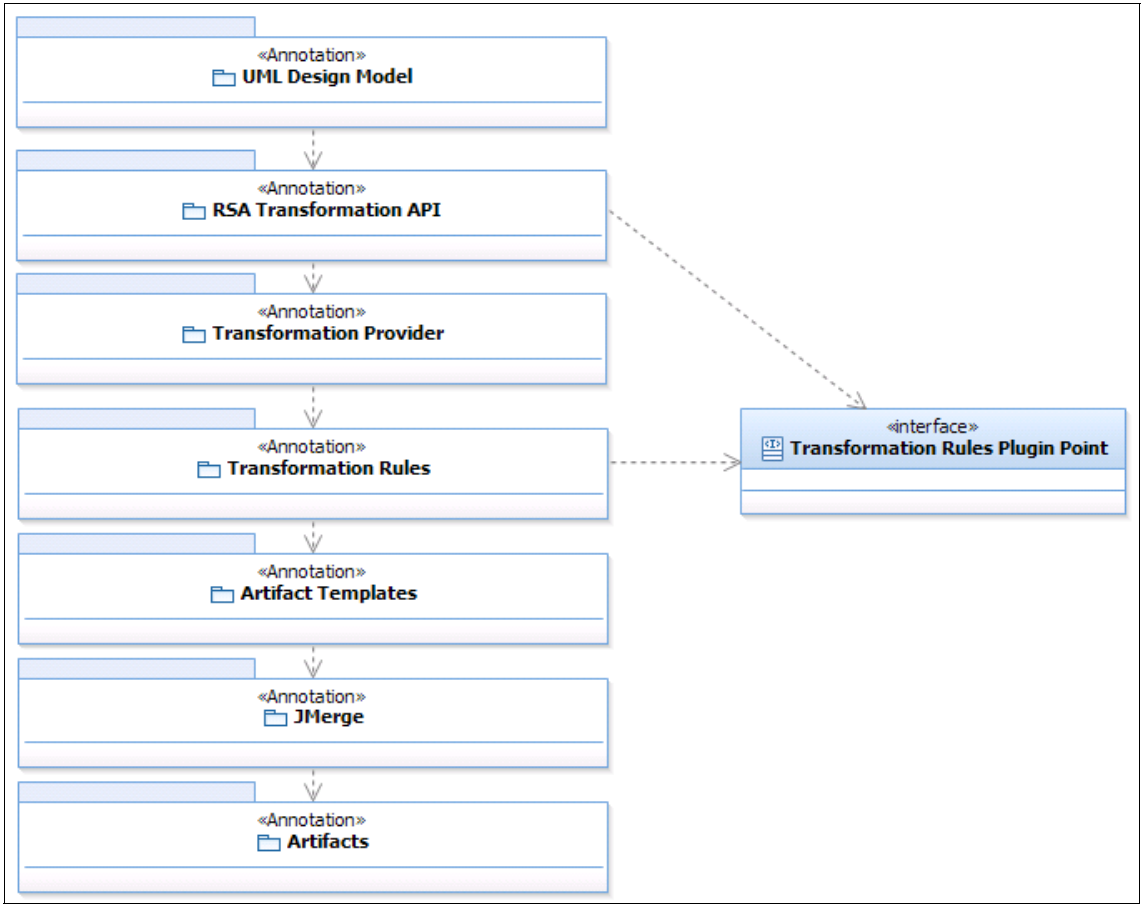

Figure 3-5 Layers of transformation components

Most notable are the plug-in extension points provided by the IBM Rational Software Architect API. These can be used to extend existing transformations by writing custom transformation rules. They also allow you to configure existing transformations, which is exactly what happens when you are creating a new transformation configuration.

Based on this principle, you can adopt various changes. In a true MDD, that is using most capabilities provided by these tools to work as effectively as possible, you will probably create your own UML profile support with your own custom transformations. The benefit comes from the fact that your custom transformations are fully aware of the extensions that you described in your own UML profiles, for example, stereotypes or relationship types.

3.3 Domain-driven design

Although MDD is one important cornerstone of a holistic modeling approach, domain-driven design (DDD) is another. MDD focuses on the different types of models and their automatic generation into source code. DDD focuses on the business functions that you need to implement. Thus, the decision for an MDD paves the way for a DDD. This way the fact that master data is a discipline that is primarily based on business initiatives is acknowledged. It revolves around business goals, the data, and corresponding processes that the business requires to achieve its goals.

This section describes the most important aspects of a DDD approach. DDD is predominantly based on the following principles:[1]

- Base complex designs on a model.

  A complex design should be based on a model. Undoubtedly, a complex design is what is represented when designing a master data management system, in particular when implementing a complex style such as a transaction hub with multiple upstream and downstream synchronizations. If it was not based on a model, it would almost be impossible to capture the business details and convey them consistently to all teams involved.

- Focus complex designs on business logic.

  Such a design should be focused on business domain logic instead of any number of technical aspects driving the implementation and realization. The business is the sponsor of such initiative and so everything should revolve around the business domain logic. This also represents the best correlation to the business strategy, which is the aim. The models can also be used to relay back the captured business logic to validate its correctness with the business.

The absolute focus on the domain within the context of an MDD also helps overcome the traditional gap between business and IT. We acknowledge that the technical details are relevant and important, which is why we use DDD principles within the larger context of an MDD. However, as opposed to the aforementioned book about DDD, the approach taken here goes a step further and demonstrates two important aspects:

- All required models are tightly integrated.
- There is an even stricter separation of business from technical concerns.

[1] Evans, et al., *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003, ISBN 0321125215

Despite the fact that any tool, or in fact any combination of separate tools, can be used to achieve the goal, we believe that the decision for a single tool and UML design approach is an important one. We can only achieve a largely consistent model throughout the entire lifecycle of not just the project when we are using a tightly integrated tool set or a single tool. The entire system has to be built in such way that maintenance teams also will later be able to use the design artifacts, as well as test teams determining their test of business scenarios. The advantage of UML is that typically it also can be read easily by individuals who have never worked with this notation before. Only a few of the possible diagram types have been used to increase readability even more.

To address the challenges outlined and support the selected design approach, a number of models have to be taken into account and worked with. Most importantly, the models can be grouped in the following distinct functions. For now, this discussion focuses solely on the following business design-related models:

- Analysis models
- Implementation models

There are a number of models required purely for analysis. This does not mean though that the development team will never look at them. Rather, everyone will have to gather intelligence about the solution to be implemented from analysis models, and any decisions are based on information obtained from them. It is here where the business domain model comes into existence, based on the skills, knowledge, and experience of the business people knowing their domain inside-out.

Another group of models is dedicated to the implementation. Again, this does not mean that the analysis team can ignore the existence of the implementation models. They will have to consider them with each and every design decision. Those models serve a different purpose and are of a much more technical nature, even though not entirely technical. The implementation models are where we bridge the gap between business and technical aspects. They are the foundation foran MDD taking the step from design to automated coding based on code generation.

The domain model plays a central role in both groups and functions. It is the glue between the business analysis and the technical implementation. It ensures a consistent language and model, not just for business or technical people, but even more importantly across that traditionally tricky border. It also makes it easier for two technical teams discussing the implications of changing an interface or discussing expected versus current behavior of the application. Therefore, it is discussed twice in the following chapters, first from a business perspective, then later from a more technical perspective.

From the content in Part 2, “Smarter modeling” on page 129, you that we are working with three variations of the domain model:

- One purely related to business analysis
- One enriched instance related to design
- The third instance further enriched by purely technical aspects

The metamodel appears multiple times too for a similar reason. Every team, enterprise architect, business, design, and implementation needs to understand the scope in which it operates and how all relevant artifacts depend on each other. Hence, there are multiple views, each focusing on the relevant aspects for the respective teams.

The correlation of the domain model as the new logical model in the middle to the enterprise conceptual model requires definition. We denote that the establishment of a domain model does not make any conceptual enterprise model obsolete. Instead, both models work in unison. The goal of the enterprise model is largely to document all data-related entities across all systems throughout the entire enterprise. Therefore, an enterprise model is of a classifying nature to keep it simple considering the many domains that we typically find here.

The enterprise model defines all entities within an enterprise and helps define the problem domains that we are looking at during the master data management implementation. In any project, we typically only address a subset of the entire enterprise domain in the master data management implementation. Domain models really only cover parts of what the enterprise model describes. They usually do that by adding more domain-relevant details. The enterprise model can help set the technical scope in such a respect that it defines the data elements that are part of the overall targeted master data management solution in addition to the identification of the systems where this data is stored.

Because the conceptual enterprise model records entities across all systems, its level of detail must not be too deep or else no one can oversee the model any longer. A domain model, however, requires an appropriate level of detail for the domain that it represents. The result is that there are many domain models dedicated to the various business aspects and systems in place. They also vary. A domain model for a fashion retailer, for example, might require a certain depth in the product domain, detailing all required fashion attributes such as products, collections of products, their sizes, colors, and many more entities and attributes. Such level of detail, however, is certainly not required for the customer relationship management domain, where we might want to store only the products in which a customer is generally interested.

Master data includes data, entities, and attributes that are used across all domains independently. To keep with the fashion retail example, here the level of detail is not as high as in the product domain of the same enterprise, but it might be deeper than for the CRM system, if product data should be integrated into the master data domain.

In addition to the key model, the domain model, a number of other related and highly interconnected models play an important role in the successful implementation of DDD. Leaving out typical architectural artifacts such as system context, architectural overview diagrams, operational models, and so forth, the models most interconnected with the domain model are the following:

- Business glossary

  Defines a common language for everyone.

- Requirements model

  Ensures traceability.

- Component model

  Associates the design artifacts with an appropriate component.

- Data model

  Describes the model in which the data eventually gets persisted.

- Use case model

  Provides information and definition about system behavior.

- Business process model

  Determines sequences of actions carried out by the business.

- Service model

  Represents the definition of services that other systems can use.

- Metamodel

  Gives an overview of how all modeling artifacts are interconnected.

With MDD, many models are established, possibly even a domain model. However, the challenge is to make all the models work together toward the same goal. You need to do more than simply establish a domain model or create the business process and use case models if they do not acknowledge the objects from the domain model that they are to manipulate. Similarly, a number of services are included in each use case that is defined, which also manipulate data objects from the domain. It is here where this approach ensures consistency by defining the relationships between all models.

The following sections describe the intent and purpose of each model.

3.3.1 Analysis models

The *analysis models* provide the business view of the design. They are based on the customer's verbal and textual input and possibly include already existing models. They include definitions about the domain, domain design and business process, use case, and user interface flow design. These are the models that we focus on here:

- Business glossary
- Requirements model
- Domain model
- Business process model
- Use case model
- Metamodel

Business glossary

The *business glossary* is one of the most forgotten models, and its relevance is often underestimated. Consider a new member to the project team. This person might have arrived from a similar project with a different customer or system. This person might assume that he knows what a certain business verb refers to. More often than not, such assumptions go terribly wrong and people, thinking that they are both talking about the same thing, do not realize the mismatch for some time.

The glossary serves a greater purpose than just allowing new members to the team to learn the project lingua. It should be used throughout the entire project and system lifecycle for communication between business and technical people, as well as designers, architects, and developers. In addition, naming conventions for the implementation should also consider names from the glossary to end up with a meaningful implementation.

It is wise to get such a glossary started as early in the project phase as possible, though it is not possible to create a final version right away. Many verbs and terms only find their way into the glossary when someone asks for a certain meaning. The glossary will grow gradually throughout the phases of the project. Hopefully, there will be fewer and fewer terms to be added towards the end. Also be sure to chose terms carefully and define them precisely. Over time you will return to and rely on the definitions given there. Changing them afterwards can prove to be costly.

Such a glossary should also extend across systems, bridging the gaps and possibly exploring different meanings of the same word for each system. Most importantly, the glossary acts as the enabler and provides the basis for the domain model itself. It is for that reason that we see the glossary not solely as a business initiative, but also including enterprise architects that have a clear view across all systems in the enterprise.

Requirements model

The *requirements model* is a standard model as far as IT architects are concerned. As such, this should not be further explored, except for its relevancy in conjunction with a DDD and its inter-connectedness with other models therein. There are the non-functional requirements such as availability, security, reliability, performance, and so forth. Those kinds of requirements bear no relevance to the domain model itself, all of them being runtime-related qualities of the system. The same is true for the non-runtime qualities such as maintainability, portability, and manageability, to name a few. There are some non-functional requirements, though that will have an impact even on the domain model. These are all categorized as business-related constraints and include topics such as geographical, organizational, regulatory, and risk constraints. You can see how the inclusion of certain regulatory requirements can affect the domain and other models if a new regulatory requirement is imposed on your business. Service execution flows, component layout, and the operational model might also have to take these parameters into consideration. Everything that is built into the system should also be traceable back to its origin, the requirement.

Unfortunately, there is no immediate pressure to add a requirements model to the design. Nothing, during the stressful setup phase of the project, indicates why it should be used right from the start. The reason why it should be used is that it provides traceability. This means that at any point in time someone (and there will be at least your own project manager interested in this) can run a query and see which use case has been implemented where and results in which service or number of services that have been identified for this. We are also going to explain how traceability provides the foundation for impact analysis when dealing with change requests.

In respect to the domain model, the functional requirements will play the main role. Every entity identified and added to the domain model will have to prove its relevancy against these requirements. This is also true for other artifacts created with every functional requirement that will have to be satisfied, and it needs to be proven that it has been satisfied through traceability against this model.

Domain model

The *domain model* is the centerpiece of all design and implementation activities. Due to its nature of being closely related to the business, the domain model can be seen as a visual extension of the business glossary. It puts all verbs and terms into context through visual associations, which are then clearly described. It will be the only model always and under all circumstances being referred to when discussing business logic, enhancements to existing business logic, and changes.

Figure 3-6 depicts a simple business domain model.

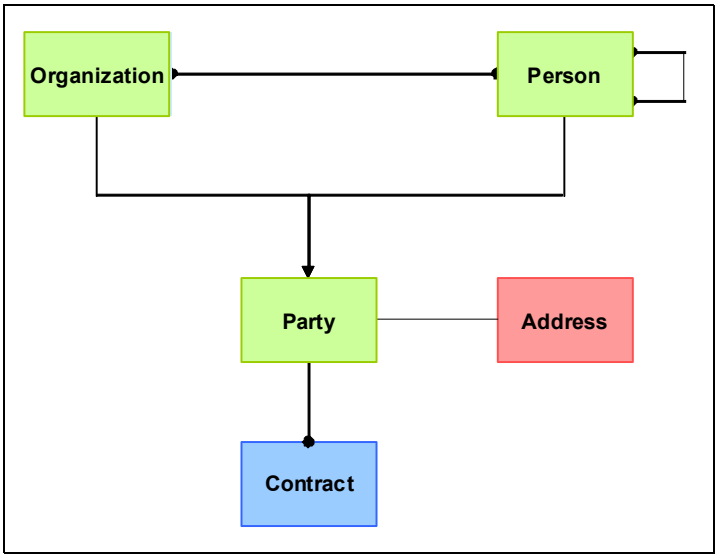

Figure 3-6 Sample of simplified business domain model

No area that is relevant to the system under development should be left to interpretation or assuming that people just know what will have to be done. Instead, the purpose of a domain model lies in exactly describing such relevant concepts in order for everyone to be able to follow them. Do not include any details that are irrelevant to the business domain because that only clutters the design, causing one to question these areas at best or fail to understand them altogether. A domain model with all relevant but no unnecessary detail is the result.

The key concept of a domain model is that it captures all essential detail about the business. To succeed it will have to determine entities that describe the business sufficiently. We need to distinguish between entities that are accessible through a business key, entities that have their own lifecycle, and entities that resemble objects that are dependent on entities with business keys to be retrieved, the value object, or value types. The crucial part of this is to prevent concurrent updates to the same object instances using the business key as the lock to get inside the system.

You can decide on the logical data model of the InfoSphere MDM Server as the domain model to be used within the project. The standard domains that the InfoSphere MDM Server provides are party, product, and account. Consider that this domain model is based on industry-proven patterns and thus represents a robust model, and represents a good choice as such. It also represents a generic model, which might be a reason why one would not want to use it for this purpose.

Figure 3-7 depicts a simple server domain model.

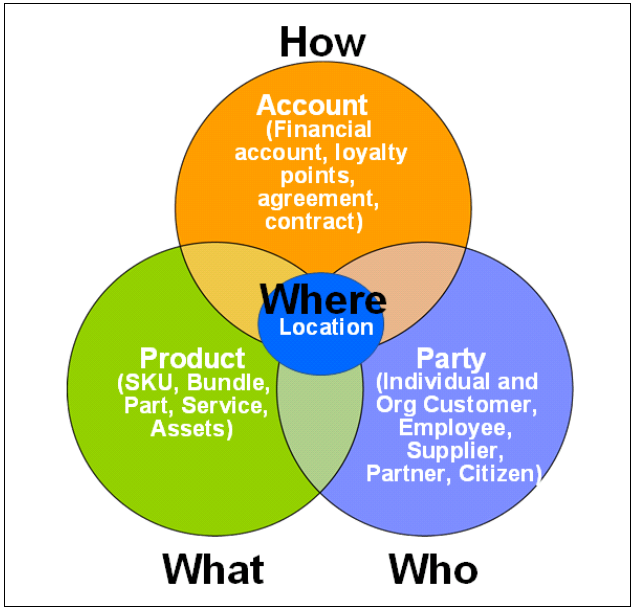

Figure 3-7 InfoSphere MDM Server domain model

Whether you adopt a data model as it ships as the domain model or develop a customized data model, it is a good idea to fall back on existing analysis patterns for the domain model. This prevents you from reinventing the wheel over and over again. In the best case, you can use an existing pattern from a previous project, or a public resource can help you identify the right model. If that does not lead you anywhere, there is always the possibility of falling back on proven industry-specific models, such as the IBM InfoSphere Industry Models. If that is not an option, you probably have to define the domain model from scratch, for which you should plan sufficient time.

After you have defined the domain model, you can host the following other design artifacts, as shown in Figure 3-8:

- Business rules
- Lifecycle

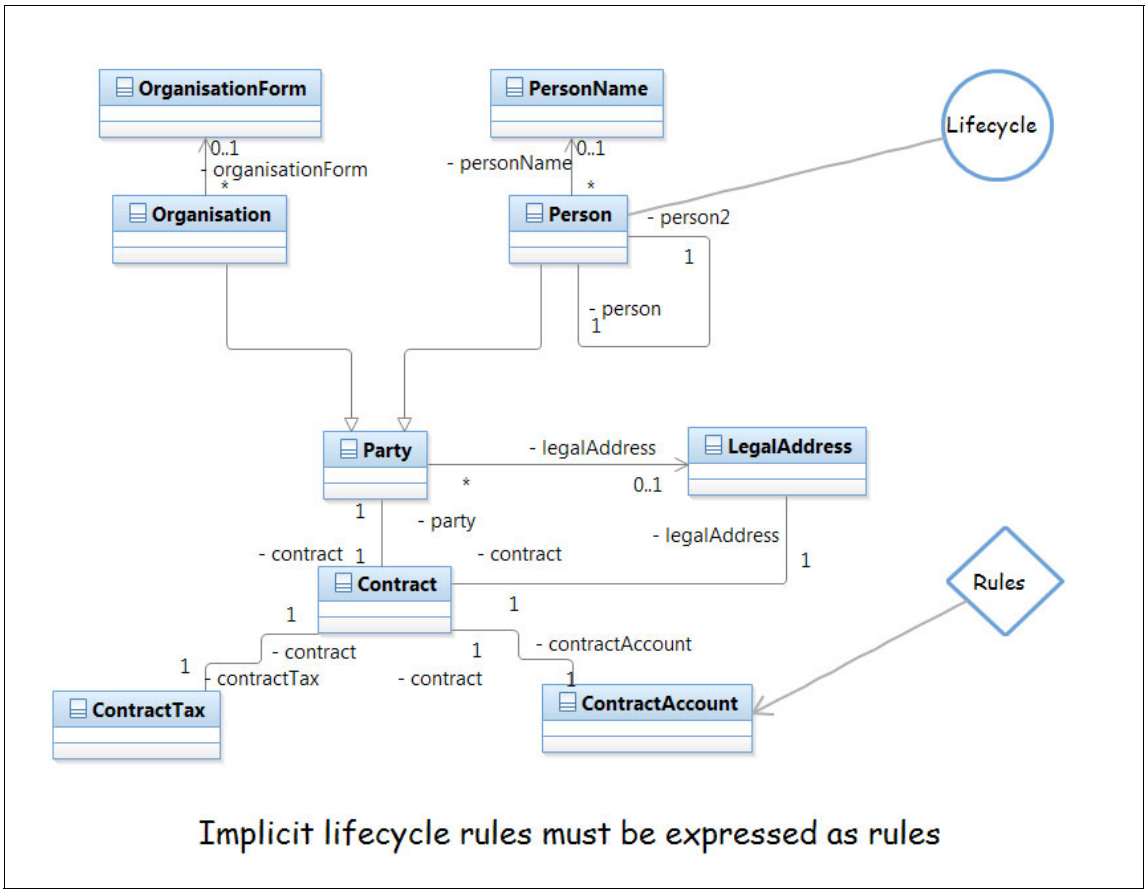

Figure 3-8 Sample of simplified lifecycle and rule

Business rules

During the modeling process, the term *business rules* is used in two different contexts. First, an organization can consider everything as a business rule that fits into the following areas:

- Validations and consistency checks
- Business constraints
- Generally accepted and enforceable business flows
- Associations and relations between business entities
- Decision models and condition matrices

This definition is much too broad to be useful in the context of design models. Therefore, one important outcome of the design process must be a clear distinction between flows, associations, checks and constrains, and actual business rules. According to the Business Rules Group, business rules maintain the following characteristics:

- Primary requirements, not secondary

  Rules are essential for, and a discrete part of, business models and technology models.

- Separate from processes, not contained in them
  - Rules apply across processes and procedures.
  - There should be one cohesive body of rules, enforced consistently across all relevant areas of business activity.
- Deliberate knowledge, not a by-product
  - Rules build on facts, and facts build on concepts as expressed by terms. Terms express business concepts, facts make assertions about these concepts, and rules constrain and support these facts.
  - Rules are basic to what the business knows about itself, that is, to basic business knowledge.
  - Rules need to be nurtured, protected, and managed.
- Declarative, not procedural
  - Rules should be expressed declaratively in natural-language sentences for the business audience.
  - A rule is distinct from any enforcement defined for it.
  - A rule and its enforcement are separate concerns.

- Well-formed expression, not ad hoc
  - Business rules should be expressed in such a way that they can be validated for correctness by business people.
  - Business rules should be expressed in such a way that they can be verified against each other for consistency.
- For the sake of the business, not technology
  - Rules are about business practice and guidance. Therefore, rules are motivated by business goals and objectives and are shaped by various influences.
  - The cost of rule enforcement must be balanced against business risks, and against business opportunities that might otherwise be lost.
- Of, by, and for business people, not IT people

  Rules should arise from knowledgeable business people. This is an interesting point not to be overlooked. Although business people might not necessarily have a precise understanding of what business rules in a closer sense are, they are the ones to define them. It is up to the analysts and designers to separate this knowledge from other kinds of business knowledge.
- Managing business logic, not hardware/software platforms

  Rules, and the ability to change them effectively, are fundamental to improving business adaptability.

A business rule is the part of the business logic that is related to domain entities, as opposed to flows or model relationships and constraints. Everything you have modeled inside the static domain model does not represent a business rule anymore.

Business rules always depend on information that is stored in the domain model. As such, they must be checked upon within the constraints of this model. Thus, the logical place for rules is a class on the domain model. The class to be selected depends strongly on the initiating event and therefore the attribute that is changing and consequently causing the rule to be checked. Originating on this class, there can be a number of reads of other related classes that must be checked within this rule, which in turn requires traversal operations through the domain model that are accommodated for by the implementation teams. These objects can have business rules or lifecycle logic as logic associated with them.

Rules can be modeled as an activity diagram located on the originating class. Here, business rules determine operations to maintain the integrity of the data within the domain model.

Although externalized rules are important in this particular aspect of business, you can implement the domain rules that are outlined here locally, without a sophisticated rules engine, because of the low frequency of changes to the domain logic. The consistency rules that are required to keep the domain-related master data in good shape are simply not volatile enough to justify such a step. This architecture saves you from planning for an external rules engine, such as IBM ILOG®, at least for this aspect of the domain. However, there can be other rules that are much more volatile and that the business wants to update frequently to adapt to changes in their operations.

Lifecycle

The *lifecycle* is described for some special entities. These are typically core entities, such as persons and organizations or contracts. These typically carry an explicit lifecycle. Explicit lifecycle states will get stored with the entity. The preferred place for such a lifecycle is within a state diagram directly associated with the class that represents the entity in question.

Some entities can carry their lifecycle implicitly, in which case the state will not get stored, but can instead be determined on demand. This is based on the interpretation of attribute values or object states.

Lifecycle operations can be placed in a state diagram on the respective entities. They represent a specialized part of the business rules mentioned above.

Packages

Packages, sometimes called *modules*, can help you reduce the complexity of large domains. They allow you to focus on a small subset of the entire domain instead of always being faced with the model in its entirety. The content of a package is identical to the fragment within the domain that it represents. It does not deviate from the representation there. The thing that you should watch out for while creating these packages is that they should cover a logical set of business concepts. Ideally, each package can be implemented by itself.

You might want to create packages for the following reasons:

- Size
- Structure

The size of a domain can be measured by number of entities, number of relationships between entities, and number of distinct logical areas within the model. Sometimes, the domain model has gotten so large, that it is virtually impossible to grasp all aspects at once. Also, it is not practical to cover it all in a single model. Many projects will occupy a number of dedicated teams developing in parallel. Only rarely do you have the luxury of an extended timeline that allows you to pursue a sequential implementation.

Creating a package turns the focus on the smaller scope and concepts that are addressed by the package. As a result, it is much easier to identify packages that can be worked on in parallel. It also is much easier comprehending the business problem that we have to solve as part of that subdomain.

Structure of the project plan is something we have already referred to. Packages provide a perfect opportunity to structure the project plan. You have to identify the logical subdomains and define them on a level of granularity that you deem doable within a project iteration. At the same time you have to consider business process flows. The combination of subdomains and dependencies obtained from business process insights allows you to identify packages that can be developed in parallel as well as the packages that have a logical dependency and thus must be implemented sequentially. You can bring the insights that you have gained from the package or subdomain into the project plan. This approach takes you a step further than simply defining the work breakdown structure (WBS) and already works to the definition of dependencies. Packages can significantly contribute to the project plan and the teaming structures where subject matter experts can be assigned to each package, or alternatively and depending on the team structure and size of the team, assigned to special sub-teams.

Packages that require another package to be implemented to function require a sequential dependency. Other packages can be implemented in parallel because they have no impact on each other. The downside of such an approach is added communication overhead to explain to the project staff members the dependencies and the constant shift in priorities and focus according to the packages and their dependencies identified. This overhead, however, is worth your while because the complete focus on a small domain ensures better comprehension of the problem domain. As a consequence, this produces higher quality results and fewer errors.

Figure 3-9 shows a simple domain model with its logical packages.

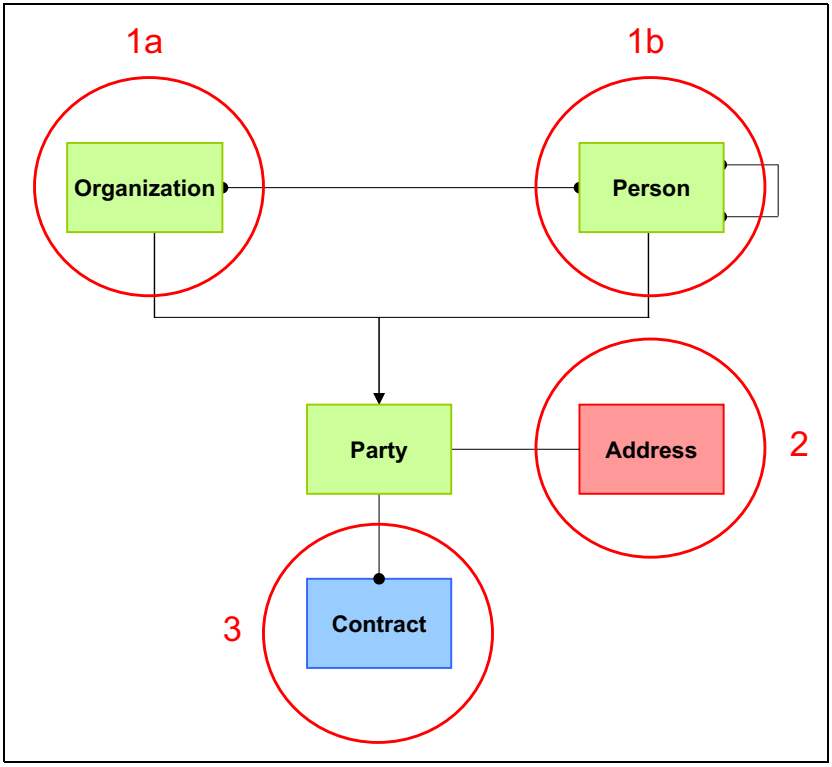

Figure 3-9 Sample of logical packages

This is the highly simplified domain model previously shown in Figure 3-6 on page 103. The highlighting circles (red) show the package definitions. In this case, there is package 1a that deals solely with organizations, as well as package 1b that takes care of natural persons only. Apart from the one association between them, both packages are not related to each other. Hence, there are barely any dependencies that allow for parallel development of the two packages. The dependency only comes into play for a certain type of organization, which we can deal with last when implementing the organization. Hopefully at that point in time, the person already allows for that dependency to be maintained.

The second package, package 2, takes care of the address subdomain. Addresses are dependent on the existence of the parties, either organizations or persons, to which they belong. They could not be created without one. Hence, this package has a strong dependency within the project plan and must be dealt with after the two parties have been finished.

The third package, package 3, deals with contracts, which again depend on parties with their respective addresses to exist. Again there is a dependency in the project plan that needs to be considered. You can only make the contract part work if you have the two other packages implemented already.

The advantage of this approach is the opportunity to define a clear release plan for the project. Release 1 could provide for a party-centric part of the system dealing with organizations, persons, and addresses. This includes packages 1a, 1b, and 2. The benefit is that it also allows the test team to focus on exactly this area of the system. The 2nd release could then provide for the complete system complexity, including contracts in addition to the party-related data. This adds package 3. Again the test team would be able to deal simply with the contract-related additions to the system instead of having to face the entire complexity at once. You will have to carry out regression tests and some special dedication to the interconnecting areas.

Business process model

The *business process model* is required to document the intended flow between use cases within a transaction. Besides the dependency information that we can derive from it for our project plan, it is vitally important for any test team to be able to read and understand the relationship of service invocations across use cases and business processes. Only with this insight, the team will be able to execute the required sequence of services to simulate such a business process and provide for a proper system test.

From our experience you render your maintenance team inefficient without providing them with the business process model for your system. During maintenance in particular, you are required to navigate through a process model in an attempt to understand the dynamic system behavior. By doing so, you are able to map your observations to the problem that the system experiences and possibly find a solution to fix that problem.

Figure 3-10 roughly outlines the relationship between the business process and the use case model. More importantly, it also shows the relationship to the domain model. The diagram at the top represents a normal business process diagram where an event invokes a business process. The key here is that the business process operates on data objects from the domain model. These are represented by the red object instances. This is again one of the key differentiators where we establish a close connection between the domain and the business process model.

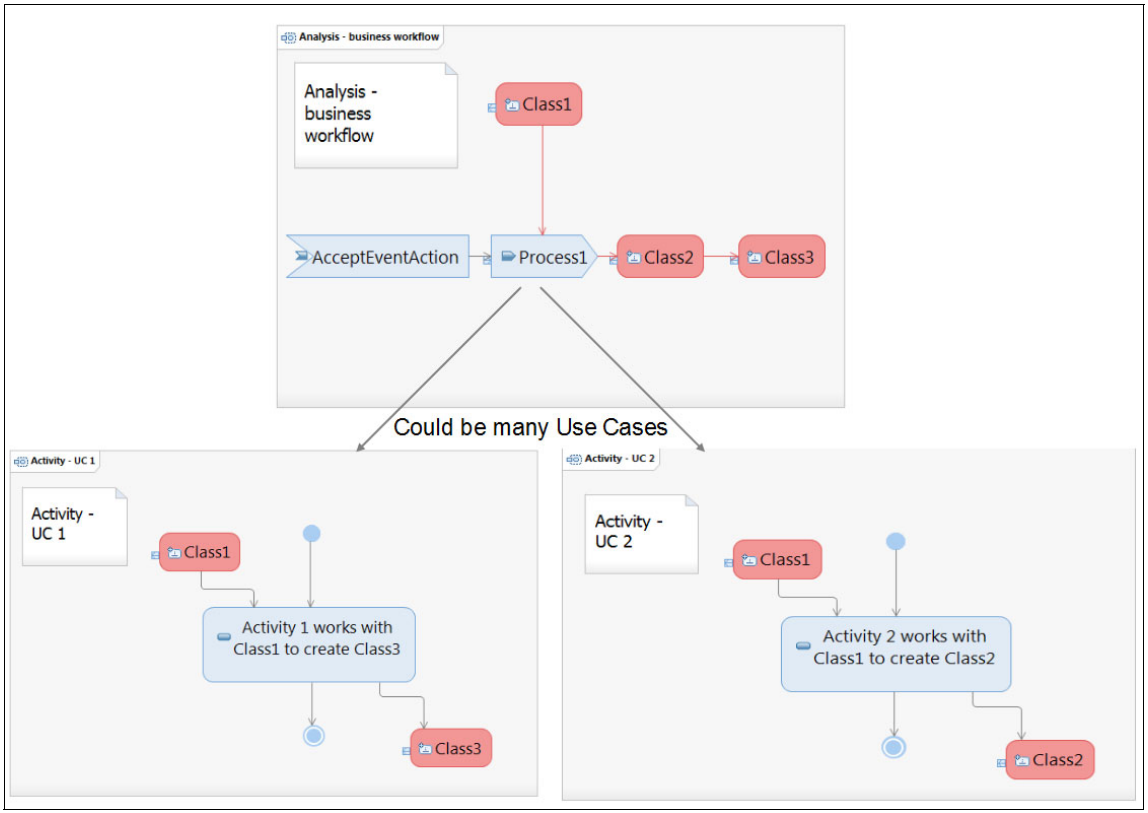

Figure 3-10 Sample of a simplified business process model

The diagrams at the bottom in turn represent a drill-in into the business process. As you can see, it invokes one or more use cases. These deal with the same red objects, again representing object instances from the domain model. The key concept that we have to keep in mind here is that it is the same objects that the use cases manipulate that the entire business process encompasses. There can be no domain objects manipulated by use cases that are not part of the business process, unless one use case returns an additional object, which gets modified by a subsequent object again but is not part of the response. The argument against this, however, is that all objects that are manipulated even through such a process should get returned by the process.

Use case model

The use case model defines the system behavior. We typically require the definition of use cases to understand the logical blocks that have to be dealt with through a front-end or attached system. They are also considered for the service design in an attempt to define the service provider signatures for any given service. Use cases are not the only input for service design. The services could provide exactly the function defined by the use case, but do not necessarily have to. We can define a service such that it only addresses a subset of the function described by the use case. We decide on that in the eyes of reusability. In that case we have to define additional services to fulfil the requirement of the use case to its full extent.

All you require is a single use case model for the new master data management system that you want to build. However, time allowing, you can create use case models for the existing important upstream and downstream systems as well. For example, suppose a front-end UI already exists and is adapted to work against the new master data management system instead of its old back-end. In that case it makes perfect sense to create a use case model for the old front-end behavior first. As soon as you have added the use case model for the new master data management system, you can compare the two use case models and either change the existing front end or create a new one. The benefit that you get from having both use case models available to you is that you know precisely what will have to be different compared with the old system. The difference is the result of the gap analysis between the two models that we have created.

An important aspect of the use case model is its dependency on the domain model. It comes from the primary goal of a use case, which is to manipulate data. This data is represented in the objects of the domain model. Consequently, we must describe the inputs and outputs of a use case as domain model object structures.

Figure 3-11 roughly outlines the relationship between the use case and the domain model. The left diagram represents a normal use case diagram where an actor invokes a use case. This could indicate that a button on a front-end application will be clicked, which in turn invokes any number of services that represent the use case.

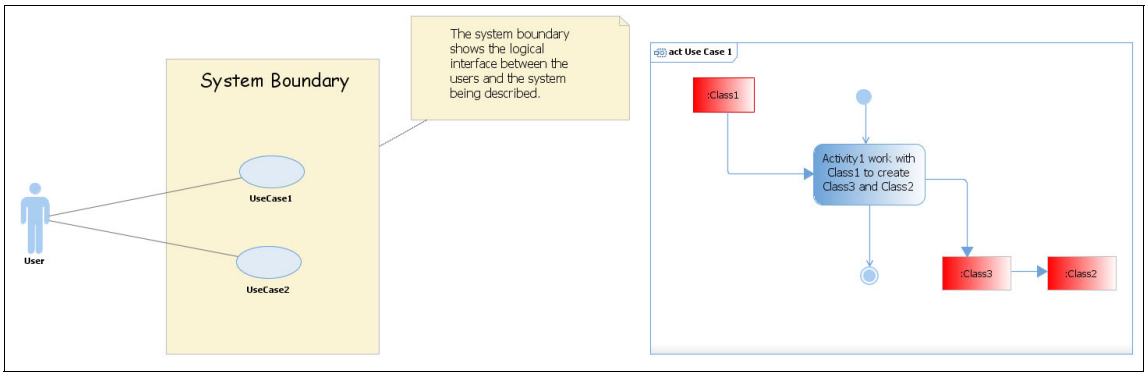

Figure 3-11 Sample of simplified use case model

The diagram on the right side on Figure 3-11 is a drill-in into one of the use cases. As you can see, the use case deals with the red objects representing object instances that are coming from the domain model.

Metamodel

By now you have gained an insight into the most relevant models related to the business aspect of our interpretation of DDD. We are now at a point that is equally sensitive to the business glossary model that we initially introduced you to, the *metamodel*. Similar to the glossary, it is also often forgotten, though the benefits are significant. Especially with large projects, you create more and more models and other design artifacts. Sometimes the number of models and design artifacts make it hard to keep track of the purpose of all of them. It thus makes perfect sense for us to describe them, including their relationships with each other in one additional model, the metamodel.

In addition to the description of the relationship between all models, this is also the perfect place to describe where and how the models are associated with the component model. This definition is important to describe the exact placement of the domain model in respect to the overall system architecture.

In respect to the completion cycle, the same is true as for the glossary. You will not be able to create a final metamodel at the beginning. Especially following given methodologies like this one, it is possible to lay out a draft about all models that you are going to use and how they are interconnected. But even then, it is virtually impossible to describe most key concepts at such early stage. Hence, this is a kind of model that you want to maintain over the lifetime of a project and make sure that it grows while you learn and complete your models.

Figure 3-12 is an example of how the models selected map to a metamodel. The domain model is clearly placed at the center of the metamodel. All other models are referencing or using it. Although it might look as though the business process model does not use it, it does. It does so on the grounds of the business processes manipulating the data that is defined through the domain model. There is no direct linkage drawn here, because the business process model references the use case model instead, which in turn is using the domain model. Hence, the business process model must also be based on the domain model.

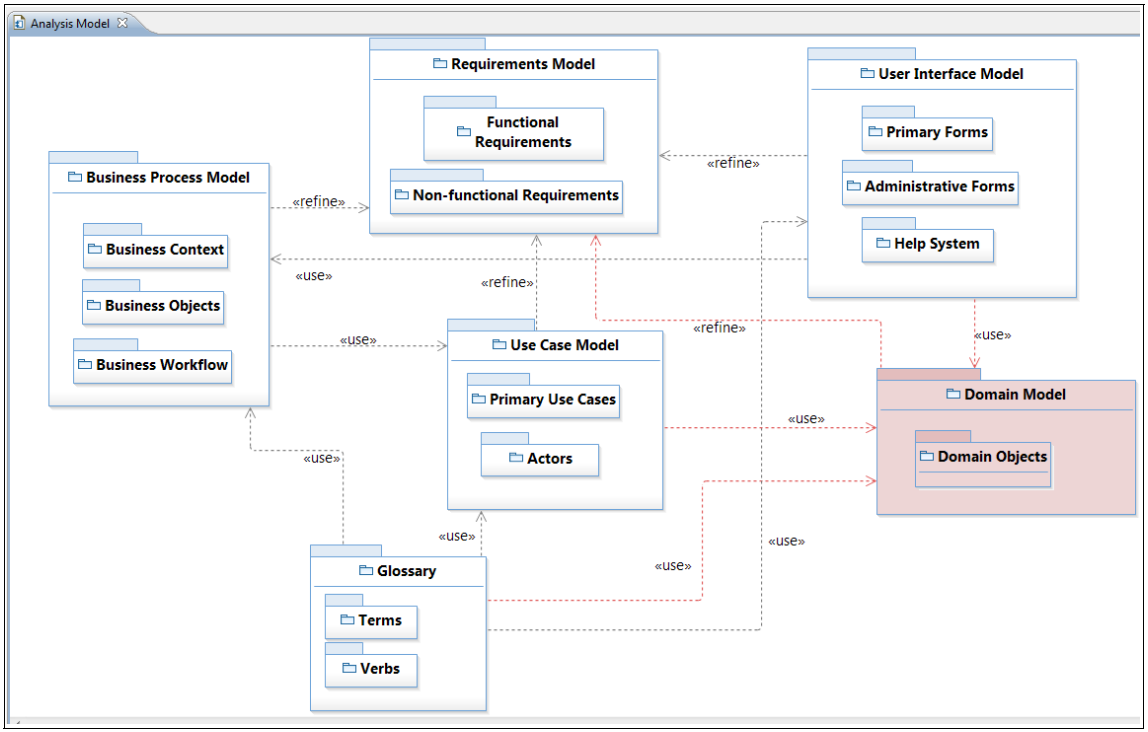

Figure 3-12 Sample of metamodel

3.3.2 Implementation models

The implementation models provide the technical details to the business view of the design. It is here that we add further information and enrich the business details with enough technical specification that we can commence with the technical implementation based on these models. They include additional definitions of the following:

- Component model
- Data models
- Domain model
- Service model
- Metamodel

Component model

The *component model* is another well-known model to any IT architect. Again, it is not the details of the component model that we want to elaborate on. Instead, we need to discuss its relevancy in respect to the placement of the domain model. The domain model at the core of this design approach needs to be associated with some component. It is vital for everyone involved to understand and relate to the precise location of the domain model. Only a clear understanding of the exact placement allows every team member to make judged and informed decisions about the impact of any domain-related changes enforced on the project teams, for example, through changes.

In addition to the placement of the domain model, we also need to be aware of the communication protocols between the various components. Components that are related to the one implementing the domain model will have to speak the language of the domain model or use transformation to translate into the domain language. Newly built components typically use a protocol that resembles the domain model.

Figure 3-13 highlights the location of the domain model within a high-level component model.

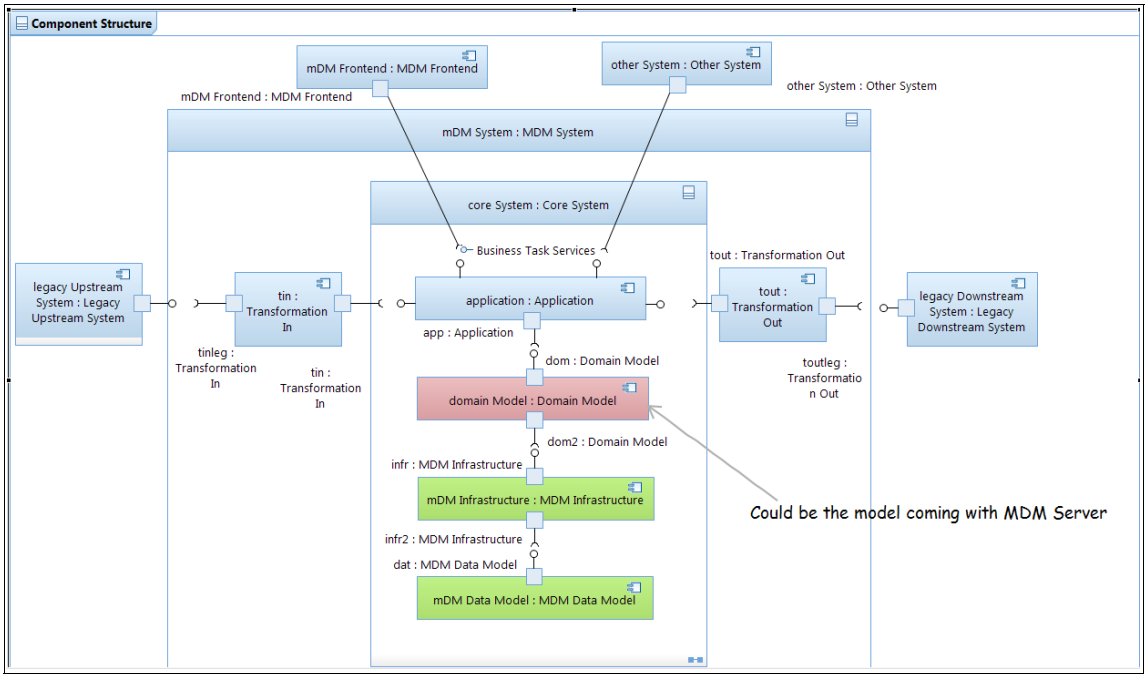

Figure 3-13 Sample of simplified component model

The component model in this diagram provides you with an abstract view and determination of the location for the domain model colored in red. If you adopt the data model from InfoSphere MDM Server as the domain model, then this component would already be provided by the product. It thus would cover, to some degree, a part of the system that we have to build. On the left side, a layering model has been defined in which the domain logic comes to rest between an infrastructure and an application layer. On the right side, the exact location in respect to the component model can be seen.

Data models

The physical *data model* represents the ultimate model data will be persisted in. Depending on the proposed solution, this will be a self-made model, custom fitted to the project requirements, or it will be the available model of a purchased product. In addition, there are other resources, such as proven industry patterns or other public or industry-specific resources. The major decision revolving around the data model should be whether we can and want to turn it into the domain model or whether we have to introduce an independent domain model.

The criteria for choosing one approach over the other is based on adoption. Faced with the choice of selecting a data model, we have to decide whether we are willing to adapt to a given data model or whether the data model needs to be adapted the requirements. Rarely can you find a data model that completely fits the business as it ships without modification. Many existing models are typically based on experience and best practice and are generic. Generic models, as a rule of thumb, are designed to be a good fit for many enterprises, but are unlikely to be an exact match. Self-made models, however, are an exact fit to the business. When you decide to start with a generic model, consider customization to make the model fit the enterprise or consider education about the new model so that everyone understands it and can follow discussions based on the terminology of that model. Neither is a simple task. Working based on a domain model established by the enterprise itself is more likely to serve as the common platform required for discussion regarding integration and other master data-related aspects.

Unfortunately, simply choosing a model does not resolve all open items immediately. At least two aspects require further analysis, naming and granularity of services. Every organization determines a terminology of its own, reflecting the way that it does business. It is highly unlikely that an external data model reflects this terminology. It uses similar structures at best and mostly is based on a different naming standard. It is difficult to deal with this mismatch. You might choose to adopt the terms from the data model as it ships or find a mapping between it and your own business terms. Both approaches have technical and business implications. Imagine the education effort that you have to invest in for the adoption of entirely new terms. That is lengthy and cumbersome. Similar to the question about the naming, you will have to decide on the complexity and granularity of the services operating on this data model. InfoSphere MDM Server, for example, provides fine-grained and coarse-grained services. Both serve their own purpose and make the product flexible. In respect to the integration with other components, however, the product recommends usage of its coarse-grained services. The reason for this is in performance. You will encounter less communication overhead by invoking one coarse-grained service capable of providing data into many entity-related objects over many invocations that you require when relying on the fine-grained services.

Irrespective of your decision about which model to adopt, always plan on consuming the coarse-grained services of InfoSphere MDM Server. In respect to the service mapping, this requires good insight into the product capabilities. It also requires a little more time determining the context that allows for the decision on the right coarse-grained service, as opposed to relatively quickly mapping onto the fine-grained services.

It is probably wise to adopt a custom domain model if your project is challenged by lack of adoption of a standard data model or your enterprise has already defined its own domain models. Acceptance throughout your own organization will significantly improve even if that is no guarantee that there will not be any objections. In other cases, you might want to opt for the adoption of an available data model without modification to save the time for the required building or customizing of a data model. In that case, at least get the standard naming added to the glossary for cross-reference purposes.

Because most models come with many entities that we do not require for our special needs, it is difficult to convey only the relevant ones to all teams involved. In such a case, it is a good idea to create a version of that data model that only shows the relevant detail and nothing more. This way you do not distract your team members and divert their attention to areas that bear no relevance to the actual implementation.

Domain model

As soon as the *domain model* has been defined, it will play a vital role on many other design artifacts. You will use instances of its objects in service design, as well as use case and business process modeling. You can attempt a service design based on the existing domain model alone. This would be referred to as a bottom-up approach. However, services should also be built against the use cases that describe the system behavior. This is the top-down approach in comparison.

Technical aspects should be clearly separated from business aspects. Still, we do require something that allows us to bridge the gap. We need to communicate between business and IT, and we use the domain model to do so. The difficulty that you will encounter is that you will probably struggle to get the balance between deep business insights and technically required details for the implementation right. Some of the technical aspects are:

- Groups of entities accessible through a given business key

  These groups describe the relationship of entities that belong logically together and how they relate to the entity holding the respective business key. This knowledge is required for read operations, as well as for initialization and the creation of specific instances, possibly in conjunction with the satisfaction of relevant business rules.

- Lists

  Your domain model will probably hold a number of objects that have a one-to-many relationship to some other object. This includes objects or attributes that serve information about business effectivity dates and are time sliced. These objects require the definition of lists to retrieve the entire sequence of information stored. Because this is not detail that is required on the business domain model, we have to add operations to support these tasks on the implementation level.

Service model

The *service model* can be created from two different starting points. Starting with the use cases, we follow a top-down approach. The resulting services are typically coarse-grained and generally fit the business logic that needs to be integrated into the application. This much fits the approach taken by InfoSphere MDM Server. It is the ultimate goal that you aim for to achieve the best performing design and implementation. However, designing use-case-specific services leaves little potential for reuse in other contexts. To make the service design more flexible, you can choose to follow a bottom-up design approach and create fine-grained services. On the bottom level, these designs are simple create, read, update, and delete services on the objects of the domain model.

Both approaches have their advantages and disadvantages. In the context of an SOA, there is a wide range of possibilities in between these two, fine-grained and coarse-grained. We recommend using system and business insight gained through the modeling of the domain to identify additional services.

Certain domain objects are frequently handled at the same time. These represent the value objects or value types. There is no need to define a fine-grained service for each value object. Instead, we can provide one service that allows the update of multiple objects, which at least encompass all value objects. This is referred to as the middle approach.

Irrespective of the approach that we are taking, we eventually end up with a set of services of different natures that include fine-grained create, read, update, and delete services, coarse-grained strictly use-case-based services, plus additional services somewhere in-between. By applying these three approaches we should end up with high-performance-oriented services where required and a high potential for reuse, while at the same time providing business-task-related services. It is important for us that they are highly reusable in other contexts as well. This is required due to the nature of master data management environments, where often many systems are attached and integrated with the single master data management system.

You need to make the same decisions independent of your choice using a data model product, such as InfoSphere MDM Server, or creating a domain model on top of the available model and persisting it. The problem is just somewhat more apparent when working with an off-the-shelf product. In the case of InfoSphere MDM Server, for example, an exhaustive number of services is already provided. Most of the services can be used in many different ways, which is largely due to the generic data model underneath and with the services also adopting such a generic approach. As such, they can be called similarly as initially described, similar to create, read, update, and delete services, which are referred to as *fine-grained service calls*.

Alternatively, they also cater to the manipulation of many different entities of their data model, which comes closer or fits exactly to the specific business needs. This is referred to coarse-grained service calls. Saying this, keep in mind the best practices that a product brings to the table. In case of InfoSphere MDM Server, this preferred practice includes using coarse-grained services where possible.

Another important aspect to the service design is the service provider signature. Service provider signatures should be clear and intuitive. Some things to consider are the following:

- Entity identification and entity references
- Directionality
- Signature granularity and ambiguity in domain model
- Call-by-reference and result handling
- Cardinalities and alternative input

In most scenarios, services have to identify an entity to modify and that can be identified through appropriate business keys or their respective technical keys. Which alternative is used depends on the orchestration and the consumer issuing the request. In any case, such reference objects should show clearly within a service provider signature. The object or object hierarchy to be modified should also clearly show. The result is a strongly typed service with its intentions clearly described. An example for a signature could therefore look as follows:

```
Service(RefObj1, RefObj2, ModObj)
```

In respect to the use of InfoSphere MDM Server, know that this product is following a slightly different approach from its signature. Due to its generic nature, the services are based on categorization, meaning that you provide type codes, and they are not as strongly typed.

Another important aspect is directionality. Consider creating a relationship between two parties. For all cases where this relationship lacks directionality it is completely sufficient to use a signature Service(PartyObj1 PartyObj2, `RelationshipObj`). However, the direction of a relationship frequently carries a certain semantic meaning. For example, a person is another person's father or that a person is another person's legal guardian. Directionality must be dealt with in the domain model by assigning unique names to association ends, and similarly in the service design where each signature must clearly reflect the role that an object will play within the service:

```
CreateLegalGuardian(GuardianPersonObj, PersonObj, RelationshipObj)
```

Designing a service that actively handles multiple entities from the domain model brings a different consideration. Should objects appear as part of the service signature or should they be hidden by the associations provided in the domain model? A `CreatePerson`-service, for example, might require data on the person as well as the person's name. Logically, the two entities `Person` and `PersonName` are associated in the domain model. This leaves us with two options for the service interface:

- We define an explicit signature that takes both objects as separate entities, for example, `CreatePerson(Person, PersonName)`.
- We create an implicit signature `CreatePerson(Person)` that can assume the name to be provided as part of the person entity.

The first alternative is easier to understand but can lead to broad signatures depending on the number of objects to be managed through that service. The second option allows for a later modification of the service, for example, by including an additional object without having to change the service provider interface itself.

Call-by-reference in conjunction with the handling of the result data sets can cause issues that can frequently show when services handle multiple modifiable data objects that are passed to the service. In our experience, many programming languages allow for by-reference calls and thereby provide a way to modify objects without clearly indicating this by means of a return type. Extensive usage of this programming pattern makes applications harder to maintain, as it is not clear whether a service will modify and object or leave it as is. As a rule of thumb, the service signature must clearly indicate whether a service possibly makes changes to an incoming object. This is the paradigm imposed through the definition of side-effect free services.

Services can allow lists of objects or optional elements as part of their signature. Lists of objects are required, for example, to be able to add all identification criteria that a person has, such as their SSN, their TIN, and their passport number, provided that they are all stored in the same entity. Optional elements apply to situations where an element can be present but cannot be guaranteed that it will be present. An example of this is the `PersonName` to a `Person` in an `UpdatePerson` service. In both cases the signature should clearly state how the system will behave depending on the provided data. For lists this includes cases such as no elements, one element, or many elements. For optional objects as part of the signature, we should define how the service treats the possibly missing object.

Layering

Service *layering* is an important aspect in large systems, which can benefit tremendously from a thorough definition of appropriate service layers. However, we should also take into account that layering and the resulting separation of concerns can also lead to a mixture of concerns. This could, for example, mean that business logic on an application layer and consistency rules on a domain layer can both work toward the same goal, only on different layers. In some cases layering can lead to a more complex project structure and thereby result in an increase in communication, increased design efforts, more analysis efforts, and more interfaces to design and implement.

There are also benefits that allow you to carry out detailed planning, which is the definition of separate teams that can work in parallel and have dedicated skills. Irrespective of your choice, you will find that our approach will work under most circumstances and greatly allows for a design and implementation across team boundaries.

Figure 3-14 shows another representation of the layering of the various services around the InfoSphere MDM Server database.

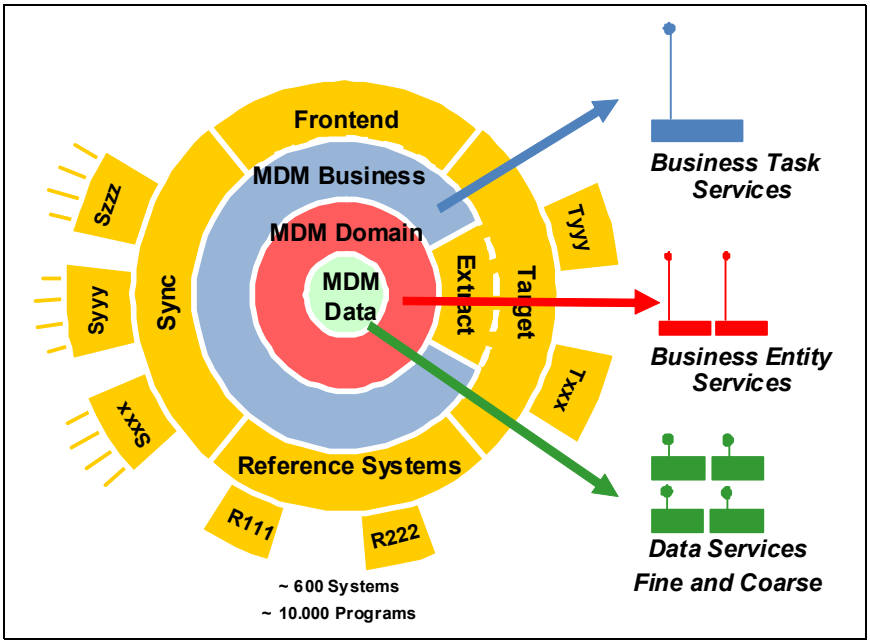

Figure 3-14 InfoSphere MDM Server layer

Figure 3-14 denotes the following distinct services layers, from the bottom up:

- Data services
- Business entity services
- Business task services

InfoSphere MDM Server itself provides two levels of services:

- The data services that are typically of a fine granular nature
- The coarse-grained data services, which allow modification of multiple objects with a single service execution

You have to define custom composite services to provide for your own business-oriented services. This capability is part of the product and is considered best practices in an attempt to provide the highest granularity in an attempt to reduce the network traversals when using an external business process engine that is invoking InfoSphere MDM Server services.

The *business entity services* have been explicitly designed and oriented along the business domain model. This complies with a bottom-up service design approach and fully satisfies the SOA principle of highly re-usable and business-oriented services. These services should acknowledge the existence of value objects for which you do not require fine-grained create, read, update, and delete services.

The *business task services* then represent the highest level of abstraction. They require the deepest analysis and are based on a mixture of bottom-up, top-down, and middle approaches. They are therefore even more coarse-grained than the entity services that aggregate value objects, but can still be somewhat more finely granular as the use case definitions dictate. The deviation from the use cases in terms of service design is acceptable considering an improvement in reusability.

Semantics

Consistent and proper *semantic meaning* is another core aspect of good service design. In an infrastructure without appropriate semantic definitions you will sometimes notice a high probability of uncontrolled growth of services and frequent errors. The reason for such uncontrolled growth is simple. The service designers are overwhelmed by figuring out how a certain service works, or whether it even exists. Instead of wasting time and effort, they can choose to design new services that fit the individual needs, possibly duplicating or at least nearly duplicating existing services. This is also an issue of missing SOA governance.

Similarly, if service signatures lack semantic meaning, there is no clear reference on how to use them. Designing composite services might turn out to be a cumbersome trial-and-error approach until a working combination of the existing services can be found and the sequence of service calls leads to the desired effect. Still, there is no guarantee that all possible uses have been considered and that there are no data constellations that have unwanted side-effects.

To determine the strategy for the semantic meaning to be used and applied to the service interfaces, take into account the following key factors:

- Number of services
- System architecture
- Performance
- Reusability
- Maintainability
- Extensibility

The number of services should not be a determining factor when you are defining the semantics of the service interfaces. However, sometimes it can be a good practice for you to limit the number of services that the system should provide. A system does not get any easier to use with the number of services growing into the hundreds or even beyond that. On the contrary, when it comes to extending such a system in the future it becomes difficult for anyone to determine whether a certain service that we require already exists and whether we can modify it for a different use or whether we have to create an entirely new service. Chances of proper and consistent service design over a multitude of designers increase when using SOA service registries and repositories against which service designs can perform a lookup. However, this requires you to put in place a proper SOA governance, including the required tools and processes.

The system architecture also impacts the service design to some degree. As we have previously mentioned, each business service is a mixture of low-level persistence services and more complex flow logic. Both types exist in any case, but the question is how they are distributed across the available components. For example, if the architecture provisions a business process execution environment to orchestrate service calls, then the system underneath can provide reusable and reasonably generic services. Here “reasonably” denotes the generics required to achieve the required degree of reusability. However, if such orchestration is not required or desired then the business task services have to be provisioned by another components. This could possibly be the master data management system itself or any other external component placed near the master data management system, keeping in mind the performance impact that could have.

Performance has another impact on our decision about the right service interface definition. Assume that we leave our service design with create, read, update, and delete style business task services only. If now the component responsible for the orchestration of these fine granular services, in respect to the domain model, is placed externally to the master data management system itself, you will have to prepare for degradation in performance due to the many traversals across the communication path. Consequently, always aim for the use of coarse-grained business task services in such an architectural setup. This again is a slightly different story as soon as you provision the orchestration within the master data management system itself. Still, if you define coarse-grained services at all levels of your service layering, you will get the best performance. This is the case because you thereby eliminate unnecessary communication. This is the design that you should aim for even if sometimes time constraints lead you to simply orchestrating fine-grained services because they are easier to identify and describe.

In most cases, we suggest considering the other three major factors for the definition of services:

- Reusability
- Maintainability
- Extensibility

Reusability describes the possibility to use existing services in different contexts, where the context can be anything from direct use within a business process or as part of more complex composite services. Maintainability addresses the need to analyze and fix defects or trace data through existing services. Extensibility measures whether a service is limited to a certain purpose or whether additional functionality can be added at a later time without completely reworking the service.

Metamodel

Analogous to the analysis metamodel, we also introduce a *metamodel* for the implementation models with the same purpose and benefits in mind. This model keeps all models required for the implementation together and provides the required guidance to our team members. Typically, you would start with a metamodel outlining all modeling-related artifacts. However, this will only develop over time and only be finished when everything else is there.

Part 2

Smarter modeling

The chapters in this part represent the practical guidelines for how to build a set of models that are interrelated and interconnected and that work together toward a single common goal. This part includes the following chapters:

- Chapter 4, “Smarter modeling in practice” on page 131

  This chapter brings together the information presented in the chapters in Part 1, “Master data management and model-driven development” on page 1. It acknowledges the variety of systems, data models, and their transformations and the mappings and service design. It also establishes one consistent and highly integrated model. The goal is to provide the foundational models that can be used for automated code generation at a later time.

- Chapter 5, “Sources” on page 245

  This chapter discusses InfoSphere MDM Server sources and provides the context for investigating InfoSphere MDM Server related sources. It provides an overview and a framework that was used for outlining the modeling steps presented in this book. It also discusses customer models in the appropriate modeling chapters.

- Chapter 6, “The modeling disciplines” on page 261.

  This chapter discusses the modeling disciplines. Everything that is done in this modeling methodology described in this chapter is based on a *metamodel*. That model describes the artifacts that are required and how they relate. This chapter also provides an overview of the verbal description of each of the business aspects, which is called the *glossary model*. While the domain model contains the visual representation of the business aspects dealt with, the glossary model provides the supporting information.

  It is also important to understand how to model with InfoSphere Data Architect and the modeling disciplines and lifecycle. These principles are outlined in this chapter to help you get started with modeling.

- Chapter 7, “The models” on page 281

  This chapter provides details about the models and, in particular, the categories into which these models fit. This information provides a good basis for understanding the functions of the particular models. This chapter presents the following model categories:

  – Enterprise models
  – Analysis models
  – Design models
  – Implementation models

- Chapter 8, “Mappings and transformations” on page 411

  This chapter discusses mappings and transformations. *Mappings* are one of the core deliverables of any project involving InfoSphere MDM Server. Because InfoSphere MDM Server ships with a data model that is ready to be used, this chapter describes activities related to the final deliverable, including reporting and accompanying activities.

  This chapter also discusses *transformations*, which are important but less visible elements in the modeling process.

- Chapter 9, “Deliverables” on page 461

  This chapter provides an overview of the types of final deliverables for a master data management solution. It includes information about reporting on the models and artifacts that are created through modeling and outlines the need for frequent validation of the models and their dependence on traceability information. In addition, this chapter positions the requirement for impact analysis and how the impact analysis depends on the traceability that is embedded into the models.

4

Smarter modeling in practice

This chapter brings together the information presented in the chapters in Part 1, “Master data management and model-driven development” on page 1. It acknowledges the variety of systems, data models and their transformations, and the mappings and service design. It also establishes one consistent and highly integrated model. The goal is to provide the foundational models that can be used for automated code generation at a later time.

However, due to this aspect being very technical, all code-generation-related tools and plug-ins required will not be described. Instead, there is a focus on establishing the basis on which such code generation tools operate. Therefore, the focus mostly is on the standard tools, rather than customized tools.

Following this approach, the emphasis is to maximize the possible reuse of the modeling approach. We acknowledge the fact that only the introduction of custom UML profiles allows us to follow the paradigm of model-driven development (MDD), including code generation. However, the development of such profiles or corresponding customized plug-ins is not described here, as it is beyond the scope of this book. For more information about this topic, refer to the materials listed in “Related publications” on page 559.

Nevertheless, the intent is to lay down the foundation and put everything in place within a UML model that allows introducing custom extensions that can harvest the standard model artifacts later to achieve code generation and therefore an end-to-end MDD environment. As part of this foundation, the introduction of appropriate stereotypes and other modeling conventions are based on which customization can take place. We provide you with all relevant details about customization and code generation, but to achieve a true MDD environment exceeds the scope of this book.

4.1 Modeling foundation

This section describes specific aspects of modeling and highlights the following questions:

- How are you modeling?
- What is the problem that you need to address?
- Why is it a problem?
- How are you solving this issue?
- Why are you solving it this way?
- What do you achieve with this solution?

Additional information: Although this book does not include a description of how to create the models that are required, it does provide an outline of how to model on a high level. For information about additional resources, refer to , “Related publications” on page 559.

When following standard modeling procedures in a non-customized tool, you might encounter issues that prevent you from reaching your overall goal. The type of issues that you encounter depends on the type of tool that you select, but regardless you will encounter issues that you need to solve.

This book focuses on the IBM Rational and InfoSphere toolset and, thus, describes a modeling solution that is based on the capabilities of these products. There are gaps that have been identified, which will be discussed to provide insight into why they have been determined to be gaps.

Important for us is the description of the modeling approach selected to overcome the deficiencies identified. In some cases you will notice the introduction of stereotypes to describe pieces of information that cannot be otherwise persisted. In other instances additional models are provided to describe the level of detail required. Remember that all solutions presented here are with the overall goal in mind, which is the generation of code at a later point in time. That is why there is a focus on the why and explaining what the achievement will be. Although not totally comprehensive, the most detailed modeling description possible within the scope of the book have been provided.

4.1.1 Modeling approach

This section describes a modeling approach that describes the model *categories* and *types*. It highlights the core parts of the overall model and defines their relationships. This foundation model explains the predominant model types and the roles that they play, and can be seen as the *meta-metamodel* for the methodology.

Terminology note: This discussion uses the term *asset*. Remember that a reusable asset for the models is not provided. However, you can build an asset and then reuse it as often as needed.

To build and apply the methodology efficiently, you need a common understanding of the content of the models. The high-level discussion here provides the initial insights about a possible structure for your projects.

Figure 4-1 illustrates the foundational meta-metamodel.

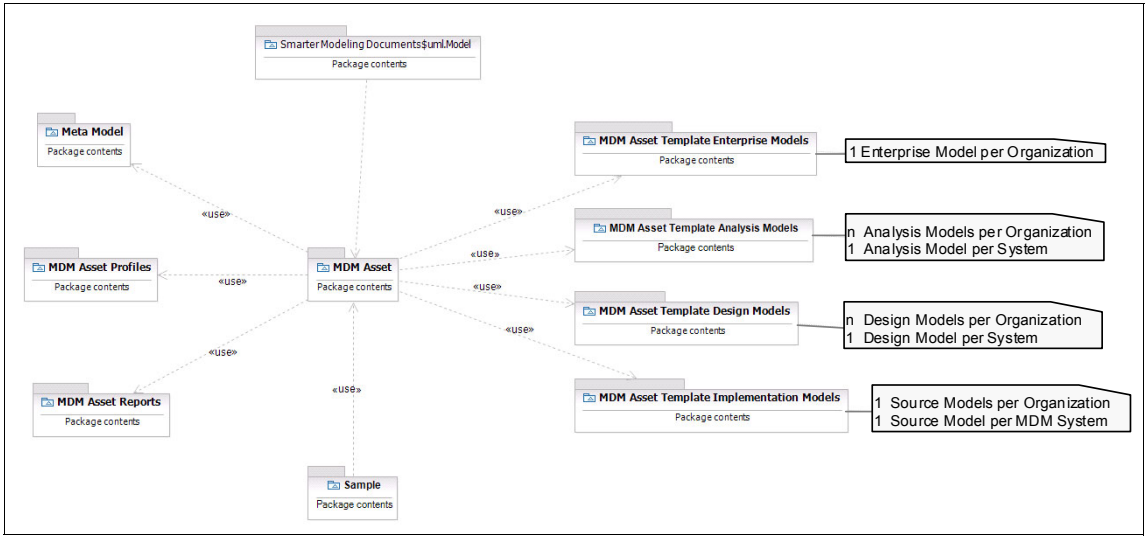

Figure 4-1 The foundational meta-metamodel

At the core of any project is the Master Data Management Asset (MDM Asset), which contains static core models and documents. Examples of such artifacts are the models that describe InfoSphere MDM Server. These do not change and as are thus considered static assets. They do contribute to the creation of the required custom models and deliverables, which are stored in the four main model types:

- Enterprise
- Analysis
- Design
- Implementation

As such, the asset needs to be built once and then it can be used to craft the custom models from that common starting point.

The *metamodel* is one of the most important core models. It describes all artifacts, the tools, and the projects, including their relationships with each other. While it might seem that the metamodel represents a static model that is part of the asset, its importance justifies a separate discussion as a means of highlighting it.

The MDM Asset is supported by a number of profiles that allow you to specify the stereotypes and tool-related extensions and customizations that are required to support the creation of the models as they are needed. MDM Asset Reports support the automated generation of asset-related information and provide reports against the custom models that are created as part of the project activities. These custom models are the basis for custom deliverables, which typically are provided in a generally available format, such as Microsoft Word documents and, thus, will be generated through reports also.

The area to the right of the MDM Asset describes the four project templates for the four major areas that are expected to participate in the overall modeling methodology. Similar to the thoughts on the MDM Asset, you do not need to start every project definition from the beginning each time. Instead, you can create a common model framework that contains all the reusable modeling artifacts that are common and generic to most projects.

For example, if you know that as an enterprise architect you are going to model a business activity model, an event list, and KPIs, as a few of many recurring definitions, then you can already provide for project structures showing exactly those. This in turn prevents anyone forgetting about them when working on the real implementation models. Those templates can also contain additional model artifacts, as far as they are reusable or serve as a sample, which speeds up adoption of the sample project. When it comes to a real implementation project you simply derive the implementation models from these templates and thereby have a starting point that is far beyond a blank sheet of paper. This in turn speeds up the project setup.

As shown in Figure 4-1 on page 134, the following main model types host all required models:

- MDM Asset Template Enterprise Models
- MDM Asset Template Analysis Models
- MDM Asset Template Design Models
- MDM Asset Template Implementation Models

Based on these template projects, you can generate the required project framework within the project solution workspace. The instantiation in general depends on the enterprise and the systems that are in the scope of the project through the following models:

- Enterprise models
  - One per enterprise
- Analysis models
  - Many per enterprise
  - One for each system from the system context

  - One for the master data management system to be built
- Design models
  - Many per enterprise
  - One for each system from the system context
  - One for the master data management system to be built
- Implementation models
  - Many per enterprise
  - One for each system from the system context
  - One for the master data management system to be built

The entire asset is enriched by a number of sample projects. While we cannot provide you with the sample, you can build up a sample over time. In this case we have built up all projects related to the case study that was previously outlined. This is closely related to the documentation provided because a sample always has a documenting character, yet is focused on a particular problem. We have created a sample project structure based on the case study and the pattern for project instantiation from their respective template projects, as illustrated in Figure 4-2.

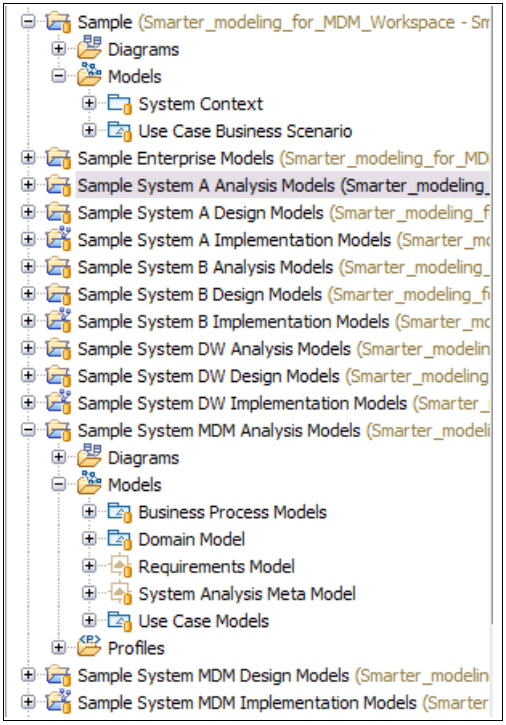

Figure 4-2 Sample project structure based on the case study

4.1.2 Repeating models

Although this discussion goes deeper into modeling-related aspects, some models do repeat. That is, artifacts appear multiple times in different projects within the entire project solution workspace. The scope for each modeling level is limited to the model details that are required, while ensuring that the model is enriched with the missing details at the appropriate lower modeling level.

Consider the following typical examples:

- *Use case modeling* includes only rudimentary diagrams that contain use case bubbles and their relationship to some actors. The remaining information is described in free-form text that accompanies the use cases.

  This approach, however, provides no benefit within an MDD, because you cannot directly operate on the text that is provided in the documents that describe the use case behavior. Because use cases provide the information that is required, such as input and output objects and the flows that they describe, a way of describing these aspects *in a model* rather than free-form text is required.

  In this case, you can enrich the original use case model from the analysis models and create a version in the design models that contains greater detail that you can incorporate into the service design on the same model level as the design models.

- The *domain model*, which is the most central model in a smarter modeling environment, also has multiple versions. The first version is a *business analysis model* that features only the business insights gathered and required to describe the static and behavioral aspects in relation to business concepts.

  You can create another version of the model that transforms the analysis domain model into the design domain model. You can enrich the model with further details, such as additional information about business keys, error handling, or time slicing. In some cases, you might create the required level of detail automatically during the transformation or add the appropriate level of detail manually.

Time slicing: Another important add-on to the original model is the concept of *time*. Do not add time in all its facets from the beginning on a business level, because doing so increases the complexity of the model unnecessarily. Although you can and should identify the attributes that require special handling of time, do not overwhelm the model with too much detail.

Consequently, the business analysis model marks time-sliced attributes and objects, but it also describes the relationships between entities excluding time. The implementation domain model then adds time where appropriate. You can implement this time slicing in a transformation step automatically with all associations, where objects that are marked as time sliced are checked and cardinalities are changed to reflect the **:n* nature of time-sliced behavior.

The benefit of time slicing is that everyone can understand whether an *n:m* relationship is due to business requirements or due to time, in cases where it transforms from an *n:1* to an *n:m* relationship cardinality. Another transformation then creates the third version of the domain model as a technical source model. This model adds information about purely technical aspects, including the concept of navigating a timeline through appropriate predecessor and successor concepts.

4.2 The metamodel

As we started out on this endeavor describing our thoughts on the methodology, everything was relatively easy to comprehend. Over time, more and more aspects were added and all contributed to describe the whole. A complete picture formed step by step. As we set out to describe our thoughts and vision, we learned that it was practically impossible to convey our message without building a model to describe our activities. Thus, the metamodel was born.

While it appears to be large, you will quickly learn to navigate it and use it as guidance for your own solution implementation project. As we moved on more and more towards a full MDD approach, the metamodel took another role in providing the required type definitions and constraints for the models being created and the transformations between them.

This section focuses exclusively on the metamodel for the artifacts. Other metamodels, such as the ones related to tooling, surface later, when we require that information in the sections dedicated to a particular concept.

4.2.1 Artifacts

The metamodel itself describes all artifacts related to the modeling of master data management solutions, as shown in Figure 4-3. It provides an overall perspective and hides the details of each section. In the following sections we zoom in on specific sections as they are being discussed so that they will be more easily viewed.

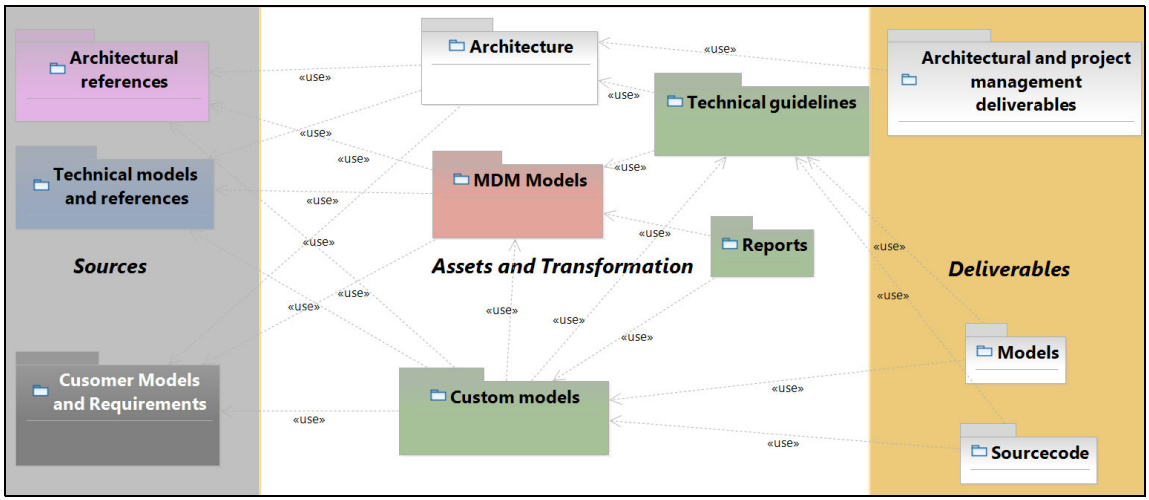

Figure 4-3 The metamodel

The required artifacts are grouped into the following major categories:

- Sources are shown on the left side.
- Deliverables are shown on the right side.
- Asset and transformation artifacts used or created by the modeling itself are gathered in the middle.

Sources represent both existing InfoSphere Master Data Management Server (referred to as *InfoSphere MDM Server*) models and documentation and customer data models, documentation, and other pieces of information that can be helpful when creating final deliverables. In addition, sources can also include IBM or other third-party reference models and any other resources that help in the delivery of the master data management solution. These sources are essential when creating final deliverables and are taken as given input into the custom solution modeling process.

The goal is to support the creation of the final deliverables for master data management solution projects. Typical deliverables include data mapping information between source data models and the master data management data model, as well as the mapping to the target data models. In addition, it is usually required to create custom composite service logic, which is described in appropriate service design documents. The creation of custom service code that resembles the composite service logic is out of the scope of this document. Other deliverables can also be derived or their creation supported by this asset, such as the definition of composite service design guidelines, documentation of architectural decisions, business glossary information, and draft project plans.

The key part is therefore taken by the modeling approach itself. It is here where the required transformation of source information is modeled to eventually create and generate the required project deliverables. The modeling approach itself requires two steps:

1. The import of any existing source information into the master data management solution model represented by the artifacts in green.
2. The creation of custom UML models, such as data mappings and service designs, as displayed by the artifacts in red. These custom UML models are exactly the models from which the final deliverables are being generated, ideally through usage of appropriate reports. We are referring to assets where we deem it possible to create a model that is reasonably static once and derive further variations from it during customization of the same. Transformations consequently refer to the models that have been created during such customization process.

Figure 4-4 serves as guidance through the metamodel.

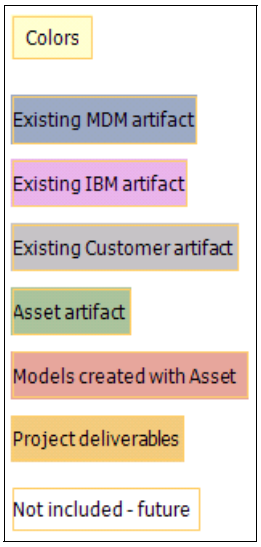

Figure 4-4 Metamodel color legend

In addition, the following stereotypes are in use:

- `<<document>>`

  Refers to documents of any kind and in any format, including PDF, PPT, DOC, XLS, and so on.

- `<<source>>`

  Denotes any existing artifact that is available for import.

- `<<library>>`

  Represents our asset-related artifacts that we only create once and then reuse in subsequent projects to transform other artifacts.

- `<<realization>>`

  Points towards all artifacts that have been created manually or through a transformation.

- `<<specification>>`

  Is the final deliverable as derived from the model, but also stands for non-model based sources.

The grey architectural artifacts on top are out of scope of this initial model and document but are left in place to demonstrate how such models can participate in data mapping, service design, and architecture stream and how they can support the project management and deployment teams.

The following sections describe all artifacts individually. Within each of the three major categories you will find further groupings to provide you with additional structure and to be able to quickly locate the artifacts that you are most interested in based on your line of work and responsibility within the project.

4.2.2 Sources

This discussion distinguishes between the following types of *sources*. The distinction is not based on the type of artifact, such as a model or a document, but rather the topic to which it is related.

- Customer sources
- Industry sources
- InfoSphere MDM Server sources

Customer sources

Customer sources are all directly related to the project that we are there to fulfil. They include models such as the physical and logical data and domain models, but also refer to written specifications that we cannot directly import or work with in our models. These specifications have to be converted into a model representation before being put to use, as depicted in Figure 4-5.

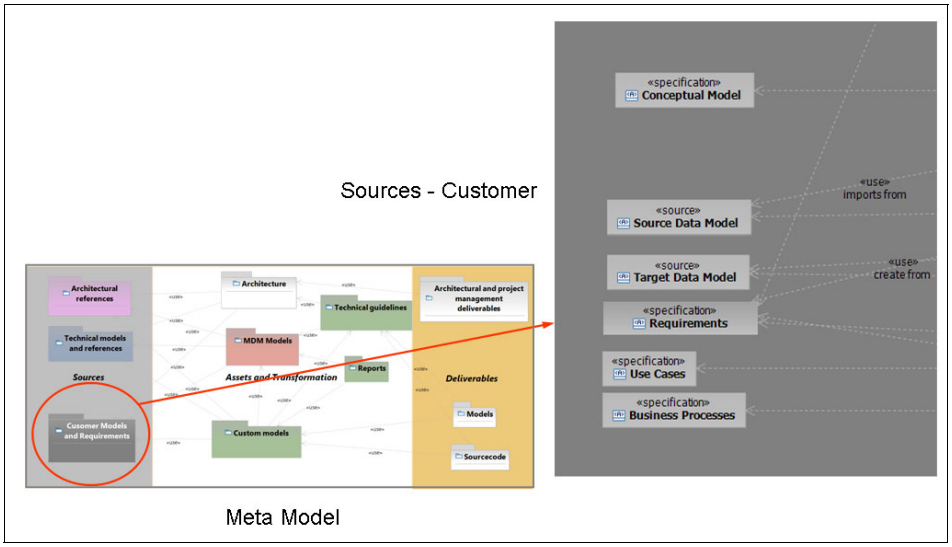

Figure 4-5 View of the metamodel with focus on customer sources

The following customer sources are required in this methodology:

- Business process
- Conceptual data model
- Requirements
- Source data model
- Target data model
- Use cases

Business processes are an important part of the business. They are a description how the business operates. Unfortunately, if multiple business processes following different rules operate on the same data, they are also the cause for data quality issues and inconsistencies in that data.

Therefore, one of the goals of any master data management initiative must be to standardize on a single set of processes. However, because the master data management system serves many other systems, you cannot embed the entire process, but only the portion that focuses on the data management side of it.

The solution design must include the business processes because you need a good understanding of the system behavior, in particular, an understanding of the upstream synchronization and the downstream extraction and loading of data.

The same will be true for our own system later, for which we should provide a system-related process model that describes in which order services have to be executed to achieve a certain goal.

The *conceptual enterprise information model* provides crucial insights into the larger picture of the information environment. You can learn about and understand the major entities and how they are related to each other. In the best case you can even trace each entity to the systems where these are hosted and thus get an idea of the scope of the project from an information point of view.

Requirements are the ultimate starting point for any project. They provide us with the reason why we are commencing with the project in the first place. Hence, they should exist prior to starting the project, even when the typical project refines them before initiating the solution design.

The requirements are provided by the customer and determine the parameters for the solution to be built. Requirements are separated into functional and non-functional requirements. They can be captured in a dedicated tool such as Rational Requisite Pro or simply written down in an Microsoft Word document or Excel spreadsheet.

Alternatively, they can also be captured in a UML model. Model-based requirements management, whether it being in a dedicated tool, such as Requisite Pro, or integrated into a UML model is advantageous because this allows for integrated traceability. Changing or dropping a requirement can be evaluated during impact analysis.

Both the customer's *source systems* and *target systems* provide at least a physical data model that resembles the ultimate model that data will be persisted in. Depending on the proposed solution, this will be a self-made model, custom fitted to the project requirements or it will be the available model without modification of a purchased product.

Some systems do not just have a physical data model but are also described on a slightly higher level through a logical data model or a domain model. If a logical data model exists, we import it too and create the corresponding model from it. If it does not exist, we can transform the physical data model directly into a logical data model abstraction.

Use cases are another important aspect in system design. They describe the functional behavior of the system. When integrating with other systems it is the preferred practice to describe or provide use cases for each system.

Their relevance for our methodology lies in the fact that they provide us with the insight that we require to come up with an appropriate service design.

Usually, use cases are high-level diagrams supported by much more detailed documents describing the flows required. To fully support a model-driven approach, ensure that the most relevant aspect of the use case design is included in the model itself.

Industry sources

In addition to customer-related sources, you can use any number of external and independent models and other pieces of information in a project to increase the likelihood for a successful project outcome. These sources are available in the form of ready-to-use libraries that can be imported directly into your set of models.

Figure 4-6 provides a view of the industry sources and their position in the metamodel.

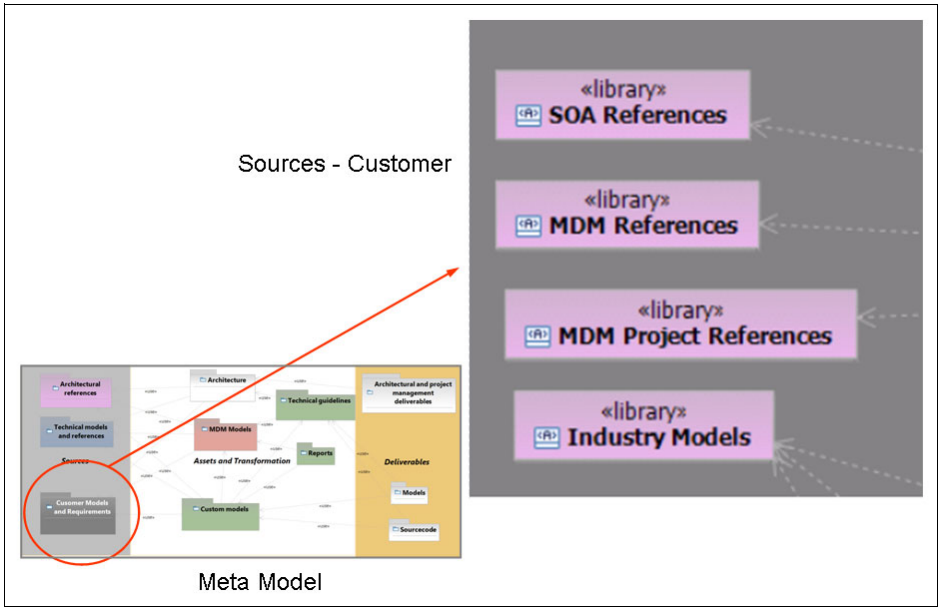

Figure 4-6 View of the metamodel with a focus on industry sources

These are the industry-specific sources:

- Industry models

  InfoSphere *industry models* are a wealth of knowledge for anyone who is responsible for designing industry-specific data models and the business processes and services to support these processes. Many aspects of the data model are also derived from these models. They provide good insight into the design and development of a custom domain model. However, these models also support the creation of the enterprise-wide conceptual data model and the physical data models.

  However, because this discussion focuses on InfoSphere MDM Server, which includes its own physical data model, you do not need that portion of the industry models for this purpose. Instead, you can use the industry models to derive a business process design without having to analyze all existing processes first or starting from scratch, which requires more time before reaching the point of having mature processes available.

- Master data management project references

  Project references are another great source of information for any solution implementation. Contracts permitting, it allows core principles to be harvested, derived, and adopted to other projects. This approach supports reuse instead of reinventing the wheel many times, risking running into the same issue over and over.

- Master data management reference architecture

  In addition to the SOA reference architecture, there is an information management reference architecture that includes blueprints for master data management. Such *reference architectures* are a good source for getting the placement of a master data management solution right within larger projects, and thus keeping them on the right track.

- SOA reference architecture

  IBM SOA experts have defined an *SOA reference architecture* based on experience gained from multiple projects in various industries over the past years. You can use the reference architecture to organize your own SOA design, development, and deployment to obtain better results and return on investment. The reference architecture, also referred to as the SOA solution stack, defines the layers, architectural building blocks, architectural and design decisions, patterns, options, and the separation of concerns that can help your enterprise better realize the value of SOA.

  Because InfoSphere MDM Server fits into an SOA, consider the placement of the product in the overall architecture based on this reference architecture.

InfoSphere MDM Server sources

The most important sources are related to InfoSphere MDM Server itself. These include documents as well as models that we can work with to describe the system behavior through our own models. Figure 4-7 provides a view of the InfoSphere MDM Server sources and their relative position in the metamodel.

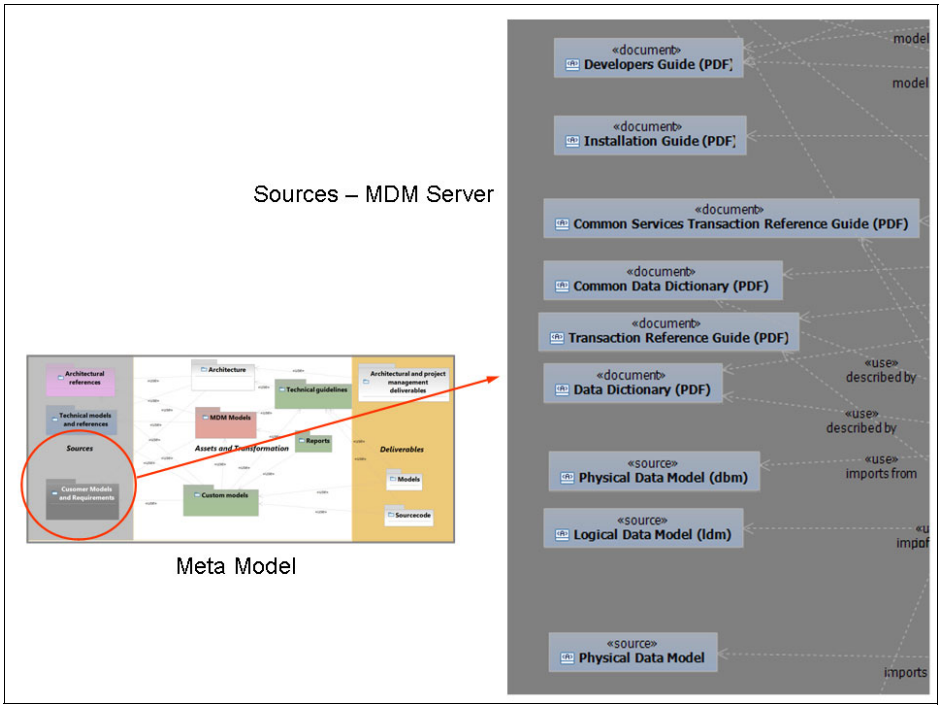

Figure 4-7 View of the metamodel with a focus on InfoSphere MDM Server sources

The following InfoSphere MDM Server related sources are provided:

- Installation guide
- Common data dictionary
- Common services transaction reference guide
- Data dictionary
- Developers guide
- Logical data model
- Physical data model
- Transaction reference guide

The *InfoSphere MDM Server Installation Guide* is provided with the product. It outlines possibilities for installing the product. It only acts as a base guideline because the exact installation scenario can only be determined by careful analysis of the functional and especially the non-functional requirements, which are unique to each customer environment. However, it can already be used to identify the most relevant building blocks as input for a custom operational model that we have to create during the course of a project.

The *InfoSphere MDM Server Common Data Dictionary* provides information about all common entities and attributes within InfoSphere MDM Server. It includes code tables, rules of visibility, access tokens, and specifications, among many others. It is one part of the overall InfoSphere MDM Server documentation describing the data model provided. It provides critical input to the data mapping design, as well as the glossary model.

The *InfoSphere MDM Server Common Services Transaction Reference Guide*, which is packaged with the InfoSphere MDM Server, provides information about all common service transactions within InfoSphere MDM Server. It includes management of code tables, rules of visibility, access tokens, and specifications, among others. It is one part of the overall InfoSphere MDM Server documentation describing the services provided. It provides critical input to the composite service design, as well as the glossary model.

The *InfoSphere MDM Server Data Dictionary* provides information about all entities and attributes within InfoSphere MDM Server. It includes management of party, product, and account domains. It is one part of the overall InfoSphere MDM Server documentation describing the data model provided. It provides critical input to the data mapping design, as well as the glossary model.

The *InfoSphere MDM Server Developers Guide* describes the core concepts and capabilities of the InfoSphere MDM Server and is thus a crucial document for any solution implementation.

The InfoSphere MDM Server logical data model is always the same. We can easily import it into our model by importing from the `.ldm` file provided in conjunction with the product.

Alternatively, we could also transform the physical data model into the logical data model. The transformation is simple, because most tables are represented on the logical data model exactly as they appear in the physical data model. However, because the `.ldm` file is provided, we advise using this exact file to avoid any unnecessary errors.

The InfoSphere MDM Server physical data model is the ultimate model that data will be persisted in. Depending on the proposed solution, this will be a self-made model, custom fitted to the project requirements, as can be implemented with the InfoSphere Master Information Hub, or it will be the available model of a purchased product.

In addition, there are other resources, such as proven industry patterns or other public or industry-specific resources. The major decision revolving around the data model should be whether it can and should be distilled into the domain model or whether an independent domain model should be introduced.

The physical data model represents the core piece of InfoSphere MDM Server. Although it is extensible, map all custom attributes against the existent objects and attributes that this model provides through a `.dbm` file.

The *InfoSphere MDM Server Transaction Reference Guide* provides information about all domain-specific service transactions within InfoSphere MDM Server. It includes management of party, product, and account domains.

It is one part of the overall InfoSphere MDM Server documentation describing the services provided. It provides critical input to the composite service design as well as the glossary model.

4.2.3 Asset and transformation

This largest part of the methodology includes the following major disciplines that are interrelated and individually discussed:

- Architecture
- Asset
- Project
- Transformation

Architecture

Architecture represents an important aspect in any system development. However, even though we believe that the decision for an MDD can support architecture too, there is probably much more use of architectural definitions in the model that can be picked up by business, designers, and developers alike. It is for this reason that you can find a number of architectural building blocks in our metamodel. However, we also acknowledge the fact that these activities are primarily of a supporting nature in respect to the analysis, design, and implementation of a master data management system, which is why we only mention a few of these aspects.

Figure 4-8 provides a view of the InfoSphere MDM Server architecture asset and transformation and their relative positions in
the metamodel.

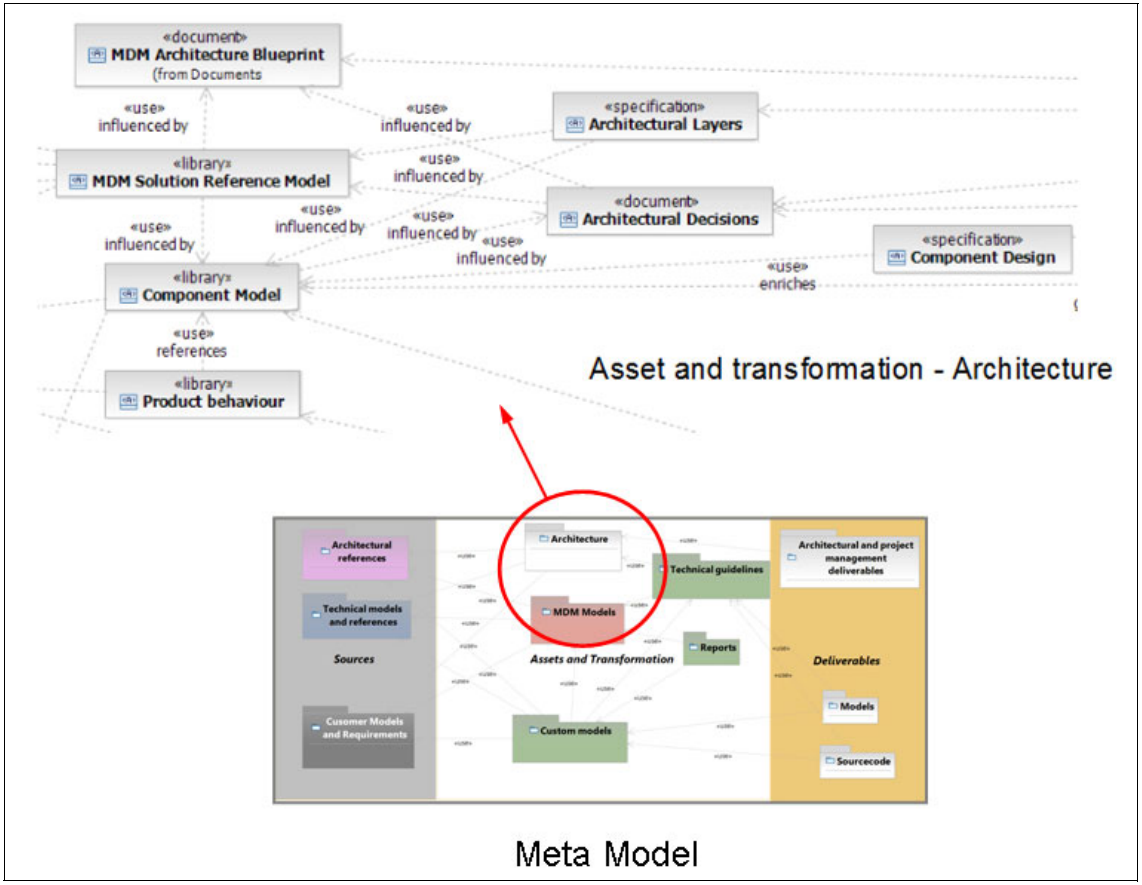

Figure 4-8 View of the metamodel with a focus on architecture asset and transformation

These are the main architectural building blocks:

- *Architectural decisions* are important, especially in large-scale projects that master data management projects typically are. Proper documentation of these decisions is therefore crucial to the success of the projects, and these templates can help you focus on the most important decisions. In addition to the requirements, you should trace components or other design elements back to the architectural decision that created them.

- *Architectural layers* are an important concept for structuring the solution in logical ways. It allows the definition and assignment of responsibilities to defined architectural layers and the components realized in those layers. In particular, in respect to service modeling, such a layering concept can help define a clear service layer while considering reuse and keeping the domain focus at the same time.
- After the functionality for each component has been defined within the component model, describe how you are going to realize those functional behaviors through appropriate *component designs*. This design is part of the overall technical architecture.
- The *component model* is equally well known to any IT architect. Again it is not the details of the component model that have to be elaborated on. Instead, its relevancy derives from the fact that the core of this design approach, the domain model, will have to be placed and associated with some component. It is vital for everyone involved to understand and relate to the precise location of the domain model.

  One important aspect related to the location of the domain model is to understand how the surrounding components communicate with the component that represents the domain model. Typically, such interaction happens through a protocol that resembles the domain model. In addition to the domain model, some other design artifacts will be the result of the project activities. It is good practice to associate these artifacts with their corresponding components to provide a complete picture of system development.
- The master data management *architectural blueprints* allow you to describe important recurring architectural patterns. They provide the foundation for a detailed technical solution architecture, which has to obey the patterns described in the blueprints.

  Base these blueprints on common industry experiences, where most patterns have already been encountered and some patterns have proven themselves better than others, depending on specific situations to be solved.
- The master data management *solution reference model* acts as a guideline regarding common architectural patterns that we apply during master data management solution implementations. It represents best practices regarding architectural decisions and thus represents the foundation for exactly that work product.

- Because every project has its own specific requirements that affect the *operational model*, you can only provide the building blocks required for a final operational model and models for a couple of common deployment scenarios, such as a single developer system and a high-scalability, high-availability environment where all components are distributed, clustered, and set up with failover capabilities.
- Sequence diagrams and other artifacts describe InfoSphere MDM Server *system behavior*, which is often required to provide information about the functionality provided by InfoSphere MDM Server.

Asset

The definition of an *asset* is most importantly aimed at the group of artifacts that are first of all master data management related and secondly of a highly reusable nature. The benefit of adopting a product such as InfoSphere MDM Server lies in it being both a master data management standard and highly reusable. Many artifacts are provided by the product and can be used. However, to provide their full potential, they often have to be customized. The customized artifacts are listed under the transformation section with the asset artifacts representing the unchanged models. Guidelines and reports are highly reusable, but not necessarily master data management related. We still expect most reports and guidelines to be reusable and adoptable to a large variety of projects.

Figure 4-9 provides a view of the metamodel, with a focus on assets.

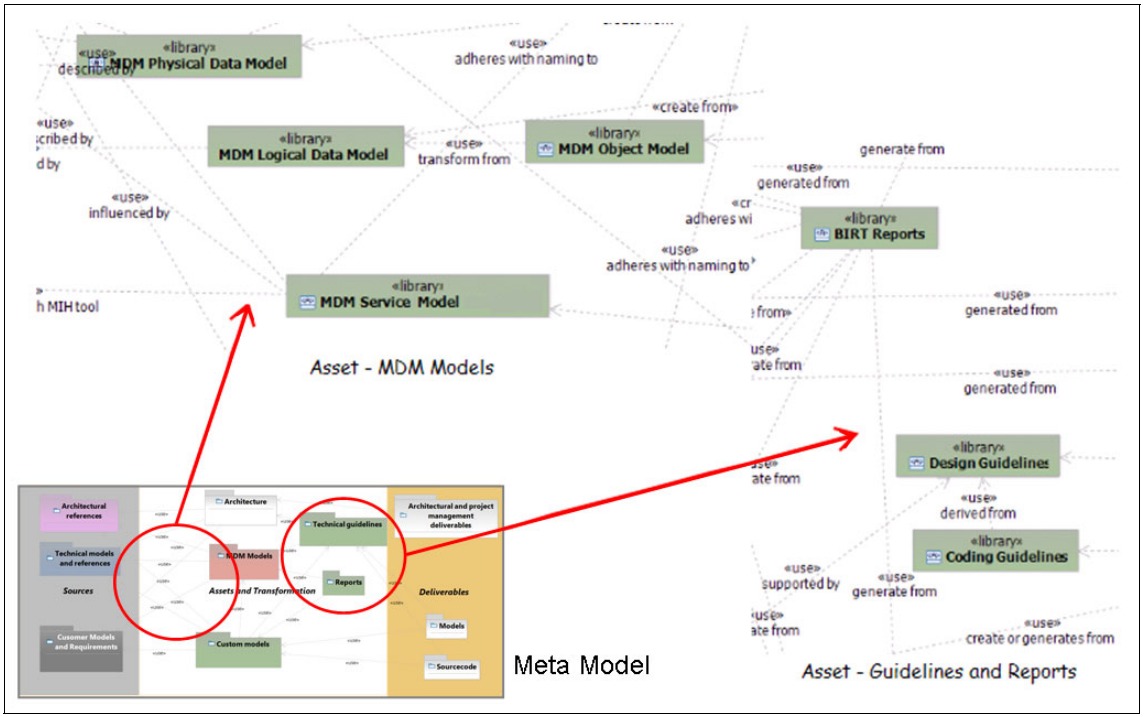

Figure 4-9 View of the metamodel with a focus on assets

These are the artifacts with an asset characteristic:

- Business Intelligence and Reporting Tools (BIRT) reports

  BIRT reports allow standardized documentation to be dynamically created from the models and thus provide the textual representation of our custom deliverables in the expected format. We expect to build up an asset that holds the most commonly used BIRT reports as a repository. BIRT is based on standard Eclipse capabilities, and because the Rational platform is built on Eclipse, this capability based on BIRT reporting is also usable for us.

- Coding guidelines

  Coding guidelines increase readability and maintainability of the source code created. Although you could argue over its importance in a model-driven environment, we deem these to still be an important aspect to the overall goal, creating readable, maintainable source code. Also, the generated source code should adhere to the coding guidelines defined to make it easier on human beings to read and interpret even the generated code. This is even more important during the phase where the code generators are created and bugs still need to be analyzed, identified, and fixed.

- Design guidelines

  Design guidelines are required to provide for a consistent design across all team members and over the lifetime of the project. If not adhered to, maintainability and extensibility can be seriously impacted. Additionally, we need to ensure and enforce a strict design, because our code generation depends on it.

- InfoSphere MDM Server logical data model

  The InfoSphere MDM Server *logical data model* is almost a 1:1 representation of the physical data model, yet it is required to transform to a representative UML model and define the service operations against it. It is provided through the appropriate `.ldm` file.

- MDM object model

  The *MDM object model* can be imported through MIH-related tooling and used within the service modeling. Services operate on the internal object model, which is referred to in this document as *service designs*. You can import the model as it is shipped and store it as an asset. Before using it in our service design, you will have to add the extensions and additions as defined for the data models, which leads us to the customer master data management object model.

- MDM physical data model

  The InfoSphere MDM Server *physical data model* is always the same. We can easily import it into our model by importing from the `.pdm` file provided in conjunction with the product. Alternatively, we could also reverse engineer the physical data model by importing the DDL that we use to set up the database during installation. However, because the `.pdm` file is provided, we advise using exactly this file to avoid any unnecessary errors.

- MDM service operation model

  To be able to carry out a model-oriented service design, you first must include all existing services in the model. This represents somewhat of a challenge because there is no such service model available and delivered with the product. However, we are using a tool provided for the InfoSphere Master Information Hub that also allows for reverse engineering of the existing object and service model. This in turn provides us with the ability to run it once and then store the model so that we can work against it from other models that require such a *service operation model* as a basis, for example, as required by the composite service design model.

Project

There are many more project-related artifacts that we have not listed because our focus is on the analysis, design, and implementation of a master data management system. However, we do recognize the importance of a supporting project management and artifacts that are related to that discipline. Section 10.3, “Setting up and executing a project” on page 511 supports this statement. See the only artifact listed here as a representative for many others. As an example, we expect you to generate tracking and reporting related information directly from the models instead of querying them from team leads and manually adding the queried data into Excel spreadsheets.

Figure 4-10 provides a view of the metamodel with a focus on project assets and transformation.

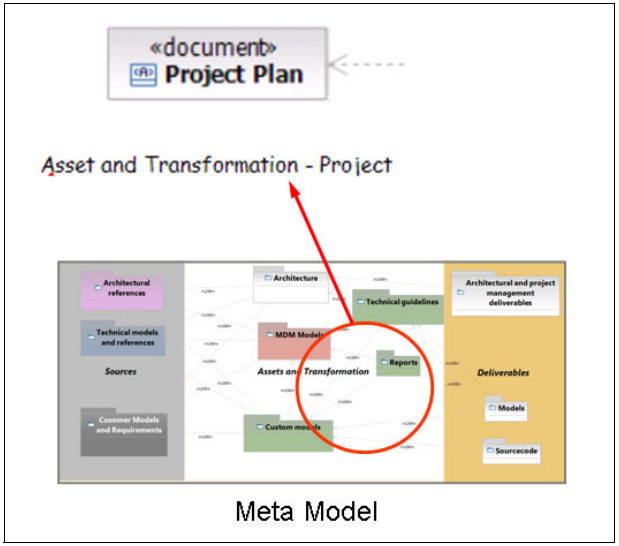

Figure 4-10 View of the metamodel with a focus on project assets and transformation

The single representative for project-related artifacts is a project plan. The project plan provides one of the key assets for a project manager. Because most master data management projects are following similar steps, some high-level activities can always be derived from a template project plan and adapted to the custom plan.

Transformation

These artifacts are the intermediates in many ways even though in many cases they stand for the target model as we require it. However, based on the fact that in most cases we still require the final deliverables, these artifacts are merely the input into the generation process for those final deliverables. Ideally though, those deliverables should not contain any content that we have not captured in any of the models mentioned here. Also, often we do not just create these models from scratch. Often they are derived from other artifacts. These can be either source models or asset models. They can be created manually or through appropriate model transformations.

Figure 4-11 provides a view of the metamodel with a focus on transformations.

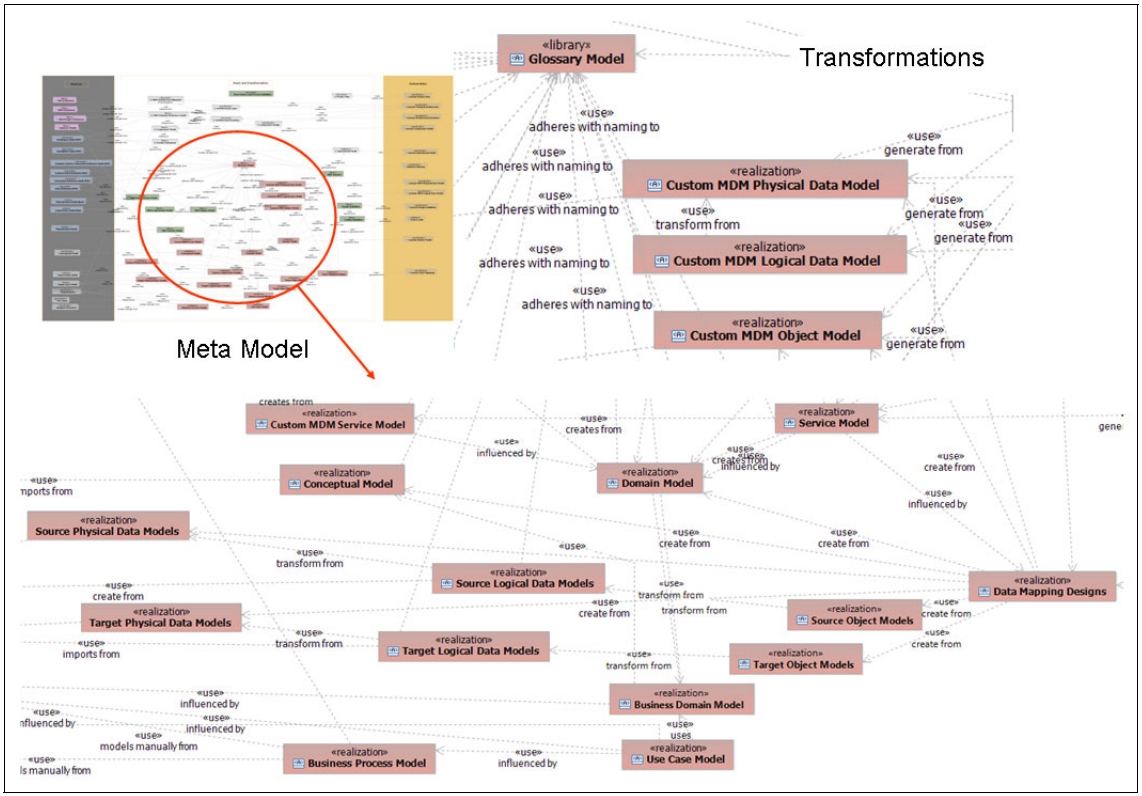

Figure 4-11 View of the metamodel with a focus on transformations

These are the transformation-related artifacts:

- Business domain model

  The *business domain model* is the centerpiece of all design and implementation activities. Due to its nature of being closely related to the business, the domain model can be seen as a visual extension of the business glossary. It puts any number of verbs and terms into context through visual associations, which are then clearly described. It is the only model always and under all circumstances being referred to when discussing logic, enhancements to existing logic, changes, and so forth. It is represented as a UML class model.

- Business process model

  You need to build the *business process model* manually, based on information obtained from the business, unless there is a model already existent in a format that you can import easily into your tools. You also need an understanding of the dependencies between use cases and how the system in question is expected to behave under certain conditions. The business process model has a close relationship to the domain model and the use case model. Thus, you need to ensure that these models are consistent and complete.

- Custom logical data model

  The InfoSphere MDM Server *custom logical data model* is predominantly derived from the master data management logical data model. In addition to the standard entities, it can provide additional entities that function as extensions or additions to that model. It also can exclude some entities in an attempt to optimize the data model. However, because we add customization by means of InfoSphere MDM Server Workbench, we do not directly create from the master data management logical data model, but instead transform from the custom master data management physical data model.

- Custom object model

  The InfoSphere MDM Server *custom object model* is required to enrich the standard InfoSphere MDM Server object model as necessary to cover the requirements captured in the business domain model. We derive information from the base InfoSphere MDM Server object model, and we can extend its abilities by using new objects (extensions and additions to the data models). InfoSphere MDM Server Workbench provides a means of customizing the master data management object model. Late we discuss how we integrate the workbench into our modeling approach.

- Custom service model

  Similar to the custom object model, we also add service operations to access the data of our newly built extensions and additions. Together with more complex services used, for example, to meet non-functional requirements, they make up the *custom service model*. Again, we derive most information from the service model as it ships but extend it with the custom data services that come with the data objects.

- Custom physical data model

  Similar to the physical data model, the *custom physical data model* represents the exact physical data model as it applies to your project environment, including all extensions and additions required.

  Consider the extensions and additions for all aspects related to the data model that InfoSphere MDM Server does not provide as it is shipped. As such, this model is a counterpart to the InfoSphere MDM Server custom object model.

- Data mapping designs

  The data mappings in the data mapping design are essential to be able to track the information flow between disparate systems. The overall mapping is not as easy as it appears. We have identified at least three different ways of mappings including static, dynamic, and something we call service-flow-specific mapping, considering variations in mappings depending on the service operation that is invoking our objects.

  We only apply data mappings on the following levels:

  - Between physical data models for static mappings
  - Between object models for dynamic mappings

- Business glossary model

  The *business glossary* is a critical model to consider. For example, a new member to the project team might have experience with a similar project with a different customer. Although this person might assume that terminology is consistent with the two project, more often than not, such assumption causes miscommunication, where terminology is mismatched for some time. Thus, the business glossary is key throughout the entire project and system lifecycle for communication between business and IT people, as well as designers and architects and any developers. In addition, naming conventions for the implementation should also consider names from the glossary to ensure meaningful implementation.

Start the business glossary as early in the project phase as possible, but remember that you cannot create the final version right away. Many verbs and terms only find their way into the glossary when someone asks for a certain meaning. So based on this, a glossary will grow gradually throughout the phases of the project.

Such a glossary should also extend across systems, bridging the gaps and possibly exploring different meanings of the same word for each system. Most importantly, the glossary acts as the enabler and provides the basis for the domain model itself.

- Source and target logical data models

  Apart from the custom logical data model, *logical data models* also exist for each of the *source* and *target* systems. The logical data models are the binding glue between conceptual and physical data models. They contain much more detail than the physical data models by focusing on the business concepts that they address, similar to what the business domain model does, which can also be seen as a logical data model. However, it does not contain as much logic related to the physical data persistence as described in the physical data model.

  We use logical models as a hop over, which can trace information from the object model all the way into the physical data model. Due to its descriptive nature, it also allows us to gain a deeper insight into the business domain defined through this model. It is also used to build the static data mapping between different data sources.

- Source and target object models

  We previously used the *custom object model* to add custom objects to the object model and thereby cover the static data requirements within InfoSphere MDM Server. To establish dynamic mappings on the object-level, as previously described for the data mapping design, we also need such models for the *source* and *target* systems. If they do not exist, you can derive them as a UML representation of the respective logical data models.

- Source and target physical data models

  The *physical data models* of the source and target systems that we have to integrate with should always be available. In case they are missing but the system implementation already exists, we can easily import the physical model into our model by connecting against the database, importing some physical model diagrams into the tool, or alternatively by reverse engineering it from the DDLs used to set up the database during installation. If the respective system, in that case likely to be a target system, does not yet exist, we can engineer a physical data model from scratch.

We require the models to build the data mapping describing the flows between the physical data sources. In addition, they can be used as a transformation source to derive the logical data model, should none exist for the system in question.

- Service model

  The *service model* is based on a variety of considerations. The complex business flows are derived from the provided use cases, but also consider more fine granular service operations based on knowledge gained through the domain model in an attempt to increase reusability. Within an SOA, we could also define create, read, update, and delete services, which are then orchestrated elsewhere.

  A definition based on a use case is a top-down approach, whereas the definition based on domain model aspects is a bottom-up approach. Both approaches are equally relevant, and keeping both approaches in mind will create the best possible service model that considers both performance and reusability among the key aspects. In addition to these approaches, apply your knowledge about the system to be built as gained over time, and apply a third approach, the middle approach, which usually leads to services that are not as fine grained as the create, read, update, and delete services from the bottom-up approach, but are also not as coarse grained as the use-case-based services from the top-down approach.

- Use case model

  The *use case model* needs to be built manually based on information obtained from the business. Usually it is written in Word documents and contains purely descriptive information. However, for us to achieve the goal of an integrated, consistent model, we require the use cases in the model, including all the information otherwise only available in written language.

  You need a deep understanding of the functions that each use case has to carry out and how the system in question should behave under certain conditions. The use case model has a close relationship with the domain model and the business process model. Thus, ensure that these models are consistent and complete.

Note: This discussion separates the use case model into the following interlinked components:

- The actual use cases
- The use case analysis

The use case analysis is a more formal representation of the use cases using UML activity diagrams, which are discussed later.

4.2.4 Deliverables

The goal of any solution implementation project is the final deliverables. Many deliverables in our types of projects are merely of a documentary nature. You will not find all models or documents describing all models in this paragraph. This is because some models are merely used to create other required deliverables. For example, we have not listed the domain model because we see it as an input or a transformation model. It is required to define a data mapping and the corresponding InfoSphere MDM Server data models that are able to store all attributes contained in that domain model. In addition, we also have to deal with the creation of source code as deliverables too. We categorize our deliverables into the following major groups:

- Architecture
- Data mapping
- Data model
- Service

Architecture

Figure 4-12 provides a view of the metamodel with a focus on architecture deliverables.

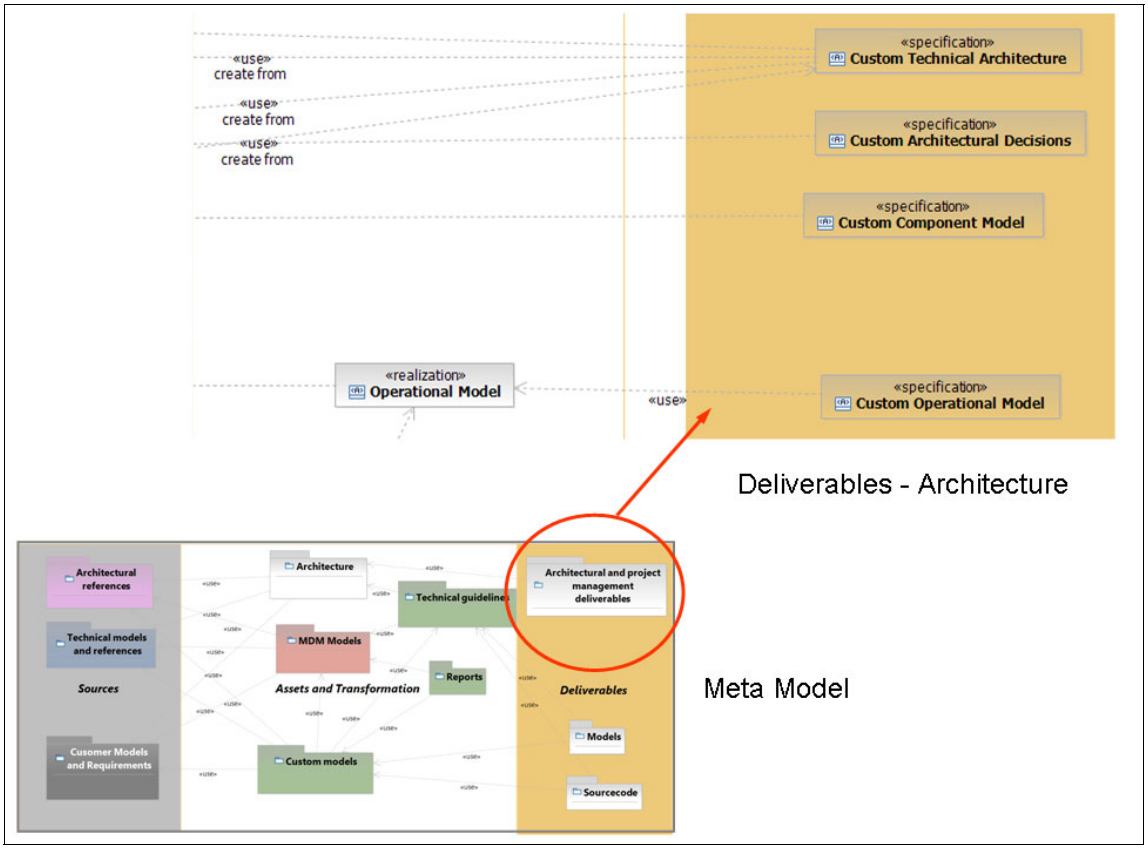

Figure 4-12 View of the metamodel with a focus on architecture deliverables

We are aware that many more architectural artifacts exist and have to be produced during a project from a master data management implementation perspective. The following architecture deliverables are key:

- Custom architectural decisions

  Architectural decisions build the foundation for any project. They document why the components have been laid out the way that they are in adherence to the functional and non-functional requirements. It is vital to have a single repository with proper governance, and as such it is a good option to maintain these decisions within the context of a model too. Describing these decisions within a model allows implicit cross-referencing from other parts of the model.

- Custom component model

  The *component model* is a well-known model to any IT architect. Again it is not the details of the component model that have to be elaborated on. Instead, its relevancy derives from the fact that the core of this design approach, the domain model, will have to be placed and associated with some component. It is vital for everyone involved to understand and relate to the precise location of the domain model.

  Another important aspect related to the location of the domain model is to understand how the surrounding components communicate with the component that represents the domain model. Typically, those interactions happen through a protocol that resembles the domain model.

  In addition to the domain model, some other design artifacts will be the result of the project activities. It is a good practice to associate these artifacts with their corresponding components to provide a complete picture of the system development.

- Custom operational model

  The *custom operational model* is required to be able to deploy all components of the solution built within the constraints of the project. Because every project has its own specific requirements that affect the operational model, we can only provide the building blocks required for a final operational model, as well as models for a couple of common deployment scenarios, such as a single developer machine and a high-scalability, high-availability environment where all components are distributed, clustered, and set up with failover capabilities. The custom operational model defines the building blocks that you use in your environment.

- Custom technical architecture

  The *custom technical architecture* defines how to apply architectural patterns and practices to a specific project. It depends on the overall enterprise architecture and on project-specific requirements.

Data mapping

Data mappings represent one of the core activities in many data-driven solution projects. This applies even more to a master data management type of projects. It is here that we are interested in the mapping between objects and attributes on a number of different levels. First, we need to be aware of the mappings for reasons of transformation during initial or delta loads into the system. Complex system implementation styles sometimes require a form of continuous synchronization stream. Here we often have to undergo transformation too. In addition to all that, we are more and more often required to provide data lineage information.

On a system design level this is often provided by means of data-mapping specifications. A solution that can cater to the provisioning of such data on both design and run time certainly provides benefits to an organization.

Figure 4-13 provides a view of the metamodel with a focus on data mapping deliverables.

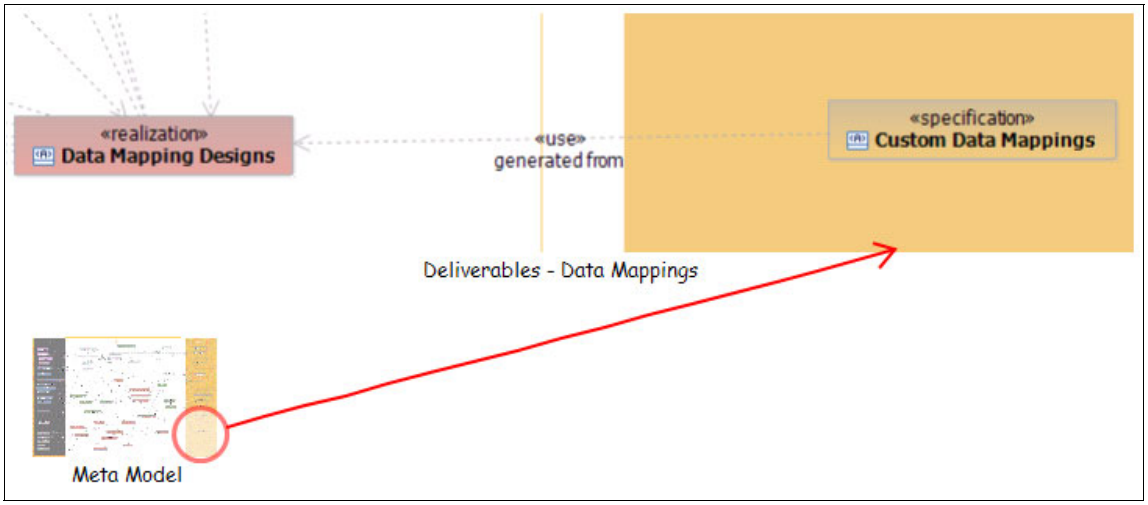

Figure 4-13 View of the metamodel with a focus on data mapping deliverables

The artifact representing this is custom data mappings. Custom data mappings are essential to be able to track the information flow between disparate systems. This represents the documentation generated from the mapping model.

Data model

Even though InfoSphere MDM Server brings a complete, robust, and ready-to-use data model into our projects, data modeling is still one of our core tasks. Only seldom can we reuse attributes from the predefined master data management data model exclusively, and often we are required to add a few attributes that we cannot map. The result is a customized version of the available data models. For these we have to provide appropriate documentation again.

Figure 4-14 provides a view of the metamodel with a focus on data model deliverables.

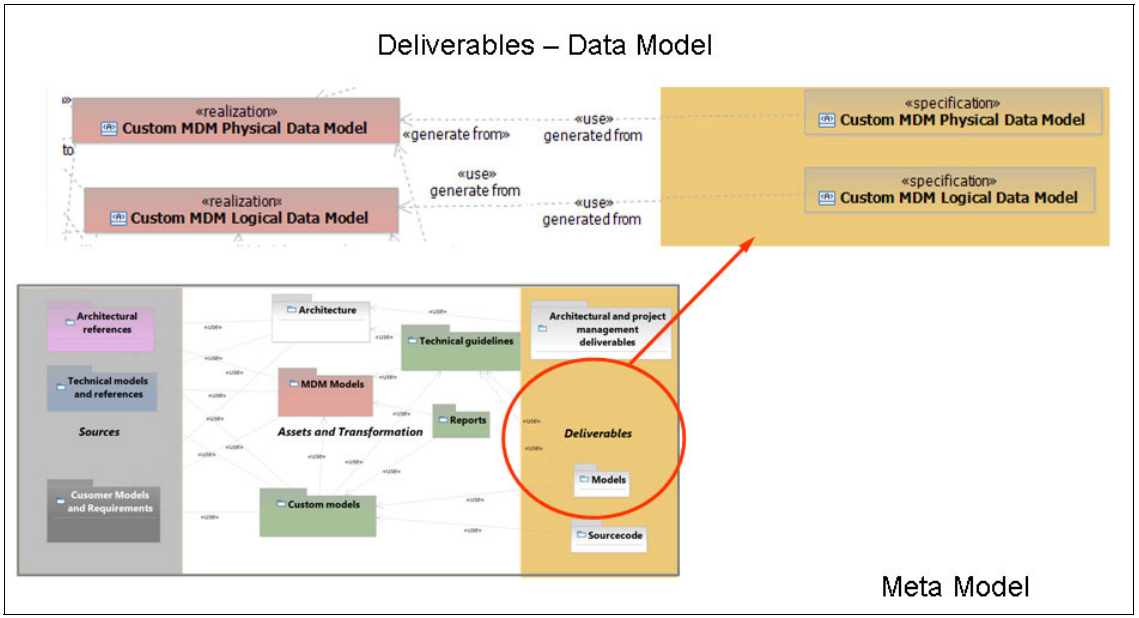

Figure 4-14 View of the metamodel with a focus on data model deliverables

The following data models are delivered:

- Custom MDM logical data model
- Custom MDM physical data model

The custom logical data is merely a by-product. It bears no significant relevance. However, because InfoSphere MDM Server comes with a logical data model, we keep maintaining it and have it reflect our customization. The reason for its insignificance is mostly in the 1:1 abstraction from the physical data model. This in turn means that it does not deviate at all, and thus we can rely on the physical data model for most of our activities. We do require a UML-based object model representing our data structures, and one of the ways of getting to it is through transformations from the physical to the logical and to the object model.

The custom physical data model represents the ultimate data model. It lists all extensions and additions that we have defined to store even those attributes that cannot be mapped to any of the attributes from the default data model.

Service

Complex master data management implementation style solutions are predestined to participate in an SOA. They have to integrate with any number of systems, and because we do not know which system will get added or removed at a later point in time, we must decide on loose coupling through services. However, the same as with the customizations that we require for the data models, InfoSphere MDM Server can provide only the services that work against the standard entities. As soon as we add a new entity, we are also required to add new services. Furthermore, we sometimes want to add more of your own business logic into the services. In such a case, we are also required to define and build new custom services.

Figure 4-15 provides a view of the metamodel with a focus on service deliverables.

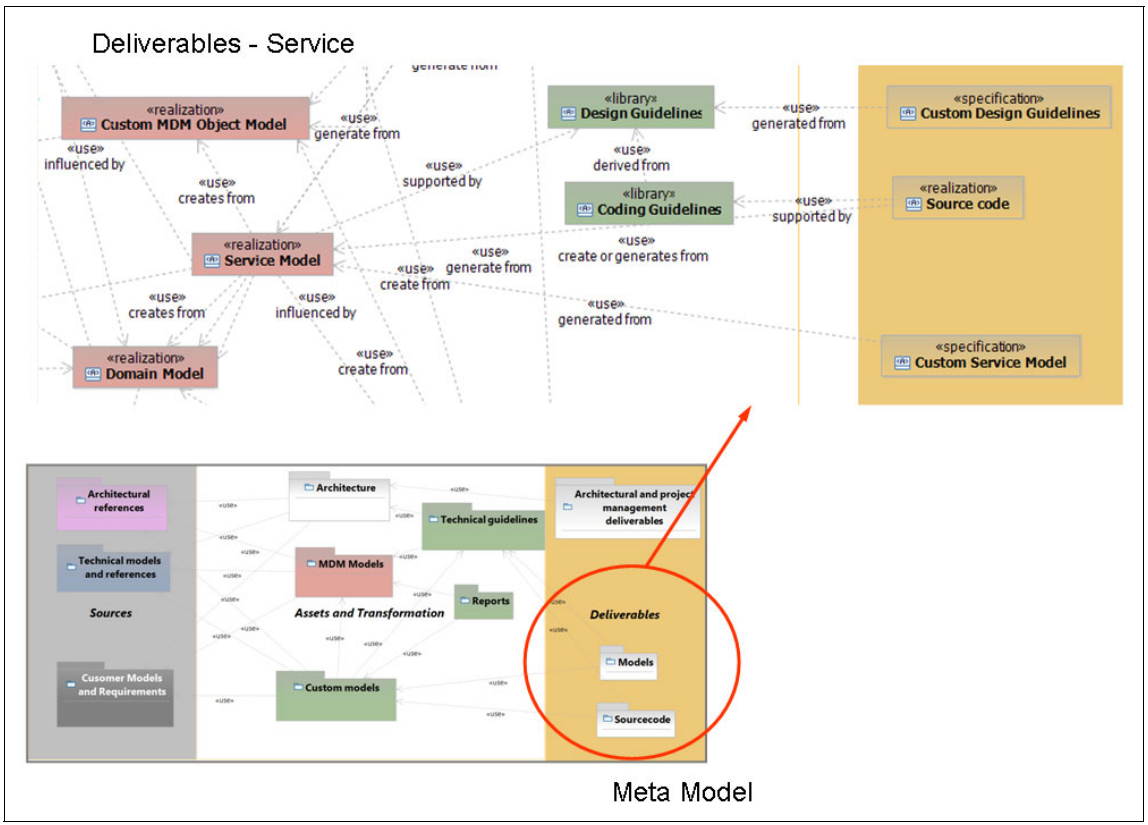

Figure 4-15 View of the metamodel with a focus on service deliverables

The following service deliverables are key:

- Service model

  The *service model* deliverable represents all business services that we expose to external systems. Here we refer to the printed representation of the service model. It should outline inputs, outputs, prerequisites, general service flow, and exceptions, to name but a few aspects. Because the service model also contains internal structures, the deliverable is only an extract.

- Source code

  Source code is required to implement the custom components of InfoSphere MDM Server.

4.2.5 Relationships between artifacts

Each of the aforementioned artifacts plays a role in the overall methodology. To be able to understand it, it is important not just to be able to describe what each individual artifact is, but even more so the relationship that each artifact has with any other. It is here that most information regarding the dependencies is stored. It describes in which sequence artifacts can be created and how they depend on the input of other artifacts to contribute to the next steps.

Table 4-1 lists the artifacts and their relationships to other artifacts within this metamodel.

Table 4-1 Artifacts and their relationships to other artifacts

Artifact	Related to	Artifact
Architectural Decisions	Influenced by	MDM Solution Reference Model
	Influenced by	MDM Architecture Blueprint
Architectural Layers	Influenced by	MDM Solution Reference Model
	Influenced by	Component Model

Artifact	Related to	Artifact
BIRT Reports	Generate from	Data Mapping Designs
	Generate from	Custom MDM Physical Data Model
	Generate from	Domain Model
	Generate from	Product behavior
	Generate from	Custom MDM Logical Data Model
	Generate from	Operational Model
	Generate from	Custom MDM Object Model
	Generate from	Service Model
	Generate from	Component Model
Business Domain Model	Influenced by	Requirements
	Adheres with naming to	Glossary Model
	Influenced by	Conceptual Model
Business Glossary	Generated from	Glossary Model
Business Process Model	Influenced by	Requirements
	Influenced by	Industry Models
	Models manually from	Business Processes
Coding Guidelines	Derived from	Design Guidelines
Component Design	Enriches	Component Model
Component Model	Influenced by	Architectural Decisions
	Influenced by	Requirements
	Models manually from	Developers Guide (PDF)
Conceptual Model		Glossary Model
	Imports from	Conceptual Model
Custom Architectural Decisions	Create from	Architectural Decisions
Custom Component Model	Generated from	Component Model
Custom Data Mappings	Generated from	Data Mapping Designs

Artifact	Related to	Artifact
Custom Design Guidelines	Generated from	Design Guidelines
Custom MDM Logical Data Model	Transform from	Custom MDM Physical Data Model
	Adheres with naming to	Glossary Model
	generated from	Custom MDM Logical Data Model
Custom MDM Object Model	Adheres with naming to	Glossary Model
	Create from	MDM Object Model
Custom MDM Physical Data Model	Generated from	Custom MDM Physical Data Model
	Influenced by	Domain Model
	Adheres with naming to	Glossary Model
	Creates from	MDM Physical Data Model
	Transform from	Custom MDM Object Model
Custom MDM Service Model	Creates from	MDM Service Model
	Influenced by	Domain Model
	Adheres with naming to	Glossary Model
Custom Operational Model	Creates from	Operational Model
Custom Project Plan	Creates from	Project Plan
Custom Service Model	Generated from	Service Model
Custom Technical Architecture	Create from	Architectural Layers
	Create from	Architectural Decisions
	Create from	MDM Architecture Blueprint

Artifact	Related to	Artifact
Data Mapping Designs	Create from	Target Physical Data Models
	Create from	Custom MDM Physical Data Model
	Create from	Domain Model
	Create from	Conceptual Model
	Create from	Source Object Models
	Create from	Target Object Models
	Create from	Custom MDM Object Model
	Create from	Source Physical Data Models
Domain Model	Adheres with naming to	Glossary Model
	Influenced by	Industry Models
	Transform from	Business Domain Model
Glossary Model	Imports programmatically from	Transaction Reference Guide (PDF)
	imports programmatically from	Common Data Dictionary (PDF)
	imports programmatically from	Common Services Transaction Reference Guide (PDF)
	imports programmatically from	Data Dictionary (PDF)
	Imports from	Industry Models
MDM Logical Data Model	Imports from	Logical Data Model (LDM)
MDM Object Model	Transform from	MDM Logical Data Model
	Adheres with naming to	Glossary Model
MDM Physical Data Model	Described by	Common Data Dictionary (PDF)
	Adheres with naming to	Glossary Model
	Described by	Data Dictionary (PDF)
	Imports from	Physical Data Model (`dbm`)

Artifact	Related to	Artifact
MDM Service Model	Described by	Transaction Reference Guide (PDF)
	Imports through MIH tool	Physical data model
	Described by	Common Services Transaction Reference Guide (PDF)
	Adheres with naming to	Glossary model
MDM Solution Reference Model	Models manually from	SOA references
	Models manually from	MDM references
	Models manually from	MDM project references
	Influenced by	MDM architecture blueprint
	Influenced by	Developers Guide (PDF)
	Influenced by	Component Model
Operational Model	Models manually from	Installation Guide (PDF)
Product behavior	Models manually from	Developers Guide (PDF)
	References	Component Model
Service Model	Influenced by	Data Mapping Designs
	Creates from	Custom MDM Service Model
	Creates from	Domain Model
	Influenced by	Domain Model
	Adheres with naming to	Glossary Model
	Creates from	Custom MDM Object Model
	Supported by	Design Guidelines
Source Logical Data Models	Adheres with naming to	Glossary Model
	Create from	Source Data Model
	Transform from	Source Physical Data Models
Source Object Models	Transform from	Source Logical Data Models
Source Physical Data Models	Imports from	Source Data Model

Artifact	Related to	Artifact
Source code	Supported by	Coding Guidelines
	Create or generates from	Service Model
Target Logical Data Models	Transform from	Target Physical Data Models
	Adheres with naming to	Glossary Model
	Imports from	Target Data Model
Target Object Models	Transform from	Target Logical Data Models
Target Physical Data Models	Create from	Target Data Model
Use Case Model	Influenced by	Requirements
	Models manually from	Use Cases
	Influenced by	Business Domain Model
	Uses	Business Domain Model

4.3 Structure

Setting up the right project structure is not an easy task. More often than not, the structure is selected based on thoughts at that moment in time. From there it grows and evolves over time only for you to discover better ways when too much content has accumulated and it is too late for restructuring. Setting up the structure right from the start is therefore crucial, especially in some of the larger projects that we encounter in the master data management solution space. It needs to be logical and reflect the systems in place. It must allow for many people to work on the same model without disturbing each other and possibly without having to merge. With all the right structures in mind, we still have to consider that while a code merge is kind of manageable, a design merge is a significantly larger effort.

In the following sections we first describe the influencing factors on our project structure. Next we discuss a somewhat idealistic way of structuring the project, which is independent from any tool used. We then move on to describing the structure in the context of the Rational toolset.

4.3.1 Ingredients

There are many external factors that affect our decision on one structure or another. This is not different for the selection of a package structure in a development or modeling environment. Before we can define the right structure, we need to be aware of key factors that influence our decision for the structure. We determine the right structure based on these factors:

- Team development
- System context
- Development method
- Domain-driven design

Team development

Project structures can support efficient and effective team development or interfere with it up to a point where teams are barely able to make any progress. There are many influences on such an outcome. One of them is certainly the selection of the right project structure in the tools that we are working with. Creating it correctly allows for the most proficient design and development processes, while a wrongly created structure imposes obstacles on the design and development process in the best case.

In respect to the team development, we have to understand which teams the project has and where the subject matter experts are that have the responsibility of capturing their knowledge in a model used for development. Typical team setups are aligned with the system context and component model.

Other aspects again include organizational functions such as these:

- Business
- Architecture
- Design
- Implementation

Two other functions are often forgotten about. They are also beneficiaries of such models:

- The test team that usually finds it easier to navigate a structure following a logical layout
- The project management, which is expected to set up tracking and reporting against the model based on the project methodology defined

The biggest problem in respect to the definition of the right structure in a team development environment is the decision about the granularity of the packages that we have to work on. Deciding on the content for each package in this context is another issue, but it is directly related. Packages that are too coarse-grained span too many aspects of the system and either prevent concurrent updates from happening due to explicit locks on the packages or require the merging of modeled elements and relationships. While the merging of the elements is still somewhat manageable, the identification and merging of element relationships is cumbersome.

Packages that are too fine-grained pose issues of a different kind for the teams and the source control system in place. Many small packages are more difficult to handle for both the teams that are assigned owners of these packages and for the source control system managing them. In addition, there is the issue of managing dependencies across packages. This in itself is difficult because modifying such relationships requires checkout of both packages instead of working with a single package only.

System context

The system context provides a natural means for determining one of the most important aspects of building a new system. It provides an insight into the surrounding systems that our new system is or will be in touch with. The reason for its importance lies in the fact that we have to provide for some form of integration. If we have to implement integration aspects, we also have to determine a structure for how we want to handle the models related to these systems. Some of the models that we require with respect to such integration are the interface and data models, use cases, and business processes.

The system context from our case study fictitious insurance company, for example, defines the requirement to integrate the master data management system upstream with system A and system B, as well as the Data Warehouse downstream. It also requires a UI for the data steward. This is depicted in Figure 4-16.

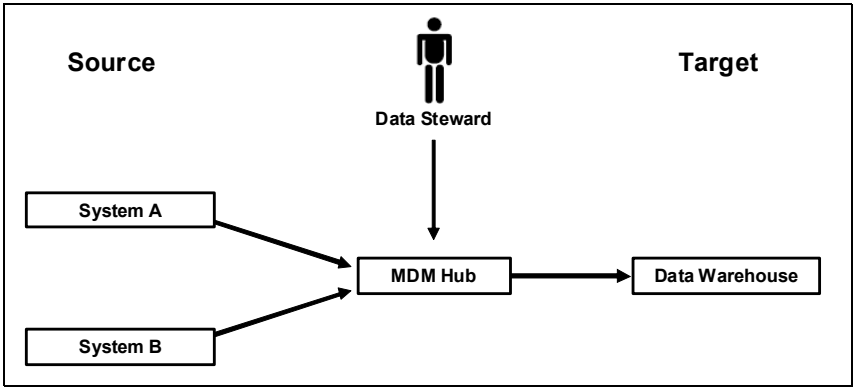

Figure 4-16 Sample system context

Based on the that system context, we identify at least four models areas:

- System A
- System B
- MDM System
- Data Warehouse

Each of these systems can and should be modeled separately. The reason for this is that most system aspects are independent of other systems. To shorten time for the modeling tasks, these system models can be defined concurrently. With such a structure we ensure keeping the interference across systems at a minimum. The system experts that carry out the modeling are separated from any other system.

In addition, such a system landscape is often managed within the confines of an enterprise architecture. Even based on a small system landscape, it becomes clear that there needs to be a single place where all aspects join together. Presuming that an enterprise architecture indeed maintains the system landscape, the most important aspect in conjunction with a domain-driven development is the conceptual enterprise information model. It describes all entities available across all systems in the enterprise. It is here where the mapping to the individual, system-specific, logical data models should be defined. This allows us to track the data flows and usages.

Development method

Another important aspect in determining the right project structure is the development method itself. MDD projects often require a different structure than extreme programming projects that explicitly rule out design. A small project or traditional waterfall type of projects, where the various aspects of the system are defined one after the other and therefore grow naturally in that order, might again be different.

Sometimes structures are set up acknowledging the architecture and therefore creating packages that match the components defined for the system. They place all models required right with the components defined for the system. However, a true separation of concerns is not always guaranteed when following such an approach.

Here the focus lies on the domain-driven development methodology. The difference is that the domain model is always at the center of each problem. It functions as the glue between business and IT concepts and binds the various models together. For example, the use case model uses objects from the domain model, and so does the business process model. Also, IT-related models are based on another abstraction of the domain model, such as the service design model, which is operating on objects from the domain model.

Based on the domain-driven development principles and the models defined therein, a baseline for the project structure is already visible. It is the categorization into:

- Analysis models
- Design models

Because we eventually aim for source code generation and also have to bind everything to the enterprise, we also add the following models into the equation:

- Implementation models
- Enterprise models

Domain-driven design

A separation down the line dictated by the development method appears to make most sense because there are traditionally entirely different groups of people working on them. Detailing this high-level structure in this context further, there will be sub-level categories as follows:

- Enterprise models
  - Conceptual model
  - Architectural models

- Analysis models
  - Business domain model
  - Business glossary
  - Use case model
  - Business process model
- Design models
  - Domain model
  - Component model
  - Deployment model
  - Service model
- Implementation models
  - Domain model
  - Service source code
  - Physical data model

However, a project typically requires more structuring to represent most aspects of the system that we have to design. Based on that, the extended project hierarchy could look as follows, keeping in mind that the added subcategories of the models do not represent a complete view of any system but are only used for demonstration and descriptive purposes:

- Analysis models
  - Business Domain Model
    - Subdomain Customer
    - Subdomain Account
  - Glossary Model
  - Use Case Model
    - Customer Use Cases
    - Account Use Cases
  - Business Process Model
    - Batch Processes
    - Front-end Processes
- Design Models
  - Domain Model
    - Subdomain Party
    - Subdomain Contract
  - Component Model
    - Sub Component x
    - Sub Component y
  - Deployment Model
  - Building Blocks
  - Operational Models
    - ALOM

        - TLOM
        - POM
    - Service Model
- Implementation Models
    - Physical Data model
        - Table 1
        - Table 2

These sub-structures are equally relevant when we consider the team development aspect. Often teams own more than a single part of the entire project. We need to observe and define the package structures accordingly.

4.3.2 Tools

The tool chosen to implement the MDD methodology has no significance. However, when it comes to the definition of project structures, the tool plays a vital role. It impacts how the resulting model structure looks. Some tools are more strict or relaxed in the way that they allow definition and structuring of the projects that they contain. We have to adopt to the capabilities of our tools and deviate from any idealistic project structure to one that our tool supports.

Because this book uses IBM Rational and IBM InfoSphere tools, we have the following options for structuring a project using IBM Rational Software Architect:

- Workspaces
- Projects
- Logical units
- Fragments
- Packages

The base for any IBM Rational Software Architect setup is a workspace that needs to be started and created first. Each solution implementation should be represented by its own individual workspace.

A workspace holds one or more projects. These are represented by folders containing a `.project` file on the file system and are created within the IBM Rational Software Architect environment. They are a means of creating a rough, high-level outline of what we want to achieve. Creating a separate project for each system and each model type (business, design, or implementation) therein makes sense in the context of a large-scale master data management solution implementation.

Within such a project, the logical units are placed. Logical units are created by adding a UML project. They are slightly more fine-grained than the projects and have a file extension of `*.emx`. Such files can be exchanged between team members directly or through source control systems while maintaining references between different logical units. They should be used to define the structure within each system and model type.

The structure defined through logical units should already suffice for smaller projects. However, larger projects might require even further breakdown. A large domain model, for example, could require different subject matter experts and thus further categorization. This can be achieved by introducing fragments. Fragments maintain an `.efx` extension and can also be exchanged directly or through a source control system separate from each other.

Keep in mind, that the requirement for merging designs will become more and more relevant the more fragments we create. This is due to the increasing number of dependencies that we introduce between them. With a certain number of smaller fragments it becomes virtually impossible to avoid using a source control system. While we should not avoid them anyway, we now have to manage the different version of each package and possibly even avoid having anyone work on a related package.

The first four all are means for structuring a project in such way that they can be individually exchanged between members of the design team and worked upon independently. They are supporting storage in a source control repository. Packages then represent a means of further structuring in a team environment, but without allowing us to work on them separately.

The mapping from the models previously discussed to the tool-related structuring elements is as shown in Table 4-2, with *fragments* remaining to be used as a project-specific structural mechanism.

Table 4-2 Mapping models to structuring elements for master data management solution x

Project	Logical unit	Fragment
System A Analysis	Business Process Model	
	Business Domain Model	Subdomain 1
		Subdomain 2
	Use Case Model	
	UI Model	
System A Design	Component Model	
	Domain Model	

Project	Logical unit	Fragment
	Service Model	
	Deployment Model	
	Logical Data Model	
System A Implementation	Physical Data Model	
System B Analysis		
System B Design		
System B Implementation		
MDM System Analysis		
MDM System Design		
MDM System Implementation		
DWH Analysis		
DWH Design		
DWH Implementation		
Enterprise Models	Conceptual Model	
	Glossary Model	
Profiles		
Reports		

4.3.3 Putting it all together

The initial sections in this chapter described possible structures from idealistic points of view. We have seen different structures depending on the ingredients that we focused on. Now we outline how we can put this all together into one large, consistent project structure.

Here the focus is on the IBM Rational toolset. We define the structure according to its capabilities and best practices while honoring the aforementioned aspects of team development, system context, and the development method using the following tools:

- IBM InfoSphere Data Architect for all data modeling related aspects
- IBM Rational Software Architect for all UML models

The reason for mentioning is that they define the location for some of the projects for which they are responsible. One example is the location of the data models. IBM InfoSphere Data Architect stores them in a folder named `Data Models`. Although we can also decide to move any of these models, we do not see any additional benefit to storing them in a custom folder. The overhead and burden of moving those projects has no relation to the benefit that we gain by adhering to an idealistic structure. This is why we leave some models in their default location.

Because we focus on the domain-driven development methodology, the separation between enterprise, analysis, design, and implementation models with their respective packages defined therein is an essential ingredient.

4.3.4 Enterprise models

We have not identified many models that we require in our methodology and that are in relation to enterprise architecture. Even though there are only two artifacts that we related to, the enterprise information model and the glossary, we still set up a dedicated project for the enterprise architecture. This is also because there are typically more models related to it. An exemplary project structure can look as shown in Figure 4-17.

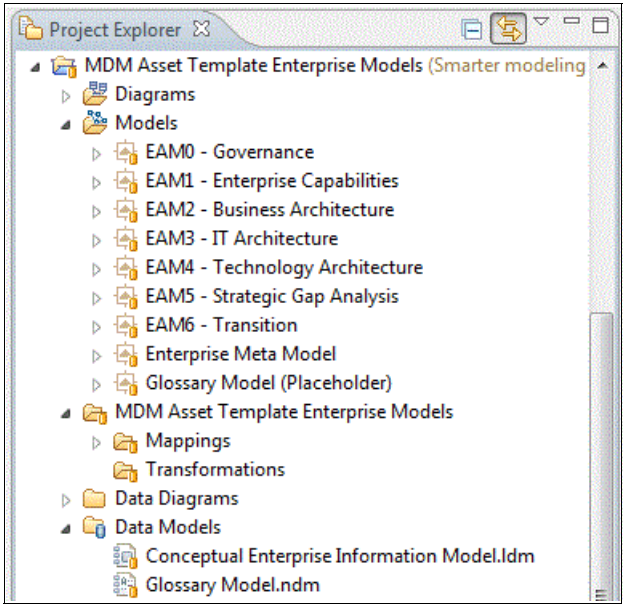

Figure 4-17 Structure for master data management asset template enterprise models

There are a number of models defined within the confines of the enterprise models project. There are three aspects that we elaborate on further:

- Enterprise metamodel
- Glossary Model (Placeholder)
- Data Models

You will see us defining metamodels in many locations. We believe in their value for a number of reasons. In the simplest form, they provide project team members with a quick overview of all artifacts that belong to one particular project and how they are related to each other therein. In a more complex step, we can rely on the definitions contained in the metamodel to drive our transformations between models.

There is one so-called (Placeholder) model located in the project. Its existence plays no role in the overall project but is based on a break in the tool chain. Because we are using both IBM Rational Software Architect for all UML models and IBM InfoSphere Data Architect for all data models, we have two tools at work. Unfortunately, they are not as tightly integrated as we would like to see. One of the consequences is that we cannot draw any IBM InfoSphere Data Architect based data model directly into our UML-based metamodel. Because we still want to describe the role of the data model within all other enterprise architecture models, we created the placeholder model. This we can include in our metamodel and describe its dependencies.

The location for the data models appears a little out of place. We would expect all models to be placed in the models folder. Also, from an enterprise architecture modeling perspective, the conceptual enterprise information model belongs to the business architecture models. It is again due to constraints with regard to the integration of the two distinct tools, IBM Rational Software Architect and IBM InfoSphere Data Architect, that dictate the location for this model. IBM InfoSphere Data Architect first creates this model in this location. Because we do not believe in any additional value in the mere relocation of the model, we keep it in its original place, a folder named `Data Models`.

4.3.5 Analysis models

The analysis models are the starting point for our methodology. It is here that the core business-related models come into existence. The resulting project structure therefore looks as shown in Figure 4-18 for the analysis models.

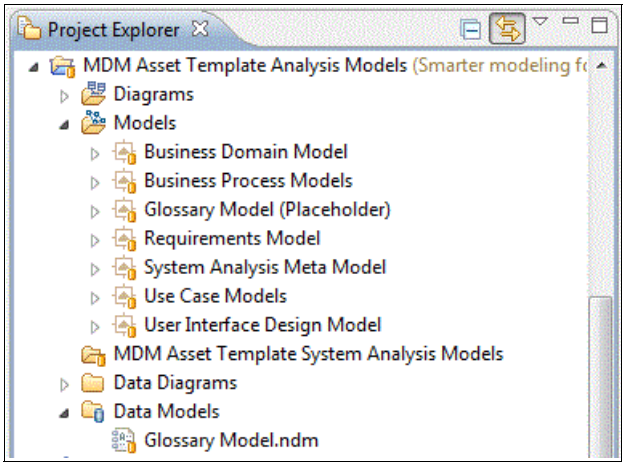

Figure 4-18 Structure for MDM Asset Template Analysis Models

There are a number of models defined within the confines of the analysis models project. These are three aspects that we are going to elaborate on further:

- Requirements Model
- User Interface Design Model
- Glossary Model

The requirements model is something that we have decided to add directly within IBM Rational Software Architect. We could also rely on additional tools, such as IBM Rational Requisite Pro or IBM Rational Requirements Composer. However, in respect to our methodology these tools do not represent any additional benefit. This is not to say that you should not use them or that they are not useful. To the contrary, they provide you with a structured and elegant way to manage your requirements. In the first case, we have skipped these tools to avoid the introduction of yet another tool. Every additional tool represents yet another challenge in respect to the traceability that we require. Based on the possibility of modeling the requirements in IBM Rational Software Architect directly, we achieve both modeled requirements with a level of detail suitable to our methodology and avoiding hopping into another tool.

At this point we have excluded any description of the user interface design models. In respect to our modeling methodology, it represents just another model. We drive the user interface integration from it, again in conjunction with the domain model.

The glossary model is again in the IBM InfoSphere Data Architect based model. Its use is predominantly focused on model users. However, we have foreseen the export of the IBM InfoSphere Data Architect based glossary into a glossary that is much more suitable to a wider business audience. IBM InfoSphere Data Architect glossaries can be exported to the business glossary provided through IBM InfoSphere Information Server. This, however, represents to us at this point merely a final deliverable that has no relevance to the modeling itself. There is better integration between IBM Rational Software Architect and Business Glossary, which makes it more tempting to use the IBM InfoSphere Information Server Business Glossary in the future.

4.3.6 Design models

The design models are the next logical step toward our solution implementation. Figure 4-19 depicts a possible project structure.

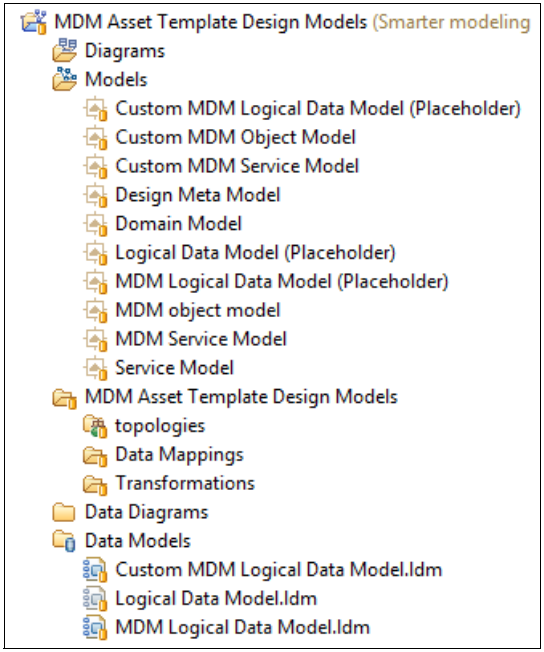

Figure 4-19 Structure for MDM Asset Template Design Models

There are a number of models defined within the confines of the design models project. These are two aspects that we elaborate on further:

- Business Domain Model
- Topologies

The business domain model appears for the second time. The first time that it surfaced it was related to the analysis models. Here we enrich it with further, somewhat more technical, information. One example is the transformation of cardinalities from x:n to x:m based on the definition for time slicing in the analysis model. While this can be dealt with in an automated transformation, we sometimes need to manually intervene too. This can be in case of a definition for multiple inheritances in the analysis models and that we need to resolve. Perhaps further business analysis resolved such multiple inheritances to a combination of single inheritance in combination with a rule set. We can keep the analysis models, as they were showing multiple inheritance, but resolving those technical challenges in their model counterparts on the more technical levels.

While we are not dwelling on topologies, you can already see how we can embed this kind of a model in the overall set of project-related models.

4.3.7 Implementation models

The last set of models in the development chain is the implementation models. Figure 4-20 shows the definition of some of the required models on this level.

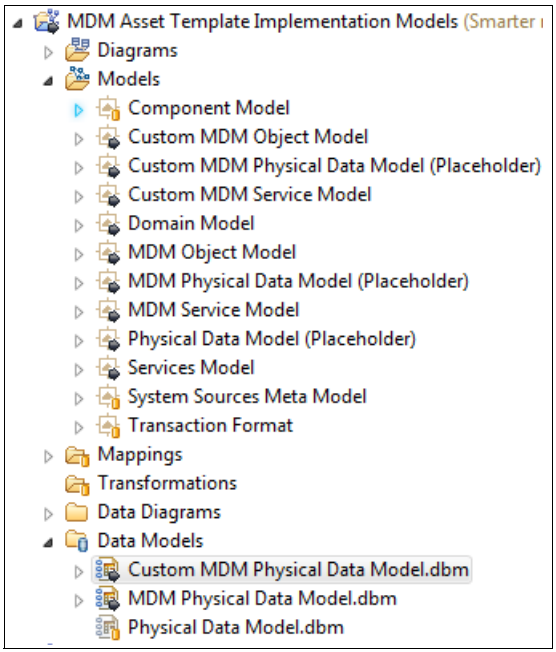

Figure 4-20 Structure for MDM Asset Template Implementation Models

There are a number of models defined within the confines of the implementation models project. These are two aspects that we elaborate on further:

- Transaction format
- Physical Data Model

The transaction format is one of the technical aspects. It describes the communication protocol, and here in particular the message format. It also considers the objects that it has to transport, but still, the focus lies on the header information. In respect to InfoSphere MDM Server, there is one thing in particular that we have to observe. There are business-related elements in the header. This includes point-in-time attributes, as well as the access token as required for multi-tenancy implementations. Therefore, we cannot rely on the introduction of these elements on this lowest level alone. Instead, we have to adopt the business elements at least one level up, on the design level, already.

As opposed to the other data models, the canonical, the domain, and the logical data models, this physical data model represents a true implementation-related model. It represents the lowest model from which we derive the SQLs and DDLs to populate the database schema. Based on these thoughts, we have separated this model type from all other models. Traceability and mappings maintain the relationship to other data models.

4.3.8 Sample case study projects

Considering now the system context, this structure repeats for each system involved, including the master data management system itself. Figure 4-21 represents an example structure.

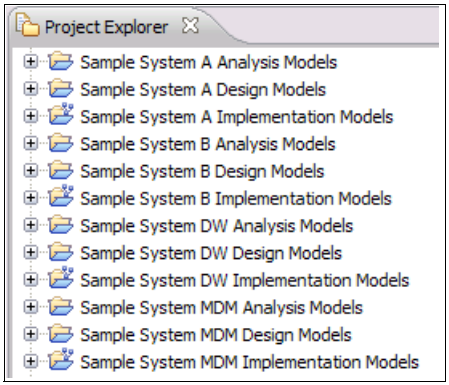

Figure 4-21 Structure for sample models against the case study

The structure outlined follows our definitions in 4.1.1, “Modeling approach” on page 133. That meta-metamodel defined the number of projects that we want to derive from the system context. Consequently, you see one project for enterprise models, as well as one analysis, design, and implementation model for each system.

4.3.9 Additional models

The foundation has defined the following additional projects in support of the project-related models:

- Profiles
- Transformations
- Reports

Profiles

Figure 4-22 shows a few exemplary profile definitions. The number of profiles and the content therein depend on the level of automation you want to achieve.

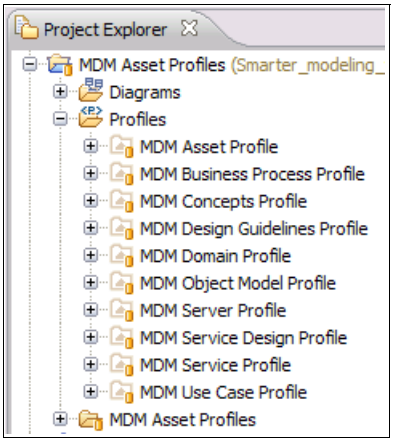

Figure 4-22 Structure for MDM Asset Profiles

Transformations

With respect to the transformation, you have to observe a special behavior. The definition of the transformations is defined in the transformation models. As soon as that has been defined and deployed within the tool, you can configure these transformations. Here you specify the parameters required, such as input and output models. These transformation configurations will be placed where the transformation is required. If, for example, the InfoSphere MDM Server models require a transformation of the analysis domain model to the design domain model, then the corresponding transformation configuration resides in a InfoSphere MDM Server related design model project.

Reports

Another major area regarding the supporting infrastructure and related projects is covered in the reports section. This is the case because our intention is to base as many deliverables and project management ad hoc reports as possible on automated generation through BIRT reports.

Figure 4-23 shows a report model structure.

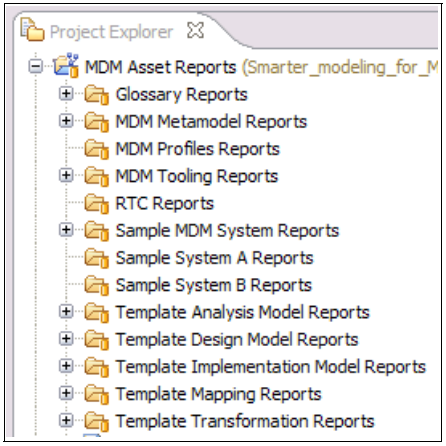

Figure 4-23 Structure for MDM Asset Reports

Most notable is the structure that is closely related to the overall project structure. The benefit is that we are able to aggregate all reports in a single project and thus can dedicate a specialized team or individual to maintaining these independent of any of the other solution-related projects.

4.3.10 Templates

During our previous discussions and related diagrams you might have noticed that we often refer to template projects. We believe in templates as a good starting point for specific solution implementation projects. They can provide you with the same structure every time that you instantiate them.

For this purpose, create template projects. Due to the nature of IBM Rational Software Architect projects that host models, we are not just suggesting that you create a sample UML model, but instead create an entire project consisting of the models described starting in 4.1.1, “Modeling approach” on page 133 and storing that as a template.

After we have such a template created it would be ideal to be able to select that from the project wizard as the template so that the new project that we create is already populated with all models as defined in the template. Unfortunately, IBM Rational Software Architect does not provide for this. Instead, it allows only for selection of a default model to be created within a new project, and only when certain types of projects are being selected, such as an UML project. In case of creation of a general project, this option of populating the project with a template model does not exist.

There are two options for overcoming this issue:

- Copy
- Plug-In

Copy involves the definition of the aforementioned project structure containing all models. Such a project will be stored as a template project. When we have to create the models for a new system, we then copy the template project and paste it using the system name. This way we copy all models contained in that template instead of just selecting one model as is the only option through the default new model wizard. We recommend using this approach in small projects of short duration or in projects with a large number of members or where members are frequently changing.

Plug-ins are another option to overcome this challenge. The creation of a new project wizard could allow for copying the entire template project structure. This can be done without programming required, but such a plug-in would have to be developed once and then installed on all machines of designers working with these models. This approach is much better automated and integrated with the tooling. Long-running projects could be supported following this integrated approach.

Based on the previous chapters, we propose the definition of the following templates:

- Template Enterprise Models
- Template Analysis Models
- Template Design Models
- Template Implementation Models

The instantiation of these templates largely depends on the system context that we are working with. In our case study system context we have the following systems:

- System A
- System B
- Master data management
- The data warehouse

Following our template instantiation rules we achieve the correlation listed in Table 4-3.

Table 4-3 Correlation

System	Template	Instantiation
All	Template Enterprise Models	Enterprise Models
System A	Template Analysis Models	Analysis Models
	Template Design Models	Design Models
System B	Template Analysis Models	Analysis Models
	Template Design Models	Design Models
Master data management	Template Analysis Models	Analysis Models
	Template Design Models	Design Models
	Template Implementation Models	Implementation Models
Data Warehouse	Template Analysis Models	Analysis Models
	Template Design Models	Design Models

Because not every system requires all models defined within a template, we suggest the deletion of the models that are not required. For example, in most cases we do not have the use cases of the existing systems available and predominantly require them for the new master data management system that we want to build. In such a case, we would remove the use case model from system A and B Analysis Models.

4.4 Design guidelines

A consistent modeling approach implies strict and consistent guidelines. This is true even more so when we are aiming for automated code generation. When we talk about design guidelines we actually mean two different things:

- Guidelines that are applicable to domain-specific models
- Guidelines that are context-independent and generic

Note that the generic guidelines encompass both guidelines applicable to model instances and those applicable to standard modeling practices. Any deviation from the established set of guidelines will potentially result in a model that is less maintainable and less useful in automated generation processes.

Think of a designer who joined the team recently and now wants to learn more about a specific concept and how it is represented in the model. She will struggle figuring out the correct meaning and right way of modeling if three different designers have realized the concept differently. Guidelines should reflect best practices and proven patterns. This prevents us from reinventing the wheel and spending time on resolving problems that have been dealt with long before. Especially in the context of model-driven methodologies, guidelines help make sure that a model can be processed automatically and can actually serve as a basis for code generation.

To achieve this goal, guidelines cannot contain any ambiguities or alternatives. They should state as clearly as possible which modeling elements and which notation must be used in each and every context that is relevant to the domain that is available. Although we discuss only design guidelines in this chapter, they can be found throughout the entire model space. Our view on design guidelines includes the following:

- The metamodel
- UML notation elements
- Naming conventions
- Model-related documentation
- Model annotations
- Design repositories
- Change logs
- Reuse
- Sources for design guidelines

4.4.1 The metamodel

As every part of our model, the design guidelines are also based on a metamodel. Each concept described previously is just an instance of this metamodel. The impact of that is that we cannot simply take the model guidelines discussed here and apply them directly to the source code model.

Let us have a look at a specific example. A loop in the design can be defined such that it complies with standard UML elements and without any consideration of programming languages. However, in the source code model, a loop can be realized differently dependent upon the language that we use, the available context, and so forth.

UML designers might not be aware of certain restrictions or features of the programming environment, nor are they necessarily experts in the field of implementation patterns and technical architectures. It is therefore important to differentiate between design and technical implementation guidelines. The common denominator of these guideline instances is a metamodel.

In the following sections we describe the design instance. This is practical because the implementation guidelines will be mentioned later on in the context of the implementation models and transformations. Most implementation patterns will be described and implemented through transformations.

4.4.2 UML notation elements

Although this requirement might appear trivial at first, it is important to make consistent use of the UML notation elements. The benefit of UML is that it can be used as a generic modeling language. This advantage turns against us when we want a design to follow one approach, not many. For example, there are several ways to model detailed business flows, but teams need to agree on one approach. This is even more important in MDD, where we turn each design artifact into a piece of source code. The guidelines must specify which elements of activity diagrams and which kind of associations we have to use in a particular context. The context and the intention need to be explicitly and clearly expressed so that the designers can follow them by the letter.

We might require specific notations to describe logical blocks that are used repetitively throughout the design. Dependent upon the complexity of the model, designers might decide to use conventions, stereotypes, extensions of existing UML elements, or even new elements as part of a domain-specific language. Regardless of the actual approach, these logical pieces must be used in a consistent way, too.

UML guidelines should also encompass layout rules. They might even demand the sequence of modeling steps and tool-specific guidance. Just think of a case where a modeling element might either only become part of the diagram or the actual model dependent on the tool flow.

4.4.3 Naming conventions

Often naming conventions are neglected or considered less important. To designers it might not make any difference whether a service name contains white spaces or underscores or whether a camel-case notation is used. It usually does not even have an impact on the readability of a model. Even when an expression is abbreviated, as long as the abbreviation is understandable, the readability generally does not suffer.

In combination with an MDD, however, all of these details play an important role with regard to code generation, traceability, and maintainability. Code generation requires consistent use of names. Any deviation from the defined conventions can cause generation failures or errors in the source code generator. Loss of traceability is another consequence when we are not observing the naming conventions. Given that a designer chooses a non-consistent name in one of the diagrams and later on, due to this inconsistency, this name needs to be changed, we lose the ability to trace back the history of this diagram. Being able to see the history of a specific element becomes especially important in the context of change logs.

Inconsistent names also make it much more difficult to find related elements by name. This in turn natively impacts the maintainability of our implementation. Imagine a situation in which a change of a diagram name results in a duplication of the contained logic. Dependent on the impact of the change, teams might decide not to remove the old element immediately, because the deletion would necessitate the change of many related elements. With every temporary duplicate that we introduce it becomes harder to maintain the design model, implementation, or glossary. Reduced maintainability might also manifest itself through the inclusion of potential deviations to cover every possibility that the design might have to provide to describe a certain function. The framework team does that with good intent and in an attempt to prevent generation failures. Still, it is these special handlers that clutter the generation logic with unnecessary fragments and divert our attention from the main concern.

Which elements should naming conventions include and what are the general rules for good naming conventions? As a general rule, the uniqueness and consistency of names is supported by means of the glossary. The glossary provides a valid base for all known functional names. This includes domain objects and their attributes and also services names, business rules, and so forth, that must reflect these naming standards. For example, if the domain model object is called MobilePhone, a corresponding service should not be named addCellular.

Names should follow a clear convention regarding upper and lower case. We recommend adopting the camel case notation used for programming languages such as Java. The advantage of this convention is its readability and its independence of special characters. Even if naming conventions depend on the frameworks and the tool chain that you use, we think that it makes sense to adopt a common standard for the entire design. The differences could be covered for in the generators that can also transform from our design standardization to the tool and framework-specific standardization.

Because we mainly aim for InfoSphere MDM Server based implementations, the names in the master data management domain must comply with these standards. For example, the master data management objects end with business objects, and the table names do not contain underscores, whereas the field names do.

Abbreviations represent another common challenge. We have to use them with great care, especially in the context of systems that only support names of a certain length. While it might look handy to use brief expressions instead of long names, overusing abbreviations leads to confusion and makes it harder to understand the design, especially for project members who have joined the team recently. If we want to use abbreviations, then they must be well-defined and accompanied by rules that clearly state when, where, and how to use them.

The same holds true with regard to redundancies. It might be easier to retrieve an element if the name of the model or its location is part of the name. It is not a good practice to use this kind of redundancy throughout the model. Let us look at an example. A service *Add Beneficiary* can exist on all levels (that is, on the use case, design, and implementation model). To make it easier to find the correct element, a designer could decide to add the context of the service to its name: “Add Beneficiary use case”, “Add Beneficiary Analysis Model”, and so forth. However, a good naming convention should not reflect the location of the model element, but rather its purpose. That is why we recommend using stereotypes instead of naming conventions for this kind of classification. It is the cleaner approach with regard to design and coding patterns, where names reflect what the objects do instead of indicating their technical realization. Realizing this distinction through stereotypes (such as `<<UC realization>>`, `<<serviceDesign>>`, and `<<Implementation>>`) is a valid solution.

Another point of discussion is the natural language that we use. In an international context the decision will almost always be English. This is true for the design model, as well as the source code. Exempt are the models and documentation that we create as part of our deliverables. These can be country and language specific. The models that are affected include the glossary, use cases, and user interface definitions. It is usually advantageous to leave these words without translation, so that we do not lose any context that can be hard to translate otherwise.

4.4.4 Model-related documentation

Model-related documentation can be put almost anywhere. We can add notes to diagrams, comments to UML elements, and models. We can relate this information either to a graphical element or to the model. The key is to provide documentation to all of these. The documentation of a model describes the purpose of the model as opposed to the documentation of the model diagram that describes the function that this model represents in the context of the elements that make up the diagram. The same is true in respect to packages and the elements that they contain. Documentation of the package provides us with an insight into the purpose of all elements within that package, whereas the description of each element refers to the distinct function of that element only.

The trick is to describe each and every element that we draw or add to any model, whether it is model relevant or merely a structural separator. Detailing our thoughts on why we introduced each element increases the level of understanding. One issue that we have observed often is missing documentation on associations, which affects all associations, including the ones between classes, object instances, use cases, and n traceability references. Lack of documentation in this area leaves room for interpretation and poses an issue regarding the subsequent design, implementation, or test activities.

Documentation should be kept close to the element to which it pertains. Diagrams only help to visualize what the model expresses. Regarding the content, we can apply the same reasoning to documentation as to the models themselves. That is, documentation should also be separated by abstraction. It can either belong to the metamodel layer, the model layer, or the realization layer, and should be located accordingly. Source descriptions should be such that they can be used directly in the generation process.

There are also comments that represent meta information in the context of the project. This information usually includes facts like the name or the ID of the author, and the creation and change dates. TODO and FIX comments and information about the scope and the release are also meta information. Most of this data only needs to be added if the configuration management does not provide an adequate level of information. Others are explicitly separated from the versioning control system. We discuss this aspect in greater detail in the following chapter.

4.4.5 Model annotations

Annotating the model means adding supplementary information directly to model elements. This information can be used for generation or analysis purposes in that they might help us identify and specify similar objects and concepts. Usually, annotating UML models means using stereotypes and properties on these stereotypes. Stereotypes are frequently applied to *runtime* aspects of the model and even more so the code to be generated. We also can apply the concept of stereotypes to documents and meta information pertaining to the element that is available.

Note that stereotypes can be considered a specific form of domain-specific language. As such they must be maintained and kept separate from the business project. In Rational Software Architect, the recommended approach to maintain stereotypes is to use UML profiles. A UML profile is developed within a separate project. Although Eclipse already supports UML profiles as part of the UML tooling, IBM Rational Software Architect uses a special XML format for profiles (`.epx` files). Profiles can contain only a limited set of UML elements, including classes, stereotypes, enumerations, and attributes.

Here we focus on stereotypes because we consider them the most important source of domain-specific meta information in UML. Apart from the definition of the stereotype itself, a stereotype supports a limited set of those features used together with classes such as generalizations, attributes, and associations. However, one important reason for defining stereotypes in the first place is to modify or complement the behavior of a meta class. This is done by creating a meta class extension that relates the stereotype to an existing meta class. This is depicted in Figure 4-24.

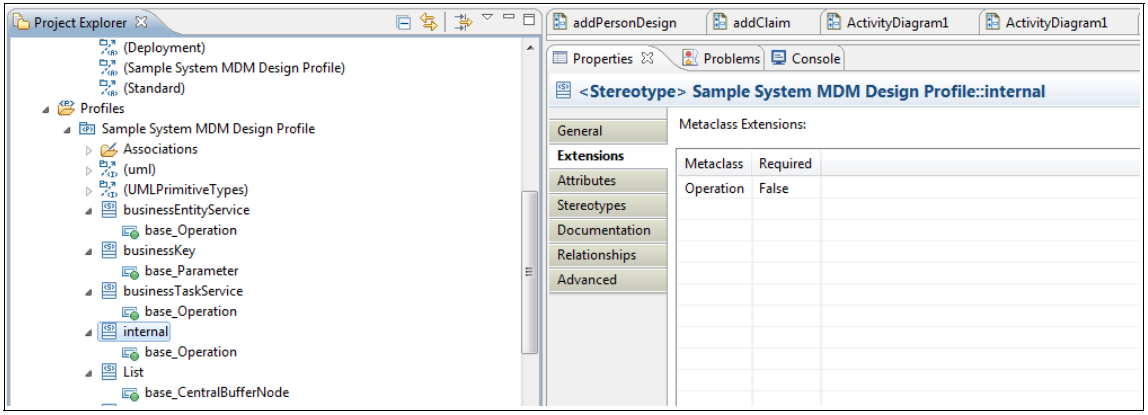

Figure 4-24 Applying stereotypes to meta classes

You can decide whether to apply the stereotypes to all of the instances of a meta class or to specific instances only. In addition, you can also change the appearance of a meta class, for example, by defining a new icon on the stereotype.

What would a real-world scenario that applies all this information look like? Consider a domain object of a specific type and that should always expose a certain set of common attributes. This could be, for example, the date when it was updated. You could define a stereotype for this domain object type and provide the common attributes with this stereotype. After the meta class has been associated with this stereotype, instances of a class using this stereotype will immediately contain the set of common attributes. In other words, this approach allows you to customize the UML metamodel according to your needs. Another good example for this approach is the modification of the appearance of an element depending upon the attributes that have been set or enabled on this element. Say, for example, that you want to make it a little more explicit as to whether an input pin has a multiplicity of *n*. You could easily define a style for this attribute in a profile. So, using profiles also enables you to expose certain attribute states. This is depicted in Figure 4-25.

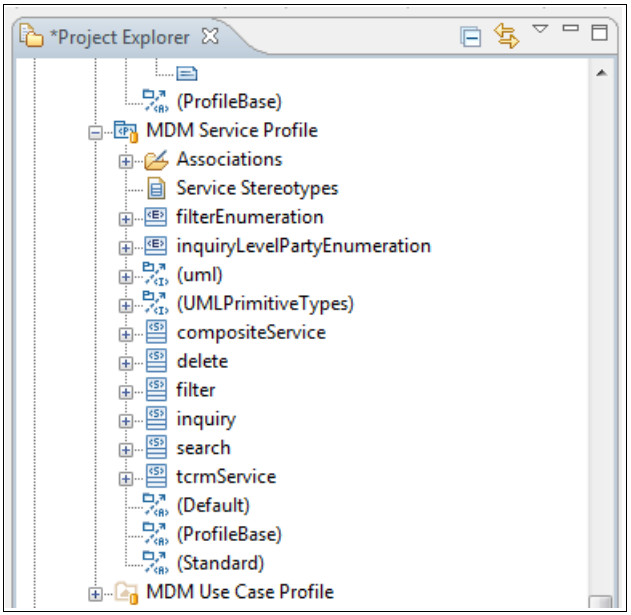

Figure 4-25 Example for a UML profile

After we have defined all required stereotypes in a UML profile, the profile needs to be deployed. Rather than referring directly to the profile project, it will be delivered as separate artifact. This artifact can be imported into IBM Rational Software Architect and be resolved using its pathmap (that is, the logical path Eclipse uses for resource references). It is important to remember that stereotypes should not be defined within a model project because this would effectively prevent other people from reusing the stereotype. Of course, if you do not plan to share any of your project-specific stereotypes with other projects, you could still refer to stereotypes within a profile project, but we do not recommend this approach. The better approach is to build up a separate UML profile repository.

Using deployed stereotypes only ensures that designers or developers do not define meta information on the fly, but comply with the metamodel provided by the team members responsible for the metamodel. While it might appear handy that some tools allow you to do that on the fly, we advise you against doing that. The risk that an element gets quickly introduced and not communicated is simply too big. The consequences can include us forgetting to include this new stereotype in our code generators and subsequently encountering transformation errors.

Here we will important examples of annotations. These examples serve to demonstrate the usage scenarios and categories of stereotypes, but are not meant as a comprehensive list of possible types. We grouped the examples into the following categories:

- Project lifecycle
- Service level
- Service signature
- Service context
- Property behavior
- Domain abstractions

A first group of annotations refers to the project lifecycle of UML elements. The stereotype `<<deprecated>>` is applied to services that are still in use, but that will be removed in a future release. This stereotype translates directly to the @deprecated annotation in Java. After all references to a deprecated service have been removed and it is sure that this service will not be used again, it becomes `<<obsolete>>`.

Other stereotypes refer to the service signature. Most importantly, a stereotype should identify the type of service that is available. In our modeling context, the stereotype of a service denotes the orchestration level, that is, the layer of a service. The most granular services, the create, read, update, and delete services operating on a single master data management object, are stereotyped <<dataService>>. The <<businessEntity>> services on the next higher level orchestrate these data services and provide operations on single business entities (the master data management-independent domain objects). <<businessTask>> services eventually use several of the entity services to provide task-level functionality.

Another important piece of information refers to the provider of the services. As you will see in the Design Models, services will become operations bound to a so-called provider class with the stereotype <<serviceProvider>>.

Also, business tasks can either be exposed to the outside world or be meant for internal use only. In the latter case, the service operation will be tagged with <<internal>>.

Apart from the service level, the service signature can optionally contain further information about the service. An important distinction concerns the create, read, update, and delete services. So your generator needs to know whether a service is an <<add>>, <<update>>, <<delete>>, <<upsert>>, <<read>>, or <<terminate>> operation. Note that <<delete>> refers to actual deletions, whereas <<terminate>> means to set the object's end date, which does not result in physical deletion. The <<search>> operation differs from a <<read>> operation in that it denotes a search by any attributes. A read operation, however, uses the technical primary key to access data. Also, a search operation commonly returns a custom response object, whereas a read operation returns an object or a list of objects of the specified domain object type.

Other stereotypes refer to meta or context information provided with a service call. So, if an element is marked as <<filter>>, it serves to provide a filter criterion for a search.

Functional behaviors fall into the next category of stereotypes. Rules are one of the most prominent members of this category. Rules will be discussed in greater detail in the next chapter, so for now we focus on the stereotypes. We suggest using specific stereotypes for each type of rule. While those rules related to one domain object only are stereotyped as <<objectRule>>, rules related to a specific operation encompassing more than one business object are marked <<flowRule>>.

Another group of stereotypes modifies the behavior of single properties. While some of these modifications, such as the multiplicity, are already part of the service signature, others, like simple constraints, can also be added using stereotypes. Think of a stereotype <<default>> that provides a default value of a property that has not been set or a stereotype <<constraint>> that enables you to define the maximum length of a property.

Another group of stereotypes refers to more complex domain abstractions. These stereotypes are usually applied to domain objects, but are used within a separate context and cannot easily be transformed to source code without further knowledge. The most prominent examples of these stereotypes are the ones dealing with time dependencies. So, denoting an object as time-based could mean that it holds a start and an end date, but it could also mean much more than this.

As you can see, stereotypes cover a wide range of possible usage scenarios. It is therefore essential to distinguish clearly between them. The design will become difficult to read and understand if you fail to define the delineation or if several stereotypes aiming at different areas of concern are used together in one context. Apart from this, designers might be tempted to overuse stereotypes. Stereotypes make perfect sense when we use them with care and deliberation, as we have shown. Arbitrary introduction of stereotypes can lead to an explosion and consequently result in complex designs.

Stereotypes only make sense if they can be applied in a consistent manner to multiple elements. This means that they are context-free. If a stereotype means one thing in one place and a different thing in another, it becomes difficult to distinguish. In respect to our generators, this means that we have to introduce exceptions and alternatives, which make it difficult to achieve an end result of high quality. Stereotypes always have the same meaning throughout the entire model. An example of a poorly defined stereotype is <<loop>>. What does a loop mean in a specific context? Loop through indices or loop through content elements? Can data be manipulated within the loop or is it fail-fast? Most of these concerns are not raised before the implementation phase commences, but nonetheless, designers must take care to use their stereotypes in a precise and consistent way.

Stereotypes are well-defined. This requirement goes along with the requirement for consistency and independence of the modeling context. Stereotypes must be specified such that they can be used to transform the model to the next lower level of granularity. Also, there must be a common understanding of their meaning among designers and developers. There is a lot of room for error if both designers and developers are not fully aware of the exact definition and full scope of a particular stereotype.

A good approach to avoiding this problem is to model a flow explicitly before turning it into a stereotype. Attaching that snippet to the stereotype for further reference serves as visual documentation, in addition to the documentation on the stereotype itself.

We have to take great care to identify the right place for our stereotypes. We should not attach them anywhere, but select the elements that they relate to carefully and deliberately. Let us say that we mark a diagram deprecated. Does this mean that the corresponding activity is also deprecated?

Standards first: If the UML standard provides an adequate concept, use it instead of defining your own stereotype or notation. Having said this, it is also worth adding that the standard notation should not be used for the sake of standards only. If adoption of the standard is stretching the definition too far, we are at risk regarding the maintainability of our design and implementation. The consequence could again be that we have to add exceptions and alternatives to generators, which again makes the design ambiguous. This holds true in general, not only with regard to stereotypes.

4.4.6 Design repositories

Repositories are another source for design guidelines. We can consider even UML profiles as a specific form of repository. Here, we focus on two forms of repositories that contain:

- Utility services
- Patterns

A utility service for us is a small, low-level design artifact that encapsulates a domain-independent functionality, such as string and date operations. We could argue that you would not require any utility at all for this kind of functionality, because it would not be worth modeling them in the first place. However, defining these functions within a separate package serves two purposes. First, defining these utilities in one place avoids the need to have them in several places. Even if the function only consists of an opaque activity with no content, it still makes sense to have a unique name and a common understanding of what the function provides, also in terms of input and output parameters. Second, if the functionality behind such an activity is not obvious, then the repository serves as a place where reusable activities with a well-defined purpose are stored. Therefore, we suggest defining a subpackage for a utility service and placing domain-independent utility-like activities within this folder; this process is described in 7.3.2, “Service model” on page 346.

While discussing the creation of pattern repositories is beyond the scope of this book, it is worth mentioning that IBM Rational Software Architect supports this kind of repository by means of templates and plug-ins. Designers also have access to a range of predefined patterns within IBM Rational Software Architect. Our challenge is in the definition of which patterns should be used, in which context they should be used, and how they should be used in our concrete project. Our recommendation is to focus on a single repository and define a set of patterns that we can use. For each concrete scenario, there should only be one preferred pattern. Adhering to a limited set of patterns helps us improve the conceptual integrity of the design, reduce the complexity in our generators, and as a result increase quality.

Figure 4-26 shows the pattern library.

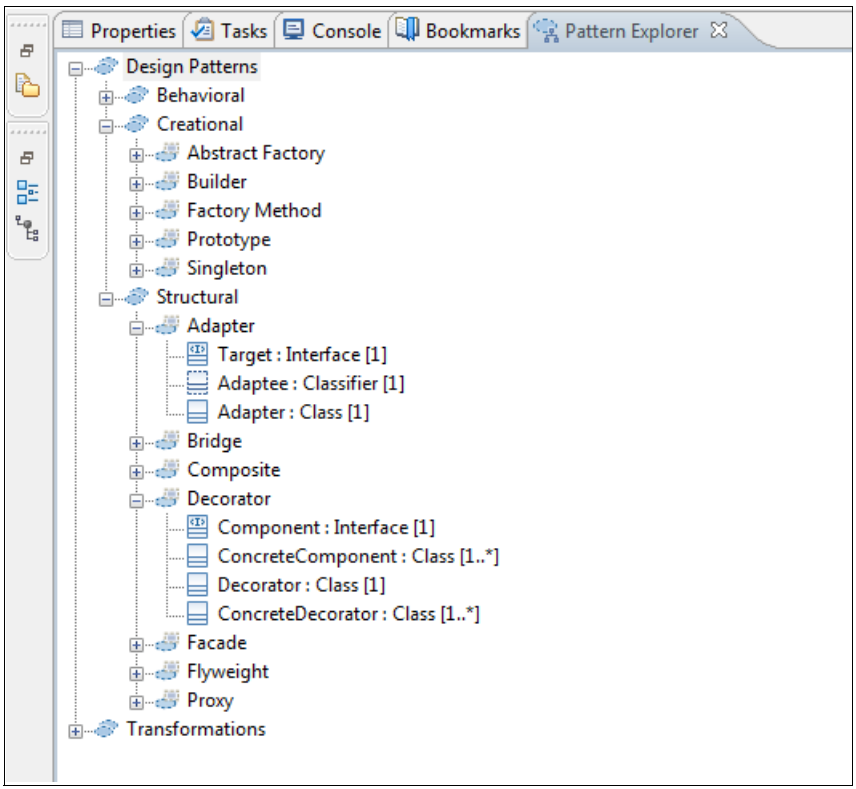

Figure 4-26 Pattern library

4.4.7 Change logs

A suitable format for protocol changes over the lifecycle of a project development is an important factor for a good design approach. Ideally, the impact of a defect fix must be directly deducible from the change log. But why do we need to maintain a change log in the first place? Should it not be sufficient to associate model and source code changes with a defect by means of the software configuration management system? The answer is that we cannot always use the version history. We already know one reason for this restriction. It is the potential lack of traceability if elements have been renamed. The other reason for not relying solely on the version history is that many of the actual changes might not affect the business functionality and might therefore not be associated with any defect or change request at all. A simple example of this mismatch is the graphical reordering of a diagram. While this action results in a new version of the corresponding source element, it is not related to any requirement. Hence, the change log provides a means of distinguishing between relevant and irrelevant changes and determining the actual scope of the changes. It serves as an input to release notes.

Another motivation for change documentation is related to the flow of information within larger projects. This applies equally to all cross-team boundary information flows. If you are working in a small team where the link between designers and developers is given at any time you might not consider this a problem. The developer identifies an issue, the designer fixes the defect, checks in the design, notifies the developer, and tells him where to look, and perhaps even which portions of the design changed. But there are several scenarios where this simplified and personalized handover will not work. Some of these are large, distributed teams (for example, on-site design versus offshore development) or design releases that contain numerous changes that are handed over as a single package. In the latter case, a variety of changes is possible. There can be entirely new functionality, which is probably reasonably easy to spot. Signature or flow changes in services that have previously existed can be more difficult to identify. Most difficult to deal with are deletions, because the deleted elements no longer exist. We need to identify them in another way, for which we rely on the change log.

Given these scenarios, we need a dedicated approach. While a version control system can only aid us in finding changes that were made based on a given task, it cannot resolve all the changes grouped by design elements, for example, a single service. Again, we could argue that it should be possible to list all changes made because the previous delivery and group them by affected element. This is true, but insufficient, because just knowing the element that has changed means that you have to compare current and previous design, as well as the current implementation. This is cumbersome and error-prone at least.

To overcome these problems, we recommend a structured change log as part of the model. Independent of the version control mechanism, information about each applied change should be captured, including, but not limited to:

- Author/contributor provides a reference for who can be contacted for details and further questions.
- Date of modification can be useful as an indicator, for example, if a defect is reopened and “fixed” multiple times.
- Scope information, if applicable. Depending on the project setup, designers can already be working on the future scope. Looking at a change log, it should be clear for which unit of work a change is relevant.
- Change trigger, if applicable, provides the number of a defect or change request. It forms a logical change set and thus helps with traceability, impact or effort analysis, risk evaluation, test definition.
- Details on change applied, selects a fixed set of terms to use for characterizing the type of change, for example, created, flow changed, signature changed.

All these suggestions are based on the most common requirements that we have seen. Each project has to determine its own parameters and ensure that they are maintained throughout the models. Based on our experience, the less free text and the more structured information we provide, the easier it becomes to process changes. Standard or custom-built tools can examine the change and react accordingly. It could trigger code generation or create tasks in a work flow system. For the people processing the actual change, fine-grained information significantly reduces the amount of time needed to interpret the new design and check which modifications have to be made.

4.4.8 Reuse

While reuse is well known in the field of programming, reuse in design models is less common. Nonetheless, reusing design blocks have the following benefits:

- Reduced risk
- Increased dependability
- Effective use of specialists
- Accelerated development
- Increased maintainability

Being able to reuse something that was designed, developed, and tested multiple times increases reliability and, therefore, reduces risk. Reuse addresses the following risks:

- Reduced risk in exceeding deadlines because reuse can significantly speed up the design after initial additional efforts organizing.
- Reduced risk in creating inconsistencies because the artifacts being reused have already proven themselves.
- Reduced risk in clarity because reuse simplifies the design.

Reusing artifacts means falling back on solutions that have proven themselves under certain conditions. Because they are proven, they are more dependable, even if we might have to adapt to the current environment.

The team consists of subject matter experts with identified areas of responsibility regarding the business areas, as derived from the domain model. Ensuring that these specialists bring their best to the team by providing elements for reuse is the best way, which then can be used through other business areas.

The logical consequence of proper reuse is less implementation effort. We also have to test fewer activities, and fewer errors will occur in reused components. Changes are focused on one small area that is being reused. The downturn is the impact that a single change on a reused activity can have. Still, all this adds to a reduction of effort, and therefore an acceleration in development.

By reusing dedicated methods, the overall model size decreases. Additionally, the impact of changes, like updates to the data model, will potentially only concern a small number of methods. This results in less effort for the designer to respond to change requests.

How to model reuse

The question that prevails is, how can we identify areas of potential reuse?

The following actions help identify areas where reuse can be applied:

- Identify generalizations.
- Identify generics.
- Identify reusable aspects.
- Identify multiple relationships.
- Apply common design patterns.

Generalizations are what they indicate, a means of abstracting reusable features and propagating them onto the reusing objects. Hence, every generalization represents a possibility to design for reuse when acknowledging these facts.

Identification of generics is an advanced topic with regard to UML design. While generic behavior can usually be identified on the design level already, it often turns out to be difficult to model generics adequately. UML supports the modeling of generics, but designers might not be familiar with the notation. Covering such a technical topic in UML models requires early involvement of technical architects.

Similar to generics, modeling aspects of the business logic is a topic rarely covered by business designers. Usually, we find these elements on lower modeling levels only, but it is important to consider them when setting up design guidelines.

Whenever an object carries multiple relationships to other objects, there is an opportunity to design activities to manipulate that base object. Other activities can then take care of processing the relationship-specific actions and calling the common base objects activities.

Internal services (that is, services that are only used within the activity flow of an exposed service) are usually good candidates for reuse as well. This is the case because they can be specially designed such that we can use them in the context of several services. Considering their granularity, small pieces of logic are usually much easier to reuse compared with complex business logic. It is important to remember that one potential drawback of reusing activities is creating a procedural, complex design. Also, make sure that reuse does not violate the layering rules of your application or that you end up with a design that is difficult to maintain.

Common design patterns provide additional options for reuse. These are a few examples:

- Aggregates are units of objects that are related to each other and that have a meaning together. There is one access point into an aggregate. Other objects are being called starting from that access point. In the case of our domain mode, this is the entity that provides access to the value types.
- Factories allow initialization of many objects to guarantee consistency between them. This typically requires calling methods on multiple objects. These methods are reused when initializing the factory.
- Lists provide information from objects that have other objects depending on them in x:n relationships. One example is objects that are time sliced and that have to hold multiple versions of the same data object over multiple periods of time. Methods for iterating through such lists or in fact reading or working with such lists have a potential for reuse. For example, time-based objects or attributes are distributed throughout the model, yet the access method will be provided by a base class and will be used by services for all different purposes irrespective the class that it is accessed from.

Reuse in services

An important aspect of reuse is the context-unawareness of components. Most importantly, we need to apply this pattern to the services that we will discuss later in this book. Services do not know the context of the caller. If you find yourself inside a service that needs to know things about the caller, revisit the use case or service granularity. Services should be reusable and loosely coupled, and including context-specific behavior violates this principle. Think about the consequences if the service needs to behave differently for just one attribute depending on which source system an object originates in. The service design and implementation need to be modified for each service consumer that might exist, which is simply not acceptable.

4.4.9 Sources for design guidelines

In this chapter, we only provide you with limited insight into design guidelines. The topic is simply too large and too specific to any individual project. Apart from the sources mentioned in this chapter and the dependencies on the requirements specific to InfoSphere MDM Server systems, you can rely on several other sources. Guidelines should comply with standards and specifications set by external organizations, committees, and architecture boards. They also should be synchronized with existing pattern repositories, tooling requirements, and company-specific business norms and terms.

4.5 General design concepts

This section provides examples of important concepts common to master data management and other information driven solutions. Note that this selection is far from complete, but provides a basis for the most frequently occurring conceptual issues in master data management solution projects. Trying to introduce and discuss these concepts later can cause interruptions in the project.

For example, almost every project requires a concept regarding error handling. However, error handling is often discussed when implementation has commenced. Then the design has to undergo a change to reflect the error-handling concept that is agreed upon. Adding such a basic concept to the other business logic that you have to implement is a major undertaking that can slow an entire project temporarily and that can result in missed milestones.

Because you need definitions for all these topics during the course of a master data management implementation or any other project, address these definitions well in advance. Dedicate a large block of time to an iteration and lay down all or at least most of these concepts. Many of these concepts are such that a project will depend on them irrespective of the business problem that is being tackled. These concepts apply whether you are dealing with customer data or product data. They do not stop at one subdomain, but apply to all subdomains. Address these issues and define all known concepts prior to encountering them during the implementation of a business problem solution.

Do not ask questions too late when attempting to implement the business logic of a business service. Irrespective of the questions that you ask, however, you need to address the following types of concepts prior to the implementation of a master data management solution:

- Base concepts
  - Error handling
- Domain concepts
  - Search
- Functional behaviors
  - Lifecycle
  - Rules
  - Filter
  - Time slices
- Meta concepts
  - Change log
  - Reuse
- Service contract
  - Business keys
  - Encapsulation
  - Provider signature
    - Request
    - Response
- Service implementation
  - Data extensions
  - Flow
    - Control flow
    - Object flow
  - Object types (for example, lists)
- Stereotypes

This list is not complete. This particular topic is too broad for the discussion in this book. This discussion focuses on some of these concepts to provide an idea about of most important concepts and their categorization and modeling aspects. In addition to the creation and definition of these concepts, it is equally important to verify the validity and applicability of them in the context of the project. Often you need refinements to existing concepts after the implementation has reached a mature stage.

Modification to concepts during the course of a project: Even with the best planning, modifications to concepts during the course of a project can happen. However, implementing miniature proof-of-concepts within the project provides the following project-related benefits:

- Concepts that have been defined are thoroughly tested and verified.
- Any additional insight gained during these tests can be fed back into the concept prior to any other design aspects using or depending on them.
- When discussing and verifying these concepts, you increase the awareness among all project team members with an early opportunity to identify and name subject matter experts for these topics.
- The awareness of the project spans to project management, which can then acknowledge the existence of such core concepts and ensure sufficient ramp-up time for creating and validating of these concepts.

The sections that follow discuss the following concepts in greater detail:

- Base concepts: Exceptions and errors
- Domain concepts
- Functional behaviors
  - Rules
  - Validations
  - Lifecycle
  - Filtering of data
  - The dimension of time
- Service contracts: Business keys
- Service implementation
  - Object creation
  - Loops and conditions

4.5.1 Base concepts: Exceptions and errors

This section discusses exceptions and errors, how they can be handled, and modeling guidelines for dealing with them.

This discussion uses the term *exception* to refer to objects that are thrown by the application and that are declared in the service signature. By contrast, this discussion uses the term *errors* in a more generic sense. From a developer's point of view, errors usually refer to non-recoverable situations. The idea of errors and exceptions is twofold. Exceptional behavior is defined as a situation that does not occur in the standard flow of an application. The reason for an unexpected outcome can be of a technical or business nature. The exception contains details, such as an error code and a message, that is resolved by means of the error code. Note, however, that messages should be resolved only in the context of the actor who receives the message.

But what are the differences between *technical* and *business* errors? If a technical error occurs, the user is usually not interested in the details of the error or the root cause of the problem. In this case, the application only needs to determine whether the exception is fatal and, therefore, cannot be recovered from or whether it is possible to recover from the error. However, in discussing exception warnings, be careful, because the technical error might be ignored.

The design flow of technical errors is usually as follows:

1. If the exception type is outside the scope of an application, it is handled immediately by logging the stack trace and wrapping it into the exception type that is provided by the application or framework used.
2. If the exception type is known to and handled by the application directly, it is simply passed to the internal caller.
3. The exception is declared in the service signatures and then passed to a generic handler that creates a corresponding service response that is returned to external callers.

This pattern then recurs throughout the entire application. Consequently, you do not need to model such behavior explicitly in each and every activity. Consider only the addition of the exception to the service signature in the detailed implementation design.

The handling of business errors is much more involved, both from the user's and the modeling perspective. First, you need to decide whether an application state is actually exceptional from a business point of view. If the check for a length constraint fails, you are confronted with an exceptional situation that can be handled rather easily. For example, the application throws a business exception, transforms that exception into the corresponding service response by a generic handler, and then passes the exception back to the caller. If the service was called by a front-end application, the user can correct the input errors and re-send the information.

However, the process gets more complicated when you are looking at operations that we refer to as *getBy services*. If you pass a technical key into a getBy service, you expect only one record to be returned. If you do not retrieve any record at all, you can reasonably assume no result to be an exception. However, what about a getAllBy operation to which you pass only a business key? In this case, you might not be sure how many records you will obtain. You might receive more than one record or no records at all. Here you cannot assume that receiving no record is an exception. This result might reflect that there is indeed no record in that list, which is also indicated through the definition of the corresponding list that specifies 0:*n* records.

Every operation state that is covered by the standard flow will not raise an exception. Services should not make any assumption about what a potential caller will consider exceptional. If in doubt, the services should not throw a business exception and should return an empty list instead. In the end it all comes down to precise specifications provided through good design and based on thorough business analysis.

There are more noteworthy aspects of business exceptions. As opposed to technical exceptions, details about business exceptions are regularly sent to the caller. However, dependent upon the number and complexity of layers, the original exception cannot be returned as is. This is the case because we have to provide more and more meaning and context to the exception before eventually confronting the business user with it. This is why one error code on an upper layer will usually be mapped to more than one low-level error code. They can also contain a collection of error codes, which is a common scenario for input validations. Instead of returning the exception right after the first validation failure occurs, an application might collect those failures before returning them to the user. By doing this, the user has the ability to correct all input errors at once.

The error handling provided might not satisfy your system's business requirements. In that case, you have to provide for a mapping between your expected exceptions and the ones that are thrown out of the standard product. This makes the need for such exceptions or error concepts even more important and relevant.

Because error handling is part of the business requirements, a simplified version of the handling will already be used in the use case specification. This means that exception outcomes must be marked in the analysis phase. Also, companies often have some kind of error repository in place, so we can expect to see these error codes in the analysis model as well. Note that the idea of error pools is a common one. Instead of keeping error codes scattered through the entire application, we suggest using some type of pooling mechanism. In the simplest form this is represented by a package that contains all errors.

Technical errors are not explicitly modeled unless there are requirements regarding their handling. We only work with business exceptions on the analysis and design level. The implementation model then introduces the required technical errors and considers them in the service flow. The implementation model is also the right place to model the actual transformation of internal error codes to external ones. This transformation is a recurring pattern and should be based on some kind of template. This is even more important with regard to separation of flows and aspects.

Because error handling is commonly considered an aspect of a service, it should be kept separated from the flow where possible, as illustrated in Figure 4-27. Think of it as making the model easier to read and understand.

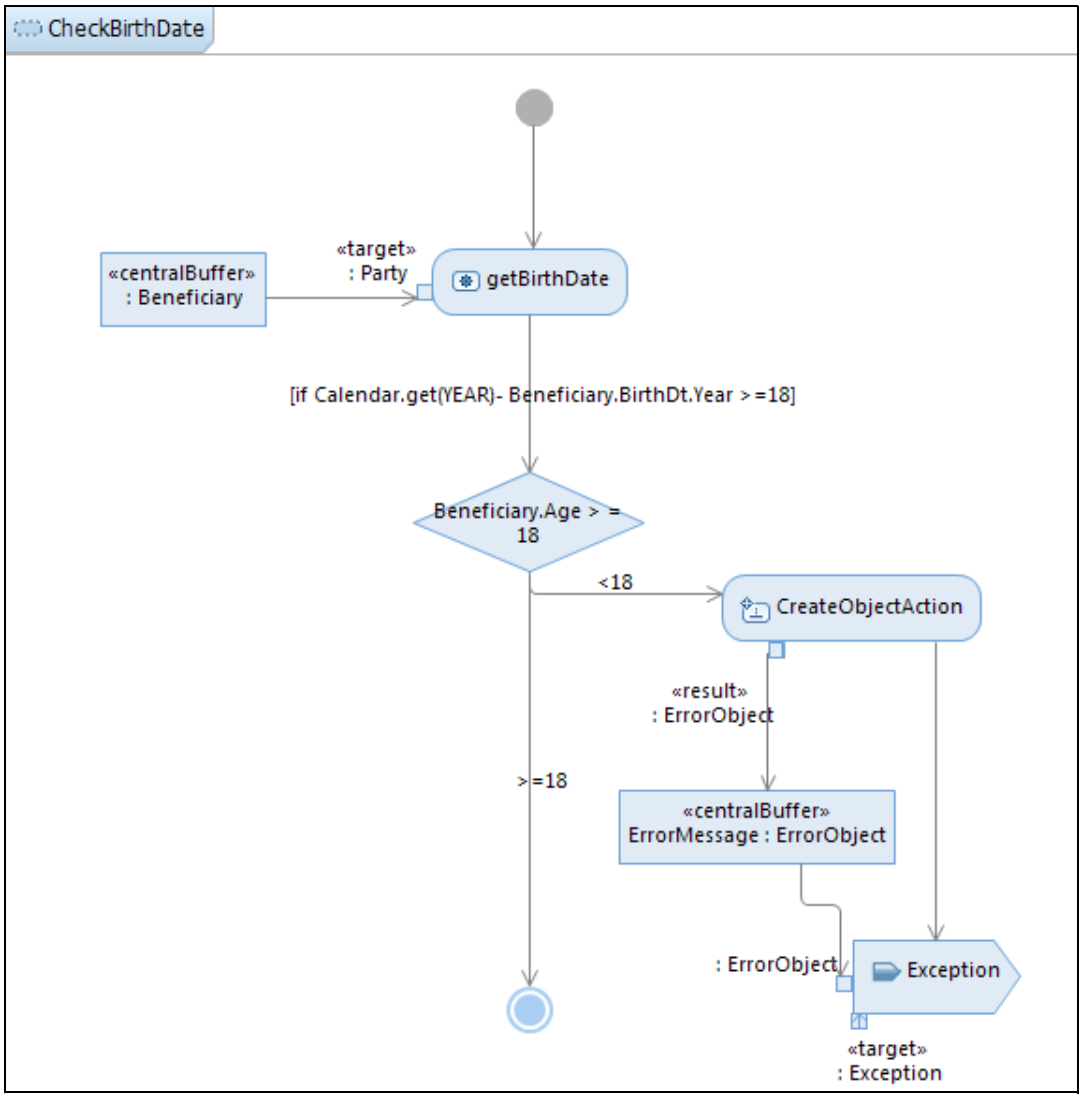

Figure 4-27 Schematic use of error signal

Unfortunately, UML models do not provide an easy means of modeling aspects other than using specific stereotypes, which must then be associated with a service by some convention. Use a condensed notation within the flow to address this deficiency.

As opposed to exceptions, warnings will not be modeled as signals but as “standard” output pins. In most cases, it will not be necessary to say anything about warnings in a specific use case unless you need to provide a customized warning message. Because warnings usually do not require any use-case-specific modeling, we suggest designing their handling within a base activity. This behavior can be injected into the activity instance at hand by means of a stereotype. With this stereotype, you can also provide context-specific information, such as the warning code and the localization. This approach avoids overly detailed activity diagrams and redundant modeling.

One last comment about collecting errors. Although it would be entirely possible to use error signals in this scenario too, it makes sense to treat the collection similar to what we said about warnings. So, instead of explicitly throwing a validation error, use a customized output pin, as indicated in Figure 4-28.

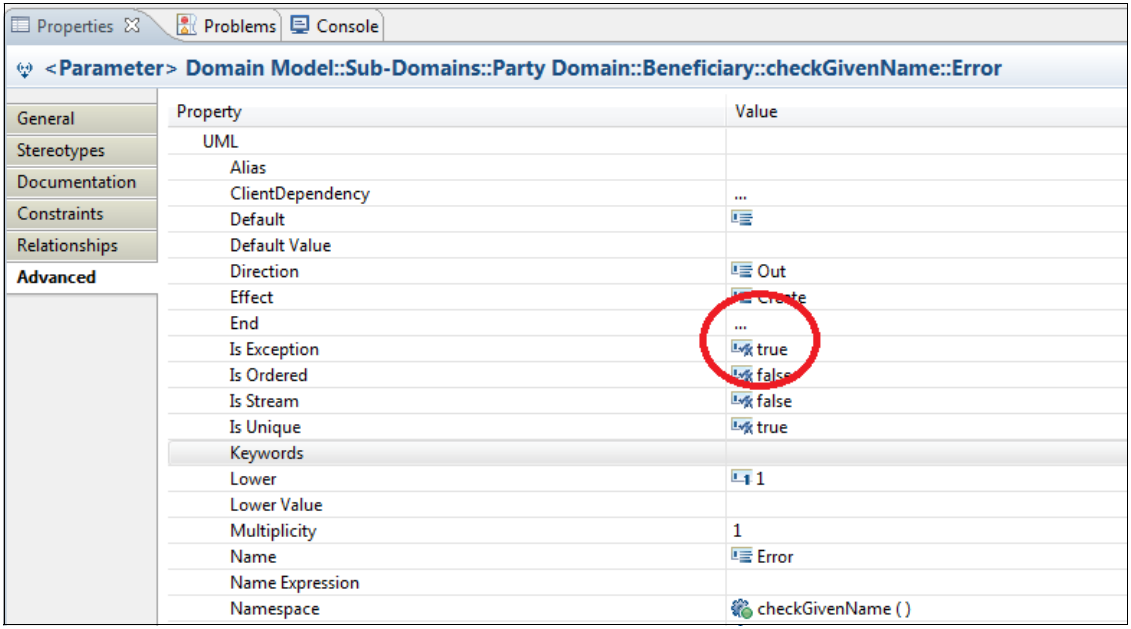

Figure 4-28 Setting the isException property

4.5.2 Domain concepts

This section discusses *search as a domain concept*. Searches are a relatively simple construct. They are based on their own request objects but return objects from the domain. Consequently, dedicated search objects are required. InfoSphere MDM Server provides a model that you can use as a baseline, as illustrated in Figure 4-29. The transaction reference guides reveal this distinct search object structure that you can use to model and refer to these objects on search operations.

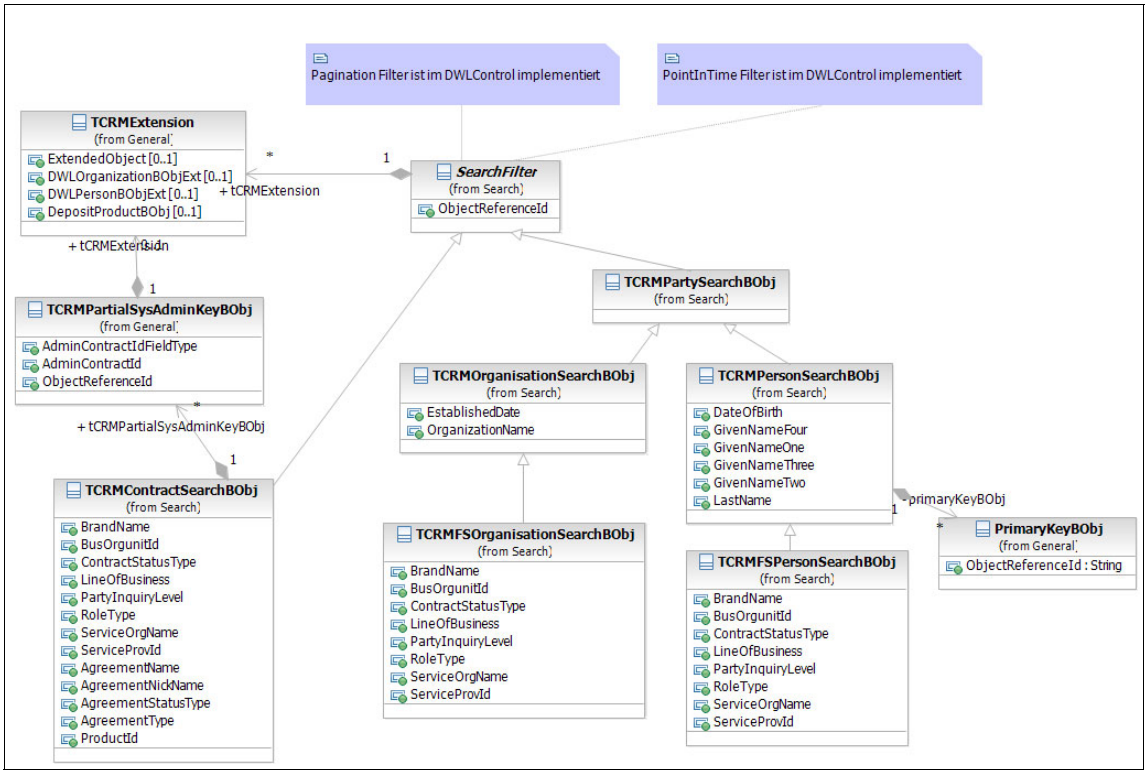

Figure 4-29 Master data management search object model

In addition, you have to deal with the corresponding search service operation. First, mark this operation with the stereotype `<<search>>` and then provide documentary attributes in the form of stereotype attributes, as shown in Figure 4-30.

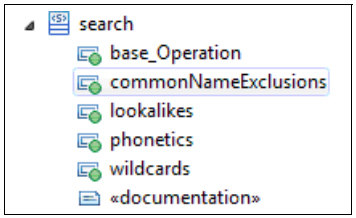

Figure 4-30 Definition of the stereotype search

The use of a stereotype allows you to explicitly declare a search service operation that makes sense due to an entirely separate functional execution compared to `get()` or `getAll()` types of operations. Using the stereotype-based attributes again allows you to describe behavior in a consistent way for each search service operation, while still providing distinct information for each search.

4.5.3 Functional behaviors

In this section we discuss *functional behaviors* that are associated with the following operations:

- Rules
- Validations
- Lifecycle
- Filtering of data
- The dimension of time

Rules

In a broader sense, *rules* can encompass anything starting from consistency checks to flows that depend on a condition. This interpretation makes it difficult to understand the nature of rules and how they should be handled in the design model. Rather than adhering to the broad definition, consider that rules should denote only the following concepts:

- They must be independent of the actual flow.
- Their business lifecycle must be independent of the flow.
- They must contain business checks and conditions that are beyond the verification of length restrictions.

- They must return true or false or throw an exception instead of returning true.
- They must not change data.

Ideally, rules can be applied directly to a domain object. Just as these objects can contain any kind of consistency checks and validations, they can also encompass business rules. However, constraining rules to one and only one object will not be sufficient in many cases. Sometimes more than one object will be involved in the processing of a rule or the rule will work in a different way, depending on the service it is run by.

In the first scenario, some objects are value types and exclusively depend on the primary object. For example, if a person's name cannot exist without a person then it is valid to say that rules touching a person and their name belong to the person. However, there might also be scenarios where business objects that are related to each other, but that are not hierarchically ordered, interact within one rule. These rules must then be moved to the next higher level, for example, the services level.

In the second scenario, a specific rule can be executed only if it is called in the context of a specific service. A simple example for this scenario is a rule that can be run only in an add operation but not during an update. Although this distinction mainly applies to simple checks, such as the constraint that certain attributes must be present when a record will be inserted into the persistent store, rules might also be affected by the operation type. Again, rather than attaching a rule solely to the object, it also needs to be aware of the actual operation.

Rules are not tied to the point in time when they should be executed. Rules simply check conditions for a particular context of data. The result of a rule is always either true or false. Based on that, rules can be executed at these times:

- Right at the start of a transaction
- Before a certain step of a transaction
- After a certain step of a transaction
- At the end of the overall transaction

If the rule belongs to a particular step of the transaction it will likely be processed in the context of this *subtransaction*. Usually, rules require a business context. For example, a rule that will be fired after a deletion has been processed cannot be executed before the record has actually been deleted. Alternatively, if a rule must be fired depending on the number records available, it might refer to the persistent state before the actual add/update operation is executed.

Rules must be modeled as separate activities that are attached to an object or a specific operation. A rule should be marked with a specific stereotype to make sure that it can be distinguished easily from normal activities. A context object must be passed to a rule, and the activity design should be specific about the objects that we pass into the rule and use the typed operation parameter. You must distinguish between pre-execution and post-execution rules. That is, the design guidelines must contain a corresponding naming convention or a separate stereotype for each of the possible scenarios. An example is provided in Figure 4-31.

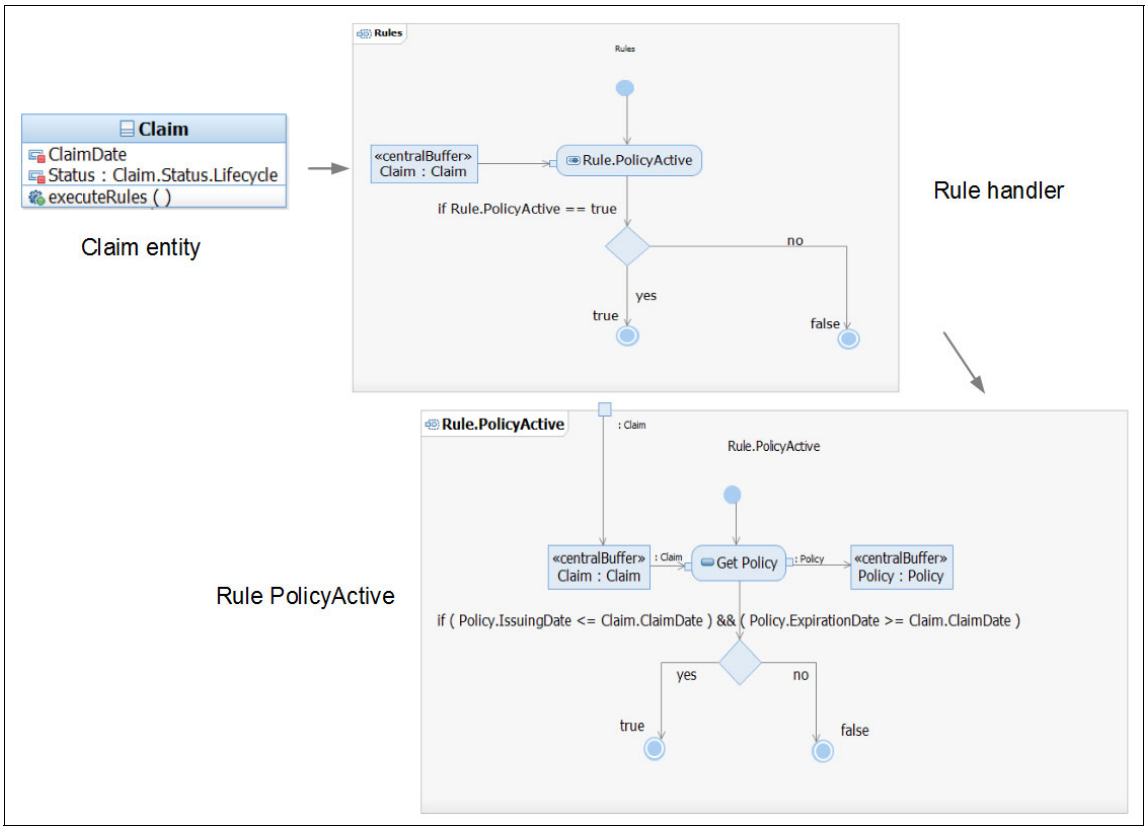

Figure 4-31 Sample rule attached to the claim entity

All of these elements are present in both the implementation and the analysis model. However, the analysis model will not contain details about how the context is passed into the rule and how the outcome will be handled by the application. This is similar to error-related guidelines where the possible outcomes will only be marked.

Depending on the requirements, it might be easier to use a simplified version of stereotyping in the analysis model. Instead of distinguishing between rules and validations, we can use one stereotype only to denote this kind of activity. However, it is nonetheless necessary to clearly separate common flows from rules in the analysis model already to avoid the situation that standard activities become indistinguishable from specific rule activities.

Validations

Validations or *checks* belong to domain objects. They are modeled as pre-conditions, post-conditions, or body constraints. Some simple constraints, such as length limitations, can be directly attached to the corresponding attributes. These constraints should also be present in the data model already. However, most constraints are much more complex.

The *Unified Modeling Language* (UML) provides tools to model constraints as *Object Constraint Language* (OCL), as free-form text, or Java expressions. Unfortunately, that does not match the requirements of designers or developers in our experience. Instead we suggest that you rely on activities to model these constraints. This approach makes it easier to understand and trace model constraints. In addition, it restricts you to only few model types and, therefore, does not overload the project with too many UML base model types that all require different interpretation.

Validations must be stereotyped and named accordingly. In general, each operation should be divided into the following parts:

- One part that encompasses all pre-conditions
- One part that defines the body, including body constraints
- One part that contains post-conditions

The major issue regarding constraints is that they are not context-free. For example, if you add a name to a person's record, it is likely that the last name is a required attribute. However, in an update operation, you can leave out the last name if you just want to change the person's first name.

Another example is based on creating a party relationship, which requires two parties. Neither the add nor an update of the same relationship might affect any of the two related parties. Those parties are still required in the service provider signature though, because we have to use them as reference points.

Even multiplicities change depending on the service operation. Although this scenario can still be covered by setting the multiplicity of the corresponding input parameter, this approach fails when the multiplicity constraint does not apply to the input parameter itself, but applies to an underlying attribute. A constraint can also refer to the type of object that is passed into the service. This type is visible only in the actual activity instance but is not necessarily visible in the operation signature.

Constraints depend on the context and the state of the application. They are not static, but are dynamic by nature, which can contradict the idea of service contracts. Thus, both the static service contract and the dynamic service behavior descriptions are important. Be aware that following this pattern also means to that you have to distinguish between type-related specifications, such as *Web Services Description Language* (WSDL) descriptions, or class relationship models and dynamic specifications, such as object diagrams.

Modeling validations is a task that occurs in the analysis and the implementation phase. The only difference of significance between the models is the level of granularity. Similar to rules, validations do not contain any details about the context and outcome handling in the analysis model yet. Instead, they are modeled as separate activities.

Lifecycle

Similar to the business rules, you do not want to model changes to the *lifecycle* of an entity directly into the service flow. We consider lifecycle changes *state transitions*. Consequently, we model them as state transition diagrams. The key challenge in respect to the definition of state diagrams lies in the fact that use cases must not repeat, reinstate, or contradict the definition of state transitions. The flow aspects across all modeling artifacts must be complementary. We increase readability when we assemble all state (or more specifically lifecycle-related) information solely in such a state diagram. We are not required to describe any related logic anywhere else anymore.

We explicitly store the state of an object in a status attribute. Changes of such an attribute are triggered by signals, calls, change events, or time events. The lifecycle listeners are attached to these events. We exclude direct calls to the state machine in the modeling approach because that would contradict the principle of loose coupling. The change of the state can in itself trigger yet another event. This could be required when the change of state must be propagated to other entities that then are also affected in their state. To avoid endless propagations, this behavior should be limited to those scenarios in which a clear hierarchical relationship exists between one trigger and its successor.

Figure 4-32 provides an example.

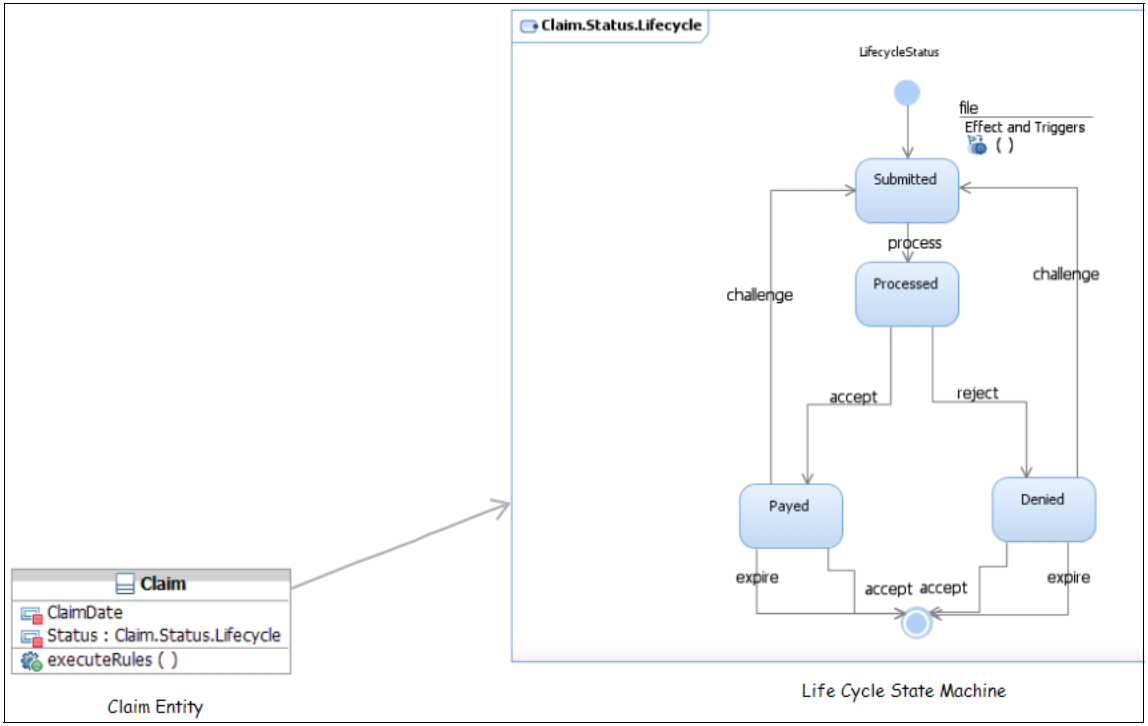

Figure 4-32 Sample lifecycle state machine attached to the claim entity

In the state transition diagram, the states can and should be related to the actual attribute of the object in question. We also have to ensure that the state diagram belongs to the object that contains this state information.

Note that these guidelines must be applied to both the analysis model and the design model. The main difference between the analysis and the design phase is the granularity of what is being modeled. The elements that are used are the same.

Filtering of data

Filters are specific search objects that serve to limit the search scope. Usually, searches will return lists of objects depending on the criteria that the user entered. Filters can contain one to many attributes. If more than one attribute is contained, each should be filled or additional attributes must clearly be identifiable as optional criteria that only serve to limit the search result provided by the main filter criterion.

For example, this approach is valid with regard to names, where the last name would be the main criterion, and first names and titles only help to restrict the search. This guideline prevents ambiguities regarding the search scope and the priority o the criteria.

Search operations must be started with the keyword search and are also marked by a stereotype. Apart from search operations, getAllBy operations can also accept search filters as input parameters. Note that search filters are used exclusively. By default, it is not valid to combine standard business objects and search filters in the same operation signature. If several levels of searches must be supported, the operation signatures should be overloaded rather than being extended. For example, if a search for a person can be based either on the last name only or the last name plus tax identifier, two search filters should be provided instead of one (with the tax identifier being an optional attribute). Of course, it is absolutely valid to establish a filter hierarchy by inheritance to avoid duplicate filter attributes. Having said this, we must emphasize the fact that filters are not domain objects and should otherwise not be treated as though they were.

Filters belong to different search domains. We mentioned in the discussion about business keys that these keys mainly serve to provide a certain view of the data. In a similar vein, search filters also act as view substitutes. The difference between key-based and search-based views is the form of the result. Although we can reasonably hold that key-based views encompass valid and complete business objects, this might not be the case for searches. So search results will not consist of domain objects, but customized response objects.

You can expect to find the following typical scenarios for filters:

- Status filters

  Objects are filtered by an object status, usually the lifecycle status.

- Filters by name

  Objects present in this view are matched by name, by parts of the name, and so forth.

- Filters containing relevant attributes

  An attribute is considered relevant if it helps limit the number of matches. A person's birth date is a relevant fact. However, a person's gender is not helpful but might be used as a third or fourth criterion.

- Filters by business date

  Let us assume that an object can be active, passive, or historical. To distinguish active and passive records, master data management systems rely on business dates. For example, if the start date of a certain record lies in the past and if the end date of the same record is either not present or lies in the future, it is considered to be active.

- Filters by technical date

  Technically speaking, these filters are known as point-in-time filters. That is, records that are filtered by technical dates represent a *snapshot* at some point in time. This snapshot, however, might not contain entirely consistent data, depending on the state of the system at that time. We discuss the dimension of time in greater detail in “The dimension of time” on page 226.

The analysis model contains information about searches. It also provides information about the search criteria. The actual assignment to filter objects does not occur on this level. Their introduction is up to the implementation designs that are transferring the search requirements into an adequate request and response structure.

Create a *stereotype filter* and add it to the operation type. Add in a stereotype attribute named `filter`, as depicted in Figure 4-33. This name was used because it matches the name on the tcrmParam specification on the InquiryParam object.

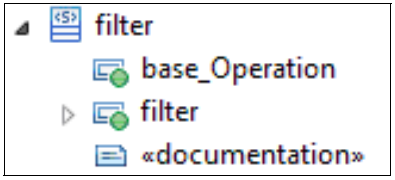

Figure 4-33 Definition of the stereotype filter

Create an enumeration for this filter and add the literals ALL, ACTIVE, and INACTIVE, as shown in Figure 4-34.

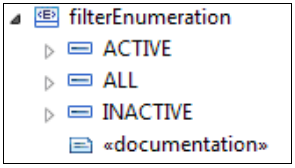

Figure 4-34 Definition of the enumeration filter

The advantage is that you have a large description field for each literal and are, therefore, in the position to describe what each means in great detail, as shown in the sample depicted in Figure 4-35. Now bind this FilterEnumeration to the Filter stereotype attribute and specify the default if you like.

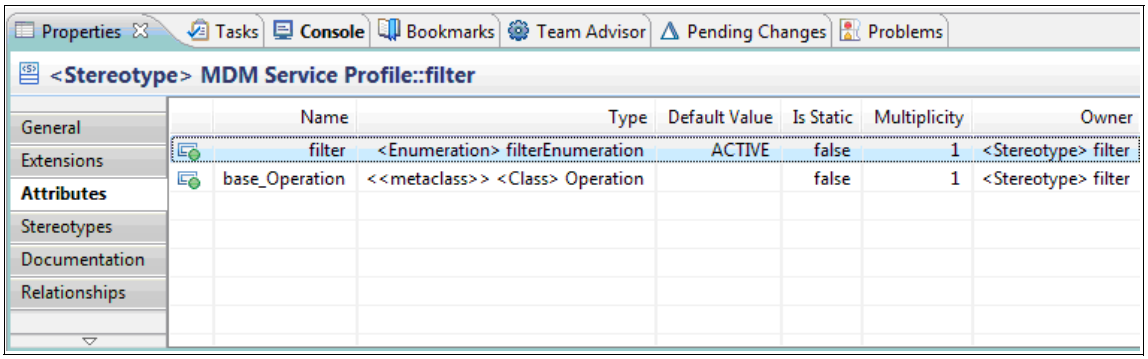

Figure 4-35 Sample filter stereotype applied

Now, after publishing the new profile and adding it to your project, you are ready to apply the Stereotype to your own `get()` operation.

The InquiryLevel is a little more tricky because a hard binding to any given enumeration, as outlined previously, is not possible due to this only being set at run time by the service operation. It is always the same values entered (for example, 0, 1, 2, and so on), but it always means something entirely different, such as returning all party-related objects or returning all contract-related objects. However, there is a similar solution. Create an inquiry stereotype, associate it with the operation type, and add an InquiryLevel stereotype attribute to it, as shown in Figure 4-36. Specify a multiplicity of * and use the type String.

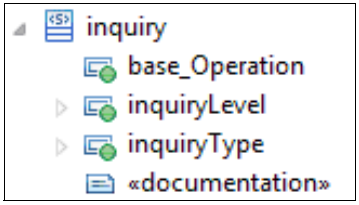

Figure 4-36 Definition of the stereotype inquiry

Now you can bind this stereotype to the operation and, instead of specifying the literals on the stereotype enumeration as was done before, you can now specify the literals on the stereotype attribute on the service operation, as shown in Figure 4-37.

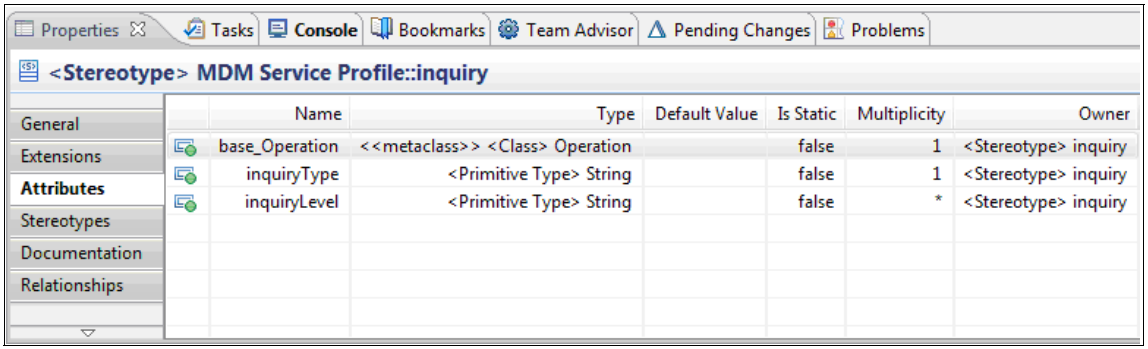

Figure 4-37 Sample inquiry stereotype applied

This method ensures that every `get()` service can implement its own values while still being able to describe these values, at least with a short description in that string field on the stereotype attribute.

The dimension of time

In the context of information-driven systems, the *dimension of time* is an important part of the system. Thus, you need to include it in your modeling. Unfortunately, adding the dimension of time is not an easy part of the modeling. This concept should be given sufficient ramp-up time. You will struggle to implement such a complex topic when you adjourn it at a later point in time, or even decide on introducing this concept at a later phase in the project.

Consider this concept to be one of the key concepts to be addressed right from the beginning. Try to keep time-related modeling separate from the usual domain model and other modeling concepts. The following aspects of time are described in the subsequent sections:

- Validity
- Effective dates
- Defaults
- Multiplicity
- Consistency
- Dependencies
- Granularity
- Splitting or coalescing
- Filtering

Validity

The idea behind establishing time-dependent behavior is to define *validity* constraints. The validity is defined by means of two fields: the start date and the end date of a record. You encounter a number of rules. For example, there is one rule saying that the start date must always lie before the end date. A record can either be active or inactive. This depends on whether the reference time lies within the time period established by the start and the end date or outside that time frame. A time slice that is currently active does not have any end date at all or indicates this through a well-defined end date, such as 31.12.9999.

Alternatively, time refers to the technical time stamp of a record. The meaning of that is when it has been added, updated, or deleted. In a typical master data system, each and every change of a record will be recorded such that the entire technical change history of the record can be traced. If a record has been deleted or updated, a technical end date will be set to mark the deletion of the old status. This type of history can be used for audit or research purposes. Users that access the history have to provide a specific filter criterion, which is either the point in time when the record was created or changed, or a time period in which the modifications have occurred.

Depending on the requirements, you might decide not to use this kind of history at all. Some companies already have other auditing systems in place or just want to prevent the performance impact of a technical history. From a modeling perspective, there are two aspects that are noteworthy. Persistable objects should have a stereotype denoting whether they support technical auditing. Second, there might be specific searches on the technical history that should be designed such that the point in time corresponds to the search criterion.

Although this approach seems simple enough, it has some drawbacks. Using a technical history might not satisfy a business user's needs. It provides only a snapshot of a specific point in time regardless of whether this snapshot is consistent from a business perspective. Also, the business user cannot actually influence what is going to be stored to this kind of history. Regarding consistency, instead of storing the individual time stamp of every database access separately, you could assign one time stamp only to each transaction. By doing this you can guarantee transactional consistency of what is provided by a search query to the technical history. Still, this approach does not resolve the business issue. So, apart from the technical history, you need another type of history. This is referred to as the business view of time or effectivity date.

Unless otherwise specified, the reference time will be equal to the current time stamp. This means that we look for all time slices that are currently active. Alternatively, and depending on the service signature, a caller can provide explicit ranges. This range can be expressed either as an as-of time stamp or as a from-to range. In the first scenario, the reply will contain all records that are valid as of some date, whereas the second scenario refers to both the start and the end date of a record.

From a domain perspective, it does not make any difference whether the start date is a date in the past or the future. However, it might have some impact from a technical point of view. For example, InfoSphere MDM Server does not currently support adding or filtering future start dates. Having said that, we also think that the domain model should not enforce this restriction unless required by the business. In a real-world scenario you will probably see future end dates more often if you think of temporary contracts or addresses.

Effective dates

Business dates refer to the validity of a business attribute. However, there are scenarios in which validity is not clearly defined. Assume that a customer informs the bank that the customer's home address will change soon and that the new address will become effective on a specific date in the future. A data steward would possibly enter two dates into the system:

- The date on which the bank has been informed of the change
- The date on which the change actually becomes *effective*

Thus, effective dates add another layer of complexity to the dimension of time concept.

Think a moment about what a business user might require with regard to the dimension of time. If the address of a customer changes, this address must be changed in the database as well. However, deleting the address is probably not a good idea because it might be used by some other customer as well. So instead, a master data management system will terminate the record that associates the customer with this specific address, which means that the association holds both a start date and an end date. The termination merely sets the end date of the record.

The concept of “slicing” a business record by time into different pieces is known as *time slicing*. The collection of time slices belonging to one business attribute or object is called a time sequence. In this specific case, we might argue that we could still go ahead and delete the association instead of terminating it, even if we cannot delete the address directly. This only is valid approach in those cases where we are not interested in the business history, where it does not make any sense to track start and end dates for a specific reason or if the system we design for is not in charge of the record. Usually, the concept of time will not be applied to every domain object, but only to some of them.

Business times go along with additional characteristics regarding whether they are active. First note that the use of only two status values implies that we cannot differentiate between “not active yet” and “not active anymore.” This is a distinction that we ignore because it is usually not needed. In a time sequence (that is, the number of time slices referring to the same business attribute), there can be only one active slice at any given point in time. Naturally, this constraint does not hold if the business attribute itself has a multiplicity of greater than 1. This is true if there is no distinction between several items of a list of this attribute. An example of such a business attribute is a person's nationality. If a person has more than one passport, there is no natural order of his nationalities. In this case it is valid to say that more than one instance of the same attribute is active at the same time. The easiest way to reflect this behavior in the data model is to relate the time-dependency of nationalities to the entire list of nationalities rather than to each record separately.

Unless specified explicitly, a record cannot have any active slice at a point in time, for example, if the time sliced attribute refers to a relationship, such as whether a person is married. If not, there is no active relationship between this person and another one of type marriage. We can debate this from a business perspective and whether time slicing is really required. The one aspect that we cannot debate is the need for the effectivity dates.

Defaults

Defaults surface throughout the entire domain model. This is also true in respect to time. You need to understand the meaning of each of these dates to drive the right decision from it. Some systems use a specific date to identify a specific condition. For example, the first of January 1970 is used for organizations that did not provide the date of incorporation, or the first of January 1900 is used for people whose deceased date is unknown. Some or all of it should be revealed through profiling. Even if these are stored and used in the source systems, we do not have to feel obliged to use them in the master data management system. If the requirements state that such an attribute be mandatory, then rework the requirements instead of inventing or taking over an arbitrary default date.

In the worst case, they are ambiguous and lead to issues with rules or other constraints. You will be better off investigating and defining the right meaning for it. Often we can then also operate without such dates.

There are meaningful default dates as well. It is valid to assume that the start date of a contract role is equal to the effective date of the contract, if the holder of the contract already existed in the system. Although not every date in the system has a default date, the start date of a time slice usually has one. It relates to the part of the process that creates the time slice and can therefore use either that time or be derived from another date provided by the caller.

Relying on default values also has implications. If the caller does not provide the unique ID of the slice, updates and insertions both result in a new time slice that starts with the default date, which is usually the current date.

The same holds true with respect to end dates. If no end date is specified, it is not set at all.

There is another semantic definition that states that if the end date is not provided, it is not known to us. If we are aware that the current time slice should be open until we are advised to change it again, then we could also use the common default 31.12.9999. The only valid reason for using a default end date is a technical one. If the database used does not support index scans on `is null` clauses, then providing no default end date might raise performances issues. Because this aspect is technical, it should not be part of a business model. Consequently, it should only be added to a platform-specific model, which is represented by the implementation model in this case.

Multiplicity

The *multiplicity* of an association changes because one business record is suddenly related to more than one association record. Where there might only be one instance without the consideration of time, we might end up maintaining a list of instances that resemble the time sequence.

This is a major point of confusion in respect to the design of a system that makes use of time slices. It is also a practical reason to keep the time dimension separate from the raw domain model that does not incorporate any time-related aspects. The benefit is such that you can distinguish between real multiplicities and those only caused by time dependencies.

Consistency

A *consistent* time sequence is non-overlapping. It can only contain gaps. If a rule states that a customer always has to have an address and the address is time sliced, then this relates to a gapless time sequence. There are, however rules, that state that a party-to-party relationship such as a marriage might not always be available. This could be because a person might be married repeatedly but not continuously. In this case the time sequence can contain gaps.

Dependencies

Multiple attributes that depend on the same time sequence are subject to the same time slice handling. It becomes less clear which time-sequence-dependent attributes actually belong to if entities or attributes are related to more than one time-dependent entity. Often, these attributes are also not directly related in the domain model but possibly spread out over entities that are not directly related, only indirectly related through one or many other objects. This spreading out of attributes makes it even harder to spot dependencies and requires clear definitions by the designers.

We provide an example of such a dependency issue in this scenario. Assume that an attribute is part of the association between a party's address (that is, the link of this party to the actual address) and a party's role in a contract. This means that we consider the attribute to be part of exactly this role-dependent location information. An example for this might be a flag indicating whether the customer has agreed to the reception of product offers for the specified contract at the specified address. The information depends on two time-dependent entities, the role of the party and the address. How can we decide which one defines the time sequence? In some cases, this implies that we create a new time slice whenever either of these two entities changes. In other cases, these independent changes of a time sequence can raise confusion. This is why we need to define the behavior both on a flow level and the domain level.

Granularity

Time slices can vary in *granularity* depending on the business context. A common scenario is to create time slices on a day level only. If so, it is important to normalize start and end dates such that they correspond to the start or the end of a day. There can be two options depending on your definition of time periods. that is, whether the end date is included or excluded. Even if business rules only require day precision, it helps prevent inconsistencies to comply with this guideline.

Splitting or coalescing

There might be situations in which two adjacent time slices actually represent the same information. This situation can be caused by the insertion of a dependent attribute that causes a split of the related time slices. Determine whether you want to remerge two time slices with the same attribute value. You can merge them without losing data. Whether you coalesce time slices does not make a difference from a functional point of view. Just be aware of the additional overhead that splitting or coalescing adds to the application.

Filtering

Using *filters* is common in the context of time. Time filters include both technical and business filters. *Technical filters* refer to the technical time stamp of a record that is also known as the “Known as of” time. *Business filters*, in comparison, refer to the date of business effectivity. These two filters come in these types:

- From: Start of a time period
- To: End of a time period
- AsOf: Point in time where a record is considered active

First, separate time-dependent behavior. The business analysis model should be modeled merely with an indication that time is required but does not reflect the cardinalities for time. Instead, this model focuses only on the true cardinalities that are required by the business functionality. The conversion that changes the cardinalities under the aspect of time is added in the design model.

The indication for time on the analysis model can either be made by using a separate model that only contains time-related associations and constraints by modeling time-dependencies as an aspect of the domain model or by using specific stereotypes. We prefer on the use of stereotypes on the business domain model. This is named `<<timeBased>>`. In addition, we are going to transform, based on that information, into the corresponding design model, which we would do even without the consideration of time.

Modeling time as an aspect means designing either a signaling pattern that corresponds to the aspect or interception mechanism or a method wrapper. Model either of these patterns in a base type to separate business flows from the handling of time slices.

Any activities that are, for example, part of use case and that require special attention regarding time slices should also be modeled separately. How we model the technical handling of the time slices in the design and the implementation models depends on its representation in the master data management system. In most cases, a time sequence will consist of a list of records that have a common key. Although using successors and predecessors is actually more intuitive and less confusing, it does have drawbacks, such as the mismatch between linked lists and relational data models or the ambiguity that the approach has regarding gaps.

4.5.4 Service contracts: Business keys

This section discusses *business keys* and modeling guidelines relative to service contracts. In general, this discussion differentiates between business and technical keys. *Technical keys* are not known externally and correspond to unique identifiers of one record. That is, technical keys are usually primary keys of database records. *Business keys*, however, refer to a business entity that differs from the technical database record. In a sense, business keys provide a means of creating a specific business view of a business entity.

Business keys are known to the business world from the perspective of a master data management system or at least are used within the context of the system landscape. Business keys often correspond to reference or native keys provided by other systems. Exclusively saving business keys to the master data management system means applying the reference usage type of master data management.

Because business keys are not maintained by the master data management system, tracking the lifecycle of these keys becomes a major concern. You not only need to know how unique business keys are, but you also have to make sure that this notion of uniqueness does not change over time. If it does, you have to find other business keys that provide the same business meaning. But let us first look at the uniqueness itself. Usually, in a real-world business scenario you have to store different business keys with different meanings. Although key A might denote a certain type of contract, key B might identify a natural person. Key C again might stand for hierarchical relationships between stakeholders. So, although these keys partly cover the same master data management entities, they are different from each other. Considering this example, you can reasonably see that the requirement that business keys might not change over time depends on the data selection.

Systems providing a specific business key can decide to reuse keys after a certain time, for example, if the range is insufficiently large. Another consequence of business keys is a change of search scope, if, for example, a user enters a contract-related business key into a front-end application. The user can reasonably assume that she will get back all the parties related to this contract. Dependent on the search depth, a service might provide additional information such as addresses, names, and so forth.

However, what about further contract relationships that one of the parties returned holds? Should a search by a contract key 1234 also return the contracts 2345 and 3456 of a party just because the party also has an active relation to contract 1234? The challenge is to define an adequate search scope. If it is too narrow, the application will produce too many database queries. If it is too wide, a single query will take too long. A reasonable approach is to limit the search scope to the actual, consistent, and non-contradictory view that the key provides of the business. Consequently, a contract key 1234 should not return other contracts such as 2345 or 3456.

Another good indicator for searches is business domains. If you start a search within a certain domain, you do not expect the search to return anything about relationships that originate from other domains. As a simple example of this situation, assume that you want to rent a car in certain city. You begin by looking for car rental companies in that city. After deciding on a car rental company, you next select an available car at the selected location. In this search, you would not expect to see a list of available cars at other car rental companies in the area because you previously narrowed the search to exactly one company. That is, you searched the *geographical domain* before looking at specific cars.

Of course, this is a simplistic view of search behavior. It might also be possible that the search is less focused and, therefore, providing alternative car rental companies at some point might be a valid approach. However, master data management systems should not default to yielding as much data as possible.

Another important characteristic of business keys is that they are used only once in a search. After a user has entered such a key in a front-end application, the key will be used to look up one or more technical keys. Only these keys will be used in further service calls because looking up records by means of a technical key is generally much more efficient than using the business key over and over again. Naturally, the use of technical keys requires that the search has reached the layer where technical keys are actually known to the system. Usually, an application-based orchestration layer does not know anything about technical keys yet, whereas the basic services layer is aware of these technical details.

Hence, the first call to the base service will yield a technical key that will then be used in further steps of the search without being explicitly dealt with in the upper layers. This approach implies some degree of abstraction regarding the handling of keys. Technical keys are not dealt with explicitly, but only by means of business objects or data transfer objects passed between different layers.

We consider business keys to be entry points to business views. This means that it is, in general, not possible to access information with more than one business key at once. The reason is that the application cannot decide which view to take. Although we could establish rules of priority, this would be more confusing than helpful. Nonetheless, rules of precedence are needed whenever users combine business keys and other search expressions in searches. If a user searches by last name and a contact equivalent, the latter one has precedence over the former one. Also, technical keys should take priority over business keys. In general, designers should avoid situations where potentially conflicting keys are present in a search or get operation. If there are views that require the use of more than one business key, you need to clearly state their roles and precedence. These views should go along with specific queries that accept exactly the parameters needed to create the view.

Although the concept of business keys can be complex, modeling them is comparatively easy. Objects that serve as business keys will be marked with the stereotype <<`businessKey`>>. Technical keys will not be defined in the analysis phase yet. However, in the design model, some of the technical keys might actually be exposed in the service signature and, therefore, it makes sense to provide a stereotype for technical keys too. We name it <<`technicalKey`>>.

4.5.5 Service implementation

This section discusses *service implementation*, in particular object creation, loops, and conditions.

Object creation

Although *object creation* seems to be a simple concept, it requires a clear definition and accompanying guidelines to avoid overly complex diagrams that contain repetitive, infrastructure-related, and glue activities, such as CreateObjectActions. The term *creation* can also be ambiguous. Does *creation* mean that an object is instantiated but not initialized? We hold that this is the case. That is, if you want to set default values and initialize instance values, these tasks are outside the scope of object creation. Rather, you need another stereotyped action for this kind of activity. We suggest calling it <<`initialize`>>.

With regard to simplicity and understandability of diagrams, it makes sense not to define shortcuts for common activities, such as the creation and initialization of error messages, data transfer objects, and so forth. So, rather than using the ObjectCreationAction over and over again and setting values on the newly created objects, define a stereotype that covers exactly this kind of activity.

Loops and conditions

Usually there is no need to create specific guidelines for elements that are already covered by the UML specification. The same holds true with regard to *loops and conditions*. Unless you want to define a specific kind of loop, such as looping through a tree-like structure, the general guideline is to use the loop element available whenever possible. Figure 4-38 shows an example of the use of this element.

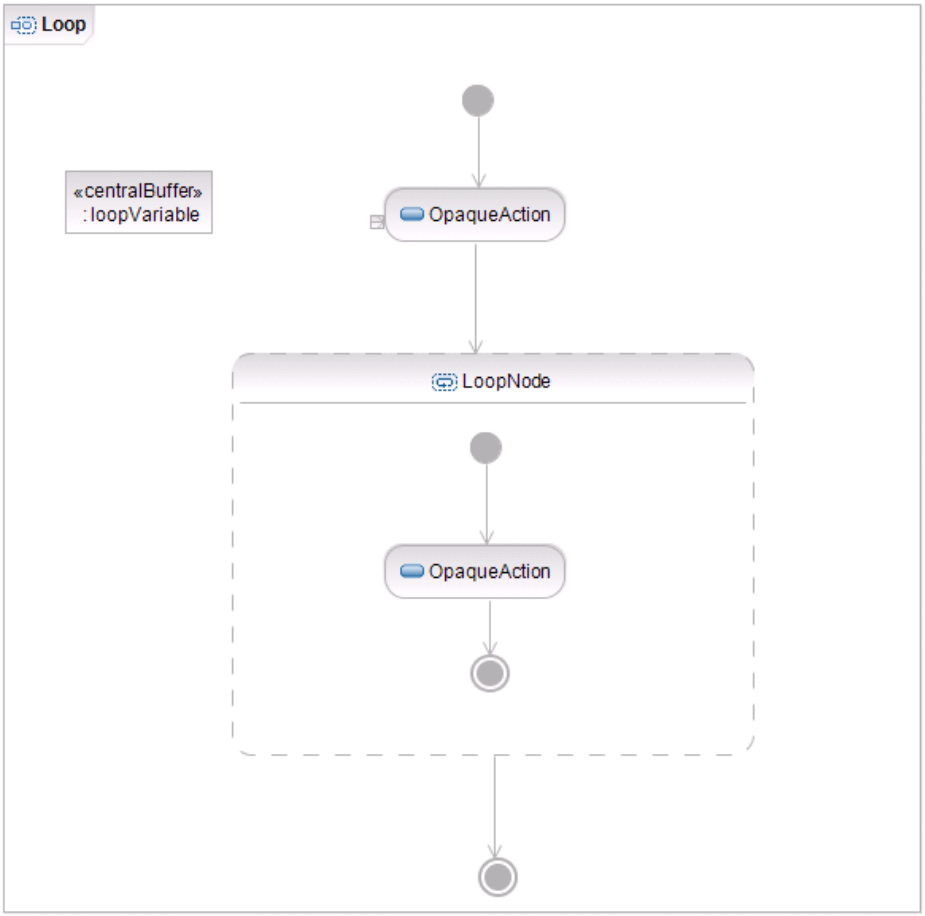

Figure 4-38 Loop element

The same line of reasoning is applicable to conditions. Hence, we do not suggest using a stereotype for conditions unless you plan to design a component that considers a condition to be an entity with a lifecycle of its own. An example of this kind of component is a grammar parser. However, apart from these rare circumstances, you will hardly ever need much more than a simple condition element, as there is one already available in UML.

4.6 Traceability

Even at this point in the book, we have highlighted the requirement for traceability so often that it makes sense to outline in more detail. We provide a few insights into how we envisage realizing traceability. We place the topic at the beginning of all the core modeling chapters because we want to make you especially aware of it prior to you creating any artifacts.

There are great benefits to remembering traceability at all times. It is also advantageous to add traceability while we are creating the models. If we do not add it right away it might be deemed an irrelevant detail. Based on that assumption it could easily be dropped from the task list in the urgency of the project. Also, when trying to add the traceability information at a later stage we tend to have forgotten certain details. We therefore advise you to add it near the time when you are creating the models.

However, there is no need to add trace information right when we are drawing an element in any of our diagrams. First, most models are still too volatile right after they are drawn up and are likely to change again. Each change would then mean double the effort by taking the trace information into account in conjunction with the element to be changed. Secondly, it is typically more efficient to deal with one task at a time. Because the task of adding traceability information is a mechanic one, apart from the thoughts that had gone into drawing up the diagrams prior to that, you are better off carrying out this task at one time and thus connecting all elements of one diagram at the same time. That said, it is sufficient to revisit a diagram right after we have finished it and to add the trace information only then.

Now we discuss a key aspect. Does traceability information have to be visible or can it merely be embedded in the model, just allowing us to trace information, but not visualizing it?

The answer is twofold. First let us look at why we are adding such information in the first place. Reasons include these:

- Providing additional information
- Reasoning the existence of an element

- Proof of completeness
- Providing the ability to provision complex reports

Additional information about an individual can be provided by linking it with other elements in other parts of the overall model. Because our models build logically up on top of each other and thereby strongly complement each other, we can use the value that each element provides. If, for example, we draw a use case bubble and describe it with all relevant details, it appears to be okay with us at most times. However, adding the link to the requirement allows us to read further details about the use case itself. Now one might argue that a use case is merely another form, a visual representation, of a requirement. Still, as we will show in our methodology described in Chapter 6, “The modeling disciplines” on page 261, we expect even there the different models to be complementary to each other, where each model adds further value to the whole. Another example, therefore, is the ability to trace back a class to a glossary entry that holds additional background information, as we find it in various facets of data modeling.

Just because we are adding traceability information, we therefore justify the existence of this element. Because each and every aspect that we are modeling has to fulfil a special purpose while at the same time avoiding the implementation of anything unnecessary, we are constantly faced with making decisions and providing reasons as to why we want to do certain things and why we cannot do others. Traceability provides a good means to demonstrating exactly these thoughts. You can imagine the power, having your architectural decision model embedded in the design tool of your choice and being able to trace design elements back to a particular architectural decision.

At some point in the project we are faced with the task of proving completeness to our project management and the customer. You could tell them that we have respected every requirement brought up to us and convince them of that. As a project manager or customer you have the choice to believe such a statement and move on, or question it and move on, or demand rework. In any case, this is often based merely on belief and trust. Trust is a good thing, but it is better invested during other phases in the project. Based on the traceability information provided, we can demonstrate and prove that we have it all. However, to be able to do exactly that, traceability information has to be gapless, thorough, and consistent across all models defined during all phases in the project, from requirements to design models, implementation models, and possibly test models if you want to extend our methodology by including testing, which we strongly encourage you to do. By honoring all models and their connectedness, we are now able to provide proof that every single requirement manifests itself in a certain design model and eventually some piece of source code.

As you can easily interpret from the previous paragraphs, embedding the traceability information is one thing. Everything stated previously revolves around reading all the information back out of the model again and providing it to some party interested in it. Mostly, this party is your own project management. If you happen to be a project manager, we encourage you to become familiar with the ability to draw reports out of such models. We do not want you to create all reports yourself, but at least be aware that you could and should rely on such reporting. Hopefully, you have someone on the project who knows about all the traceability embedded and also knows how to extract this information again through BIRT reports. Your other choice is querying the state of affairs and writing it down again in an Excel spreadsheet to relay the message back up the chain to your program management. The manual way is more prone to errors, not just because of the manual creation of the report, but also because of different interpretations about readiness, completeness, and other factors to be reported on. The model always tells you the truth, and as such it should be the center of any tracking and reporting activities. For further information, refer to the resources listed in , “Related publications” on page 559.

Let us go back to our original question. Do we have to have traceability information visualized? It depends on who the consumer will be and on the complexity of the specific domain on which you are working. For example, there is no need to visualize the trace between a data model’s entity and the corresponding glossary entry. You would not get any additional value out of it. Instead, it is sufficient to be able to report on it and thus interpret embedded trace information that is not visible. There are, however, at times complex concepts to be conveyed to other people. In such a case, it can be good to visualize the dependencies to extract the diagram and use it for further explanations. We are fully aware that such a response might be unsatisfactory, but at the same time hope that you understand the considerations that you will have to undertake to determine whether a visual representation is required. The result will be the same when drawing the reports.

At this point, we elaborate a little further on the aspect of traceability by taking you through a concrete example. Even though you will read more about the glossary and conceptual data model, we will use this as the base for our continued discussion. For now, simply assume that all that we need to do is trace back each and every entity that we are about to create in our conceptual data model to a glossary entry to enrich it with additional information stored centrally in the glossary.

Because we are creating both models within the realms of the InfoSphere Data Architect, one could be tempted to use the built-in data mapping editor. However, first, this would generate too much overhead with yet another diagram being created. We do not require any additional diagram because we only expect us to have to report on the traces back to the origin, in this case the glossary.

Secondly, the data mapping editor only allows for attribute-to-attribute mapping between entities contained in logical or physical data models. Mappings between objects are not supported and cannot be used in our example because we have not modeled any attributes on the conceptual model. Instead, we propose an approach to model the traceability dependencies based on adding traces to each element that are not visual, as illustrated in Figure 4-39.

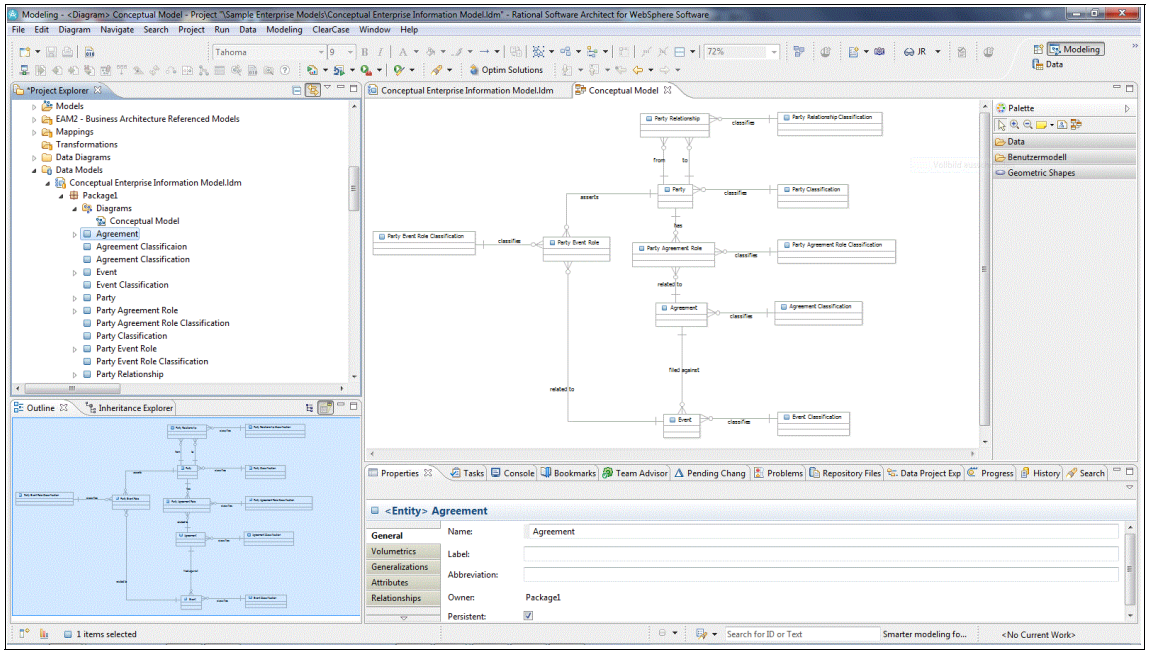

Figure 4-39 Adding a traceability dependency to a data model in InfoSphere Data Architect

After we have the conceptual enterprise information model created and displayed, as in Figure 4-39, we now can start adding the traceability information. Instead of adding a visual element to the diagram, we add a dependency link to each and every one of our existing elements. As we stated earlier, it is important to be consistent about this step. It does not matter if between two models you use “Trace” and between two other models “TraceMe” as the type as long as you remember what you have used. Matters become more complicated when you mix different types between two models. However, the preferred practices include a recommendation for your design guidelines so that every designer always uses the same design conventions.

It also is good practice to provide consistent and meaningful names depending on what you are trying to describe. For example, you might want to trace not just to the glossary model, but also to the requirements model to prove the relevance of each entity and justify its existence. Therefore, rename the dependency consistently to express something more meaningful and so that this dependency later on can stand for itself when you have created many more of these. Traceability reports could later consider the context of the model. Therefore, a detailed description of the dependency, including the model name, is not necessary. Still, the point is to have a name as unique as possible to later be able to distinguish one dependency from any other one that might also be included in such a report.

The key now is to select the target entity that the dependency wizard will guide you through. Locate the Target section on the Properties tab of the dependency and click the ellipses button (**...**). This opens a dialog like the one shown in Figure 4-40.

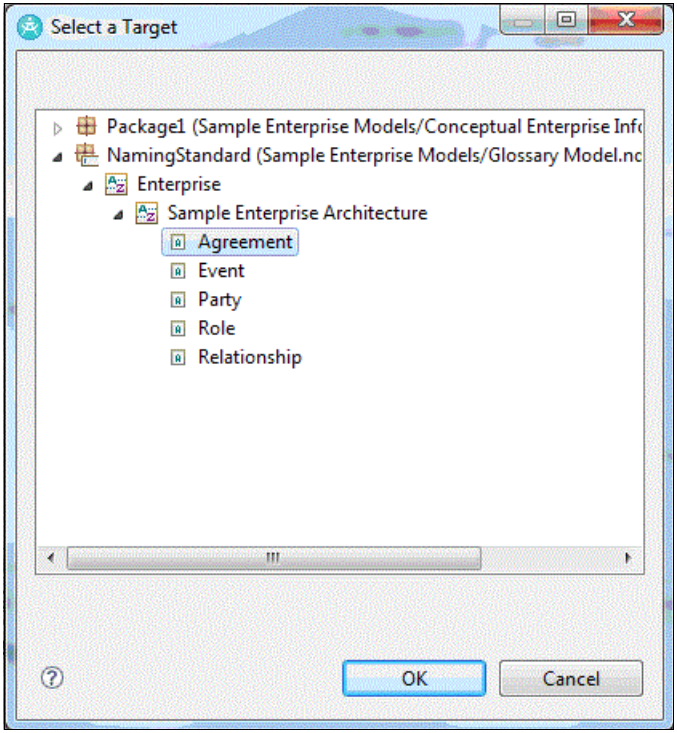

Figure 4-40 Adding a traceability dependency to a glossary model in InfoSphere Data Architect

Select the target entity, which in our case is a glossary entry. When doing so, remember that this dialog only shows projects that are open, so remember to first open the glossary before adding this dependency. We now have successfully added the traceability information between an entity in a logical data model and a term from the glossary. The next time that you are interested in these dependencies is when you are required to draw a report for someone from the model. Reporting is covered 9.1, “Creating reports” on page 462.

For completeness, we also outline the visual traceability aspect. We have already introduced you to visual traceability. Even metamodels, such as our methodology metamodel, represent a visual traceability, a subset of which is shown in Figure 4-41.

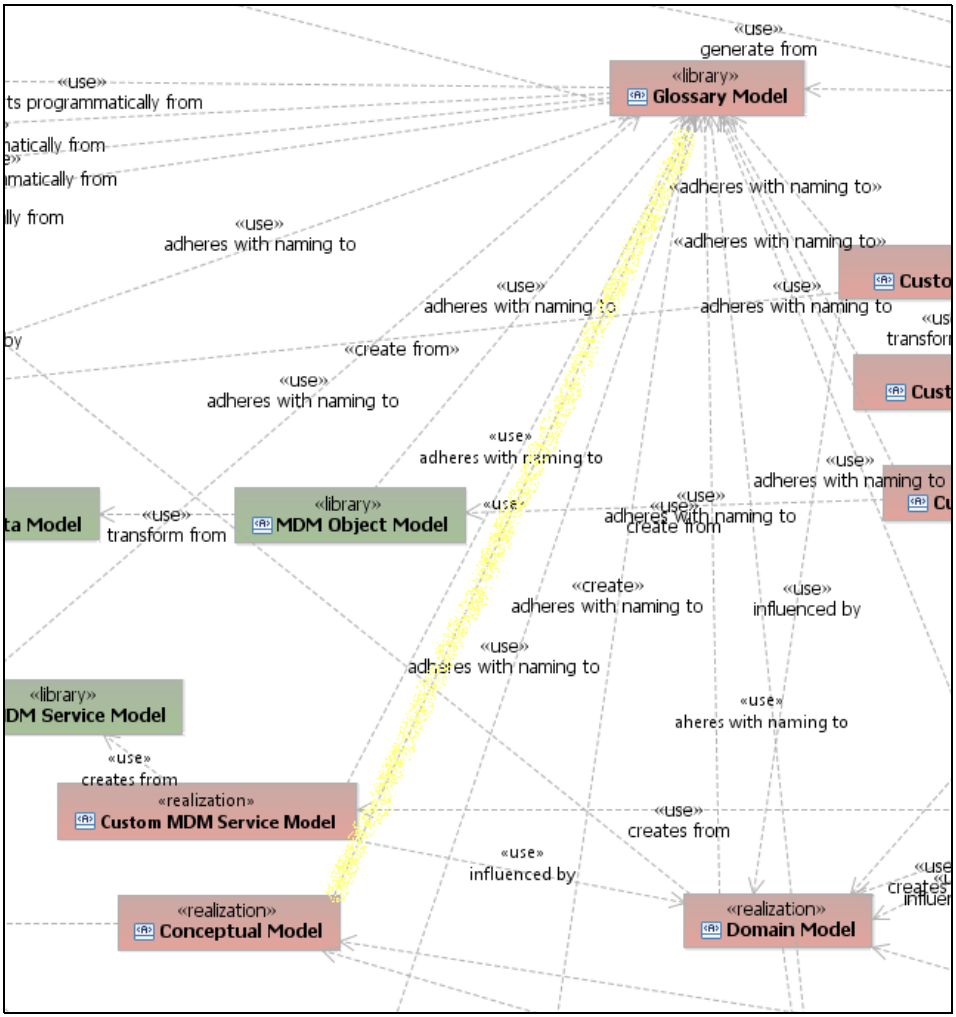

Figure 4-41 The metamodel with highlighted trace

On a meta level, visual traceability describes how an element of one model is related to another element of another model. However, at this level, it is rather unspecific. So instead of describing it globally we now are required to describe the traces demanded by the metamodel for each and every element that we have drawn within each model. Therefore, the traces that we add for each element can be seen as the firm instantiations of the traces described in the related metamodel.

These diagrams typically use the following choices:

- Embed in same the diagram.
- Create a new diagram containing all the elements.
- Create a new diagram for each element.

Embedding traceability information to the same diagram that we used to draw the elements for our model can be convenient. However, we are also in danger of cluttering our design with additional information that might not be of interested to all readers of our diagram. For this matter, we only recommend taking this approach when our model is made up of many small and easy-to-read diagrams. Because most projects (master data management projects in particular) encounter some complexity that demands a few reasonably complex diagrams, we suggest that you do not adopt this approach.

Creating an exact copy of the original model diagram and then adding all related elements to be traced against is possibly the best idea for a number of reasons. First, when you are faced with a diagram containing many elements, you do not have to create as many diagrams, which is cumbersome at best. Second, it provides a high-level view of how the elements between the two interconnected models relate. This can be of great help understanding a certain design better. Third, it gives you an immediate glimpse as to whether your traceability mapping is complete or still has gaps. Any element that is not connected to a counterpart in the other model is still left untraced and thus has to be taken care of. The downside of this approach is that the diagrams practically become unreadable, so you cannot use them for anything else but conveying trace information. However, we assume that you will create other diagrams dedicated to a particular problem anyway, and thus still be able to explain the concept about which you are concerned.

The last option, drawing up one diagram for each element, is mentioned merely for completeness. Any reasonably sized model will provide you with such a number of elements that is time consuming and undesireable to follow up on embedding traceability into the model.

5

Sources

The methodology used in this book starts with the sources. This is good because we rarely have the luxury of starting from the beginning. Even then we have to consider existing sources for the endeavor. For the purpose of this book and the methodology contained herein, we classify the sources into the following categories:

- Server
- References
- Customer

5.1 Introduction

When embedding IBM InfoSphere Master Data Management Server (also known as *InfoSphere MDM Server*) as a master data management system in an existing environment, consider the various resources that this product delivers. These resources include the functional aspects embedded in the product itself, and the documentation and other artifacts that the product delivers. These resources also relate to the way the technical components that the products provide us with are used. Even if we adopt a master data management product other than InfoSphere MDM Server or integrate with other systems besides master data management, these resources are similar.

References, in addition to the specific resources that the InfoSphere MDM Server provides, are countless. We are referring to the numerous additional resources that can also contribute to the expected solution. Clever adoption can accelerate the solution design, and in particular support you in setting up your project. Such references can be industry-specific, master data management related, or generic. Each of these types provides its distinct value to the overall solution that we are going to build. Industry-specific references, such as those provided by the IBM InfoSphere Industry Models, provide us with deep insights into your particular area of business. References, such as the IBM Reference Architecture, however, are neutral to any particular industry pattern, but provide us with a robust view of any type of master data management solution. The generic references, of which the IBM SOA reference architecture is a good example, outline all best practices related to service-oriented integration independent of the type of solution that we are going to deploy, in our case a master data management solution. We consider all of these types of references to enrich the content that our methodology produces.

Most importantly, you will already find a lot of information in your own organization. While it might not all be available as Rational compatible UML or XMI formatted resources, you should find it easy to either convert it or enter it based on the information that you have in front of you. Some of these models are the data models that come with your data sources, but perhaps you are already the owner of a comprehensive set of use case and business process models that can be integrated into the modeling approach presented in this book to achieve a quicker and better outcome for your master data management implementation and integration.

In the context of this book, we investigate the InfoSphere MDM Server related sources a further. This will provide an overview and a framework that was used when outlining the modeling steps presented in this book. Some customer models are discussed in the appropriate modeling chapters, but we do not highlight them separately as the source from which the models were derived.

One area not discussed in this book is related to the external references. You can read up on them separately following the information provided in “Related publications” on page 559. We have adopted some of the proven practices outlined in these types of references and developed the methodology gained from insights into these references. Read those references if you are not already familiar with them.

5.2 Documentation

InfoSphere MDM Server provides a wealth of information. Using the information provided gives you a head-start into setting up your own project. The artifacts provided by InfoSphere MDM Server can be split into the following categories:

- Documentation
- Models
- Runtime

Documentation plays a major role, especially in the early phases of the project. Acquaint yourself with the product first and come to terms with its true capabilities and how to adapt it best for your planned implementation. While it might appear a little strange at this point in time that we mention documentation as an artifact to be consumed in this methodology, we think that notion will make more sense as you read this book. The documentation itself is aimed at a number of different user groups and will be consumed by groups of people with differing requirements. These include architects, designers, and developers. In the following paragraphs we describe how some of the documents provided maps to the methodology.

With respect to the models, it is much easier to see and understand what role they play in the design of InfoSphere MDM Server in which InfoSphere MDM Server participates. The most obvious models are the logical and physical data models against we have to map. Other models, like the service operation models, exist in the wrong format, considering that we want to pursue a Java RMI based integration as opposed to web service style integration. Although the web service models are publicly available for their respective XML Schema Definitions (XSDs), the Java object model and service operation definitions are buried within the depths of the products and have to be carefully extracted from it or manually created.

The runtime execution framework is one of the major distinguishing elements that InfoSphere MDM Server provides. It provides much in regard to an SOA integration, including services to communicate with the product. The product also provides master data management specific capabilities, including data matching and de-duplication, as well as many other features that make up a master data management solution. Almost more important than the functional aspects are the non-functional aspects. These are features related to performance, scalability, reliability, stability, enhanceability, and many more. A product such as InfoSphere MDM Server is proven to deliver on all of these aspects in comparison to the requirement for extensive testing when you intend to develop such runtime execution framework all by yourself. It keeps your back free, and you can rely on its capabilities in these areas. This in turn allows you to concentrate on the two most important aspects of any master data management solution:

- The business issues that you need to address
- The system integration that you require

Because the execution environment does not contribute to our modeling methodology other than providing us with a stable and extremely reliable environment in which we execute our modeled and implemented artifacts, we do not go any deeper into this platform. Refer to the InfoSphere MDM Server documentation if you need to know more about this.

We start by looking into the documentation and the information provided therein. While doing that, we at the same time look into adopting that information and including it into our models. Considering the typical sequence that we have to follow describing InfoSphere MDM Server specific functionality, we are dealing with these:

- Technical architecture
- Data mapping and data modeling
- Service design

5.2.1 Technical architecture

Technical architecture includes numerous aspects, most of which we have excluded within the scope of this book. The fact that we have excluded it from here does not mean that this information is irrelevant to the methodology, but it certainly does not have the same impact as other models and pieces of artifacts that we explicitly include in this book.

Architecture splits into many different categories. As technical IT architects embedding a product such as InfoSphere MDM Server, we are most concerned with the component and operational modeling, as well as the definition of product behavior. The information supporting us with these tasks is predominantly captured in two documents:

- Installation guide
- Developers guide

Technical architecture accompanies the project along its way, but after the major architectural definitions have been put in place, we can focus on embedding our solution. In our case it is based on InfoSphere MDM Server. Using other products or even a complete custom development, however, requires similar tasks to be carried out.

Installation guide

The InfoSphere MDM Server installation guide is a good starting point, not just for the installation of the product itself, but also when it comes to determining the non-functional factors and the topologies in which the product can be laid out when it comes to operational modeling. It appears to be strange that the guide that you are probably going to read first plays a vital role again towards the end of your architectural activities. In any case it is wise to revisit the information contained therein, especially the section dealing with the installation scenarios.

Looking a little closer into operational modeling we understand that we derive our topology from the components that make our system that we are building. Unfortunately, the product does not deliver a component model that we can directly import into IBM Rational Software Architect or any other modeling tool. It neither provides topologies that we could import, which seems logical, because, unless we roll out InfoSphere MDM Server without any form of customization and as a single tool to address the master data management challenge, we will always define and build additional components. Because also the component model is not provided, you consequently will have to go and add the topologies manually. The good news is that IBM Rational Software Architect has got a strong topology modeling editor built-in.

The resulting structure in project explorer could look as shown in Figure 5-1.

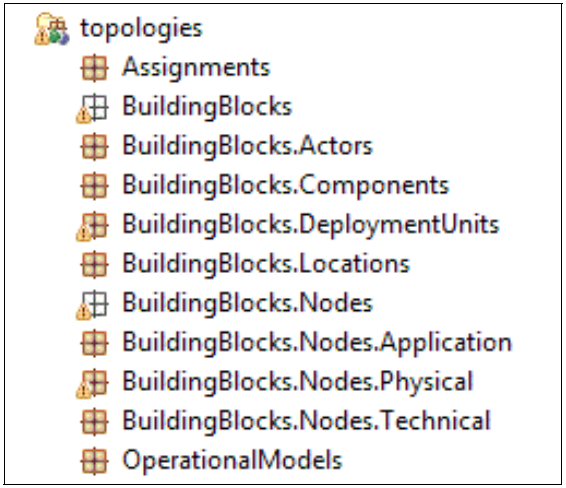

Figure 5-1 Sample package structure for modeling master data management topologies

Contrary to the description contained in the Installation Guide, you can build up your model based on thoughts about reuse. As you can see, we have dedicated a topology package for each individual aspect of topology modeling. These can all get reused in the various operational models that we can and have to create. Just consider your environments. You most likely will have one environment with every component running locally for the developers, a thoroughly planned distributed topology for the production environment, and anything in between for your test environments. The elements contained in the building blocks package can then be reused in all three topologies.

As far as our modeling recommendations go, there is not much to pay attention to in addition to the usual topology modeling. Because this part is so loosely coupled to our methodology and only occurring at such a late stage of our implementation cycle, we are effectively finished at this stage and “only” getting our system ready for deployment. Hence, at this stage we would like to simply point you to IBM Rational Software Architect documentation on topology modeling. Additionally, you will find good resources on IBM developerWorks® also outlining how to get started with topology modeling in Rational Software Architect.

You can see the operational model mainly depending on two artifacts:

- The Installation Guide
- The custom component model

The components that you require for custom component modeling are derived from the Developers Guide.

Because we are only remotely touching on architectural modeling in this book, we leave our description about operational modeling as this. Refer to the product installation guide and other resources that discuss operational modeling for a much more detailed description.

Developers guide

This guide in conjunction with the corresponding Developers Sample Guide should be known to all developers who are working on the customization of InfoSphere MDM Server. However, we do not want to restrict access to this audience only, because the technical architects will also find a large amount of relevant information in this guide.

One particular aspect is related to the description of all the products components. Unfortunately, there is no model provided that you could import into IBM Rational Software Architect or any other tool. The only way to work around this limitation is to draw the component model manually based on the information contained in this guide.

The resulting component model diagram could look as shown in Figure 5-2.

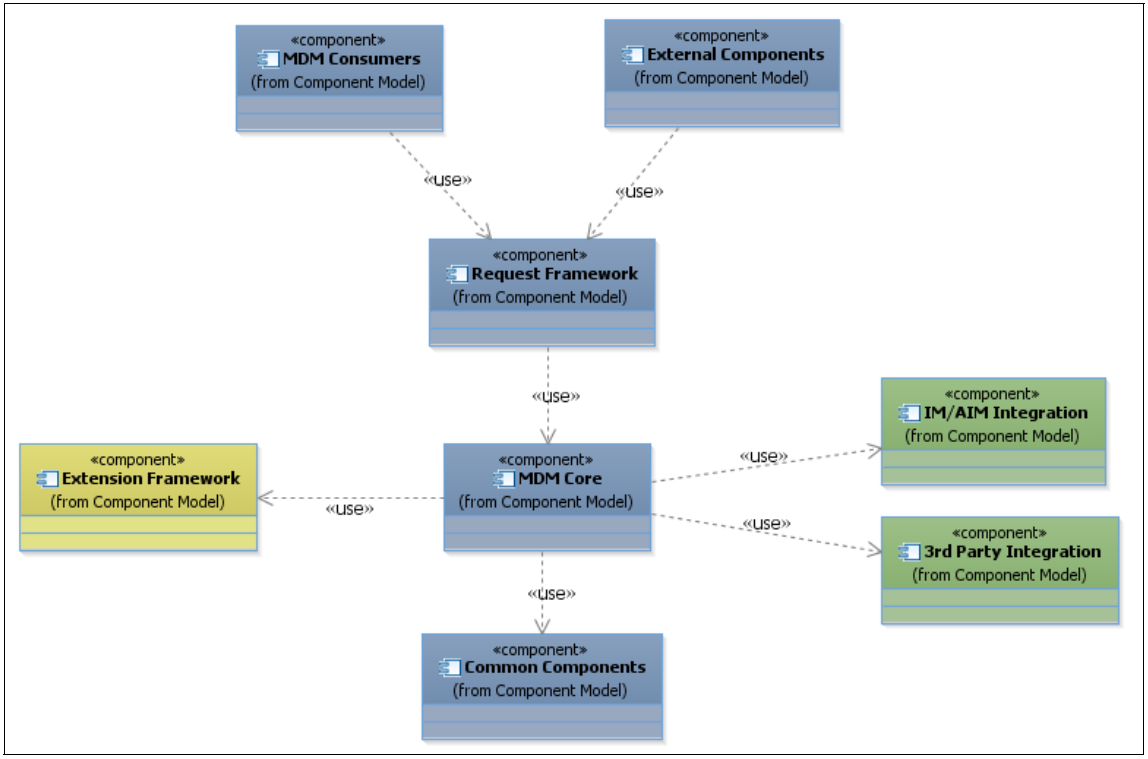

Figure 5-2 Sample high-level InfoSphere MDM Server component model

Figure 5-2 represents only a high-level view of the components that make InfoSphere MDM Server. As opposed to the component model described in the Developers Guide, you can already see one significant advantage of modeling it in Rational Software Architect. We refer to the ability to apply some structure to the models and embed the components in a logical (and in this case also nested) structure. Navigation through such a nested structure takes place using dedicated navigational menu and context menu items on the UML elements as contained in the diagrams or the project explorer, respectively.

Again, this model is only remotely related and loosely coupled to our implementation methodology, so it effectively bears little relevance to it. Even when we are faced with the task adopting this component model in the context of our overall master data management solution component model, we only have to ensure the right interaction between these components. The complete master data management solution component model, however, is relevant and important in respect to the functional mapping, that is, the identification of the appropriate component to fulfil a task that we require as part of our solution.

Still, we do not have to watch out for any modeling aspects in particular. You can simply refer to the documentation about component modeling that Rational Software Architect or any other resources provide.

Apart from the architectural information, you also have InfoSphere MDM Server Workbench and the User Interface Generator User Guides, which are somewhat related to the development tasks.

5.2.2 Data mapping and data modeling

The first task revolves around data mapping. Because InfoSphere MDM Server provides a data model, we have to identify the attribute that suits our business attribute most. Having said this, there are exceptions in that sometimes we cannot identify an attribute on the available data model to which we can map our attribute. Sometimes there is one that appears to be a close match, but we are not entirely sure whether it is the right match. In these cases, we need to extend the existing data model. This is when we have to execute data modeling even though we have a data model available to us.

For this task we require appropriate dictionaries, because without them we have no understanding of the true meaning of an attribute and consequently cannot decide whether mapping the business attribute is a fit. Ideally, all InfoSphere MDM Server related entities and attributes exist on the glossary. However, the product does not deliver a pack that we can simply import into the IBM InfoSphere Information Architect or IBM InfoSphere Information Server provided glossary. Instead, we see the following documents to support us in this undertaking:

- Data dictionary
- Common data dictionary

Both of these documents provided by InfoSphere MDM Server are closely related and provide the exact same structure. The only difference is that they provide information about entirely different entities and attributes that they hold in regard to the InfoSphere MDM Server data model.

Having said this, we have already stated the content of these two guides. They are in essence describing each attribute within each table that InfoSphere MDM Server provides. The information contained therein is limited to the name, a description of the intended purpose of the attribute, the data type, the length, and a nullable and a primary key constraint. As part of our methodology we anticipate such vital documentation to be provided within the model itself. Because these dictionaries describe the physical data model, they should accompany them accordingly. Both the physical data model and the data dictionary should be present within the InfoSphere Data Architect.

The logical place for the dictionary is the glossary. The advantage of using the InfoSphere Data Architect instead of distributing the PDFs that the product delivers is first that all relevant information is captured in a single location. Secondly, such a glossary can be used to validate the model. Although this aspect seems irrelevant in respect to the physical data model provided, it becomes more important in conjunction with the extensions and additions that we are building to the available data model. Furthermore, it is not just the data model entities and attributes that we want to validate, but we also want to enforce naming conventions for our source code based upon the same definitions. The reason for this is that we have an entirely consistent and readable source code at least in respect to the models that we are working against. We discuss this aspect of model validation in more detail in Part 2, “Smarter modeling” on page 129.

Because there is no such glossary model provided that is based upon these dictionaries, we are faced with the one-off task of importing them to our model. This task means manually adding all the details to the InfoSphere Data Architect based glossary due to the fact that the dictionaries are provided in PDF format only. As opposed to our custom extensions and additions, the information that is captured from those PDF files is reusable, because it relates to generic InfoSphere MDM Server artifacts.

Service design

Service design is the next logical step in the sequence of tasks after the data mapping and modeling is complete. You only can define the composite services to be used when you know the data mapping, first because InfoSphere MDM Server provides a large number of services that you can reuse, and second because the service interface operates on the data objects that you are going to persist. Thus, rather than creating services that you can use, you can determine logically which services are applicable, based on the provider signature.

This determination though is specific to embedding a given product into your system landscape and could possibly be a little easier when simply persisting the business object model in physical tables of the same form. When you are working with a custom domain model and are at the same time using a given data model, such as the model provided with InfoSphere MDM Server, you first have to apply the mapping to determine the data objects that are mapped to the domain objects. After you know of the InfoSphere MDM Server data objects, you have to validate which services allow you to operate on these objects.

InfoSphere MDM Server provides the following documents for support:

- Transaction Reference Guide
- Common Services Transaction Reference Guide

Again, we look at both documents at the same time. The information contained therein is similar to different functional domains of InfoSphere MDM Server. On the functional side, this is view is getting much more interesting because in addition to the obvious descriptive information, we now also have information that we already want to use and build upon in our service design. We anticipate that the information contained within these two documents will contribute to the service operations model that describes the service operations that InfoSphere MDM Server provides and can be consumed in subsequent composite service design.

Before jumping to any conclusions let us start evaluating what information the InfoSphere MDM Server provided guides present us with. It is the following:

- Name
- Description
- Example
- Usage information
- Pre-conditions
- Mandatory input
- Inquiry levels
- Filter values
- Transaction behavior
- Request structures
- Response structures
- Special notes

On first glance it is already apparent that we have two distinct property types. One is purely descriptive, such as *description* and *example*, while the other is *descriptive* and *functional* at the same time, like *filter values*, *inquiry levels*, and *mandatory input*. When we try to identify the optimal placement for each of these attributes without taking any significant notice of this fact, we can see three obvious options:

- Glossary

  The information contained in these guides could be placed in the glossary. This placement comes in handy when we are considering the transaction reference as purely descriptive. One pitfall though already is that the glossary entries still would have to be created and linked to the service operation using an appropriate relationship. The advantage following this approach could allow for enhancement of the analysis rules that check against the glossary, such as ensuring adherence to naming standards.

- Service operations

  Here all required data is added directly on the operations that represent the services. The benefit is that there is no need to maintain a separate glossary model. The downside is that we cannot validate against meta information, as in the glossary-based approach. As we will also learn shortly, we will not be able to map all information in an identical fashion, which makes this approach a little more difficult to design and manage, but already providing us with everything we hope to see to drive code generation.

- Mix and match glossary and service operations

  This approach requires us to describe the numerous properties in two places. For example, we could put the *description* and *example* on the glossary, and functionally related aspects such as mandatory *inputs* and *inquiry levels request/response structures* directly on the service operations where we could also use them to drive code generation. This would separate the two distinct property types, but makes the model maintenance and drawing reports from them more complex. The glossary entries would also have to be created and linked to the service operation using an appropriate relationship.

Looking at these options, we seem to find some truth in all of them. Before we attempt to dig into the core of this problem and outline a solid solution from our point of view, we consider problems we see with some of the functional attributes:

- Mandatory attributes

  The same object type could participate in many service operations. Within each of these operations we use a new instance of this object. The problem herein is that such an instance can declare the mandatory attributes and also specify its own defaults. The impact is that there is not a single class hosting all service operations, but many subtypes instead. The resulting design is therefore more cumbersome. As a good example of this you can imagine a party object with two attributes:

  - Birth date
  - Gender

  Both are mandatory during the initial add operation while they become optional on subsequent update operations using the same party object type.

- Inquiry levels

  The issue is also explicable from a service operations point of view. While the attribute is always named the inquiry level and described by a list of values, the meaning of these values is always a different one. Also, the range can vary from operation to operation.

As an example, you can picture a `getParty()` operation, where an inquiry level of 0 means to return party-related details, whereas the same inquiry level number 0 on a `getContract()` operation indicates the return of contract-related data instead.

- Filter values

  This aspect remains stable across all get operations, as opposed to the inquiry level, which is service operation dependent. Consequently, the value of filter ACTIVE indicates the return of an active time slice across all operations that make use of this filter criterion.

Now that we have taken this slight detour, we can return to our investigation regarding the optimal placement for each service-related attribute and its appropriate modeling. Because we focus on the design and modeling aspect of it and are already clear on also wanting to use the functional information for code generation purposes, we now take the view coming from the modeling tool, IBM Rational Software Architect. Here we can adopt one of the following approaches, where the latter is an InfoSphere Data Architect based approach:

- Using body constraints

  Body constraints can be described in a number of ways, of which one of them is to use Object Constraint Language (OCL) as a formalized modeling language and the standard approach for UML-based designs. It can be transformed into any coding pattern rather easily. Also on the upside, this approach enables us to keep the implementation separate from the design. The downside is that there can only be one coding pattern, and only when describing the Filter, Inquiry Level and Mandatory fields. This way it gets confusing, not just because it is hidden from sight, but also due to the mixing of a number of attributes. Also, without introducing our own tags, it will be virtually impossible to distinguish between the individual describing attributes when generating the service descriptions. Due to these limitations, we decided against this approach for any of the functional attributes that we need to describe.

- Command or strategy pattern

  This approach requires appropriate sub-types of the classes to be used. For example, if we have a *person* class we have all attributes as optional (least restrictive), because we do not know the usage scenario later. If there is a *mandatory* attribute on an add, we create an add sub-type of that person class and can even stereotype it “restriction”, for example, for the document generator to be able to navigate along well-defined generalizations.

This is a relatively simple approach to modeling, and it would satisfy the document generation requirement, but it implies a certain way of coding that might not be desirable in the first place. We believe that the strategy pattern provides the best means of encapsulating this code and, therefore, we prefer this pattern over the command pattern, which could also be used in this scenario.

We adopt this approach for the mandatory attribute description, which is depicted in Figure 5-3.

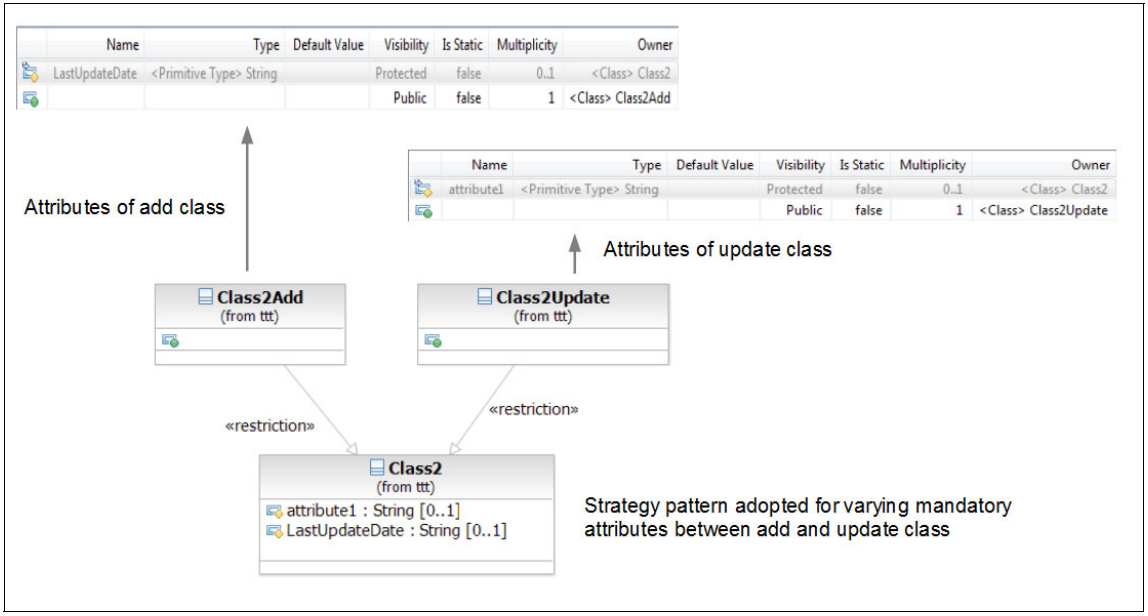

Figure 5-3 Strategy pattern adopted for varying mandatory attributes between add and update classes

- Templates

  This is the UML standard way of describing dynamic bindings. They are meant to model features similar to those known as STL in C++ and generics in Java or C#. However, this carries much more overhead than the previous strategy pattern, while not providing any additional value over it. Therefore, we ruled this approach out immediately.

- Glossary

  Even though using the glossary model from InfoSphere Data Architect does not work because of a tool break, there is the possibility to create an Rational Software Architect based equivalent. The advantage is to store all data in a center repository that everyone has access to and would allow you to base analysis and validation on it. However, this is a work-intensive approach.

The IBM WebSphere® business integration fabric contains a tool set that does exactly this. However, adopting the glossary and ontology profiles from there is a challenging undertaking, and thus we avoid following this approach, even though we entertain the idea of a central glossary often and encourage you to build one up yourself.

- Stereotyping

  The stereotype approach seems particularly useful for those dynamic attributes such as InquiryLevel and filter. However, we suggest careful application of stereotypes and warn against overly adopting stereotypes as the cure-all solution. This approach allows for building up a central repository for these values in a profile and additionally providing information required for validation and code generation at the same time. Again, this approach also allows for the usage in analysis and validation, as well as using the information from code generators.

6

The modeling disciplines

Everything that is done in this modeling methodology being described is based on a metamodel. That model describes the artifacts required and how they relate. However, equally important as that domain model is the verbal description of each of the business aspects, which is called the glossary mode. While the domain model contains the visual representation of the business aspects dealt with, the glossary provides the supporting information.

Also important is understanding how to model with the InfoSphere Data Architect and understanding the modeling disciplines and lifecycle. Also, what neither the metamodel nor any other description tells you, is when and who will be working with any of these artifacts. Those principles are outlined in this chapter to help you get started modeling. We start with the glossary model.

6.1 Glossary model

Even though the domain model also contains descriptive information about the entities and associations contained in the model, the glossary plays an additional role. The glossary definitions usually exist before we are able to draw up a domain model. It is for this reason that we also have included this aspect at the beginning of our modeling description. Before deciding on any particular way of modeling a glossary, be clear about its intended:

- Purpose
- Usage
- Tools

A glossary is much more than a simple collection of words with a description of their meaning. Instead, it also contains additional valuable information. The information represented in a glossary includes:

- Verbal definition

  Textual description of the business or technical meaning of a term, entity, class, or any other related type. The description has to be precise enough to make it unambiguous while at the same time easy to understand to be comprehensible by business and technical staff alike.

- Synonyms

  Often two or more words relate to the same meaning. They are used synonymously. This can be, for example, the result of two systems describing the same entity with identical semantic meaning. In this case it is important to document that these two terms relate to the same aspect.

- Abbreviations

  In addition to the textual description, we often rely on abbreviations to simplify repeated use of the same terms in documentation and other sources of information. It is important to clearly associate an abbreviation with the related term definition.

- Taxonomy

  Terms can be used in a hierarchical relationship. Taxonomies are a typical example for this kind of use. It might be important to highlight the hierarchical relationship of one term to its parents and children.

- Localization

  Especially in today's international work environment it becomes more and more critical to be able to relate to the same definition in multiple languages. While this is not necessarily the case for reasons of the project members, it becomes more important for the business users who eventually rely on the solution that we build.

- Relationships

  While terms might not always describe the exact same thing, as with the synonyms, there can be slight or bigger deviations of one meaning. The key here is that, with all the differences that there might be, there are still some similarities. It is these similarities that we are interested in, as well as the relationships between these terms.

- Lifecycle

  It is not enough to only persist the glossary information once. We have to maintain it during the course of the entire lifecycle of the project and beyond.

The glossary is often not used to its potential. We have mostly seen it as a means of providing a list of words that are common throughout a project. Its usage stopped there, which is a shame, because there are a lot more opportunities in the context of a project than merely listing the words. We typically use a glossary for all of the following:

- Documentation

  This refers to the aforementioned list of commonly used terms and verbs. Focus on the user groups that rely on this glossary. Whereas most glossaries are predominantly aimed at business users, a glossary in the context of such a project serves two user groups, business and technical staff.

- Dictionary

  Apart from simply providing the true business meaning within the context of the native language that the project is adopting, ensure that you capture the nuances in accordance with other cultures that rely on this in some way also. This includes simple language translations and semantic translation, which is required because the same word could mean entirely different things in two countries or languages.

- Consistency

  The definitions within the glossary should be used to ensure and maintain consistency. Do not introduce your own arbitrary words and definitions where a glossary definition already exists. If a definition does not exist already, ensure that it finds its way into the glossary and is therefore acknowledged by other team members too. The importance in that is that you have to ensure that everyone is working in the same context based on the same definitions with the same meaning. Also, because there are almost always multiple teams involved in such a project, this is one of the few ways in which you can ensure that everyone is aware of the same definitions and that everyone uses them.

- Validation

  Based on the consistency definitions, take one further step. Drive validation through the glossary definition. While ensuring consistency is merely related to ensuring that everyone is using the same definitions, model validation ensures that has really happened and is adopted in the design and implementation artifacts that you produce.

Considering both the purpose and usage of a glossary, select the tools that help you achieve these goals. These are the tools in the IBM portfolio:

- IBM InfoSphere Data Architect

  This tool provides an integrated glossary. It serves all the purposes outlined and supports you in all usages. Only in respect to the validation capability does it lack tight integration with the IBM Rational Software Architect. The consequence is loss of validation capabilities for the UML and source code related artifacts. This tool suffices to take the initial steps. We also use this same tool for its data modeling and static data mapping capabilities, and therefore are able to increase the value of this tool.

- IBM InfoSphere Information Server Business Glossary

  This tool represents the most comprehensible approach to providing all capabilities that we have discussed. It also provides integration with IBM Rational Software Architect. In addition, it also allows exposure of the captured information to external business users in an easy and integrated way, where the IBM InfoSphere Data Architect approach would rely on its reporting feature to derive the business information that we need to expose. We decided against usage of this tool because it would require introduction of yet another tool that we have to license, learn, and adapt to. We do not rule this tool out altogether, because it makes perfect sense in some of the more complex business setups that we have seen.

Modeling with IBM InfoSphere Data Architect

Now let us start looking into building up a glossary based on the IBM InfoSphere Data Architect. The first step, if not already done, is to add a new model to our project structure. You can use the New Project Wizard to do that. The built-in wizard conveniently provides you with a useful template Data/Glossary Model.

Refer to our description about the suggested project structure, in 4.3, “Structure” on page 172, to determine the right place, remembering that a glossary will always be placed in a data models folder within the selected project.

Now we are ready to add new terms to the glossary created. The Contained Words section has the list of names added to the glossary. The resulting glossary could look as shown in Figure 6-1.

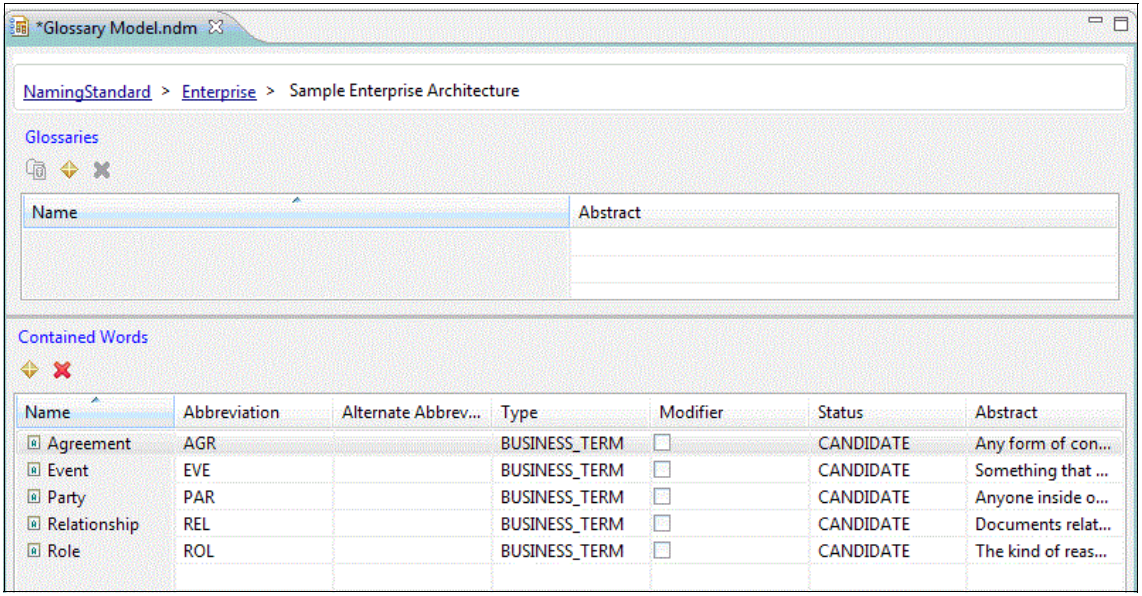

Figure 6-1 The IBM InfoSphere Data Architect glossary editor

Take time to investigate the properties that you can set on each term added. Be sure that you complete the properties as completely as possible in accordance with our initial introduction to the purpose of a glossary and the elements that such glossary should hold. In particular, it is worth dedicating some time to the provisioning of the related words and the synonyms, but also adding proper and comprehensive descriptions is of great help to anyone using the glossary later. Figure 6-2 shows part of the property section for a glossary entry.

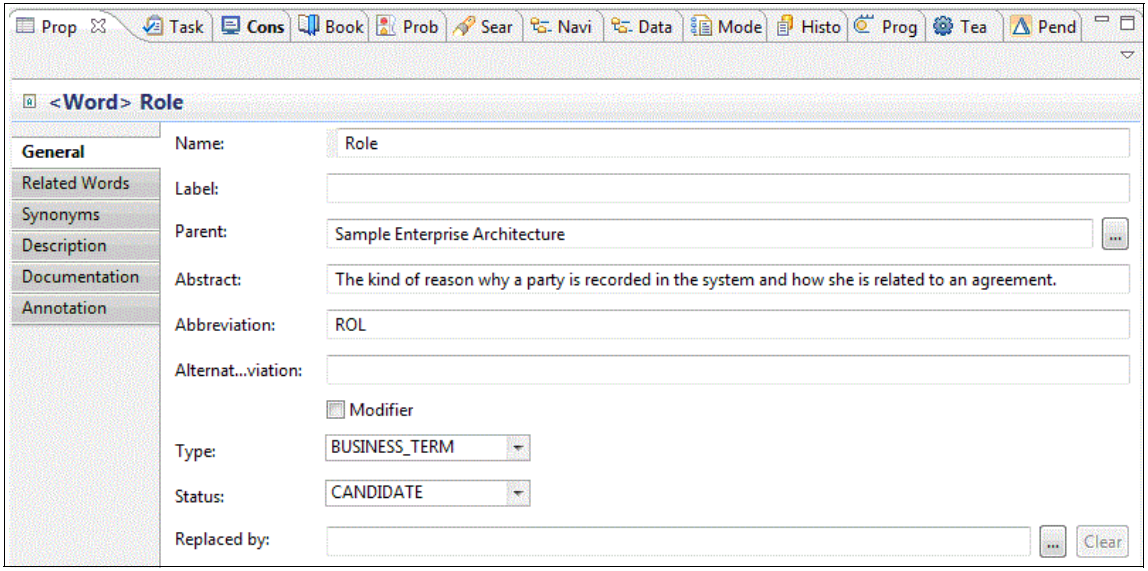

Figure 6-2 The general properties in IBM InfoSphere Data Architect glossary editor

Even though we continue to rely on the glossary in respect to model validation, there is nothing else that we have to be aware of at this time when we create the glossary model. The steps related to validation are described in Chapter 9, “Deliverables” on page 461.

If we need to provide an external document outlining the glossary entries as deliverable, we rely on Business Intelligence and Reporting Tools (BIRT) reporting. We describe the general concept of reporting in Chapter 9, “Deliverables” on page 461.

6.2 The modeling disciplines and lifecycle

At this point we have already discussed most of the general aspects of our modeling methodology. You have learned how everything that we do is based on a metamodel that describes all artifacts that we require and how these artifacts relate to each other. You have also seen that we distinguish between sources, assets, transformations, and deliverables. So far, we have discussed all general aspects of our models, including the important information about our traceability requirement. Everything we draw, generate, or otherwise describe should be traceable against the business requirements and therefore function as proof that we have implemented everything that we were asked to. We also outlined the IBM InfoSphere MDM Server related sources and our first asset. The InfoSphere MDM Server sources provide a head-start into solution implementation. However, they also demand that we acknowledge some of the core principles upon which the product is based when modeling a solution around the product. The asset represents the foundation for everything that we do in our modeling concept and thus applies to all artifacts. This is the glossary model. It provides insights and details about each and every system aspect that we are dealing with in order for everyone to develop the necessary understanding of the system to be able to build and use it.

6.2.1 Disciplines

To be able to focus on only small parts of the entire metamodel at once, logical groupings have been formed. These groups are based on the common modeling theme that is underlying them. We distinguish between five modeling disciplines:

- Architectural modeling
- Business modeling
- Data modeling
- Data mapping
- Service design

We explain how the artifacts from the metamodel relate to these modeling disciplines in Chapter 7, “The models” on page 281. It is important for us to note at this stage that we can execute some of the activities in these disciplines in parallel, whereas others are strongly sequential. For the sequential part it is important to understand what the dependencies are and what we have to do to commence with the next activity.

Figure 6-3 outlines some of the most important transitions between the disciplines.

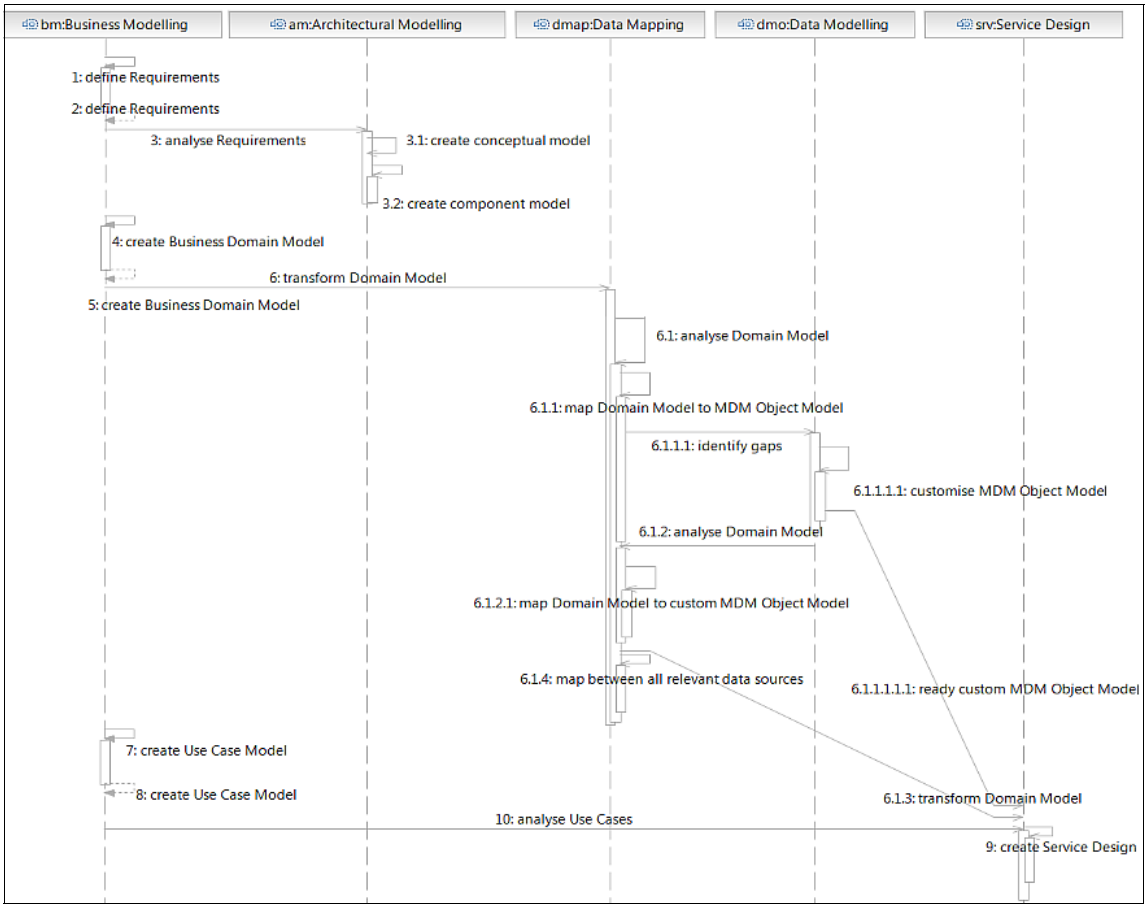

Figure 6-3 The core dependencies over time between the modeling disciplines

Figure 6-3 shows a few of the elements that are relevant to follow the methodology. It all starts with business modeling. Here you first need to gather all requirements related to the project. After the elements have been defined, you can commence with architectural modeling. Among other artifacts to create is a component model. In respect to a domain-driven design (DDD), a component model holds a special value in that it helps to determine the exact location of the domain model and which component is responsible for realizing it.

The enterprise architecture team can also start with the extensions to their conceptual enterprise information model. It is important to understand that these activities are parallel to the definition of the business domain model on which the business modeling activities continue. This is happening all the while that the glossary is also maintained. Both business and enterprise architecture have to discuss and develop both models together, where each team is drawing up the model that means the most to them. While the business does not necessarily have to be able to understand the conceptual enterprise information model, The model must incorporate an understanding of the business. Discussions typically should be based on the grounds of the new domain model. After this has been established you can then make the transition in the data mapping.

First analyze the models that are available and then start the mapping process. During this exercise you might encounter a few attributes that cannot be mapped and thus require an addition to the data model. After these are completed you can finalize the data mapping. The entire cycle could repeat multiple times. In the meantime, the use cases are also finalized in the business modeling discipline. After all three types (that is, use cases with their respective business processes, the custom master data management object model and the domain model, and the corresponding data mappings) are finished, you can start with the next tasks involving service design.

The major activities in each discipline have already been discussed. This initial diagram, however, focuses solely on transitions between the modeling disciplines and thus does not represent a complete list of all artifacts. For this, refer to the metamodel and put all artifacts contained therein into the context of the five modeling disciplines. Before doing this, be aware of another structural pattern applied in an attempt to make the presentation of the modeling disciplines easier to understand. This pattern is based on the following model levels that describe model-driven development (MDD) environments:

- Enterprise models
- Analysis models
- Design models
- Implementation models

Actions are mostly related to the creation of the artifacts contained in the metamodel. In some cases you also find us performing updates too. This is, for example, the case in respect to the glossary that is continuously maintained while we create additional artifacts. Additional actions involve data mappings and transformations. These actions create or manipulate the artifacts.

Figure 6-4 outlines the general principle of the detailed diagrams that you will see for each discipline in the upcoming sections.

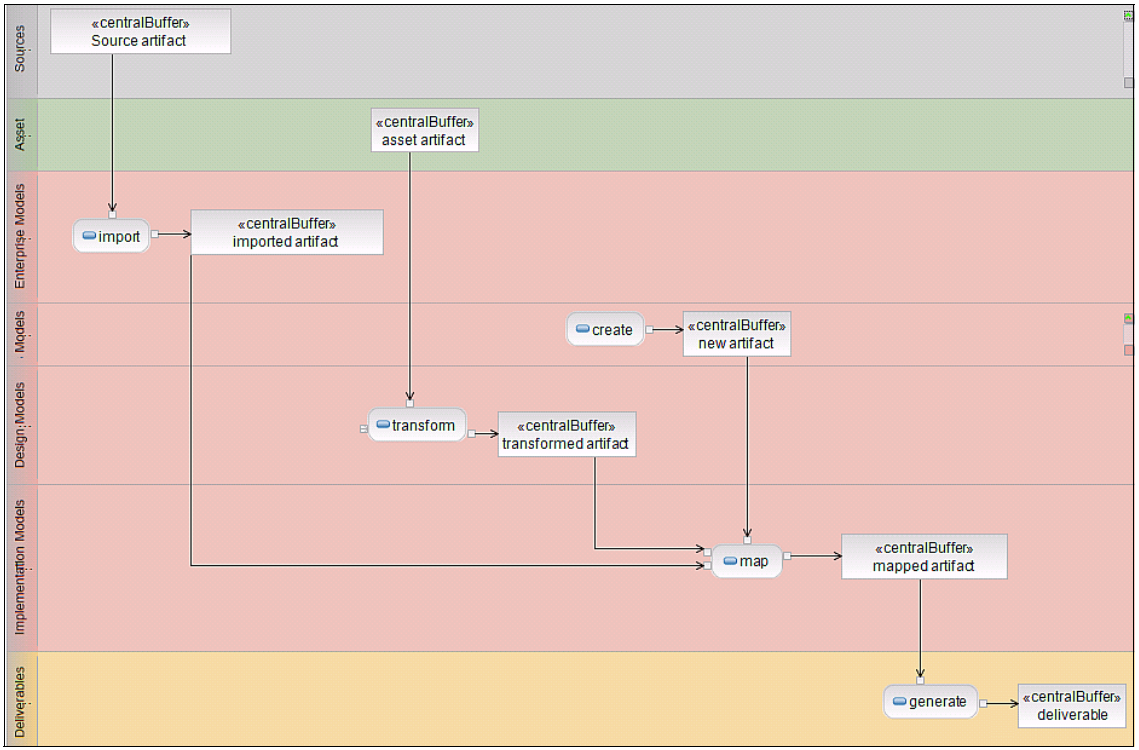

Figure 6-4 A metamodel for the modeling disciplines

In Figure 6-4 you see seven horizontal swim lanes. The four red-colored swim lanes in the middle represent the four modeling levels:

- Enterprise
- Business
- Design
- Implementation

In addition, there is a swim lane for sources and assets on top and one for deliverables at the bottom. In these swim lanes we place artifacts and actions. The artifacts are represented by square boxes stereotyped with `<<centralBuffer>>`. The actions are shown in ovals in comparison. We now place each action into the swim lane where the action takes place. These actions either create or manipulate one or many artifacts and produce an artifact as an outcome again. The arrows between the artifacts and actions denote the object flow. We skipped the inclusion of a control flow. This is represented by the horizontal axis. You can imagine this as a timeline, and consequently actions are executed from left to right.

Both combined structures, the modeling disciplines and the model layers, will help you identify the tasks that your teams have to carry out during the course of the solution implementation project. The reason for providing you with the different structural outlines is that it is incredibly difficult to describe all the artifacts, their dependencies, and their correlation to roles and the modeling layers in the context of an MDD. We hope to provide the greatest possible insight into all of these perspectives while still keeping a clean structure in this book.

6.2.2 Lifecycles for each discipline

The chapters in this book that describe modeling are grouped by *model levels perspective*. That perspective leans on the four modeling levels that correlate to the MDD layering definitions. In addition, two additional chapters outline the major tasks regarding data mapping between the various data sources that we operate with and the transformations that we have to perform between the modeling layers. These chapters are eventually concluded by one additional chapter that discusses the creation of BIRT reports to generate the documents for our final deliverables.

Now we look into each of the following modeling disciplines:

- Architectural modeling
- Business modeling
- Data modeling
- Data mapping
- Service design

Architectural modeling

Figure 6-5 describes the activities to be carried out in relation to the architectural modeling.

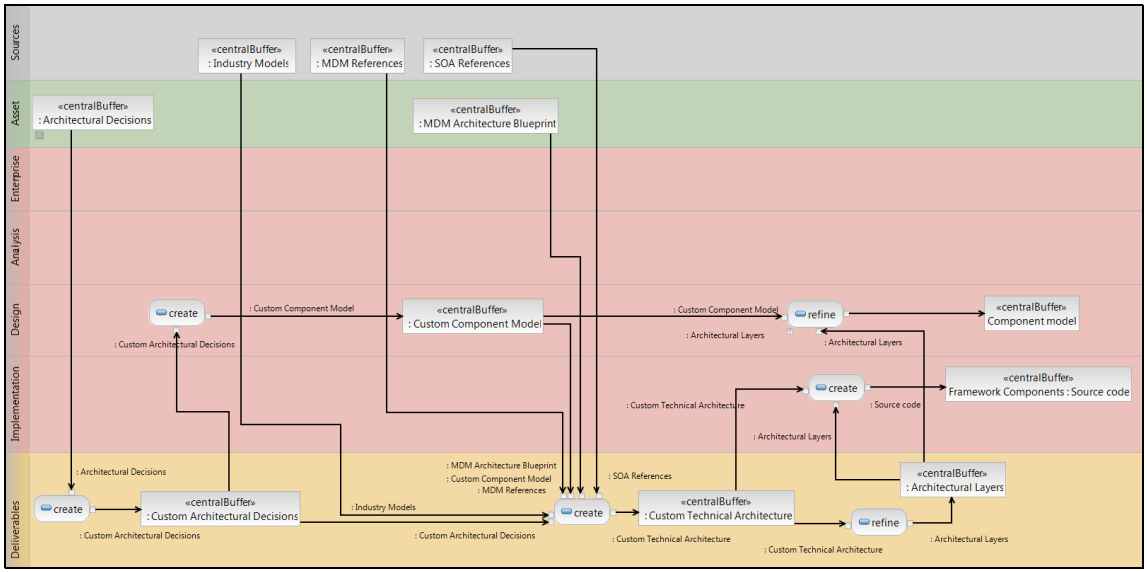

Figure 6-5 Architectural modeling

Figure 6-5 skips a few elements that are typical to architectural design on purpose. This is done because the artifacts skipped are not directly related or relevant to our methodology. However, for completeness we elaborate on certain artifacts not mentioned within Figure 6-5. Ideally, we already have a blueprint of our system landscape somewhere in our enterprise architecture drawer. If we do not, we must create it now to obtain the best possible overview of our systems surrounding the domains that we need to touch upon to establish our new master data management system. From there on, we can derive a system context by placing our new master data management system in the middle and connecting all relevant systems to it. The process of deriving it from the landscape provides distinct advantages. First, we are less prone to overlooking something that might be of importance. Secondly, we can explain immediately why something might be relevant to our new system. Thirdly, by following this approach, we can already determine from which systems we have to migrate information, which ones we will have to synchronize, and where we have to establish online transactional interfaces, if we choose to implement a transactional style master data management hub. Again, the advantage is on deriving this information by working with the models that we have created instead of thinking of the information out of the blue.

All the while we are performing such activity we are probably already making architectural decisions or making note of decisions to be made. More often than not, we are running into similar questions, so why not draw back upon a repository of architectural decisions already encountered in other projects facing similar tasks? Based on that we can create our own custom architectural decisions that document why something is the way it is. This approach is another kind of traceability. Now imagine that you could link these decisions directly to modeled elements. The process of documenting architectural decisions cannot be fixed to a single point in time though, but represents an ongoing process instead.

After we have established the system context and begun documenting our architectural decision, we are building up our component model for the system. Again, it might make sense to using best practices, but in the end, it is crucial that the model represents components that fulfil the functional and non-functional requirements to 100%, as agreed on the scope for the project. Initially, you will want to start with a logical component model representing functional business requirements, but then move on to adopt it to a technical component architecture.

The layering concept is a crucial aspect when you define the technical architecture. It is a common approach, but only one of the many ways to structure a system. Whereas too many layers might be perceived as a system too complex and posing too much overhead on the overall system functionality, too few layers let the system appear unstructured, and you miss the opportunity to categorize and group functions that logically belong together. This said, possibly the single most important layering aspect in respect to our methodology lies in the association of the domain model to a layer. We want to stress the fact that everyone in the project team must be aware of the place where the domain model abstraction resides and the impact that is having on the overall system. Based on that logical grouping through layers and other architectural concepts, you might have to revisit certain models again, refining them to reflect that logical grouping also in the components that make up the system.

Business modeling

Figure 6-6 describes the activities to be carried out in relation to business modeling.

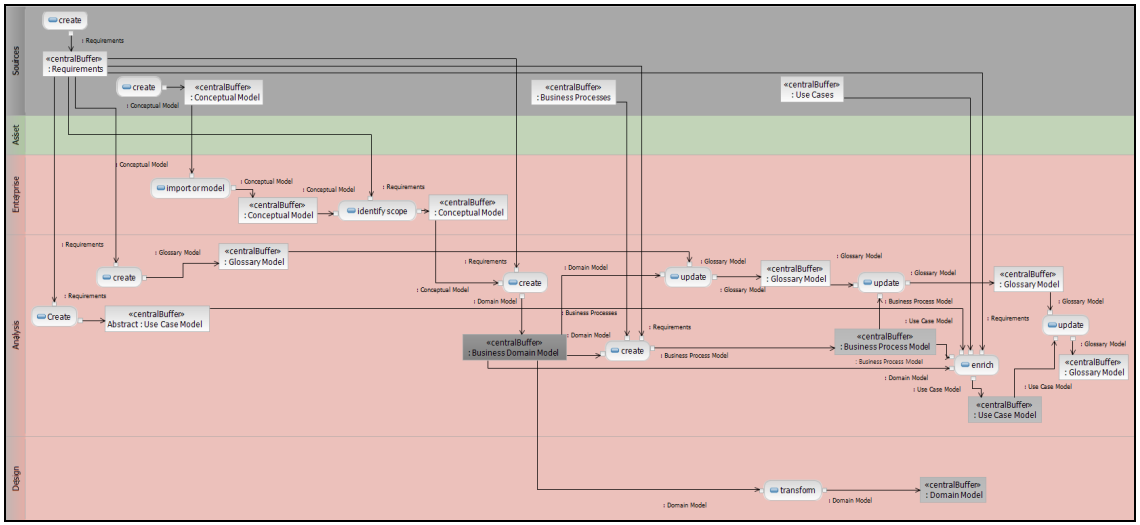

Figure 6-6 Business modeling

First and foremost, we start with the requirements. These should make it into a model, which is either stored separately but still providing the required traceability (as we can achieve using IBM Rational RequisitePro®, for example) or can be embedded as a model into the IBM Rational Software Architect based modeling framework. Everything that is designed or implemented needs to be traced back to any of the requirements defined.

As the scope is defined we now need to establish the glossary to set the terminology required throughout the entire project. This is not the only time when we are updating the glossary though. We need to continuously update it with all new terminologies that we discover during all activities that we perform.

Also based on the scope, the enterprise architecture and the business must work on defining the goals and the conceptual model based on those goals. After the conceptual model is fixed, the business domain model can be derived and created. Remember that the business domain model usually represents a much more detailed model than the conceptual model can ever represent due to the conceptual model addressing all data across all systems in the enterprise, whereas the business domain model only focuses on a subset of that. After the business domain model is established it can be transformed into its implementation model equivalent. A more detailed design will continue from there.

The business domain model also functions as the base for the business process and use case modeling. After all, these processes and use cases are there to manipulate our data and therefore need to reference these objects. These models can be created by importing from existing models, reverse engineering of existing systems, adoption of common industry processes, or invention of our own specific processes and use cases.

There is something noteworthy in respect to the use case modeling though. This is that we are defining a two-step process to build them. The first step creates use cases in the traditional way, much as it is described in much of the literature, tutorials, and trainings. There is not much of an added value to a traditional use case that holds most of its information in free form text and that only includes the actors and the use cases themselves as model elements. For us to be able to use the use cases for the subsequent service design, we need to enrich them with additional information.

This information includes a description of the input and output objects in order for us to be able to identify the scope of the use case in respect to the business domain model. Secondly, we are already adding activities to the use cases, which allow us a high-level description of the use case flow within the model itself. This is largely derived from the use case description itself, but storing this information within the model has benefits over storing it in free form. The benefit is mostly attributed to better traceability, but also that it already paves the path for the service design, which can then continue to enrich the use case activities with the design-specific elements according to design guidelines.

Data modeling

Figure 6-7 describes the activities involved in data modeling.

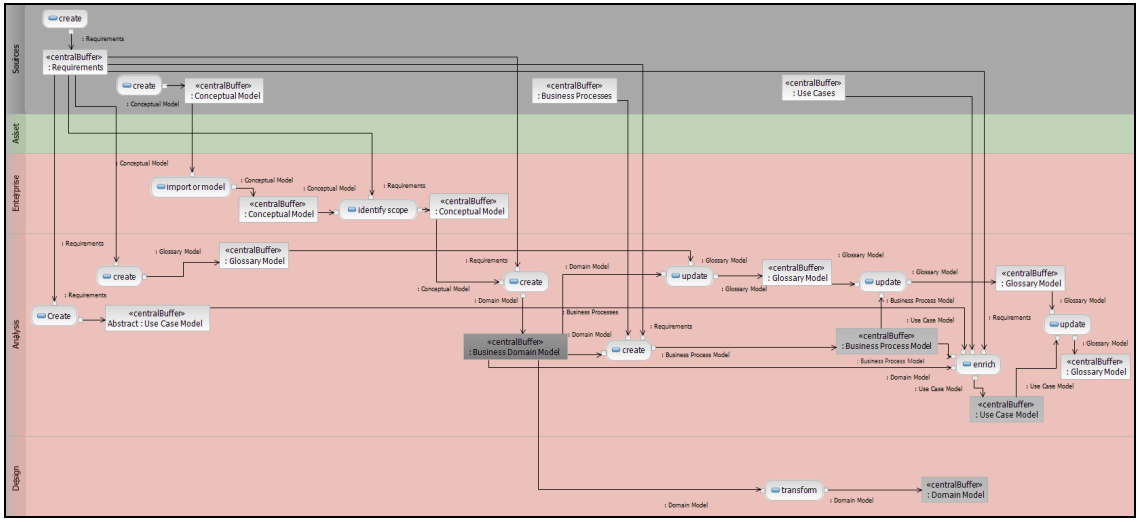

Figure 6-7 Data modeling

The first thing to be created is the glossary. The glossary is required during every step in our analysis, design, and development process. Because we most likely discover new terms and verbs during the process, we have to continuously enhance the glossary with every piece of knowledge obtained to keep it accurate and functioning as a base for design and development steps.

Enterprise architecture and the business now need to work on the definition of the goals and define the conceptual model from them. After the conceptual model is fixed, the business domain model can be derived and created. We need to remember that the business domain model usually represents a much more detailed model than the conceptual model can ever represent due to the conceptual model addressing all data across all systems in the enterprise, whereas the business domain model only focuses on a subset of that. After the business domain model is established, it can be transformed into its analysis and implementation model equivalents. A more detailed design will continue from there.

These design steps include the import of the customer source and target data models. This can be done in parallel by the appropriate business owners. These models play a vital role, not just in the following creation of the required data mapping deliverables, but they also can be referenced by ETL processes and tools that are responsible for the correct transformation between the models of the systems involved.

In respect to our task establishing a new master data management system, one of the more central activities and tasks has to be carried out now. This task includes the customization of the predefined InfoSphere MDM Server object model. The customization takes place based on the knowledge obtained through the domain model. This dictates the number of extensions or additions that we have to define and build into InfoSphere MDM Server. Because the custom master data management object models are a key deliverable in any master data management project, we need to run BIRT reports against the models to generate the documents required.

Last but not least, we have to transform the newly create logical data model into its corresponding object model. This is required not just because of the tool shift, but also because we require a more technically oriented model that can be enhanced for the technical source implementation. It is here that we require such a model when creating the required source code for the custom composite services that we are going to design.

Data mapping

Data mapping is one of the key disciplines to be performed when integrating InfoSphere MDM Server. This is largely required because of the product bringing its own data model to the table while the existing customer systems still have their own unique models. The output is required not just for the service design to be able to map between the right attributes, but also for the transformation step in any ETL process that we apply to load or unload data into or from the master data management system, as shown in Figure 6-8.

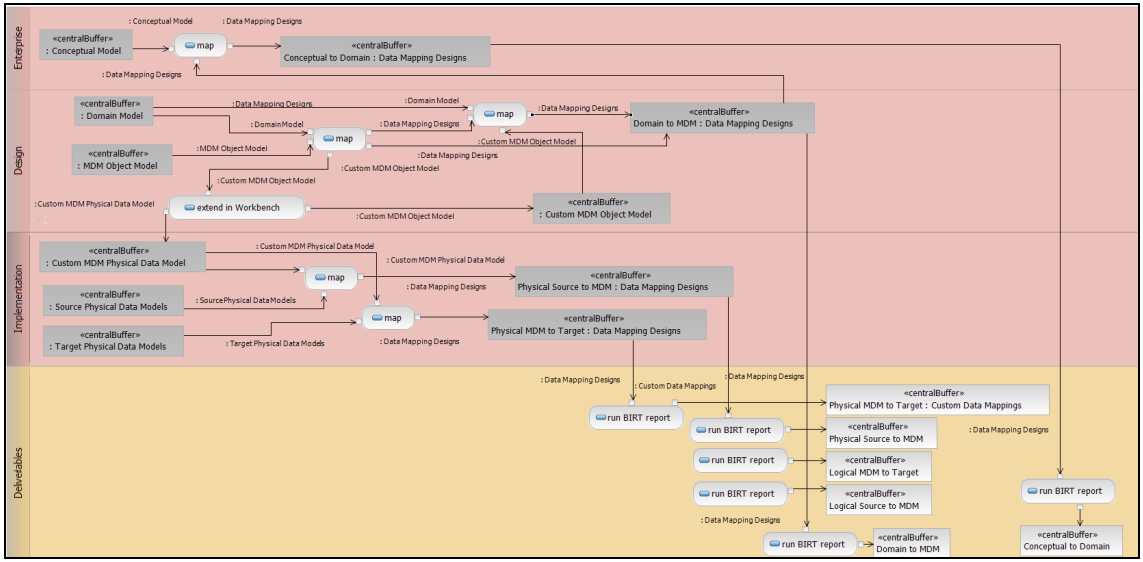

Figure 6-8 Data mapping

There is no given or necessary sequence between these activities. However, we need to map between the most important models. We assume that prior to us commencing with the master data management project, there have already been mappings, for example, from the conceptual model to all existing target and source models in place. Such mappings also allow for a system and data-specific scope definition. After the selection of an entity from the conceptual model has been made and mapped to certain entities in the logical data model of some systems, then only these systems with the identified entities are within the scope of our data mapping exercise.

Considering a standard approach to data modeling, we have physical and logical data models for any number of both source and target systems. We do not necessarily have to model physical and logical data models. In fact, the logical data model does not play any role in our data mapping. Instead, we only use it for transformations between the physical and UML-based object models, where we carry out the static and dynamic data mappings, respectively. Initially, it is important to start from the physical models. We need mapping between all physical models to be able to trace the flow of information. Secondly, we require mappings between the InfoSphere MDM Server object model and the domain model that we have established, in addition to a mapping between the domain model and the conceptual model. The key aspect of mapping against the model is that this can trigger more data modeling. It does so on the basis that we identify gaps in the existing data model in that we cannot map an attribute from the domain model to the existing physical data model. Now we need to break out to the data modeling to define the required modification before we can continue with the data mapping.

Based on these mappings, we have all information available to consistently describe the data flow between both the participating systems and across all layers, starting with the conceptual Enterprise Model on top, all the way down to the Implementation Model at the bottom of the layering stack. Despite the fact that we have all relevant information based on the mappings outlined above, we nevertheless mentioned that we also should have mappings between the physical and logical models. This is advantageous to ensure traceability between the layers and can be helpful to maintain a consistent set of models. Functionally it is not needed, though we always encourage maintaining traceability also to sustain capabilities that support impact analysis and therefore allow for later impact analysis when changes have to be made.

Service design

Figure 6-9 describes the task creating composite services for InfoSphere MDM Server based on some of the other models already provided or created.

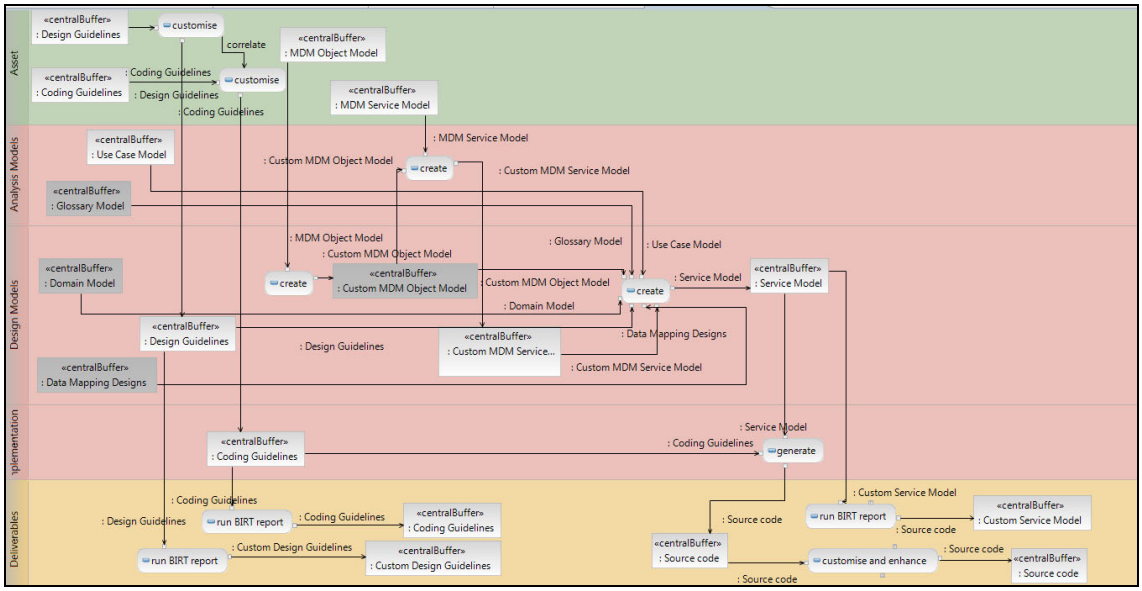

Figure 6-9 Service design

Before creating any design, it is important to establish the design guidelines that have to be adhered to throughout the entire lifecycle of the project. Because of our final goal of also generating large parts of the custom service source code, we also have to establish corresponding coding guidelines. For simplicity, consistency, and ease of use, we recommend embedding them in the models as well. There are many advantages to this approach. First, we describe the design in the environment where we carry out the design itself, and thus the guidelines are more representative, as though we describe them entirely separately. Secondly, we can embed all information required right in the model and therefore achieve tight integration between design and coding guidelines. The latter especially allows us to drive the code generation directly from the guidelines described on the model and therefore again allow for greatest possible consistency.

For the service design we have a number of streams that influence different aspects of the design. From a top-down perspective, we must make sure to cover all the functional business requirements. As part of the service model we focus on combining the requirements identified in the use cases with the objects from the Domain Model to describe business services, or more precisely business task services, as we call them. All of the logic here works strictly based on domain objects.

Secondly, we deal with a bottom-up approach, making sure that we offer sufficient operations from within InfoSphere MDM Server to cover all the functional (and possibly non-functional) needs. This is accomplished by combing the standard and custom master data management object models with the standard master data management service model to build the custom master data management service model. The services built range from low-level create, read, update, and delete services to more coarse-grained composite services. It is, however, important to understand that our basis for work remains the (custom) master data management object model.

Finally, we have to close the gap between those two models. That is, we break down operations on the domain objects into orchestrated operations on the master data management objects. These so-called entity services are again part of the service model and make up the most fine-grained level of services.

All of the above service design happens on the analysis level. Both the service model and the custom master data management service models are input to the code generation and result in source code artifacts.

7

The models

In preceding chapters much of the information is relevant to the topic of modeling. The result of modeling is the development of models, of which there are many, each with its particular use. As described, a model represents the abstraction, and a sufficient description of the problem domain for which a particular solution is being created. Models can generate source code and documentation and generate additional models.

This chapter provides information about the models and, in particular, the categories into which these models fit. It provides a good basis for understanding the functions of the particular models.

This chapter presents the following model categories:

- Enterprise models
- Analysis models
- Design models
- Implementation models

7.1 Enterprise models

Enterprise architecture and the modeling that comes along with it plays a key role in any enterprise. These provide technical direction aligned with the business strategy defined. Although this situation is not exclusively relevant to master data management solution implementations, it is incredibly important for master data management initiatives, especially because of the involvement of so many disparate systems that are spread around the organization. This book does not describe enterprise architecture as the sole focus. Instead, we describe the artifacts and models of an overall enterprise architecture model that we truly require to define, create, govern, and operate our final master data management system.

7.1.1 Enterprise architecture

Before we can point out the models that we believe that we cannot work without, we first look into enterprise architecture, what it is, and how it attempts to document the required organizational, business, and technical aspects. So, we do not describe everything that makes an enterprise architecture model. Instead, we focus on a single core model and demonstrate how we use this model in our methodology. We investigate two independent enterprise architecture methods to validate our thoughts about the models we require. These two methods are:

- TOGAF Enterprise Architecture
- IBM Enterprise Architecture Method

TOGAF Enterprise Architecture

Start by looking at the enterprise architecture definition of The Open Group Architecture Framework (TOGAF). The entire process is driven by the requirements for the new system and guided by a framework and its principles. A simplified diagram of the framework is depicted in Figure 7-1.

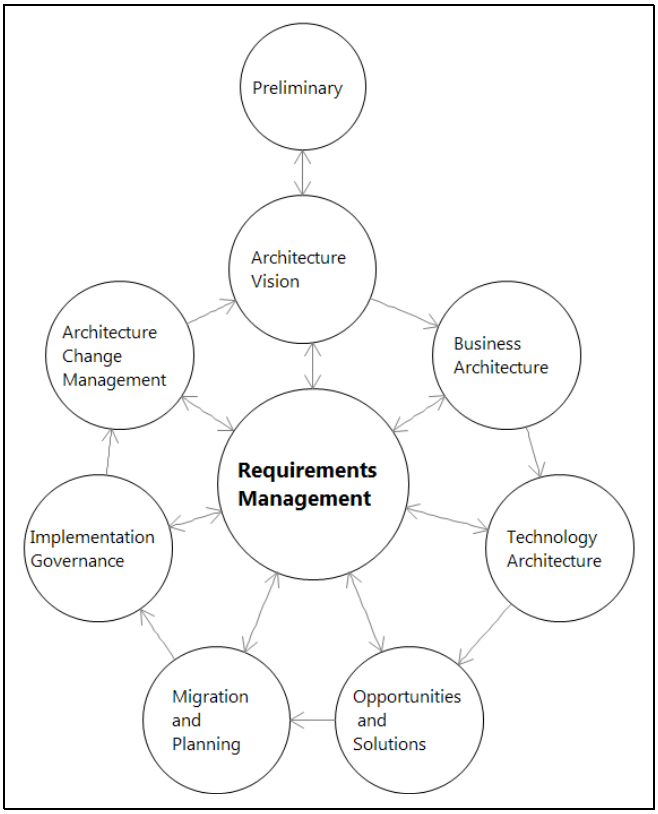

Figure 7-1 Simplified TOGAF diagram

Starting with the TOGAF, we develop an architectural vision that we describe in this chapter. The following are three key steps for the entire process, which revolve around the business, information system, and technology architecture. Even though the TOGAF and the corresponding IBM methodology are different in many respects, we take a slightly closer look into those three areas when describing the IBM enterprise architecture methodology.

After we define these three major cornerstones of the enterprise architecture, we can continue outlining the solutions, starting with migration planning and governing the solution implementation. Governance is vital and is often not observed, with the result that the system solution might deviate from the standards set and defined in some aspects. The consequences can be anything from irrelevant to significant. One example of that is that we are imposing tasks on one subsystem that should not be carrying that burden. Appropriate governance can help reduce the risk of such an issue and therefore ensure a consistent technical infrastructure that adheres to the standards defined. Change management is another part of the process that often is established only when the first change is raised. However, it is not difficult to predict the changes, so why wait for the appropriate change management process to be established?

IBM Enterprise Architecture method

In contrast to the TOGAF Enterprise Architecture method, we find a similar method within IBM, named the Enterprise Architecture Method (EA Method), as shown in Figure 7-2.

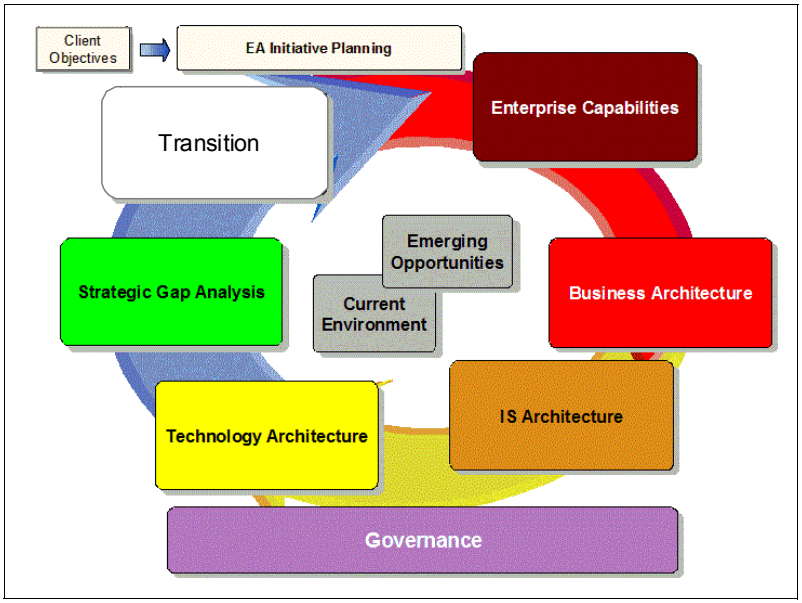

Figure 7-2 IBM Enterprise Architecture Method

While the naming deviates and the important aspect of governance is not embedded as a step in the process but rather accompanies the process all the way just as it should, we find the same three main ingredients. This method also accounts for the business, information system, and technology architecture. It is these three steps that we are going to look much closer at, because these steps are the steps in the process where most of the relevant solution details are described.

7.1.2 Core enterprise architectures

Our following investigation respects the following common major architectural models is based on the IBM Enterprise Architecture Method:

- Business architecture
- Information architecture
- Technology architecture

Business architecture

Business architecture requires numerous models, includIng:

- The business activity model
- A business event list
- Business KPIs
- Business structure
- Roles and locations

These models all play a major role in any project started in the enterprise. The business activity model and the business event list provide a base for any use case and business process modeling. KPIs are required to be able to determine whether the solution is operating to its expectations. Structural and location information provides a base for operational modeling when determining the placement for our solution components. After we include this architectural model into our scope, we rely on information contained here. In respect to our master data management system integration, however, all these aspects are only of lower priority. Although these models are important, embedding these models into our methodology is beyond the scope of this book. Also, we do not require any of these models to create the data mapping or service design that any master data management implementation project needs. Those reasons are why we exclude these models from any further observations and discussions here.

The remaining model, however, is of greater relevance to us. It is the enterprise information model. This model represents a high-level, conceptual representation of all entities that spread across all systems within the enterprise. Therefore, this model first provides us with important insight into the data and its relationship that we are going to design our master data management system against.

Second, and probably even more important, is a hidden fact of the enterprise information model. This fact is the traceability that such a model should provide. It is the traceability that determines which system implements any of the entities contained. Therefore, we are provided with a good initial and quick definition of the data-specific scope of the solution. As soon as all entities that are candidates for the master data management system are identified, we can pinpoint the systems that hold exactly these attributes.

Another vital aspect of the enterprise information model considers the reverse direction. Originating with the general idea that an enterprise information model should provide a high-level overview of the data persisted and exchanged in the entire enterprise, you must keep it up to date also when adding new entities into the system landscape. Such a thing can always happen when the master data management system becomes primary for new attributes and objects that previously did not exist anywhere in the enterprise. If, for example, we are introducing a new contract type into the organization by adding it to the master data repository, we must extend our enterprise information model so that it reflects the newly added entity on that conceptual level.

Information architecture

Information architecture is a slightly more technical aspect of the overall enterprise architecture. It remains on a logical level though and thus does not break down to specific technological aspects, such as products or technologies that are used to realize the concepts described. As described for the business architecture models, there are numerous models that make the entire information architecture model. These models include a model of all applications, data stores, and deployment units. Here it gets interesting again because modeling deployment units is part of the definition of the operational model to describe the final deployment of the solution that we are going to build. As such, we are required to come up with such a model subsequent to the creation of our component model.

In addition to the component model, we take the non-functional requirements (NFR) and use cases as input into our operational model. While it is up to every system and project to specify its own NFRs and use cases, there are certainly a number of standard NFRs and use cases that apply throughout the enterprise. Then, these components should be described on this logical level of the enterprise architecture. They also provide a good means of getting started with requirements and use case modeling for the project itself. The information can be only referred to and taken as a base.

As such, there are no new models to be created within the context of our methodology. The most important aspect therefore lies in using the information contained on this logical level of the enterprise architecture models. We suggest describing and adopting reference architectures here, such as the IBM SOA reference architecture or the IBM master data management reference architecture. These or similar architectures should be turned into enterprise assets to be able to create a consistent system landscape around the new master data management initiative. Equally important are the patterns that we can describe based on the reference architecture, the applications, and data stores defined in conjunction with placement guidelines for components, and considering the consequences and implications of the standard NFRs. What we refer to here is the requirement of a description of recurring architectural integration patterns. In respect to our master data management implementations, we already outlined the existence and relevance of the implementation styles from reference to transaction style. Depending on the selection of any of these styles and decision whether a certain system, including the new master data management system, should act as primary or secondary, we are faced with migration and possibly synchronization paths. How we are going to build these paths could be described by generic patterns that each specific implementation must adhere to.

Technology architecture

Technology architecture finally maps to concrete products, standards, and technologies. We describe these items based on components, nodes, and reference architectures. The major difference comes from these models that are technology-oriented as opposed to the logical nature of the information systems architecture. For example, if an enterprise purchases IBM InfoSphere Master Data Management Server, this product shown here is the concrete product. Components, integration patterns, and access are described based on the product capabilities.

Again, these models are not necessarily required for our methodology. However, we consider them a major influence towards our integration architecture in that we adopt as much as possible from the logical reference models, select the components from the products that we choose, and thus end up with our custom component model. In any case, we should be able to trace back to these models again when we are adopting one or the other aspect of them.

7.1.3 Modeling

From the previous descriptions, you can see that enterprise architecture plays a vital role in providing “food for thought,” meaning that we can and should derive much useful information from it. The standard use cases, for example, should present the baseline for use case modeling. The concrete solution architecture should be based on the insights gained through reference architectures and the predefined patterns.

However, we do not deem any of these models central to our methodology except for the conceptual model. If you decide to model the models that we do not explicitly describe, pay special attention to traceability. Everything that you describe could and should be traced back to. In respect to our methodology, we focus solely on the conceptual data model, which comes out of the business architecture modeling within the enterprise architecture. This conceptual data model is the only artifact that we actively integrate into our data modeling aspect. Our primary goal in embedding this model is to provide the traceability required for impact analysis and data-related scope identification.

In addition, we suggest that even at that level of the enterprise architecture, you introduce and maintain an appropriate glossary that pertains to all enterprise architecture relevant insights. This glossary should fit seamlessly with the glossaries defined by business and IT later on. Eventually, you should end up with a single glossary that holds all the different views and references for the different systems and the terms that they include. The references should be used when one term breaks down into more specific ones, as well pointing to additional useful information. Second, the references should also be between similar, or rather identical, terms that are named differently, thus identifying the known synonyms for one term.

Figure 7-3 outlines the relationship of the relevant modeling artifacts to the models and projects.

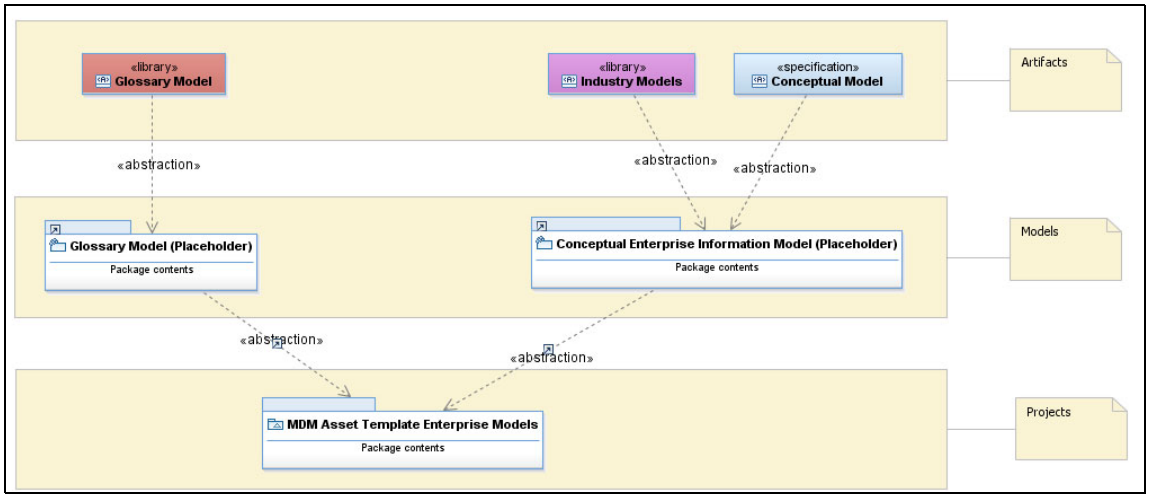

Figure 7-3 Artifacts, models, and projects in enterprise modeling

To describe the relationship, Figure 7-3 depicts the following distinct levels:

- Artifacts
- Models
- Projects

Artifacts

A thorough and consistently maintained glossary can add many advantages to any project apart from it representing the lingual backbone of a project. It does enable more complex operations based on the traceability that should be set up against it. Naming can be validated against the glossary, for example, when setting up a data model. Therefore, the tool's capabilities around this model add to the governance in that they enforce naming standards and prevent deviations away from them.

The Industry Models are referenced and listed here as one example of integration of appropriate reference architectures. They are included because they can contribute to the shape and content of the enterprise information model. Any additional reference architecture or source that you pull in additionally should be added to this metamodel.

In addition to all models outlined related to the enterprise architecture discipline, we add the glossary model into the equation. Despite the fact that this model is predominantly a business one, enterprise architecture adds its own distinct set of terminology to it. Project success, especially in large-scale environments, such as master data management initiatives, depend to a certain degree on everyone using the same language. Our methodology already caters to that by adopting a domain-driven design (DDD) approach. Glossaries accompany the domain to set the terminology required throughout the entire project. This time is not the only time when we update the Glossary. We need to continuously update it with all new terminologies that we discover during all activities that we perform.

Projects

To start setting up the enterprise architecture project, you need to create a project in the IBM Rational Software Architect. If you already have a template available, you can also copy your template project and create the identical structure again through pasting it as a new project.

Models

Consider the inclusion of a number of data mappings. This inclusion is helpful in the sense that the conceptual enterprise information model includes entities contained in all systems. While traceability already provides you with most information that you require, data mappings can add value in that they also provide meta information about the entities. This information can make the definition a little easier to understand and follow. Also, in comparison to the traceability, which can be hidden and non-visual, the data mapping represents a visual work product. Both, however, can be queried by appropriate reports. See the corresponding chapters in this book for further information about data mapping, traceability, and reporting.

7.1.4 Conceptual model

In this section, we provide you with the information to get you started with the conceptual model based on the IBM InfoSphere Data Architect. Refer to our description about the suggested project structure in 4.3, “Structure” on page 172 to determine the correct place, remembering that a glossary is always placed in a data models folder within the selected project.

The key to the successful modeling of the conceptual data model is to think in large logical building blocks that are valid for many areas of the enterprise. If you are starting from scratch, we recommend that you get further insights first. The IBM InfoSphere Industry Models provide us with a good starting point. You gain insights into the most sensible structures, based on global industry expertise. It is possible though that you already started drawing up a conceptual data model at some point in time. If you have, you effectively are presented with two choices:

- Keep it in the tool in which you modeled it originally.
- Adopt it into IBM InfoSphere Data Architect.

You are faced with a break in tools if you follow the first choice, thus impacting your ability to later perform impact analysis or model validations. That is, you will lose your ability to trace anything back against the conceptual enterprise information model.

Whether you import or draw up from scratch, the resulting conceptual enterprise information model diagram could look as shown in Figure 7-4.

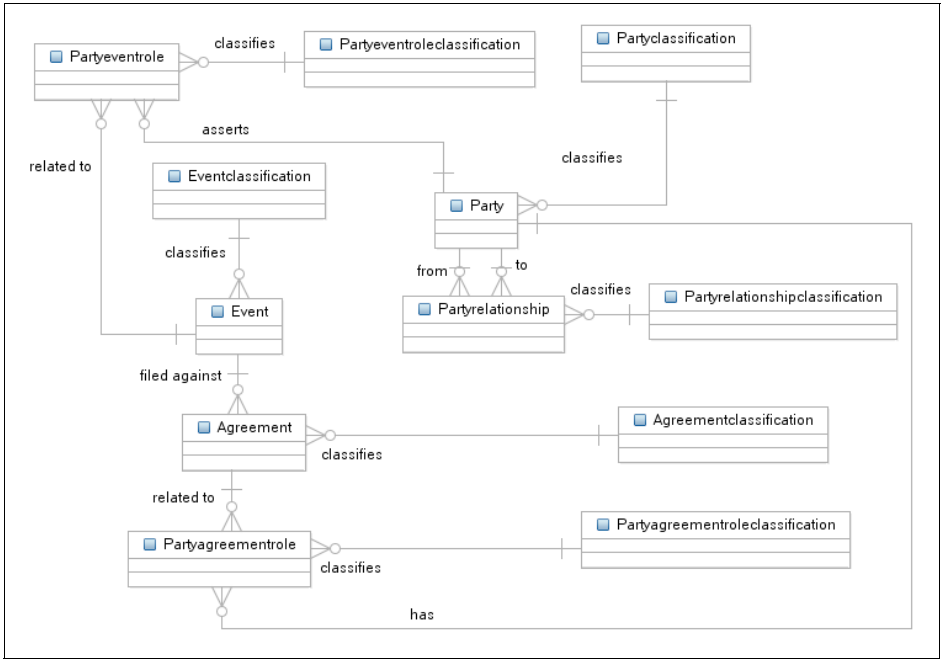

Figure 7-4 Sample conceptual enterprise information model

As you can see from this diagram, modeling the entities suffices. After all, all you need to achieve is an overview of the most important entities within your organization. The details come in at a lower level, the logical and physical data modeling. These tasks are in the design and implementation modeling. Hence, instead of using many specific entities, the conceptual information model is based on only few entities that are described by classifications. Attributes are not required on this level. The typical approach, therefore, is identifying and modeling the entities, the corresponding entity classifications, and the roles and relationships that are connecting the entities in a business sense.

7.2 Analysis models

The analysis models form the bridge between the business side that describes the requirements and the IT side that takes care of the implementation. It is important to find and agree on a language capable of satisfying the need of both parties involved. When you think of complex projects, it is important to involve parties with knowledge of the business needs. Business analysts or people from the IT side might not always be able to fully grasp the requirements. The business side, however, does not usually deal with formal models and is likely to feel overwhelmed by the complexity, level of detail, or simply the tools and language used. It is therefore the intention of both standardized UML approaches and our smarter modeling approach to refine the models and close the gap together. Business can still express their requirements in informal ways, while we can focus on of the most appropriate design. This situation in turn can be described and revised together with the business again before moving on to actual IT design.

Contrary to the lower-level models of design and implementation, the business should be considered as the owner of the analysis model. Business analysts and others support the creation and maintenance of the models, but are not the major drivers behind the contents.

Figure 7-5 outlines the main models and the artifacts that they represent.

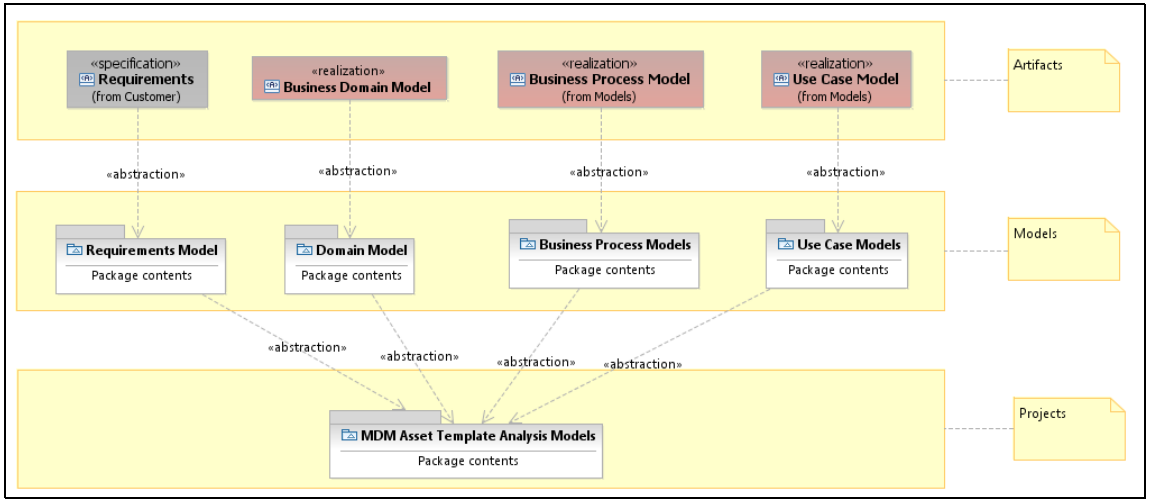

Figure 7-5 Artifacts, models, and projects in analysis modeling

As part of the analysis, we need to deal with various static and dynamic aspects that later influence the system design and implementation. These aspects are represented by the artifacts on top and manifest themselves in the corresponding models in the middle. All these models are bundled in the project drawn at the bottom of Figure 7-5.

The Requirements Model is created at an early stage in the project and provides both functional and non-functional aspects for the master data management system at a high level. As such, it can cover both dynamic aspects, such as response times and scalability, and static aspects, such as the number of records to store. Apart from traceability aspects, it is not directly related to other downstream models.

The domain model is the centerpiece in terms of static modeling. It captures all relevant business entities, their relationships, attributes, and lifecycle in detail. Apart from providing common wording and clear definitions, the entities from the domain model are referenced in the dynamic models. The model is refined, transformed, and mapped later as part of the design and implementation models and eventually results in the definition of the physical data model.

The business process and use case models capture the dynamic aspects of the master data management system and its surroundings. The business process model shows the coarse-grained processes within the organization and how they break down into individual use cases. The use case model then describes the underlying functional requirements in detail. In later project stages, the use case model is transformed and further detailed to form the basis for service design.

Our preferred project structure reflects the above models. The analysis models project contains independent models for each of the previously mentioned aspects. We explain each model type in the following sections:

- Requirements Model
- Domain Model
- Business Process Model
- Use Case Model

7.2.1 Requirements Model

Requirements are best managed where we design and build the remainder of the project. Such a statement comes naturally to a model-driven development (MDD). It demands appropriate tool support or integration of the correct set of tools.

In our methodology, we allow for either way. You can select the best possible tool that also integrates with the other design aspects and development toolset. We base the description of our approach on the IBM Rational toolset, consisting of IBM Rational Software Architect and IBM InfoSphere Data Architect in addition to that. You can also include complementary tools. One that immediately comes to mind is IBM Rational Requisite Pro. It integrates well with both Rational Software Architect and IBM InfoSphere Data Architect. It works in such a way that you can describe your requirements in a structured way. You then have the ability to work against those structured items within Rational Software Architect and IBM InfoSphere Data Architect to create traceability information to them. The downside is that it requires yet another tool, which is a slight burden to take considering that all we are interested in is the traceability. Another possible tool and approach, and similar to the IBM Rational RequisitePro based approach, is using the much more modern IBM Jazz™ platform, namely the Rational Method Composer. The pitfall of having to learn and understand yet another tool remains.

All we are interested in with respect to our modeling methodology is the traceability. This aspect is a crucial one of MDD. We must be able to trace requirements to the components that implement them. For some tasks, we must operate on a much finer granular level than components, but essentially the impact remains the same. Here we present you with another alternative where you can use the same tool that we already positioned for this methodology, IBM Rational Software Architect. You can also model requirements as artifacts in a separate IBM Rational Software Architect project. The resulting project could look like Figure 7-6.

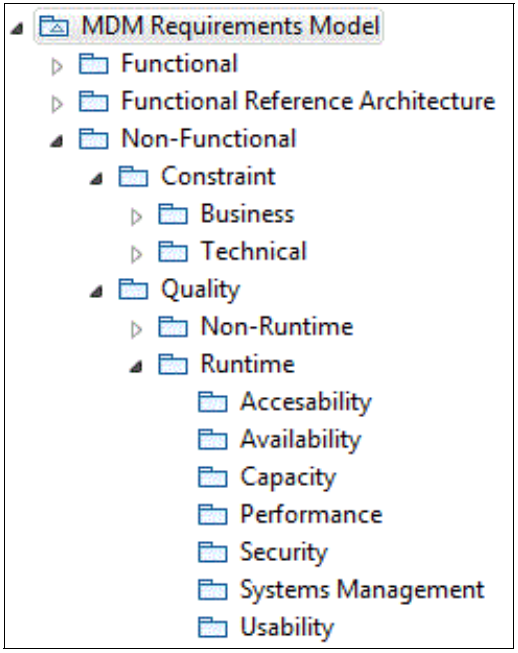

Figure 7-6 A requirements model in Rational Software Architect

The easiest way to add this information into the tool is probably by manually creating these entries as part of the project. Each folder should contain the requirements that describe this category with an artifact element. After you have the requirements persisted, you are now able to use them like any other UML element available within the realms of IBM Rational Software Architect. In particular, in respect to our traceability requirement, you are now able to link any UML element to any of the here-described requirements through appropriate realization associations. The advantage is in the ability to describe the requirements in the most integrated way without having to adopt yet another external tool, while avoiding the complete break in the tool chain compared with an approach using Microsoft Word or Excel.

7.2.2 Domain model

The domain model provides us with a static view of all the data that is relevant to our newly designed master data management system. It serves many purposes, both for the immediate project and for the organization:

- Extended glossary and common vocabulary for discussions and analysis
- Reference for data handled in the dynamic models
- Analysis foundation for data mapping and other data-related requirements
- Specification for logical and physical data models

Each of these aspects reflects in the requirements that we must cover when designing the domain model for our specific master data management project. The latter point is not relevant when we are mapping to InfoSphere MDM Server. From a mere technical point of view, establishing the model is fairly simple. The most commonly used UML approach is a class diagram. Fortunately for designers, class diagrams are available in various modeling tools and offer much flexibility. However, when you think of the generator or transformation requirements as needed in an MDD, then too much flexibility poses a problem instead of a benefit. Not only that, as soon as more than one designer works on a model, we might likely end up with slightly different approaches to different aspects, ranging from naming to which way attributes are defined. Eventually, this situation reduces maintainability and usability in the end from a business point of view as well.

Overcoming this issue is not a simple task. Projects vary widely, different client environments have different requirements for a Domain Model, and it is unlikely that we are able to anticipate all of them. What we can do is recommend how to design the model, which we are then able to transform because we are familiar with the encountered concepts. Everything that goes beyond the recommendations can be introduced into the modeling approach, but needs an adequate reflection in the transformers and downstream modeling approaches as well. If you decide that you need additional concepts, be sure to remember this information.

7.2.3 Elements

The domain model is more than a component for our master data management project. Maintaining a consistent vocabulary that captures business processes and use cases is something that helps the business to provide structure and understanding. Such models not only provide aid in the project, but we can later reuse them to describe organizational changes or simply to ease the process of onboarding new people to the project.

Start by looking at the UML elements that we recommend for use in the domain model:

- Primitive
- DataType
- Class
- Enumeration

Primitives, or Primitive data types, are the low-level building blocks that make up the domain model. Contrary to programming languages, which typically bring along a fixed set of predefined primitive types such as integer or char, UML allows the definition of other Primitives in addition to them. In our insurance example, we have a unique identifier attached to persons we are dealing with that is referred to as the *Customer Master Reference*. Instead of simply declaring it as a string and hoping that everybody knows and agrees on what the information looks like, we could define a Primitive named CustomerMasterReference and use it as the respective type for all such attributes. In this context, Primitives are actually a good example of the different purposes of the domain model.

From the implementation point of view, it is not fully clear to which extent the information is used. In the database, we might not care about it at all and simply declare the column as char or varchar. On the interface level, if we think, for example, of an XML-schema based definition, we can use this information to automatically perform validations on the data. However, even there we can simply use a string.

Primitives are an important carrier of syntax information from the specification and even the business perspective. Assume that we have two different objects in our domain model that have a CustomerMasterReference-attribute. While they might be semantically identical when simply declaring them string attributes, there is no way to determine whether the attributes can actually be compared with each other or if they use different formats. Using the Primitives, we can close the gap and know immediately the exact format.

One limitation of this type is that from standard UML the Primitives do not provide any means of carrying additional information, such as regular expressions, length constraints, min/maximum values, or any other format details. We suggest that you use standardized tagged values using a UML profile to add this information. This information can then be considered part of the implementation.

When it comes to more complex information that needs to be handled, we recommend the use of DataTypes. DataTypes are little more than multi-field primitive types. They are similar to classes, although this comparison might be confusing at first. While we think of classes as having an identity, DataTypes are only defined by their values. We can have two independent class instances with the same attribute values, but two DataTypes that have the same value are considered identical.

The use of a DataType also implies that the contained information does not exist outside a container, which is typically a class. An example of a DataType could be a phone number. While we must store the information as an indivisible string in the Primitive, in the DataType we are able to identify components such as the country calling code, the area code, and the actual phone number. On the analysis level we recommend the clean usage of DataTypes, as opposed to classes.

As we get closer to the implementation, it might not be possible to uphold this separation. Some programming languages support data types (for example, a struct in C), whereas others, such as Java, do not. Arguably, this strict separation is detailed, and from a business point of view it is not required or logically understandable. If you find yourself unable to make distinctions between class and DataType, start of with a class. The worst case is that you end up with too many business entities.

Classes are the most important elements of the domain model because they represent the actual business entities. Each class can have a number of attributes. For the attributes you can use the full capabilities of UML to add information, for example, visibility and cardinality. The transformation algorithms need to be able to handle the modeling concepts adequately. Classes become objects with different attribute values representing all the information that the system needs to handle.

In most cases, we do not want to describe individual data values on the design level, because it does not make sense to include such static or sample information in the design. There might be special cases that require the consideration of data values in the design. Think of our insurance sample. We can have different marital states for our clients. Now assume that we have different rules for premiums or other functional logic embedded in the master data management system. We need to find a way of explicitly referencing someone who is married as opposed to someone who is not. If we look at the previously introduced UML concepts, the closest thing that we currently have storing this marital status is the Primitive. But can we rely on everyone to know that the value to store is ‘Single’ and not ‘S’ or ‘Unmarried’ or any other value? No. We need a fixed anchor point to rely on and UML has just the concept to handle this: The enumerations.

An enumeration allows us to define not only a type, but also include literals that represent the future values. By providing these agreed-upon values, we establish common wording and at the same time provide a point of reference to specification, design, and implementation.

7.2.4 Relationships

Capturing the business entities is a helpful starting point because it provides a glimpse of what you need to know, for example, about a client. However, a set of unrelated entities does not get us far when it comes to the overall system. We also need to identify and characterize the relationships between the elements. Start by looking at the forms of elements that we consider helpful in this context:

- Association
- Composition
- Association class

An association establishes a plain data relationship between two elements. A composition, in comparison, is a more powerful form of association. The latter type of relationship includes a directionality that indicates a lifecycle binding. We consider one object to be dependent on the parent object that is holding the composition. If the parent element ceases to exist, this event also ends the existence of the dependent element. This method is a powerful form of expression in the domain model when it later comes to system design.

Using a composition between classes A and B means that we can now define a use case that deletes object B without having to mention object A explicitly. It is automatically deleted too. This situation apparently adds complexity to the overall design, while keeping it simple. We encourage you to use this relationship. If you do, you also must acknowledge this relationship in the service design, implementation, or transformation between the different models.

The final element, association classes, is simply a UML add-on to the regular associations. It allows us to define additional attributes for an association.

7.2.5 Classification

UML provides different information about a relationship. For the domain model, we want to highlight the following items:

- Cardinality
- Name/role
- Direction

Cardinality is one of those properties that most of us are familiar with. Using the cardinality in the domain model, we detail the minimum and maximum occurrence of the related elements. Typically, we find 0 or 1 as the minimum cardinality, indicating optional/mandatory elements, and 1 or * as the maximum cardinality, indicating either a single element or a list of elements. There are a few exceptional cases where we might want to strictly limit the number of elements, for example, 2 - 4.

An association adds little value to the domain model if we cannot describe its meaning. We need to describe it to keep the design meaningful. As with all other UML elements, adding an appropriate description helps, but unfortunately the information is not machine readable. We need them, for example, in attribute names, table names, and method names that all need to reference the association. We need another anchor point for that information, and the name property immediately comes to mind. We can provide a meaningful name as both identifier and characterization to our association, but the more complex the models get the more difficult it gets for two reasons:

- Direction
- Ambiguity

Looking at our sample Domain Model, we find an association between *policy* and *claim*. You certainly can come up with a good characterizing name for this association, but does it work for both directions? Claim “isFiledAgainst” Policy, but not the other way around (The association does not work in both directions). Policy “hasAssociated” Claims, but less so the other way around (The association works in both directions, but does not work as well if you say "Claims "hasAssociated" Policy). The questions are which name to pick and whether there is a major and a minor entity that indicates something like a natural reading direction. Even if there is, we sometimes must traverse the association the other way round. It is then that we use a wrong and possibly misleading name. This situation does not increase readability and maintainability of the design and implementation code that we generate. In the worst case, we have two entities that share more than one association.

Fortunately, UML offers an adequate container for the problem that we are trying to address. Roles can characterize each endpoint of an association and give the related object semantic meaning in regard to each association. These roles are helpful, because it allows us to assign different names based on perspective. Coming from the claim, we can address the policy in the role “filed against”, whereas looking from the policy we can refer to the claims as “has associated”. The names are unique, meaningful, and we can continue to use them downstream when we move to the actual implementation of the model.

This situation leaves us with an important aspect, which is n:m associations. As an example, each beneficiary is entitled to multiple policies, and each policy can have multiple beneficiaries. We must translate this relationship into an association class or table on the physical data model to be able to resolve the references. Now we have another problem, because we do not want to create two association tables just because we have two role names. We require a single name to serve both directions. While this configuration is not as precise and meaningful as the roles, it resolves the issue regarding the name of the single association table. This situation is why it is important to provide a name on the name property of an association. Think of the name as a short description of the association, which is independent of the roles. It merely adds additional details about the purpose of the connected elements.

We can express navigability using arrows. In our experience, dealing with this level of detail is not adequate for the analysis level. Instead, it is the task of the designers to enrich the design models with such detail before handing their models on to implementation. Do not attend to this detail unless you are sure that the visual information about directionality provides additional business insight.

7.2.6 Guidelines

Given our knowledge of the above UML elements, a question that frequently arises in the context of domain modeling is whether to use attributes or associations. What criteria can we apply to determine when to use either of the two on the modeling level, or are they even fully equivalent? The answer, from our perspective, is that it depends. From the point of view of transformations or the final result, it makes no difference at all. We could abandon the concept of associations, or we can define every fact that cannot be reduced to a simple Primitive, as an association.

In some environments, you might want to draw the line based on the types of the involved UML elements. For example, every relationship with a class is an association, while every relationship with a DataType is an attribute. In other cases, you might be focused on visualization. When we are using a class or DataType as an attribute type, it simply appears on the diagram with its name, while the internal structure remains invisible. To see the details, the respective element needs to be displayed in the diagram as well. It comes down to the individual purpose. Our suggestion is to establish a guideline that is valid for the entire domain model, regardless of the approach that you choose. But do not leave it to the individual designer to decide, because that can lead to an inconsistent model.

One difficult design-related aspect is related to multiple inheritances. While this situation presents you with tremendous flexibility in respect to your design, it also introduces ambiguity and incompatibility with technical implementations, because this construct cannot be realized in some programming languages, for example, Java. Based on our experience, we discourage you from using it due to the problems and work that it causes downstream. Sometimes, however, it is the simplest means of describing a particular problem. We believe that most cases of multiple inheritance can be resolved either through introduction of appropriate business rules or in the worst case the introduction of redundancies. Designers that pick up on this situation attempt to resolve this issue with you, in case you do introduce this concept. You can find more information about this topic in 7.3, “Design models” on page 334.

With the flexibility offered by UML, it is likely that at some point our automated generation approach fails because it encounters a concept not known at design time. This situation is especially true for custom stereotypes that are likely to occur in different domain models. Some UML tools allow for the introduction of new stereotypes by simply typing their name into the corresponding field, which is not helpful for a structured modeling approach. Inside Rational Software Architect we are in a more comfortable situation, and therefore can reduce this problem to a simple guideline. To introduce a new stereotype, define a UML profile. When doing so, simply check the transformation logic to make sure that the newly defined elements are correctly reflected. If not, update the transformations so.

Up to this point, we established several business terms. We have even been able to put them into context by establishing relationships between these terms. So far, we have not provided a single definition or description that would help us understand the purpose of any such model element. When developing the Domain Model we must make it a habit to document each model element that we introduce. UML offers documentation fields for the following levels in the domain model environment:

- Packages
- Classes
- Attributes and literals
- DataTypes
- Primitives
- Associations

Documentation should always be added at the most specific location available. For example, based on the visualization offered by your UML tool, you might be tempted to document attributes on the class level simply because the class documentation property shows. Instead, document the purpose of the attribute on the attribute or the semantic meaning of an association on the association. Only this strict application of element-related properties ensures correct treatment and the greatest possible consistency, readability, and maintainability on all design and implementation levels.

The domain model should have a reasonably stable state before you begin to work on use cases or other artifacts because of the larger number of dependencies against the domain model to many different artifacts. Saying that, we are aware and also believe that a Domain Model is unlikely to be in a final state after the first iteration or even after we start designing other models. Still, we should try to reach some stability before putting it to use. After putting it to use, we must introduce changes to the model in a controlled and organized way. This task must be done while considering the implications that this task has on other parts of our model, our software, and the entire project.

Based on everything described, an exemplary domain model could look as depicted in Figure 7-7.

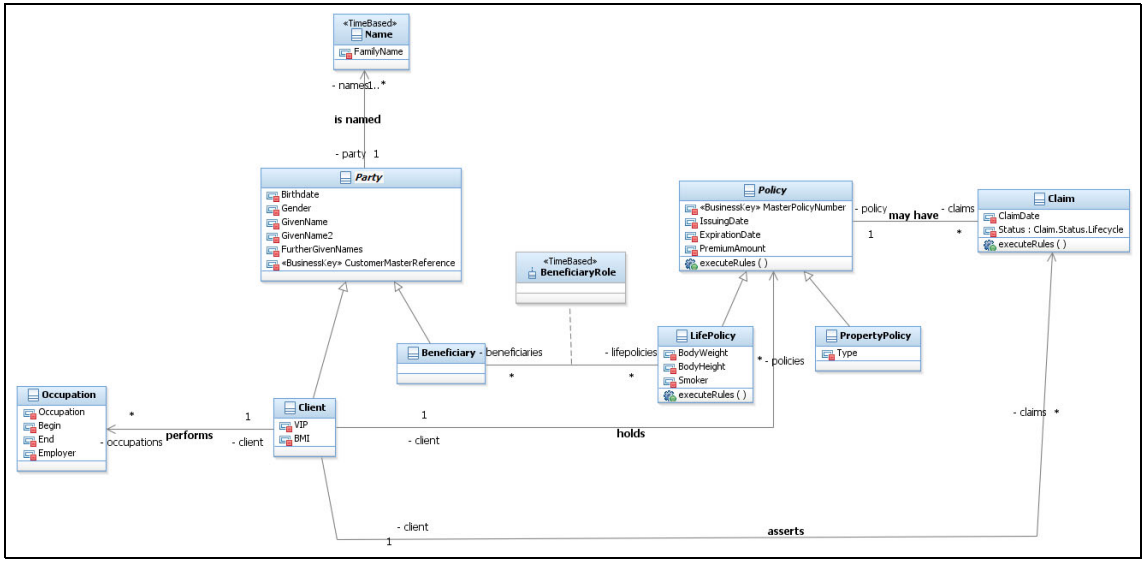

Figure 7-7 Sample domain model

Now that we have a common terminology and know what business entities and their relationships are, we can move on to the more dynamic aspects and capture how they are related to these objects.

7.2.7 Errors and warnings

Many projects leave the specification of error handling, the corresponding errors and warnings, to the late phases of the project. Although not all exceptional situations are foreseeable early in the project, you can anticipate some of them. Some of the situations that you can expect to encounter are related to the rules, lifecycle, and other integrity constraints you enforce on the domain model. Later on, you are able to account for errors that originate in the exception flow of the use cases. Many of the technical errors cannot be known and are also not part of the business model. If the application fails internally, you must take care of this failure during the design or the implementation phases.

One of the goals in business analysis modeling is to define adequate business error messages and warnings. For example, what looks like a “not found” error to the developer might mean a process violation to the business.

Similar statements are true for warnings. An optional association in the domain model might be fine, while the same association might be interpreted as a warning on the business level. In this case, you should ask further questions or take additional steps in a business process.

Therefore, errors and warnings that are related to business functionality should already be specified on the analysis level. This specification prompts you to consider the relevant scenarios, provide an adequate error message, and decide additional details that might be required to interpret the message. For example, “unable to file claim, pending claim needs to be completed first” might seem like an adequate error message when adding a claim. But what if the surrounding use case is a batch load and multiple Policies are affected? You might need additional information, such as a MasterPolicyNumber, to resolve the situation.

What you require can eventually be described only by your business, so those requirements belong in the business analysis model. In our experience, reworking the message at a late stage in the project is a major effort, because some of the surrounding services might not be designed to provide the additional information. Although this concept is not part of the core business entity model, the domain model is a good place to include these aspects. You should ensure a clean separation between the model entities and the supporting information. Within the overall hierarchy of models, you are still at an early stage, but there is no extra effort to work with structures that you can later reuse in the design or implementation levels.

Start by designing the class to represent a single error, that is, an ErrorObject. At a minimum, this object should offer containers for a unique code and a message. The idea behind the separation is that a unique identifier is combined with, for example, localized error messages. Continue to set up the ErrorPool, which is the single version of the truth for all errors within the application. Use a structural separation between those errors; this separation is for business reasons instead of only technical reasons. This separation needs to be considered only at lower model levels. You can also use a structure that is set up by logical subdomains or other criteria to avoid overloading the model with too many ErrorObjects in one place. Use specific stereotypes, such as <<businessError>> or <<businessWarning>>, to characterize the nature of the error. As you gather information about more errors, you add them as separate ErrorObjects with their corresponding information to the ErrorPool. Ensure that you provision a unique code, possibly a default message, and a description of the scenario as part of the documentation. On a lower modeling level and for more complex messages, add additional attributes and corresponding getters/setters in the specific errors for the required data to adequately describe the situation.

Establishing this type of an ErrorPool has multiple advantages. When you need to look up an error and see if exists or see what the scenario behind the error is, there is only one place to look. Designers or developers should be able to identify whether an error scenario has been previously defined, so there is the potential for reuse. You should avoid creating unnecessary codes and messages and assigning the same codes for different errors and purposes. You can avoid these issues by using the error pools described in the business analysis mode.

You should also consider traceability, which has multiple versions. For development and maintenance, if you consistently use the ErrorPool, you can effectively determine where an error is used. You can explicitly refer to the ErrorObject within the use case and service models, and introduce similar structures into the final implementation. Traceability is also used during testing and debugging when you have a point of reference that connects the error code at run time to an entry point in the model. It is then possible to trace the error back to its possible origin in the overall system.

The concept behind warnings is identical and consists of the following elements:

- WarningObject
- Stereotype <<businessWarning>>
- WarningPool

Using these descriptions, make the definition of error scenarios and adequate descriptions part of the business design and introduce a single point within the model to contain all the available errors. In the remainder of this book, we refer to these structures in the models on the analysis and design levels.

7.2.8 Packages

In a complex project environment, the domain model quickly becomes an enormous collection of information. You use the model on different layers of modeling while adding further details and information. It is important to have a viable structure as a foundation, or you are unlikely to maintain or find the information contained within the model.

Here are three things to consider when building your model:

- Business entities versus supporting information
- Logical building blocks
- Model elements/element types

Moving forward with functional modeling, you quickly realize that if you think of the domain model as holding all relevant data structures, it eventually covers more than your core business entities. You are then not referring to data types or associations, but also to such things as the maintenance of a list of all relevant errors or warnings. Such information should be structurally separated from the entities to avoid confusion (this element is our first structural element). You should have separate packages for the more functional aspects and the actual business entities that you handle in the functional models. Within the domain model, create different packages. You could use the following structure and distribution rules as a guideline:

- BaseConcepts
- DomainConcepts
- SubDomains

The BaseConcepts package is a predefined structure that is required in every project. The contents are not related to the business entities or to the specific domain that you are currently developing. The error or warning pools are good examples of information you find in these packages, as they are basic infrastructure components that you use in any implementation project. Inside the BaseConcepts package, you maintain subpackages that hold the different concepts. For example, you can introduce ErrorHandling and WarningHandling as packages to store the respective concepts. Within these packages, you can further chose to introduce more packages to distinguish between technical and business-driven errors, or possibly break down all errors into domains, depending on the requirements in the individual project.

The DomainConcepts package holds the structures required for the domain, but does not make up the actual business entities. Data structures required for searches are a good example. Although the search structures are likely to deviate from your regular model entities, they are specific to the client domain. If you think of our insurance example, we could define a PartySearch-object. This object is not an element of business, as there is no real world entity reflected here. Alternatively, this object contains domain-specific elements, such as the CustomerMasterReference, making it unsuitable for storage in the domain-independent BaseConcepts package. For each of the concepts you identify on this level, you should create separate packages within the DomainConcepts package, to have the individual building blocks clearly separated.

The SubDomains package holds the actual business entities. This package represents all the contents that make up your domain model. You should provide an appropriate subpackage named Signals in the domain to hold the necessary model elements.

Although you have successfully separated common, domain-specific, and business-related aspects, you still have many elements in the SubDomains package. Therefore, you should separate the package into individual subdomains. Such subdomains are defined based on cohesive principles. Include everything that is required so you can define the independent blocks of business logic, but keep subdomains small enough to create enough modules that can be handled without external dependencies.

In a best-case scenario, you have these subdomains reflected in the overall project approach, so you can develop both the business logic and the domain model in each subdomain. In complex environments, you could also decide to introduce multiple nested subdomains. There you could split the example insurance domain model into the subdomains Party, Policy, and Claim, and then create further subdomains within Party for Beneficiary and Client. Because the domain model usually is a little larger than our small sample domain model, we can often justify further decomposition.

Based on your modeling environment, your package has various elements, such as Classes, Primitives, DataTypes, and sometimes individual Associations. In complex models, you might be overwhelmed. Although you could reconfigure the environment to display or filter elements per your individual preferences, you must repeat these settings in every designers' workspace.

You can also introduce another structural package. Depending on the elements used in your model, you can create packages by the type of modeling element, such as DataTypes, Primitives, and Associations. For Associations, you sometimes have a package already in place that was created by IBM Rational Software Architect. Now the business entities (in Classes) remain the only elements on the root-level of your domain package.

The transformations that move our models to the next level are mostly unaffected by these structural considerations, but there are exceptions. These exceptions require *fixpoints* to locate all that elements that they need for their activities. The fixpoints are the SubDomains, BaseConcepts, and DomainConcepts packages, or whatever package structure your transformations work with. They are unaffected by the internal structures of these fixpoints; they make sure only that all elements from the hierarchy are considered.

7.2.9 Rules

Rules are an important component of our business logic. You must include them in the business analysis models. Rules and the domain model are closely linked, especially if you consider that errors and warnings are part of the specification of the domain model. Rules are difficult in respect to their definition, classification, and use. Although there can be control flow-related rules that, for example, calculate new data values, we do not include them in our analysis, because we consider them to be a use case flow-related aspect. We distinguish between two different kinds of rules:

- Object-specific
- Object-independent

Rules are object-specific when they can be bound to a single domain object. You must be careful with interpreting this statement, as it is solely a statement about the execution of the rule and not the data it can handle. Looking at our insurance example, we can define an object-specific rule on the Policy. It fires when any objects are manipulated, which can then traverse through the domain model to evaluate other related objects. The rule can check, for example, if a policy holder is above the age of 18 or if there are any unresolved Claims against this Policy. In both cases, this data is not inherent to the Policy, but requires that you invoke other use cases to retrieve the missing domain objects. From a business perspective, this modeling approach is not a problem and allows for a relatively simple definition of the necessary constraints.

Object-independent rules, however, are rules that cannot be tied to a specific domain object. Before we can clarify what we mean, we must distinguish between a regular conditional flow and an object-independent rule. The differences are:

- Configuration
- Autonomous business lifecycle (possibly externally regulated)

Configuration refers to the fact that the rule evaluation does not have to follow a fixed pattern. Instead, only the general logic can be fixed, while the actual decision is based on variable data that you pass into the rule. You can modify the rule evaluation without modifying the built-in logic. Even more so, your business can change the behavior of the IT system without requiring any changes.

Configuration is closely related to the independent business lifecycle. By business lifecycle, we refer to the underlying requirements or specifications for the business rule. You can change rules that define the way you do business. For example, you could change the maximum application age for a life insurance policy from 60 to 65 for marketing reasons. Although the company's internal business can be the driver behind the modification in many cases, external regulation might be the main factor behind a change.

Figure 7-8 describes a simple rule that verifies the age of a policy holder.

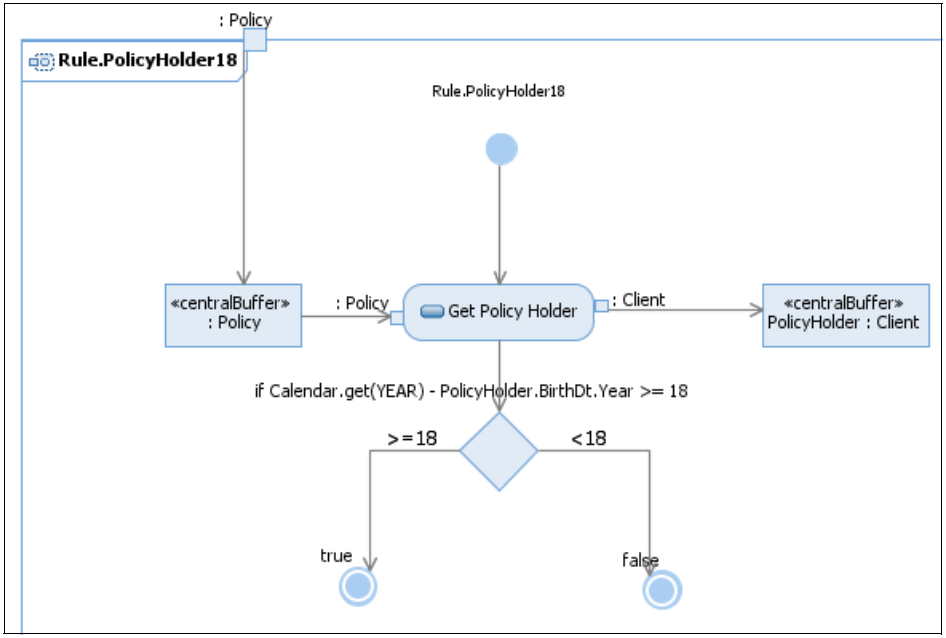

Figure 7-8 Sample rule - policy holder is above the age of 18

Each rule becomes a stand-alone UML activity. Depending on the underlying logic, it can either return true or false. It can also raise an error in case the validation fails. The elements within the rule are no different from the rules in use during use case analysis modeling. You can refer to other reusable activities that help you navigate the object model and look up other objects. You should refrain from starting activities that manipulate data, as these activities change the rules from a validation or consistency check to only manipulating data.

Object-specific rules should be kept with the object they are bound to. The object on the domain model itself is the appropriate container for them, as shown in Figure 7-9.

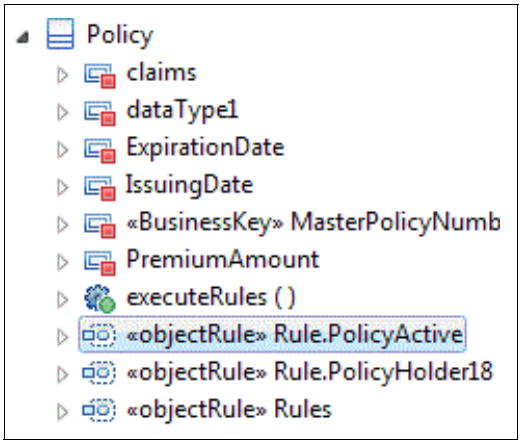

Figure 7-9 A domain model object definition that hosts object-specific rules

Figure 7-9 shows two distinct business rules bound to the Policy domain object. As you can see, we tag all of these rules with the `<<objectRule>>` stereotype, which serves both as a visual and a machine-processable classification. In addition, we apply a naming convention that states that each rule should be called “Rule.”, followed by a brief and meaningful description of the rule. Each object-specific rule has exactly one parameter available, which is the object to which it is bound. You need to add it as in input parameter to the activity that represents the rule. Afterward, the object is accessible and usable within the activity flow.

Apart from the actual rules, there are two additional elements: An activity simply named “Rules”, bearing the same stereotype as the actual business rules, and a method named executeRules. The latter is not immediately required on the analysis level, but helps you understand the link between rules and object. If excuteRules is not introduced, you must add it during the design or implementation phase at the latest.

Whenever a change on the domain object Policy is persisted, the rule engine triggers the executeRules method. This action runs the attached object-specific rules, because the Rules activity is bound to this operation. You can think of this generic Rules activity as a controller that coordinates all the individual object rules. It executes all rules in a specific order and can collect the returned errors afterward. In rare cases, this controller can even add additional logic, for example, the business can define overruling definitions. For example, a rule can fail if another one successfully passes. This situation represents an example where the controller could help you pre-process the rule results.

The object-independent rules are not part of the domain model. Instead, we have a dedicated container within the use case model, the package Business Rules, as shown in Figure 7-10.

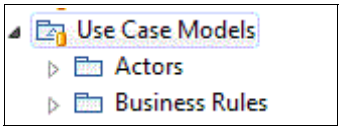

Figure 7-10 Use case package structure that hosts object-independent rules

This container is not intended to be an unstructured container for all rules, but holds adequate substructures as you see fit when designing the rules. For some rules, this container might have the same domains that we already use within the domain or use case models, in other cases, you might have to find different groupings. Each object-independent rule is created as an activity with stereotype `<<flowRule>>`, but is otherwise similar to the previous description of object-specific rules. There is no limitation about the available objects, as with the bound-object in the object-specific rules. Instead, each of these rule activities can define any number of parameters. You should design the rules based on your business requirements and need for reusability.

Unfortunately, there is no mechanism to automatically trigger object-independent rules. Contrary to the executeRules operation, which is implicitly started based on any change to the underlying domain object, for this type of rule, you must explicitly decide when it is appropriate to execute it. It needs to be embedded into the control flow as with any other activity.

7.2.10 Lifecycle

Before you look into the details of a lifecycle, it is important to note that its definition starts during business analysis modeling. Although an eventual implementation can use defaults and add technical information to the lifecycle, the general idea is for the business to provide a definition of different states in which an object might be.

From the modeling point of view, the lifecycle consists of the following components:

- States

  States are static information about the situation an object is in. The current state is typically made available on the underlying object by a dedicated attribute. Objects must always be in a well-defined state. The object must not have undefined states as defined by the lifecycle.

- Transitions

  Transitions link the otherwise independent states to each other. Each transition implies directionality, meaning that an object that is currently in state A can move to state B, but not necessarily the other way around. When you define the lifecycle, you must consider all the necessary transitions. They form the basis for working with the lifecycle at run time.

- Signals

  Signals are what links the decoupled lifecycles and the actual control flow within use cases or services. They offer a single point in the model where you can define a scenario. From the flow, you can then indicate that this scenario occurred, while from within the lifecycle, you can define adequate reactions to such situations.

Figure 7-11 represents an example lifecycle for the claim object in our case study. It is by no means complete, but provides everything you need in a lifecycle at least once.

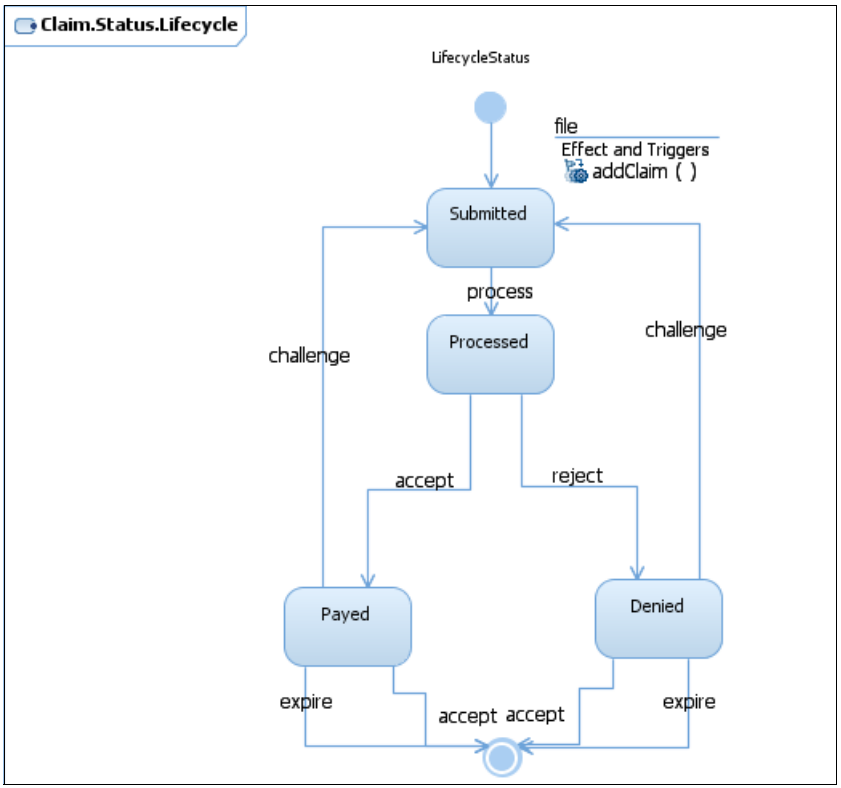

Figure 7-11 Sample lifecycle - claim status

The initial state is “Submitted”, and then moves on to “Processed”. From there, it can take alternative paths to “Payed” or “Rejected” based on the outcome of the processing of that claim. The client can challenge that result or accept it. The trigger that leads to the initial “Submitted” state is an addClaim operation. Each transition has a trigger that we did not include, except for the initial transition trigger.

Within the entire business logic, the lifecycle plays an important role. For example, you can react to information in the lifecycle by defining rules that fire only when an object is in a certain state. Alternatively, you can take on a more passive role and rely more on triggers. When the state of the claim object changes from “Processed” to “Paid,” you want to trigger, for example, a payBeneficiary service.

You could also define business logic that looks at every domain object of a specific type and launches an action based on its state. One example could be to archive all Claims with the “Denied’ status. The lifecycle can also provide hints about the related objects. A Claim that is in the ‘Submitted’ state cannot have payment details. After it is accepted and transitions into the “Payed” state, you should see such information. This scenario shows the importance of defining the lifecycle on the business level, because business rules and flows depend on the lifecycle.

The processes, rules, and the triggers are all centered on the object that carries the lifecycle, as opposed to a specific flow. Although you could design use cases that explicitly launch the necessary action, such an approach is error-prone and difficult to maintain. Instead, you want to decouple the lifecycle management from the flow logic. The use cases describe the flow in the business scenario, while the lifecycle is independent from flows. The link between the two is that a flow can cause modifications that trigger a state transition. You achieve this design goal by using events, which you raise in the flow, and have the lifecycle processing decide whether this leads to a transition.

You define a lifecycle using so called *state machines*. The modeling steps are:

1. State Machine
2. States
3. Signals
4. Transitions
5. Attribute
6. Iterations to refine the model

We start of by creating the lifecycle model and a State Machine. To avoid confusion between different elements, use a name such as “BusinessObject.Attribute.Lifecycle”. In our example, we use the name “Claim.Status.Lifecycle”. Using this convention, you can easily and visually associate the state machine with the attribute it describes, because state machines can be maintained only on classes, but not on individual attributes. Inside the state machine, we then proceed to define the individual states. The names should be meaningful and reflect the underlying business state. A state name should always be an adjective, not a verb, for example “Submitted”. Because the statuses are business-motivated, you can provide a precise definition as documentation for the state element.

Before you can introduce connections between the individual states, you must make sure that you have elements available that help you identify under which circumstances such a transition occurs. In our example, we use UML signals. You need to consider all possible transitions and the conditions under which the transitions are followed before you proceed with the modeling. Create appropriate signal events in the respective package, and ensure that you choose meaningful names, as you are going to reuse the signals you create here in use cases and dependent service designs.

Next, establish the actual transitions. For a well-defined state machine, you need the Initial and Final states. Create all the possible transitions between the states. Use verbs for the transition names. These names describe what happens during this transition. You also must specify when such a transition happens. In our example, we accomplish this task by using triggers. There can be many triggers to a transition. We modeled one such example, where either the client can accept payment or enough time elapses to trigger acceptance. The condition for the trigger is one of the defined signals. It is perfectly fine to use the same signal for multiple transitions, as only the combination of the current state and received signal must be unique, but not the signal itself. When you move from the modeling activities to creating the use case flows, you use the same signals to explicitly trigger the lifecycle.

The final step to introduce the lifecycle is to create the attribute on the domain object that holds the information. Create the attribute as you would any attribute, but for its type, simply select the previously created state machine. The object must always be in a defined state, so ensure that the attribute is mandatory, but offer a default state in the state machine.

The resulting lifecycle-related content, which matches the lifecycle, is shown in Figure 7-12.

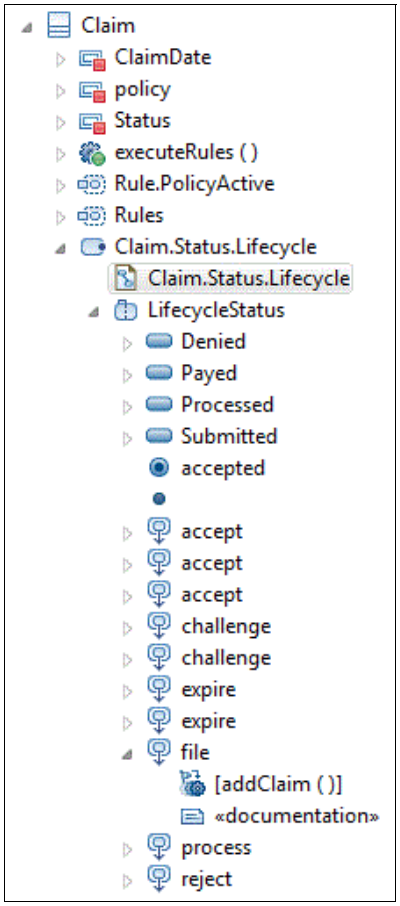

Figure 7-12 Domain model object definition that hosts status with an explicit lifecycle

We must consider a few more complex issues when we design the overall logic:

- Implicit or disallowed transitions
- Parameters
- Nesting

In our example, the state machine holds transitions for all those state-to-state modifications that we allow from a business perspective. But what if an event is received that is not associated with any valid transition? You can either interpret this event as an error or ignore it. It is a preferred practice to explicitly model all valid transitions, which includes all transitions that cannot be ignored when they are triggered. Eventually, our lifecycle model shows all the wanted transitions and the ones where we need to trigger special actions, even if they do not require a state change.

You need to provide parameters to the state machine. To accomplish this task, you must first modify the relationship of the lifecycle attached to an object. In addition, you sometimes require context information during processing of the lifecycle. This information can be a secondary object that is the root cause for the trigger, which provides additional information during processing. Because each signal can have its own attributes, you can define the required attributes and fill them from the flow as required. The disadvantage to this method is that you do not achieve a full decoupling between the flow and lifecycle.

Triggering a transition can cause a ripple effect, that is, this transition triggers other use cases. Depending on the domain model and the business scenario you address, you might need to automatically trigger another object's lifecycle transition. Assume that you do have a lifecycle defined on Claims *and* Clients. The Client status could be defined as "HighValue" or "LowValue". You could classify Clients as HighValue if they put in a great deal of money into your business, while LowValue Clients put in only a little money. Perhaps the transition from high to low is defined as having two paid claims. The transition from "Processed" to "Paid" on the Claim status could make this Claim the second paid Claim. This transition triggers the status transition on the Client as well.

You must avoid deadlocks, which is why you need to validate the lifecycles to check that no two mutually exclusive lifecycles are defined, where one requires the other one to switch states first. Similarly, when triggering the lifecycle from the flow, you must make sure that you send the correct signal, or possibly even more than one signal.

7.2.11 Business process model

You rely on the business process model to describe what the core processes are and the individual steps and outcomes from these processes. From the modeling point of view, you break down a business process into a series of use cases and the logical flow between them. You can use this breakdown to determine the impact of process changes on use cases and whether the object handling specified for the individual use cases works in a larger context.

There are various modeling tools available for business process modeling. The obvious choice is a Business Process Model Notation (BPMN) based model type. Although the BPMN is sufficient for stand-alone models, it is either isolated from all other models or requires the export/import of the domain model, which breaks traceability and cannot guarantee consistency. Instead, use activity diagrams for the actual business process model. Using these activity diagrams, you can establish a relationship with the other models in play and promote traceability from business process model, through a use case model, service model, and the actual implementation.

7.2.12 Guidelines

Because you are still at a high level, the business process activity diagrams do not show many details. The only modeling elements in use are:

- Initial/Final Nodes
- Decision Nodes, where the condition is an informal, text-only description
- Create Object Actions, which refer to the actual use cases from the use case model

You need only these elements, but there is nothing that should stop you from using additional elements, for example, by introducing swimlanes to provide more structure for your diagrams. Depending on the actual business scenario, other elements, forks, and joins can be of interest as well. They can be added when needed. The resulting sample business process with a rather simple flow can look as shown in Figure 7-13.

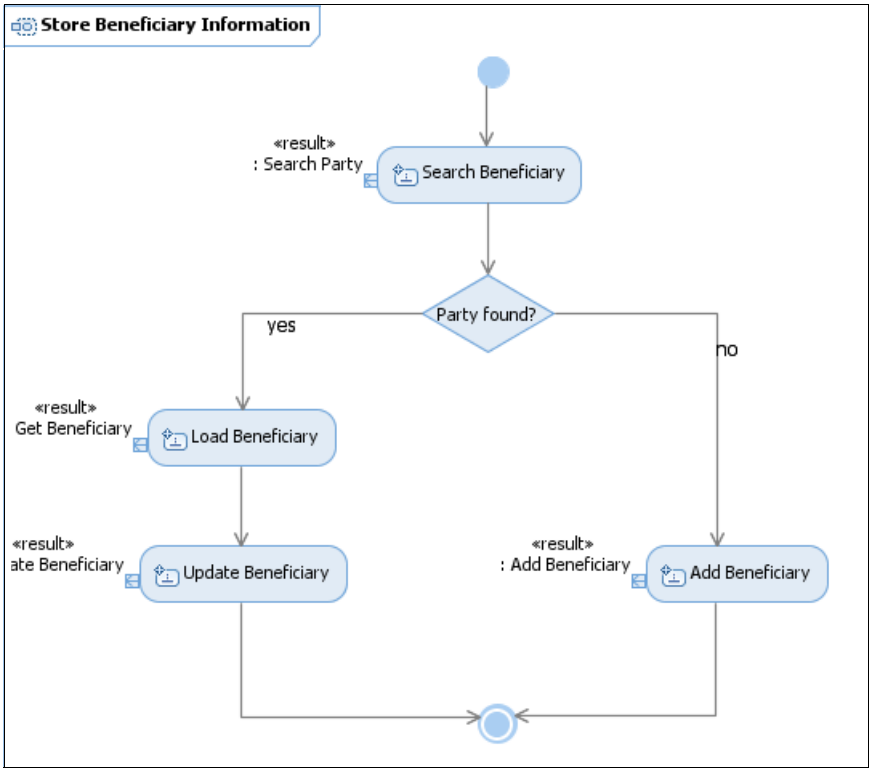

Figure 7-13 Sample business process - store beneficiary information

Starting from the initial node, perform a search for an existing Beneficiary. Depending on the outcome, either add a Beneficiary or load the existing Beneficiary and update it afterward. In our example, we establish a traceable link to the next modeling level, that is, the use case model. We establish this link by using the actual use cases as part of the business process model. We add the use cases to this diagram, which results in Create Object Actions that reference the use case. This design element improves readability. The formal output pin allows us to see which use case we are referencing. The action label adds the description you need to understand the use cases' role in the current process. This description is necessary because use cases can be more generic than what is currently required in the process.

Although it is true that the typical project approach develops the business process model before modeling individual use cases, this situation does not contradict our example. You can add placeholders at any time you find yourself without existing modeling elements. You can use a simple activity that describes the missing element and then drag it into the diagram as you would for use case-related activities. Modeling is an iterative approach and the model is refined with each iteration. Ensure that the model is revisited after the use cases become available.

You often must work with multiple business processes. As with any model, there is an overview diagram that serves as the entry point into the underlying processes. If you have only a few processes, you can link them directly from this diagram. If you have several processes, link them to different structured entries. For our example, we chose two different groupings that target different audiences:

- Business Processes by Domain

  Grouping the business processes into individual domains provides a more fine-grained view of the logical structure of the overall system. All subsequent models, especially the Use Case and Service Model, use the domain structure to define strongly cohesive system components that make up the building blocks for the application. The domain grouping is a typical point of entry for project planning and impact analysis. In the former case, it is helpful, as it allows the definition of functional groups. Similarly, it allows quick determination of how many business processes are likely to be affected if a certain domain changes. However, for people that are unfamiliar with the overall system and the respective domains, it is less helpful as an entry point.

- Business Processes by Business Unit

  Enterprise applications and master data management applications in particular involve several business units. Designers, developers, and project management can typically think in domains. They must have discussions with the business units to finalize the requirements. Therefore, it is important that the business units find an appropriate entry point as well, for example, to accommodate review processes and entry points into the respective use cases. Each business unit is represented as a package.

These groupings provide only an entry point, which leaves the actual processes to be structured. To prevent a model overload, create a package structure that follows the domain grouping and places the individual processes in the respective package, as shown in Figure 7-14.

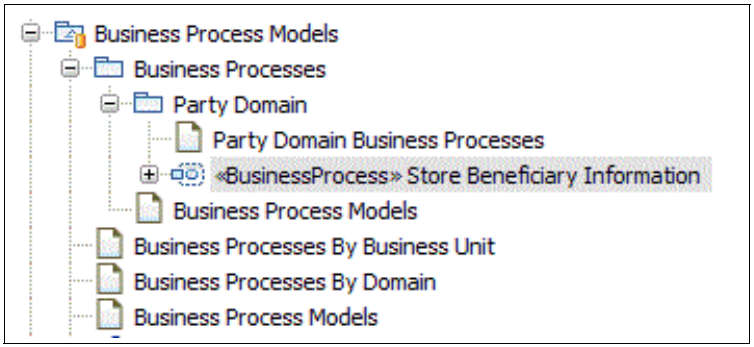

Figure 7-14 Business process package structure

Each activity should carry a unique name so you can identify that activity among the list of processes. Do not underestimate the number of levels and number of model elements in play. Depending on your model's complexity, you might wonder what f element you are currently handling. A common name such as "Add Party" can appear on several levels, that is, as a business process, a use case, or a service (and even then on different levels). As a preferred practice, use the following standardization for business processes:

- Create the activity diagram using the structure described in Figure 7-14.
- Specify the same name for the activity and the contained diagram.
- Make the diagram the activity's default diagram to allow for easier navigation.
- Apply the stereotype <<businessProcess>> to the activity.

7.2.13 Use case model

The use case model serves as a link between the high-level business process model and the fine-grained service design. The use case level provides a higher level of business detail and is the main component of the design and implementation of business services. In accordance with our goal of traceability from business requirements to implementation, the use case closes the gap between individual steps in the business process and what is likely to become coarse-grained business task services. This bridge is composed of the following elements:

- A formal description of reusable elements of business logic
- Traceable links from domain objects in flows to their usage in services

Selecting the correct granularity for the use cases is a complex task, and unless you are highly experienced, it is unlikely that you will find the optimum level in your first attempt. While you are developing the use case model, it is unlikely that you have a clear vision of system components and services. The bottom-up approach to designing use cases is one option you can choose from, but you are likely to end up with a broad range of fine-grained use cases. Some of these use cases might not even be required (based on your business processes) or are even redundant.

A better approach is to start by breaking down business processes into repeatable units of work with clearly defined pre- and post-conditions. After you start to see a structure and have a couple of use cases defined, you can look into common elements, which are typical candidates for reuse. Reuse can originate in a number of structures, either “composite” use cases, which are use cases that consist of other use cases, or use cases that share some logical flows. For example, both Add Client and Add Beneficiary can include persistent address information. You could turn the persistence of address information into a use case. Although you should aim for reuse, it is not a requirement, so revisit your use cases from time to time and check whether you can identify common elements. If so, consider revising the structure. If a use case has generic functionality, such as searching for any type of party, do not name it “Search Person” because you first came across it in a person-related business process; call it “Search Party”, which promote reuse.

One of the common mistakes of designing use cases and dealing with the business flows inside them is related to business rules and lifecycles. Use cases should describe business flows and object flows. Although the flows must obey business rules, they must not duplicate them. If there is a business rule that states that a minor cannot own an insurance policy, do not check for this element in the use case flow. Similarly, if you find yourself describing a use case flow that consists of a check and throwing an error, always ask yourself whether this flow is a candidate for a business rule.

When working with business rules, you should consider the enforcement of data integrity constraints. The same is true in respect to object lifecycles. Do not check whether a transition from “filed” to “processed” is legal from within the use case flow. The whole idea is a separation of concerns, not having to watch for relevant transitions in any use case in the vicinity of a certain object. The only thing the use case should describe is raising an event that is relevant to the lifecycle; then the event can be processed by the state engine.

During use case modeling, always keep the link to the business processes in mind. You are working top down, so the business processes give a general guideline of what you need to be able to achieve with your use cases. Be sure to check logical flows, pre-, and post-conditions. Even more important are the incoming and outgoing objects and whether they make sense in the context of the containing business processes. You need to resolve any mismatch, if your use case states that an object must be read before it can be updated and your process does not contain this step. The use case must always be able to work in the context of the process model. Similarly, the use cases must provide the required functional elements to the business process model.

7.2.14 Guidelines

Now it is time to turn your attention to the actual modeling techniques. Thinking about use cases, you mostly think of those simple diagrams that contain only two basic elements: the actors and the use case bubbles. This type of model is sufficient to provide an overview of which actor is using which system function. It also describes the system functions that exist at all. However, it ignores some relevant details, which are typically covered in accompanying documents. These documents provide an additional description about the use case. The issue that you have with the missing details is that it is a break in the tool and model chain that we can trace across. Also, many use cases ignore the centric domain model. The manipulation of the objects described in the domain model is the sole purpose of most use cases, which is why you need to establish that relationship to our central domain model.

To address this situation, you can use two different stages of use case modeling:

- Use Case Description
- Use Case Analysis Model

The older use cases describe the business flow in plain text and in a standard document. They might already outline the system actor relationship, which is related to the modeling approach. The use cases also can list the domain objects that are manipulated through each use case, which is also related to the modeling approach depends on the availability of the domain model. The expertise and insight provided is vital for implementation, so base the subsequent models that you build on those use cases.

The downsides of these documents are that they do not support structured processing and future usage in the form of automatic generation and transformation towards the service design to which the use cases contribute. These documents also do not support impact analysis or tracing of the requirements down into implementation models. Although informal language always leaves room for misinterpretation, people with little technical and modeling skills can use the language to define their requirements.

The use case analysis model requires the existence of the domain model. Do not start detailed use case modeling or even service modeling before the domain model exists in a reasonably stable state. The less stable the domain model is, the more the rework is required. The analysis models should be created by business architects that are skilled in UML-based modeling. Using the original use case model with textual descriptions together with the objects from the created domain model, you can create structured flow diagrams. In our use case, these flows are represented by activity diagrams. They form the link in all directions:

- Business Process Model consumes and orchestrates the use case specific activities.
- Domain Model objects are manipulated through the use case activities.
- Service Model use cases provide insight into and represent the backbone for the service design.

We restrict this modeling phase to only a few required elements and leave out many details. The design can be transformed and enriched with details so our business analysts can concentrate on the business aspects instead of too many modeling-related features. We use, for example, simplified standards such as the “Object.Attribute” notation to access attribute values, or plain text descriptions to perform transformations and checks instead of more accurate model representations. You should include business error conditions and their formal descriptions, which includes:

- Error description
- Necessary trace information

Many applications lack adequate error messages, and most users receive complex technical information that does little to explain or resolve the problem. Providing adequate descriptions that can be understood by the users and provides guidance in resolving errors is vital for user acceptance. Its definition is a task that can be performed only as part of the business analysis modeling.

Apart from the textual descriptions, some scenarios also call for enterprise or application-wide error pooling, in which case the error design should also include the error identifier. This aspect of error analysis is often underestimated in the error handling. It seems obvious at the time of design what caused the problem. It is not until the actual testing that users begin to notice that what appears to be the error can be part of complex processes where it is not immediately clear where the error originates. You should include contextual information in the errors. A good example is the business keys for entities.

Given the choice of UML activity diagrams, how do you provide details about all aspects of the high-level use case model? This question is important, especially when it comes to the actors. As the service modeling guidelines show, from an SOA perspective, the actor using a certain service must not influence its behavior. Therefore, the analysis models are the last level on which actors can be found. However, they play only a minor role in the analysis model, possibly providing a hint whether a service needs a generic interface for many actors or if it can have a specific interface for only one or few actors.

Using these well-defined modeling techniques alongside the domain objects, you now have a common language to specify the business flows. Because of the simplicity of the activity diagrams, business analysts are able to understand the formal descriptions more easily. They can also verify that all modeled details are understood correctly.

7.2.15 Classification

The basic use case model in UML is rather simple and straightforward. Figure 7-15 shows one such example.

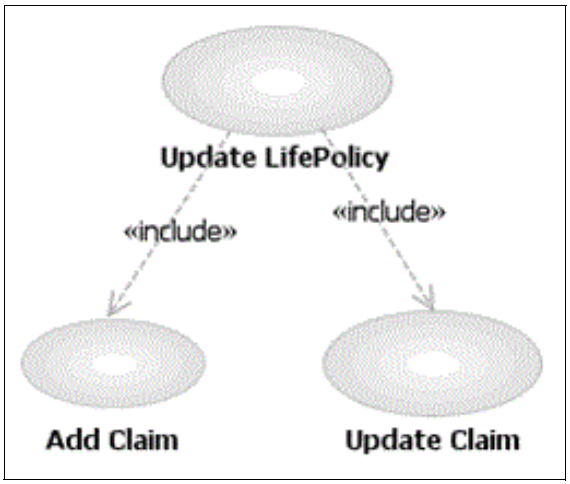

Figure 7-15 Sample use case - Update LifePolicy

Figure 7-15 shows the use case “Update LifePolicy”, which has an “include” relationship with two other use cases, “Add Claim” and “Update Claim”. Whenever you have inclusive relationships, you have a possible reuse of the service and use case levels. Each use case should carry a name that allows for easy identification. Whether you use a meaningful name or a unique ID depends on your preference, if it is agreed to by all project participants. The unstructured textual information about the control flow and alternative flows can be stored in the documentation property of the respective model element.

Although the individual use cases are rather simple, a complex solution, such as a master data management system, can involve many use cases. Therefore, define a project structure that categorizes the use cases for easy retrieval and reference, as shown in Figure 7-16.

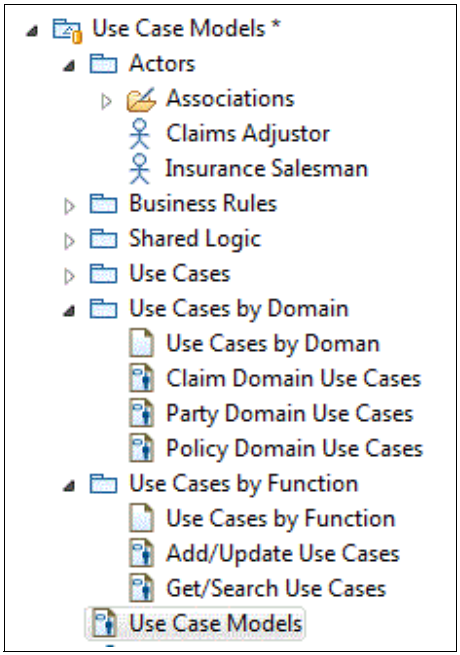

Figure 7-16 Use case package structure

Starting with the Rational Software Architect default project structure for a Use Case Model, introduce similar groupings as you did for the Business Process Model. These logical groupings have no effect on the actual use cases; they introduce structure to the model to make it more manageable. As with all model packages, each package holds a default diagram that links to the other available elements to help navigation. Preferred packages are:

- The Actors default package holds all actors that work with the different use cases.
- The Use Cases default package holds all available use cases.
- The Business Rules additional package lists all rules that are flow-specific instead object specific.
- The Shared Logic additional package stores all activities that are common across many use cases.
- The By Domain additional package returns to the previously introduced domain grouping and holds one Use Case Diagram per domain.

- The By Function additional package groups the use cases by function. You can isolate search, read, or update use cases.
- Other possible groupings include the previously mentioned business units or subsequent master data management system components.

Out of the many choices, activity diagrams are the most adequate approach to modeling use case logic for our situation. Our primary goals are:

- Establish and maintain traceability with the Service Model.

  There could be many services for each use case, and also one service for many use cases, depending on the service design on the subsequent design model level.
- Establish and maintain traceability with the Domain Model,

  You always want to know which domain objects are used in control flows.
- Maintain human readability,
- Avoid unnecessary formalisms that make model overly complex.

Traceability and less formal models are hard to combine, which rules out many choices for the use case modeling. Impact analysis becomes more precise if you can determine which parts of a service are affected by changing a single step in the use case, although this action requires complex mapping. It is unlikely that each control flow step from a use case turns into a single subactivity in the service design. For business analysis purposes, it is sufficient to see which services the use case maps to without exactly knowing what happens to the individual steps.

Each use case flow is attached to the corresponding use case. It carries the use case's name, but to separate it from other model elements of the same name, especially later designed services, add the <UCAnalysis>> stereotype (Figure 7-17).

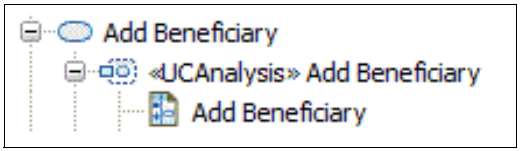

Figure 7-17 Use case definition with activity

For each activity, we define a default diagram, which again bears the same name as the activity and use case. This diagram describes the use case flow logic, as shown in Figure 7-18.

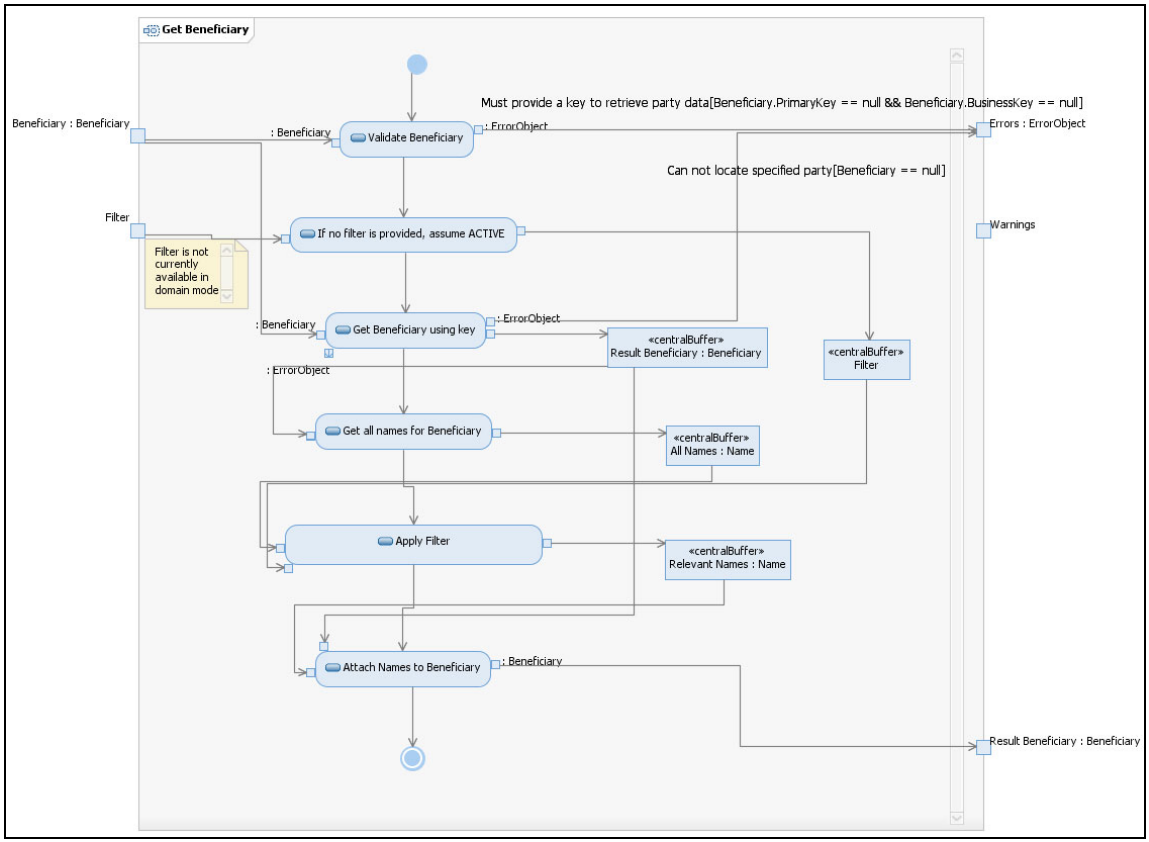

Figure 7-18 Sample use case - add beneficiary

Figure 7-18 on page 329 shows a sample use case analysis. Creating the diagram based on the textual use case description is a rather straightforward process. Each step from the use case becomes an element in the diagram. Be careful not to overload it with lengthy activity descriptions. You might even encounter length restrictions, depending on the system environment and design guidelines that relate to it. Mostly those restrictions apply to design and implementation levels and not the business analysis. Complex activities should still carry a title that summarizes its actions, while the complete description is embedded into the documentation property.

The regular control flow begins in an initial node. A series of action elements follow that you use to describe each individual step. The control flow concludes in a final node. For alternative flows, rely on the condition element. If you repeatedly find yourself creating the same element, such as sorting names alphabetically, or filtering entries according to a certain pattern, consider the introduction of a shared logic activity. Following this approach, you effectively create a number of nested subactivities. These building blocks reduce modeling and maintenance efforts.

Modeling input and output is more complex. It is inappropriate for the use case model to contain a formal service signature. Instead, you should make sure that all relevant objects are included in the model and that the necessary relationships are established. You can accomplish this task by using Activity Parameters and Activity Parameter Nodes. Each parameter has the following set of relevant attributes:

- Name: A unique name for the parameter.
- Direction: Specifies whether this parameter is an **`in`** or **`out`** parameter.
- Type: An object from the domain model.
- Multiplicity: The number of elements to expect.

Whenever you create either an activity parameter or an activity parameter node, the corresponding element is automatically created. Both elements are linked, but maintain redundant information, that is, the name and a type. Unfortunately, the parameter's type does not show in the activity diagram, so that you must duplicate this information in to the parameter node.

Relationships between objects also require our attention. A use case, for example, can process a person and an address. You could assume that the address is in some way related to the person without further information. But what happens if the use case processes a list of persons and a list of addresses? Is there one address per person? Is there a relationship at all? You can argue that if there is a relationship, it should be supported by the domain model; why not attach the address to the person?

Because you are still on the analysis level, you do not want to specify a formal signature or make any indication as to how the service interface must look. To solve this problem, introduce a UML relationship between the addresses and the persons with the name “belongs to”. This way, you have a traceable relationship as part of the model, which you can formalize when moving on to the implementation models.

Additionally, you must deal with generated objects, objects you read from a data source, or simply the input objects that you manipulate and store in an object other than the input object itself. Use a Central Buffer to model objects inside activities. Using naming conventions, you can easily identify the objects in question within the control flow. A name such as “all addresses” symbolizes a collection of all addresses. Also, introduce a `<<list>>` stereotype to indicate that you are dealing with a collection of elements of the specified type. You notice that introducing central buffers quickly clutters the diagrams. Use these elements only when objects appear for the first time or existing input data is manipulated; for all other scenarios, especially read-only checks, or manipulating data in a central buffer, simply reuse the existing element instead of creating one.

Now that the regular control flow is available, you must turn your attention to elements that can lead to a premature end of the flow, or trigger activities not directly linked with the use case that is available:

- Errors and warnings
- Triggering of lifecycles

Errors and warnings

Errors or warnings can appear at any point in the flow. Depending on how you transform the textual use case description into the activity diagram, they can appear as the result of a condition element or simply as the output from performing a certain action. The domain model hosts all errors for your system in the BaseConcepts package. Now you use these ErrorObjects. Before you decide how to proceed with the errors and corresponding error objects, consider the error handling strategy. Errors can be collected, for example, if multiple errors arise within the same use case. You return only one or many errors under these circumstances. In a fail-fast approach, you return only a single error and terminate the use case execution with the first problem you encounter. Your modeling technique considers both alternatives.

After you identify an error situation, the first step is to create an adequate ErrorObject. Do so by adding a Create Object Action to the activity and selecting the correct class from the domain model. You now have the empty error. Add a Call Operation Action and use the ErrorObject's setter methods to add the appropriate parameters if additional details need to be provided. This action can result in complex branches within the design, as some lookup operations must gather the required details.

With the ErrorObject fully populated, its further behavior depends on the error handling strategy. The preferred practice is to pick one approach consistently throughout the entire application. You might decide to distinguish between technical and business, or rule/validation and regular flow errors. If you chose the fail-fast approach, you can add a Send Signal Action and connect the previously created ErrorObject to the element. The idea behind this modeling approach is to provide an immediate exit to the control flow. No further action is necessary and there is no further control flow executed within the application unless the error is explicitly handled.

If you want to collect all the errors that occur within flow, choose a different approach. One solution that comes to mind is to collect the errors that occur and then use a signal to throw the entire collection. This approach is a feasible one, and could be used in a simple scenario, such as a single rule that needs to pass out multiple errors. However, in a more complex scenario, you can no longer assume that all errors occur from within the same flow. For example, if you try to add a policy and want to return all rule violations at once, you might want to wait for multiple rules to complete execution. You must manually handle the occurring signals whenever you collect many possible errors and if you resolve the same signal approach you use for fail-fast.

Instead of cluttering the design with such receivers, use a different model. Each activity within your flow is equipped with an additional out-parameter/parameter node named “Errors”. Specify the type as ErrorObject. Set the “IsException” property to true, indicating that this parameter is used to describe exceptional situations. In addition, increment the upper bound to unlimited so we can return multiple errors. Within the use case flow, create a data flow from each error object to the Errors parameter node. Connect the control flow to the next processable statement after the error situation. The semantic behind this approach is to return all collected error objects together with the activity's regular return value. Collect the errors delivered after invocation and, if required, add them to this activity's Errors parameter. This way, you can pass the errors up the invocation chain, and collect and return them as appropriate.

Handling warnings differs slightly from the previously described error handling. The basic details are the same, that is, create and populate the WarningObject. However, with warnings, you typically always collect the information you encounter throughout the entire control flow and bundle it into the response. Using the Send Signal Actions, and thus disrupting the current flow, is therefore inadequate. Instead, provide for an output node and corresponding parameter named “Warnings” that is typed to multiple WarningObjects. Within the flow, connect each of the populated warnings that should be returned to that warning parameter. As with the errors, you must decide on the next processable activity within the flow and continue with regular control flow. At the end of execution, the Warnings parameter is returned like any other return value from the execution. Thus, you can return the required information all the way to the top and perform whatever actions are necessary.

Lifecycle

The final topic to address is how to trigger lifecycles. You can use the signal mechanism to temporarily suspend the control flow. These signals might be different from the signals you used in the past. It is our explicit intention not to create an asynchronous processing for the lifecycle event, as the outcome might be relevant to downstream actions. The only intention is to decouple control flow and lifecycle. The unique signal is raised, processing is turned over to the state engine, the lifecycle is processed, and the control flow resumes at the point of the raised signal.

Figure 7-19 shows an example signal.

Figure 7-19 Sample lifecycle signal

Figure 7-19 shows the elements involved in triggering a lifecycle transition. You can see a send signal event and the signal “Party Created”. Remember that the decision whether a lifecycle needs to be triggered is based on explicit knowledge about the required lifecycle transition. The signals you can use are in the appropriate package on the domain model. Among the defined signals, you should find the signal that appropriately reflects the situation. Now, add a Send Signal Action, click **Select existing element**, and navigate to the signal within the domain model to pick the appropriate type.

7.3 Design models

The previous section focused on capturing the business requirements through analysis modeling. You created the business domain model and business processes and use cases. Dynamic aspects, such as lifecycles and rules, were also described. The information you collected in those models is dedicated to the master data management system you build, even though you could also create business processes and use cases for the other systems you must integrate. These systems and their models have not played an important role up to this point. As such, we have not spent much time looking at the surrounding systems, especially those systems that are the source systems for your data. The business analysis models are still business-related. Now, you need a mediator between business and technology, which are the design models.

There are three different influential streams at this level:

- Analysis models
- Source system models
- InfoSphere MDM Server models

Your main task within the design phase is to bring all of these aspects together. You add logical, platform-independent information to your system design. This action has different tasks. For the source system models, your main task is to establish the data mappings to the new master data management world. For the analysis models, you need further refinements. Although you have a certain degree of formalism into the models, this formalism is primarily a form of documentation and is not yet a specification for an implementation that you require. The InfoSphere MDM Server models must be tailored to the solution you have in mind. Even though you use a product whose features influence your business analysis modeling to some degree, you still need customization. Starting with the available object and service models, you need to attempt mapping or decomposition and apply customization where necessary. This chapter does not explicitly deal with mappings, but provides the underlying models that are required for them.

Because you adopt a product that comes with its own models, the models are split into two distinct categories: InfoSphere MDM Server related models and custom design models. For the sake of categorization, consider the customization of the InfoSphere MDM Server models part of the InfoSphere MDM Server models. Irrespective of the product, you have the following distinct model types:

- Logical data models
- Domain and object models
- Service models

Figure 7-20 explains the relationship between these individual models.

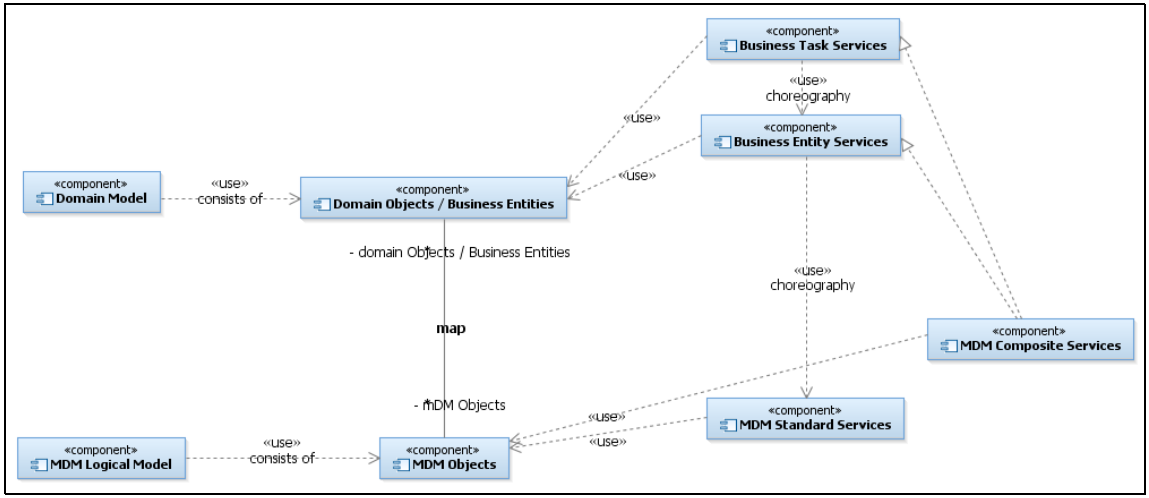

Figure 7-20 The domain model and the composition of custom domain services

The domain model is based on its domain objects. These objects, with their corresponding attributes, are mapped from and to the master data management objects. This mapping requires transformation, but more importantly, it represents an abstraction layer. Instead of exposing a master data management-specific language to the layers that communicate with InfoSphere MDM Server, we are hiding this logic and instead expose the custom domain model. This action leads to the interactions that are maintained with other systems for which you rely on for services. InfoSphere MDM Server provides its own set of services, which are the data services that also are required when you model extensions and additions. These services can be fine-grained, where you manipulate only a single DataObject, or coarse-grained, where you can modify multiple related objects at the same time.

In addition, you can use InfoSphere MDM Server to create *composite services*. These services build the foundation for the business entity services, which are services that are used against the individual domain objects. Business task services are also built on the grounds of InfoSphere MDM Server composite service logic. As opposed to the business task services, the business entity services manipulate domain objects in the context of a use case and represent the ultimately exposed business services of your master data management system.

Figure 7-21 outlines the main design models and the artifacts that they represent, independent from a product. These models are the ones represented on the top layer of the previous domain layering diagram, as shown in Figure 7-20 on page 335.

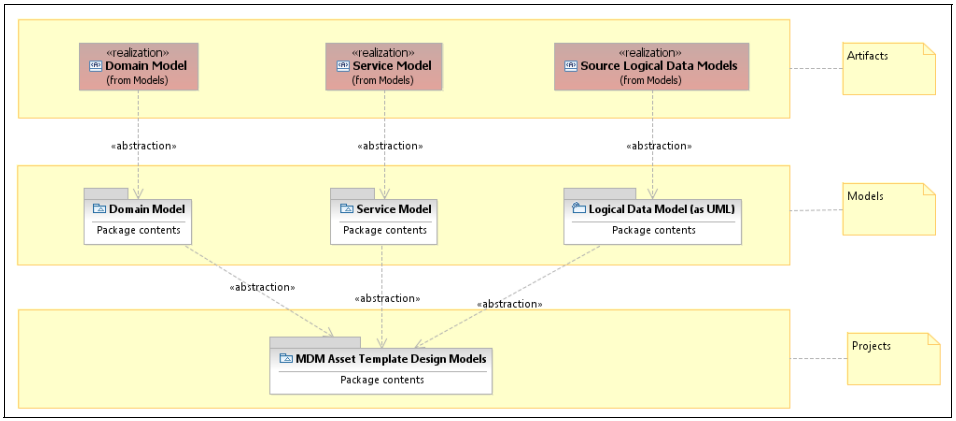

Figure 7-21 Artifacts, models, and projects in design modeling

Even though they appear only as a single realization instance, the logical data models show us the logical data structure of each of the involved systems. There also is a logical data model that describes the InfoSphere MDM Server. Rely on them when you require transformations between physical data models and object models. The physical data models are part of the implementation models and are described there.

Similar to the logical data models, you need one domain model for each system. To be more precise, you need one representative object model for each system. This requirement is based on the requirement for dynamic mapping that you established on the UML object model level (for more information about mappings, see Chapter 8, “Mappings and transformations” on page 411). In respect to the business domain model, you need only one. Use the one you established during business analysis modeling as a base and transform it into the design domain model equivalent. Enrich it with further details or resolve business modeling helpers, such as
multiple inheritances.

The service model describes the business task and business entity services that are not provided by the product. Effectively, you must establish a service mapping, but this mapping also includes additional business logic. It represents the fine design for your business logic implementation. These services are highly influenced by the use cases defined in the business analysis models.

Figure 7-22 outlines the main InfoSphere MDM Server specific design models and the artifacts that they represent. These models might still undergo customization, but they primarily represent static models that you have available when you start using MDM. This customization adds or removes some objects and entities, but does not change the models entirely.

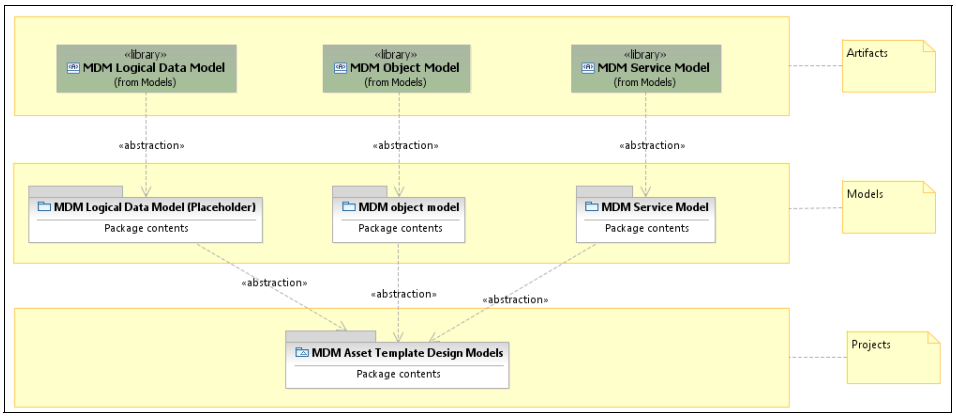

Figure 7-22 Artifacts, models, and projects for InfoSphere MDM Server resources in design modeling

The InfoSphere MDM Server object model serves multiple purposes. It represents a dynamic mapping target, but also provides you with the objects that you reference in your service design. Starting with the object model that ships, you must fit all the pieces from your model into the product, and whenever you are not able to do so, you must introduce customizations.

Complementary to the service model, you have the InfoSphere MDM Server service model. It gives you access to all the services the product provides. As with the corresponding object model, you must make sure that all of your business flows can be mapped to available services within the master data management solution, either as shipped or by using other customized services.

The master data management logical data model is similar to other logical data models. We do not largely depend on it for most of our tasks. If you use it at all, use it as a transient step during transformations between physical and domain or object models. The transformations are rather simple, because the logical data model is almost identical to the physical data model that InfoSphere MDM Server provides.

The InfoSphere MDM Server models require customization in many cases. Figure 7-23 shows the custom master data management models and their relationship to the design project.

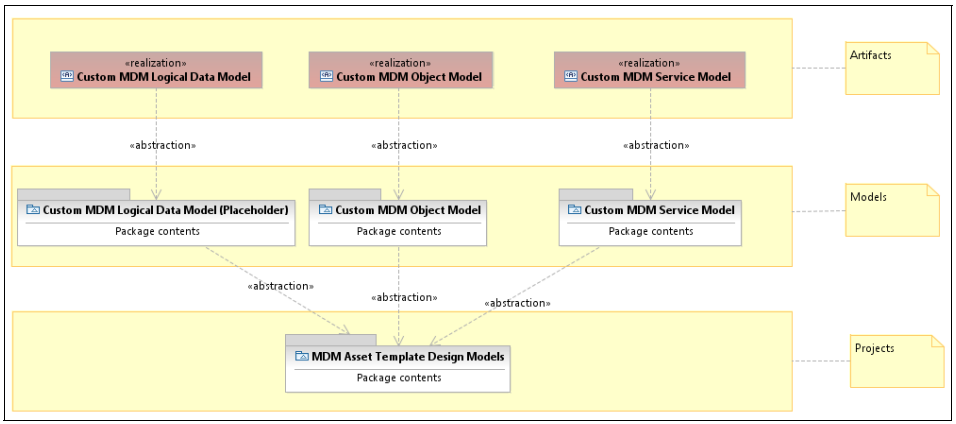

Figure 7-23 Artifacts, models, and projects for custom InfoSphere MDM Server resources in design modeling

Our project structure reflects these models. The design models project contains independent models for each of the previously mentioned aspects. We explain each model type in the following sections. They are:

- Custom models
  - Domain model
  - Service model
  - Logical data models
- InfoSphere MDM Server models
  - InfoSphere MDM Server logical data model
  - InfoSphere MDM Server object model
  - InfoSphere MDM Server service model
- Custom InfoSphere MDM Server models
  - Logical data model
  - Object model
  - Service model

7.3.1 Custom models

The *custom models* are independent from any underlying product. They describe the functionality, as required by the business, but on a level that is already technical. Custom models, however, are not exclusively limited to the master data management system you are going to build, but also include the models from the existing source, and possibly target systems too.

The models we describe here are:

- Domain model
- Service models
- Logical data models

Domain model

The business domain model originates during the business analysis phase. This model mainly reflects the business' view on your newly designed master data management system. With the MDD, you must take the domain model to the next lower level and add content to it that is not entirely business-related, but is still abstract enough that it is platform-independent. Those details would be added only on the implementation models, which also host the domain model, but enriches it even further.

Until now, all your efforts surrounding the domain model are purely business motivated. You captured all aspects strictly on a business level, not worrying about design or implementation-specific aspects. You can use certain design elements that can cause problems in an implementation, such as multiple inheritances. You deliberately used sparse details, for example, regarding the navigation of associations, which is required for interobject traversals. You also explicitly hid certain information behind stereotypes, for example, time-related aspects.

From a business point of view, you have a model that fits the target audience, that is, the business users and business analysts whom you expect to write specifications using this model. However, before you can move to an actual implementation or specification, you must ensure that the model is rich enough. So, during the design phase, you need to change the design.

Guidelines

It is important to note that from now on you explicitly maintain two different versions of the domain model. You do not want to clutter the existing business domain model with specification or possibly implementation relevant details. Thinking of the most extreme scenarios, you can even introduce or modify associations or other relationships, which must not in any way influence the perspective the business has on this model. Having two models has its downsides in terms of effort and inconsistencies. However, this situation holds true until you consider transformations. There is no need to manually re-create the business domain model on the design level. Instead, use a transformation. A transformation has several advantages. It saves you from making additional effort, allows you to trace the existing model elements from the analysis level to the design level and later on onto the implementation level, and you can integrate some automatic changes or refinements into this transformation.

The following sections describe some of the most significant changes between the model on the analysis level and the design level in more detail. They are:

- Time-related aspects
- Inheritance
- Navigation
- Operations

Time-related aspects

In previous chapters, we described the complexity of the time issue in the overall model. The business domain model hides this complexity from view simply by applying a <<timeBased>> stereotype. This abstraction is no longer suitable when you define interactions against the model on the design level. You might need to deal with the sequence of time-based values, not only the current value. Looking at the domain model, you must determine the appropriate method signatures to get and set the underlying values. You can change the model according to the following algorithm:

- Ignore all attributes and associations that are not <<TimeBased>>.
- Skip all attributes and associations that have the upper boundary set to "*".
- Replace all upper boundaries on attributes or association with "*".

After this transformation, you are left with an unchanged model, except for the time-based elements, which are now capable of storing the entire time line instead of one or more time slices. The only change we made here is switching from a static to a more dynamic view of the domain model in respect to the time-related aspects. Figure 7-24 shows such a sample transformation on a simplistic model.

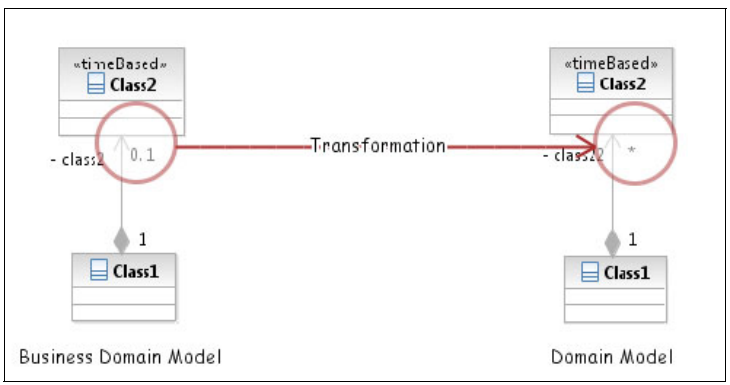

Figure 7-24 Transformation of timeBase objects from analysis to design

Another aspect you must uncover on this level is the actual validity periods. Business logic can depend on the inspection of, for example, the remaining validity periods. In addition to modifying the cardinality, we introduce startDate and endDate attributes, which you can then use in the service flows and rules.

You do not need to deal with more technical or platform-specific information yet; that information is handled by further transformations on the implementation model level. So, even though you change the multiplicity of the associations to *, you do not define how to implement these associations. Again, keep the design as simple as possible and avoid cluttering it with unnecessary details.

Inheritance

In any object-oriented approach, inheritance plays an important role. You must ensure that you understand the motivation behind inheritance, as it is reflected in the respective levels of modeling you defined. Shared structural elements that result in a model with fewer redundancies is one of the benefits. In our insurance example, we must define only the core attributes for a Party once, and they are automatically available on both Client and Beneficiary. You can overcome the apparent structural mismatch between the rich semantic of our domain model compared to the limited capabilities of the target platform by performing careful analysis on the design level.

The first steps in refining your analysis model are to look at the domain entities and see if you need structural consolidation. The resulting definitions of inheritance can differ between the domain model on the design level compared to the domain model on the business analysis level. Traceability information that you apply during transformation keeps these different interpretations lined up and related. Do not underestimate these structural changes, because they make translating the use cases much more difficult. Such change is little more than convenience or optimization, because it simply allows you to avoid redundancy in data structures.

However, there are also inheritance structures that cause problems when you attempt to translate them to implementations, or try mapping them to the InfoSphere MDM Server model. Such structures describe multiple inheritances. In some environments, a business or designers want to design multiple inheritance. We allow only structures that resemble multiple inheritances on the business analysis level. It is a structural element that we strictly dismiss on the implementation level. So, it is you task on the design level to eliminate these elements.

Although that elimination is only necessary on the implementation model level, you need knowledge about the business to resolve these elements. This situation is why you establish that resolution on the design model level. Such a design element is in itself so complex that you can provide an automated form of restructuring as part of the transformations. Instead, as part of refining the domain model in the design stages, you must revisit the classes that underlay multiple inheritances and apply a remodeling strategy on an individual basis. The strategies we identified are:

- Introduce redundant attributes.
- Introduce interfaces.
- Flatten and extract logic.

The introduction of redundant attributes works best if the inheritance is used only as an element of structure, that is, you are trying to avoid redundancies and extract common attributes in to a general type. In this case, you could eliminate the inheritance problem by defining the attributes on each of the concrete subtypes, as shown in Figure 7-25.

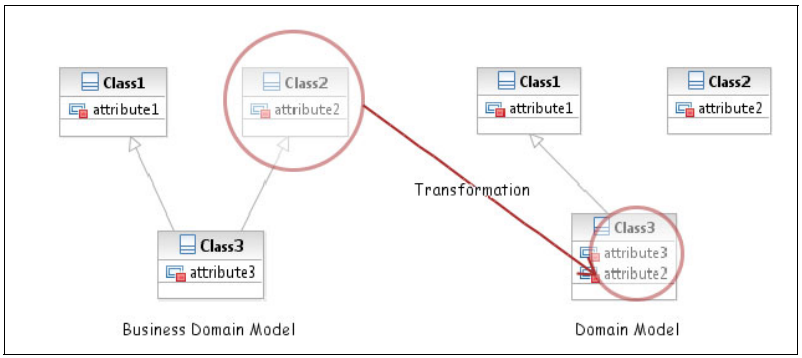

Figure 7-25 Transformation of multiple inheritance from analysis to design redundancies

Interfaces represent another alternative to resolving multiple inheritances, if redundancies are not the reason why multiple inheritances were introduced on the business analysis level. You must find a way to simulate such inheritance from within services and service signatures. In the domain modeling, we looked only to Classes as complex elements. Interfaces are a technical design element that can help you with this dilemma. You can use an interface to define a common set of attributes and a single point of reference that you can use in flows. After you perform this action, unrelated types behave as though they were a common supertype, as shown in Figure 7-26.

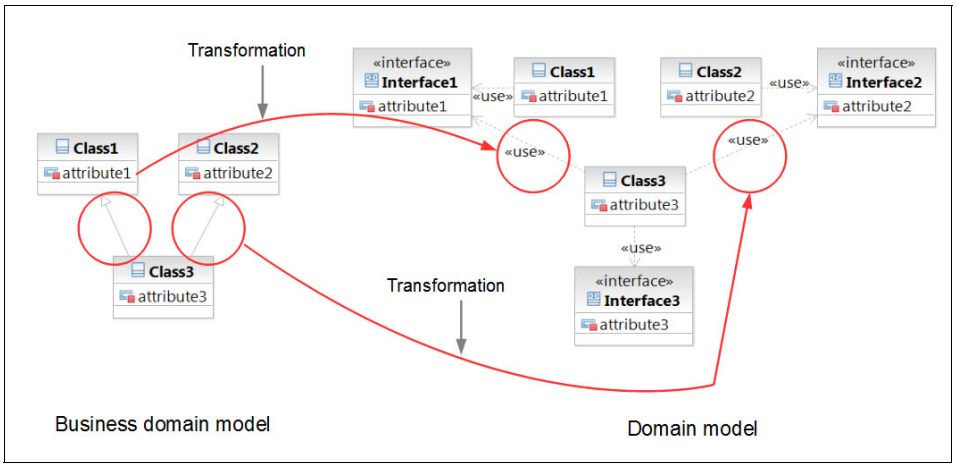

Figure 7-26 Transformation of multiple inheritance from analysis to design interfaces

Flattening or extracting the implicit business logic that is behind multiple inheritances is another option. This approach is probably the most time consuming and difficult one, but is also the most rewarding one. There might be cases were the inheritance structure is simply introduced to create model handles in terms of rules.

Flatten the structure, which is similar to the resolution you apply to redundancies. In addition, explicitly define the rules that are implicitly embedded in the definitions of multiple inheritances, as shown in Figure 7-27.

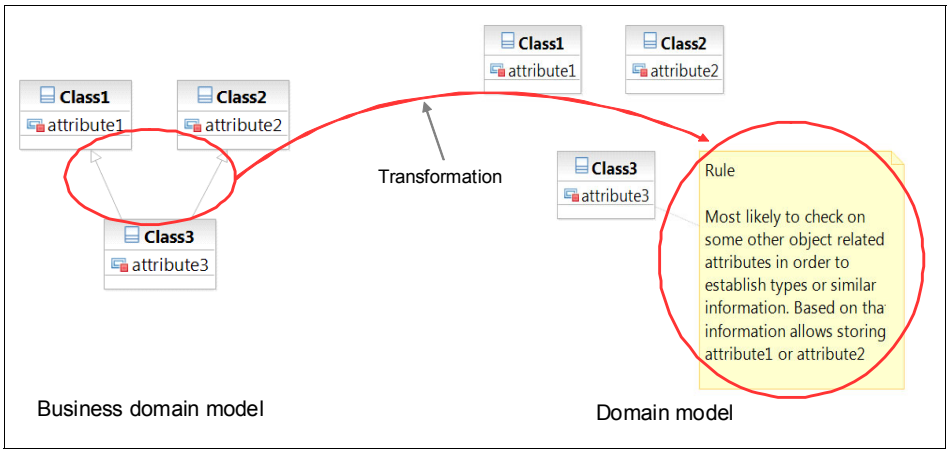

Figure 7-27 Transformation of multiple inheritance from analysis to design rules

Navigation

The term navigation, sometimes also referred to as *direction*, refers to which direction you can traverse along an association. We deliberately decided not to require the inclusion of directionality into the equivalent domain model on the business analysis level in our example. The reason is that the high degree of uncertainty often influences accurate identification of such navigation needs. We also do not deem it to be an element relevant to express business requirements. Unless you are a highly experienced business modeler, you should stay away from this level of detail and instead rely on your designers to query you about it and have them add this information.

When you start with the domain modeling during the design phase, you mostly have little or no navigation criteria specified. What we did include are roles. These roles serve an important purpose when you move towards the implementation, as they become the link names in case you establish references between objects.

Looking at two entities from our insurance sample, we have an association between the *claim* and the *policy*, which use the following general UML alternatives regarding navigation:

- None
- Unidirectional (in either direction)
- Bidirectional

The most likely uses in our domain model are unidirectional or bidirectional associations, meaning that you can traverse from either both or just one direction. Depending on which association you choose, you either have a Claim-to-Policy or a Policy-to-Claim traversal or navigation from both directions. Theoretically, you could see none-navigable associations in the domain model as well, representing something like a virtual link that would never be followed directly. For the remainder of this chapter, we ignore this element entirely, assuming that if you encounter something like this element at all, it has no technical equivalent.

Figure 7-28 shows a split bidirectional navigation that increases semantic correctness on the associations.

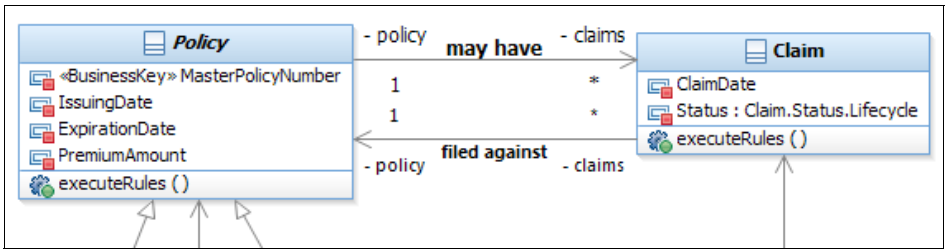

Figure 7-28 Sample for split bidirectional navigation

In this example, you can introduce a general rule into the transformation from business domain model to the domain model on the design level. For each association that has no explicit navigability specified, assume bidirectional navigation. This general rule gives you the highest flexibility and is likely to be the default case. After the transformation, it is up to the designers to revisit these associations and make adjustments as necessary.

Operations

The business domain model is composed of entities, attributes, and relationships, which carry definitions about type, name, or semantic meaning. On the design level, you must describe how you want to retrieve the value of a certain attribute, or specify the value of another attribute. In line with the way we model services, our preferred way to accomplish this task is by using Call Operation Actions.

As one of the domain model refinement steps, you must introduce getters/setters for each attribute. Because this procedure is strictly mechanical, you can have the transformation perform the necessary changes. During transformation, the following rules apply:

- Introduce methods named “set” and “get” followed by the attribute name. In the case of Boolean attributes, use “set” and “is” instead.
- If the maximum cardinality of an attribute is greater than one, the set-method is to add an element to the collection.
- If the maximum cardinality of an attribute is greater than one, the getItems-method is to return the entire collection of elements.

For conceptual integrity, these conventions correspond to the ones used by InfoSphere MDM Server.

7.3.2 Service model

The *service model* is a platform-independent representation of the business logic. We formally define operations on the available business domain objects to implement those business requirements. All these definitions do not yet include aspects of the target platform into the model. The enhanced formalism eliminates ambiguity, thus laying the foundation for the optimization in the implementation models, while relying entirely on the business terminology, thus still enabling discussions and problem analysis on a business level.

Although it is entirely possible that service models are available for some of the source and target systems that surround our master data management solution, these do not play important roles in our overall approach. You can add them to the respective Design Model projects and use them as reference when you try to set details about business logic or how existing systems treat certain concepts. Service models can also be used by ETL developers that might want to use the service definitions to extract data from a source system or load data into a target system using these services. Here we must establish an entirely new set of services for our master data management system.

The service model relates to the following models:

- Use Case Analysis Model
- Domain Model
- Object Model (and its customization)
- Service Model (and its customization)

The *use case analysis model* is the most important input to the service model. As we show throughout the remainder of this section, the services we design here are based on the requirements found in the use cases. We rely on the use case analysis model, because it already contains a certain degree of formalism that you can now refine into machine-processable building blocks.

The *domain model* in its refined form on the design level is the second key factor in service modeling. It provides you with the objects that you handle and manipulate inside the services. The services abstract the objects from the InfoSphere MDM Server object model. Because the service model is in fact a service mapping, you must acknowledge the existence of the individual object models on either level, upper level in the form of the domain model, and lower level in form of the object model. You must reference the InfoSphere MDM Server service model on the same grounds. It provides you with the low-level services that you map against and eventually start to persist your data.

An important goal for the service model as a binding element is to promote traceability in multiple directions. Regarding the usage of domain model *objects* and master data management *services*, traceability is automatically given as soon as you use these elements within the service flows. The situation for the use cases is a little different. The goal behind traceability is to define which service designs realize a particular use case.

Consider a need to evaluate a change request. The client indicates a change to either a business process or a specific use case. In your impact analysis, you must know how many services this change affects. If you can see how deeply the individual services are affected, you would have a better indication of how the change affects the services. However, each of these services can be composed of several fine-grained steps. Simply maintaining all the links between these steps while adding, updating, and deleting them if they are no longer valid consumes a major portion of your available time. We therefore propose to implement traceability but skip detailed impact analysis on this level

Service types

Before we describe the package structure and where to place the different elements within the service model, we must ensure that you have a common understanding of the types of services you are dealing with. We also need to ensure that you not only understand the elements on a logical perspective, but also manage to convey this information in a machine-readable format, as this information must be used in downstream activities.

Figure 7-29 describes the service layers we apply.

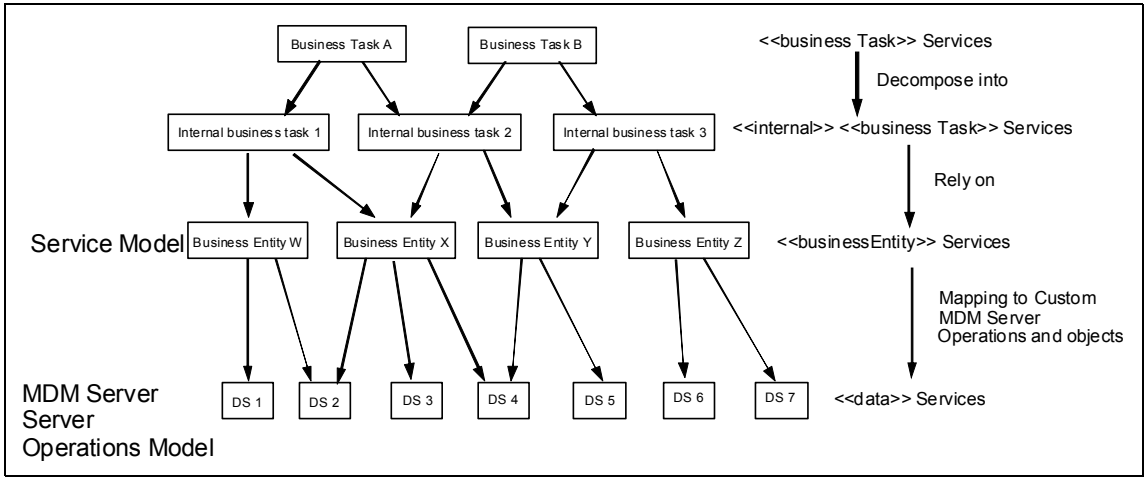

Figure 7-29 Service layers

We describe an approach with four different levels of services. Time permitting, and provided you have experienced InfoSphere MDM Server experts available, you can consider merging the two middle layers into one. Separating the functionality helps in this context to explain the individual functions each layer needs to provide. We work with the following layers:

- Business Task Services
- Internal Business Task Services
- Business Entity Services
- Data Services

Business Task Services are at the top of the hierarchy. These services are available to service consumers outside the master data management system. They make up our external interface. It is best to think of these services as reflecting the top level of Use Cases we identified in the Use Case Model. This type of service deals with complex business flows and handles multiple objects from the domain model. It decomposes the logic into more fine-grained and reusable units.

The internal Business Task Services are still coarse-grained services that capture business logic, but are of no external relevance. Do not allow consumers to start these services. They are building blocks that you can reuse throughout the entire Service Mode in an attempt to reduce maintenance efforts and avoid duplication of logic. Internal Business Task Services typically handle more than one type of domain object.

Business Entity Services are the most fine-grained services you handle on the Service Model level. They form the link to the InfoSphere MDM Server object and InfoSphere MDM Server service models. A business entity service represents basic create, read, update, and delete functionality, but in contrast to a service operation in master data management, where you work with master data management objects, the granularity is defined by the business domain model. The services are the logical abstraction from the master data management model. Inside the business services, do not worry about InfoSphere MDM Server specifics or how an individual domain object is mapped to the internal structures, but instead focus on handling the domain objects as a whole.

Designing the Business Entity Services is a complex task and closely linked to establishing the InfoSphere MDM Server mapping on the object level. It should be left up to product experts, because apart from functional aspects, there are also non-functional aspects to consider, for example, optimizations regarding retrieval operations. Specification of these services goes hand-in-hand with the need for additional data services in the InfoSphere MDM Server service model. From a logical perspective, you need to think of the entity services as an orchestration of lower-level data services.

Data services are not a direct part of the service model. You do not have to design them as part of the service models. They still come into play in such way that the service model describes the mapping to the data services. All higher level business services perform a service choreography against the data services. The data services already belong to the InfoSphere MDM Server models. Note however that in the service model, especially when it comes to designing business entity services, there is no difference in the modeling technique, whether you start a service from the service model or from the InfoSphere MDM Server service model.

Model structure

The service model holds most of the design work you do in the context of the system design. The number of elements increases with each model. A business process breaks down into individual use cases, and use cases can introduce some elements of reuse. In the service model, you can break down every use case into a number of services. Each service in turn can break down in to many subservices (for more information, see “Defining a service” on page 350). As a result, you need a structure that is easy to handle in terms of readability and collaboration. You work on different aspects in parallel, so you need a structure that follows the typical cohesion principles of object orientation or component modeling.

You should use the following distinct packages on the root level of the Service Model:

- UtilityServiceLibrary
- BusinessRules
- BusinessEntityServices
- BusinessTaskServices

The UtilityServiceLibrary (Utilities for short) holds control flow logic that is not unique to the current master data management system. The library contains basic functionality that you require within your client-specific services. Examples are StringUtilServices (such as toUpperCase or subString) and DateUtilServices (such as getCurrentTimestamp). When you transform the service model on the implementation model level, you can map the contents of the library to available concepts or function libraries available within the platform environment.

The BusinessRules package reflects a structural guideline that we previously used in the Use Case Model as a container for the object-independent rules. It serves the same purpose for the Service Model. Each rule becomes a stand-alone operation, attached to a provider, which in turn is in the BusinessRules package.

The BusinessEntityServices package is the central container for all entity services. As described in Chapter 6, “The modeling disciplines” on page 261, these services take on a special role in the overall context because they are the bridge to the InfoSphere MDM Server specific structures. Therefore, you should keep them separate to allow for independent maintenance and optimization.

The BusinessTaskServices package, however, is the main container for all the business flows that you realize within the Service Model. A single container should not be this large, which is why you must further subdivide it into logical building blocks.

Defining a service

Before you turn your attention to designing business flows, you must look at the formal definition of a service. In UML terms, we describe a service by a single operation. The operation in turn needs to be attached to a Class. Because you know only defined packages, what makes up the class in this context? Within the Use Case Model, the domains provided a sufficient structure that avoids overloading a single package. We simply added each use case directly to the package. If you look at the different types of services, and you want to focus on reuse and decomposition into logical building blocks, the number of services is larger than the number of use cases. If you define a single Class only and add all of our services as operations, this action can be confusing and cause problems in later transformations because of the number of operations defined.

To avoid overloading a single element, use service providers. In modeling terms, a service provider is a stand-alone class with the <<serviceProvider>> stereotype. In the insurance example, you now find the different service providers for Business Task Services within the Party domain, as shown in Figure 7-30. The Client, Beneficiary, and Party for those elements are shared between Clients and Beneficiaries.

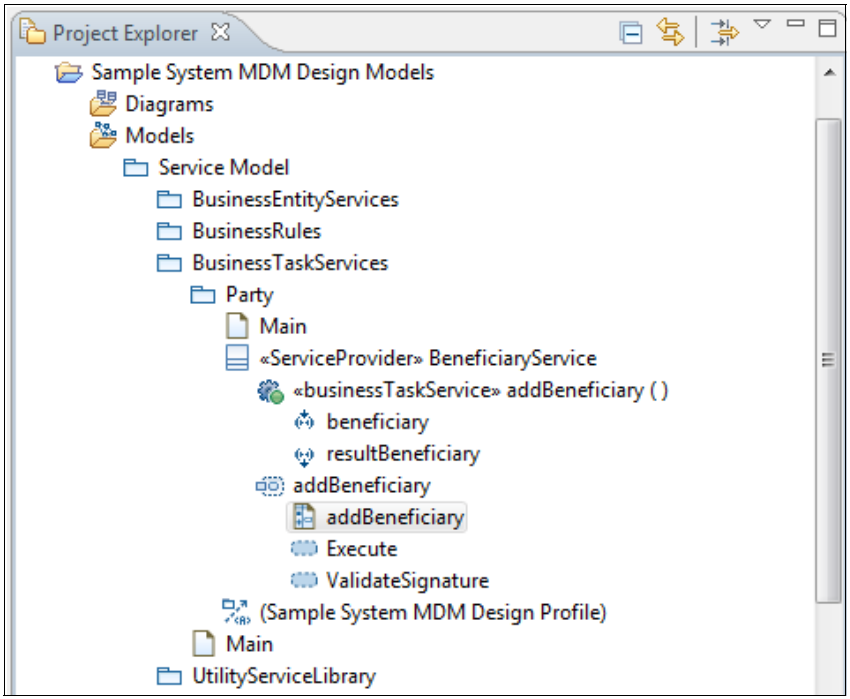

Figure 7-30 Sample structure of service model

For Business Entity Services, the decomposition into service providers comes more naturally, because we can immediately understand that the addBeneficiary-service on the entity level belongs to Beneficiary, and addClient belongs to Client. As a rule of thumb, each object entity from the Business Domain Model becomes a service provider in the corresponding domain package of the BusinessEntityService hierarchy.

A point to consider carefully is naming conventions. In a large project environment, creating stable naming conventions at an early stage is both crucial and complex. It is unlikely that you are able to anticipate conflicts at an early stage, yet changing names for either services or even entire service providers can trigger a long chain of modifications. Here are some aspects to consider:

- Refrain from overly long names. There can be system limitations regarding the maximum length of services or classes within a system. However, ensure that the names are unique and self-explanatory.
- Avoid umlauts, special characters, blanks, and so on. There are components that are unable to process them.
- Check your domains and domain model classes carefully. If you refer to our example, you notice that *party* is both a domain and a domain object within our model. Similar effects can apply to other objects, such as Beneficiary, which can be seen as a single domain object and as a collection of objects. To avoid conflicts, use a prefix or suffix to identify the feature. In our case, we refer to the party domain service provider as PartyComposite, and the domain object service provider is simply Party.

After creating the service providers, you can turn your attention to the individual services. In the service model, each service is composed of the following components:

- An operation on the service provider class, which defines name, signature, and type of service.
- Optional object instance diagrams that document service usage with respect to the service signature.
- An activity that contains the actual business flow.

Operation

The operation provides the formal entry point into each service. Apart from establishing the name, it is used to define other characteristics, most importantly:

- Type of service
- Signature

Although you can derive the “major” type of service from the package hierarchy, you want to ensure that each operation can be correctly recognized without navigating the package hierarchy. Therefore, tag each operation with either the `<<businessTaskService>>` or the `<<businessEntityService>>` stereotypes. Looking back to the types of services, you must separate the internal Business Task services from the rest. Again, use a stereotype, in this case, an `<<internal>>` stereotype. For example, the internal services are never published as part of a web service definition, because you do not want to make them available for consumers.

The method signature, however, defines the domain objects the service deals with. Contrary to the use cases where we chose to informally introduce all involved objects as a simple parameter node, here we strictly define the order of parameters, types, names, and so on. This definition must be done in a formal manner, as the information from this model is used downstream to generate code. Details regarding parameters and return values are described in “Service parameters and return values” on page 355. You can work under the assumption that parameters and activity input/output pins always appear as matching pairs.

Object instance diagrams

While reading the list of service components, it might have struck you as odd that we suggest object instance diagrams to detail the service signature, especially after we determined the method signature to be a formal guideline. Unfortunately, the method signature cannot express the relevant information in detail. Consider the following items:

- When designing the domain model, we capture all existing relationships among objects. Although this process is powerful, it also opens the service signature to speculation because the signature does not show the detailed information required to properly manage its use.

  Take the addBeneficiary-service from our sample project. The only defined parameter is a Beneficiary domain object. However, you can use the domain model to perform various actions: attach a name, attach policies, attach claims to the policies, attach clients to the policies, and so on. The method signature offers us no way of prescribing which attached objects are handled, which attached objects are required, and which attached objects are prohibited.

- You might argue that it is best to allow only those objects that are explicitly listed in the method signature and ignore or even prohibit any further objects. However, this definition limits the complexity our services can reach.

  Assume that we want to define a service that allows the creation of a client with various policies at the same time, where the validity (attached to the ClientPolicyRole) can vary for each policy. The signature must accommodate one Client, and n combinations of ClientPolicyRole and Policy. To offer this service, you define a method whose signature holds one Client, n ClientPolicyRoles, and n Policies. What the explicit method signature is unable to achieve is maintaining the link between Role and Policy, except for the weak link based on the order of elements in the two lists. However, examples quickly show that this method also does not work in all circumstances. Hence, you must maintain object relationships as part of the service signature.

Combining the above two situations, we conclude that we can only compromise between formalism and flexibility:

- Rely on a formal service signature that shows the logical building blocks that make up the service. These blocks are strongly typed, named, and can be validated.
- Add object instance diagrams that show the object networks to expect in request and response.

We prefer object instance diagrams over class diagrams, as they allow specification of cardinalities that exactly match the service that is available. This situation is in contrast to extensions of the class mode, which either clutter the domain model with additional associations or have too little precision to describe just how many objects are actually in play when reusing the existing associations.

Activity

To accommodate the actual business logic contained within each service, use UML activities. UML activities were used in to create a semi-formal specification for the use cases, and we now use them to create a formalized notation for the service design. Although we describe manual steps in this section, transferring the use case analysis activity to the design activity should eventually become part of a transformation.

Inside the service provider, create an activity with the same name as the operation. This activity contains the entire logic that makes up the service. Before you move on to the contents, add the `<<serviceDesign>>` stereotype to the activity to mark it as a service design and clearly separate it from, for example, the corresponding Use Case Analysis activity. Do not forget to create the relationship between the service design and the corresponding use case in form of a realizes-relationship.

Inside the activity, we distinguish between two separate phases:

- Signature validation
- Service execution

The first phase, validation of the service signature, is in the pre-execution state. These validation steps run before the actual service execution. Ensure that the service invocation is formally correct and the services are interconnected. Ensure that the parameters passed to the service method meet the requirements of the service. From a model point of view, use a structured activity named “ValidatePreconditions” that drills down into all the relevant details.

In our example project, we have a “addBeneficiary” service, which expects to receive a Beneficiary domain object with exactly one name attached to it, in contrast to the domain model, which allows multiple names. If the checks become more complicated or many checks must be performed, different activity nodes should be introduced to form logical groups. This situation not only increases readability, but also allows for easier extraction and reuse if the same checks must be enforced across multiple services.

The second phase, service execution, contains the actual business logic and details the previously designed use cases. Similar to the signature validation, use a structured activity named “Execution”.

Design elements for control flow

When it comes to the service design, you are at the heart of the business logic. At this level, you need the fully formal specification of all scenarios. To achieve this necessary level of formalism, you need various language aspects. In contrast to a textual language, a graphical version enhances human readability. Thus, your challenge is to find a compromise that is easy to understand, but also where you can express details in a machine-readable format. UML in general, and the activity diagrams in more detail, provide a wide range of concepts to formally express logic, constraints, and flows. You can ensure that, even without specific extensions, UML offers a place for almost all of your requirements. However, the flexibility of UML is high and a number of concepts are similar. Which concepts are used under certain circumstances is often a matter of agreement between the involved parties.

Service parameters and return values

For each of the service designs, it is not only important to define what it does, but also how it interacts with the external service consumers. In the introduction to this chapter, we described the different approaches you must take to obtain a clear definition:

- An operation with parameters
- Object instance diagrams that define nested structures that can be handled
- An activity with pins corresponding to the parameters

For the remainder of this chapter, we do not consider the instance diagrams much, as they are currently little more than a visual representation of the structure, although they can be considered for validation. Considering a full fledged automated model-driven testing, you probably want to define these instances, as they define test scenarios as soon as you add attribute run-states. They do not serve the service design itself directly.

Regarding the parameters and activity pins, keep in mind that both concepts are linked. Each pin represents a parameter and each parameter must be offered as a pin so that you have well-defined interaction points. On the level of operations, you can use UML to define each parameter's type:

- IN
- OUT
- INOUT
- RETURN

An operation can have at most one RETURN parameter. The reason for this constraint is to create clear responsibilities and cohesive services. Think of an operation that returns both a Claim and a Policy object. Would this service be responsible for claims or policies? And how does changing one object affect the state of the other one? Ideally, a service should never return more than one object and the RETURN parameter provides a means of expressing this constraint. You cannot entirely rule out the possibility of having more than one return parameter. A good example for this scenario is a service that returns a technical wrapper object. Although programmers use this wrapper object in the signature, there is no need for the designer to model a purely technical object. Instead, he can simply return all of those objects that are returned within the wrapper structure. In this specific case, using a notation with multiple OUT parameters helps us understand the interface of the service. Therefore, we decide to disregard the RETURN parameter for our service design and rely on the OUT parameter only.

The IN parameters are used in the service signature for all data that is strictly fed to the service but not modified inside of it. The data type must be chosen from the list of available types from domain model, primitives, and the custom object model. Taking the example of a service that adds a claim to a policy, this claim could be one that is persisted, and the policy simply serves as a reference. Inside the Activity, create the corresponding Input Pins to introduce the individual parameters into the flow. The number, name, and data type of these pins correspond to the operation parameters.

The general principle is similar for OUT parameters. We already described the issue of multiple parameters and artificial wrappers. Therefore, allow an operation to define multiple OUT parameters. Again, each of them is reflected by an output pin on the respective activity.

Regarding INOUT parameters, the design goal is to have a well-defined signature that reflects what is changed and what is unchanged. We want a stable basis for design, we do not want objects to implicitly change, and we want to be able to clearly see in what form they were fed to an activity and how they are returned in comparison. The problem with object states, specifically resolving business keys to technical keys, is something that finally motivated us to allow the INOUT notation after all under certain circumstances. Although it is not preferable to use the notation for objects that are modified based on the business logic covered by a service, it is helpful for such objects that require it only as a point of reference.

Returning to the previous example of adding a claim to the policy, logically the flow persists a claim and requires only the policy as a point of reference. But while the policy can be specified in form of a business key only, we might decide or even need to resolve the technical key within the service, and this situation is where the INOUT parameter comes into play. We can use the parameter to offer the resolved object, although it is not explicitly modified by the business logic, to the calling service for further usage.

In your own projects, you might decide to use the INOUT parameter for various purposes, but this situation is one where we think it makes sense to allow modifications to an existing object. Reflecting this type of parameter on the Activity level is unfortunately a little more complex, because you get to pick only from input and output pins. So, you must introduce both an input and an output pin, bearing the same name as the parameter (our only way of bundling the three together). By convention, the input pin delivers the unmodified input object, while we feed the modified object to the output pin for use after the activity.

For IN, OUT, and INOUT pins, there is no restriction regarding the number of times a value can be read or written to them. Input pins constantly deliver the originally passed input value, and output pins eventually return the last value assigned to them.

Although it is not possible to create object flows to the input pin, it is possible to attach other objects to the original input, thus effectively modifying an input object. We can define only a design guideline against it. Objects obtained from input pins must not be modified in any way, including linking them to other objects, unless they are either returned by an OUT or an INOUT parameter.

Errors and warnings

Errors and warnings are independent classes in a dedicated section of the domain model. We add design related technical errors and warnings as part of the service design. In a fail-fast error approach, you do not need to worry about output any further, because we stick to the Send Signal Action approach, and immediately end our control flow without returning output values to the caller.

The situation is different for pooled errors and warnings. In both cases, you need a point of reference both internally to set the data and externally to work with the data. You see a significant difference between the actual source code and the design. On the design level, you need to introduce explicit OUT parameters and corresponding pins. By convention, they are called “warnings” and “errors” and carry the corresponding stereotypes. When you implement a InfoSphere MDM Server, this information does not need t be populated explicitly as part of a response, but is handled by the master data management meta structures that you implicitly have available on all your operations.

Starting operations

When you look to the Domain Model and the Service Model, you can see that we intentionally choose similar structures for interaction with attributes and services. In both cases, we create operations. You can use these operations to use the same element for all interactions from within our service design. Regardless of whether you must start a method on an object (for example, to obtain or set a value), or you want to start an entire service, add a Call Operation Action to the activity and select the respective method from
the models.

After you add the action to the activity, specify the name to match Class, Operation (for getters/setters), and Operation (for services). This action provides visual help when someone else tries to understand the control flow. In the next step, establish the correct data flows. Connect central buffers or activity input pins to the input pins offered by the operation. If required, capture the object flows from the output pins. At the least, this action should be done for all strict output parameters that resemble modified data. For INOUT parameters, it is up to the service logic to decide if it makes sense to pick up the modified object or continue work with the existing object. In most cases, you do not want to influence the warnings returned by the operation. In that case, establish a direct connection between the operation’s warning pin and the current activity’s warning pin, indicating that the warnings are forwarded. You can inspect the warnings and decide whether you want to ignore or modify them.

When you call an operation, you cannot automatically assume that it succeeds. In many cases, there might be no way for you to deal with failure except to cancel you own activity as well. If you are working with a fail-fast approach for errors, cancellation is automatically the case if we do nothing to the contrary, the signal is sent from within the operation, and control flow never really resumes afterward. In the case of the error-pooling approach, the error or multiple errors are returned on a specific output pin. If you think you cannot deal with the problem within the current activity, proceed as you did with the warnings and create a data flow from the operation’s output pin to your own pin, thus forwarding the errors.

The situation changes if you need to inspect or possibly even overrule the error. Think about a service in our example context that retrieves Clients based on the business key, the CustomerMasterReference. When you build this service, you must decide what happens if the client cannot be found. You can either return nothing, or we raise an error. In the former case, you must deal with the possibility of not receiving a result in the calling activity, while in the latter case you must deal with the error itself. Assume that you have a rule that checks whether the CustomerMasterReference provided is unique. One way to accomplish this task is to simply start that lookup operation. If you chose the error-approach, the operation you call from the rule raises an error, either in form of a signal, or by returning it through the corresponding output pin.

How can you deal with this error and make it disappear, so that your rule succeeds and the control flow continues? Again, you must take both alternative error handling concepts into consideration. For the error pooling, when the control flow resumes after start the operation, you must inspect the ErrorObjects, see if any were returned, and whether the ignorable “not-found”-situation was among them. If you encounter errors other than not-found, you must forward them to your own error pin, while in the other case you can simply continue with your control flow. Because of the signal approach, your design slightly changes for fail-fast errors. Instead of continuing with the control flow and inspecting the error pin, you must introduce an Accept Event Action for exactly the error signal you want to ignore. From that action, you resume your regular control flow. What this situation logically means is that any non-accepted signal continues to terminate your activity, while you proceed with regular control flow if you do catch the respective signal.

Object state

Service signatures specifically state which objects go in and come out of a service; the only unknown factor is the state of the objects as they come in. Mandatory versus optional settings, or specific rules attached to an object, should ensure that you do not persist data that you consider inconsistent. But there is an unknown factor when it comes to the use of keys to identify an object. Are you working on business keys that still need a lookup? Or have you already performed a lookup and therefore hold the technical keys? From a high-level perspective, the difference between the two might not even be obvious.

Whenever you use an object in a service, you must check if you have a technical key; if you do not, perform the lookup. However, a service does not indicate whether it retrieved the object and that the “new object” contains the technical keys. If your service calls subservices A, B, C, and D in that order and passes the same object into all services, you resolve the business key four times. However, your orchestrating service has no idea whether any of were resolved. From a design point of view, you do not even know whether you already hold the resolved object without looking into it.

This situation is where a set of stereotypes comes into play. For the objects that you handle, explicitly specify what you know about them, which can be the <<businessKey>> stereotype (it is possible that the service contains only a business key) or <<technicalKey>> stereotype (you are certain that the technical keys are present).

Figure 7-31 shows the usage of the <<businessKey>> stereotype in the signature of addClaim, where the claim comes in as a regular object to be persisted while the policy serves as a business key only.

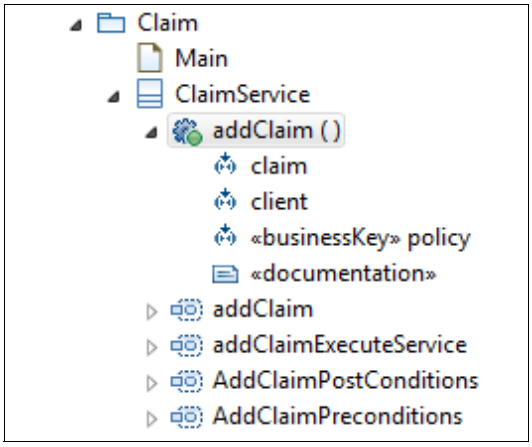

Figure 7-31 The usage of business keys in the signature

Access to attributes and variables

When you describe the business flows in a structured manner, you must interact with the entities from the domain model. For example, you might want to evaluate a client's VIP status, or specify the IssuingDate of a policy. In other cases, you might need to be able to transport information independent of a domain object. In either case, you could use the following different structural aspects:

- Interaction with domain entity attributes
- Handling of variables

Because you refined the Business Domain Model to include setters and getters for each of the contained attributes, you can now use them as part of the service design. Within your business flow, use the UML call operation action to start the respective getter or setter operation, as shown in Figure 7-32.

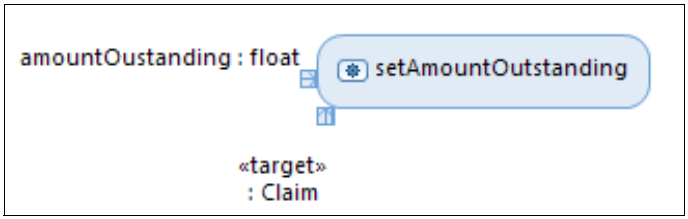

Figure 7-32 A call operation

Add an action to the activity, select the existing domain object on which the operation is to be performed, and then connect all available input/output pins to specify the necessary parameters and obtain the result. As action name, choose a meaningful description, preferably the name of the underlying operation. During transformation to the Implementation Model, the invocation can be translated to a method invocation on the runtime object.

Regarding variable handling, the first thing you must do is find an appropriate container within UML that you can use to transport data. Select the CentralBuffer to be that container. To create a variable for usage within the flow, add a CentralBuffer to the activity and specify the type for your needs. This type can range anywhere from a domain model object to simple primitives such as strings or integers. To identify it as a temporary variable, mark it using the dedicated stereotype `<<variable>>`. If the buffer is designed to hold more than one value, also add the `<<list>>` stereotype.

Buffers are meant only to store data that is created elsewhere. Thus, to specify the current value for the variable, create a data flow towards the CentralBuffer. In the opposite direction, using the current value, create a data flow from the CentralBuffer to the element. During transformation to the Implementation Model, the CentralBuffer translates to a newly defined variable of the respective type, and the respective data flows become assignments
and readouts.

In full formalism, you must ensure that you can use the fixed value in an object flow so that you can feed it to methods or other activities for use. Start by adding a plain Action element to your activity. For its name, select the fixed value, for example, 42 or true. For string literals, include the value in single quotation marks. If you require single quotation marks as part of the value, use backslashes as escape characters, as is common practice in several programming languages. Next, add the `<<fixedValue>>` stereotype to the action.

It allows subsequent transformations to recognize this action as a special element. To use the value at a later point, add an output pin and select the appropriate type. Now you must define the actual value in UML by adding a value pin to the action. Again, specify the type and navigate to the advanced properties. From here, take the necessary steps to specify the value-attribute, for example, create a string literal and assign the actual value.

You formally specified the value and, using the output pin, made it available for usage within the control flow. If you try to accomplish this task in IBM Rational Software Architect, you might find that the literal specification takes a couple of steps, which is why we suggest a shortcut. Although this method is a misuse of the UML semantic, you complete the following steps when you are done creating the action:

- By applying the `<<fixedValue>>` stereotype, you know that it is a special element.
- By creating the output pin, you made the value available.
- If you use the naming convention mentioned previously, you can derive the actual value from the element's name.

Although this mixture of name and value is not entirely correct, it does simplify things. All you are trying to achieve is introduce a constant value, which should not be highly complex and time-consuming.

Basic operations library

The elements we describe now are utility functions in the sense that they are string operations (for example, substring or toUpperCase), date operations (for example, getYear or getCurrentTimestamp), and so on. Here we also describe operations that operate on collections, such as determining the number of elements in a list, and complex language features that you might need, such as instance determination, that is, whether a domain object is of type A or B.

When you look at the Service Model, it is natural that you require the same elements on this level as well. You do not try to mirror a specific programming language, but try to have the same richness that is required to adequately describe your business logic.

Our goal in this area is twofold. For service designers, we offer a comprehensive set of operations that they can use to complete their design, while from the transformation and implementation perspective, we must ensure that we can map all of the offered operations to the elements of our target platform.

On the root level within the Service Model, we have a dedicated package named UtilityServiceLibrary. Within it, you find a number of classes with the <<UtilityServiceProvider>> stereotype. Each class has a number of available operations that can be used to cover the common tasks in the service design. Our transformations, however, need the appropriate logic to translate these operations to the target platform. We do not need to design the contents in any way, but map them directly. If you feel the need to extend the available utilities you are free to do so, but you must ensure that the downstream transformations are adapted to include mappings for the respective operations.

Formalizing conditions

Handling conditions in the control flow is perhaps one of the most complex structural elements we must describe. The complexity comes from the UML modeling language itself and the actual logic to be designed.

To introduce the condition into the control flow, add a Decision Node element and connect the control flow to it. You must decide whether you want Boolean decisions, that is, two outgoing control flows or whether you want to allow multiple outgoing control flows. The former is easier to understand but requires multiple decision notes to resemble a one-in-many decision.

As you look closer at the decision node and also at the general options, you notice that UML is flexible when it comes to control flow that results from a decision. Instead of defining the logical conditions directly on the node and thus making one of the outgoing control flows the active one, UML defines the conditions as independent guards on each of the outgoing flows.

From a modeling point of view, this element is a highly powerful one, meaning that the decision node can actually split a single incoming flow into multiple active flows, if the guards are not mutually exclusive. For our Service Model and our rules, this situation is the opposite of what we are looking for. The services are single-threaded and sequential, and we need each decision to have exactly one active outgoing control flow. Because UML leaves this control flow open, the following rules apply:

- Per design convention, a decision node must have exactly one outgoing control flow that becomes active at all times. All guards specified on the flows must be mutually exclusive, and the decision must never lead to a situation where none of the guards evaluates to true.
- From the tool support, you can try and check for this convention as part of the model validation, but you are not able to guarantee correct detection of erroneous situations. The conditions can be highly dynamic and depend on data contents that you cannot anticipate during a static validation.

After you establish these guidelines, the following elements make up a condition from the modeling point of view:

- A Decision Node with a descriptive name
- n (preferably two) outgoing control flows with descriptive names and formally specified guards

After adding the Decision Node, you should specify a descriptive name. The preferred approach is to use a textual description of what the decision stands for, for example, “Person above 18 years of age”. If the underlying decision is trivial, such as the above case, you could also decide to repeat the exact guard definition as the element’s name. Note however that the name in this context is only relevant to the human reader and does not play an important role in the downstream activities.

For each of the outgoing control flows, you must specify a name and formally define the guard. You should choose an adequate name that is related to the name of the decision node; again, the name is not important to the downstream activities, but is relevant to the human reader. A good name answers the “question” picked as the name for the decision node, so in the above age examples, good names for the outgoing control flows would be “yes”/”no” or “true”/”false”. You should establish guidelines regarding this wording as part of your own design guidelines. Finally, perform the most important task regarding the decision, that is, specify the condition according to the previously mentioned exclusive criteria. UML offers various different “languages” to define the guard. For our purposes, we briefly describe Object Constraint Language and Analysis.

Object Constraint Language (OCL) is a language that allows for the declarative description of rules that apply to one or more objects. It is part of the UML standard and highly formalized, making it ideal for your purposes. In IBM Rational Software Architect, you can attach guards defined in OCL directly to the control flows. There are two things to consider:

- The logical conditions we are trying to describe are typically simple, evaluating one or more attributes on a domain object, possibly some conditional operators. Alternatively, OCL is a comprehensive language and might not be easy to understand, so basically your service designers must follow the design guidelines and have knowledge about OCL.
- You must consider what happens to the conditions in the Implementation Models. Although the advantage of standardized languages like OCL is that you find tool support for transforming OCL into other languages, you ultimately translate them to something like plain Java code. In many cases, service designers might be familiar with such programming languages and thus easily grasp the notations used for conditions there.

If OCL too complex or too powerful for your purposes, you must find an alternative. Because you can use UML to specify guards in an analysis language, you can define a formal, simple language to specify conditions on this level. During model validation, you can verify whether the conditions are correctly specified, and your transformations can take care of converting the expressions to the respective language platform.

In our example, the simple language elements are derived from the Java programming language as follows:

- Our expression can refer to any named object and any variable (here named CentralBuffer) available within the context (here Activity). In addition, we can use literals enclosed in double quotation marks, numeric values, and the keyword `null` to check for the absence of an object.
- We can access attributes within a named object by applying a simple dot-notation: Object.AttributeName.
- We can check for equality of values by using the “==” operator and perform detailed comparisons with “>”, “>=”, “<”. “<=”, and “!=”.
- We can introduce logical connectors by using “&&” (logical AND), “||” (logical OR), “NOT” in the typical precedence not-and-or, and rounded brackets “(“ and “)” for groupings for operator precedence.

Using the above notation, we can formalize the age-check as “Person.age >= 18”. Figure 7-33 shows the definition of this constraint in IBM Rational Software Architect.

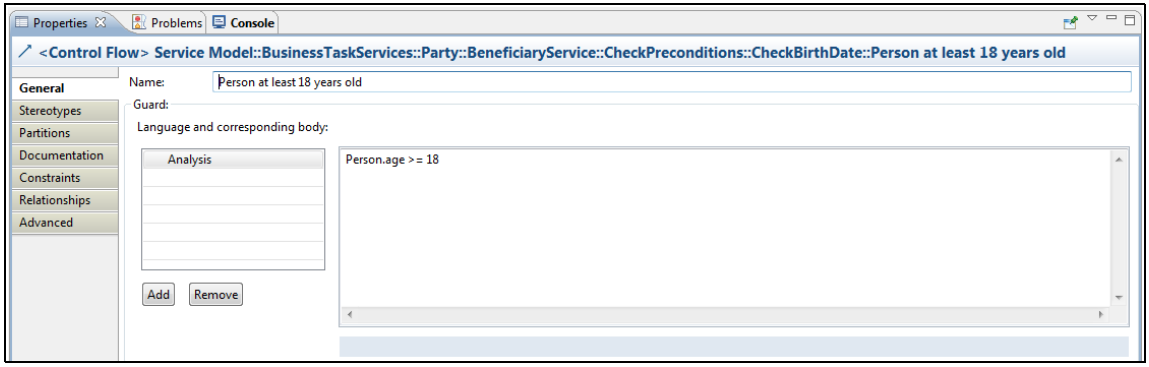

Figure 7-33 Example for an analysis constraint

If your designers can use OCL, rely on that standard. Alternatively, if OCL turns out to be too complex and you do not have the necessary skills available, take the easier approach and use the analysis-based format. It is rarely a preferred practice to overrule existing standards and reinvent the wheel, but designers quickly adapt to the activity modeling approach, and many of them are not familiar with OCL. So even though many constraints look similar regardless of whether you use OCL or a formalized, Java like notation, designers might be reluctant to learn an entirely new language just for defining constraints.

Ensuring that the guards that you define are mutually exclusive is not guaranteed. Consider the consequences of converting an ambiguous decision node into a sequence of if-else statements: you might never have a problem if the first if-branch evaluating to true is always selected.

You might find that there are various possibilities within UML activity elements that you can use apart from the decision-node, for example, a conditional node. Per the UML standard, this element presents a one-in-many choice, designed as a series of conditions (“tests”) and corresponding bodies. This element seems more appropriate choice for a constraint in the context of mutually exclusive conditions. However, if you take a closer look at the definition provided in the UML standard, this element cannot be guaranteed by the conditional node either. Taking this situation into account, and checking visualization and usability in the modeling environment, the decision node seems to best fit your needs.

Loops

In general, you use a *loop* if there is a set of one or more steps (actions or activities) that is repeated a number of times. In the most extreme form, you might never run through all the steps. The steps that are repeated are sometimes referred to as the *body* of a loop (in UML terms, it is called the *body part*). Next to the body, there is a second component called the control structures. The control structures dictate under which circumstances the body needs to be executed. There are different types of control structures.

In UML, loops are encapsulated by a structured node that is embedded in to the activity as a whole. The internal and outside structures are strictly separated, and interactions must be explicitly specified. You must ensure that all of the relevant information is explicitly connected to the loop element using input pins, where the information can be used internally, and obtain all the relevant information from the loop using output pins.

This setup prevents any interference by changes made in loops on their effects on the outside structures. These explicitly exposed structures means that you can nest loops. In code, developers easily nest two loops within one another.

From the design perspective, this situation means arbitrary object flows to and from the different levels, which need to specify which elements are in what state and on what level, can quickly become hard to understand. Again, by using the structural elements, you must define what objects are handled on what level. If you work with two lists, each of them traversed as part of one loop, you can see when you move from a collection of elements to a single element on the one list; the other, however, is still seen as a collection.

Generic pre/post-checked loops

A generic loop in the context of UML is composed of the following distinct sections:

- Body
- Test
- Setup

The body holds the repetitive elements, while the test contains the logic that decides whether there is a repetition. In addition, there is the so-called setup part where data can be specified and initialized for use within the test and body. By convention, the setup part is run exactly once before the first run through the loop, while test and body are run for each iteration. Loops can be pre- or post-checked. A pre-checked loop runs the test before the body (leading to anywhere from 0 - n repetitions), and a post-checked loop runs the test after the body (leading to anywhere from 1 - n repetitions). Although there is no immediate need to support both forms in a service design, it helps to have different concepts available and pick the one that most adequately fits your needs.

Support, handling, and visualization of the loop node in UML (introduced as of UML 2) varies widely between the different modeling environments. Within IBM Rational Software Architect V7.5, the visualization does not separate the three sections, and the different properties required to configure the loop are in different dialogs. So in the following paragraphs, we describe both service design needs and where the individual elements can be found in IBM Rational Software Architect.

Start by adding a loop node to your activity. You can connect the control flow just as you do for any other element. For naming, use a brief description of the purpose. The element is not used for functional purposes downstream, but serves as documentation and enhances readability. Keep in mind visualization; you might not be able to correctly interpret the contents of the loop when looking at the containing activity. In addition, add the <<loop>> stereotype to the element for completeness. Within the advanced properties, locate the “Is tested first”-property and adjust it to your needs. By default, loops are post-checked.

In the next step, create three different Structured Activity Nodes within the loop node. This way, they are logically bound to the loop and appear as nested structures. As names, concatenate the loop name with ".Setup", ".Test", and ".Body", and add the corresponding stereotypes <<loopSetup>>, <<loopTest>>, and <<loopBody>>. This way, you ensure that you recognize the elements in searches and other contexts.

After you create the activities, turn your attention to the interaction with the outside world. UML has the following properties on loop elements with predefined semantics:

- loopVariableInput
- loopVariable
- bodyOutput
- result

LoopVariableInput is a series of input pins, and the other three elements are output pins. The general concept is that the loop variable inputs are fed at the beginning and serve as initial values, and the result pins are used to publish the loop outcome to the outside. The loop variables are available as data pools during the execution. They are initialized with the input values and overwritten by the body output values.

Loop variables can also serve entirely internal purposes. Developers might be familiar with a "for i = 1..n"-loop; the variable i would then be a loop variable, initializing to "1" part of the setup, checking for the "n" part of the test, and incrementing part of the body activity. You apply these design elements in a similar manner. Start by creating input pins on the loop, one for each element you must take on from the outside. They are automatically registered as localVariableInput. Next, create a series of output pins on the loop node, one for each element of data that you want to exchange either between setup, test, and body, or need to feed back to the outside because the manipulation is relevant to the outside world. Register each pin as a loop variable. Output pins on the loop become result pins automatically.

What follows works based on naming conventions. Create the appropriate input and output pins on the three structured activities and apply the same name as for the loop variables. The activities read and write those values, so direct object flows do not need to be created.

You have created the necessary pins to interchange data and can basically start using the regular service design elements to include the necessary logic in to the three activities. The only task that remains is to formalize the test activity: it must have exactly one output pin, preferably named <<decider>>, and stereotyped as such. The data type must be Boolean. To formally link it to the loop node, go to the element's advanced properties and select the respective pin as the decider. If the test-activity feeds the value true to this pin, the loop repeats, but false ends the loop.

For-each loops

In business logic, you frequently encounter loops that are simpler in nature and require you to work with each element from a collection of elements, for example, "go through all clients and set VIP=true". You can attempt to achieve this configuration with the above structure and set the current element as a local variable, but UML includes support for for-each-structures in the form of *expansion regions*. An expansion region is a structural model element for repetition that can be configured for parallelism that has collections as input/output while operating with a single element from the respective collection internally.

To create a for-each loop in a service, add an Expansion Region to the activity. Similar to the loop node, provide a descriptive name and add the <<forEach>> stereotype. For formality reasons, set the mode-property to iterative (note that there is only iterative processing).

Next, add the required input and output data. There is a distinction between the iterated element and all other parameter data for the region. The latter is specified in plain input and output pins. To specify the iterated element, add an expansion node. Although UML technically allows more than one input expansion node per region, we limit this configuration to exactly one to resemble the for-each-nature. Also note that an output expansion node does not necessarily have the same number (or type) of elements as the input node. It simply resembles the reverse notation. Although there are for-each-statements on the input, each iteration can contribute zero or more elements to the output node.

Within the expansion region, we continue to use the regular elements for service modeling. To use any of the input data, simply establish an object flow from the input pin or the expansion node to the respective element. Repeat for the data to be fed to output pins/nodes.

Lifecycle and rules

When we look at the business requirements and the design elements we have described so far, you see that we are entirely focused on the mainline control flow. What we left out is special elements, that is, lifecycle and rules. Which of these elements do you need to incorporate into the design?

Object-specific rules are always implicitly part of every control flow you trigger, because they are enforced by the rule framework when changes to an object are persisted. As such, there is no need to explicitly trigger them. For object-independent flow rules, the situation is different. Remember that we defined them as being stand-alone business logic that has a lifecycle of its own, but needs to be linked to the relevant control flows. This is a problem that you encounter on the analysis level, where you must ensure that the flow rules are triggered from all relevant use cases. Your task for the Service Model is therefore to carefully trace how the Use Case is decomposed into services, where there are elements for reuse and so on, and then make sure that you explicitly start the operations that represent those rules. The details of such invocations have already been described.

Lifecycles are straightforward in their translation from the use case to the service level. There is a state-machine-based approach to defining the lifecycle, and the lifecycle transitions are (partly) triggered by signals. Similar to the flow rules, this decision is an explicit one, given the knowledge what a certain service is doing. Again, you must be careful when decomposing a service into reusable blocks, so that you can guarantee that no trigger events are missed and at the same time you do not illegally or unnecessarily trigger the lifecycle when nothing has changed. To send the respective signal, use the same structural elements used in the use case analysis model and add a send signal event to the control flow, selecting the appropriate signal defined in the domain model for that lifecycle.

7.3.3 Logical data models

This type of model relates strongly to the source and target systems that surround our master data management system. The master data management system also has a logical data model, much like the underlying InfoSphere MDM Server product. However, here we exclusively refer to the source and target systems logical data models. They provide us with an abstracted view of a set of relevant data. You can consider the domain model as a logical data model. However, the domain model serves additional purposes in that it holds rules and lifecycle information that you do not usually maintain on a logical data model. This view is independent from the physical implementation.

Whether you are able to locate logical data models for the source and target systems that are related to our master data management system depends on the organization and the data modelers that created the surrounding systems. Our main tool for describing logical data models is the IBM InfoSphere Data Architect.

The role of the logical data model is limited in our modeling methodology. Although we do not depend on it from a functional point of view, we might rely on it in respect to data mappings. Chapter 8, “Mappings and transformations” on page 411 introduces the distinction between static and dynamic mappings. Mappings has an impact on the requirements for some models. Static mappings are defined on the layer of physical data models, while the dynamic mappings are applied on the object model layer. The logical data models are between these two, as shown in Figure 7-34.

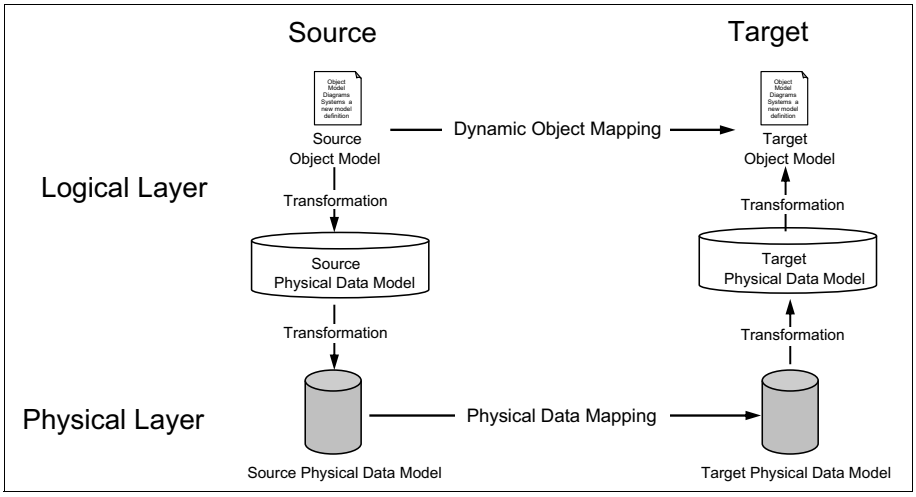

Figure 7-34 Conceptual source target mapping

Figure 7-34 represents an example where you have a domain model as the source and integrate it with a master data management system, which is the target. The complex dynamic data mappings are described on the domain or conceptual layer.

You might want to draft a static mapping too. It is much quicker to accomplish this task using the IBM InfoSphere Data Architect on the physical data model layer. If you do not have a physical data model on the source side, you must transform the domain model through the logical layer to become a physical data model. You now have the model that you need to carry out the static physical data model mapping.

On the target side, InfoSphere MDM Server does not provide you with the required domain object model, so you must make one. We describe a way you can accomplish this task later in this chapter. For now, you do not have any such model, but have only a physical data model. Because the logical data model is almost a one to one abstraction of the physical data model, you can apply a rather simple transformation. Then you apply another transformation, which IBM InfoSphere Data Architect and IBM Rational Software Architect provide by default. That transformation is from a logical data model to a corresponding UML object model. This transformation is also a rather simple one to one transformation.

At first, this approach appears to render the logical models as unusable. However, our modeling approach must be suitable for complex projects, and typical logical data models vary from the underlying physical data model and justify their existence by providing business focus on the domain. These models are the basis for a full domain model as we describe it. In our continued description about logical data models, assume that the logical data models of the source and target systems are similar and do not differ in the way they are defined or in the way they are modeled.

Guidelines

When building a master data management system, you focus on data, or more generically, on information management. As such, data models have a special place in this methodology. Source and target systems that the master data management system connects with exist. It is easy to obtain a model, import it, and create more models, such as the data mapping models, and the service design and ETL transformation models from it. The data models for the source and target systems are available either in an existing and installed database schema, DDLs, COBOL Copybooks, and sometimes even in an entity relationship (ER) or similar data models.

Example 7-1 is an example for a COBOL Copybook definition for our simple sample system A containing the customer data.

Example 7-1 Example for a COBOL Copybook definition

```
01  CUSTOMER.
            05  BIRTHDT PIC X(8).
            05  MAIDNAME PIC X(32).
            05  SEX PIC X.
            05  GIVNAME PIC X(32) OCCURS 6 TIMES.
            05  VIP PIC X.
            05  BMI PIC S9(4).
            05  CUSTREF PIC X(16).
            05  FAMNAME.
                10  NAME PIC X(32).
```

```
    10  FROMDT PIC X(8).
05  OCCUPY.
    10  OCCTYP PIC X(64).
    10  FROMDT PIC X(8).
    10  TODT PIC X(8).
    10  EMPLOYER PIC X(64).
```

Even though the format is a technical one, it also serves as an example for a logical data model, because it is defined on the application level. The physical data could be entirely different and not revealed to you at all. The problem with this situation that you cannot use any of the information for your data mappings directly. You must adapt this information and use it to create the logical data model. You must import this structure into IBM InfoSphere Data Architect through the COBOL model import wizard to create a simple entity structure in a logical data model. The result could look as shown in Figure 7-35.

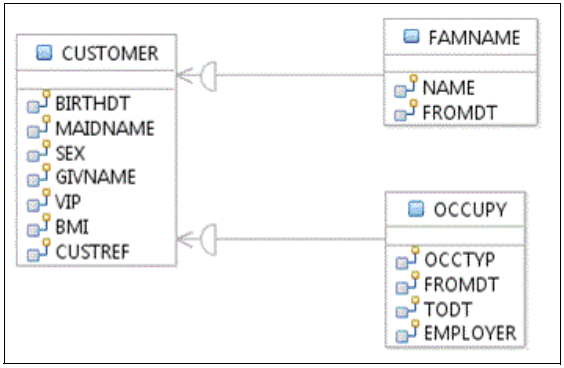

Figure 7-35 Sample logical data model imported from COBOL Copybook

You now have everything you need to carry out static data mappings. Because you do not always start on this layer and often have only the physical data model available to us, either through reverse analysis or with a DDL script, you should transform the logical data model to the layer you require. For static data mappings, use a transformation to the physical data model. For dynamic data mappings, use a transformation to the object model layer. If you have these models available to you, you do not have to import the logical data model, because it has no further relevance in this modeling methodology. You use this model type as a transient model to bridge the gap between the conceptual and the physical layer if neither is available to us.

You do not have to consider any specialties during the creation or import of the logical data models. See “Related publications” on page 559 for additional information about this modeling aspect.

7.3.4 InfoSphere MDM Server models

This section describes the InfoSphere MDM Server models. Contrary to the other models, which exist for each of the source and target systems, these models are solely available for our newly designed master data management solution. In our example, we build our solution so that it incorporates InfoSphere MDM Server as the product that is the central part of our core master data system. You can define custom logic, rules, flows, data structures, and everything else that defines a system, but always remember that you must design with the predefined product-related artifacts in mind.

InfoSphere MDM Server comes with an industry-proven data model and many predefined services. You occasionally encounter situations where the shipped product does not do what you need it to do. These situations are where the customization capabilities are useful. Whether it is new table that you must include in the physical, logical and object data models or a new service that represents our custom business service logic, you can use the customization framework to adapt the product as needed.

Do not overuse customizations: Be aware of the customizations that you make, and do not customize too much. Always consider whether it is better to adapt a product's default capabilities rather than expending the effort to customize.

You need to consider on which model you make these changes and what the effect of these changes on the current model might be. You can use*InfoSphere MDM Server Workbench* to perform many of the required customizations. The overall modeling approach described in this book uses this tool.

Figure 7-36 outlines the relationship of the InfoSphere MDM Server Workbench to other models.

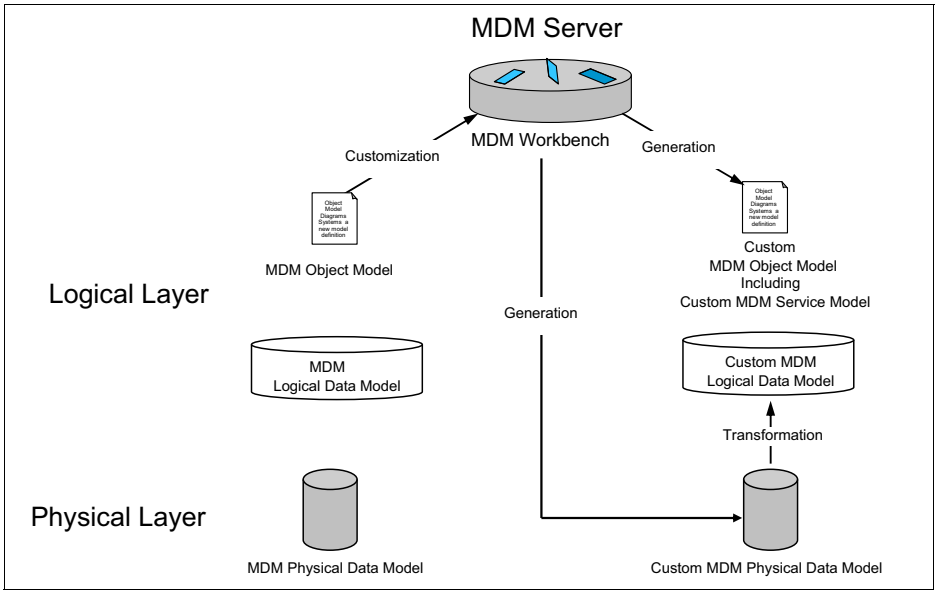

Figure 7-36 Customizing the logical data model using the InfoSphere MDM Server Workbench

As you can tell from Figure 7-36, the customization mainly refers to the implementation mode. The reason for using it in the design model is to create the mapping between the domain model and the master data management object model and the mapping between the service model to the master data management service model.

Additional information: For more information about customizing models with InfoSphere MDM Server Workbench, see “InfoSphere MDM Server Workbench” on page 549 and “Related publications” on page 559.

Related to design, InfoSphere MDM Server provides three distinct models. Unfortunately, none of the three models are currently available as an explicit model file in the master data management product package. This is why we split our description about these models into two areas: how we obtain these models in the first place, followed by guidelines about their customization. We refer to the static models the product provides as assets while the customized variants are the transformed or custom models.

InfoSphere MDM Server includes the following design models:

- Logical data model
- Server object model
- Service model

Logical data model

Similar to the physical data model, which is provided as a `.dbm` file, InfoSphere MDM Server also comes with an IBM InfoSphere Data Architect compatible `.ldm` file that contains the logical data model. It is nearly equal to the physical data model and is therefore a one to one representation with only a few exceptions.

Import guidelines

Because of the close relationship and similarity of the logical data model with the physical data model, it is easy to use the physical data model and transform it in to an equivalent logical data model.

The product has IBM InfoSphere Data Architect compatible **.**`dbm` and **.**`ldm` files and an Erwin compatible `ER1` file that you can import. In IBM InfoSphere Data Architect, perform an import through the data model wizard. However, this is not the preferred approach because there is no additional benefit to it. There is no reason to use this import option if you can import an IBM InfoSphere Data Architect compatible **.**`ldm` file directly.

Importing a **.**`ldm` file import directly is the preferred practice for importing. As with the steps described for the import of the physical data model, copy the **.**`ldm` file from its original location and paste it into the project. Instead of pasting it to a location in the implementation model project, as you would with the `.dbm` file, copy the `.ldm` file to the InfoSphere MDM Server design model project. After you copied it in to that location, refresh your workspace or project, and you can see this **.**`ldm` file.

Object model

The object model is used in two situations:

- When you transform the custom master data management independent domain model to your master data management object model. The object model might require further customization, which results in the custom master data management object model.
- When you design your services. You use the objects from the master data management object model or the custom master data management object model.

This model exists only in the master data management runtime environment. The master data management object model is static because it represents the product model as it ships. This model is modified and reused for every master data management solution project you implement. The customized models are essentially the base model, but are enriched or reduced based on your exact project requirements. These customized models are not meant to be reusable, unless you build add-ons to this project in subsequent phases of the project.

After you create the customized object model, you can use it for further modeling steps. Eventually, you must import a model that you can apply to a Java implementation. The code generator can also produce other artifacts, such as DDL scripts, XML Schema Definition (XSD) files (`.xsd`), and so on. Our main focus is on the Java implementation, because most of the dynamic parts of the design are transformed to Java code. In other word, the imported model should be engineered so that it can be eventually used in a Java implementation model. You do not actually regenerate source code from the standard master data management model, but a consistent master data management model helps you create correct customizations.

Import guidelines

The master data management object model is not provided with the product as an IBM Rational Software Architect compatible UML model, but it exists in a runtime manifestation. Instead of creating this model from scratch, which is a cumbersome and error prone process, consider importing it. Keep in mind that this importation is a one-time task that you need to repeat only with new versions of InfoSphere MDM Server.

You can import the standard master data management model from the following sources:

- The data models
- The Java API
- The XML schema

One of the options we have available for importation are the data models. Keep in mind that this option resembles database reverse-engineering. The consequence is that the master data management objects are named from the database instead of the Java API. The product delivers both the logical and the physical data model. We base our further import description on the physical data model. Relying on the logical data model that is also provided simplifies the following steps and makes the initial three steps obsolete.

To use this option, complete the following steps:

1. Create an IBM InfoSphere Data Architect data project and connect it to the physical master data management data model in your master data management environment. This environment can either be a DB2 or Oracle database.
2. After you extract the schema, you can save it as physical data model (`.dbm` file). Alternatively, you can import the `.dbm` file that is provided with the product.
3. The physical model must then be mapped to a logical data model (`.ldm` file). You can map the model as is. The mapping serves only to transfer the master data management data model to our modeling context. Alternatively, you can import the `.ldm` file provided with the product.
4. Take the logical data model and apply an IBM Rational Software Architect standard LDM-to-UML transformation to it.
5. After configuring and running the transformation, create a UML-based master data management object model.

The second alternative is also based on reverse-engineering. You create the model from the Java classes that are part of the master data management API. The result of this procedure is also a UML-based master data management domain model. However, in this case, the names of the objects match your needs exactly. Separate this model into the domains that the master data management uses, which are Administration, Common, Party, Contract, Product and Finance.

To use this option, complete the following steps:

1. Install InfoSphere MDM Server Workbench and import the master data management `.ear` file, as described in the master data management documentation. Afterward, the following projects are available in your workspace:
   - FinancialServices
   - Party
   - Product
   - BusinessServices
   - DWLBusinessServices
   - DWLCommonServices
   - DWLAdminServices
   - CrossDomainServices

2. Next, create class diagrams from these projects. Select the component package of each of the projects listed above and add all the business object classes in the package to a diagram. Exclude those business objects that are not real entities, but transfer objects or request/response structures. This situation applies, for example, to all of the search objects.

   This step produces `.dnx` files that can be used to transfer the domain object model to UML. To transfer the model, complete the following steps:

   a. Select all objects in one `.dnx` diagram, right-click, and click **Harvest**. A UML representation of the dnx class diagram opens.

   b. Repeat this step for all the `.dnx` diagrams you created.

The third option is based on reverse-engineering the UML model from the XML schema files that are used to specify the XML interface. The resulting objects unfortunately do not carry the exact class names that you need for the service design or other customizations. So you need to customize the source files or the transformation to make the names match the Java API.

To use this option, complete the following steps:

1. Refer to the `.xsd` files in your workbench and generate `.dnx` diagrams from these files. The files you need to transform are in the `XSD` folder of the CustomerResources project. The `myTCRM.xsd` file serves as an entry point.
2. After the diagram is generated, you can harvest the elements in this diagram and add them to your UML model.

Instead of harvesting the XSD elements and types, you can apply a transformation to the XSD by creating a transformation configuration and clicking **XSD-to-UML**. Use the `XSD` file as your source and an existing or a new UML model (`.emx` file) as the target (keep in mind your proposed package structures and location for these models). In the configuration UI, you need to select the merge strategy that the transformation uses. Because you create the default master data management model as a new UML model, you can choose either strategy. Running the transformation yields the UML representation of both the XSD elements and types that are defined in the XML interface.The actual types end with “Type”, so they do not exactly correspond to the names of the Java types.

From a formal point of view, creating the master data management model from the XML schema files is the preferred option. However, the result of the transformation differs slightly from what you want in the UML model. You can achieve the result you want by using InfoSphere MDM Server Workbench based re-engineering, which creates the object model directly using the Java classes that you are interested in. Thus, you get the best results from this option and must perform the least amount of rework or additional transformations that are required.

You could work with any naming standard in the design, but this situation would be rather confusing because it means that custom elements would use the same preliminary naming convention. You then need to perform further transformation before adding the naming standard to the implementation models, where you definitely must sort out this issue again. Apart from this issue, both harvesting and transforming the elements preserves the context of the types, that is, you can still tell by the stereotype where an element came from. Although this situation might be helpful in some cases, we do not use these stereotypes in our models and can either ignore them or remove them entirely. Using the data model is only a valid option if you do not plan to go beyond the scope of the data model. Otherwise, you need to provide a custom transformation to bridge the gap between the data model and the Java model.

A note of caution regarding the use of the XML schemas: Using the web services interface is not an option in InfoSphere MDM Server V9, because the API exposed through the web services differs from the Java API and the XML interface. However, beginning with InfoSphere MDM Server V10, you can extract the TCRM schema from the web services specification, because both the RMI and Web Service interfaces are standardized.

Service model

The *service model* is required for the custom service design. This service design is not just a traditional design that requires the definition of appropriate code that represents the service execution flow. It also necessitates the mapping against the services that InfoSphere MDM Server provides. This model is also closely related to the master data management object model in that it consumes the objects provided by that model.

This model also exists only in the master data management runtime environment. Similar to the master data management object model, we must create this model before we can use it. The master data management service model is also static because it represents the product model as it ships. It is another one of the assets that you build and reuse for every master data management solution project you must implement.

In addition to the static model information, that is, the signature the service provider offers, the transaction guides also reveal helpful information about the communication protocol, that is, how the information is being transmitted across the wire and transported between consumer and provider. The protocol model should be manually derived from these guides and built similar to the protocol model outlined in Figure 7-37.

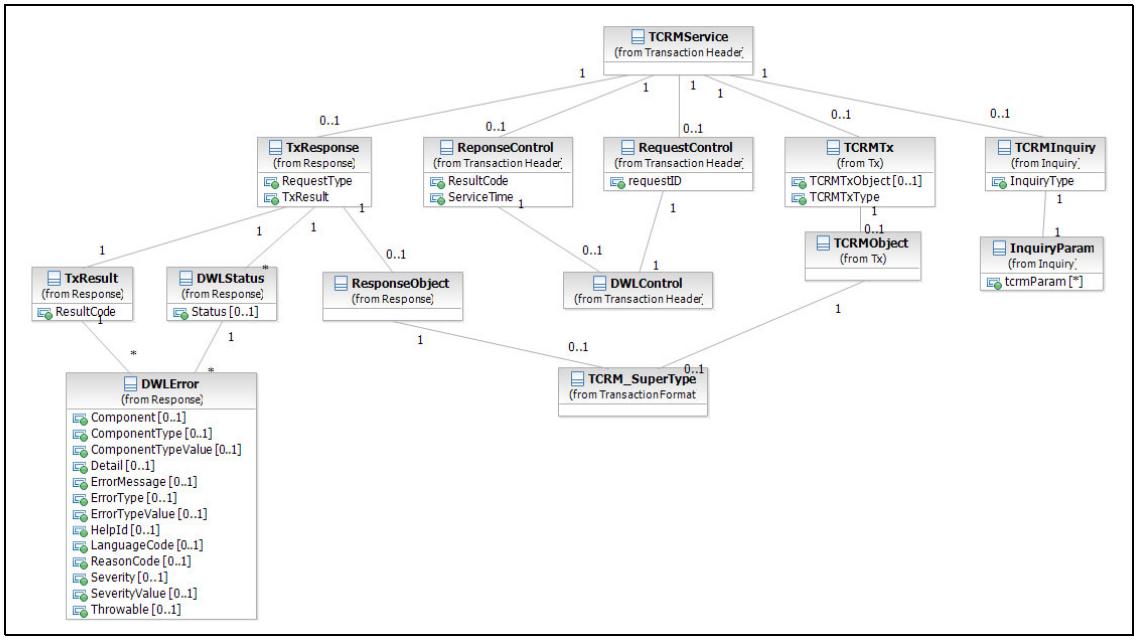

Figure 7-37 Master data management transaction format

Import guidelines

Reverse-engineering the Java code is also a means of extracting the service operations.

Use the business objects in master data management to extract the master data management object model. The services, however, are not bound to these objects. Instead, the business objects rather resemble data transfer objects than rich domain objects. The domain functions are implemented in a different set of classes called the components. The methods that are exposed through these components can be used only by internal services. Therefore, they are published through EJB controllers and web services interfaces.

If you want to extract service signatures from the master data management model, you have two options. You can either use those methods that are part of the components or use the external controller interface. If you use a version of InfoSphere MDM Server earlier than Version 10, do use the web services interface, because the names of the types the web service operations refer to differ from the ones you find in the Java API. If you need to import only the names of the operations, using the web services interface is an option. But then again, you could use the information stored in the meta configuration of the master data management data model.

The preferred approach is to reverse-engineer the service operations from the controller interfaces. The process is then the same as described in the second option in “Import guidelines” on page 377. Apart from what you extract from the technical artifacts, the master data management installation package contains further information about the service interface. The *IBM InfoSphere Master Data Management Transaction Reference Guide* (which comes with the product) provides more details about the parameters, usage scenarios, and constraints. So, you should not rely on the services interface only, but also refer to this guide to make sure that you use the operations correctly. Transferring the information from the guide to the actual model is even more beneficial, but those guides are only available in PDF format. Nevertheless, all of these information should be become part of the UML model.

7.3.5 Custom InfoSphere MDM Server models

This section describes the custom InfoSphere MDM Server models.

Custom master data management logical data model

The customization guidelines depend on the capabilities of InfoSphere MDM Server Workbench. However, as you can see in Figure 7-36 on page 375, which outlines the customization with InfoSphere MDM Server Workbench, the input is not the logical data model. Instead, you start with the object model. As soon as you have a need for customizations, your logical model is no longer the master information source. Instead, after you complete the customization steps, it is an end product. These steps include loading the object model and applying the required customizations. Afterward, you generate the implementation model-related artifacts. Then take the resulting physical data model and transform it to its corresponding logical data model.

Custom InfoSphere MDM Server object model

You can think of the InfoSphere MDM Server object model as a lower-level equivalent of the business domain model. Although the business domain model shows the business entities from a business perspective and for a specific client domain, the object model shows what InfoSphere MDM Server can handle from a product's perspective. This model is highly generic and based on industry preferred practices.

In complex environments, you might not find a one to one match between the domain objects and the objects that are used in InfoSphere MDM Server. InfoSphere MDM Server has a highly normalized, generic, and reusable model. If you try to bring the two models together, one domain object frequently requires mapping to more than one object on the InfoSphere MDM Server side or vice versa. In our insurance example, we have a domain object named Party that contains a birth date and attributes to store given names. If we look to the InfoSphere MDM Server object model, however, the birth date is an attribute on `TCRMPersonBObj`, while the names belong to the separate `TCRMPersonNameBObj`. In other cases, you might not even be able to find suitable equivalents for a domain object, because the data was never meant to be stored in InfoSphere MDM Server by default. Our Client-related attribute `BMI` for body-mass-index is such an example.

The following steps lead to customization:

1. Obtain the standard InfoSphere MDM Server object model.
2. Map the objects from the domain model to the InfoSphere MDM Server object model.
3. Introduce customizations to the InfoSphere MDM Server object model.

InfoSphere MDM Server comes with a model that serves as the basis for any mapping activities in any master data management project. For more information, see 7.2.6, "Guidelines" on page 301.

After the InfoSphere MDM Server object model is available within IBM Rational Software Architect, you can create the data mapping. In an attempt to map all domain attributes to any of the existing attributes, you occasionally encounter an attribute that you cannot map or where the difference in semantic meaning would be too large. To resolve this problem, you customize the data model.

The master data management object model is a design master that you can customize by adding or removing attributes or entities. You cannot use existing classes from the InfoSphere MDM Server object model during customization. Structurally, these classes are off-limits because they belong to the core product. When adding an element, you must determine whether it should be an extension or addition. This task is resolved on the implementation model level.

If you are certain that objects from a specific domain are irrelevant to our current context and only clutter the model, you can selectively delete them and reduce the model to only those components you need. There are arguments for and against this step: A slim-lined model leads to the reduction of objects, while having a larger perspective of all the available domains within InfoSphere MDM Server means that you should keep all the objects for future usage. After you make the customizations, it is more appropriate to refer to this model as the InfoSphere MDM Server *custom* object model.

When you introduce a new element to the model, keep in mind the following considerations:

- Extensions versus additions
- Naming conventions
- Attribute types and ranges
- Associations
- Enumerations and code tables

The first decision we must make during customization is whether the data is presented as an addition or an extension. This decision should be left to product specialists, because it influences the semantic possibilities of the stored data. From the design perspective, you must ensure distinction and relationships.

The distinction between both forms of customization is made by two different stereotypes. Each extension is tagged `<<mdmExtension>>`, while the additions are tagged `<<mdmAddition>>`. Keeping the nature of extensions in mind, you must specify which entity you extend using a generalization relationship from the extension element to the InfoSphere MDM Server object. However, additions introduce a new relationship to the model together with a new entity, as shown in Figure 7-38.

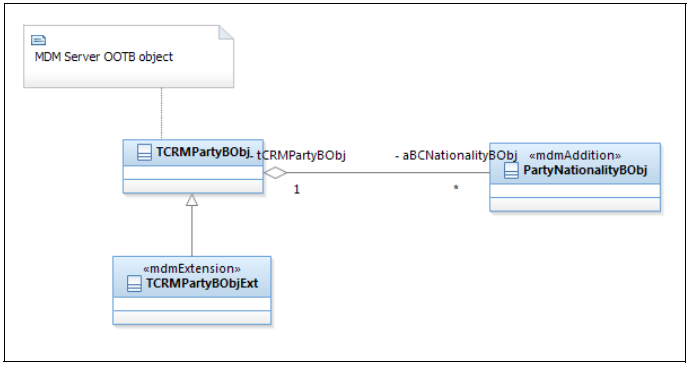

Figure 7-38 Relationship between the master data management object, extension, and addition

An important goal during customization is to adhere to the InfoSphere MDM Server guidelines regarding naming. We refer to each object in the model as a *business object*. The exact naming convention depends on the type of element. If you want to introduce the Claim as an addition, the new class should be named `ClaimBObj`. Alternatively, if you want to introduce the body-mass-index as an extension to the `TCRMPersonBObj`, the class should be called `TCRMPersonBObjExt`, which means that the object is extended; the suffix Ext indicates that the object is an extension. Similar rules apply when determining the attribute names. For a full set of rules, see the *InfoSphere MDM Server Developers Guide* (included with the product).

Contrary to the business domain model, the design model does not use arbitrary primitive types, but instead relies on standard primitives such as integer, time stamp, or string. You must ensure that all downstream transformations can deal with these primitives and can generate adequate definitions for the respective output. One such output is the DDLs for the physical data model. An important aspect in this context is dealing with length, which is mainly a relevant factor for character data. You have different options for capturing this information:

- Constraint
- “Standardized” primitive
- Properties

Constraints are available on several UML objects, but not on the attribute level. However, you can use the constraint on the containing class to ensure that the length for you attribute should not exceed a certain number of characters. The downside of using the constraint approach is that the information is spread out and not easy to read. As an alternative, you can introduce a primitive type for a specific length, for example, “StringLength50”. You can then ensure that the transformations are able to handle this primitive. Keep in mind though that this concept can blow out of proportion fairly quick, if we have several different lengths within the domain model. It should therefore be used with care.

You can also use UML profiles, which is perhaps the most intuitive modeling approach. Within your custom profile, you can bind an additional property ‘length’ to you primitive type and use it to enter the missing information about the length. These considerations deal only with the question of making a modeling container available. There are further considerations to take into account during an actual implementation, such as the different character sets that influence length restrictions.

Associations are synonymous with the relationships that we find between two objects. Although they are simple in nature, you must remember one general principle: You must not modify objects from the InfoSphere MDM Server object model. If you establish an n:m relationship between an addition and a standard object, you must transform the associations into physical attributes. This action contradicts the above principle, and you must modify the original objects after all. To work around this problem, you must introduce an extension. Apart from this requirement, the associations are rather simple from the model perspective. You provide cardinalities and appropriate role names; to adhere to the guidelines for master data management customization, the latter should be derived from the corresponding object names. For example, a relationship with a `TCRMPersonBObj` object should be called `tCRMPersonBObj`.

Enumerations also require special attention. They are represented by code tables within InfoSphere MDM Server. Their purpose in the overall context varies. Some enumerations are used to represent business attributes, such as a client's VIP status, while others are used on a more technical level, to distinguish between different types of information stored within the same logical structure. If you can map the business domain model requirements to existing structures, there are no additional modeling tasks.

In our insurance example, both the Client's VIP status, and the PropertyPolicy's type information, are examples of information you can map to the standard model. In other cases, we might need to introduce new code tables. We perform a number of steps, starting with the definition of a new class in the object model. In addition, we place the `<<mdmCodeTable>>` stereotype on the respective element. We establish the n:1 relationship between the class that represent the code table and the class it describes.

InfoSphere MDM Server custom object model

The mapping of the business domain model to the InfoSphere MDM Server custom object model also holds true for all the services you design. Just like you had to map the business domain model to existing structures in the InfoSphere MDM Server custom object model, you must decompose every service into the services offered by InfoSphere MDM Server. We distinguish between several different levels of services:

- Data services
- Business entity services
- Business task services

Data services are the lowest level of service-oriented interaction with the product. These services are described in the *IBM InfoSphere MDM Server Transaction Reference Guide* (provided with the product). After the initial importation of these services, they become part of your IBM Rational Software Architect model. The following groups of services, all of which are relevant to downstream modeling activities, are available:

- Create, read, update, and delete services
- getBy services and getAllBy operations
- Search services

Except for a few exceptions, we expect each of our objects from the object model to come with at least four services, which are get, add, update, and delete. All of them are part of the group of services called the data services, as they operate directly on and reflect physical data structures. In addition to the create, read, update, and delete services, we frequently find getBy or getAllBy services. They fall into a similar category as the get service, but differ in purpose. Using the getBy service, you can retrieve objects from the model using other associated objects as a point of reference. If we assume that both Claims and Policies from our insurance example are objects in the model, getAllClaimsByPolicy is such a service. Search services also represent a special lookup against the data. As opposed to the get services that depend on technical or business keys, searches are typically based on attributes and combinations that do not represent a key. The search for a person based on first and family name is one example for a search that does not take any key.

From the modeling perspective, the InfoSphere MDM Server object model has a number of classes, each equipped with a set of operations. Each of these operations represents one of the available services within InfoSphere MDM Server. With these operations available, you can begin designing your service design and start building custom composite services. These new services map against the service operations contained in the InfoSphere MDM Server service model. The following scenarios require customization of this model:

- Custom objects introduced to the InfoSphere MDM Server object model
- Missing operations

Every time you introduce a new object, you also need to provide the necessary create, read, update, and delete data services to interact with the object or the `get` or `getBy` type of services. Whether you require a search service depends on the business context and whether this object should be part of the overall search functionality. In addition to new objects, you might require new operations simply because your design lacks certain features. Performance or search requirements often call for additional services, and are designed to meet specific functional or non-functional requirements.

Similar to the InfoSphere MDM Server object model, the InfoSphere MDM Server service model is also a master model. This model is not a master model for static data structures, but for the service operations that manipulate them. Customization is typically performed through InfoSphere MDM Server Workbench. It ensures that the appropriate service operations are added whenever you introduce a new object into the object model.

7.4 Implementation models

The main challenge of the implementation model is to map the concepts from the analysis and the design phase to InfoSphere MDM Server. While it is true that you already established the initial relationship between these two worlds in the design phase, you now need to ensure that the solution you designed is feasible and workable. Implementation models encompass two different categories of models: the InfoSphere MDM Server product-specific models, and the models that are provided through the source systems, namely the physical data models.

The implementation models differ widely because they are either UML models, data models, or Eclipse models. We focus on the first two model types and describe Eclipse-based models later. We do not describe the target physical data models in greater detail here, because of their similarity to the source physical data models. For both types of systems, up- and downstream, we need to describe only the one type. Based on that information, you are able to derive the target-specific information in the same way and map your design models to the InfoSphere MDM Server environment.

Figure 7-39 shows the models and how they are related to each other.

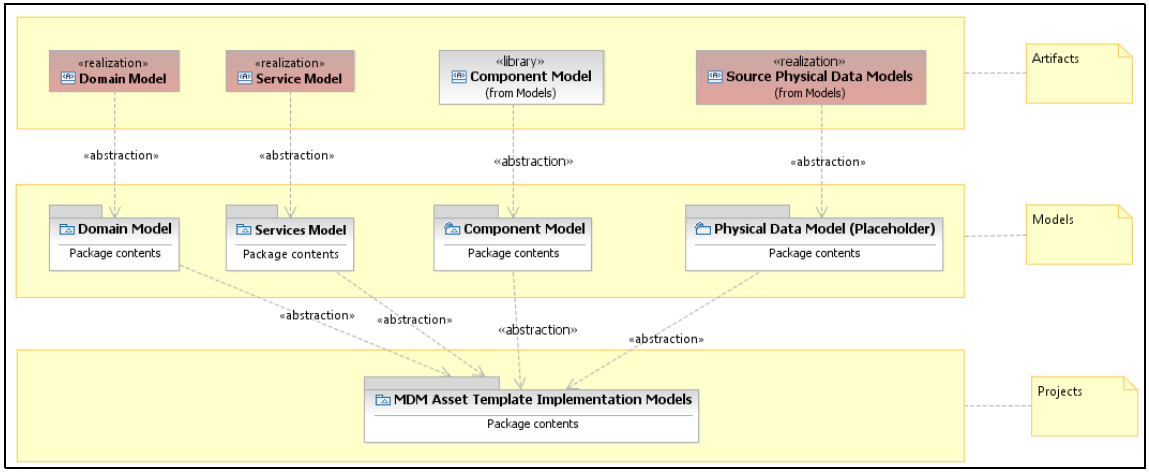

Figure 7-39 Artifacts, models, and projects in implementation modeling

The domain model on this lowest level of the modeling chain is based on the InfoSphere MDM Server custom object model. For that reason, we use the terms *domain model* and InfoSphere MDM Server *custom object model* synonymously. It is the ultimate object model that abstracts the product and its customizations in such a way that it allows us to store all information that is contained in a domain model as described on the business analysis, and therefore the domain model on the design level too.

Composite service design represents one of the core disciplines associated with the product. Ensure that you create new business-oriented services, business tasks, and business entities, and map them to underlying standard data services, including additional data services that are related to the extensions and additions you might need to build.

The component model requires special attention, especially if you use a product such as InfoSphere MDM Server. Many features, functions, and models that you require are provided by this product. For that reason, it is important to understand the product's components and to make them fit into the overall component model that describes the entire master data management system that you build. We also rely on this model to define placement of the features and model artifacts that we create.

The physical data models for both InfoSphere MDM Server and the source and target systems have a diminished relevance, especially in relation to the domain and service model. We do, however, require them to describe the data mappings that are the foundation for our description about the flow of information, as shown in Figure 7-40 and Figure 7-41.

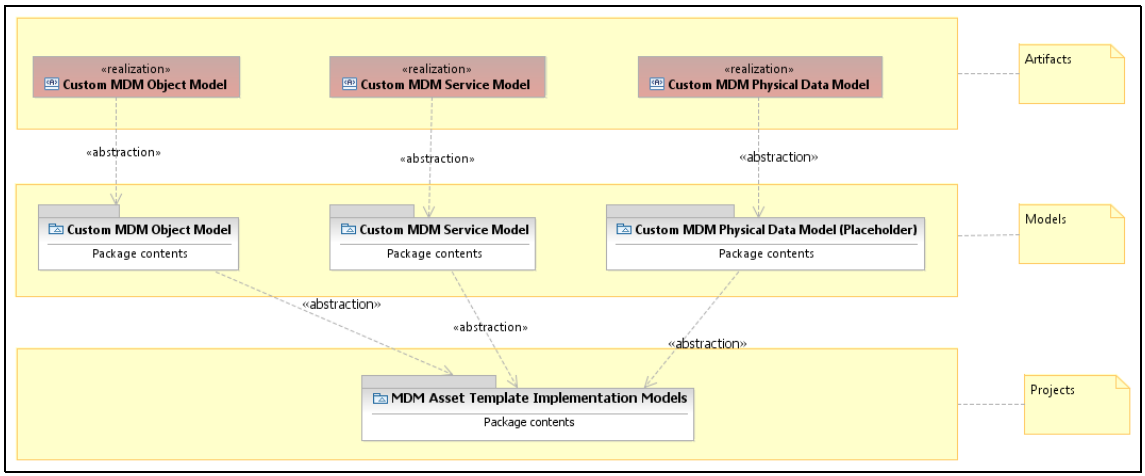

Figure 7-40 Artifacts, models, and projects for custom MDM Server resources in implementation modeling

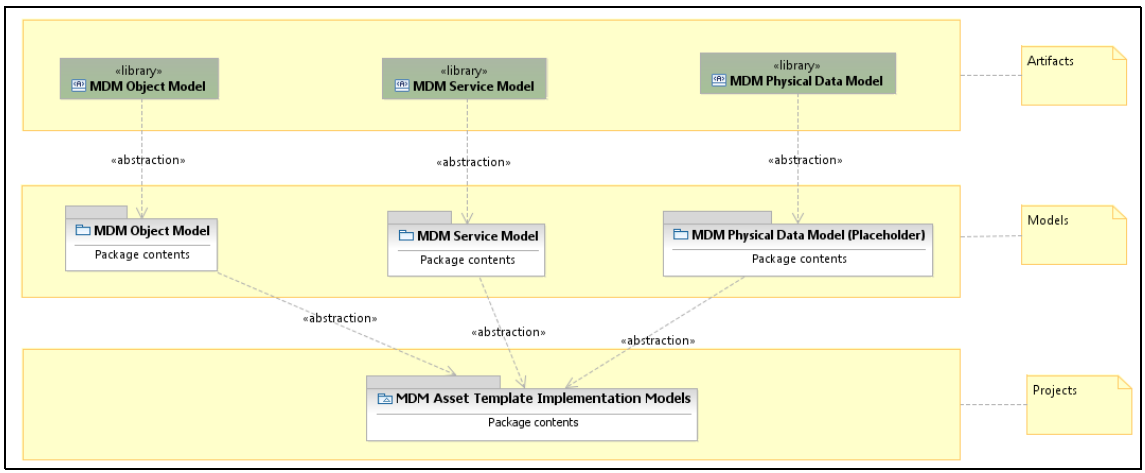

Figure 7-41 Artifacts, models, and projects for InfoSphere MDM Server resources in implementation modeling

7.4.1 InfoSphere MDM Server Workbench

This section describes InfoSphere MDM Server Workbench, which can help you to customize models with InfoSphere MDM Server. You can begin with the models that are included with InfoSphere MDM Server and use InfoSphere MDM Server Workbench to customize these models to fit your needs during the project. Again, remember to consider how you apply the required changes.

We start with a diagram with the same perspective of InfoSphere MDM Server Workbench and its relationships to some of the adjacent modeling artifacts, as shown in Figure 7-42.

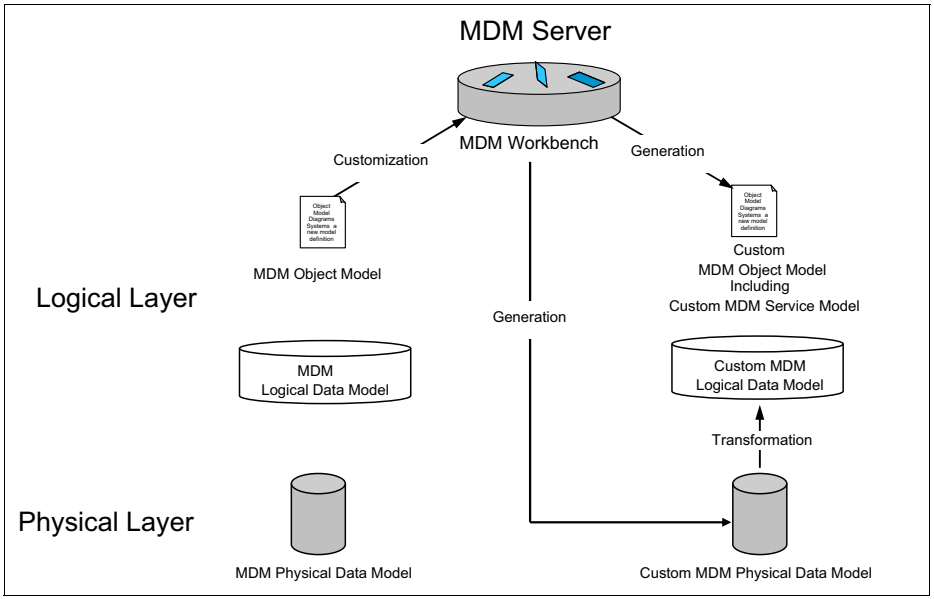

Figure 7-42 Customizing the logical data model as a by-product from InfoSphere MDM Server Workbench

Additional information: For more information about customizing models with InfoSphere MDM Server Workbench, see “Related publications” on page 559.

It is important for you to understand that the InfoSphere MDM Server Workbench relies on the Hub Module model. This additional transient model represents the core of the tree view based modeler UI provided by the InfoSphere MDM Server Workbench. The Hub Module model describes the extensions to the data model and transactions provided by the InfoSphere MDM Server. This description is kept in an Ecore metamodel, which in conjunction with the GenModel is used to generate deployable artifacts. The Ecore model is represented by the `mdmxmi` file, which can be considered the visual representation of the addition to the data model.

To a certain abstraction level, the Hub Module model can be regarded as a merging of a custom InfoSphere MDM Server object model and parts of the master data management composite service design. The generated business objects represent the custom master data management object model, and the generated composite service skeletons provide the basis for implementing the service logic, as defined in the master data management composite service design.

Table 7-1 shows the analogies.

Table 7-1 Analogies

Metamodel	**Hub Module model**
InfoSphere MDM Server custom service model and service model	Transaction or inquiry
InfoSphere MDM Server custom object model	From Addition or Transient DataObject generated Artifacts: Business objects
InfoSphere MDM Server custom physical data model	Addition/Extension/Code table (DDLs and SQL scripts)

The generated artifacts are implementation models. We describe this tool here, because we also use the object model on the design models level and must keep the logical data model in synchronization with its physical counterpart. To some extent, InfoSphere MDM Server Workbench combines the design and implementation model.

The Hub Module model represents an intermediate step, where there is a transformation from the platform independent model to the platform-specific model. Because the Hub Module model must be defined manually, there is break or gap in the seamless top down transformation from the models to the physical servers. Here the developer needs to pick up the platform independent models and bring them into the Hub Module model manually. There is no automated update in case of any changes of the platform independent models, and especially not a round-trip update. For this reason, do not use InfoSphere MDM Server Workbench exclusively for modeling.

Additional information: For more information about InfoSphere MDM Server Workbench as a transformation tool, see 10.4, “Tools selection” on page 533.

These artifacts are generated by InfoSphere MDM Server Workbench:

- XSD and DTD.
- DDL history triggers a SQL script.
- Java code (business objects, EJB session beans for additions only, and services).
- PureQuery and SQL queries.
- Web services.
- EJB deployment descriptors.
- Configuration files.

InfoSphere MDM Server provides three distinct models that are related to design. Unfortunately, neither of the two models is currently available as an explicit model file in the master data management product package. For this reason, we split our description about these models into two areas: how we obtain these models, what are the guidelines about their customization. We refer to the static models the product provides as *assets* while the customized variants are the transformed or custom models.

The following InfoSphere MDM Server design models are available:

- InfoSphere MDM Server custom object model and domain model
- InfoSphere MDM Server custom service model and service model
- InfoSphere MDM Server physical data model

InfoSphere MDM Server custom object and domain model

The domain model remains the center of the overall model. However, in the implementation model, we focus on the master data management customizations. We also must acknowledge the existence of predefined models and artifacts that come with InfoSphere MDM Server. As a result, we not only must deal with adding further details to the domain model, but how we implement the specification we established for the custom master data management object model. Nonetheless, the domain model we created in the design phase is not complete yet. There are a few concepts that are independent of master data management and require implementation knowledge and cannot be modeled in the design phase yet.

Implementation concepts

Before we go deeper into the InfoSphere MDM Server related details, we start with two examples for implementation concepts. Although these concepts do not seem to be related to InfoSphere MDM Server at first sight, they cannot be applied without InfoSphere MDM Server expertise.

The first concept we describe here is related to time, which is complex domain concepts. The main question of time slices is how to map these slices from the domain model to an implementation. We have several options at our disposal and need to decide which one to use:

- Lists

  We already changed the multiplicity of the `<<timeBased>>` attributes to * in the design model. So, now our task is to decide how to deal with this multiplicity in the implementation. We can simply transform those attributes that were originally been designed as time based to lists, more specifically to Vectors. This is a pattern that is fully supported by InfoSphere MDM Server, so it makes complete sense to use this approach.

- Predecessor / Successor with time invariant keys

  Instead of defining lists of time slices and associating them through a common key, we could also define a predecessor-successor relationship between time slices. Although this approach seems appealing at first, it comes with some drawbacks. It is a concept that cannot be mapped easily to a relational data store.

- Temporal SQL

  Instead of defining time slices on a Java implementation level, we could also use the time-dependent features available in IBM DB2 10. Although it is tempting to use these capabilities, they can support us only in a complete time slice implementation. The reason is that the business logic is involved in such concept cannot be provided by a generic integration of temporal capabilities into a database. If we are to adopt such a feature, we must add business-related logic on a higher layer. If InfoSphere MDM Server does not natively support DB2 temporal concepts, use the concepts of lists.

The second concept we describe here is the separation of utility functions. In 7.3, "Design models" on page 334, we defined what functions the utility library contains and where these functions should be stored, but it is the responsibility of the implementation model to map these functions to utility methods in the respective implementation language. Before we refer to libraries outside the scope of InfoSphere MDM Server, we must look at the libraries available in InfoSphere MDM Server. In InfoSphere MDM Server, we find three types of libraries: master data management libraries, other IBM libraries, and third-party libraries.

Make sure that you use master data management functionality wherever available before you think about including external libraries. In some cases, the correct functioning of the application depends on the selection of the correct library, so using master data management libraries is not just a matter of conceptional integrity.

Mapping to the InfoSphere MDM Server implementation

Business objects represent domain objects or parts of them in InfoSphere MDM Server. They contain a reference to the actual persistence object, the *entity object* (EObj). The entity object eventually refers to the actual persistence object, which is based on the persistence framework PureQuery.

InfoSphere MDM Server already comes with an extensive set of business objects. They are categorized into different domains, the most important of which are *Party*, *Contract*, *Finance*, and *Product*. Apart from these domains, InfoSphere MDM Server also provides administration services, domain-independent business services, and technical services, and the respective business objects. However, if you find that none of the master data management objects available really fulfil the requirements, you can also define extensions for existing objects or entirely new objects that are additions.

Operation signatures and naming conventions

The business objects expose most of their attributes as *strings*. The underlying business objects, however, uses strong-typed attributes. The types the business objects use are standardized to a certain degree. For example, primary keys are of type long and date times correspond to time stamp values.

1:n associations are exposed as Vectors. If the association does not contain any elements, the `getItems` method on a business object returns an empty vector. As mentioned previously, use this pattern consistency for the sake of conceptional integrity. Just as the domain objects encapsulate the business objects, the business objects hide the details of the entity objects. Do not directly operate on the entity objects in your domain object. This method is another preferred practice for which the implementation model is responsible.

The business objects do not expose create, read, update, and delete services themselves. Nevertheless, conceptually speaking, these services belong to the business objects. The naming conventions for the create, read, update, and delete methods are rather simple. The respective operations start with add, update, get, delete, or `get<ObjectName>BObjBy`. In the latter case, the key word `By` is followed by a business key. Regarding validations which, the naming conventions are easy to remember, too. The respective methods are implemented in the business objects and are called `validateAdd`, `validateUpdate`, and `validateDelete`.

The package names correspond to the ones defined in the design model. During code generation, these packages are transformed to those Java package names used by InfoSphere MDM Server.

Implementation considerations

Having described the basics of the master data management object model, we are able to identify the most important questions regarding the mapping between domain model and custom master data management model:

- Can you use existing master data management entities or do you need to define extensions or additions?
- Do the domain objects map to one or more business objects?
- Do the business objects represent domain objects, value types, or attributes?
- If the type is not a predefined one, what type do we need to use?
- What should be the maximum length of an attribute?
- Is an attribute nullable?
- Does an attribute have a default value?

Although we need to answer some of these questions in the design phase, it is only in the implementation phase where we can provide a final specification of the mapping between the domain model and the master data management object model. It is in this phase where we must decide how to implement the objects and associations we established in the design model.

Generally, we should define the implementation model as strict as possible. We could define each association on both sides and allow m:n relations everywhere in the model. However, this action defeats the purpose of those definitions that represent integrity constraints on the lowest level, that is, the database level. Instead, we must establish the navigation direction and the cardinality of a relationship in both the custom master data management model and the domain model in the implementation model.

We also must decide whether we can use a standard master data management object or whether we need to implement an extension or an addition. Laying out the mapping and identifying the need for extensions or additions is probably one of the most complex tasks in the implementation model. Only in a few cases can you match domain objects with all their attributes and master data management objects directly. The reason is in the semantic difference of both the InfoSphere MDM Server model and domain model.

The business concept in InfoSphere MDM Server is generic so that it can work with many different use cases and across multiple industries. The domain model we define during business analysis is specific and dedicated to our business scenario only. An example for this type of deviation is the introduction of distinct subtypes of organizations in the business domain model. In InfoSphere MDM Server, all of these entities correspond to the `TCRMOrganizationBObj` type and are classified through a type code. You can emulate subtypes by assigning different type codes to your organizations. However, you might then encounter the next challenge: Some of the organizations can require additional attributes, while others might even require a different set of attributes. Although you can create an entity extension in the former case, the latter case demands an entirely new entity.

The definition of associations poses another challenge to us. In the example of organizational hierarchies, you could decide to create party relationships between organizations, but you could also be required to model a hierarchy. Hierarchies are part of the master data management object model and can be used for this purpose. Apart from hierarchies, the master data management object model contains several other cross-domain objects. Thus, another task in the implementation model is to decide which of these concepts should be used.

Applying this situation to our concept of time slices, we also must take InfoSphere MDM Server specific features into consideration. Assume that we decided to use the list-based approach previously described. This decision can imply that we need to define an addition just because of the time-dependency of an attribute. For example, if a party attribute is time-sliced there is no means of resolving this dependency other than defining an addition that holds the party ID as a foreign key. You can now see how decisions we take on a higher level might have an impact on the master data management implementation level. At all times, you must be aware of this dependency.

InfoSphere MDM Server service model and custom service model

Another important deliverable is the *service model*. The idea behind the implementation is similar to what we described for the domain model. We need to determine which services are related to InfoSphere MDM Server and represent data services and whether we can use any of the available services or need to implement a new custom service. Especially the latter task resembles an activity we know from data mappings. In fact, we consider this aspect service mapping. You find some challenges in respect to this situation outlined in the chapter about mappings. Another similarity between the domain model and the service model is that there are only few implementation-specific specifications that are not related to master data management. One of these few examples are technical exceptions which need to be added to the implementation model regardless of the underlying master data management product.

Data services belong to the master data management-specific model. The main reason for this decision is that data services refer to data objects which again correspond to the business objects. The data services are one of the assets that are available in InfoSphere MDM Server. To find an existing master data management service and its specification refer to the master data management transaction reference guide.

It is important for you to realize that the master data management service model offers several interfaces. Even if we know that the data services are part of the master data management logic we still need to decide where these services are implemented and which interface they use and expose.

Look at the list of those interfaces master data management supports:

- XML over Internet Inter-ORB Protocol (IIOP)

  This interface is the classical remote interface of InfoSphere MDM Server. Services are exposed through their names, some meta information like the application they are part of and the type of transaction interface they use and the XML schema of the parameters. A caller takes this information, creates an XML string that contains all the parameter values and calls an EJB method of the InfoSphere MDM Server controller.

- Web services

  This interface is a standard SOAP service interface that uses its own request and response objects. While the old XML schema and the schema the Web Services Description Language (WSDL) uses are not identical in version 9 of InfoSphere MDM Server they are related to each other in the latest version 10 of InfoSphere MDM Server.

- Transaction interface

  Transactions are exposed through the Java API, too. So, a Java class can call an InfoSphere MDM Server service by using a transaction base class and passing the transaction name and the transaction object to this class.

- Component interface

  You can also use the component class of a business object. This component comprises all the methods of the associated business object or a group of associated business objects. As opposed to the other interfaces we mentioned, using the component means to instantiate the component and call the method on the component directly.

As the list of interfaces suggests you should use one of the external interface protocols such as XML over IIOP or SOAP, if your service is exposed to the outside worlds. For new applications, it is probably a good idea to use the web services interface as an external interface. If your service is for internal use only then you can use the transaction interface. In our setup you can declare most data services internal services, because we use them to expose our business services. Using this interface means that you call a service of any domain in the same Java application. Only in those cases where a service that belongs to a specific entity calls another operation within the same scope this service can call the component directly.

Data services belong to business objects and are implemented in the respective component classes. A create, read, update, and delete service or similar service that belongs to the Party business object with the Java class name `TCRMPartyBObj` is part of the PartyComponent. The same logic applies to those services that are not part of the standard master data management package.

Regardless of whether you implement an addition or an entity extension you must always ensure that your component contains only data-related services.

The extension attributes are automatically handled by the base component. Remember, that everything that is beyond the scope of one entity should not be implemented within the component. Apart from those create, read, update, and delete services that are generated automatically by the master data management tools, a component should contain only simple getBy services.

InfoSphere MDM Server includes a get services solution. Of course, we could implement a standard transaction and pass the business objects into the service signature, take those attributes that are filled and use them for the lookup, but master data management server uses inquiry services for this purpose. In greater detail, InfoSphere MDM Server distinguishes between persistent transactions and inquiry transactions the latter of which take a list of parameters to perform a lookup with the aid of these parameters. Hence, another responsibility of the implementation model to transform the get services into the inquiry format supported by InfoSphere MDM Server.

Coming from the data service we now concentrate on business entity and business task services. There are two options how to implement these services. At any time, you can create Java classes outside the scope of InfoSphere MDM Server. How exactly you implement these classes depends on your overall architecture. You can expose these classes as web services, Representational State Transfer (REST) services, Enterprise JavaBeans, and so on. However, if you decide to use the functionality available in InfoSphere MDM Server then you use so-called business proxy classes. These are simple Java classes that are derived from a base proxy class and provide the same interface as the additions and extensions. After you implement a business proxy class and configure it in the database meta repository of InfoSphere MDM Server, you can call the business proxy with the same transaction syntax as the standard services. There are some restrictions we must observe that prevent us from using some of the features available on the controller and component level of InfoSphere MDM Server.

The most obvious reason for using business proxies is that they are seamlessly embedded into the InfoSphere MDM Server infrastructure. If your service orchestrates a number of master data management business objects to provide functionality for a business entity, we suggest using business proxies because this scenario exactly corresponds to what business proxies are designed for. If this lowest level of orchestration yields business entities or our domain objects, there is no immediate need for using business proxies. If we decide on using business proxies regardless we benefit from the advantages of the InfoSphere MDM Server platform, such as the provisioning of the service interfaces.

Master data management protocol

Closely related to the services is the communication protocol that the use to communicate with their consumers. InfoSphere MDM Server provides several interfaces all of which share certain characteristics. They differentiate between the actual workload and the context provided with a service. Dependent upon the type of interface you use, the workload is either transferred as XML string, as SOAP body or as typed Java object. Regarding the context information InfoSphere MDM Server requires that we distinguish between two types of context information. The first group of context information contains meta information that you need to call the correct service. In the implementation model, this information must be modeled separately from other context information. The reason for this separation is simply that while some of the interfaces need these information, others do not. For example, if you call a generic “process” method on the master data management controller you must pass the transaction you want to use to the controller. You do not need this information if you call a typed method on a component interface.

The other part of the context information a caller needs to provide, is passed as part of the workload. InfoSphere MDM Server uses a data container called *DWLControl* that contains all information relevant to the current context. In the DWLControl object, you specify authentication and authorization information, the caller's language, the caller ID, and the ID of the request. Other information such as the original transaction name are automatically saved to the context. Regardless of whether you use the web services interface or the XML interface, the control element always precedes the actual transaction call. Apart from contextual meta information the DWLControl object also serves as a context object during the lifetime of a request. While you can pass virtually any information to this object and use it as request cache we strongly suggest using this object with great care. The reason for this is that putting all untyped information into this object decreases maintainability and complicates error resolution. You should apply the same set of preferred practices to the DWLControl object as to a session context of a web application.

What makes the DWLControl object even more interesting and demanding is the fact that it also contains business-relevant information. Some of the filter objects we defined in the design phase are part of the DWLControl “header” in the implementation model. This situation applies to both the ACTIVE/ALL filter and to inquireAsOfDate-based queries. This situation imposes an additional mapping task on us in that we determine how we must map input parameters to the InfoSphere MDM Server service signature, including the header.

We suggest creating separate object mappings or comparable dynamic mapping diagrams to specify this mapping. The transaction guide reveals all information about the communication protocol. We must model it manually derived from that guide and built similar to the protocol model outlined in Figure 7-43.

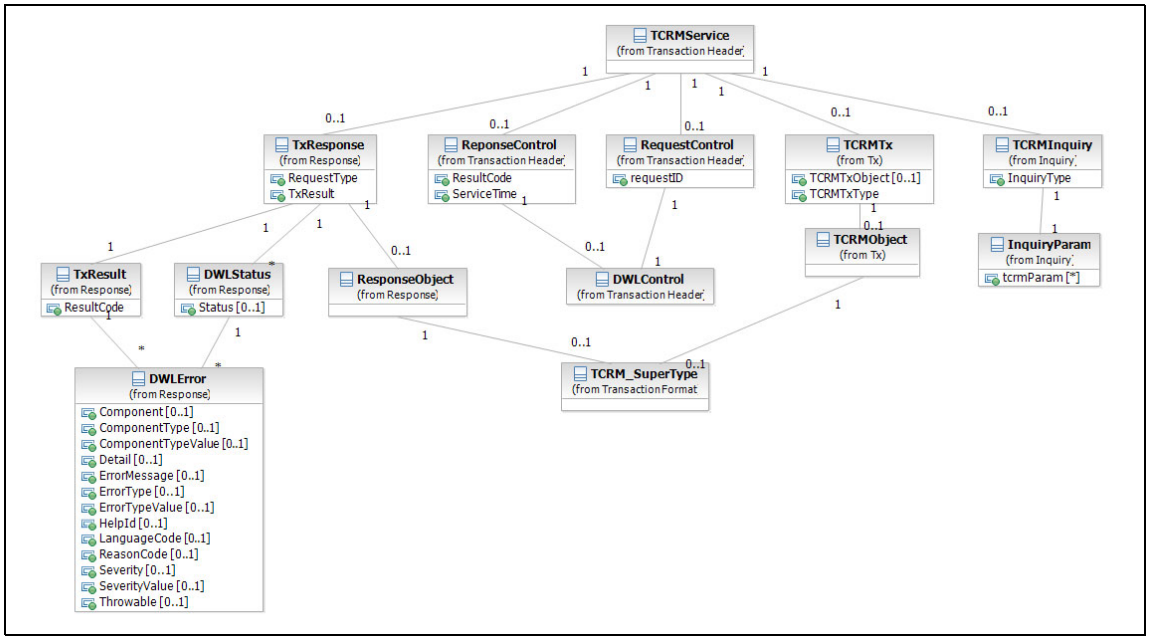

Figure 7-43 InfoSphere MDM Server transaction format

One other aspect of the protocol, we would like to mention, is the specification of exceptions. As stated, if your service is outside the master data management, you can define your method signature independent of the master data management error model. But with InfoSphere MDM Server as our base product we need to map the reason codes master data management returns to our business model. InfoSphere MDM Server reason codes refer to master data management objects and do not directly correspond to the business errors we designed.

Component model

InfoSphere MDM Server as a product can be described through its components. If you refer to the InfoSphere MDM Server developers guide you find information about the components that it is made up from. In the chapter about InfoSphere MDM Server sources, we explained how you can manually create a UML-based copy of the component model that is described there.

This situation is only one side of the story, because we also must integrate the product as one of many subsystems into the larger context of the overall master data management system. To be able to adopt the components provided by InfoSphere MDM Server, you first need a better understanding about the functions and roles that the components provide, as illustrated in Figure 7-44.

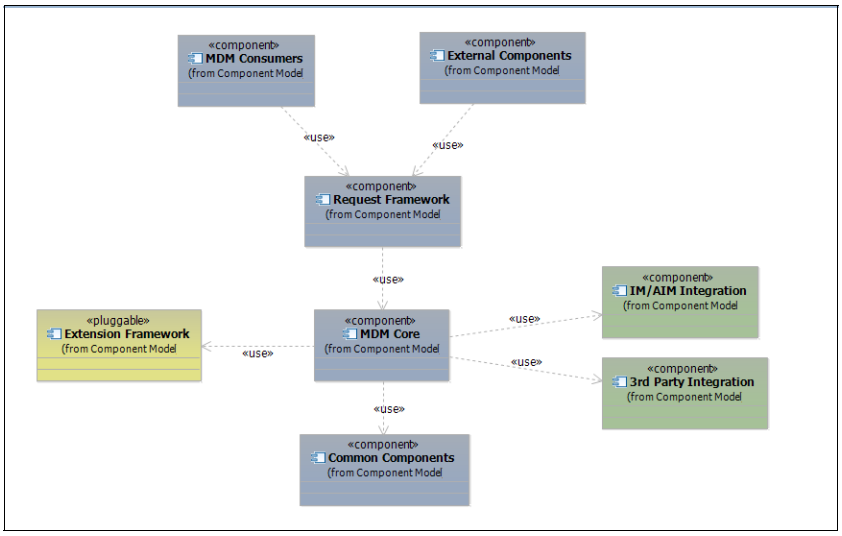

Figure 7-44 InfoSphere MDM Server component overview

One of the core components in InfoSphere MDM Server is the request (and response) framework. This framework is responsible for converting the XML requests to Java objects and the responses back to XML. An important design decision we must make about this framework is whether we need to provide a custom implementation of the request or the response parser. This decision depends on the complexity of the service interface. We need to evaluate runtime information in the XML parser, if you use many sub types in the internal implementation, but expose only the super type in the interface. In this case, we must ensure that you do not use incompatible types with your internal services.

Another common candidate for customizations is the protocol format. Clients might require a format that differs from the standard master data management XML format. If so, you might need to provide a custom implementation for the request parser. We outlined another option in the chapter about architecture which is based on the newly introduced Adaptive Service Interface (ASI) within InfoSphere MDM Server V10. This interface allows customization of the protocol format but is not meant to host large amounts of business logic.

Another important component of InfoSphere MDM Server is the extension framework. Wherever we deal with business or technical aspects that are not part of the core logic we need a mechanism that helps us separate these aspects from the actual flow logic. The extension framework provides a means of decorating standard and custom transactions. It provides hooks for pre- and post-transaction handling and that allows us to execute our custom logic in a way that is controlled by InfoSphere MDM Server. An important feature of these behavior extensions is their configurability. You can associate them with different types of triggers dependent upon whether they must be called for certain business objects, certain transactions or for every persistent transaction. These features come in handy when dealing with cross-domain functionality such as the handling of lifecycles and rules.

In the design model, the lifecycle corresponds to a state machine supported by triggered actions. InfoSphere MDM Server does not include a full-fledged state machine, so we need to map the lifecycle to those master data management components that match best our requirements. Regarding the state representation, InfoSphere MDM Server provides a status field for some of the core entities. Both, the party and the contract, can be assigned a state. Behavior extensions then come into play when we must model the transitions between states which are based on the recipient of appropriate trigger events.

We implement a lifecycle handler that triggers the state transition, if needed. This handler is also responsible to inform other components about state changes. Interested components simply register to the change events the handler publishes. Note, however, that we might need to provide an additional means of integrating the extension framework with business proxies. By default, the proxies do not participate in the extension framework.

This situation leads us to the transfer from the design level concepts that are related to domain objects to the implementation level of the mapped and corresponding custom master data management objects. One requirement is that we make the features of a master data management component available to the next higher level, the design. The other solution is to decompose the business concept and split it into parts that can be handled by InfoSphere MDM Server without any modifications. If we apply this solution to the lifecycle, we must move the entire lifecycle handling to InfoSphere MDM Server objects and transactions. While this action is entirely possible we do not recommend this approach in this scenario, because it means to move rather complex transition logic to the level of create, read, update, and delete services. To illustrate this situation, imagine a situation where we need to decide upon receiving an “UpdateParty” event, if the lifecycle status changes. In this case, the granularity of the services and the granularity of the logic acting upon them do not match.

Business rules are also implemented as behavior extension. The behavior extension, or rather a rule handler referenced by the behavior extension, is responsible for the collection of all relevant rules and firing them according to what is stated in the design model. The rule itself should be encapsulated in a separate class. This approach continues with the separation of concerns that we established on the business level already and also followed on the design level.

InfoSphere MDM Server provides support for external rules and so we can integrate the handling of our business rules with the InfoSphere MDM Server provided functionality rather easily. Note, however, that we need to decide again whether we apply the rules to InfoSphere MDM Server objects or whether we would actually prefer to apply them to the domain object. In most cases, the latter alternative is the wanted one which makes it again a little more involved to use the functionality of InfoSphere MDM Server. Dealing with rules, we face the additional challenge that some of the rules depend on the context of the method. They are only applicable to certain methods. Although InfoSphere MDM Server provides support for this scenario, the implementation of these rules requires special consideration.

Validations seem to fall into the same category as rules. However, there are some important differences between rules and validations. InfoSphere MDM Server differentiates between internal and external validations. Internal validations refer to one InfoSphere MDM Server object only and are implemented in the validateAdd and validateUpdate methods of the respective business object. Strictly speaking, external validations are those validations that are implemented with the aid of the validation framework that is part of the Common Components of InfoSphere MDM Server. This framework provides support for validations that span more than one InfoSphere MDM Server entity. They can refer to so-called soft entities which are entities represented through an XSD file. Regardless of the entity type, external validations are based on database configurations. Our task is to decide whether a validation can be implemented as an internal validation or not. If it cannot, we need to determine whether we should use the external validation framework instead. Generally, we recommend using the validateAdd and validateUpdate methods for simple validations such as length validations, check for type codes, and so on. Our decision to use the external validation framework depends on the following factors:

- Does the nature of the validation suggest the use of the external validation framework? For example, can the validation be mapped to InfoSphere MDM Server entities?
- Does the use of the external validation framework fit into the overall architecture?
- Do the project members have sufficient knowledge about InfoSphere MDM Server to use the framework?

Notifications are yet another example for behavior extensions. Imagine you need to inform other systems about changes in the master data management system. Notifications provide one possible solution to this requirement. They are provided by InfoSphere MDM Server. The principle behind notifications is a simple one. After a transaction is executed successfully, a notification will be sent out to a message queue. Systems interested in this notification can subscribe to the notification topic. Often, however, you find that it is not sufficient to send change notifications only. Dependent on the data model of the target system, it might need many more pieces of information to apply the changes to its data model. InfoSphere MDM Server can also send the entire service response to a queue. However, even the response might not contain all the information target systems require. Think of a system that tracks certain account roles. If the master data management system sends a notification about a change in one of the affected parties, the target system probably needs to query the status of all the other parties related to the one that was changed, too.

InfoSphere MDM Server functionality applies to the error and the code type component. This component is probably one of the ones you need often. We often must deal with external code types. Handling and storing them independent of the source code is a common requirement. InfoSphere MDM Server cannot possibly provide every feasible sub type of an entity and code types provide a rather easy means of simulating sub types. InfoSphere MDM Server uses “typed” code tables to save code types. Each code type is stored in a separate table.

Another feature that is closely related to the core components of master data management is the support for *business keys*. We included an extensive discussion of business keys in our design considerations. Here, we focus on how we map business keys to InfoSphere MDM Server. The answer is twofold. You can either use the technical keys InfoSphere MDM Server creates internally and associate them with your business keys with those entities that are already defined in the InfoSphere MDM Server data model, or you can explicitly define external business keys in the service signature. With the latter approach, InfoSphere MDM Server provides additional features such as checks for uniqueness.

We already mentioned inquiry transactions. InfoSphere MDM Server groups several extensions around this component. For example, you can define numeric inquiry levels and define what objects are returned for a certain inquiry level. Also, you can define child groups and associate them with inquiry levels.

The search component which is part of the Common Components is also one that we often rely on. Many business processes start of searching for information, for example, to decide whether new data must be entered or existing data modified. Make sure that you group the searches you require and check whether you can standardize them as far as possible. Often, you see that searches always use a combination of the same parameters and comparisons. If so, you might benefit from the search framework that comes with InfoSphere MDM Server. Whether you can use it or not really depends on how schematic your searches are and requires investigation. In the worst case, you must introduce custom SQL and implement it separate from the search framework.

Similar considerations apply to other components provided by InfoSphere MDM Server. The Event Manager and Duplicate Suspect Processing are two of these components. For every component, the decision process is similar. We always start looking at the design model. Next, we create the technical design before we actually try to match this design with what is available in InfoSphere MDM Server. This process helps you avoid ending up with technical solutions that look like they solve your problem, but in fact do not.

Physical data models

Of all the data models existent in the entire modeling approach, on the implementation level we only deal with the physical data model. Again, we distinguish between the InfoSphere MDM Server related physical data model and those models that describe our source and target systems. We require these models for two reasons, data mappings and transformation into corresponding objects for service integration design. In respect to the source systems, we do not require these models anymore, after we have them transformed and used in the static data mappings that we apply on the level of the physical data models.

Here we outline how you can create the required source physical data model. There are many possible ways to achieve this goal, for example, through importing an IBM InfoSphere Data Architect compatible `.dbm` file, importing DDL files, or reverse engineering directly from a database connection. In this example, we have a DB2 database that is holding the schema. We create a new physical data model and specify that we want to reverse engineer the database. By providing the database access details we are now able to import the schema of our interest into the IBM InfoSphere Data Architect and derive the data model from there.

The resulting data model could look as shown in Figure 7-45.

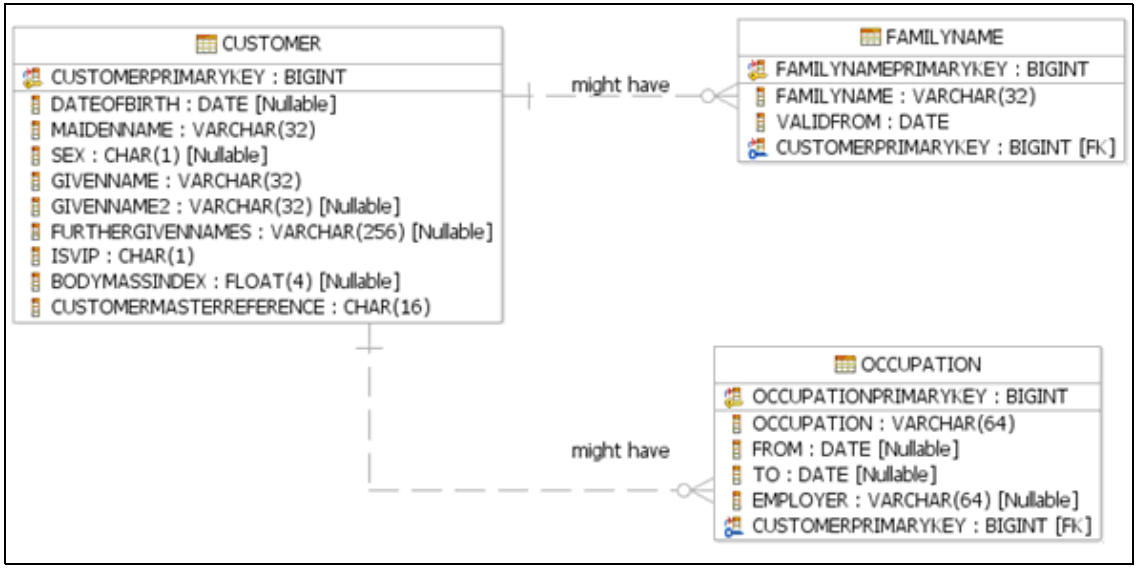

Figure 7-45 Sample physical data model - Source System A

The good thing is that there is no further customization required to these models, whether we use the Cobol import wizard, reverse engineer an existing database or create the data models from scratch. You might want to watch out for the typical information such as data type and field lengths to be available so that a data mapping against this model really makes sense.

Service interface descriptions require IBM Rational Software Architect based UML class diagrams instead of IBM InfoSphere Data Architect data models. In order for us to be able to use them in such way, we require these data models to be further transformed. You can learn more about transformations later.

The physical data model of InfoSphere MDM Server, as the central data store in our solution, requires a little more attention. The standard model is provided as part of the InfoSphere MDM Server installation. Instead of reverse engineering it from a database connection, you can import the IBM InfoSphere Data Architect compatible dbm file that comes with the product. Additions and extensions are provided as part of the transformation and generation process performed by InfoSphere MDM Server Workbench. This process transforms the UML implementation model into source code. In respect to the physical data model, this code comes as a DDL. You can read up on further details of this process in the chapter about transformations.

We also must consider other aspects that surround the physical data model of InfoSphere MDM Server, especially how other related models interact with it. This situation impacts some decisions that we must make about the data model. The first decision is the selection of the database. The selection of the database vendor has an impact on how we map our design to the implementation model. The reason for this dependency is the database dependency to default values, data types, and so on. For example, if the default value of the end date of a record is NULL or 9999-12-31 makes a huge difference in some database products because an index can be applied only to the latter default value.

Another decision is related to storing the extension attributes, which are impacted by the non-functional requirements. If the performance of the database is more important than readability and maintainability, then you probably decide to store the extension attributes within the base table. If performance is not that much of an issue you can also go with a separate table for each extension. Dependent upon the queries you plan to run against the database you can deviate from a strictly normalized model.

Regardless of whether we look at extensions, standard tables or additions, we also need to decide whether we use the history tables in InfoSphere MDM Server. These tables track each change in the operational tables. From a technological point of view, all of these history tables are based on database triggers. If, for example, a party is updated, this action triggers an update of the history table, too. You can decide whether you want to use these triggers for update operations only or for add and delete operations as well. Whether you use the history tables or not is a business decision. Even though this situation does not appear in most business analysis models, it must show in the list of functional requirements that define our system. History tables are a common means of informing other systems about changes that happened within a time frame. Alternatively, using history tables has further implications. For example, you can use InfoSphere MDM Server to define whether each subtransaction within one business transaction should obtain a separate time stamp or whether the time stamp of the sub transactions should be synchronized. Although this setting does not seem important, it has a major impact on how historic data is presented to external users and systems.

InfoSphere MDM Server uses an optimistic locking concept, which means that it does not prevent other threads from reading potentially “dirty” data while another transaction has acquired a lock for this data. You implement this concept by using a time stamp that tracks the latest update of a record, that is, the lastUpdateDate. You can write data only if your transaction knows the correct last update date. It is important to align this feature with how you designed your services and where you set the boundaries of the business transactions.

The way you map domain objects to the InfoSphere MDM Server objects can affect the performance of the InfoSphere MDM Server system. Using many small update operations has a negative impact on the overall system performance. This impact is why the implementation model is also where you apply implementation-specific optimizations, again based on the non-functional requirements you must work against. One possible optimization strategy is based on the combination of multiple subtransactions within one single business entity. This situation goes beyond the ordinary data mapping and includes service-specific skills and knowledge, which produces the most effective and best performing mapping. In addition to manually optimizing these queries, InfoSphere MDM Server provides configuration options for optimizing complex queries.

Other parts of the implementation model

After you describe the data and service models using UML, you must complete the final transformation task and generate the source code by using the UML-based implementation models. You reach a level of detail that cannot be covered fully by UML. The UML designers did not have source code in mind; only a comprehensive and valid design language. Thus, it is almost impossible to describe certain programming constructs with UML.

The models described here are not based on UML. The first model is the InfoSphere MDM Server Workbench model. This model is used for defining additions, extensions, and some elements that are specific to master data management and source code generation. We take a closer look at this model in 8.2, “Transformations” on page 437. The same situation holds true for the Eclipse Ecore model, which is the second model that is beyond the scope of UML. This model is used together with some other Eclipse-specific artifacts and defines those parts of the implementation that are not covered by the InfoSphere MDM Server Workbench model. The implementation model has two parts. The first part communicates with the UML models we used throughout this book, while the second part associates our implementation model with a code-centric view. We describe the second part in 8.2, “Transformations” on page 437 because it resembles the step of source code generation to automatically create our final deliverable.

8

Mappings and transformations

In this chapter, we describe mappings and transformations. Mappings are one of the core deliverables of any project that involves InfoSphere MDM Server because of two reasons:

- The product includes a ready use data model. This data model differs in most cases from the data models of the systems you must integrate. It is those differences that you must describe and document.
- Mappings revolve around the services, because in addition to the data model, InfoSphere MDM Server also provides a number of services that are available for immediate usage without modification. Often, these services do not exactly represent what you require for the business task you must expose through a service. It is here where you require service mapping.

Transformations are one of the most important and least visible elements in the modeling process. They should be considered primary elements in a model-driven methodology. Transformations are applied to different levels of the model, while mappings typically apply to models on the same level. Although mappings describe the flow of the data, transformations modify the models that operate on that data. Transformations do not statically map two models, but provide additional features, such as the ability to model constraints, bridge the semantic gap between two model levels, and provide a model-to-text conversion.

Thus, transformations are non-static by nature. They analyze model elements dynamically and create model elements using the logic that describes them.

8.1 Mappings

You do not need an InfoSphere MDM Server to decide if you need data or service mappings. There are many information-related projects that need data mappings, such as when you design a data warehouse and must load the data into it. Even then, you typically discover a new database schema that describes the data warehouse, but is not equal to the data model of the source systems from which the data is loaded. You do not necessarily need service mappings, because you usually load the data warehouse through a direct database load.

This section focuses on data mapping. We also describe data mapping in a service context. We keep the description of this subtopic to a minimum, because we do not want to divert your attention from the more imminent issue of data mapping tasks. Also, we take a closer look at service-related data mapping in 8.1.3, “Service-related data mapping” on page 426, which also involve service design. For these reasons, we focus predominantly on data mappings in this section and only remotely touch on a few challenges related to service mapping here.

At this high level, we introduce some of the complexities of data mappings. This introduction provides a foundation for further descriptions in this chapter.

Figure 8-1 shows the data models available through our case study.

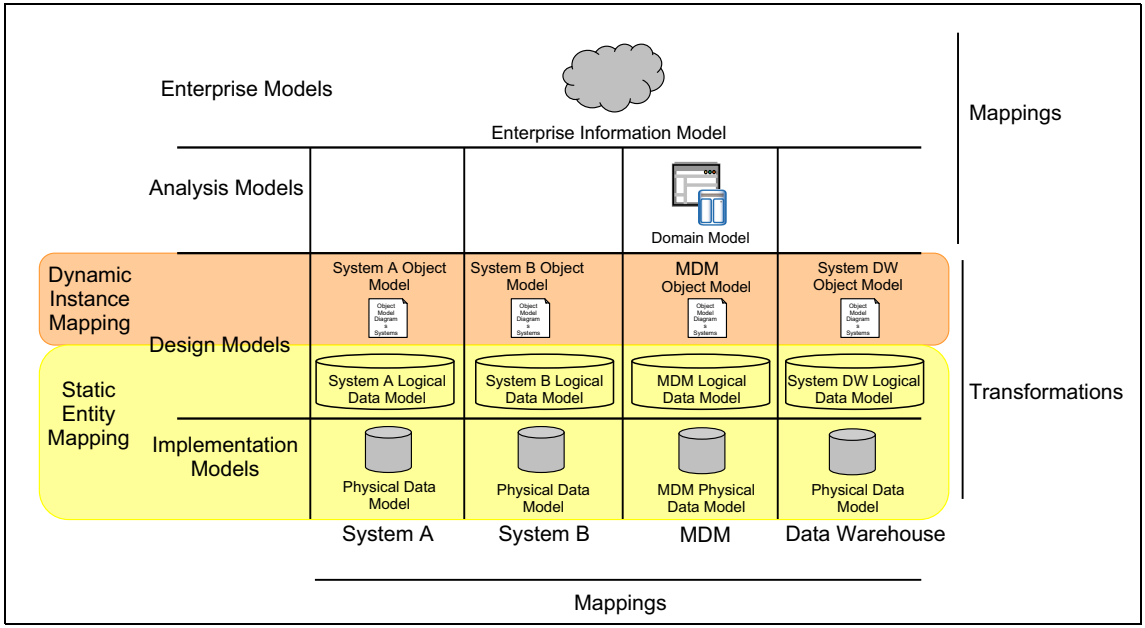

Figure 8-1 Mappings against the systems for our case study

There are a number of different elements described in this diagram. The most obvious element is the representation of the systems A, B, MDM, and Data Warehouse. Another key dimension includes the four types of modeling levels, enterprise, analysis, design, and implementation, through which we describe the entire system behavior.

In the cells, you find the type of data model that describes the system on the corresponding modeling level. For example, the master data management physical data model is the implementation level data model for the master data management system. We assume that the domain model is described only for the master data management system. Other systems can store additional or entirely different domains. These systems are not relevant to our master data management system implementation, which is why we exclude them from our data model view.

In addition, there is a single conceptual enterprise information model. It describes all entities across all systems. It is important for you to be able to decide on the correct course of action when you discover that a particular data model is not available. For example, a source system might be available without its logical data model. The question now is whether we need it to achieve our goal, and the answer depends, for example, on whether we must have dynamic instance mapping and if we establish that mapping on the object model level.

Based on this representation of all available data models, we can distinguish between two core aspects:

- Transformations
- Mappings

Although you learn more about transformations in this chapter, we want to roughly outline them here. It helps if you differentiate and define mappings more clearly and precisely. The difficulty in distinguishing between a transformation and a mapping is that both are mappings. For example, the master data management physical data model is largely identical to the master data management logical data model. Even though these models are two distinct data models that are not realized through transformation, the InfoSphere MDM Server modeling philosophy does not leave you with another choice. InfoSphere MDM Server Workbench generates only the master data management object and physical data models. If you want to maintain an updated master data management logical data model, you must create it either manually or by using automated transformations.

In this chapter, we chose the latter approach, because the physical data model can be transformed easily into the corresponding master data management logical data model. Even if the transformation is a simple one to one transformation, what we describe in the transformation is which attribute of the master data management physical data model maps to which counterpart of the master data management logical data model. However, in respect to transformations, we achieve the mapping by using transformation definitions and obtaining traceability information at the same time.

Based on that approach, we argue that transformations are vertical mappings. They are not the mappings we refer to here. Exceptions include the domain model and the enterprise information model, both of which are on a higher modeling level and are not transformed, but are mapped. A different data model view could place both data models in a different location on the horizontal axis as virtual systems so that there is no vertical mapping, but horizontal mapping only. In this case, the transformations are vertical.

The data mappings in our sample case study mappings are horizontal. What is significant is that you must perform data mappings between the same model types on the same level but across different systems. For example, in data warehousing, you might define transformation rules between data models on the same level, but these rules are derived from appropriate data mappings. It is these data mappings that eventually define the transformation behavior in your ETL job. How do you know which of the data models must be mapped? We diminish the importance of the logical data models in that we do not believe that they are required in data mappings.

Static entity mapping can be performed by IBM InfoSphere Data Architect. Unfortunately, this product is limited to this type of data mapping. However, it has the advantage that it can quickly and easily provide you with a simple view of the mappings that take place. This situation is why this mapping can be used for initial discussions, prototypes, and other simplified scenarios. Another scenario is when the dynamic aspects are not relevant in data modeling. Based on our example, we want to map data from each source system's physical data model to the master data management system physical data model. We must document the physical data model outbound mapping. Although this mapping could be relevant to all data models, we foresee its usage predominantly on the physical data model level. The yellow colored section in Figure 8-1 on page 413 defines the modeling level in which we would apply static entity mapping.

Dynamic instance mapping represents a bigger challenge in data mapping, as you need more sophisticated tools to perform this mapping. We introduce these tools in 8.1.5, "Tools" on page 430. The disadvantage of these tools is the complexity that is required in describing the required instantiation of entities. When you describe data mappings to this level of detail, you have everything that you need to design and eventually carry out a transformation. We assume that dynamic instance mapping takes place on the design model level, where we are already working with object models that represent the underlying data model entities. This mapping is represented by the orange colored section in Figure 8-1 on page 413.

8.1.1 Usage

As opposed to many other artifacts in our methodology, the data mapping artifact is mostly transient. It serves as a means of transforming the input artifacts, in the form of data models, into deliverables. The deliverables are documents that must be created and describe the transition of data from one systems data model into another model. You can easily extend this part of the methodology to include automated generation of the ETL jobs required at run time. The ETL jobs represent the runtime manifestation and code-based implementation of the document specification.

The description of the data mapping is not just relevant for describing the differences between the individual systems, but also to document the data lineage where you must describe where an attribute comes from and where it is flowing to. Based on the same information, you need transformation capabilities in form of ETL jobs at least during the initial load of our master data management system. After we load the system with data, we require online interfaces to maintain the data. These online interfaces consist of interfaces that serve front-end systems and other attached systems, and batch interfaces. They serve the integration purpose of our master data management system. The following scenarios are the most common usage scenarios:

- Migration
- EAI and SOA integration

Migration

Every time you set up a new data store, you must populate it. Either you use an online interface and allow data to stream in as it comes, or you pump all the data records in to the data store at once. The latter scenarios is the typical choice and is what is assumed to be the choice for migration scenarios. This scenario is commonly known as the initial load of the database, which is sometimes completed with one or more delta loads. Such bulk loading is typically preferred over a load through online activities, because it populates the entire database in a short period. You do not know when the population is completed when you use online transaction processing to push data into your data store. A record that is not updated is not pushed into the data store. Still, both options are valid, depending on the business scenario and the problem that is being resolved.

Regardless of the approach taken to move the data, you are faced with another choice: How are you going to load the data? The choices are:

- Direct load

  In a direct load, you write all data straight in to the database tables. This approach is a fast one, but bypasses any business logic you might have embedded in your application. Thus, you must apply a similar logic in your direct load to guarantee consistent data that also complies with and successfully undergoes rule processing.

- Service load

  A service load scenario uses the business logic that is already in place. Instead of directly adding all records to the database, we start the services provided by the application. This scenario applies the business logic in a way that is similar to a request from online transaction processing. Performing a data migration through a service-based initial load takes much longer compared to a load directly into the database that bypasses the business service execution logic.

The benefit is that you do not have to worry about data consistency. The service load scenario matches the activities you carry out in a dynamic mapping environment, such as an SOA integration.

You typically rely on standard ETL tools, such as IBM InfoSphere DataStage, to perform the data migration task for you. Both scenarios have identical transformations from source to the conceptual target model and the physical model. It is only that the method used to perform the steps to load differs.

EAI and SOA integration

Dynamic integration is required for integration with online transaction processing. The discussions so far in this chapter have outlined outlined why it is necessary to carry out an ETL driven transformation every time the data models do not match. Also, in a master data management system integration, a mismatch often occurs, so every time you integrate a system that is not adapted to the new standards, you need an intermediate step that deals with the translation between the two differing standards.

This scenario shows why you need a service-oriented data mapping. The key considerations you must make in this situation and regarding the usual data mapping is related to the complexities of service mapping, with their respective functional and flow dependencies.

8.1.2 Data mapping types

Data mapping is the activity of linking source data structures to target data structures. It defines where in the target system a certain source attribute is stored and how it is stored. You need data transformation to achieve this setup. This transformation is described through data mapping.

A master data management implementation introduces a new data store, where typical data-centric implementation projects have the following characteristic in common:

- The data structures and data types of the upstream source systems do not match the structures and types used by the master data management system, which requires more or less complex transformation logic.
- Redundancy in the upstream system must be eliminated by assigning priorities to the upstream systems, defining data survivorship rules, or implementing de-duplication mechanisms.
- Existing data is in the source system, which is not required anymore and needs to be excluded during the data migration and synchronization process.

- New data that is not contained in the upstream systems must be part of the master data management implementation, so a data enrichment process must be defined and implemented.
- Considering a standard software implementation (such as InfoSphere MDM Server), the model as it is shipped might not meet your requirements. In this case, you must extend or customize the model.
- The standard software data model comes with certain constraints (mandatory fields, cardinalities, business keys, and so on) that do not match the source data and must be modified, disabled, or bypassed.
- Finding a home for the source data in the master data management system is not an automatic process. Although it is simple to find a home for some attributes, there could be a number of options for others. Standard software such as the InfoSphere MDM Server also provides generic data structures to minimize the amount of customization required.

We are now taking a closer look at the data mapping based on a more conceptual system setup that shows reduced complexity in comparison to the full case study system context. Figure 8-2 shows that conceptual overview.

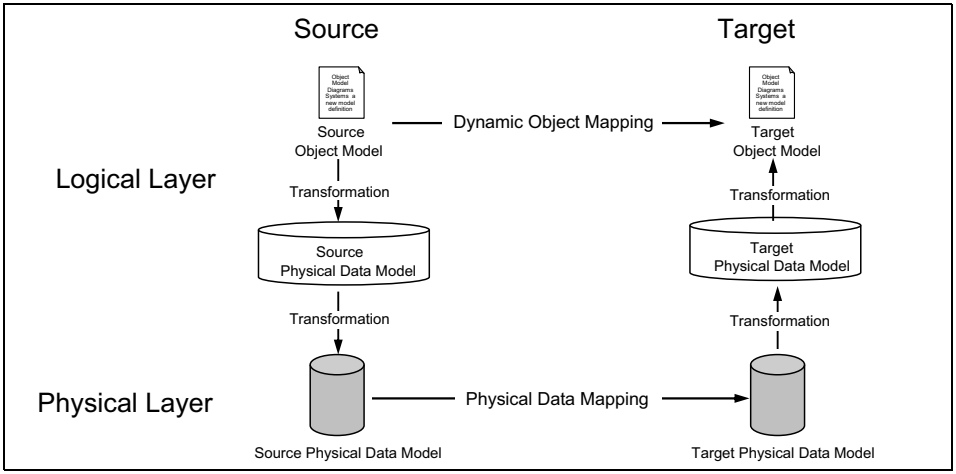

Figure 8-2 Conceptual source target mapping

Figure 8-2 shows how the different data models relate to each other. Data modeling tools provide mechanisms to transform the models between the different layers. Data mapping, however, is much more of an intellectual activity that requires tools such as the IBM InfoSphere Data Architect.

Although data mapping can take place on all modeling levels, we rely on data mappings only on two levels, because of the limited significance of the logical data model, for which we do not require a separate mapping here.

The low-level mapping defines the layer on which the data is mapped, which is the physical layer. Physical data mapping describes the mapping between the source and target database systems. Even though this mapping is essential for the final implementation, it is only sufficient in small and straightforward mapping exercises. On some occasions, we do not have a physical data model. This might be because the source application is a COBOL based business application that exposes its data structures only through appropriate Copybook structures, but does not reveal the physical database structures that it is based upon.

If data mapping becomes more complex, it is necessary to also perform mapping on the higher conceptual layer.

It is necessary to understand the source data model. Only a full understanding of the source domain model ensures that the final physical mapping fulfils all your requirements.

With the two mapping levels, we must consider different types of mapping when we describe data mappings. Unfortunately, this task is not as simple as taking two data models and drawing lines from one attribute to another. Instead, you must be aware of many factors that impact the way you map, which in turn impacts the tool you can use, because not all tools provide you with the same capabilities, and not all tools support the same mapping styles.

The following general styles are identified:

- Static
- Dynamic

Static

The *static* type of data mapping is the most simplistic approach you can take. It is focused on entities and their attributes. The static aspect means that we consider only the entities without considering their instantiation at a later time. We explain what we mean by using simple example based on our case study physical data model (Figure 8-3).

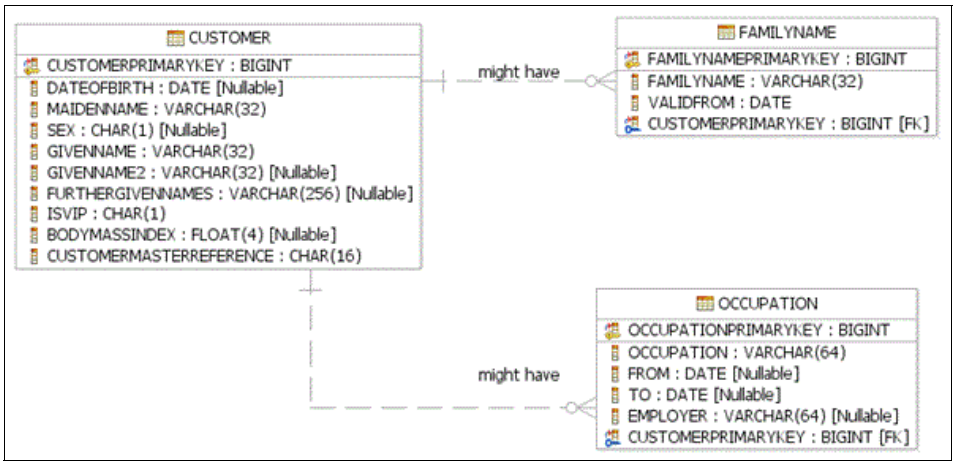

Figure 8-3 Sample physical data model - System A

In this sample implementation based on InfoSphere MDM Server, you must confirm a mapping where the `maidenname` attribute of the customer entity from source System A correlates to one or more of the name attributes on the `personname` entity on the InfoSphere MDM Server data model. The resulting mapping is shown in Figure 8-4.

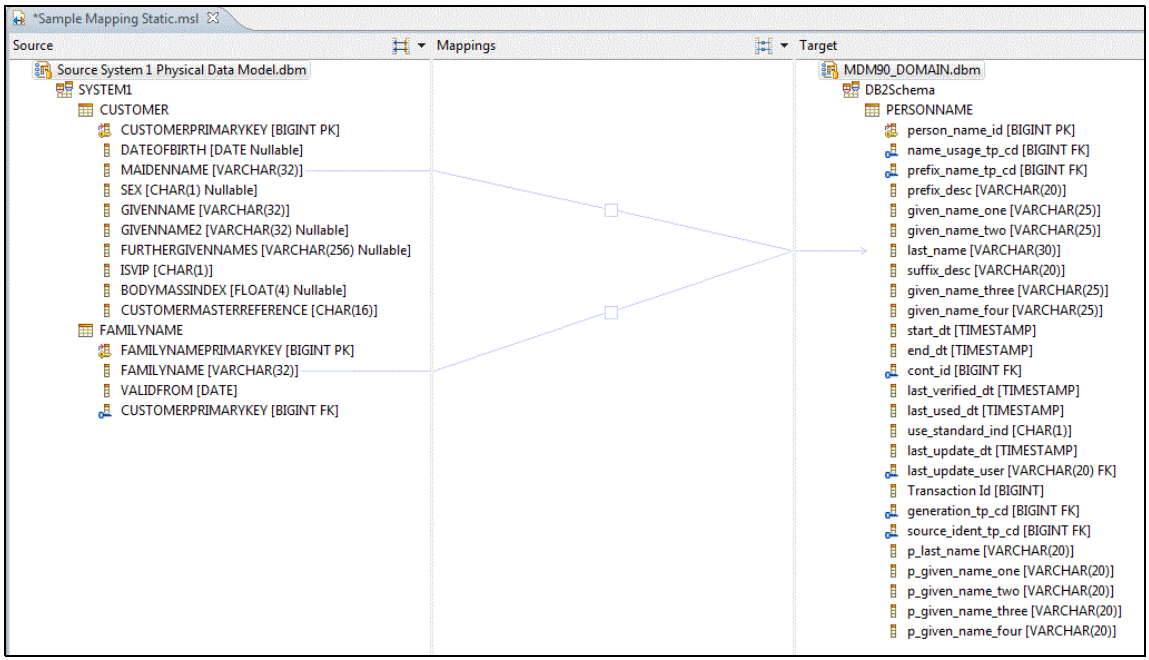

Figure 8-4 Sample static mapping

Looking at Figure 8-4, you cannot tell whether you store one `personname` record or two, which is the main issue with static mapping. Designers, developers, and testers want to know and understand the dynamic aspects of data handling.

However, this information is usable when it comes down to deciding whether you have an attribute available or if you must build an extension or addition to the existing data model. Also, when you want to provide a quick indication regarding the location of the source attributes, this representation suffices. This is also the case in simple prototypes.

Dynamic

A central aspect in *dynamic* data mappings is the distinction between the conditions when you create an instance (instantiation condition) from the expression that defines an attribute value of an instance (mapping rule). It is common that a mapping rule is specific to a selection rule, which means that different instance values rely on different mapping rules. Mappings sometimes even depend on other parts of the context, such as the operation name.

Again, we start with our example based on our case study physical data model. This time, we do not map directly on the physical data model level, but first transform the physical data models we want to use for mapping to a UML object model. Depending on the rules for the logical data model and the object dependencies, you might take a small detour in first transforming from a physical to a logical data model, before you can continue to transform to the representative object model. The resulting object models could look as shown in Figure 8-5 and Figure 8-6 on page 423.

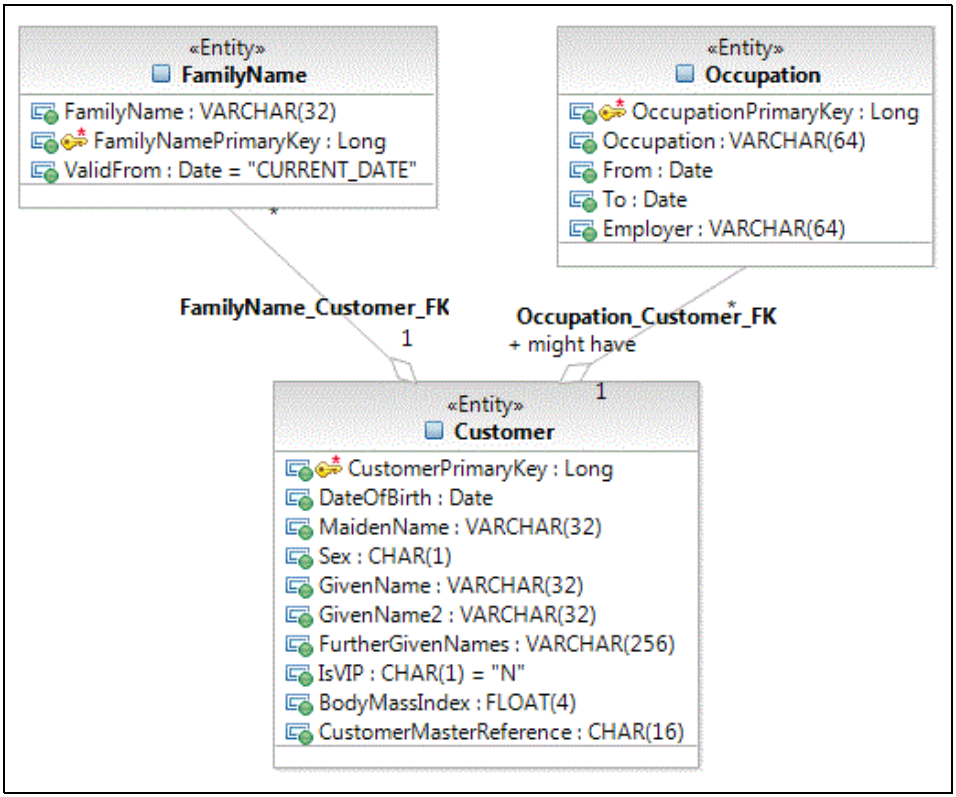

Figure 8-5 Sample transformed object model - system A

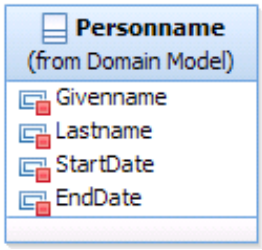

Figure 8-6 Sample transformed object model - system MDM

Now you have the basics for your dynamic mapping. Effectively, you have two choices now for modeling the dynamic map. The difference between the two options is in the amount of work they require and the aid in the form of a clear visual representation and usable input for further generation processes you get out of it. Perhaps you want to adopt both of these choices. Perhaps you prefer one representation for migration design and another representation for SOA integration design. As a preferred practice, pick one choice and apply it to any mapping you require. A single mapping representation increases readability. Our preferences are based on activities because you can easily add additional activities that are required during a transformation, such as lookups against other data sources. Generally, the following choices are available:

- Aggregated attribute classes
- Activities

Aggregated attribute classes

This approach is probably more design intensive and requires an intermediate step before you can carry out the mapping itself. In the intermediate step, we define attribute classes and aggregate them on the main class. What this action does is that it breaks out the individual attributes from the main class and creates a class representation from them. With Rational Software Architect, these classes are required, because the attributes cannot be directly mapped. Depending on the tool you choose, this choice might impact your decision about your dynamic mapping approach.

The resulting class models that you derive from your domains could be the models shown in Figure 8-7 on page 424 and Figure 8-8 on page 424. You work with a simplified version of the master data management model. Limit it only to a single object so that you can focus on the problem.

Figure 8-7 and Figure 8-8 show the resulting intermediate class models.

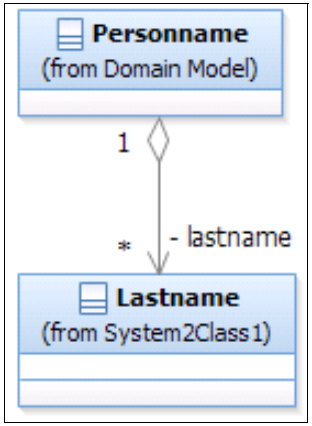

Figure 8-7 Sample simplified domain model with aggregated attribute class - MDM

Now you can start mapping. Remember that your goal is to describe the correct object instantiation, so create an object diagram that shows instances of classes. In our sample, these are the classes from our intermediate class models. When you want to describe the mapping of the `maidenname` attribute on the customer entity, instantiate the corresponding classes and draw them on the object diagram. The same situation is true for the `lastname` attribute on the `personname` entity. Again, draw the appropriate classes onto your diagram. Repeat the same steps for the `familyname`. It is important that you draw a new instance of the `lastname` and `personname` classes so that integration and migration designers, developers, and testers have the correct information.

The resulting dynamic mapping looks like Figure 8-8.

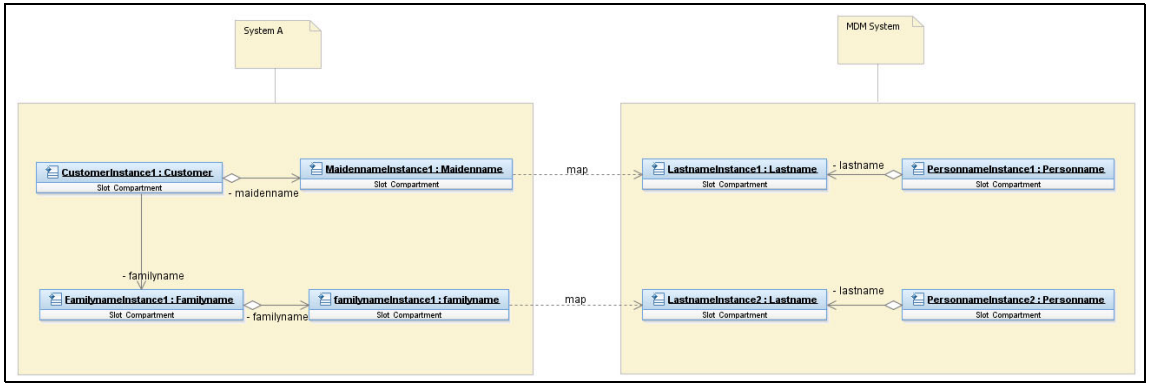

Figure 8-8 Sample dynamic mapping based on aggregated attribute classes

Activities

The second choice to describe dynamic data mapping is a little more visual in respect to the mapping process. The focus here is on the actions required during the mapping rather than the classes that are generated. The classes are also shown and the resulting diagram contains the same mapping relevant information as shown in the choice that uses aggregated attribute classes.

Here, you operate one the source classes and create the target classes as you require them. You have a clear indicator for each instantiation you are working with. The mapping is described in a second action. Here you consume one object, extract the required attribute, and map it to the wanted target attribute on the instantiated output object. Repeat this design process for each mapping that you must describe.

Based on the representation shown in Figure 8-9, you can see how simple it is to add other mapping-related activities in to the mapping process. Often, you must carry out a lookup against an external data source to determine the mapped result.

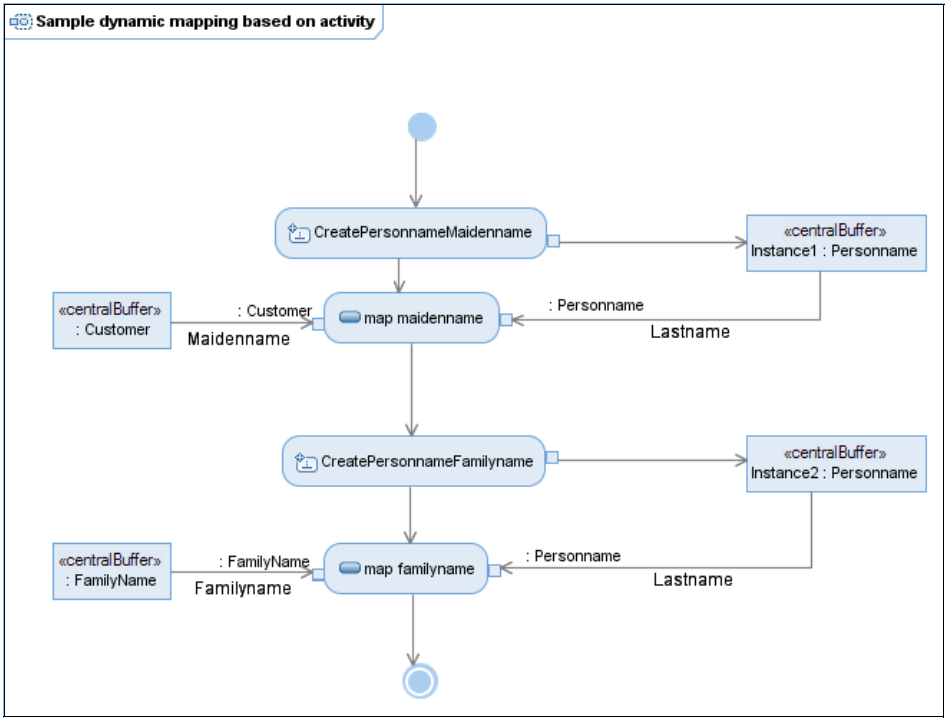

Figure 8-9 Sample dynamic mapping based on activity

8.1.3 Service-related data mapping

So far, we have only described the two styles of mapping, that is, data and service. There is an additional factor of the data mapping that we have not outlined so far. Although we do not want to cover it in detail here, you should have some insights and rough guidance. You will find further references when we describe service design in far greater detail.

It is necessary to be knowledgeable in all multiple data models and the services that manipulate them, but mostly about the services and the data they manipulate. To provide a better understanding of this issue, we give you a short analysis of service design. We can name four distinct functions:

- Receive an incoming message based on the owning protocol.
- Execute the flow logic.
- Map to the lower-level protocol (derived from data model of that lower-level layer).
- Execute lower-level services or persist data (if you are already at the lowest level service).

Take a closer look at the mapping task. When you have two different vertical data models at play, you have a mapping. The following scenario makes the problem a little clearer.

First, there is a domain layer that consists of some specific business objects. Next, there is the MDM layer that uses generic objects and classifications. Our domain model has two objects, a General Agreement and a Contract, both of which are connected through a “has” association. On the MDM layer, that maps to:

- General Agreement maps to Contract + ContractComponent.
- Contract maps to ContractComponent.
- Has maps to ContractRelationship.

If you apply static mappings, you either leave out “has” or always instantiate the “has” in our service. This situation is not entirely correct and optimized. We need a service-dependent mapping that allows for optimized communication with lower service layers. The static mappings do not allow that situation and result in an increase of service calls across layers.

The following types of cases require special attention during data mappings:

- Functional dependencies
- Flow logic

Functional dependencies

Based on our example scenario, here is the illustration of a functional dependency that we must acknowledge. If, on the master data management level, you execute a `getContract()` service, you provide the following parameters:

- Reference: Business key (or Primary key) of the required master data management object.
- `ContractBObj` to change: None.
- Filter: Assume that you want to inquire about specific time slices.

The filter highlights the mapping issue. If you do not use the filter, you do not have to look at the ContractRelationship that is mapped to “has”. So, this lower-level object does not have to be mapped. However, if you use the filter to retrieve only ACTIVE ContractRelationships, you must map this object.

Flow logic

Here is another example based on the same scenario and the same objects, but with different service calls. This time, we start `addContract()` with the following parameters:

- Reference: Business key (or Primary key) of the general agreement object. You want to attach the contract object to it.
- `ContractBObj` to change: The new contract object to be added.
- Filter: None.

You must map the ContractRelationship that is mapped to “has” every time you use this service, because you must be able to attach the Contract to the Agreement through the ContractRelationship object of that lower level.

Start `updateContract()` with the following parameters instead:

- Reference: Business key (or Primary key) of the Contract object. You want to change that object only.
- `ContractObj` to change: The Contract object to be updated.
- Filter: None.

Even though the object model is still the same, you no longer require the ContractRelationship that maps to “has”. The only thing that changed whether you need the “has” ContractRelationship is the type of service that you want to execute. The problem is dependent on the execution flow and the type of service that you start from the higher level service flow logic.

8.1.4 Requirements

Data mappings have their own requirements. You must recognize these requirements and decide on the correct mappings. Some of these requirements are based on usage, data-driven migration, or service-driven EAI and SOA integration and static or dynamic mappings.

The following additional requirements impact your decision regarding the tool and the style of mapping that you adopt:

- Traceability
- Versioning
- Interoperability: Tool integration
- Accessibility

Traceability

In 4.6, “Traceability” on page 237, we described model traceability. Everything described in that section is still applicable when it comes to data mapping. One of the most important aspects of traceability is its ability to analyze the impact that a change to any part of the mapping has on the mapping for the entire system. This aspect is even more important when the complexity of the data mapping increases because of the number of systems and the size of the data sources you map between, and the level of reuse that you base you data mapping upon.

Traceability in respect to data mapping has another meaning too, where it is known as *data lineage*. You must document the data flow to provide input to external audits against legislative compliance requirements. Data mapping represents only one part of the overall data lineage requirements, but it is an important part. A complete picture of the data flow also includes processing rules and other business logic that we do not cover in typical data mappings. Nevertheless, the data mapping provides you with the backbone for such data lineage reports. The static data mappings generally define how data flows from one system to the next, while the dynamic mappings include more detailed data flow, possibly including data processing that takes place during the transformations.

Versioning

The development of mapping rules is similar to a normal software development process. You must handle changes or refinement of mappings. A good mapping specification can handle multiple versions of mapping rules, providing a sound baseline for each version.

A good mapping tool:

- Provides a mechanism to track differences between two versions of a mapping.
- Can draw a baseline for a version of the mapping spec
- Has a status for a mapping rule to distinguish whether it is in work, finished, or historical

Interoperability: Tool integration

When attempting correct tool integration, metadata handling is the cornerstone of a sound mapping tool. Usually in major data integration engagements, several tools come into play. Integrated access to the glossary boosts mapping definitions and enables tool assisted discovery of relationships between data entities. ETL developers and testers are the main consumers of mapping rules. Truly integrated tools accelerate ETL development, and you can use them to easily compare the specification in form of mapping definitions to the implementation that manifests itself in ETL jobs.

Accessibility

Most of the mapping activities we have described so far are related to the actual task that defines the mapping. The mapping is accessed by the business analyst that describes it. After the data mappings are concluded, you must develop, deploy, and use them. Based on these data mappings activities, you have more need for access. This demand increases only with the level of reuse you try and apply to the data mapping definitions. The roles and tools that require access to our data mappings are:

- Business Analysts

  They originate and describe the data mappings.

- Developers

  They must read and interpret the data mappings to build the required transformations.

- Data Stewards

  They must be aware of the data mappings, because they are responsible for the data quality and accuracy, which is influenced by the logic built into the data mapping.

- Enterprise Architects

  They are responsible for the overall view of the system landscape. As part of an information architecture, they also need to be aware of the data flows across systems.

- Tool Integration

  Data mappings are not an activity that is carried out on its own. This activity depends on a good understanding of the data through data analysis, metadata, and glossary information. Close ties with the development tools prevent errors and increase quality.

- ETL / EAI / EII

  Data mappings typically result in ETL jobs. In combination with a good tool integration, you can describe your transformations and deploy them for runtime execution as part of an ETL job.

- Reporting

  Data lineage reports are becoming more common. You must be able to quickly and easily extract the relevant mapping information and display it consistently and comprehensively in the context of the larger data flow across systems.

8.1.5 Tools

Although there is a large variety of tools available on the market, not many of these tools cover various mappings, are easy to use, and provide the accessibility you need. Here we describe the following categories of tools:

- General purpose tools
- Specialized tools
- Supporting tools

General purpose tools

A general purpose tool is either defined through high availability or strong flexibility. The two main types are:

- *UML tools* have the tightest possible integration with other design artifacts in respect to an overall model-drive development methodology. They are also flexible and provide all the required functional capabilities. They do have some deficiencies in respect to accessibility. Even if the missing pieces can be built as plug-ins or in many other ways, there is a problem of the integration with the many roles and tools that require access to the data mapping. It can be costly if you must build access to all these roles and tools.

  These tools represent a viable option if your main purpose revolves around a data mapping description that drives further generation processes in a model-driven development (MDD). For our methodology, we decide to use UML-based dynamic mappings. The specific tool that we use is IBM Rational Software Architect, which is the only tool we need in our methodology.

- *Spreadsheet-based mapping* clearly lacks possibilities. Mapping definitions that are captured in spreadsheets are barely able to convey their message to the business analysts or to the developers that are responsible for deriving the correct information from it to create the jobs that perform the mapping-based transformation. The tables you must create to describe all the dimensions, including all systems and their database tables, are growing rapidly, and pose an interpretation challenge to anyone that reads them. Apart from the spreadsheet quickly growing in size, it is hard to describe dynamic instance-based mappings in such a spreadsheet.

  Even though this option is the default for InfoSphere MDM Server based implementation projects, we base our methodology on mapping definitions that we capture in other tools than spreadsheets. We use IBM Rational Software Architect for the dynamic instance mapping and IBM InfoSphere Data Architect for the static entity mapping.

Specialized tools

These tools provide a design process and appropriate user interface that you can use to capture the mapping relevant information. However, you must map within the rules of the tool. Most tools provide you with a quick and easy static entity mapping capability, but when it comes to dynamic instance mapping, many of the specialized tools still do not do as well at it as they do with static mapping.

Some of these specialized tools are as follows:

- FastTrack

  FastTrack is the mapping tool provided by IBM InfoSphere Information Server. Using this tool, you have good integration in respect to accessibility requirements, because it connects with the IBM InfoSphere Information Server Metadata Workbench and the IBM InfoSphere Information Server Business Glossary. You can also base your analysis (obtained from IBM InfoSphere Information Server Information Analyzer) on it, and use all of your data when IBM InfoSphere Information Server FastTrack defines the data mappings. You can use the data mapping specifications through IBM InfoSphere Information Server FastTrack to create the corresponding IBM InfoSphere Information Server DataStage ETL jobs. Even if you do not rely on FastTrack as part of the methodology described in this book, this tool provides a good and integrated data mapping option.

- IBM InfoSphere Data Architect

  IBM InfoSphere Data Architect is a tool based on the IBM Rational platform. It has good integration with our choice for an MDD platform in our example. However, the tool does not work in certain situations, which in turn results in loss of traceability or a reduction of design or development options.

  For example, you might want to use the IBM InfoSphere Data Architect built-in glossary to for enforcement of naming standards for all artifacts, but you cannot use the glossary information across the boundaries of IBM InfoSphere Data Architect. The IBM InfoSphere Information Server Business Glossary promises better integration, and supports both IBM Rational Software Architect and IBM InfoSphere Data Architect. However, in our attempt to limit the number of tools used in our example, we decide against using IBM InfoSphere Information Server Business Glossary. Instead, we use IBM InfoSphere Data Architect for static entity mapping, because it does a good job and is easy to use. But IBM InfoSphere Data Architect is also our tool of choice for data modeling, so it was appropriate to expand on its usage and include data mapping too.

  The greatest disadvantage in using this tool is that you are missing capabilities for dynamic instance mapping. We address that deficiency in such way that we require transformations to UML models, and then can fall back on UML-based mapping definitions.

- Custom built

  Custom built mapping tools are often built from scratch for just one migration or data integration project. They are often specific to the engagement and vary in the fulfilment of requirements. Almost all of these tools lack integration with other tools, such as ETL, data modeling, or a glossary.

Supporting tools

In addition to the mapping tools, it is important to have additional tools available. Some of these tools are as follows:

- Glossary

  A *glossary* is an important asset to have in any data driven project. It provides you with business relevant insights into the entities you are working with. The existence of a domain model does not diminish its relevance. Both should coexist and contribute to clear and unambiguous business language.

  One interesting capability of a glossary is its ability to store relationships between similar or even identical terms. If, for example, the same attribute exists in multiple systems, then we want to correlate them, while still highlighting the subtle differences they have.

Another useful feature is the glossary's ability to maintain further information about the attributes, such as the physical table in which it occurs. This feature increases the usage scenarios and completes the information-centric view of systems. During the entire design and development cycle, it is important to be able to rely on the terms defined. A separate glossary in conjunction with the domain model provides you with a central repository for such lookups.

- Profiling

  Especially with data quality in mind, you need a good understanding of your data. Although the business glossary contributes to this goal, you also need to reveal the hidden meaning. You achieve this goal analysis obtained through *profiling*. Profiling can be done with IBM InfoSphere Information Server Information Analyzer or IBM InfoSphere Discovery.

  In addition, you obtain metadata to carry out the profiling, which you store in the metadata repository. In respect to data mapping, you can learn a number of things from such an analysis. Most importantly, you might identify a new mapping rule based on the data stored in the tables you investigate. In the best case, all it takes is a minor adoption of an already existing mapping definition, which is also when impact analysis plays a role again.

- Metadata

  All activities, including data modeling, data mapping, and profiling, are based on *metadata* of the data sources you work with. You must understand the data sources and relay that information to other people in other teams. The IBM InfoSphere Information Server Metadata Workbench provides this metadata.

8.1.6 Process

You have defined everything you need for the data mapping apart from the process that drives it. The process is important for a successful mapping and to other artifacts that are impacted by the mapping. One of these artifacts is the master data management physical data model for which you might have to define customizations in case you can map to an existing attribute. A successful data mapping process consists of these main steps:

1. Building the foundation.
2. Mapping the data.
3. Verifying the mapping.

Building the foundation

When performing data mapping, you must know your data sources thoroughly. You can create a data mapping of high quality only when you are fully aware of all the attributes, their exact semantic meaning, and the role they play in the use cases.

Building the foundation includes the following steps:

1. Capture the business terms and make them easily retrievable. This step can be completed in two ways:
   - Create a glossary entry for every entity and attribute you encounter. Ensure that you obtain precise and useful information. Because of the glossary's similarities to entities and attributes in other systems, you should link these similar terms with each other.
   - Use a domain model. In our example, we use a domain-driven design (DDD) methodology. It provides us with a visual representation of the glossary information. Both models should coexist to maximize their effectiveness.
2. Profile the data in all cases, if at all possible.

   Often, time constraints prevent initial data profiling. However, profiling is important because it increases your understanding of the problem domain, and you can identify gaps and issues with the business term definitions early. Therefore, you can address these issues either through data quality improvements at the sources or adopt mapping rules or other business logic. You achieve the greatest success when relying on production data only. In fact, any data other than production data is not useful in that it is as arbitrary as the number of John Does in a test system. Compliance with the security guidelines regarding handling of production data is important. As part of the profiling, you also capture the metadata and therefore are able to combine multiple activities into one. Because you do not know what data inconsistencies to look for, you must explore all the data.

Table 8-1, which is based on the case study system A data model, demonstrates outcomes from profiling.

Table 8-1 Outcomes from profiling

Customer PrimaryKey	DateOfBirth	Maiden Name	Sex	GivenName	Given Name2	Further Given Names	IsVIP	BodyMass index
1	1965-Dec-16	Schroder	M	Frank	John	Fred,James, Bill	N	18
2	1979-Mar-03	Smith	F	Lisa	Sue	NULL	Y	See insurance policy

Customer PrimaryKey	DateOfBirth	Maiden Name	Sex	GivenName	Given Name2	Further Given Names	IsVIP	BodyMass index
3	NULL	CRAMER	1	MELINDA	NULL	NULL	NULL	20

Looking at these records, there are a number of things you can immediately spot:

- The birth date is not a mandatory field.
- There is insufficient data standardization (CRAMER versus Smith or 1, M, and F as type codes for the person's gender).
- Columns are used to store multiple attributes (Fred, James, Bill in column FurtherGivenNames), which is not correct.
- Free-text information is included in apparently structured fields (the comment "see insurance policy" in the BodyMassIndex column).

Source systems that store data in such a way are common in master data management projects. Thorough profiling of the source reveals these issues and you can resolve them well in advance, preferably before you design your data mapping, rules, or any other business logic that relies on this data.

Mapping the data

After you have a good understanding of the data, you can start the data mapping. Remember that the data model can change while you are mapping. Therefore, do not expect to have finalized the InfoSphere MDM Server data model at this stage.

Before you can customize the InfoSphere MDM Server data model, you must have a clear view of the data that needs to be stored. We argue that you can obtain such a view only through the first iteration of data mappings and not the other way round. How else could you know which other attributes to add to the data model? So, it is predominantly the InfoSphere MDM Server or any comparable products data model that is impacted and that might require customization during this process. Other data sources should remain stable at least during the time of mapping.

The mapping-related substeps are:

1. Identify the best fit target attribute.
2. Extend the data model where required.
3. Regenerate the data model.
4. Map the source attribute to best fit the target attribute.

Identifying best fit target attribute requires a good understanding of both data models, the source, and the target. The semantic meaning must be understood and clear, which is why you need the first mapping process step. Use the knowledge obtained through the first step and apply it. You should be able to place the attribute on an existing attribute. In some cases, this action might not be possible because the best fit attribute does not exist. If that is the case, you must either hijack an attribute with a different semantic meaning or, if possible, extend the data model to provide the missing attribute.

Extending a data model where required is an important aspect when you are working with customizable data models. This situation is true in the case of InfoSphere MDM Server, but also when you design a new data warehouse and are still free to add an attribute with the correct justifications. This step is based on the outcome of the identification for the best fit.

Regenerating a data model is necessary after you decide on the extension or customization of your data model. Go back to the first step, where you identify the best fit, after you change the data model. The best fit now should be the newly added attribute. In fact, you must perform an even earlier step, because you must add the metadata for this new attribute to your metadata repository and provide the new business term for it.

Mapping source attributes to best fit the target attribute can be done either in a static or dynamic fashion.

Verifying the mapping

Now you have a data mapping. But does it do what you expect it do? Does your data look the way you expect it to be? There are a number of ways to ensure that the mapping is correct and truly and accurately reflects the requirements. Some of these verification methods are:

1. Prototyping the mapping
2. Profiling the mapped data

Prototyping the mapping is important because it produces results from a designed and developed mapping. You can verify the data mapping based on the runtime executed transformations from the developed data mapping. The prototype is an early indicator regarding the quality of the data and the correctness of the data mapping.

Profiling the mapped data completes the data mapping tasks. Even if you carried out data profiling before, you only explored the data sources to obtain a better understanding about the data they contain. At this stage, you know enough about the data that you do not have to explore anymore. Instead, you can do a thorough investigation to see whether the mapped and transformed data complies with our expectations.

8.2 Transformations

As previously mentioned, we consider transformations an important part of the metamodel. They provide a way to link artifacts that could not otherwise be related to each other. They not only translate model instances of different notations and layers, but also enable you to generate source code from the implementation model. Due to their dynamic nature, transformations can become complex. It is important to take special care when designing transformations. In the following sections, we describe some of the most important aspects of transformations in greater detail.

8.2.1 Definition

Applying a transformation means that you transform an instance of a metamodel of type A to an instance of a metamodel of type B. These instances are usually either model files or text files. The target does not need to be based on a formal metamodel. Often the target formats are programming languages with a well-defined grammar and specification, but without a formal metamodel. This distinction has an impact on the selected tools.

With respect to the IBM Rational tool suite, you can use transformations to convert UML or other Eclipse Ecore based models to code, convert code to UML models, and convert models to models at different levels of abstraction. Transformations take place at any model level. The source and target representations can either be text- or model-based. Most importantly, transformations do not rely solely on the models themselves, but provide additional context, either with configurations, templates, or source code. They help architects and designers implement preferred practices, guidelines, and patterns.

Transformation and generation refer to different parts of the process. Formally speaking, generation refers to the last step of the entire process only. However, in the following sections, we usually use the word transformation for both parts of the process: the transformation from one model representation to another one, and the transformation into the code representation.

Transformations versus mappings

Transformations differ from mappings in that they represent the “dynamic” conversion between different levels of abstraction while mappings solely bridge the static gap between models and data models in particular. Mappings represent a documentation of the data flow, while transformations truly change the model that describes the functions that work based on that data.

Nonetheless, we also use the concept of dynamic mappings in this book. What is the difference between a dynamic mapping and a transformation?

The main focus of dynamic mappings is to map instances or instance values from one object model to another. The focal point is the flow of the data, but on the most detailed level possible. This means that depending upon the context and the instance model, a mapping between two objects changes. Context could mean a value such as an organization type of an organization instance that determines which associations are valid in the scenario. Context could also encompass the type of operation that is executed. This situation refers to the service-dependent type of mapping. This type of mapping deals with the problem that the cardinality of a nested input parameter could depend on whether the current operation is an add or an update transaction. There are different ways to design such a mapping. You can create class instances of the attributes you refer to and associate these pseudo classes with some object. Additionally, you can create specific objects for each create, read, update, and delete operation which can then again be used within an object diagram or you can use slot values in an activity. Regardless of the approach you take, these dynamic mappings differ from transformations to some degree, which includes:

- The mappings refer to instances and values of an instance, while transformations are applied to instances, but they usually do not refer to instance values.
- The mappings are considered part of the specification, while the transformations go beyond the service specification.
- Dynamic mappings are meant to describe the dynamic part of an interface. Transformations describe a context-aware mapping between two metamodels, not a dynamic mapping within one model.

Components of the transformation process

The process of transforming model instances of different granularity and detail is complex. You should encapsulate the logic behind these transformations and to make it available as a reusable component. There are different ways to improve reusability.

First of all, you must separate “dynamic” transformation logic from static parts of the code. A common means of achieving this separation is to use code templates along with parameters. This concept particularly applies to those targets that are not based on a complete metamodel, that is, model-to-text transformations. Then, instead of defining a meta representation of an element, you put the static parts of it into a template. Even if there is a metamodel of your programming language available, it is often limited to those parts of a language that are rather static. Dynamic aspects, such as method bodies, are much harder to describe in a formal way.

This situation is where templates come into play. Imagine a simple condition that needs to be transformed into the corresponding “if-else” statement in Java. You could describe the Java element in a formal way by introducing operators and operands and special subtypes of them, but it is much easier to define an “if-else” template that contains a few dynamic expressions.

Templates are known in other contexts, such as Ant build scripts where the actual build file also contains references to parameters that you want replace with the actual and current value at run time. The same logic applies to transformations. In the context of Eclipse and IBM Rational Software Architect, which are the tools of choice in this book, the template system is provided through Java Emitter Template (JET). These files look similar to JavaServer Pages (JSP) and might contain parameterized expressions that are substituted by the actual values at run time.

Apart from templates, you must ensure that you separate the generation user interface (UI) from the transformation logic. Providing a user interface is an important part of the model-driven approach. Although you could argue that you prefer scripts over UIs, a comprehensive modeling framework should provide support for both script-based and UI-based approaches. This design becomes obvious if you look at the tasks that the project members are responsible for. After the designer delivers the design model, a developer must provide the configuration of the transformation from the design model to the implementation model. The developer could do this task by writing a configuration file, but it is far less error-prone to use a UI for this purpose.

In the next step, the developer takes the implementation model and generates source code. Again, this step might require some additional configuration, and sometimes the developer also must adjust the customized implementation model to make it work with the target platform. Because this adjustment happens on the level of Eclipse based models, which are represented by rather large XML files that are not exactly easy to read, having a UI makes it much easier to do this customization. This approach is in line with the general philosophy of designing and developing everything within one integrated platform.

Scripts come into play after the development is completed. The build tool takes the output files of the configuration and customization process and produces the final source code before building and deploying it. Ideally, the build and deployment process should be able to regenerate the entire code from the higher level models together with the customizations and configurations applied on every succeeding level.

Another discussion involves reusability and separation of concerns. The separation between UI and other components can either be achieved in a rather “traditional” way, for example, by separating the UI and services layer through appropriate design and implementation patterns. The best known ones are probably strategies, factories, and proxies. However, instead of implementing a new user interface, you can also use those features that are already part of the Eclipse tool suite. Eclipse modeling projects are designed such that the editor for your domain-specific language can be generated as part of the model structure. Thus, customizing your domain-specific language and configuring your model becomes a matter of using Eclipse editors. These features are based on the Graphical Modeling Framework (GMF).

Another important component of a transformation is the in-memory representation of the source and the target model. Do not assume that you can simply take the standard modeling elements that come with Eclipse. To a large extent, finding the correct representation of your own model depends on your domain-specific language or UML profile and the layer of abstraction it operates on.

In many cases, you might find that it makes sense to introduce an intermediary in-memory model, not only to decouple source and target, but also to make the transformation process less involved and complicated. It is a preferred practice to use such a model. It does not need to be persisted, though. Using it dynamically during the transformation helps structure and separate concerns.

The implementation of the transformation logic is one of your core tasks. Fortunately, the tools you use already provide an excellent way to implement this logic. More specifically, in IBM Rational Software Architect, a transformation consists of a set of transforms that can be configured and run through a transformation configuration. Transforms are containers for two types of classes. The classes that relate source and target elements to each other are called rules. A rule is responsible for relating one source element to the target. Because of the complexities of the models, your transformation usually contains many rules. The second type is called the extractor. As the name implies, extractors are responsible for the extraction of elements from the model. Often, you see that the visitor pattern is used for operations on specific elements. By using this pattern, you avoid procedural code. Rather than implementing the transformation logic into the model parser, use separate visitor classes that are applied upon reaching a specific model element.

Further design considerations

Transformations must also fulfil further requirements in addition to the previous considerations.

First, transformation transform a higher-level, more abstract model to a more detailed, low-level model, or vice versa. They are not designed as a means of reverse-engineering the source code, though. However, the models that pertain to source systems might require reverse-engineering to some degree. Think of a database-centric application where the only documentation available is the physical data model itself. The only way to move this information to a higher level UML model is to transform the physical model to a logical model and then continue to transform this logical model in to its UML counterpart. Avoid round-trip engineering, as it is not an option in a model-driven environment.

The second important topic is model validations. It is crucial that any model generated by the code generator is validated against the respective metamodel. Also, after every change of the generator, the generator must be revalidated.

Third, a transformation must be implemented such that it can deal with abbreviations. Often, design models use simplified notations. It is then the responsibility of the transformation to interpret this abbreviation correctly. For example, the notations for fixed values and exceptions are not complete from a UML perspective. You must consider this situation in the corresponding transformation.

Last, the source models we use throughout this book use several languages. Although data models do not interact directly with the Java transformations, analysis and Object Constraint Language (OCL) constraints do. So, the transformation needs to handle both languages, that is, UML and the constraint language we use.

8.2.2 Tool support

The IBM Rational tools we base our approach on provide a comprehensive transformation implementation framework. In this section, we describe the most relevant parts of this framework, which are:

- Transformation configurations
- Transformations
- Transformation extensions
- The Eclipse Modeling Framework
- A Java Emitter Template and code generation
- Other tools and integration alternatives

Transformation configurations

A common starting point, especially for model-to-model transformations, is to use the transformation plug-ins available in IBM Rational Software Architect, as shown in Figure 8-10.

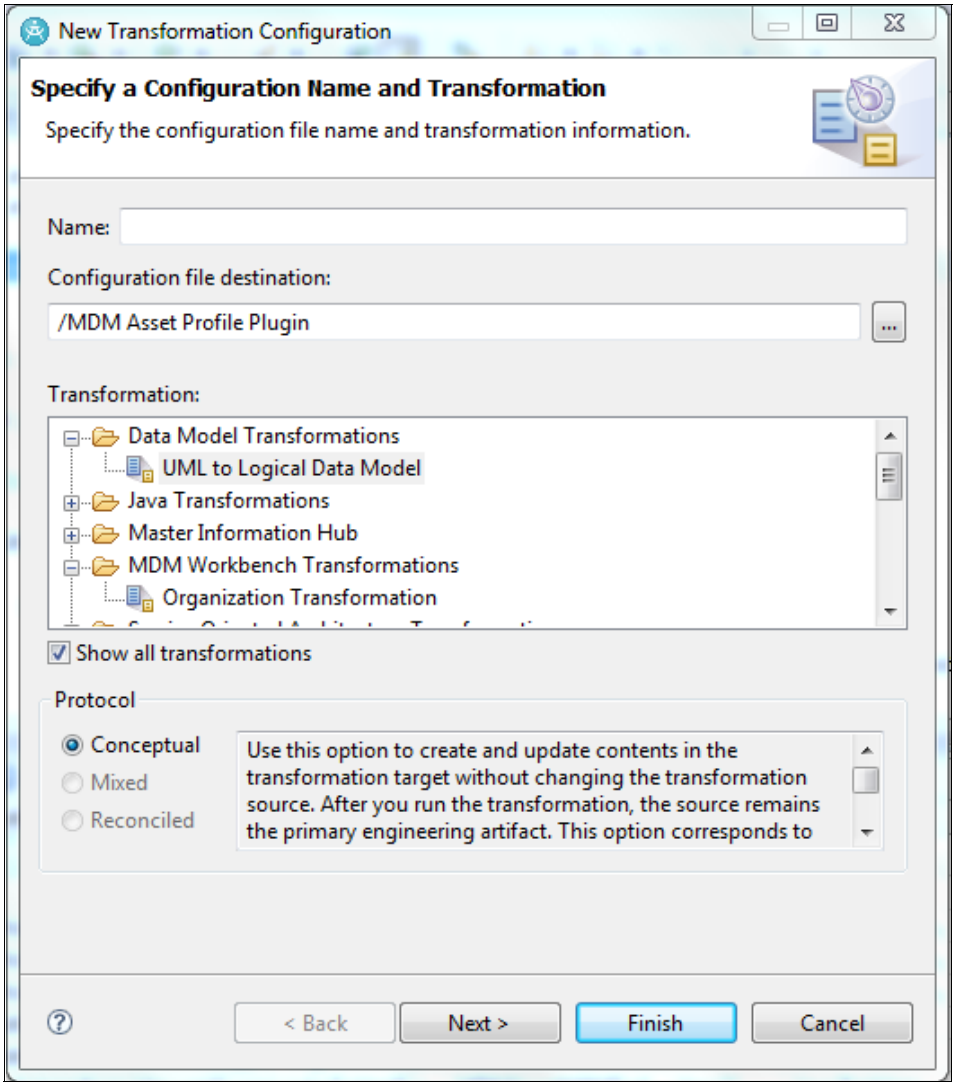

Figure 8-10 Available transformation configurations

You can use these plug-ins by creating a transformation configuration. In fact, you always must define a transformation configuration before you can run any transformation. A transformation configuration is an instance of a transformation that includes information, such as a unique name, the source, and the target of the transformation. It might also include information specific to the transformation. Therefore, the configuration provides the runtime context of a transformation. In the configuration UI, the target is expressed in terms of the target container, which provides a reference to the location where the target artifacts are stored.

Depending on the type of transformation you choose, you can choose the protocol you want to be used with the transformation. Because we aim at forward engineering only, our preferred selection is always the conceptual protocol. Choosing this option means that the source model becomes the primary artifact. Changes made in this model are propagated by running the transformation.

Another important aspect related to transformations is traceability. Traceability is commonly considered one of the key features of a generation-based approach. If you generate your code from the model on the next upper level in such a way that it adds traceability information with the transformation, then you can always trace any change in the code back to the corresponding change in the design model. Technically, IBM Rational Software Architect and Eclipse achieve this tracing by adding model references to the source files.

Assume that you created a Java file named “Order” and you want to discover on which design element this class is based. The Java source code looks as shown in Example 8-1.

Example 8-1 Java source code

```
/**
* @generated
sourceid:platform:/resource/UMLS/UMLS.emx#_NLFGKD9reuwV5GC2yg
*/
public class Order {
}
```

To enable this traceability in a transformation configuration, select the **Create trace relationships** check box on the Common page of the transformation configuration UI. This feature is limited to certain elements. We describe this limitation later in this section. The traceability option shown applies to UML-to-Java transformation only. Which options are available depends on the type of transformation. After traceability is enabled for a configuration, you can run a query from the project explorer and look up the implementation of a UML class or interface.

The configuration UI has another option that you can use to create a debug log for the transformation. Although this option is a useful one, it is not meant as a replacement for traceability. The information saved in the log file does not suffice to establish the relationship between the source and the target element.

Another notable aspect that is worth mentioning regarding transformation configurations is their support for mappings. In a UML-to-Java transformation, you can create a mapping model that you can use to define class names and package names other than the default ones and map them to the source. For example, if you want to generate applications for different clients from the same model, you can use this feature to provide customized names.

Transformations

Due to the specific nature of our models, and our focus on master data management and domain-specific business requirements, we must work with custom transformations. The standard transformations are of little value to the scenario that this book describes because of the degree of customization that must be applied to the transformations. Using IBM Rational Software Architect, you must use transformation configurations, even when you create custom transformations.

In IBM Rational Software Architect, a transformation corresponds to a transformation plug-in together with its “instantiation” that represents the configuration of the transformation. To create transformation plug-ins, you must first install the extensibility features, which are packaged as an optional product component. In addition, the modeling and XML developer capabilities in IBM Rational Software Architect must be activated. After you activate these features, you can use the authoring tool for transformations, an API, and some samples that are provided together with the default installation of IBM Rational Software Architect.

After you create the transformation plug-in through the wizard, the next step is to add the `com.ibm.xtools.transform.core.transformationProviders` extension to the `plugin.xml` file in the plug-in that contains the transformation. Describing the details of a transformation provider is beyond the scope of this chapter. You should not change the default behavior of the transformation provider and the transformation engine.

The most common starting point for the actual implementation of a transformation is a platform-independent model (PIM), such as a class model that does not contain any reference to implementation specifics. You then use the transformation to convert this class model into a platform-specific representation, for example, a Java package.

In the transformation process, you can refer to other information sources, such as metamodels, domain-specific languages, and pattern catalogs. Regarding the approach we described throughout this book, our custom transformations rely on the UML profiles and the master data management metamodel. Most of the code that makes up the transformation consists of routines to extract elements from the source model and rules about how to map these elements to the target representation.

The level of abstraction you should use in your custom transformation depends on the type of transformation we define. If the transformation that is available is a model-to-model transformation within the scope of UML, you can operate on a rather abstract level and only work with UML2 classes and xtools classes provided with IBM Rational Software Architect. UML2 is the Eclipse implementation of the UML metamodel, while the xtools classes encapsulate and extend lower-level transformation packages in Eclipse.

Example 8-2 shows the transformation logic if you use this level of granularity.

Example 8-2 Transformation logic with this level of granularity

```
UMLPackage uml2 = UMLPackage.eINSTANCE;
Class cls = (Class) ruleContext.getSource();
String className = cls.getName();
final Package pkg = (Package) ruleContext
   .getPropertyValue("targetPackage"); //$NON-NLS-1$

// Create the interface, implementation and factory classes
// and put them in the package
Interface intface = null;
Class impl = null;

if (pkg != null) {
   String intfaceName = "I" + className; //$NON-NLS-1$
   intface = ModelUtility.getInterfaceByName(pkg,
      intfaceName);
   if (intface == null) {
      intface = pkg.createOwnedInterface(intfaceName);
   }
   String implClassName = className + "Impl"; //$NON-NLS-1$
   impl = ModelUtility.getClassByName(pkg, implClassName);
   if (impl == null) {
      impl = pkg.createOwnedClass(implClassName, false);
   }
   …
}
```

The closer you get to the implementation, the more likely it becomes that you must use the lower-level semantics provided by the Eclipse Modeling Framework (EMF).You will soon seen what these semantics mean in the context of master data management modeling.

Transformation extensions

Instead of implementing an entirely new transformation, you can also extend an existing one. Although this approach seems to be fast and promising, it is limited to a few scenarios. In our scenario, extending a transformation is only an option if do not plan to apply major changes to the target model. A good example for this transformation is the “mapping” of a logical data model to the corresponding UML model. Such a transformation mainly serves to move a model file from the IBM InfoSphere Data Architect environment to IBM Rational Software Architect.

You can use extensions only to apply minor changes to the transformation, but not to change its fundamental behavior. In our methodology, an extension can be used to add model comments or further references to the target model for traceability.

The Eclipse Modeling Framework

All of the features we have described so far are bound to IBM Rational Software Architect. When it comes to model-to-text transformation and code generation, you must use lower-level features than the transformations provided by IBM Rational Software Architect alone. One reason for this requirement is purely technical: You cannot describe Java language elements by using UML elements only. To address this issue, you need an intermediary representation of the model that is based on EMF, which can be used both with UML models and the Java implementation.

You could argue that IBM Rational Software Architect provides a UML-to-Java transformation and should be able to convert the customized master data management model directly to Java code. However, the available transformation handles only a subset of Java and does not suffice for the required master data management model to Java transformation. It transforms static UML elements, such as classes and operation signatures, into Java code. Because you want to convert both static and dynamic aspects of our model into source code, using the IBM Rational Software Architect transformation is simply not an option.

The other reason for using an intermediary technology is that implementing a direct transformation contradicts the principle of separation of concerns. Imagine if you try to convert an activity into a Java operation without the aid of an intermediary model. The meaning of each element embedded into these activities can vary too much. Also, UML elements often do not directly correspond to a Java element. All the steps required to find the corresponding elements and references in the target model, relate them to the source, and eventually transform the source element to the target elements become part of one single transformation. UML elements refer to multiple Java elements and the same Java element is used by different UML elements, which makes it hard to implement independent rules for single elements. Instead, use a customized EMF-based model and apply it together with JET2 templates.

You can create EMF models in multiple ways. One option is to create an EMF model through the EMF model wizard in Eclipse. This way, you benefit from the tool chain available in Eclipse. Both the diagram editor and the tree-based design view are already provided as part of the modeling framework, so you can design EMF models just as you design UML models. Another possibility is that you import annotated Java classes or XML Schema Definition (XML) files. You can query these models by using an API that is provided.

Creating an EMF model from scratch requires only few steps, which are:

1. Create an empty EMF project and the Ecore model that goes with it.

   The Ecore model could also be created from scratch, but in our scenario, we want to feed the existing UML model into EMF.

Figure 8-11 shows how this step looks in IBM Rational Software Architect.

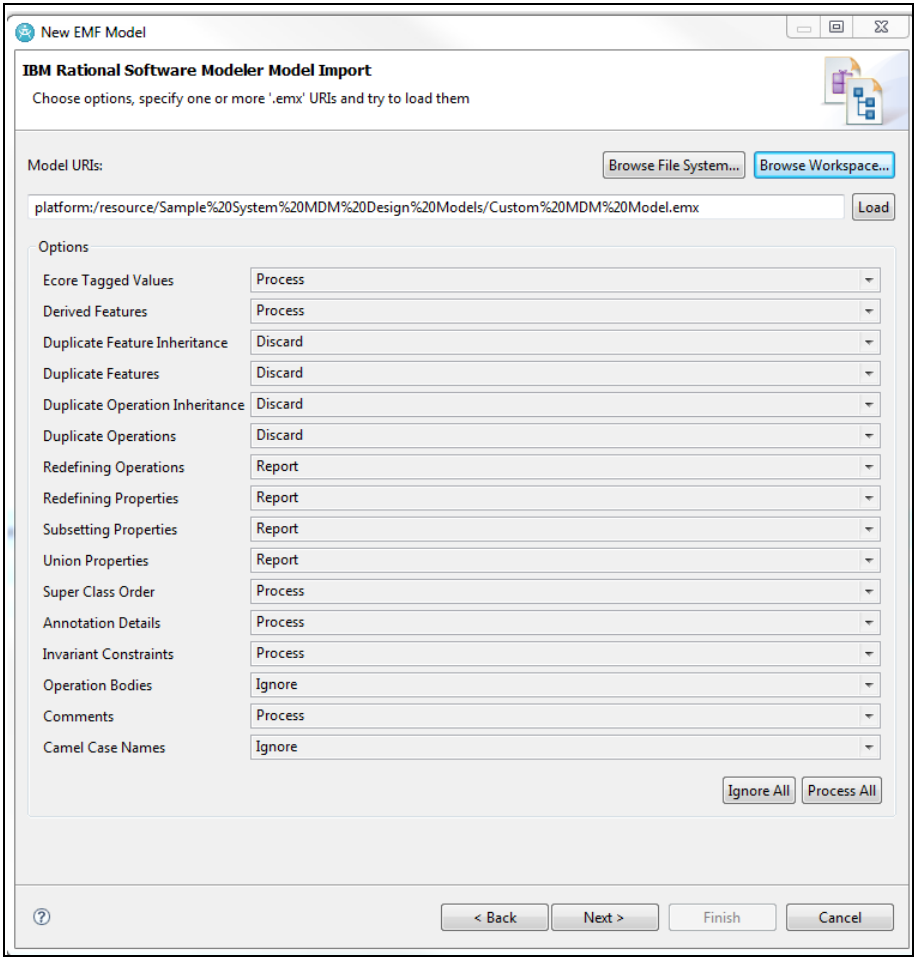

Figure 8-11 Importing an EMX model to an EMF project

2. Create the EMF model.

   You generate the generation model, which is a file with the extension `genmodel`. The generator model (`genmodel`) contains generation-specific configuration properties, while the Ecore model contains the model elements and their relations.

Figure 8-12 shows the sample EMF project and the editor projects that are automatically generated from the generation model.

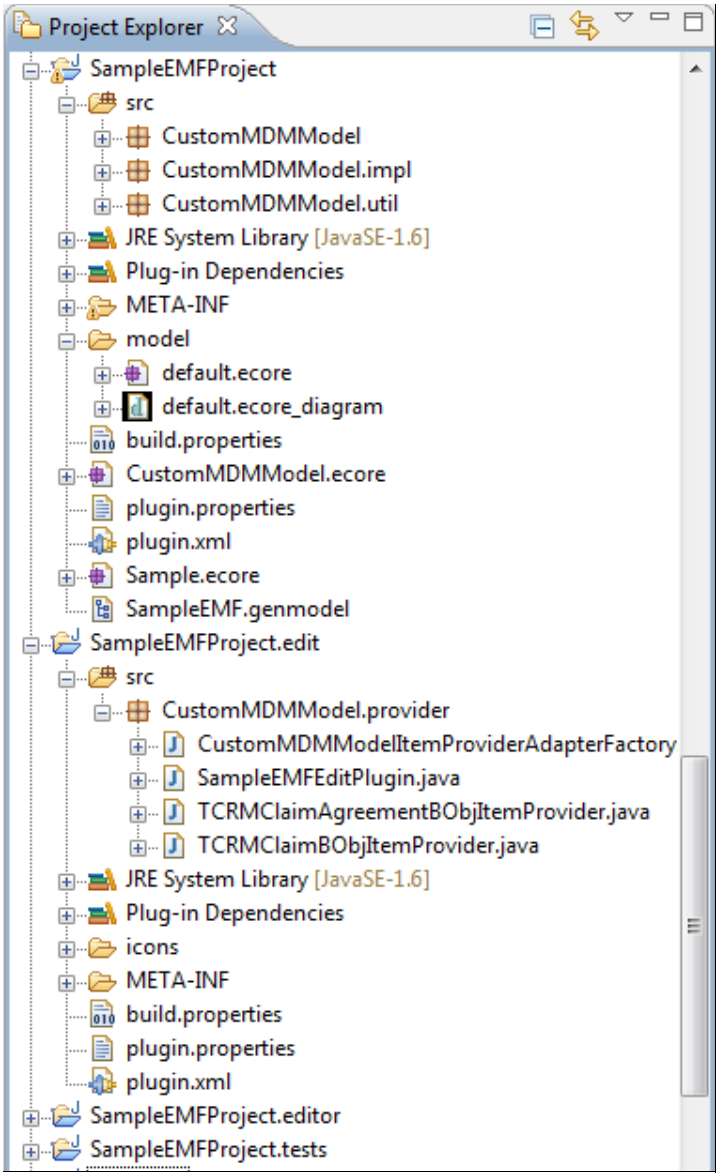

Figure 8-12 Sample EMF project

The elements of an Ecore model look similar to what we know from our UML model. All elements are based on a base type EObject and expose concepts such as classes and operations. You can customize EMF models with EAnnotation elements, which serve as a general-purpose container for many kinds of customizations. EAnnotations expose a source key, a list of detailed entries of key-value pairs, and a list of references. Together with the generated, model-specific API that can be used independently of the Ecore model, this approach provides sufficient flexibility in our scenario.

A Java Emitter Template and code generation

Based on our introduction of the concept of a JET, we can describe how you apply and customize these templates in your tool chain. First, you must create a JET transformation project. During the creation process, you are prompted for an input schema. You can either create a schema or, as we suggest in our approach, take an existing model and associate it with a JET. In our example, the Ecore model described is the model we refer to. In the JET transformation project, this schema file is called `input.ecore`. Apart from this schema, the wizard generates a project structure similar to the one shown in Figure 8-13.

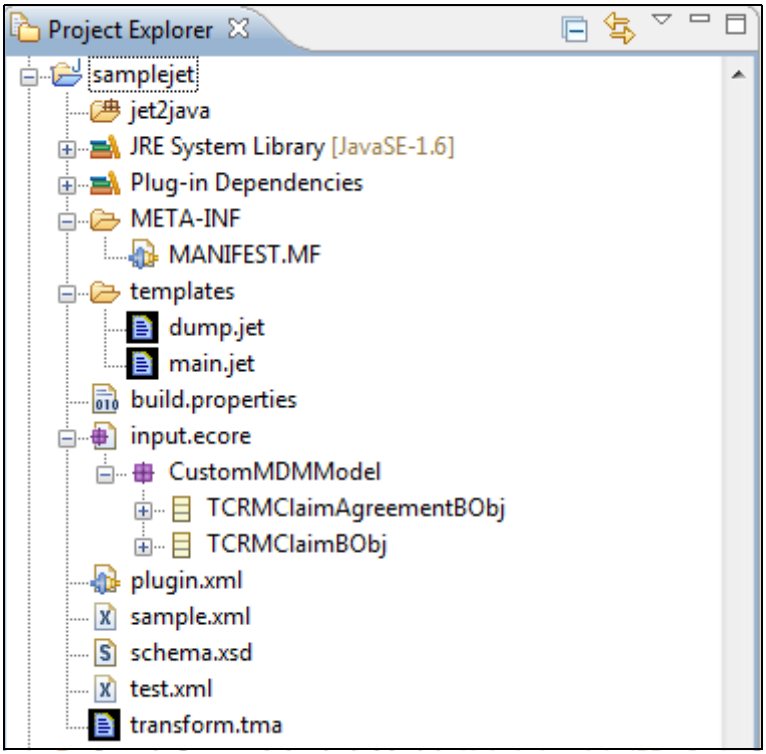

Figure 8-13 Sample JET transformation project

After you create the JET transformation project, verify its settings by completing the following steps:

1. In the extensions tab of the `plugin.xml` file, set the "`Start template`" to "`template/main.jet`", if it is not set to this value yet.
2. In the `templateLoaderClass` field, you must specify the fully qualified name of the Java class that loads the JETs. This class is created by the JET compiler and the field should already be complete.
3. In the `modelLoader` field, specify how the Transformation Input resource, for example, a file, folder, or project, is converted into an in-memory model that the JET transformation can process. The default value is `org.eclipse.jet.emfxml`, which indicates that the resource should be loaded as an XML document. For other model types, including UML models, specify `org.eclipse.jet.emf`. If you do not specify a value, a model loader is selected based on the file extension of the input model, or by the value of the modelExtension attribute. The default value is blank, which is equivalent to `org.eclipse.jet.emfxml`. Because we use an Ecore model, we can specify `org.eclipse.jet.emf` explicitly.

Example 8-3 shows an extract of a JET. As this example suggests, JET uses the same notation syntax as standard JSPs. It uses taglibs, scriptlets, and parameterized expressions. The main difference between a JSP and a template are the XPATH-like references the JET contains. These references point to corresponding model elements that are evaluated at run time.

Example 8-3 An extract of a JET

```
<%@taglib prefix="c" id="org.eclipse.emf.jet2.controlTags" %>
<%-- /selects the document root, which in this case is the EMF Resource
object --%>
<%-- /selects the contents of the Resource.getContents() --%>
<c:iterate select="/contents" var="employee">
Employee: <c:get select="$employee/@name"/> (<c:get
select="$employee/@employeeNumber"/>), <c:get
select="$employee/@address"/>
<c:iterate select="$employee/customers" var="customer">
   Customer: <c:get select="$customer/@name"/>, <c:get
select="$customer/@address"/>
<c:iterate select="$customer/orders" var="order">
         Order: <c:get select="$order/@orderNumber"/>, <c:get
select="$order/@date"/>,  qty: <c:get select="$order/@quantity"/>
</c:iterate>
</c:iterate>
</c:iterate>
```

Using a valid transformation input schema ensures that the input to the transformation is semantically valid. Apart form the input schema, you also find an input schema in the root folder of the JET project. This schema serves to create XML documents that comply with the input structure of the transformation. If you do not need to provide XML files to other systems, you do not need this schema. The actual transformation configuration is stored in a `.tma` file that is also in the root folder of the JET transformation project.

The most important artifacts are the JETs that are provided with the project. The templates can be customized by using exemplars. Exemplars are examples of how the target of the JET transformation should look like. Exemplars are taken from examples or other existing projects in the workspace and can then be associated with the template. The output of these associations is an output action, one of which is shown in Figure 8-14.

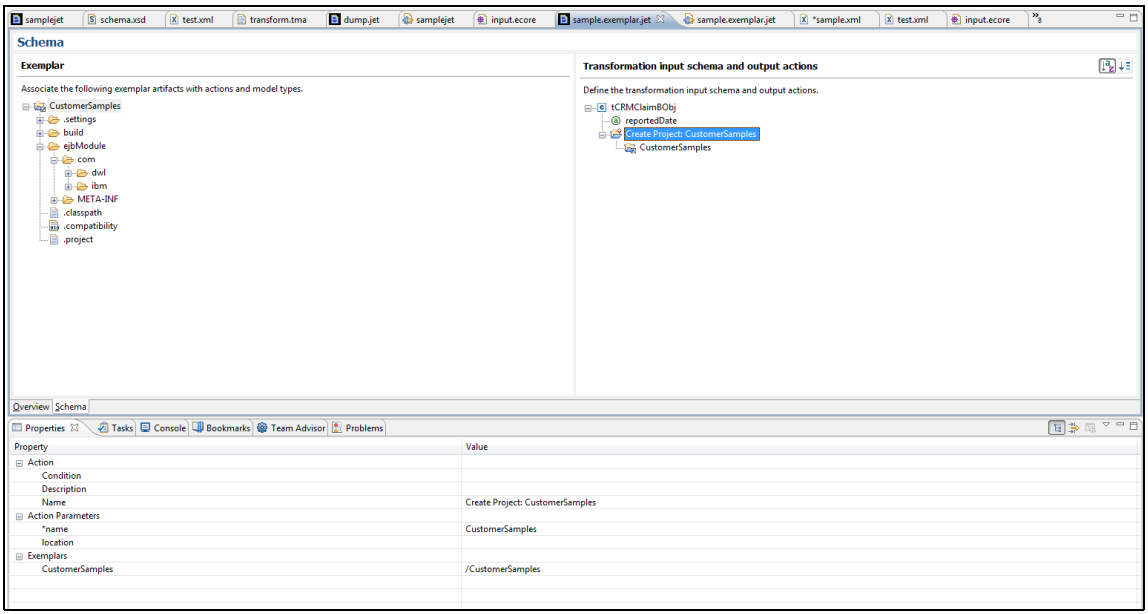

Figure 8-14 The TMA editor and an output action

Initially, these JET actions and templates are copies of the exemplar artifacts. After you associate an exemplar with the input schema, you can customize the input parameters of this output action to include values from the transformation input. These input references are provided as XML Path (XPATH) expressions. A typical variable/attribute reference is of the form `{$variable/@attribute}`.

There is also the support for custom tag libraries. The concept of tags is similar to the tags used in JSP. Similar to JSP, JET can also contain custom tags if they are registered in the `plugin.xml` file. This feature is an important one, because you can decouple logical generation concepts, such as the generation specification for a certain element, and make them reusable.

During code generation, you must ensure that you do not overwrite customizations that are applied to a lower level. This situation applies to both transforming and generating code. Regarding JET-based generations, the merge tool that comes with the generation framework is responsible for handling these situations. Eclipse, or more specifically JMerge, which is the merge tool in Eclipse, achieves the separation of generated and customized code by providing a tag called `@generated`. If you remove this tag or add a "`NOT`" to the tag, you prevent the generator tool from overwriting any changes that you manually apply. The tag is applied to classes, interfaces, and methods. The same principle must be applied to model-to-model transformations. Because Eclipse does not provide a solution for the `@generated` tag in model files, one of your tasks is to define an equivalent in your transformation plug-ins.

Other tools and integration alternatives

We mentioned only the most common, relevant, and frequently used tools within an Eclipse-based transformation and generation approach so far. There are many more alternatives. Here we take a brief glimpse at some of these alternatives.

Eclipse provides an extensive set of generation plug-ins and frameworks. When it comes to code generation, many of these plug-ins and frameworks work with a text-based notation. Another concept we have not described so far are *pluglets*. Because Eclipse is a modular framework, most of the features available in Eclipse are provided as features with their respective plug-ins. This approach has some shortcomings:

- You must deploy additional features before other projects or even other team members can use them.
- Developing plug-ins takes time.
- Maintaining different versions of plug-ins is difficult, because all of the dependent libraries must be available in the correct version, too.

These reasons are why *pluglets* were introduced. Pluglets are Java applications that are used to make minor extensions to the workbench in a simple and straightforward way. A pluglet can be tested in the same instance of the workbench, which is an advantage over plug-ins.

8.2.3 The transformation and generation process in a master data management project

This section describes how the basics of the transformation and generation process are implemented in the context of a master data management solution implementation. This information applies only to projects based in InfoSphere MDM Server, but similar challenges exist in other master data management or enterprise integration projects. Applying the model-driven strategy to a master data management project requires deeper understanding of the product and how you can integrate these components into the overall transformation and generation process.

Figure 8-15 shows where transformations are used within our model.

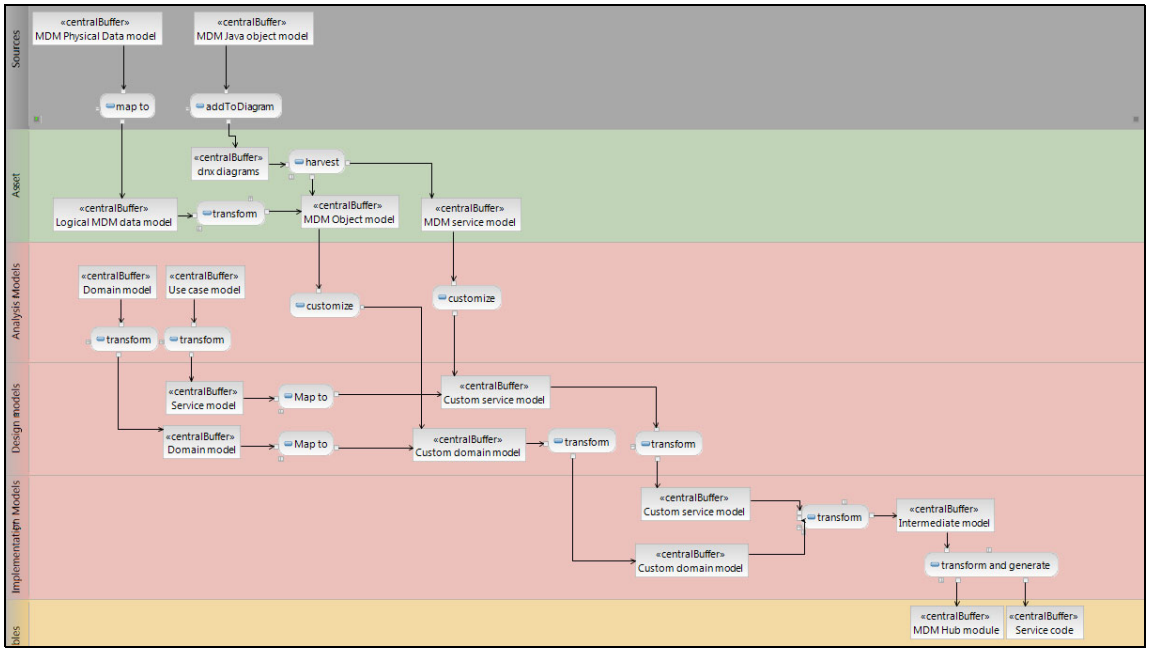

Figure 8-15 Transformations in the metamodel

The development of an InfoSphere MDM Server based application means that you must customize and extend an existing product that provides hundreds of services and an extensive domain model. In addition, this product comes with its own tool, InfoSphere MDM Server Workbench, that provides transformation capabilities that you require. The following sections outline the tasks involved.

8.2.4 Creating model-to-model transformations

The transformations between the analysis and design model and the design and implementation model are probably two of the most complex parts of the modeling process. Fortunately, you do not have to generate a domain model from use case analysis objects, but can directly use the domain model we created in the analysis phase. Also, because you already work with activity diagrams in the analysis phase, you can simply reuse and enrich these activities in the design model. Nonetheless, the transformation between the analysis and the design model is not a simple one to one transformation. Instead, you need specific transformation rules, for example, for the stereotypes, we applied exception handling and time slices at the analysis level to adhere to the design guidelines on that level.

You could use a pattern catalog at this stage, but this action is beyond the scope of this book. The transformation between the analysis and the design model is a customized model-to-model transformation that is concerned with applying design concepts in the form of stereotypes to the design model. Stereotypes might contain only a certain subset of UML elements, so in some cases, you are required to provide additional information with the stereotype. That information becomes part of the transformation. Although the transformation does not strictly require a separate model, it might be useful to introduce one such intermediate model for abstraction and separation of concerns.

8.2.5 Creating LDM-to-UML transformations

Some source systems might provide documentation only in the form of a physical data model. If so, you must apply an LDM-to-UML transformation to this model before you can use it in your context. The steps are the same as for the master data management data model. You take the physical model file, map it to a logical data model, and transform this model to a UML model. Apart from this scenario, we do not use the LDM-to-UML transformation often, mainly because we do not pay much attention to the logical data model in our approach. The UML domain model and the physical data model are much more important to our model.

8.2.6 Generating master data management source code

After you create and customize the design model, you must create an implementation model. This model is still a UML model. The main goal of the transformation between the design model and this UML-based implementation model is to transform the design concepts into information that can be used by InfoSphere MDM Server. The details of this mapping are described in the implementation model, which is why we focus only on the transformation principles here.

Assume that you want to transform a validation to a master data management component. Whether this validation is an object-related, internal one or an external one determines whether you either use the validateAdd/validateUpdate methods of the respective business object or use a behavior extension. So, depending on the location of the activity and the stereotype it uses, the transformation itself knows whether it needs to transform the validation to an internal master data management validation or an external behavior extension and a separate validation class.

Here is a brief and incomplete list of some of those components that we transform in the implementation model:

- Lifecycle

  The state machine must be mapped to the status concept in master data management. Also, the lifecycle triggers must be transformed to events, the receivers to listeners, and the lifecycle machine to a behavior extension.

- Time slices

  The implementation model is also the place where the cardinalities of the design model must be adjusted according to the stereotype of the design attribute. If it is time-sliced, it is transformed to a list.

- Exceptions

  In respect to exceptions, the design model is almost complete. Nonetheless, in the implementation model, the error handling must be mapped to the error concept in master data management. Thus, you must provide a customized DWLError class.

Apart from business exceptions, the implementation model also needs to include technical exceptions. How these exceptions are handled is not part of the design model at all. So, this situation is one of the few areas where the complete “domain” knowledge is part of the implementation model and the transformation. By default, you implement a default handling here and provide a technical exception that wraps those errors thrown by the persistence layer and internal master data management transactions. Because technical exceptions are not handled by the business logic, providing this logic does not need to be part of the domain design.

All of these artifacts can be expressed in terms of UML elements. The transformation uses stereotypes. The result of the transformation is another stereotyped UML model. This model is a platform-specific model, as it depends on InfoSphere MDM Server. This step from the design to the implementation model is also a good place to add validations and checks for the constraints we imposed on the model. For example, we want to ensure that a composition is used correctly.

You must decide which design elements can be implemented in the transformation between the design and implementation models. If you want to rule out the possibility of adding incorrect elements, you must establish a clear design guideline and provide alternative design solutions. If you do not prevent designers from using constructs that are not supported in the target programming language, then you must to find a way to deal with the constructs before you generate the source code.

This situation is where transformation comes into play. In the transformation process, you can decide how to proceed when you run into such an issue. You might not always be able to handle these scenarios automatically every time, but you can ignore an element and generate a warning or make the transformation fail.

After you apply the next step, you leave the world of UML models and create a EMF-based model file. You have two options for transforming the UML implementation model. You can either create an EMF project from this model and follow the process described in the previous sections of this chapter or map parts of the implementation model to the artifacts that InfoSphere MDM Server Workbench uses.

Figure 8-16 shows the master data management generation process.

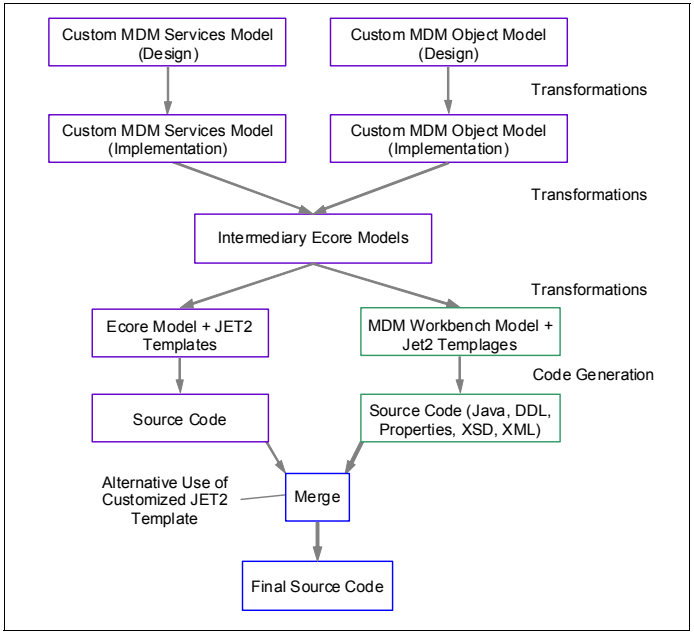

Figure 8-16 The master data management generation process

At first sight, it does not seem to make sense to use InfoSphere MDM Server Workbench at all. If you create a Hub Module model in InfoSphere MDM Server Workbench, the wizard generates an `mdmxmi` file that is the model file of the Hub Module model. In our example, you use this model to create additions and extensions. You cannot model the elements through the hub module UI, but you can use the UI to generate all source artifacts, including Java code, DDL scripts, properties files, and XSD snippets. However, InfoSphere MDM Server Workbench supports only a subset of the elements that you need to implement.

In addition, even if you can generate a certain element, it does not contain the actual logic you designed. For example, although a behavior extension can be modeled in the workbench, the actual output file contains only the method skeleton. It does not seem to make sense to model additions and extensions in the workbench after creating them in the design model; integrating an additional technology into the development process usually makes it more complicated. If you chose to use the functionality provided with InfoSphere MDM Server Workbench, you effectively maintain two tool chains in parallel: one responsible for the artifacts that can be expressed through InfoSphere MDM Server Workbench, and another one for the other artifacts. As a consequence, you must merge these two streams and make sure that both of them are always synchronized.

Despite all these complications, using the features of InfoSphere MDM Server Workbench has several advantages. Using InfoSphere MDM Server Workbench, you can ensure that you are not affected by changes in the underlying implementation. If the implementation of the persistence layer changes, you can still rely on the InfoSphere MDM Server Workbench model file and do not need to change the code of your generator; the workbench already provides a substantial part of the generation. Instead of dealing with DDL scripts for additions, you can focus on the service/activity logic and how to translate it into Java code. You can use InfoSphere MDM Server Workbench features wherever this translation is appropriate, and implement a delta generator for the dynamic artifacts.

Now we take a closer look at the technical implementation. Assuming that you use the mixed approach we described, you must consider two transformation targets, one of which would be a standard EMF project and the JET transformation project, and the other one being the `mdmxmi` model. Because you are supporting two different targets, it becomes even more important that you use an intermediate model. Then, the transformation process looks as follows:

- Use a custom transformation plug-in that uses a three-step process to transform the UML implementation model into source code:
  a. Transform the UML model into an intermediary model that is based on a custom Ecore model. You can perform this transformation in the memory. There is no need to persist the intermediary model.
  b. Transform the intermediary model to its target representations. In our example, we produce both an in-memory `mdmxmi` file and the target Ecore model.
  c. Use a merge routine that works similar to JMerge to merge the in-memory mdmdxmi model into the existing `mdmxmi` module file. This step is not strictly required, because our assumption is that you do not modify the module file you generated. It can be overwritten every time you start the generation process.
- Generate workbench code with the aid of the updated `mdmxmi` model. We do not perform this step manually in the workbench UI in our example, but through the plug-in we provide.
- Generate the other code artifacts and merge them with the workbench artifacts.

8.2.7 Other considerations

The transformation processes described in this chapter provide you only with a rough overview of what you must do and what happens internally as a result of your actions. The detailed specification and implementation of the generation tool is beyond the scope of this book. For more information, see “Related publications” on page 559. Nonetheless, some details are worth mentioning:

- The InfoSphere MDM Server Workbench plug-in is based on the Eclipse Modeling Framework (EMF). Instead of decoupling the generation process for InfoSphere MDM Server Workbench artifacts, you can modify the existing JET2 template of the workbench plug-in according to your needs.
- You can use an intermediary model for other output formats, which is an additional advantage of using this model.
- The EMF tool chain already includes support for Java docs, and in addition, you could transform documentation available in the model or text documents, and make them part of the source code or a model file.
- The intermediary model also serves as an additional entry point. Instead of feeding a UML model into the process, you could also use a persisted version of the intermediary model.

One of the most important aspects of the transformation and generation flow is to decide where to implement which step of the transformation process. You want to avoid adding technical details too early in the process, but you also must ensure that you create relatively lightweight and maintainable transformations.

Another reason for not implementing everything on the lowest possible level is a technical one. Some abstractions are simply not available at a lower level. For example, if you evaluate a stereotype in a JET, this situation means evaluating an XPATH expression you apply to a model instance.

9

Deliverables

This chapter provides an overview of the types of final deliverables for a master data management solution. It includes information about reporting on the models and artifacts that are created through modeling and outlines the need for frequent validation of the models and their dependence on traceability information. In addition, this chapter positions the requirement for impact analysis and how the impact analysis depends on the traceability that is embedded into the models.

9.1 Creating reports

Generally, every project needs to create reports. However, few projects include the ability to generate customized reports. In addition, generating reports directly and automatically from other project artifacts is often underestimated and not used to its potential. Reports are often created on an ad hoc basis, where only the needed information is gathered and the results are recorded in a separate medium, such as a spreadsheet.

In some cases, this type of process can meet the immediate need, but the information is not derived directly from the model. The content of the reports cannot be traced back to the actual project results, because the content was created and extracted independently of existing tools artifacts.

Thus, creating reports can be both an error-prone and repetitive process. Existing reports are often reused, but templates are not developed as reusable data sources. As a result, project management spends time keeping presentations and spreadsheets up-to-date because the information to which the presentations and spreadsheets refer is constantly changing. These issues also make analysis and design problematic. With no link between the models and the reports, project managers cannot discern the current progress of the project.

In such cases, creating a report involves a level of trust instead of hard data. Extracting information from a project and providing the data that is needed for reporting to a project manager is critical. Time constraints and large, complex solutions force project management to consider better processes for generating reports.

The following sections provide an overview of and guidance for preferred practices for generating reports.

9.1.1 Definition of reporting

In the context of this discussion, *reporting* encompasses the design and creation of documents that present customized information about certain design artifacts. Reports provide views on the model files, enable different users to discover and trace relationships between the elements, and make information accessible to those users who do not have any expertise with modeling tools.

The most important aspect of reports is the separation of content and layout. Although content evolves and changes over time, the design of a report usually does not. A good reporting tool helps you avoid repetitive tasks, such as adjusting the design of a report due to changes of the underlying data.

Also, regardless of whether information is presented in a spreadsheet or a presentation, the source of information is the same. For example, if a designer wants to know the current mapping between two logical models, a simple table-based report will suffice. Alternatively, a project manager who is preparing a presentation regarding that status might embed more advanced layout elements, such as pie charts or other diagram formats.

A reporting tool must support different graphical elements and also different output formats. The most common output formats are Microsoft PowerPoint (PPT) presentations, Microsoft Excel (XLS) spreadsheets, and Microsoft Word (DOC) documents (and the corresponding OpenOffice formats). If you want to make sure that a report cannot be changed by the recipients, you can use a Portable Document Format (PDF) file.

An important aspect of reports is the separation between the design view and the actual output of the report. After creating the report design, anyone should be able to generate a report instance based on current model data. This implies that the use of a report template against the data and artifacts in which you are interested must be simple, and should hide unnecessary and repetitive reporting activities from report users. Also, adding further fields to the source should not affect the function the report. A report should continue to present the same information, even if the information changes form. For example, a report based on a relational database table should produce the same information even if a column is added to the table.

A good tool to help you achieve this type of design with your reports is the Business Intelligence and Reporting Tools (BIRT), which ships with Eclipse and that represents the reporting infrastructure for IBM Rational Software Architect and IBM InfoSphere Data Architect. BIRT includes several plug-ins. Eclipse contains a version of BIRT and comes with basic templates. However, IBM Rational Software Architect includes additional templates and also a number of predefined reports. These additional templates and predefined reports allow a quick and easy setup of a report. If you have no resources allocated to design and generate reports, you can benefit from these predefined reports. BIRT reports are customizable. In the sections that follow we discuss the design of additional reports based on the models and structures suggested in this book. Your design approach depends largely on the complexity of your models and the information that you want to retrieve from those models.

Note regarding BIRT: BIRT is not intended as a replacement for complex business intelligence solutions. Instead, it provides a means to design and generate reports of model files, relational data sources, and other artifacts. In the context of the design models described in this book, reporting provides structured information about design model files, data models, and mappings. Regardless of whether the underlying source is a file that represents a logical data model or a standard UML model, BIRT can extract information from these models and present it in tables, charts, and so forth.

For the purposes of the discussion in this book, BIRT is a good choice because it is built into the tools used for modeling purposes and is easily accessible. It also provides a powerful and flexible solution and creates the reports needed in daily activities.

If you have comparable reporting tools already in use, such as the IBM Rational Publishing Engine that is part of the Jazz platform, you can replace the BIRT report generator with your equivalent tool. The key consideration is the type of data source. All IBM InfoSphere Data Architect and IBM Rational Software Architect models are based on XML definitions. IBM Rational Publishing Engine can work with any XML format and can generate reports for the same output formats. You also get additional advantages, such as the ability to assign template files, such as a Microsoft Word template (`.dot`) file, to reports, which is currently not supported with BIRT in the version embedded in IBM Rational Software Architect and IBM InfoSphere Data Architect.

The remainder of this discussion focuses only on BIRT.

9.1.2 Using BIRT to create reports

With BIRT, you can create stand-alone reports or reports that are part of a reporting library. BIRT also allows you to use simple reporting templates, which typically suffice for usual reporting requirements.

These features are important in the context of master data management modeling because many of the reports are based on similar layouts and might even share the same data. To support this kind of reporting, BIRT provides the following features:

- Report elements
- Report templates
- Folders
- Libraries
- Data sources

- Data sets
- Report parameters

The structure of the reporting project should reflect the structure of the model projects, as Figure 9-1 shows, based on the example project discussed in this book. Depending on the complexity of your project and related models, you can either create a single reporting project that separates the reports by using folders or create separate report projects for all model projects, which is useful in complex configurations. For most projects, you can create one reporting project only, as depicted in Figure 9-1.

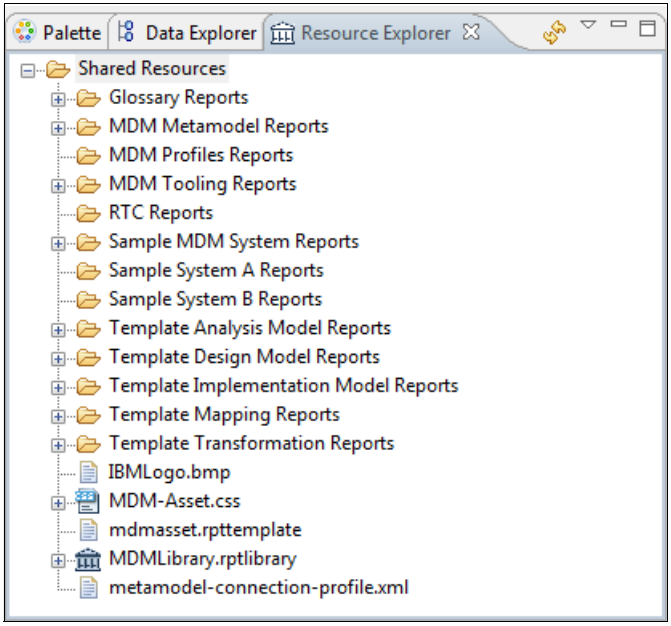

Figure 9-1 Structure of report project

Apart from the core model projects, the report project might also contain management reports, test reports, quality reports, and other types of reports. The types of reports that you use depend on the requirements within your project.

9.1.3 Creating reports

BIRT supports several data source formats, including relational databases, flat files, and XML files. BIRT also provides support for more specific formats, such as Eclipse model files and Unified Modeling Language (UML) files. For best results, concentrate on the core formats that you use most frequently, because you need to understand the formats that you use to use those formats in reports.

BIRT can also connect to Jazz and namely the IBM Rational Team Concert product. You can use this tool for source control and for project management activities to create detailed reports about the project status, open defects, and so forth. The project structure shown in Figure 9-1 on page 465 reflects this scenario in that it contains a separate folder for IBM Rational Team Concert reports.

Note: A detailed discussion of the features of IBM Rational Team Concert is beyond the scope of this book. Refer to related publications for more information:

http://www.ibm.com/software/rational/products/rtc/

This discussion focuses on the Eclipse Modeling Framework (EMF), UML, and Extensible Markup Language (XML) formats. You can use these formats to generate model reports. The choice of a format depends on the model that you query and sometimes on the report usage scenarios.

Data sources

The main advantage of using UML data sources is that they expose UML elements and attributes. Report designers familiar with the UML syntax can create simple UML reports without a learning curve. Eclipse automatically provides the most important UML name spaces or metamodels together when you create the UML data source. It also supports the inclusion of further profiles, which becomes important if the report design must reflect a domain-specific language.

Figure 9-2 shows the BIRT data source editor.

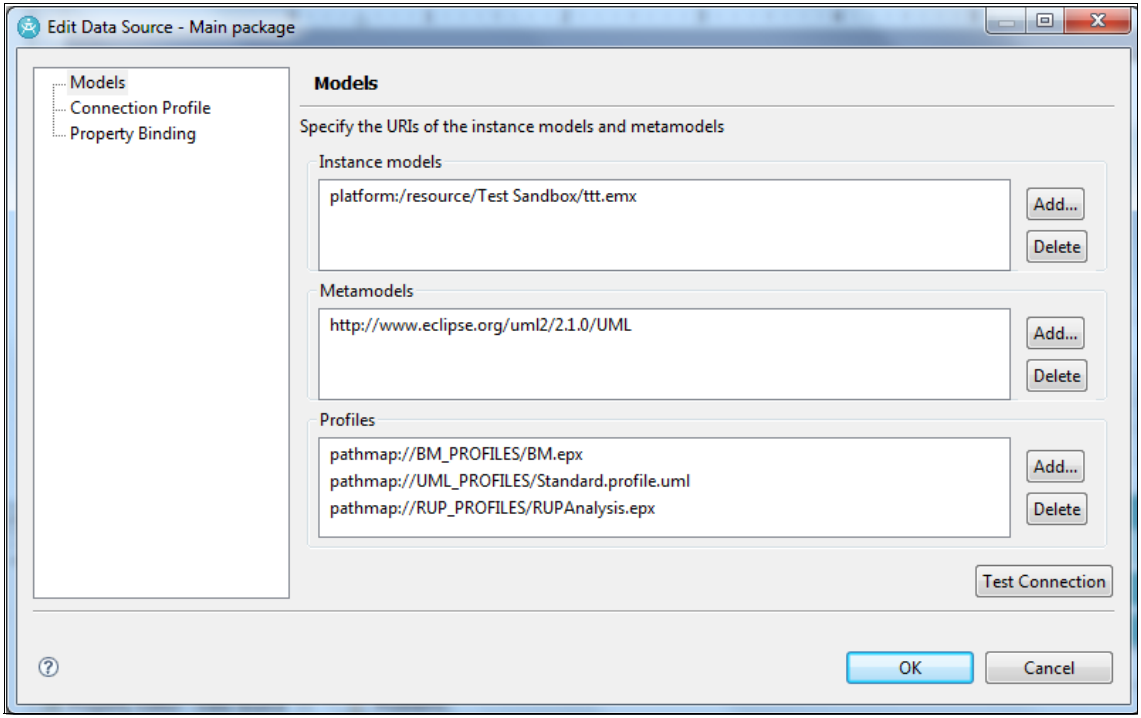

Figure 9-2 Model instance versus metamodel

The distinction between the underlying metamodels and the actual model instances is important. Apart from the packages that are registered with the workspace, you can provide additional metamodels. These additional metamodels enable you to use an alternative path hierarchy in the row mapping. For example, if the metamodel exposes a custom element called *MDMService* and if this element contains two subelements, one subelement as the header of the message and the other subelement as the actual body of the message, you can select exactly those elements from an instance of a service model that are master data management service elements.

Note, however, that you need to register custom metamodels as *packages*. You can also reference resources from other projects, but registering resources as packages guarantees that references depend only on the path map and not on the installation of certain projects or physical paths in the file system. Thus, *path maps* provide a level of abstraction between resources and physical storage. Use path maps instead of hard coding links to the file system, because moving links that are hard-coded in reports breaks the report design and requires rework. The same logic applies to the profiles that you use within data sources.

Another feature of a data set is the abstraction from the actual physical source. After you establish a connection between a data source and the underlying data store, you do not need to establish the connection again, enabling you to provide sample reports and templates that can be used immediately after changing the underlying connection. Also, the connection itself can be externalized by means of *connection profiles*, which requires that you define connection properties within a separate file and create a reference to this profile in the data source.

Report designers can use this feature to provide the best separation between model instances and report definitions. In addition, reports are impacted less by name changes of the model files against which to report. You can create a connection profile, using a data source wizard, which allows you to either enter the data source location directly or provide a connection profile store. If you select to use a profile store, you can enter the connection parameters directly. Also, you can enter more than one physical file at the same time.

Figure 9-3 shows where you can define your connection profile. If you specify properties, you can also overwrite connection parameters at run time. This feature is used mainly with report parameters.

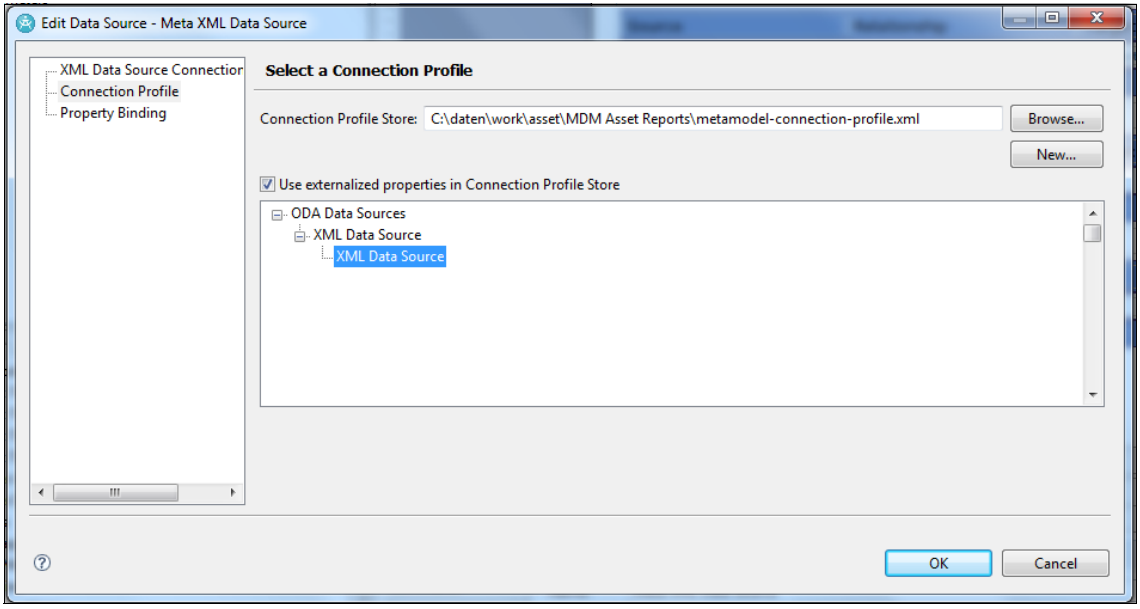

Figure 9-3 Define a connection profile

Defining a data set

After you create the data source, you need to define a data set based on that data source. *Data sets* serve to provide a customized view on a data source and enable you to select attributes from the data source, transform these attributes by means of predefined or custom functions, sort them, rename them, and so forth. In the case of UML data sources, a considerable amount of these operations is exposed in a UML-like manner.

You can create UML data sets by navigating through the hierarchical view of UML elements and attributes and by using the built-in function library. For example, navigating through a class diagram means that you have to select the class node from the hierarchy and identify the attributes that you want to include in the report. The class (uml:class) corresponds to the row of the report, whereas the attributes correspond to the columns.

In the row mapping, you also need to decide which UML type to use. The examples in this book use a uml:class type. Optionally, you can apply a filter that returns only elements with a certain stereotype. Selecting attributes in the column mapping corresponds to navigating through an XML file through XPATH. The wizard supports these path expressions, and the navigational functions and attribute qualifiers also.

Further examples for navigation against a model instance and a metamodel, which both represent hierarchical navigation, are shown in Figure 9-4 and Figure 9-5 on page 471.

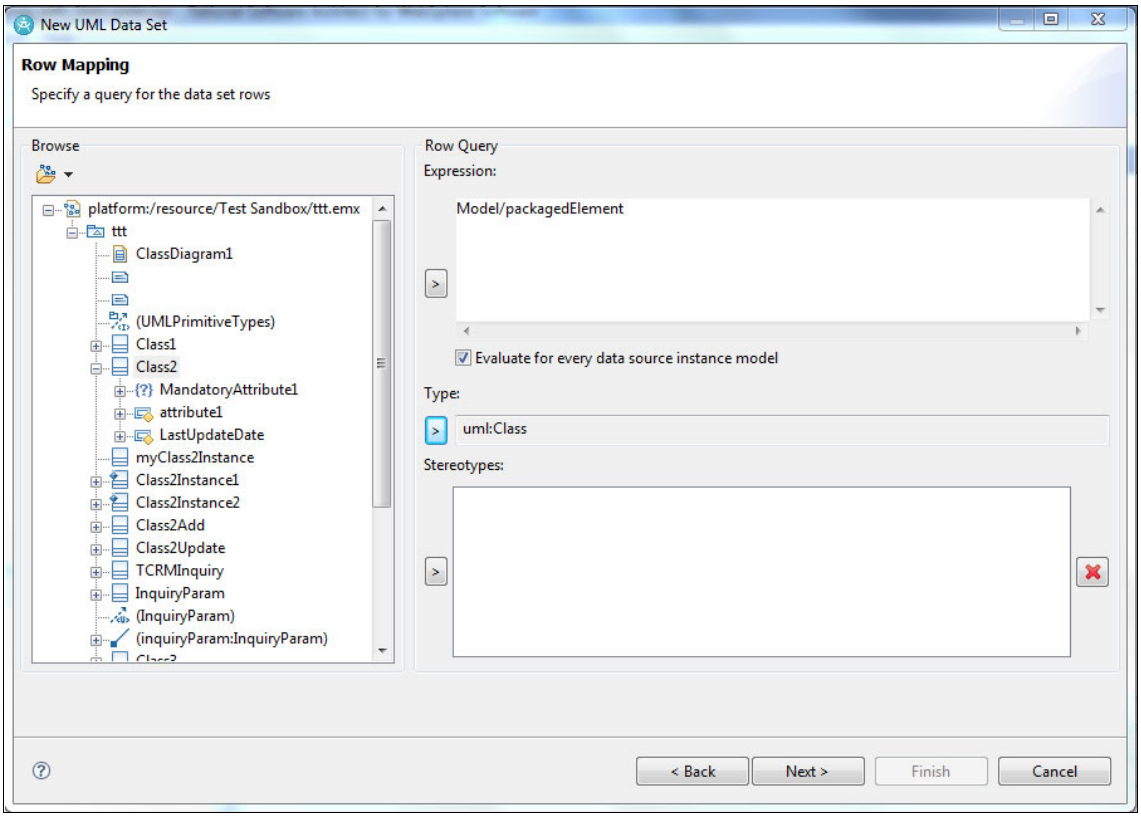

Figure 9-4 Navigating on the model instance

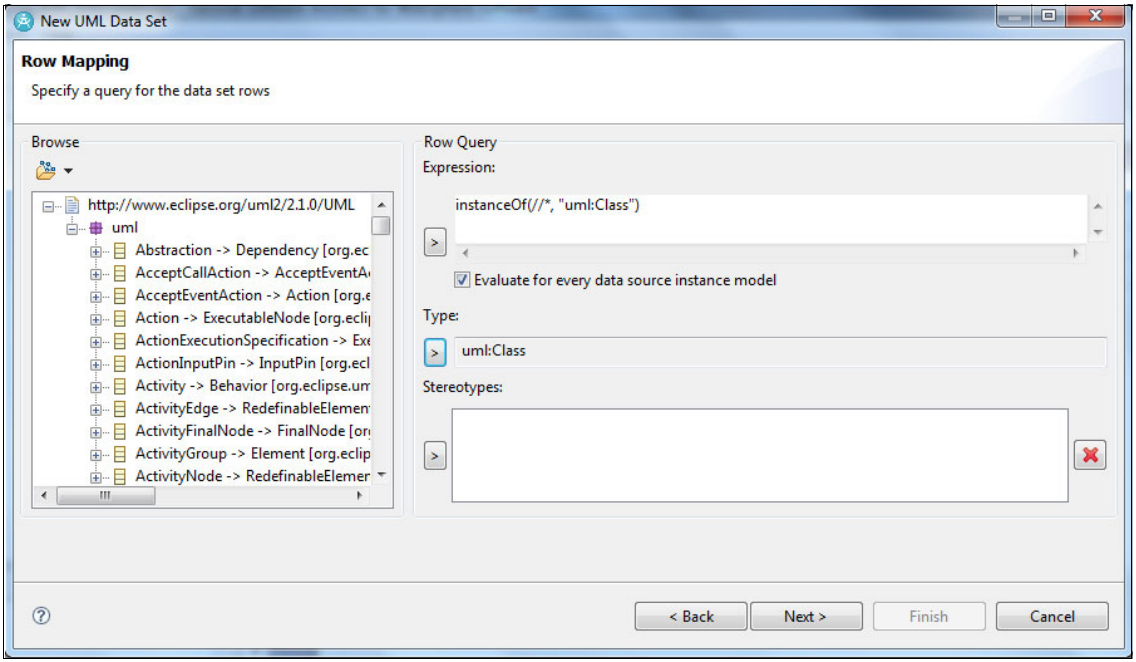

Figure 9-5 Navigating on the metamodel

Report parameters

Data sets might also contain report parameters. These parameters enable report users to filter report data dynamically while running the report or to provide dynamic content without the need to modify the report template first. The parameters provide better reuse of existing reports. For example, if a user wants to limit a report to a certain business domain while exposing the domain name as a report parameter, the layout and the structure of the report should not depend on the business domain but rather can serve any domain.

Associations between the models

The presentation of relationships between model elements is also important. Reporting on traceability is just one aspect that reveals the associations between model elements and other artifact elements. You can easily distinguish relationships between similar elements in a model, for example, if a class depends on or is derived from another class in the same model.

Discovering associations between model artifacts and modeling disciplines and stages is more of a challenge. An even greater challenge is the detection of transient relations, for example, if you need to identify the physical database tables to which you write from a selected use case or for each use case. In this case, there is no direct relationship between these models. Instead, they are connected using only an underlying domain model and the corresponding data mapping between the domain model and the physical data model.

Defining names

Remember: Reports cannot serve as a replacement for traceability. Reports can use only the information that is provided through traceability. You cannot easily create reports across models if the parts of a model are not traceable. Similarly, reports cannot serve as a substitute for a thorough dependency analysis. Reports reveal only those facts that are implemented into the model.

Provided that you have defined the required associations between the models, you can create joint data sets in the report creation process. Just as in an SQL statement, you can either perform an inner or an outer join on the data sets. You can use aliases and display names for the output columns, which improves readability, because by default BIRT concatenates the names of the data sets that you want to join, making the original names of the columns long and difficult to read.

Figure 9-6 illustrates the importance of names.

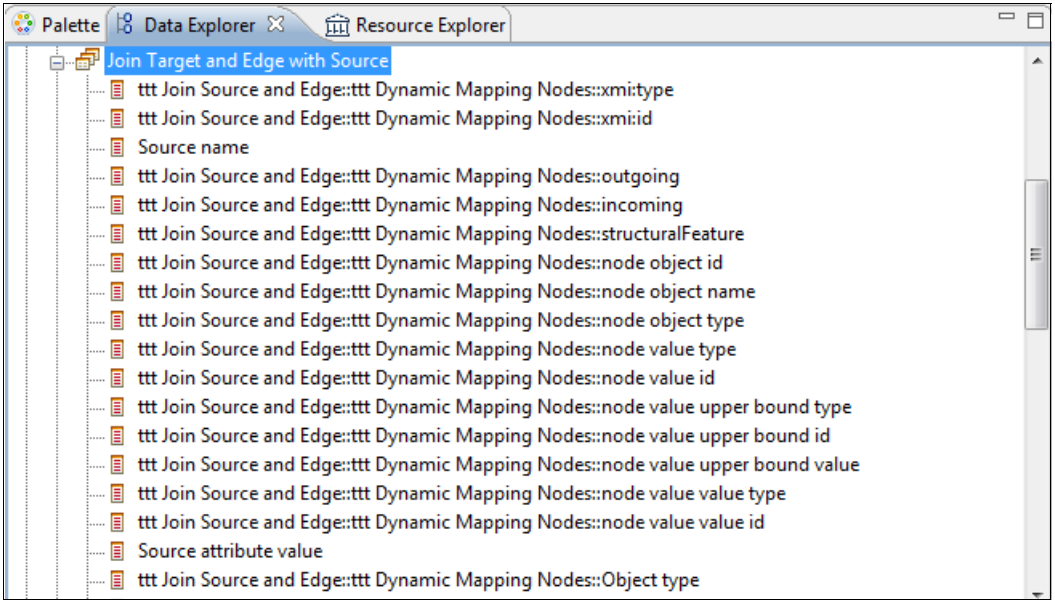

Figure 9-6 Column names of joint data sets

It is a preferred practice to define new column names for the joint sets. Use a data set name that corresponds to its role in a relationship. For example, an intermediate data set that selects partitions from a model can be named *Partitions that contain*. Then the right side of the relationship can be named accordingly. The joint data sets automatically show these names, as shown in Figure 9-7.

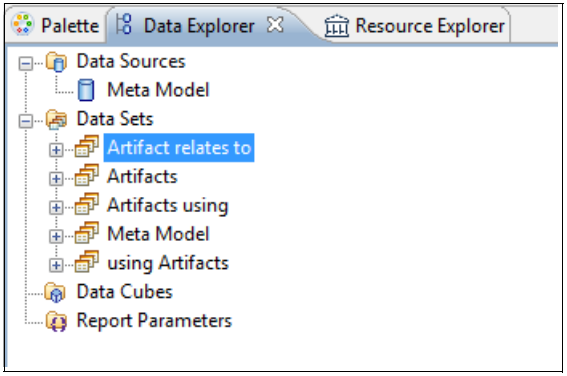

Figure 9-7 Naming convention for data sets

Try to avoid the definition of filters and parameters on different levels of data sets. Instead, use the raw data for as long as possible before applying any filters or functions to it to help you differentiate between the data retrieval and the data preparation process.

Although UML data sources provide an easy means of creating reports of UML models, associating information from different model parts is not straightforward. For example, information about classes is kept separate from information about the stereotypes that are applied to those classes. Even with built-in functions that Eclipse and IBM Rational Software Architect provide, obtaining this information is a challenge, and you might resort to using the underlying metamodel.

This type of model-driven development (MDD) is represented by EMF data sources, which operate on a lower level. Using EMF data sources requires a deeper understanding of the internal model structure for Eclipse. If you are not proficient with EMF, this task can be daunting. Your next best option is to use XML data sources. This description applies only to EMF-based models. Data models and data mappings are not based on that structure, but instead contain their own equally structured information. However, you can also report against these models by means of an XML data source.

Creating reports based on XML instances

Creating reports based on XML instances primarily means using XPATH expressions. Before you can create a data set, you first need to gain insight into the model instance with which you want to work. In case of an XML-based model, you have to be familiar with the underlying XML schema.

First, define the row mapping such that it is applicable to all matching elements on any hierarchical level of the XML model instance. More precisely, use expressions such as `//` or `instanceOf` to make sure that you find any instance of the element type for which you are looking. The most common expression that you will use is `//packagedElement`. This expression can be anything ranging from a class to an activity, but can also contain other packaged elements.

Be aware that the model file contains both diagram descriptions and structural definitions. Relations within the model file are established through generated IDs, making it a bit more difficult to tell in which part of the model the structural association is actually stored. It is a preferred practice to add comments and unique names, because this information is valuable to both designers and report users and also because searching for these elements is easier.

In some cases, you might need the parent element of an element selected in your report. BIRT supports XPATH expressions such as `../` enabling you to determine the parent element.

9.1.4 The report layout

Now that you understand how to retrieve the data on which you want to report, this section describes the layout of the reports. Reports include *data elements*, which are bound to the data sets that you created. Report data elements encompass elements ranging from simple tables to pie charts.

For model reports, you want to expose visible information and also hidden information in or between models, such as traceability links, in a representative structure using the table element. Tables can contain both groups and nested tables. In general, it is better to separate information into more than one table rather than to squeeze all the information into a single table. Figure 9-8 illustrates a comprehensive but simple and readable layout.

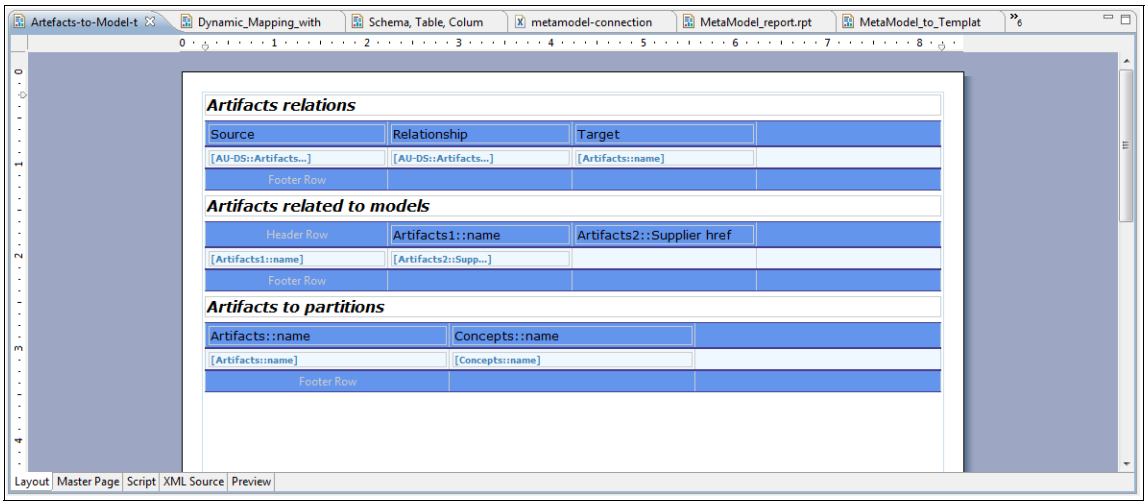

Figure 9-8 Basic layout of a report

BIRT supports *master pages*, which can provide a consistent look and feel across all pages in a multi-page report. For example, you can create consistent headers and footers across pages. You can also use master pages to apply a corporate identity and to format the resulting documents according to your required standards.

BIRT also supports the use of *libraries*. Libraries are containers for all kinds of report elements and data sets. Every time that the content of a library item changes, these changes are reflected in all reports that use the library item. When using similar model files and elements, use libraries wherever possible. For example, you can provide a base library and possibly add libraries for the business domains that you have defined.

With BIRT you cannot apply external Cascading Style Sheets (CSS) or Microsoft Word `.dot` template files. Rather, the CSS is imported and stored within the report design. BIRT also does not support all of the features that a style sheet can provide. Be sure to add sufficient time for the adoption of an existing corporate style sheet.

Apart from raw data, reports can also contain images. You can embed model elements directly into the report layout by means of the built-in functions that BIRT provides. For example, you can create a report that encompasses a list of all custom design elements or conventions for a set of elements. Rather than describing the content and the graphical layout of the element, it is usually much easier to display the element directly as part of the report.

9.1.5 Special notes and recommendations

If you want to create a report of a physical data model, you can either use the corresponding BIRT template or create your own XML-based report. For best results for standard reports that are based on a single source or model, use standard report templates and adopt those settings that are dependent on the corporate design.

Newer versions of IBM Rational Software Architect provide additional reports. You might also have your own library already. Many existing libraries also work in IBM Rational Software Architect V7.5, because this version is compatible with IBM Rational Software Architect V8 at the time of writing. If your reports do not work without modification, it often is just a matter of removing specific functions, such as getFKDetails in the case of a data model report.

Sometimes you are also required to create separate data sets for different entities. For example, you can create separate reports for columns, tables, indexes, and so forth. In some cases, it is not so obvious how the model structure is actually stored.

A good example of a model relationship that is used only through its graphical representation is a partition that is used to draw swim lanes. To retrieve the necessary information for this model, you first need to look at the XML format and limit the element search to those elements that are part of a `//group`. Then you can filter on those elements that are of the uml:partition type. You can find specific object instances and actions of the metamodel flow within these nodes. From a conceptual point of view, the approach is to search within the scope of a specific, rather than generic, structural element and then to filter on the result by means of UML types. A little more effort goes into creating reports on dynamic relationships. These can usually be traced back only by evaluating edges, for example, on an activity.

Some cases require attention beyond the scope of the model files. Traceability reports often require that you take the traceability log into consideration. Although this approach is not a preferred practice in general, you cannot avoid this approach if you plan to report on transformations as well.

9.2 Model analysis

Model analysis represents one step to towards consistency across the various models that you create. Model analysis is important due to the complexities of numerous models that are working in unison towards one larger goal. However, you need to validate even a single model. In this context, *validation* refers to ensuring that the modeled reality is correct and that it satisfies all requirements and expectations. You can validate a model using the following methods:

- Carrying out walkthroughs
- Performing simulations
- Verifying adherence to metamodels

Walkthroughs are a good way to conduct model analysis while also fostering strong relationships with the adjacent teams. The team that is responsible for creating the model is also responsible for guiding other teams, typically the teams that are dependent on the delivered model artifacts, through the model that you created. While this team explains the model to other participants, the participants can ask questions, which validates the model dynamically. You need to initiate rework if you identify invalid elements, and you need to continue to do so until that part of the model is ready to be delivered and satisfies expectations.

Simulations provide a good way to validate the function of a model based on executing functions and verifying the result against the expected result. A match indicates that the model is working as expected, and a variation indicates issues.

Consider the following aspects of simulation and verification:

- You can perform the simulation in multiple ways. The simulation can be a true and thorough execution of system elements. However, this method is not an option if you want to verify the model prior to anything functional being built. Instead, the simulation can act as kind of a *pencil test* that proves expected functionality based on the design before it has been implemented.
- Consider the input that you are going to provide, and manually process or step through all the activities that you have designed that manipulate the data, and then write down the result. An expert can confirm the result or reject it as incorrect, preferably presenting you with the correct result. Based on that result, you can rework the data manipulation process accordingly.

You need to know the outcome of the verification test in advance. This process is no different than other test techniques, but often the test team will not be in the position to provide you with the desired test cases yet. In this case, you will have to come up with the test cases yourself. Probably, you will want to use a domain expert to help you with this task. Needless to say, this is a collaborative task.

These validation options deal with the dynamic behavioral aspects of the model. Consider also the static aspects of the model. A reasonable method to validate the model is to ensure that the model corresponds to the metamodels. The metamodels provide general guideline regarding what is possible and what is not possible. Ideally, you create these metamodels prior to creating the models. If you do so, you can validate through verification that all modeled elements adhere to the description of the metamodel. If such a metamodel does not exist, you can create one now for the same reason. When you can describe the model itself by means of a metamodel, then you can understand the model itself. Sharing the metamodel with other team members also puts them in a position to verify the model that you created, with the metamodel functioning as the guide.

9.2.1 The role of the metamodel

This section discusses the role of metamodels when validating the static aspects of a model.

Data models

Although you can check data models using many different methods, we cover only a few here.

The first aspect is related to the verification of the design. You can validate the level of normalization of data models and check the referential integrity of the physical data model. Other aspects include verification against naming standards, for example, against a glossary. Additional checks are syntax-related and include, for example, verifying that names of objects and attributes adhere to the expected standards and that data types are properly declared. You can also verify the proper description of constraints against the system and check for completeness, provided that you have a measurement available that defines what complete really means, such as a requirements model that you can trace back and demonstrate completeness.

UML models

UML models are of a slightly different nature and, therefore, allow for additional checks. You can and should run a metamodel check. The metamodel in question here is the UML metamodel itself. When your design adheres to the predefined UML standards, including any extensions, you can proceed, keeping in mind the code generation that you want to support later.

Next, ensure that you are working towards the required transformations between the various models. The models need to be compliant with the input structure that is expected in the defined transformation rules. Similar possibilities apply when you have created your own UML profiles and defined additional metamodels.

9.2.2 Considerations

Before diving into an example, consider the following practical questions:

- When should you execute the model validation?

  A good habit is to validate your created models prior to passing them to another team, by including model validation in the pre-delivery process. As the team receiving a model, consider validating the deliverable when receiving models as part of the delivery, thus embedding the validation into the delivery acceptance process. Both then function as quality gates between adjourning teams.

- How do you validate a model?

  First, be pro-active. Consider using tools that are readily available when performing validations. Run the built-in tools in your modeling environment. If the built-in tools do not completely fulfill the required validations, consider creating a custom validation tool that can work on many different levels, starting from check points on a separate spreadsheet and moving to enhancements to the modeling tools that are available. An integrated approach is best, but it requires more time.

9.2.3 Performing the analysis

This section describes a validation example of a data model against a glossary. The focus of this discussion is how to create a complex, yet consistent, set of models. Thus, it does not provide step-by-step instructions about how to carry out model validation. Instead, it can help you to understand the importance of the modeling steps outlined thus far. These steps are the base for validation of the models that you have created.

Remember also that the capabilities that are provided depend on the validation tool that you select and even the versions of the tool therein. In addition, at any time you can introduce your own validation tools to verify what is important to you. In fact, validation requirements can vary between individual teams. The following information provides only a rough outline of the model validation capabilities.

Using the built-in capabilities of IBM Rational Software Architect and including IBM InfoSphere Data Architect, which is built on the same platform, model analysis requires a two-step process:

1. Specify the preferences for the analysis. This step is typically performed only once, unless the requirements for the analysis change or another glossary is added. Figure 9-7 on page 473 shows the naming standard preferences that use a simple configuration section.

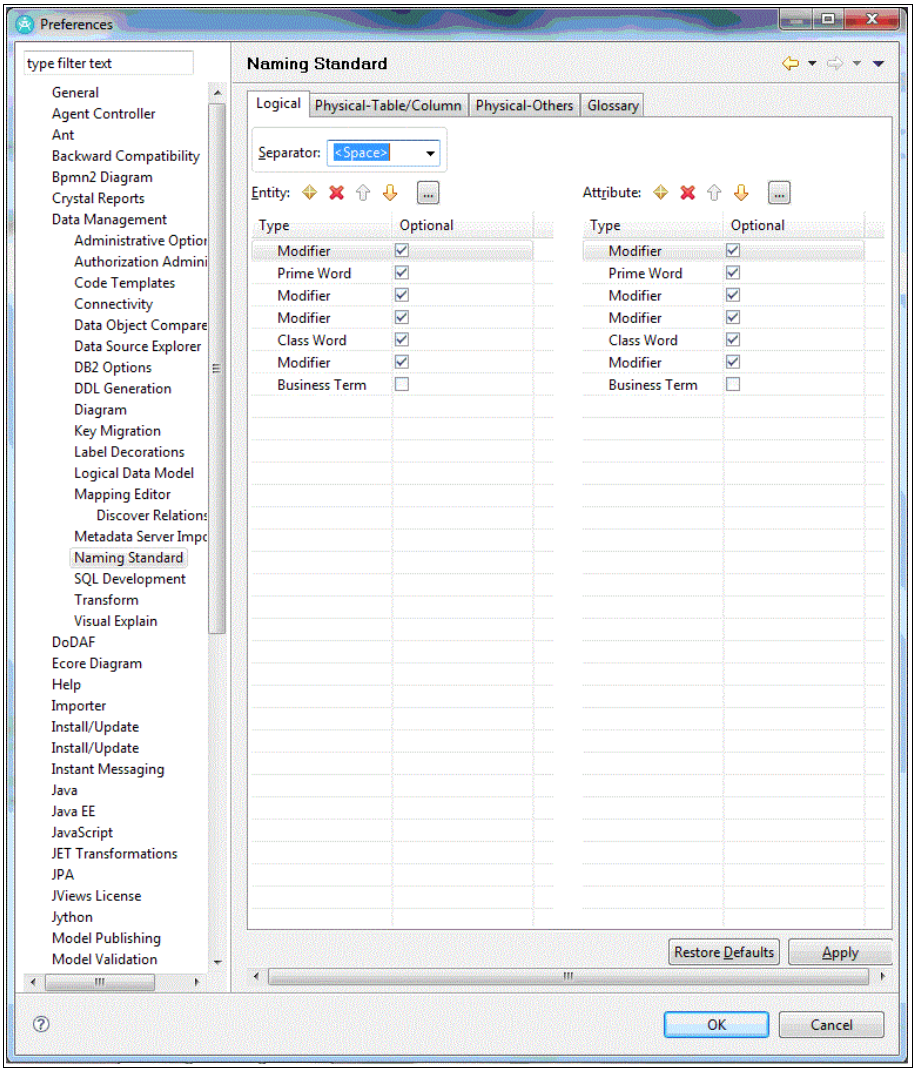

Figure 9-9 Logical naming standard preferences in IBM Rational Software Architect

You can configure the naming standard checking on the four tabs included in this dialog box. The first three tabs are related to the logical and physical models and to the glossary terms used to verify these models. This example uses only the business terms as mandatory values. On the fourth tab, you can specify the glossaries against which you want to verify the model. This setting applies globally and is always used when performing a model analysis. After you have added glossaries, you can still override or add other glossary terms in the model validation dialog box.

2. Specify the model validation preferences.

   These preference settings allow you to select the individual models and elements that you want to validate when performing the model analysis. Select and clear the items to meet your requirements. When running the model analysis later, you can change the settings that you have specified here and, thus, adapt them to specific analysis requirements.

To verify that all the entity names adhere to the terms specified in the glossary, select the **Object names** option on the Syntax check item of the Logical Data Model branch, as indicated in Figure 9-10.

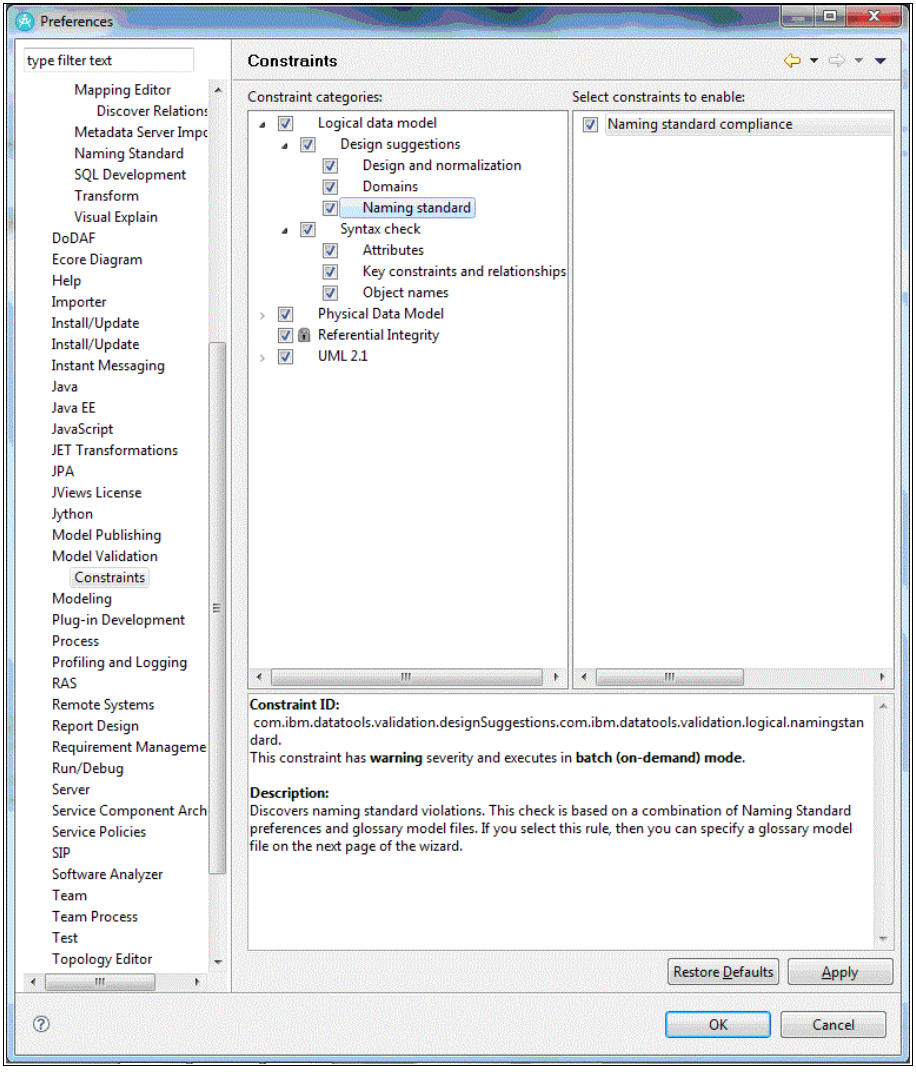

Figure 9-10 Model validation constraints in IBM Rational Software Architect

9.2.4 Validation

Now that you have created the preferences for analysis, you can validate the model:

1. Open the project explorer and select the package that contains the logical model that you want to analyze.

   Note that the Analyze Model menu is available only on packages. If it is not visible, perhaps you have not selected a package but have selected the model or another other item instead.

   Figure 9-11 shows the context menu that contains the Analyze Model item on the Package menu of a conceptual enterprise information model, a logical model in terms of IBM InfoSphere Data Architect.

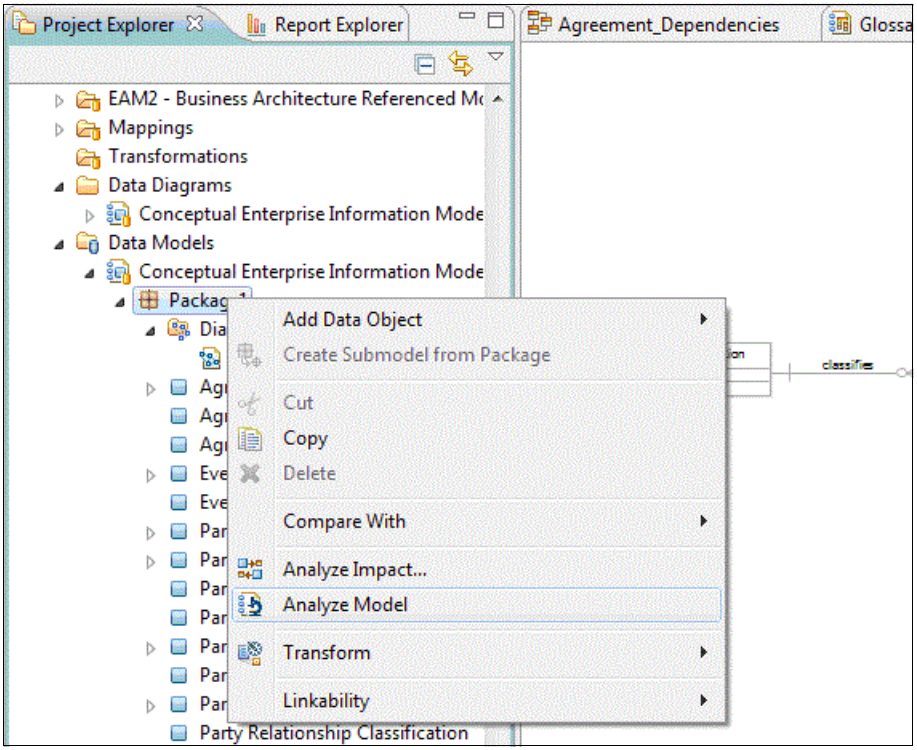

Figure 9-11 Model validation in IBM InfoSphere Data Architect

2. Specify options for the model analysis rules. The results of the validation are displayed in the console.

3. Based on these results, correct unexpected behavior or violations against the specified preferences accordingly before you pass the deliverable to the next team.

This example provides only an overview of the model validation capabilities that are provided by both IBM Rational Software Architect and IBM InfoSphere Data Architect. Refer to the topics in , “Related publications” on page 559 for more information about this feature.

9.3 Impact analysis

Impact analysis represents another way of ensuring consistency across models. An impact analysis lets you determine whether you can delete a design element without corrupting the entire model, in cases where the element is not being used. It can also provide an indication to refrain from deleting an element in cases where the element is still in use within the model.

Although impact analysis can increase the consistency of the model by preventing corruption, it can also help you address challenges related to activities that surround a project. In this context, impact analysis includes the following processes:

- Evaluate the feasibility of making the change.

  When you encounter change of any kind, first *evaluate the feasibility* of carrying out that change. Be aware of all areas that this particular change touches, and remember that the complexity and size of the models can affect the likelihood of whether you make the correct determination.

  When implementing a master data management solution, you will most likely use a large model. Thus, adding traceability information to models is critical. You can use the automated analysis of these pieces of traceability information for impact analysis. The result can provide information about any element that is remotely affected by a possible change, starting with changes in requirements. After you have a more complete picture, you can provide the result and evaluate whether it is feasible to carry out that change.

- Judge the impact of the change.

  After you determine the feasibility and all the affected model elements, you can more easily *judge the impact* of a possible change and the elements that are impacted. The issue is figuring out the impact that a change might have when built-in parts of the model are affected by the change. Can you make a change without breaking the consistent model? What is affected if you change only some parts?

- Identify risks that are associated with the change.

  After you have identified the elements and determined the impact of a change, you can then *identify the risk*s that are associated with that change. In most cases, the risk depends on the number of elements that are dependent on the change, the complexity of the area that is affected by the change, and the reuse factor on that element.

- Estimate the effort involved in making the change.

  Based on the impact analysis and knowledge about all the elements that are affected, you can determine the effort that is involved in applying that change. You can use the determined elements as rationales into estimations and, thereby, can document the estimation and underpin it with facts that are derived directly from the model.

- Guide the changes.

  After you have run the impact analysis and have obtained a thorough understanding of the aspects in the model that will have to undergo change, use the impact analysis as a guide. Describe for your designers the areas that are addressed by the requested change. This process can help to ensure the consistency of the models.

9.3.1 Performing the analysis

Your impact analysis will be carried out at a different time than your model validations. A logical point of evaluation is when enhancements or change requests are processed. You can also apply this method of analysis to the start of the next phase of a project in case new features are added that are based on already existing artifacts. Each designer should frequently assess the situation around the artifacts that are being modeled. Typically, you would analyze an area of a model when that area touches upon parts that have not been worked upon for a while. In this case, you need to understand the elements in that part of the model before adding new design elements or changing existing elements.

9.3.2 Preparation

Successful impact analysis lies in the traceability information embedded in the model during the design phase. Although adding traceability might appear to be an unnecessary overhead at the design phase, impact analysis is an important related modeling aspect. It is during impact analysis that traceability enables the discovery of important dependencies.

9.3.3 Analysis

The example described in this section is based on built-in impact analysis capabilities with IBM InfoSphere Data Architect. Similar functionality is also available within IBM Rational Software Architect. The following example illustrates which other artifacts are impacted if you change an entity within your conceptual enterprise information model.

From the project explorer, select any item of the conceptual enterprise information model. This example has only entities. Right-click to open the context menu, and then select **Analyze Impact**, as shown in Figure 9-12. Specify the impact analysis options, namely whether you are interested only in dependent objects, impacted objects, or both. Additionally, you can select to process further contained objects and everything else recursively. This option is typically the option to choose when judging impact throughout all levels of the model, starting with requirements and ending in source code or even test cases, if added to the model as well.

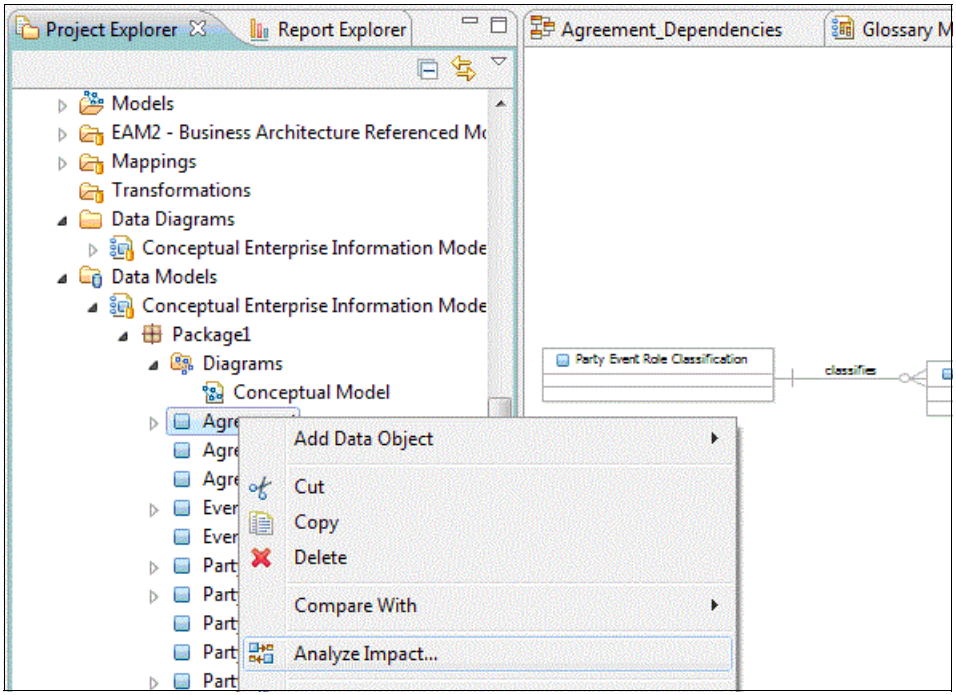

Figure 9-12 Impact analysis in IBM InfoSphere Data Architect

The resulting diagram, shown in Figure 9-13, shows only the relationships that are contained within the logical model itself and the traceability dependency that is added separately.

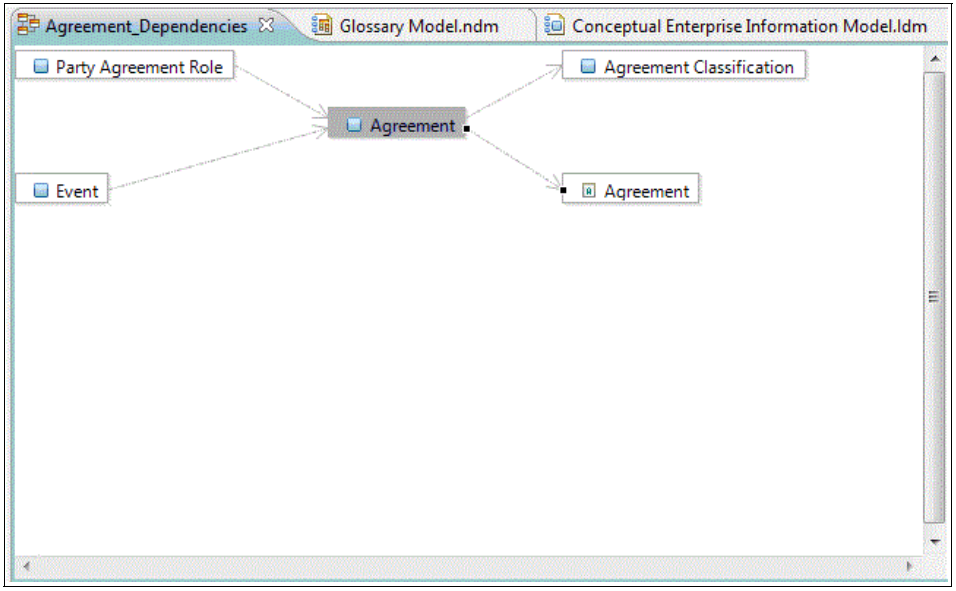

Figure 9-13 Object dependency diagram after impact analysis in IBM InfoSphere Data Architect

Based on this information, you can judge any potential impact that changing this entity might have on other parts of the model. When including other contained objects and allowing for a recursive analysis, you can traverse through large parts of the model and, thus, gain a more complete picture. However, such diagrams are typically not the best input to a project manager.

Instead, apply separate reports that can provide the results as listed in Figure 9-14. Alternatively, use the Model Report view, which also provides an exportable list. Note, however, that the list is incomplete because it does not show the dependency type or any additional information that you might have provided on the dependency type.

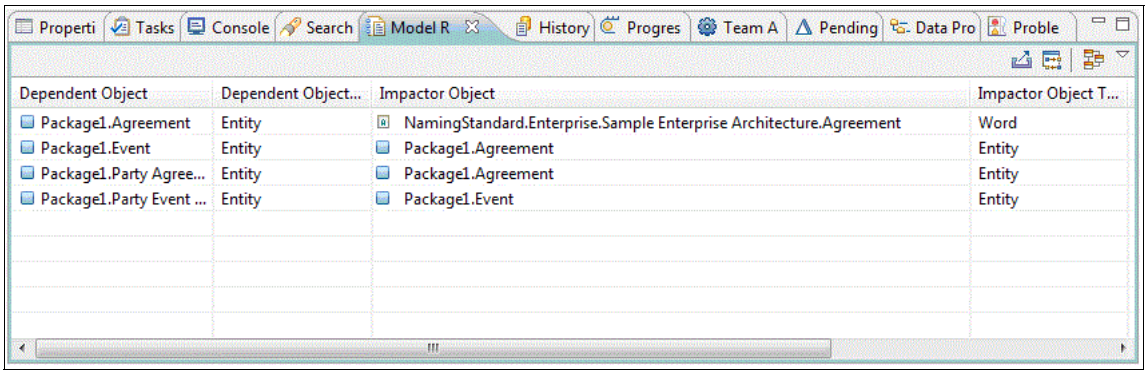

Figure 9-14 Model report view after impact analysis in IBM InfoSphere Data Architect

You can also obtain reports that you require in the format that you need using BIRT reporting capabilities.

This section provided only an overview of the impact analysis capabilities provided by both IBM Rational Software Architect and IBM InfoSphere Data Architect. For more information, see “Related publications” on page 559.

Part 3

Planning considerations

This part of the book describes implementation considerations. The previous chapters included information about modeling and the techniques that are used in modeling. You also learned about the various types of models that you can use in an implementation. The previous chapters in this book also described aspects directly related to modeling, such as outlining why a model is needed, what model is needed, and how to create a model.

This part includes Chapter 10, “Planning for your master data management project” on page 491. This chapter provides additional information that surrounds modeling aspects. Although this chapter cannot address every challenge related to modeling, it does describe, in general, the challenges that are associated with master data management. It also presents an implementation strategy and provides information about how to set up and execute a project and how to choose the appropriate tools.

10

Planning for your master data management project

Previous chapters in this book described aspects directly related to modeling, such as outlining why a model is needed, what model is needed, and how to create a model. This chapter provides additional information that surrounds these modeling aspects. Although this chapter cannot address every challenge related to modeling, it does describe in general the challenges associated with master data management. It also presents an implementation strategy and provides information about how to set up and execute a project and how to choose the appropriate tools.

10.1 Challenges

Introducing a master data management system presents several challenges. Not all challenges are related to the systems that deal with the data. This section describes the impact of the following types of challenges on a master data management project:

- General challenges
- Master data management challenges
- Design and implementation challenges

10.1.1 General challenges

The general challenges described in this section can affect any enterprise and master data management solution implementation.

Tight, fixed deadlines

Tight, fixed deadlines can pose a major challenge to master data management implementation. Master data management solutions can be complex in nature because of their positioning at the heart of an enterprise. This complexity often results in unexpected side effects on the other systems that depend on the master data and that attach to this single system. Usually, there is not enough time to resolve all of the side effects. In addition, some deadlines might be imposed by regulatory requirements, and other deadlines are self-imposed, for example, by the business strategy. Tax-related requirements are mostly introduced at the beginning of every tax year and require tax-oriented solutions to go live before that date. It is important to identify such time constraints and embed them properly into the master data management roadmap. Then you can set up the project plan.

Requirement for ad hoc integration of acquisitions

The trend to acquire other businesses is a growth option for many companies, especially in sectors where organic growth is not possible, or is possible only at a slow rate. From the project team point of view, these types of acquisitions require integration of the newly acquired systems or migration of the systems' data to discontinue those systems in an attempt to cut costs subsequent to merging the systems. Because such acquisitions come suddenly, they have the potential to interrupt a master data management project that is already in progress. These challenges are why acquisitions are, more often than not, embedded in the master data management roadmap. If your strategy includes growth through acquisition, you need to incorporate this challenge into your master data management strategy as well. In fact, ad hoc integration of acquisitions is a good reason to initiate a master data management strategy.

System integration

Most master data management projects require system integration of some kind. Replacement of some of the existing systems is desirable for many master data management strategies. However, many times this replacement cannot be achieved, perhaps because of many secondary systems that depend on the system that you want to retire. Retiring systems requires that you move interfaces from the existing system to the new master data management system.

Often, systems replacement is too large an undertaking to be manageable. In this case, the only way to achieve the goal of system integration is to follow a step-by-step approach. After the initial steps, provide the dependent systems with a continuous synchronization stream to keep them updated. The cost of such an approach is a significant increase in complexity, depending on the number of systems to support, the complexity of the interfaces, and the business rules to which they must adhere. Do not underestimate such an effort, because it can magnify any reasonably simple master data management integration project out of proportion.

System replacement

Aiming for a complete system replacement or retirement requires system integration, including all of its business processes. More often than not, you cannot introduce a master data management system successfully while simultaneously retiring or replacing an existing system. This complexity with replacing systems is due mostly to existing systems that have grown dynamically over time and, therefore, are rather difficult to replaced. The dependencies for these systems are numerous, and the amount of hidden logic to be identified and rebuilt is often unknown. In addition, there often are many business rules buried so deep within the system that they are often unveiled only when attempting to load data from the existing system into the new master data management system. Data inconsistencies and failures because of unexpected data constellations are the result.

To prevent unexpected results, you can dedicate a thorough and prolonged phase to data analysis through profiling. Base your rationale predominantly on the number of systems that are dependent on the system to be retired and that must switch to a new system. Also, the number of business processes that such a system provides is a good indicator of whether you can replace the system. The more of these processes that must be switched, the more complicated a retirement becomes while introducing the new system at the same time.

10.1.2 Master data management challenges

In addition to the general challenges mentioned in 10.1.1, “General challenges” on page 492, there are a number of master data management-specific challenges that revolve largely around the following factors:

- Integration of multiple products and systems
- Each system has its own distinct data model and interfaces
- A master data management that provides a solution for all systems
- Master data management that provides for integration with existing and new components

Regardless of whether you aim for system retirement or want to continue synchronization, you must integrate multiple systems with a master data management solution. Using integration provides enterprise common data to all systems that are dependent on it. Even when you reduce the scope of your analysis and focus only on the master data management system itself, you are confronted with products for metadata management, products to ensure data quality, and much more. (You can find more information about the context of master data management components in 2.4, “The context of master data management components” on page 65). It is important to understand the complexity involved in a master data management system, because there are many external systems to integrate and various supporting components that complement the master data management data storage and service exposure.

The side effect of having to integrate many disparate systems is that there are many, often incompatible, data models and interfaces. At the heart of the problem are not the services, but the data models that are behind the services and the objects that these data models expose. It is highly likely that the data model varies between each system, as shown in Figure 10-1.

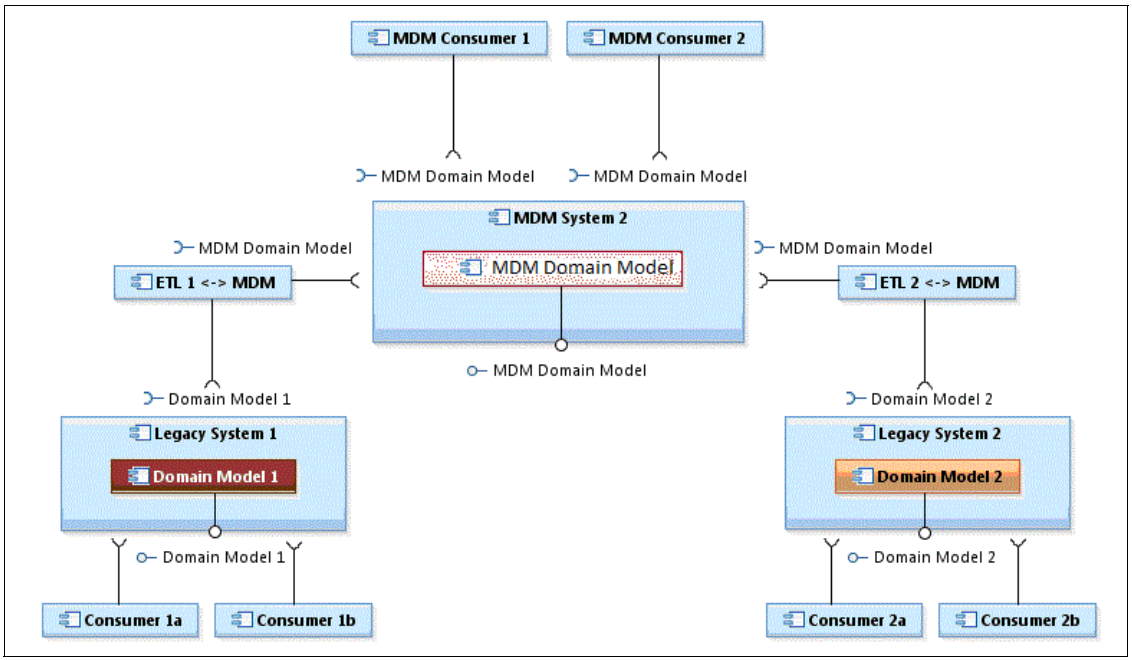

Figure 10-1 The different data models in a system context

The variations are due to the nature of the systems. An Information Management System (IBM IMS™) that does not include a business abstraction layer exposes segments that are specific to the Information Management System, while a DB2-based system exposes its relational tables. Because some systems do not have such an intermediate, application-specific logical data model defined, but rather have grown around their physical data model, these models can vary. Because the master data management system is supposed to become the centerpiece for the new enterprise, you have the following options:

- Establish a new data model.
- Use an existing data model.

The rationale for choosing one method over the other is to look at all systems that are related to the master data and establish whether there is a common data model already in place. If it is, it makes sense to use this common data model as the data model to be adopted by the master data management system. If not, you can also consider establishing an entirely new model that follows a different methodology.

IBM proposes a business-driven approach for complex implementation style scenarios. Such an approach focuses on the business view of the system and is based on the domain model. This *domain model* can serve the needs of all systems that are related to the master data management system. The domain model establishes one model that is unique to the business and that is in a ubiquitous language for everyone who is participating in the project. So, the integration is based on a common understanding. Also, communication across system boundaries is based on a single communication protocol structure. With the domain main, you must consider transformations between the various data models. However, you have the same consideration when you introduce an existing data model, because there most likely are systems that operate on a different data model.

Most systems that participate in a master data management solution provide their own interfaces and communication protocol structures. Transforming these protocols from one structure to another is essential, whether you introduce a business domain model. The same is also true if you purchase a master data management product, such as IBM InfoSphere Master Data Management Server (referred to as *InfoSphere MDM Server*), and introduce it into the system landscape. InfoSphere MDM Server provides its own data model, and you can adopt it as the central data model, based on which attached systems integrate.

10.1.3 Design and implementation challenges

After you start a master data management initiative, you can face a number of implementation-related challenges:

- Master data management projects are, generally, large.
- Many different system aspects must be considered.
- Many architect and design teams need to be acknowledged.
- A consistent approach and methodology must be ensured among all teams.

Regarding the size of master data management projects, those projects that follow a complex implementation style are typically large. The more systems there are to integrate with the master data management system, the larger the number of attributes and components that are involved. It becomes difficult to remain current with the activities going on within the entire system. In fact, the project can grow so large and quickly that no single individual can keep abreast of all of these activities. As a result, you need to choose a design approach that accommodates these difficulties.

Regarding the many system aspects, consider a number of system contexts. The more external systems that you must integrate, the more data models you must observe. Each one has its own rules and dependencies, and you must consider them all for a successful design. In organizational terms, you need experts from both the existing system and the master data management system to be part of the project team.

The size of the project requires a thorough structuring and establishment of teams that deal with each aspect of the system. It is common for a team of architects to work on different levels. Some architects look at the entire system form end to end and ensure the successful integration with enterprise architecture defined standards. Other architects define models for the numerous components that are involved in the overall architecture.

In addition to the functional view of the system, there are also logical aspects to be considered. Team structures are necessary because you must keep the size of the teams to an acceptable number of people. So, while one team works with the new data store that the master data management system provides and works with exposing the core data integration logic, another team deals with everything related to use case implementation. Both teams deal with a specific aspect of the system. More problematic is where you must bridge the gap of translating from a business model to a system-specific data model. This team requires special skills and the models must interconnect seamlessly.

Regardless of how the project team is set up, the challenge is in identifying the possible sequences in which the teams can work and the dependencies between them. It is vital to identify the tasks that can be safely carried out in parallel and separating these tasks from strictly sequential activities. Ensure consistency across all design artifacts that are created with the teams involved. The success in doing so depends largely on the tools chosen and whether a model-based design is chosen. A model-based approach using a single tool is likely to be successful in a large-scale project environment.

The number of individuals that work on the project and the broad skills that are required mean that team members can have only a rough understanding of their own contribution to the big picture. Consequences of their actions are not immediately clear, just as the dependencies to others' actions might not be visible. Planning for the correct amount of communication effort can address this issue. Other issues, such as misinterpretation of interfaces and the necessity to provide realistic test data for the respective software modules, must be taken in to consideration.

10.2 Implementation strategy

10.1, "Challenges" on page 492 described how a master data management solution includes a master data management product and related technology, organizational aspects, and, most importantly, the strategy that initiates, supports, and drives it. Putting all of these components together and asking how you are going to achieve this solution leads you to defining the roadmap and related project plans that guide the solution from start to finish.

This section clarifies the question of how to get to the roadmap and prepare the framework for the project plans that enable project managers to set up a successful master data management implementation project. IBM Information Agenda® can provide guidance.

IBM Information Agenda acknowledges the fact that strategy is business-oriented, because the main purpose behind any organization is its business and the goals it is going to achieve. Carrying out the business is supported by the business processes that help perform all tasks in a predictable, repeatable, and consistent way. New strategies often require new business processes, or at least changing some of the old and established ones. This situation is supported by an appropriate technological infrastructure using hardware, software, and services. Many IT departments like to see it the other way round. However, the IBM view is that IT is merely the enabler for the most effective and efficient execution of business-oriented processes and tasks that, in themselves, are executed only to achieve the business goals set and defined through the strategy.

Ultimately, the goal is to establish a roadmap that likely leads to success. As soon as the roadmap is defined, you have the cornerstones for your individual projects, where each phase of the roadmap manifests itself in a solution implementation project. Later, when all projects conclude successfully, you achieve the defined goal and provided a technical solution that supports the business, reaching the goals defined through the business strategy. Figure 10-2 depicts this principle.

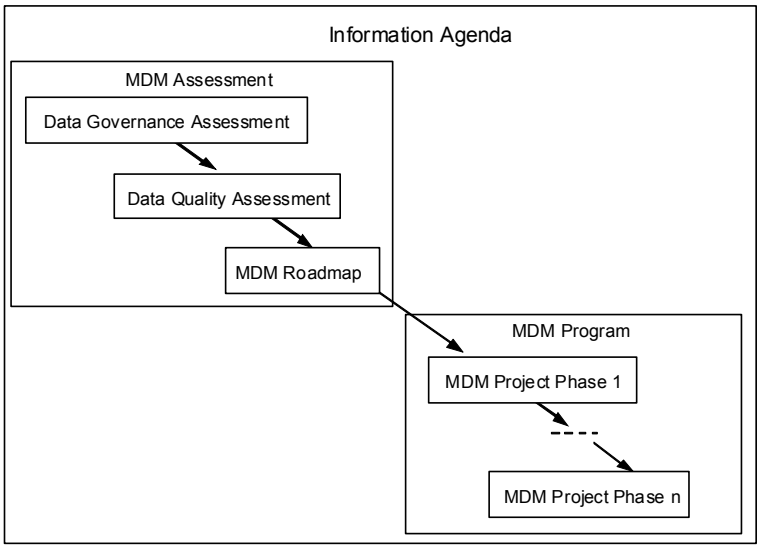

Figure 10-2 Implementation strategy for a master data management solution

To get to that goal, look at the strategy. Every organization is different, which is good because it adds market value. For this reason, consider the business goals defined in that strategy. In addition, analyze the organizational structures, and define a way to use them to the greatest effect possible. The same principle applies to the technical information infrastructure. You might need to adopt both the organizational structure and the technical information infrastructure to support the new strategy, but you can identify that necessity only by the initial analysis.

Figure 10-3 illustrates how your enterprise can become an information-based enterprise.

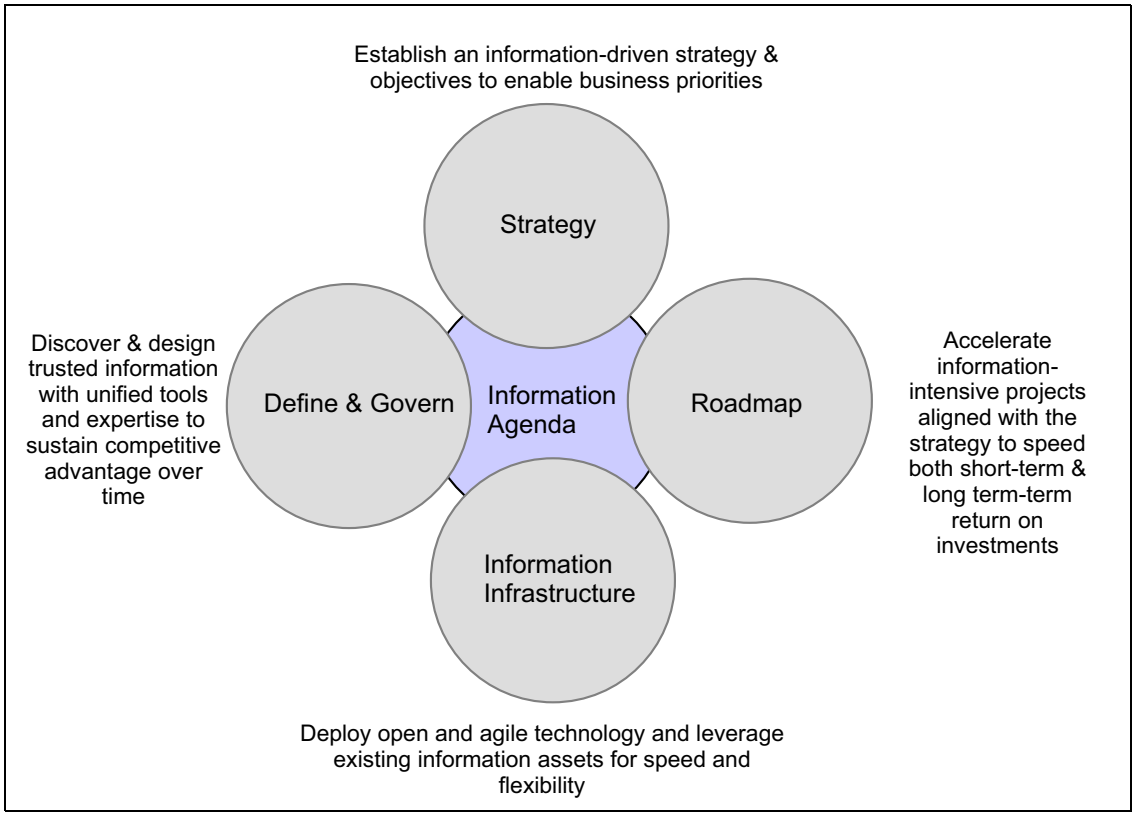

Figure 10-3 Becoming an information-based enterprise

IBM Information Agenda defines the following pillars for any information-centric strategy and, for that reason, is considered the foundation for a master data management solution:

- Strategy
- Governance
- Information infrastructure
- Roadmap

10.2.1 Strategy

A clear strategy outlines a business-oriented vision of where a company wants to go and how it wants to get there. For example, strategies that include master data management can include statements, such as building better customer relationships by offering them self-service terminals, or adhering to regulatory compliancies that require a holistic view of a person. Your strategy builds a backbone for activities carried out within the organization and also ensures the top-level support that master data management activities require. The challenge with master data management initiatives is that they are often rather long term. Thus, these types of activities need to be durable, reliable, and consistent, even when other business factors are changing over time. Before starting your journey to a successful master data management solution implementation, you must ensure that you have support from the highest possible level in your organization.

There are a number of important aspects to consider when defining or evaluating a strategy. The list that follows describes some commonalities between all of the strategies, but also includes points primarily dedicated to master data management strategies. This list provides guidance in reviewing any master data management strategy so that you can determine a strategy and ensure that it contains all of the relevant aspects of your goal for a successful master data management implementation.

- Industry specific

  Each industry has unique challenges. Do not try to adopt a banking-specific solution for the telecommunications sector, unless you can identify overlaps in their needs. For example, a bank requires a holistic view of customers for regulatory compliance, while a retail business might want to improve customer, supplier, and manufacturer relationships. Your strategy needs to reflect this type of focus.

- Proven globally

  In today's interconnected world, information spreads quickly. The best strategy is often one that has proven to be successful globally. However, global success is not the only criteria, because each business needs to ensure its own value in the market. A local- and business-specific variation of a globally proven strategy can be a good solution. Ensuring that variations are clearly formulated draws attention to them. Often, small business-specific adoptions distinguish an enterprise from others in the market.

- Focus on information management

  An overall strategy does not focus on information management alone; however, the IBM Information Agenda does focus on information management. With better information management, business decisions can be based on accurate and timely data, and better business results come from better information with higher quality. Master data management represents a solution that is focused on providing the best possible data that can establish a data-centric strategy, as outlined by IBM Information Agenda.

- Identification of key business goals

  Clear and concise business goals are the conclusion of a business strategy and make a business unique and successful. Both internal and external factors can influence the decision to reach a specific goal. For example, an internal influence might be the desire to expand on the competitive advantage. An external driver might be the need to adhere to regulatory compliance requirements that are enforced by national or international legislation.

  Consider the following examples of internal drivers:

  - Reduction of errors in processes to increase efficiency
  - Enhanced customer service levels
  - Greater customer insight
  - Increased marketing effectiveness
  - Identification of new revenue opportunities

  Consider the following examples of external drivers:

  - Regulatory compliance
  - Fraud detection
  - Risk management (operational, reputational, and financial)

- Identification of key IT goals

  IT drivers can also have an impact on your strategy. IT needs to support the defined strategy and must provide efficient management of the business processes that are required to carry out the strategy most efficiently. Sometimes, IT systems cannot adopt new solutions easily, because the existing system has been in place for many years or because the existing system was intended originally to address other needs and must be realigned after a change in strategy. Sometimes the reason for change is also as simple as reducing costs to run the business cheaper and to grow profits in turn.

  Consider the following examples for IT drivers:

  - System retirement
  - System optimization
  - Cost reduction

- Identification of key organizational goals

  The most successful master data management initiatives include the definition of new organizational goals with a shift to a more information-centric way of thinking and operating the business. Your employees might create new copies of data or re-create data in the context of their own silos, but with a master data management solution, it is crucial that they assume central ownership of the data. This situation, in turn, requires the introduction and establishment of new organizational functions, such as data stewards.

  Consider the following examples of organizational drivers:

  - Information governance
  - Enterprise architecture

After you identify the strategy items and the goals defined in the strategy, you must prioritize these goals. For this prioritization, consider all the following types of goals:

- Business
- IT
- Organizational

Your strategies and goals affect your prioritization. If your company is severely hit by fraud, countermeasures to this attack rate highly in your prioritization. If company revenue is on the decline, aims for increased marketing effectiveness rate higher.

Initially, keep the list of priorities for each of the types of goals separated. After you determine the roadmap for your solution, you can then combine the priorities within the phases of that roadmap. From the roadmap, you can then determine the final priorities or on special themes that define the scope for the project phases. Often there are too many aspects to be considered when trying to prioritize everything at once. Aiming for a prioritization against defined criteria from each department is a feasible alternative. The business units that represent the stakeholders can prioritize the business goals, while the IT department prioritizes its own goals. This way, you are presented with a directive that can most likely lead to a successful prioritization.

10.2.2 Governance

With the master data management activity of combining company silos into one central data hub, there is the possibility that your company can undergo organizational changes. Where previously there were many people responsible for their copy of the same data, now one person or a group of people are the data stewards. In such cases, many companies shift their focus from a mere IT or business goal-driven strategy to a strategy that is highly influenced by information. They aim for better support for their business decisions through higher quality data.

However, such a shift does not occur easily. In fact, this shift can be a bigger challenge to establish than creating an IT system in support of an information management initiative. When you evaluate the current organization, you often are required to restructure it. A new, revised organizational structure might introduce a new data steward as the sole person for aspects related to master data management. This person works closely with other people in the organization, as Figure 10-4 illustrates.

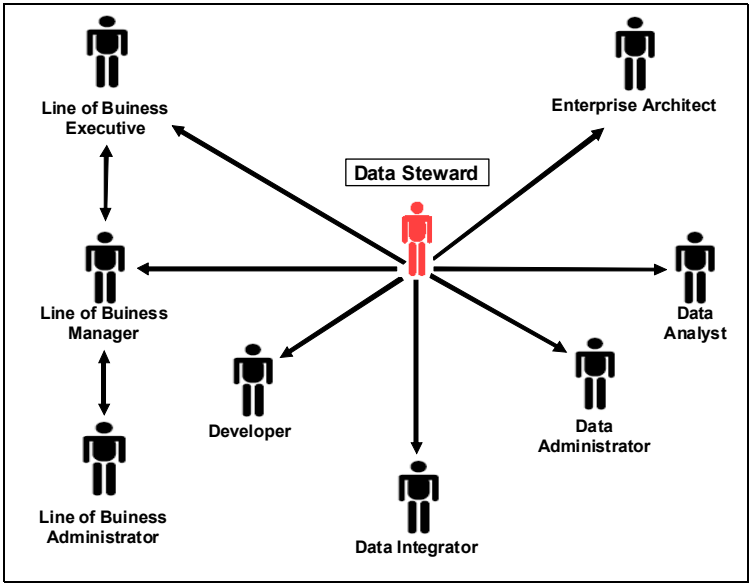

Figure 10-4 The actors of data governance

Consider the following key aspects when setting up an organizational structure and information governance:

- Employees and roles

  Some employees in the business units might be used to the idea of being the owner of their data, and they might be reluctant to give up ownership. However, master data management is not about a single employee or a single system that works with the data that the master data management system provides. Thus, you must educate employees about this new mindset.

  Often, owners of the existing silos can become converts to the new solution and can assume the role of a data steward. It can be wise to use this strategy, because the owners of existing silos have experience in dealing with this data. They certainly have the insights that are required about the data and its usage within the organization to make the best possible decisions.

- Processes

  In addition to the change in responsibilities, your employees also must adapt to changed business processes. Without master data management, there are many processes that manipulate the same data in many different ways, driving data inconsistencies. Your key to success is to ensure that there are no such contradicting processes through the introduction of master data management.

- Communication

  Changed business processes represent only one aspect of taking on master data management. Another aspect relates to the way team members communicate and work together within the new information governance framework. When changing from many silos to a central master data store, often lines of businesses cannot agree on using each other's attributes and at times abuse the semantic. It is the data steward's responsibility to coordinate such efforts and to involve enterprise architects when it comes to introducing new attributes or changing semantics.

- Information

  Information is the new center piece of your organization and needs to be treated carefully. Information governance ultimately leads to increased data quality and reduced data-related risks.

10.2.3 Information infrastructure

The foundation for any master data management solution is based on a solid technological infrastructure, which can be achieved by a central hub that distributes the master data to the other systems that depend on it. Regardless of the implementation style, you must integrate a number of systems. Consider a service-oriented architecture (SOA) approach where systems are loosely coupled and are more dynamically adaptable to changing requirements.

Be aware of the following key elements:

- Extensibility

  Extensibility represents an important factor for master data management systems. The roadmap defines a phased approach to the implementation, and you do not know what requirement is imposed next. New legislative regulatory compliance acts are defined frequently that require adoption in the master data management system. Other aspects include new business requirements, which you must adopt quickly.

  Master data management provides the needed extensibility. Carefully determine an information infrastructure that can grow while also changing as needed along the way. Also, consider whether to provision for the possibility of changing business rules and the next revised strategy, although that revision is not expected to happen often.

- Flexibility

  Flexibility in a master data management system is equally important, mainly for the same reasons as outlined for the extensibility requirements. However, flexibility also relates to the ability to implement a certain requirement one way or another, depending on the vision of the project team. A master data management solution, such as InfoSphere MDM Server, uses its strengths to provide increased flexibility, when compared with a packaged solution that is typically more static and requires that your system adapt to its capabilities.

- Maintainability

  Maintainability plays an important role because of the long duration any such central system can have within an enterprise. Over time, you are faced with the need to maintain the system that is in place, without disrupting business and without requiring specialists on site at all times.

10.2.4 Roadmap

The roadmap represents the ultimate goal of IBM Information Agenda. It is the bridge between your strategy and concrete projects that can be initiated. Because many master data management strategies are large and cannot be accomplished within a short period, a roadmap defines a step-by-step approach toward realizing the vision as outlined in the strategy. It leads to many projects that you work on in succession.

It is important that the individual steps of your roadmap are clearly defined, measurable, and achievable within the strategy. While you are building your roadmap, group certain aspects together. Identifying and building themes helps clarify and set the focus on the scope of the project. Include only a single, large topic in terms of time and priority in each project phase so that you do not overload the project duration or overwhelm the people on the project with juggling too many high priority issues.

For example, a company has a strategy that includes replacing the existing system, achieving regulatory compliance, and moving data from a newly acquired company into the company's own systems. System replacement and migration are both large topics that require a dedicated focus and project phase. Only rarely, and with small systems, can both goals be achieved in a single project. Try not to mix regulatory requirements into the migration. Instead, either include them in the system replacement project, if only minor changes are required, or dedicate a separate project phase to them, if it can be combined with similar goals. A theme can be based on business or IT goals or a specific subdomain of the business goals. You also must incorporate deadlines. These deadlines can be dates to which you need to adhere, for example, dates for required regulatory compliance legislation.

Defining the correct roadmap depends largely on the goals that are defined in the strategy. Pay attention to the following predominant aspects in defining a roadmap:

- Realistic maximum time allowed for each project in the roadmap
- List of external drivers and their deadlines
- Prioritized list of business goals
- Prioritized list of organizational goals
- Prioritized list of technological goals

After you know what needs to be done and when it must be accomplished, you can look at what is already in place to support the journey described in the roadmap and determine a gap analysis. Include the following gap analysis when defining a roadmap:

- Gap analysis of current and new strategies
- Gap analysis of business goals
- Gap analysis of the organizational governance topics
- Gap analysis of the information infrastructure

The next major block is related to data quality assessments. You can either carry them out in advance, which usually makes sense when the organization already has an information governance in place. Alternatively, you can move this task toward the beginning of each project. The advantage of including this task in each project is that the scope is more clearly defined and not all data quality tasks must be assessed. Only the data that plays a role in the scope of the project must be assessed.

10.2.5 Projects

After you establish a roadmap, you have the cornerstone definitions for your projects. This information does not provide a work breakdown structure (WBS) that is ready for use, but it can provide insights into establishing your own WBS. In addition, you might want to adopt a common methodology that moves your projects forward. The advantage is a structured approach throughout all project phases, which, in turn, can lead to predictable results.

Master data management projects typically apply a methodology for this reason. Many project methodologies can be used to drive master data management projects. The most common methodology is derived from the IBM Rational Unified Process (RUP) methodology. (You can find details about project management-related activities in 10.3, “Setting up and executing a project” on page 511.)

Consider the following logical phases with the suggested modeling methodology:

- Discovery

  In the *discovery phase*, you identify the scope of the project. The rough scope is identified when you define the roadmap. In the discovery phase, you look at the scope in more detail.

  This phase includes typical project management-related activities. You must evaluate requirements to which the master data management system must adhere and determine the appropriate scope exclusions and assumptions where there are still questions. You also need to include possible team structures in this phase to be responsible for project management.

In addition, you must allocate sufficient time to define core concepts in relation to the strategy. Enterprise Architects and the Business should work together to determine clear definitions regarding important strategic items. One such item is related to the architecture into which the master data management system is embedded. Although this is a topic related to enterprise architects only, the definition of business concepts involves both the enterprise architects and your business goals. The business needs to outline its vision, and enterprise architects must put that vision into the context of technical feasibility.

One such concept that occurs often is the definition of views about persons. What does a person represent for your enterprise? When are two persons truly the same person, and when must we keep them separate to provide distinct reports for each person? This vision includes the definition of semantics and the conceptual enterprise-wide data model so that data can be put in perspective within the greater context of the entire organization.

You also must consider either the existence of appropriate business domain models, use cases, and business processes or their appropriate creation within the elaboration phase.

- Elaboration

  Although the discovery phase is largely dedicated to the setup of the project, the *elaboration phase* moves into the analysis component of the project. If it has not happened already, and before starting the project, now is the time to analyze the semantics of the data and to create the business domain model, related use cases, and business processes. Ideally, a master data management solution project enters this phase with those artifacts already available.

  After completing these business-related aspects, move to the implementation cycle and build the required design models. In this phase, you create the data mappings and service design deliverables that define the behavior of the master data management system. In addition, you must allocate sufficient time to define core concepts, such as the search strategy, data filtering, security, and others.

  Consider the following approach:

  a. Define a key concept with business domain experts.
  b. Present this key concept to a larger group of people or teams that adopt the concept.
  c. Gather feedback on the draft concept.
  d. Refine the concept.
  e. Submit a proof of concept.
  f. Evaluate the results of the proof of concept.

g. Refine the concept.

h. Release the concept.

Typical issues addressed by a master data management system are data deduplication or suspect duplicate processing as named within InfoSphere MDM Server. The success of a master data management system is based largely upon data quality. The higher the quality, the greater the deduplication success. So, if it has not happened yet, plan and carry out data quality analysis by profiling the existing data.

- Configuration

  The *configuration phase* refers to the true coding phase. It takes all of the deliverables that were created during the elaboration phase and creates the code that is required to perform these functions. In this phase, detailed technical specifications are created and previous analysis and implementation models are enhanced and enriched with technical details.

  As outlined in this book, the aim is a model-driven development (MDD) methodology. An MDD methodology can provide for the documentation that is required within such projects and can support the generation of the required code artifacts. As a result, this phase is probably shorter than normal, at the expense of a longer initial elaboration cycle. The time that is required for elaboration is normal for the subsequent phases of the roadmap.

- Transition

  The *transition phase* includes testing and going live with the new system. Coming from an MDD perspective, use a test-driven development approach, where unit tests build the foundation for development. Based on the business domain and appropriate structuring of the business domain into smaller units, each developer assumes the role of a subject matter expert on a small piece of business logic. Therefore, the unit tests need to include a substantial amount of business logic. This approach works toward the tests that occur later, where the focus is more on business logic, and avoids running too many development and test cycles.

  Testing in this phase can include the following steps:

  - Unit tests
  - Component tests
  - Component integration tests
  - System integration tests
  - User acceptance tests
  - Performance tests

10.3 Setting up and executing a project

Model-driven development (MDD) or domain-driven design (DDD) does not dictate the method you must use. Instead, MDD or DDD supports common methods using either a generic project development method, such as RUP, or a specific method, such as the IBM Service-Oriented Modeling and Architecture (IBM SOMA). Thus, you can apply these types of development to many different project styles. If all the prerequisites are satisfied, these types of development can cater to a fabric style in which everything happens in a strict and defined order. It also allows for an agile style with many small steps that occur dynamically and concurrently. Ensure that everyone involved in the project understands the chosen method and is flexible enough to carry out such an undertaking.

For a project manager, though, keep in mind that MDD and DDD provide opportunities to support usual activities. You can identify work products that must be created and obtain estimates for each of them, which then can be individually tracked. If the model connects properly with the requirements that you have obtained and if it also provides substantial traceability information, you can generate appropriate reports from the model that can support you with the tracking efforts.

Another part of your usual activities includes decision making, communication, and control. For example, within DDD, you make decisions based on and enforce these decisions using the domain concepts. Decisions that impact the domain are not outsourced to teams that are not responsible for the domain model.

DDD also requires effective communication, even more so than a normal project. It is vital that everyone uses the same terminology, based on the glossary and the domain model. After you initiate the project, you are faced with the task of tracking and controlling the project. The model embeds all capabilities regarding this requirement to make your job easier.

This section outlines the following elements related to project setup and execution:

- Methodology
- Philosophy
- Teams
- Procedures
- Roles
- Planning
- Reporting
- Tracking

10.3.1 Methodology

The standard IBM method used for master data management solution implementation projects is the IBM Business Intelligence method, which is part of the larger IBM Unified Method Framework. This methodology describes all phases within a project and, through a set of standard document templates, supports the delivery of the required output. The methodology provisions the following phases:

- Strategy and planning
- Solution outline
- Macro design
- Micro design
- Build
- Deploy

For a project that consists of multiple implementation cycles, you iterate through the last four phases.

MDD and DDD support most common methodologies. For the discussion here, it is presumed that you are knowledgeable about the most common methodologies. However, this section describes the fit with methodologies, based on two entirely different styles. We use the following styles as examples only, and an approach is not limited to these styles:

- Fabric or waterfall
- Agile

Fabric or waterfall

The *fabric* or *waterfall* approach is derived from the manufacturing industry. It has the distinct characteristic that everything takes place under specific conditions and for a known length of time. It is up to you to prepare the project within the conditions that are required. You are then rewarded with a project that follows a strict timeline.

This strict timeline, in itself, makes this approach appealing, especially because it changes one of the three project management variables (time, resources, and budget) to a constant. As a result, the project becomes easier to plan and more manageable. This timeline applies across all teams, from analysis to design, from the implementation team to the test team. It also includes all work products that these teams create, such as the models from which code is generated and test cases derived.

Although this approach sounds appealing, you must prepare a stable setup that excludes all variables from the project. Most importantly, this approach means that all prerequisites must be satisfied before you begin the project. Establishing these prerequisites and, again, incorporating them into the project plan and defining and setting them as a checkpoint can be a challenge.

During the project, you can achieve stability by setting up a rigid change management process at the start of the project, which is a preferred practice. Also, and more importantly, this approach requires all input to be complete and accurate enough so that you can commence analysis against those prerequisites. You cannot rule out changes entirely; however, you can avoid them or process them through a rigid change management process. Thoroughly analyze the impact of each change before estimating a change and agreeing to it. The key here is to be as complete and stable as possible. Every lack of detail requires reanalysis later and possibly modification to models and code, including test cases.

An as example, consider an industrial manufacturing process that is disrupted by the discovery of two parts that do not fit together when they are on the belt. The designer needs to be called in, the details and the manufacturing process changed, and the steps rerun. If changes are required in the fabric process, you face the same dilemma in software manufacturing. Initial, you require the following input, at a minimum:

- Glossary
- Requirements
- Use cases and business process
- System context and architectural overview

If this information is not available, then include phases that create the information for you, keeping in mind that each of these phases must be completed and each deliverable met.

Although the glossary cannot be complete from the beginning, it is important for a glossary to exist before an analysis team reviews the requirements, use cases, and other information. The more strictly the work products adhere to the definitions provided in the glossary, the easier it becomes for those personnel involved in the project to communicate with each other, to understand each point made, and to avoid misinterpretation.

The requirements are split into functional requirements that describe the actual behavior of the system to be built and non-functional requirements that provide information about the conditions under which the system must perform. Requirements need to be precise and clear so that the analysis team can gather details before designing the final solution. Stakeholders need to sign off the requirement, so that the requirements can be taken as a baseline for the analysis.

Use cases and business processes are a common way of describing requirements and of outlining the wanted system behavior. They represent one input into service design. Changing use cases while the service design is progressing can lead to a service design that does not satisfy system requirements. As a result, you might end up creating more services than are required or provide services with a hidden intent. Such hidden intent is the result of service designs that were created previously and have dependencies. Changes in service designs and implemented intent that varies from the original intent leads to hidden and non-obvious intent. This situation is challenging. Typically, project members as the questions, “What is this made for?” or “Why has it been done this way and not another way?” So, these types of questions lead to additional work, because project members prefer to redo steps rather than adopt new conditions.

In addition to business-related work products, architectural input is required before designing the system. Work products, such as the system context and architectural overview, must be stable. Any change in the system context represents a major shift in scope. This shift can be good if the scope is reduced. However, the shift can be problematic if the scope is increased and includes systems that you did not acknowledge so far. A shift in scope is a challenge with respect to the design and often leads to refactoring. The outcome is the impact on the manufacturing process and all of the implications of refactoring, such as restarting from the beginning or, in the worst case, considering a dramatic change of scope.

When all this information is available, completed, and signed off, you can enter the next phase of designing the system. For the most successful implementation, prepare design artifacts based on the preliminary information obtained from early versions of the design artifacts. For example, you can prepare and evaluate a few of the common concepts that you require, such as error handling, logging, searching, and handling of business key factors. The goal is to achieve a stable design before passing the design to the next team, in this case the development team.

These concepts can affect your process at any time during the fabric phase. Using the manufacturing example, assume that you plan to run five times over the conveyor belt to finish the system, and you discover an inefficiency or incorrect design in error handling during the fourth iteration. You impact a few aspects of what you completed, delivered, and possibly tested. Such rework, therefore, triggers a rerun of the same fabric iteration or requires that you break out of that static fabric principle.

Agile

Agile methods are powerful because they do not assume that all the required elements are available at the start of the project. Instead, agile methods anticipate constant change, new insights to be discovered, and rework to be considered and dealt with.

The model-driven approach outlined in this book can support an agile process because there is one large, consistent model. Using this model, you can identify dependencies between the various models and determine the impact that they have on other parts of the model. This method helps when several parts must be changed because of new insights that are gained, errors that are eliminated, or changes that are processed. It allows for short feedback cycles and quick turnaround of rework. Whether such rework is based on a change in requirements, new insights discovered while the project progressed, or errors discovered, does not matter. The key, however, is that you plan for such rework from the beginning. This method is different from the fabric approach, where rework is either not an option or is planned as a final iteration with the sole purpose of completing all open issues.

10.3.2 Philosophy

The introduction of an MDD methodology also suggests that you must be aware of, and honor, the project culture, so that the best possible results are achieved. Although this book is not about team dynamics, this section provides information about the following main issues in introducing an MDD methodology:

- Decisions
- Communication

Decisions

Decisions are an important factor in any methodology. They must be made, and several decisions impact others. So, who decides? Is it the enterprise architect or the lead architect, the project manager, or someone entirely different? All of these persons are decision makers. What is important is to set the right borders and then operate within them.

If you focus on business-related and business-driven decisions, the borders are not clear. Everyone deals with business decisions with respect to the project, which makes decisions in this area difficult. So, the question to ask is, “What happens when we discover new information and insights about the business domain?” This question might be asked by a developer who requires additional information or by a business person who delivers new content that is approved for addition into the project. Another possibility is the identification of software bugs or errors in the design that might require adjustments to the business domain.

Such information is best evaluated by a dedicated business domain team that has all the domain insights. It is within this team that the information is aggregated and evaluated. All domain-related decisions are driven by a trusted team. Enterprise and component architects are consulted by this team to ensure that the decisions conform with the architecture as outlined. Project managers are informed of the team’s decisions so that the scope of any change is rolled out at the appropriate time. Still, the only team to determine whether a change to the business domain is required is the business domain team.

The centralization of the project has its benefits, because the business domain team has all of the insights and knowledge about the entire domain, rather than specific areas only. Thus, the impact of changing any part of the domain is best determined by this team. Because this team is largely independent of the technology that underlies the system, its focus is clear cut and allows for a proper focus and specialization of business in conjunction with only the most imminent technology aspects.

The overall skill set of this team is unique. Although the team members operate predominantly on the business side, they also have an understanding of the underlying technology that supports the system. With this team, you can truly bridge the gap between business and IT. Design artifacts can be influenced by the underlying technology, for example, when you integrate InfoSphere MDM Server with its predefined transactional service framework and predefined but extensible data model. Regarding the existing data model, design artifacts occasionally result in a domain model that can be derived by the data model of the standard product. However, in terms of the integration of a master data management system, it is key to acknowledge the customers’ business requirements first, before determining how those requirements best fit into existing structures.

Thus, the business domain team interacts closely with other teams. These teams might be directly or indirectly related to the business domain model, or they might interact with the business domain objects or business task services. Therefore, the business domain team is considered the connection between business and IT and also the connection within every project. It holds together all teams and roles and is viewed as the trusted advisor of the project and, in particular, for the project manager who is responsible for it.

After the decisions are made, the next challenge is to communicate these decisions to the team.

Communication

When setting up a project, consider the following roles with which communication is important:

- Customers, in particular, the business
- Architects, that is, enterprise architects and component-related architects
- Project management, as the personnel ultimately responsible for the project
- Design teams, dealing predominantly with business but also technical issues
- Development teams, with respect to all technical issues
- Test teams, providing additional insights regarding the definition of test cases

Communication with *customers* occurs so that you can gather requirements and validate established business concepts. As such, this type of communication is two-way communication. Requirements must be delivered to the business domain team. They also need to be evaluated, double-checked, and refined. After the requirements are clear, the business domain can be developed. It is vital to communicate constantly with the business to validate the concepts that are established and embedded in the business domain model. This type of communication is proactive communication that is initiated by the business domain team.

Interaction with *architects* takes place in a similar manner, although regarding different content. You must consult with enterprise architects to acquire additional insights into the business domain and the relationship and relevance of the business domain across the entire enterprise. System solution and component architects are consulted slightly differently. First, the business domain team must ensure that the architects allocate components and interactions based on the business domain model, including the layering concepts that define where the domain model is and the definition for interaction and communication between the system components. The architects must keep current about the development of the domain model. They also require at least a high-level understanding of the business domain to be able to make decisions about impacts that are imposed by the model and about performance-related activities.

There is also two-way communication with *project management*. Most importantly, you need to include this communication in the project plan. Then, the project plan needs to be communicated clearly, including scope definitions. The larger the project, the more clear the scope needs to be and the fewer changes to the scope are necessary. Because changes cannot be avoided in most projects, you need a clear process for dealing with any changes.

For the project manager, the two most important teams to consult are the architects and the business domain experts. However, it is the business domain team that has the deepest insights into both the business and technical aspects. The business domain team is also the owner of one of the most significant models with which many other models have a relationship. These relationships, in conjunction with a model-driven approach, enables the team to better judge the impact a change might have on the project. Thus, project managers need to consider working closely together with the business domain team and need to proactively integrate them into the decision making process. Alternatively, it is the business domain team that typically identifies project issues first and, thus, is responsible for reporting possible issues to the project manager to determine the necessary action.

Another important communication boundary is related to the other *design teams*. The business domain team predominantly focuses on business-related aspects of the model. In traditional development methodologies, such as RUP, this focus occurs in the macro design phase. Therefore, other design teams must deal with the micro design of the solution. Because the micro design is the next logical step, in addition to refinement of the macro design, it is vital for both teams to communicate closely. However, unlike other communications, it is the responsibility of the business domain team to first establish communication with the micro design teams, who require an understanding of the business domain in general.

Contrary to the business domain team, design teams focus only on specific aspects of the project, depending on the releases that are planned and based on the definition of subdomains. After the design teams begin their work, there are multiple feedback cycles. The first cycle is related to questions that arise directly during the micro design phase.

Similar to the communication with the micro design teams, ensure the establishment of communication channels with the *development teams* that are implementing the solution. This communication cycle is related to development teams that can post questions to the micro design teams. This communication might, in turn, cause further business domain-related questions. To avoid communication issues, ensure that there are direct communications between the teams that are affected. For example, after the development team encounters an issue, they report it to the micro design team. The micro design team determines that the issue is a business domain-related issue that requires changes. As a result, for ultimate project success, these teams need to establish direct communications.

Another important team to communicate with is the *test team*. It is the responsibility of the business domain team to first educate the test team about important concepts of the business domain. After the test team starts its work, it is most likely responsible for analyzing issues and coordinating test, development, design, and possibly project management activities, if a major defect surfaces that causes severe delays in the project delivery.

Independent of team-specific communications, it is best to communicate with all teams that are affected by a concept related to the business domain. Communications need to be proactive from the trusted team that is driving decisions. Alternatively, feedback needs to be gathered from those dependent teams as well. Communicate concepts related to the business domain thoroughly and consistently to allow for early verification of the concepts and to allow for early identification of potential issues that might require special consideration or attention by a project member.

Use the following general procedures to establish concepts related to the business domain:

1. Gather domain insights from the business by the domain team.
2. Work out the base concepts by the domain team.
3. Present these concepts to all dependent teams, and gather their feedback.
4. Rework the concepts based on the feedback provided and finalize them.
5. Present the final concept to all teams and set up a prototype.
6. Design and develop the prototype, and evaluate the results obtained from it.
7. Refine the concepts, based on information obtained from the prototype.
8. Communicate changes back to all dependent teams again.
9. Continue with micro design and eventually development and test activities.

10.3.3 Teams

After you address communication issues for your master data management solution implementation project, you must establish the correct teams with the correct skills. In addition, ensure that team members are aligned to achieve the goal most efficiently. Is everyone responsible for everything? Should you establish subject matter experts and allow a degree of specialization? The goal in a fast moving, relatively agile model-driven methodology is to establish a team infrastructure that can respond quickly to changes. If all team members are equal, you need to allocate someone to take care of another task before continuing with that person's original activities. This disruptive work negatively impacts quality. So, what does an agile team infrastructure look like?

To begin, have a strong understanding of the overall project structure and interactions that are required with other teams that require information. Time poses an issue in that one team is usually operating in a different phase than another team. In most project methodologies, there are natural, sequential dependencies, for example, related to macro and micro design. If both designs are carried out by different teams, the macro design team is ready to commence with the second phase of its macro design, while the micro design team is starting with the first phase. The consequence is that most teams must work with adjacent teams and also continue to provide support for other product releases they create.

These factors combined represent a serious challenge for any project team. The following key roles attempt to address these issues:

- Team lead
- Scout
- Lead designer
- Lead designer and designers
- Subject matter expert (SME)

First, establish the *team lead*. The responsibilities of the team lead include tracking the design and reporting to project management. This person is also accountable for any administrative tasks that are outside of the usual team activities, for example, service design. It is here that escalations are started when the unexpected happens or when other tasks need attention, despite a good team setup. The team lead also carries out marketing-related activities and positions the results that are achieved by the team within the overall project. The benefit is increased visibility and acceptance within a larger project infrastructure that comprises multiple teams. In addition, you need to establish strong communications between all team leads and similar team structures throughout the entire project.

Scouts are a special concept. This individual or, in a large project, members of a small team, has the best understanding of the solution and its fit into the overall system landscape. Scouts have architectural and business skills and a good understanding of the design process and the use of design artifacts in other teams, such as development and test. Scouts carry out special tasks, such as reviewing artifacts delivered by other teams, checking them for consistency, and determining whether they require a change or clarification in the architecture, design, development, and test areas. Scouts are always ahead of the design and establish the design guidelines and patterns. Because scouts have an exceptional understanding of the overall project environment, they are also responsible for establishing the processes and procedures within the methodology. They interact with the test and defect management teams and relay information as necessary. The scout and team lead adjust the guidelines and patterns during the project. In addition, they help out with design-related activities.

The *lead designer* and *designers* are usually team members with the key task of carrying out the design itself, with the lead designer usually the most experienced among them. The designers are typically dedicated to the current phase, but bug fixing or similar activities might require their attention and dedicate them to other phases. Experience and a thorough understanding of design-related activities distinguishes the lead designer from the other designers. A designer from today can easily become a lead designer tomorrow, using the experience gained from each project.

Another concept is that of *SMEs*. Any team member might assume the role of an SME for any number of topics that the team faces. This responsibility remains independent of the point in time or the phase to which the design team is currently dedicated. The relevancy lies in any subject that occurs or resurfaces during the design cycle (before the design team builds the skills that are required for the design itself), after the design cycle (when there are questions from development), or during testing (when bugs are reported and require fixing). Such assignment also supports the identification of the responsible team member with the task assigned.

Types of SMEs can be divided into the following general categories:

- Business subdomains

  The business domain can be split into business subdomains. These subdomains provide logical boundaries for SMEs so that each subdomain is assigned to one SME.

- Key application concepts

  Concepts include the search strategy, error handling, business rules, lifecycle, and business key handling.

- Project setup

  Project setup requires interaction across team boundaries, such as macro design, micro design, development, and test. This setup also includes related activities, such as those activities that involve defect management, which determine interfaces between components and teams, and that require tooling-related expertise.

10.3.4 Procedures

Established procedures within each project, within each team, and across team boundaries are also important. Although the communication aspect is important, you must define how the teams deal with the artifacts that are produced and how they consume those artifacts to create their own artifacts for the next dependent teams. This aspect is where most methodologies lack detail. Often, they present phases in a black and white style where the project includes hard transitions from one phase into the next, as shown in Figure 10-5.

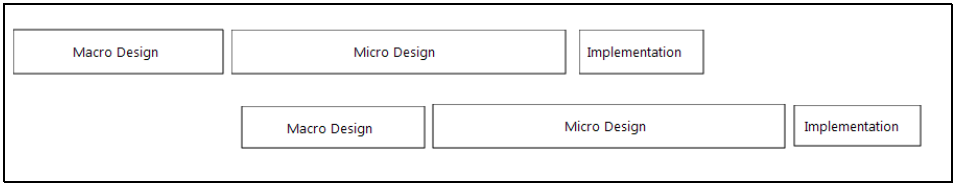

Figure 10-5 Methodology phases and dependencies across team boundaries with hard transitions

A *hard transition* is when an artifact is created by one team, then all involvement and responsibility ends when the artifact is passed to another team for use in another phase. This type of transition is rarely possible, because the initial team might have questions about the deliverable or might need to fix bugs.

Consider a softer approach that provides for transition phases and that makes the methodology you intend on adopting more realistic. This section describes the procedure based on the example of a micro design team. Despite the fact that all roles and the phases are described based on the activities of the micro design team, you can replace roles and phases and adopt them to the requirements for other teams. The point is to standardize the delivery of artifacts that are created across team boundaries. The benefit of this approach is that nothing is forgotten. This process also increases the quality of the artifacts of each team that adopts this process.

Each phase in a development methodology looks similar to what is shown in Figure 10-6.

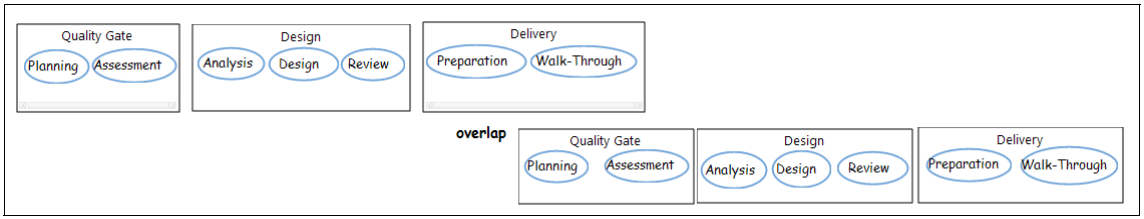

Figure 10-6 Methodology phases and their dependencies across team boundaries with softer transitions

Transitions between individual phases in a methodology are clear cut, at least on paper. In reality, there is much greater subtlety. A *gray zone*, rather than a hard transition, can help with transitions. We describe this development methodology based on the tasks of a micro design team, but you can adopt this process for any phase transitions.

A traditional method describes one phase, but the development methodology phases consist of the following distinct phases. The leading and trailing phases represent the gray zones, and the middle phase carries the load and represents the activities of a traditional phase.

- Quality gate
- Design
- Delivery

Quality gate

The *quality gate* phase starts early in the process. This phase allows for a smooth transition between the larger project phases and logically belongs to two phases of two teams that are involved. An amount of time is allocated for both teams to work on the same deliverable to ensure a smooth transition between teams.

With respect to the receiving team, although other team members can still work on the design from the previous phase, several dedicated team members can preview the contents of the next deliverable from the predecessor team. With respect to the macro design team, a few people are dedicated to the transition, and other people commence with the activities of the next phase. This type of transition period provides the following benefits:

- Gaining an understanding of the content of the deliverable
- Identifying skill gaps, inconsistencies, or faults in the deliverables as early as possible

Carefully executing this step enables the predecessor team to fix any issues before delivery and avoid an inefficient defect or change process.

The quality gate phase consists of the following major groups of activities:

- Planning

  The planning group of activities provides a preview of the deliverables from the predecessor team. Based on the results of a preview, you can plan which designer is working on which subject, determine whether you require an SME for a large and strategic topic, and provide the team lead and project management with an update on this information. Even at this stage, you must be able to estimate whether the next iteration can be concluded in the planned amount of time or if something must be moved. The goal is to get an overview of the deliverable. The exact content is assessed with the next group of activities.

- Assessment

  During the assessment activities, you take a closer look at every aspect of the deliverable. For example, if you are designing services as part of the micro design team, ensure that you have all of the information about the service signatures predefined by the business domain or macro design team. You require everything that you need to build the service flow inside the interface. Therefore, you must understand the motivation behind every service provider signature, and you need information about which use case or other business driver represents the requirements for this situation.

  This phase is typically when errors can be detected in the deliverables that are received. The end of this group of activities also concludes the transition between the major project phases, such as macro and micro design. Successfully passing this assessment concludes the predecessor phase.

Design

In the micro *design* phase, the required artifacts are modeled. As you enter this phase, you can again separate it into the following distinct groups of activities:

- Analysis

  The key to proper design is to first understand the business content and the motivation behind the business concepts. Based on that input, you can determine exactly what must be done. Although this understanding is important, there are further aspects to be considered, including the evaluation of reusable service methods. A preferred practice is to model with reuse in mind.

In the analysis activities, you determine whether you have something reusable in the design, if you can create something reusable, or if you must carry out custom design with a single purpose only. During this phase, questions can still be raised before commencing with the design of the service execution flow itself.

- Design

  Within the design, each designer performs a check with respect to the impact that the design has on other artifacts, including use cases, rules, and the data model. Complex service must be reviewed with respect to its fit into the overall and component architecture. In some instances, there might still be a required lower layer service that is missing, for example, if the system is built on a multilayer architecture. When this situation occurs, inform the team that is responsible for that layer, and continue to work based on assumptions about the required service. The outcome of this group of activities is a design that is ready for review.

- Review

  Each artifact that is designed needs to be reviewed before release to the next team, to the development phase, or the test phase if you are in development. Modern methodologies, such as the agile method, already propose good methods for carrying out these reviews in the form of peer reviews. Someone else from the team needs to review the design. Questions show where the design is lacking clarity and where improvement is needed. After the review of all the designed services is complete, continue with delivery to the next team.

Delivery

Comparable to the quality gate phase at the beginning of the design process, another transition phase is introduced to the next team in the *delivery* phase. Effectively, it is in this phase where the quality gate process of the successor team starts in parallel with your executing the delivery phase. This phase consists of the following activities:

- Preparation

  It is a preferred practice to run several automated design checks, similar to when development executes unit testing before releasing code to test teams. Furthermore, with the change log information embedded in the design during the design phase, you eventually need to create release notes, based on the embedded change log information. This creation is most likely done by running reports on the model.

There are design steps that introduce new aspects to the solution. It is a good idea to create documents that outline the topic, including the business reason, fit into the architecture, the solution as designed, and an explanation of the design guidelines. These documents can be used to carry the knowledge that is obtained to the next team and also to allow peers to dive into the topic quickly. For example, if a replacement for a project team member needs to become familiar with these topics quickly.

- Walk through

  The walk through group of activities usually starts with a meeting in which the artifacts created and the concepts behind them are explained. It concludes with a deeper review by the successor team, according to the quality phase. With the conclusion of that team's quality gate phase, you can conclude your own phase and continue to the next phase. Because all team members are not involved with these activities, they can commence with the early activities of the next phase and carry out their review of the next deliverables received.

10.3.5 Roles

The development process revolves around describing activities that must be carried out during each iteration, regardless of the methodology adopted. These activities are grouped into logical phases and are carried out by team members in specific roles. For each activity, each role has a certain involvement that must be defined. One person on one team can assume multiple roles, depending on the team size. With a master data management project, you might require additional roles.

The following roles are adopted from the description about team setup, enriched by a few additional supporting roles:

- Business domain (macro design) architect

  This role can be adopted based on which team you are looking at. If it is the development team, this role must be replaced by the micro design architect or lead designer. That person is required for an initial walk through of the artifacts delivered in order for the team to have an overview of the delivery. Later, that person is needed to address and respond to ad hoc questions.

- Team lead

  The team lead's responsibilities are largely in the areas of tracking and reporting on this process. This person communicates with the project management team and is accountable for the overall result and outcome of each design cycle.

- Lead designer

  Complementary to the team lead, the lead designer focuses entirely on the content of the deliverables. The lead designer overlooks and coordinates all team-related activities and supports the team lead in planning, tracking, and reporting.

- SME

  Each team member can assume the role of an SME, focusing on more dedicated aspects of the overall solution. Expertise can revolve around architectural concepts, such as logging and error handling. An SME might include business subdomains or packages that are defined against the business domain model. They might also include design concepts, such as overseeing and governing the change log to be correctly embedded in the design.

- Development team lead

  This role can be adopted, depending on the specific team. If you are part of the development team, this role includes the test team lead. Effectively, the development team lead is always the team lead of the team that is consuming the deliverables created in your team. The goal is to walk through the deliverables with this person during transition into the next project phase to ensure a common understanding about the content that is delivered, reduce questions later, and allow for possible rework and refinement of the deliverable before delivery.

- Component architect

  It is vital that component architects are kept informed and always have a thorough understanding of the solution design and its implementation.

- Business representative

  Because you are developing a business-oriented solution, you might have questions about the business itself. Although the business domain expert typically needs to respond to most questions, it might be helpful to have several business representatives directly involved in special matters. Plan on needing to involve a business representative at any time, especially on the topics of lifecycle and business rules, so that questions and clarifications are communicated.

Several distinct roles must be filled during a typical master data management project. For a project with an estimated size of 300 - 500 person days, the team might require five to seven full-time equivalent members in the following roles:

- Master data management business architect

  This architect obtains and evaluates the business requirements and creates the data model extensions and additions based on requirements. The architect establishes the composite service design, including definition of the business rules that are required. A master data management business architect also supports creating and performing tests.

- Master data management IT architect

  This architect adapts the master data management architecture to the business requirements in the context of the larger system architecture. This person is responsible for the technical model and service design and can also be involved with quality assurance tasks.

- Master data management specialist

  This specialist implements the model extensions, additions, and services as designed. Another task is that this specialist creates unit tests to support the larger test activities in the project.

- Master data management infrastructure specialist

  This specialist is responsible for product installation.

- Master data management product implementation manager

  This manager supports the project manager for a large-scale project in which the master data management components are a small subset of the activities to be carried out.

Involvements

Each team member in a dedicated role serves a special purpose and is involved in an activity from another perspective, either as a spectator or a participant who is responsible for a delivery. The following levels of involvement are defined:

- Accountable

  For each activity carried out, there is a team member who is held accountable. This person is not necessarily the person who is assigned to do the job but is perhaps a person in another role. For example, in the area of planning, the team lead is accountable. In the area of designing, however, the lead designer is accountable, and the designers are responsible but not accountable. Occasionally, the person who is accountable is also responsible for an activity, but initially all involvements need to be reviewed separately.

- Responsible

  Only one role can be responsible for an activity, although complementary roles can be performed by the same team member. Assuming responsibility means ensuring that all of the components that are required to perform the activity are available before the scheduled time.

- Informed

  Other team members must be informed. You might prepare them for a task coming up or simply enable a backup for your own activities. Keep project management and the component architect informed to avoid surprises. There might be multiple roles that must be informed about an activity.

- Consulted

  There might be several roles that can be consulted to obtain information that helps achieve a task. Typically, it is the role of the team member who is responsible for a task to draw in persons with whom you need to consult.

10.3.6 Planning

Defining a project plan requires multiple activities that are known to every project manager. This section describes our experience in project planning in MDD and DDD.

The definition of the total amount of time, budget, and resources that are required for the delivery of a master data management solution depends on the specific project situation. For example, the number of source systems from which master data is incorporated has a direct impact on the number of transformations and complexity of business processes that are required. The same concept applies to the number of downstream or consuming systems.

The following example lists the activities and effort directly related to InfoSphere MDM Server integration. It is presumed that the business domain model functions as the reference model does. Furthermore, enterprise team leader (ETL) processes and the adoption of external systems or transformation layers are excluded.

A simple master data management project, according to the simple batch-oriented architectural patterns, can be realized within approximately six months, assuming an effort of about 300 - 500 person days for the realization of master data management-specific tasks. These tasks include the definition of data model extensions and additions, corresponding data mappings, and composite service design for the batch load and batch synchronization. The effort also includes the implementation of the specified composite services, architectural advice, and the installation and configuration of the product.

The biggest insight into your project plan comes from building the domain model in conjunction with the business process model. Both models contribute to the details of the project plan. First, the domain model is built. Then, you need to split the domain model into smaller, more manageable pieces, referred to as *subdomains*, *modules*, or *packages*. The benefit of subdomains is that they identify work packages that can be moved into the work breakdown structure of the project plan.

Consider a domain model where you have subdomains named *person*, *organization*, *address*, and *contract*, as illustrated in Figure 10-7.

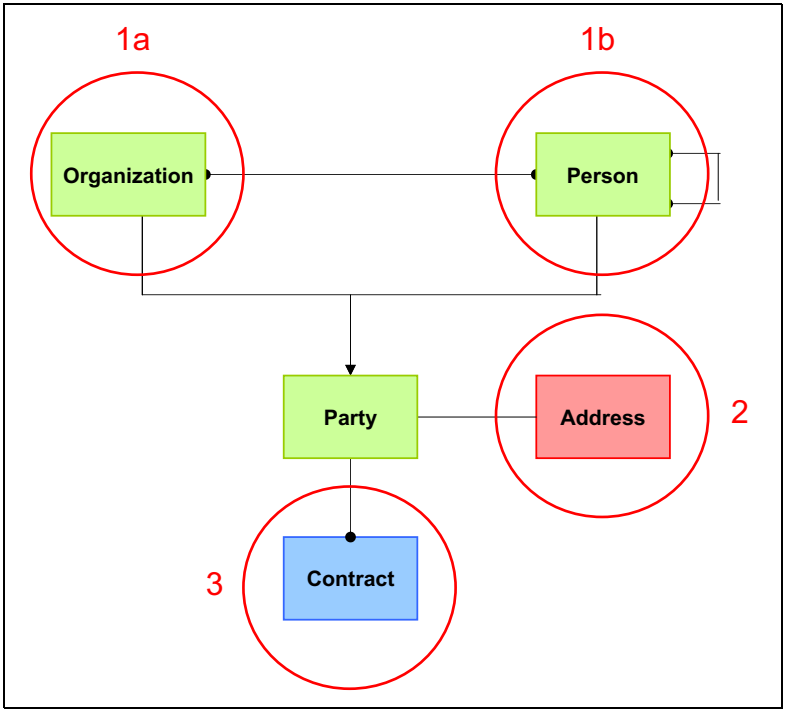

Figure 10-7 Identification of packages and their dependencies

In this example, you are aware of the subdomains. They are large excerpts of the domain model that they represent. Some of these subdomains are interconnected and some are not. The key, however, is that these subdomains are independent, and can be designed, developed, and tested independently, including all of the business requirements by which you defined this model. Based on this insight, you can determine the individual components to include in the work breakdown structure. Then, you need to put these individual components into a workable sequence, because that information is currently missing from your project plan.

You derive this missing information for the project plan by applying the insights gained from the business process. You can use this process to identify both sequential and parallel activities. For example, for the domain model shown in Figure 10-7 on page 530, which has subdomains, the person and organization subdomains do not need to have a sequential relationship. They can be developed entirely in parallel. The address domain, however, can be included only after the person and organization subdomains exist. Therefore, the address subdomain has a sequential relationship and needs to strictly follow the person and organization subdomains. You can design and develop the address subdomain first, but you cannot achieve end-to-end testability. The same relationship applies to the contract subdomain, in that it is dependent on the person and
address subdomains.

The business process model also highlights that searches exist before anything can be updated, deleted, and possibly even added. Because of that knowledge, you can include the search functionality in the project plan earlier and with a higher priority.

Now, the advantage of deriving insights from and steering a project through the insights gained from models that are created by the team is clear. These factors become most apparent in an MDD methodology. You can use them to achieve maximum effectiveness in the development of the solution.

Occasionally, outside factors can demand another prioritization, which can dramatically upset the preferred method of moving forward, as illustrated in the following scenario.

Using the domain model, imagine a decision to focus on the initial load-relevant activities first, so that all objects to the master data management database can be added. The impact is that you cannot focus on individual subdomains first but that you need to have all entities in focus. This impact, in turn, makes it much more difficult for everyone involved in the project, because of the business aspects to consider at any time. Also, rework is more likely to occur in such situations, because of the inability to have a deep look into the subdomains. Therefore, you can focus only on load-related aspects and can gain additional insights later, when investigating updated related requirements.

10.3.7 Reporting

Similar to the realization that model-driven projects can provide additional input into project planning, they need to be accessed to obtain information about their status, too. Based on the information captured, you can draw reports directly from the models that are created to determine the most accurate metrics so that you can avoid manually creating additional spreadsheets or similar activity.

Spreadsheets for a project are relevant and are occasionally required. However, the use of the model and your tools to generate reports, possibly in the form of spreadsheets directly from the model, is often more effective. In contrast, consider the occasional project manager who requests input from a team lead because the numbers are inconsistent. Then, the project manager generates new information from the input and changes the meaning of that information in the process, producing numbers that do not match the numbers of an adjacent team, because of a differing interpretation of the same data.

Creating reports that derive information from the model and that populate the values directly into documents that are required for reporting avoids these types of errors. Because project managers do not commonly create reports in UML, they might require assistance to acquire the necessary information and define the reports around those requirements.

10.3.8 Tracking

Traceability provides those people that work on the project with a means of tracking nearly any activity, be it the existence of an architectural component or entities of the domain model or the original requirement. This aspect is often underestimated and not used to its full potential. Reporting and tracking tasks must be based on such model artifacts, and an impact analysis needs to be used to determine the side affects that a change on the system might cause.

Apart from tracing specific elements, traceability is another important aspect of requirements, as well as defect and change management. Any change that is applied to the model must be traceable to a requirement or a defect. Thus, traceability also includes the definition and maintenance of change logs and solid version control.

When a new version of a model is released, anyone who is interested in this model must be aware of the changes that are applied to this version in comparison with the previous version. Because versioning itself does not provide an easy means of tracing changes, models must include descriptive change logs that reference changed elements and must include mechanisms to identify new, deprecated, and removed elements. Also, depending upon the release cycle, a model can include elements that are not yet relevant to the software to be programmed. These elements must all be identifiable.

10.4 Tools selection

The question of which tools to select is important. From a theoretical point of view, the choice of tool is irrelevant. You can use any tool to reach your goals. However, how well can a tool support you and how effectively can you apply them to your business issues?

For best results, choose one or more tools to support the methodology in the best possible way. Consider all aspects of your solution, from the planning phase of project management, to the analysis, design, and implementation of the solution, possibly including the test phase, and continuing through to tracking and reporting on the progress of these steps, closing the loop to project management again. During this time, you typically have multiple, disparate teams that are creating a large and, in itself, consistent model.

You can use multiple tools or just one. If you use one tool, a dedicated UML modeling tool is preferable. If you select multiple tools, you can choose a highly specialized tool for a specific task. You can select a tool based on the features of the tool, whether you have experience with to the tool, or the price of the tool. Acquiring tool licenses can be costly, but is worth considering. Consider also the learning curve and the effectiveness of adopting multiple specialized tools.

You might decide on the most adequate tool for each deliverable. Although this choice creates the perfect result for the current deliverable, the consequence can be a struggle with interoperability between the tools, which deviates from the goal of creating a large, consistent model. Nonetheless, using such traditional tools is still a good choice, and many companies are reluctant to switch to other types of tools.

From our experience, it is difficult to create a consistent model using multiple tools, especially when they are not connected or are only loosely connected. A better choice is to use a single tool or highly integrated set of tools for all aspects of the MDD methodology. This choice provides the required consistency and also timely and simple access to information as needed. For anyone that works with or joining the project, access to information is critical. If you cannot interlink models, which makes them navigable, collecting the required information can be cumbersome.

Before going into the tool discussion in depth, keep in mind the following general activities that you must complete when implementing a master data management solution:

- Project management
- Architecture
- Business analysis

- Data modeling
- Data mapping
- Service design
- Implementation
- Testing

Regarding tools, do not focus only on using multiple tools versus a single tool. Such a comparison does not truly reflect the situation when deciding on tools during the early phases of a project. Instead, for example, base your decision on the tools that you know, what you can achieve with the tools, whether the tools address your needs, and whether better tools are available.

InfoSphere MDM Server provides a solid base for your master data management system. In addition, InfoSphere MDM Server Workbench provides tools that support development of extensions to InfoSphere MDM Server, which can help you with your overall system design. You can use InfoSphere MDM Server Workbench to define the wanted data model and transactions and then generates the code required to implement the InfoSphere MDM Server extensions. You can then use other tools to enhance the functions of InfoSphere MDM Server so that you can obtain the most desirable design for your system.

For our discussion, we distinguish between the following types of tools:

- Traditional tools
- Modeling tools

10.4.1 Traditional tools

When you use IBM InfoSphere MDM Server as a base for your MDM system, you do not automatically obtain the most desirable design tools with it. Although it has an MDM workbench, this tool does not cover all aspects of system design. So, the choice of tools is still up to you.

In this context, *traditional* means to rely on office tools to document and specify requirements. These tools represent a number of similar tools and products that are also available. The focus on standard office tools is merely an arbitrary limitation and focus on a few tools instead of creating an exorbitant list where we mostly state something similar for each individual tool. The advantages of traditional tools are the following:

- Availability: These tools are generally available to almost everyone in the organization.
- Adoption: Because of its widespread availability and continued usage, the adoption rate is high.

- Skills: Because of a high adoption rate and widespread usage, skills typically exist with little or no learning required.
- Flexibility: There are no formal restrictions to the usage of these tools, which increases the adoption rate.

As we show, some of these apparent advantages turn out to be serious obstacles on the way to consistency and traceability. Their disadvantages are:

- Lack of traceability: Information put into spreadsheets, text processors, and graphics development products, represent silos, much as the data silos we want to avoid within our MDM context. They are not linked to the designer's environment and cannot be checked against automatically.
- Lack of integration: Traditional tools frequently use binary or proprietary data formats. This not only means that information cannot be extracted easily and cannot be processed automatically, but also that these documents do not integrate well with most collaboration and configuration management tools.
- Lack of formalization: Whether it is a word processing tool or a spreadsheet application, neither tool typically provides a formalized basis, that is, a metamodel. You can add any type of information anywhere in a document. However, although this situation might be convenient from an author's perspective, it results in the interpretation of the document becoming a burden.

In our description about traditional tools, we focus on the areas of data modeling, data mapping and service design. Project management products could be a candidate for the project management-related activities, but in the case of architecture, implementation and testing the capabilities of traditional products is not robust. The types of traditional products we focus on here are the following ones:

- Spreadsheets
- Text processing
- Graphics development

Spreadsheets

There are a number widely used spreadsheet applications. You can do almost anything with them, from writing simple to structured text, calculating data in a spreadsheet, and building entire applications. Spreadsheets are used because of their flexibility and simplicity, which makes them a tool for all purposes. Also, many people are already familiar with using spreadsheets in their daily work routines. Spreadsheets include reporting sheets, planning sheets, financial analysis, and much more. This situation makes spreadsheets even more tangible, because the skills are there and so is the adoption of those tools.

However, the point we want to make is that spreadsheets are also part of the suggested standard MDM delivery methodology. Their suggested use covers the area of data mappings and, in part, data modeling when it comes to the description of table extensions and additions. It is here that we disagree and believe that there are better tools that can be used.

Figure 10-8 shows an example of spreadsheet-based data mapping.

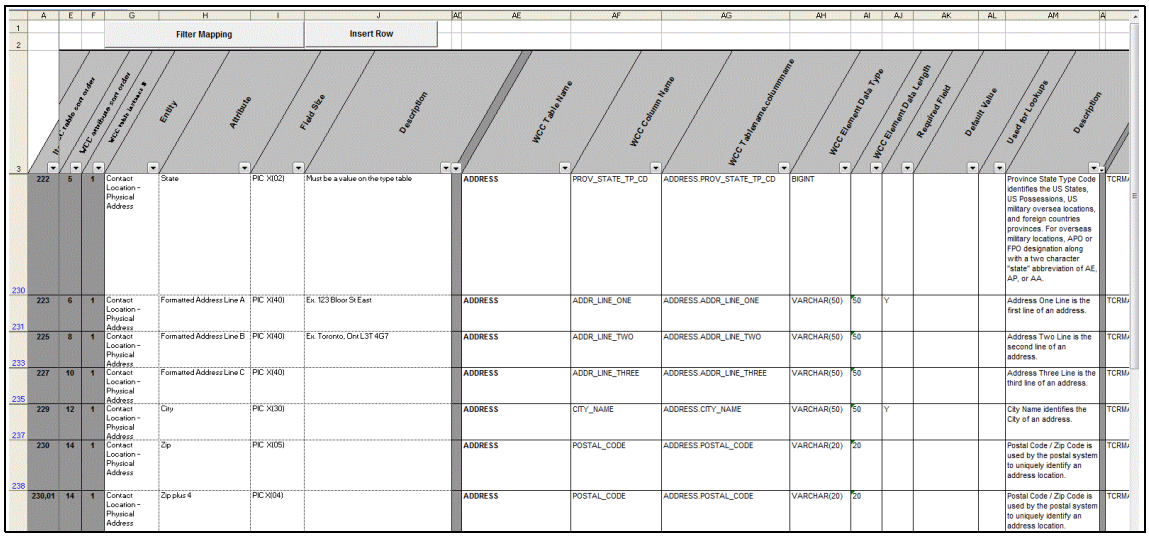

Figure 10-8 Spreadsheet-based data mapping

Spreadsheets represent multi-dimensional table structures to document the data model and the related data mapping. Each object and attribute provided by the MDM Server is listed in the rows of the spreadsheet and typically listed on the right side. On the left side, you typically enter the source and target system attributes. When mapping multiple attributes to the same MDM attribute, you can simply copy the appropriate row and add it to the source or target systems attribute. However, there are a few issues related to this action:

- Splits and aggregations are not clearly visible.
- There are varying defaults and mandatory or optional designators for service operations.
- Mapping to multiple instances is not recognizable.

Splits and aggregations are often required, especially with the ETL jobs required to populate the required input data structures. For example, the source system could store all address data, including street, postal code, and house number, in a single line.

Conversely, MDM Server provides multiple purpose built attributes that can hold the individual pieces of information that comprise an address. It is here that we require information that many attributes of a single instance of the source systems DataObject split into many attributes on the same instance of one or more MDM Server objects. Similar examples account for the aggregation aspect, where a source system attribute that holds the street name could provide for a field length of 25 characters compared to the 100 characters supported by each of the four address lines provided by MDM Server. It is here where you can aggregate the many 25 character lines of the source system into one MDM Server address line.

Another, not so simple, aspect about this form of static data mapping sheets includes aspects of varying defaults, and some fields that are sometimes mandatory and other times are not, depending on the service operation in which they are started. Defaults are required, but whether varying defaults belong in a data mapping sheet can be debated. You could also see and interpret them as rules that populate a field that depend on the values of one or more other fields. However, because defaults are part of the data mapping sheet, and they must be defined somewhere to provide a consistent master data repository, you must consider them. Our suggestion is to consider them as early in the process as possible, so the point in time when data mapping is performed is indeed a good one.

A problem of a different kind poses attributes that vary from service operation to service operation. One example for such a scenario is when an attribute is mandatory in an add service, but can be skipped during an update. In respect to spreadsheet-based data mapping, what do you enter? Is it a “Yes” or a “No”? How do you explain to developers that implement the services and use the spreadsheet that mandatory does not always mean mandatory? You must consider the different options developers have in adhering to the specifications, with one of the options certainly being an input validation before service execution. If that is done, then our optional field is never optional as soon as the input validation requires that mandatory field.

Although the two previous examples can be solved by adding some comments into special columns of the data mapping spreadsheet, this situation becomes more tricky with the third example. In this example, you have different types of data where you map the different types against the same tables and attributes in MDM Server. In our small case study, we have two types of policies, a LifeInsurance policy and a Property policy. Assuming they are both stored in the same source system and use the same data structures but different type codes, and we still map them to the same MDM Server table, which is Contract, we no longer know whether the mapping refers to the same or a different instance of the Contract table. This detail is crucial for everyone involved, from the designers, the developers, and the testers of the services that support the adding and updating of those policies. This situation cannot be simply described in words anymore and requires a more sophisticated mapping approach. This point is also one of the more difficult ones in any project, because it is what the business logic revolves around.

Before you can decide whether you have the correct tool, you must be clear about your objectives. So, outline the intention behind the data mapping as defined in the MDM delivery methodology. The primary objectives are the following ones:

- Determine the optimal placement for each customer data element inside MDM Server.
- Identify the data gaps between source systems and MDM Server.
- Identify differences in attribute format, for example, length and type.
- Identify cardinality differences, for example, the source has more than one occurrence and WCC has only one.

Secondary objectives include the following ones:

- Decide on the best way to fill the gap by defining Attribute hijacking, Extensions, Additions, or by not mapping the attribute at all and rather place it on an external system.
- Provide insights into the service design.

From our point of view, the key aspect is related to the requirement to provide insight into the service design mentioned in the secondary objectives. We believe that this aspect is crucial in the MDM solution implementation and deserves a much higher degree of attention.

So, although this spreadsheet as a tool is an option in smaller scale projects, carefully consider its usage in larger scale projects, because much of the information that needs to be spread around the implementation teams cannot easily be extracted from it. It still requires much interpretation and explanation. We prefer clear, concise, and complete documentation instead.

Text processing

Any text editor is, in fact, a universal tool that is used almost anywhere and anytime.

Consider your objectives again. After the data mapping is completed, you must define the services required to support the functionality of the system while considering the data mapping definitions, because they impact the choice of base services you can use to populate the MDM data store. The services to be defined thus represent a collection of standard services that exist in the product and arrange them to a flow composition that supports the requirements in relation to the business tasks.

When you consider all this information as individual pieces, you see some artifacts that are seen as separate deliverables in most projects. For each composite service design, there are the following deliverables:

- Functional Overview
- Scope of the Transaction
- Requirements
- Business Rules
- Assumptions
- Base Services
- Event Scenarios

A typical written representation could look like the small excerpt depicted in Figure 10-9.

Scenario 1 – ***Create a new name (not legal)***

Description:

The record provided contains an Organization name or Person name that has been newly created on the source CIN system and that did not previously exist on MDM Server.

Business Rules:

BR – 22 If no StartDate is provided the date and time of loading will be set.
BR – 23 If a duplicate name and type combination is provided, not data will be added.
BR – 24 If any error occurs during any of the steps, no data will be added.
BR – 25 If a new name is related to another existing name, a relationship must be created
BR – 26 A duplicate is determined by looking at all name attributes provided.

Assumptions:

A – 1 CIN can identify the business event as the addition of a new name.
A – 2 CIN will identify the party the name applies to by its unique CIN number.
A – 3 CIN will provide the Name Usage type (Legal, Alias, etc.).
A – 4 CIN will provide the full record of a new data.

Workflow:

Step	Transaction	Notes
010	getPartyByAdminSysKey	Retrieve the Person-/Orgname based on the unique key of the CIN system Inquiry Level = 0 AdminSystemtype = 1 (CIN System) AdminPartyId = <CIN Id provided>
20	addPersonName or addOrganizationName	Depending on the PartyType from previous request Using the PersonNameIdPK from the request in Step 010 where the Name matches the name provided or Using the OrganizationName IdPK from the request in Step 010 where the name matches the name provided. XxxxxNameIdPK = <from response in Step010>

Figure 10-9 Example text-based service design

This example represents simple flow logic and demonstrates a few things. First, you cannot clearly see which objects are handled by which standard transaction and therefore have no means of tracing your domain objects. You cannot guarantee a complete and consistent design and cannot support reporting to project management or perform an automated impact analysis in case of a change that comes your way.

In addition, you can already envisage more complex flow logic that might require more nested structures. If so, it is impossible to describe complex nested structures in a concise and complete manner where developers are able to implement the flow simply after reading the document. In most cases, such a document is not even clear to the designer who wrote the flow logic in the first place, so you cannot expect a developer to follow your thoughts.

Unlike spreadsheets, it is much harder to argue that the wrong tool is in use. In fact, most projects insist on some form of written and readable documentation. In this situation, this tool comes into the equation again. You should use an approach where all information is primarily defined and described in a single model-based tool and the final documentation is generated from that information.

Graphical design specification

Graphic development tools can provide added value to the service design. For example, they can support the documentation created with the Text Processing tool by enabling you to visualize complex flows. Starting with the same objectives related to the task of designing composite services, you define your composite service flow logic using diagrams. An example is depicted in Figure 10-10.

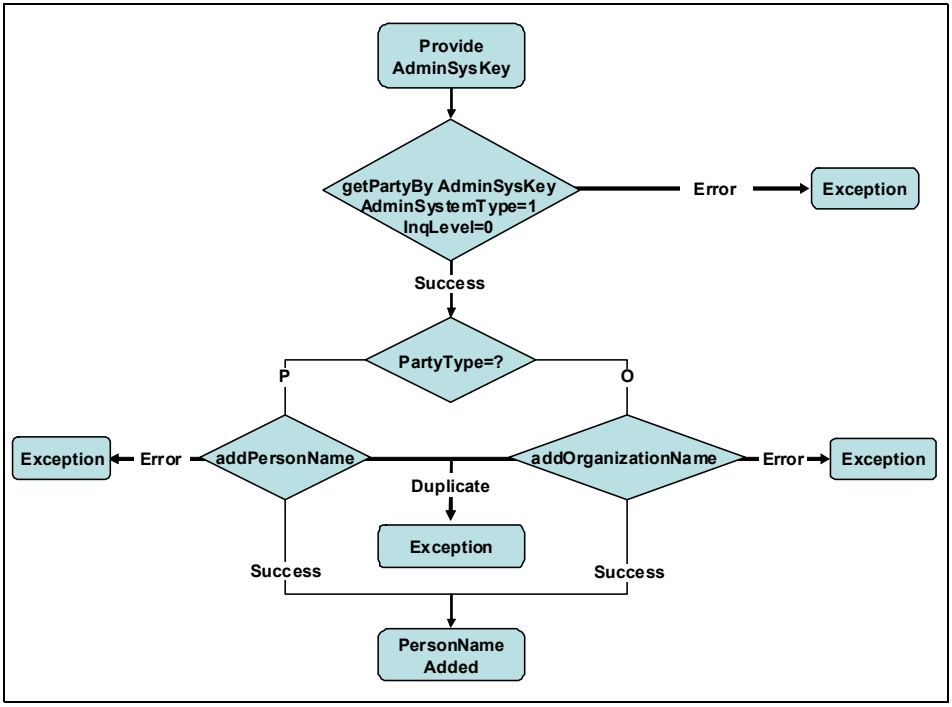

Figure 10-10 Example of a graphics-based service design

This type of representation of the same logic contained in the flow is much easier to capture and to interpret by any developer that works on the service implementation. Unfortunately, there is no easy way to build up these graphics as an asset, because we cannot store all the standard services and other necessary design elements and drag them into our custom diagrams wherever and whenever we like.

The downside of such an approach is more than the visual interpretation it provides. First, it still does not guarantee any consistency, because all objects are simply arbitrary components you can draw anywhere you want. Second, the diagrams cannot be traced, for the same reasons. Third, it is not possible to use such diagrams for automated code generation, which is the ultimate goal of following a modeling based approach. Therefore, these diagrams represent effort without gaining any major value from them, and it is for those reasons that they should not be used for that purpose.

10.4.2 Modeling tools

You can use modeling tools to achieve consistent, concise, and complete models. Models with these attributes address the most common tasks, such as support for planning through impact analysis based on traceability, and elaborate tasks, such as the ultimate goal of automated code generation. You can use the tool or tool set of your choice to get closer to bringing all models together to support your goals.

Modeling tools provide the following advantages:

- Support for traceability

  Traceability holds all models together. Models can be built to include traceability information, which in turn allows for impact analysis, reporting, and tracking.

- Tight integration

  The correct tools are so closely related with each other that all models are tightly integrated. It is important to be able to freely and easily navigate through all parts of the model to obtain the understanding that is required for a certain task. Navigation is required due to the separation of concerns. As a result, the models are spread out to many locations.

- Formalization

  Metamodels are the formalization that is required to implement transformations between the models. It is also what the design and implementation guidelines must adhere to, to smoothly generate the expected outcome.

Alternatively, modeling can also have the following disadvantages:

- Availability

  Modeling tools are generally not as widely available and access is often on demand or restricted to user groups only.

- Adoption

  Because of the complexity of the MDD methodology, the adoption rate for modeling tools is not as high as for office tools. Although modeling tools might appear more complex to the user, providing education early in the process in conjunction with modeling tools can be beneficial.

- Skills

  Skills with modeling tools are rare, probably because of a low adoption rate and restricted availability. Plan for sufficient ramp up time, and provide the education and training that is required.

- Flexibility

  The methods that we have described, in particular with automated code generation in mind, require strict formalization and adherence to well-defined guidelines. These factors take flexibility away from the user and, in turn, lowers the adoption rate.

Content note: The focus of this discussion on tools for data modeling, data mapping, and service design is the tools that IBM provides. The discussion also includes information about project management, because project management provides a good example of how a typically de-coupled activity can be integrated with the overall model.

The discussion also includes architecture, implementation, and testing because these modeling tools can handle these aspects as well, even when you consider that source code implementation, in particular, is based on source control capabilities.

The following IBM Rational Team Concert modeling products are available:

- IBM InfoSphere Data Architect
- IBM Rational Software Architect
- IBM Rational Team Concert
- IBM InfoSphere MDM Server Workbench

You can use these products to achieve your overall goals. You can include additional tools from the IBM Rational brand, but for reasons of maturity, knowledge, and lack of skills with these products, this book focuses on just these tools.

Another set of tools comes from the IBM InfoSphere brand and is represented by IBM InfoSphere Information Server. This product offers additional tools that can be useful in the larger context of your integration efforts. You can use this product to discover metadata about common data sources and store that information in its own repository. Based on that metadata, you can carry out analysis through profiling and derive rules to apply to standardization and cleansing processes. Then, you can use that same metadata to design the important ETL tasks, eliminating errors incurred when adding this information manually. A sophisticated business glossary and another data mapping tool round out this suite. For more information about the IBM InfoSphere suite of products, see the IBM products website at:

http://www.ibm.com/software/data/infosphere/

IBM InfoSphere Data Architect

IBM InfoSphere Data Architect is the IBM tool of choice for all data modeling and some data mapping-related aspects. Despite its name, which incorporates the IBM InfoSphere brand, it is based on the IBM Rational platform and is a product of the same origin. It, therefore, integrates well with Rational tools to form one large set of tools that support each other well.

Figure 10-11 shows the IBM InfoSphere Data Architect showing the physical data model of the system of the case study for this book.

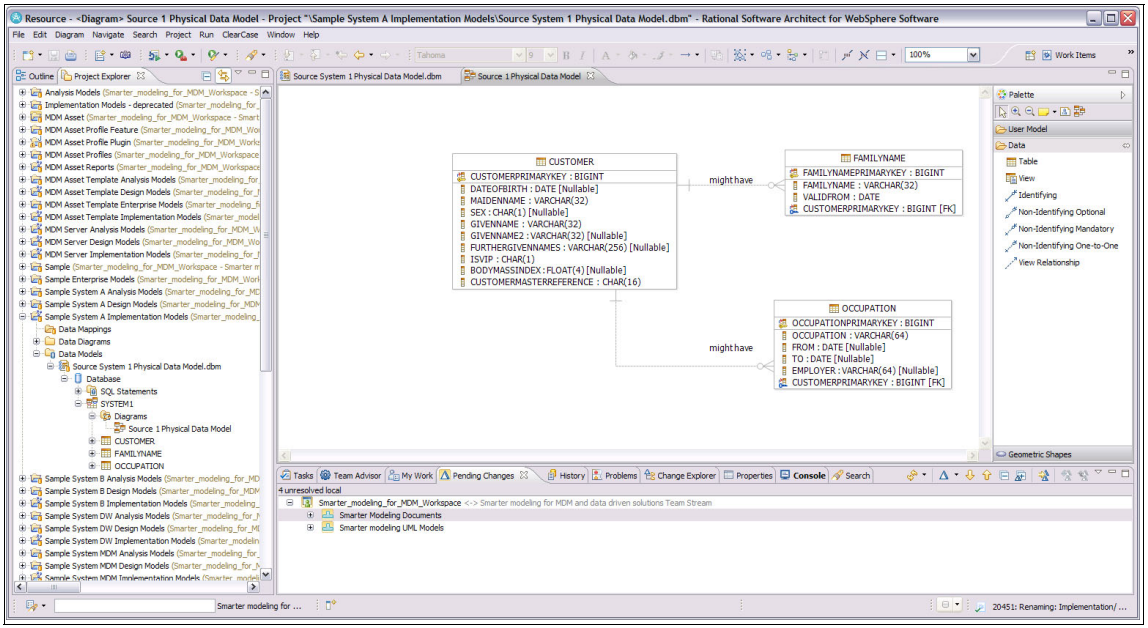

Figure 10-11 Data modeling based in IBM InfoSphere Data Architect

With respect to modeling functions, you can use this product to create the following types of models:

- Enterprise models
  - Glossary
  - Conceptual data model
  - Static mapping to logical and domain models against each system
- Analysis models
  - Glossary
- Design models
  - Glossary
  - Logical data model
  - Static data mapping on logical level
- Implementation models
  - Glossary
  - Physical data model
  - Static data mapping on physical level

The tool supports the creation of artifacts on all four modeling levels. Because the conceptual enterprise information model resembles a logical model, you can use the following types of artifacts:

- Glossary models
- Logical models
- Physical data models
- Data mapping

Based on physical data models, you can create the required Structured Query Language (SQL) and Data Definition Language (DDL) to create a database schema. This way, you can generate some of the implementation relevant artifacts automatically. You can use the glossary to force these models to adhere to your naming standards and to look up information.

Static mapping capabilities: This tool provides for only static mapping capabilities. Although you can use static mapping capabilities to identify entities and their attributes quickly, it does not suffice for more complex transformation dependencies. The missing dynamic mapping is provided through other models in IBM Rational Software Architect.

Link to other UML-based models: Because IBM InfoSphere Data Architect is not yet fully integrated into the IBM Rational platform, there is a missing link between the glossary in IBM InfoSphere Data Architect and the other UML-based models within IBM Rational Software Architect.

IBM Rational Software Architect

IBM Rational Software Architect is the core tool within your set of tools. You can use it to bind most of the models that you require in a methodology. Thus, you can use it across all of the modeling levels, from enterprise modeling to analysis modeling, for use cases and business processes, and business domain modeling. The domain model also reappears in an enriched form in the design modeling, together with the service design models. You can use IBM Rational Software Architect for implementation modeling and the implementation that creates the source code for the services that you require.

Figure 10-12 shows IBM Rational Software Architect with an activity diagram that describes the behavior of a consistency rule. Activity diagrams that describe service control flow look similar to this diagram according to our definitions.

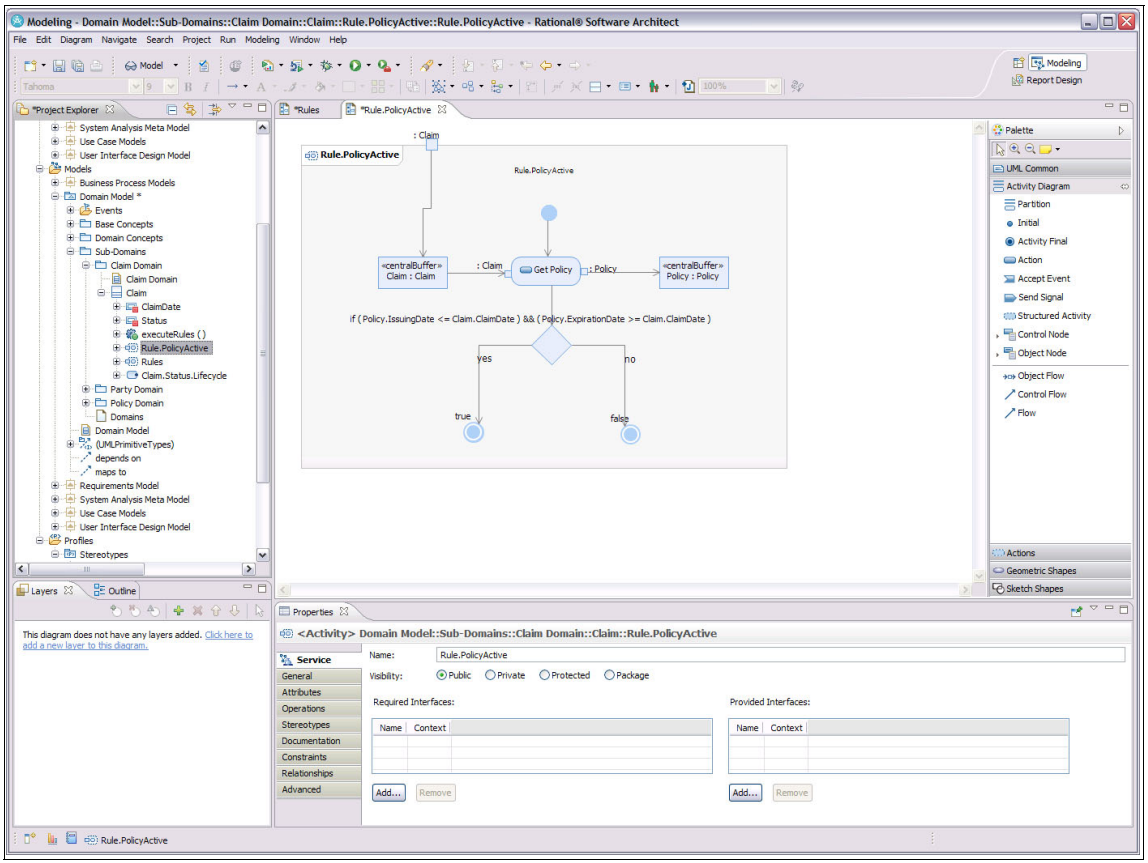

Figure 10-12 Activity modeling based in IBM Rational Software Architect

Product note: IBM Rational Software Architect was formerly divided into a pure modeling-related component (IBM Rational Software Modeler) and a component that is capable of working with source code (IBM Rational Application Developer). With IBM Rational Software Architect V8, these functions are combined within one product.

IBM Rational Application Developer is the platform of choice for any InfoSphere MDM Server implementation project, because InfoSphere MDM Server Workbench uses the capabilities that IBM Rational Application Developer provides. It combines all aspects necessary for the implementations that you typically deliver, which saves time, increases consistency, reduces errors, and enhances usability within projects. If you already have a well-integrated platform available with IBM Rational Software Architect, you must discern whether you need this powerful integrated tool set.

IBM Rational Team Concert

IBM Rational Team Concert is based upon the new Jazz platform, on which numerous products are based, such as Rational Requirement Composer and Rational Method Composer. Although IBM Rational Team Concert is based on a new and different platform, when compared with the Eclipse platform that IBM Rational Software Architect and IBM InfoSphere Data Architect on which are based, it integrates well with these tools. Integration is based on the Eclipse plug-in, which offers the functionality of IBM Rational Team Concert within IBM Rational Software Architect. As a result, you do not have to leave IBM InfoSphere Data Architect and can skip using the web-based user interface that IBM Rational Team Concert also provides. For that interface, there are additional planning, tracking, and project management capabilities at your disposal, without leaving the IBM Rational Software Architect environment.

Figure 10-13 shows Rational Software Architect with the Jazz Administration view open, from which work items can be queried and worked on without leaving the integrated development environment (IDE) in which the modeling and development takes place.

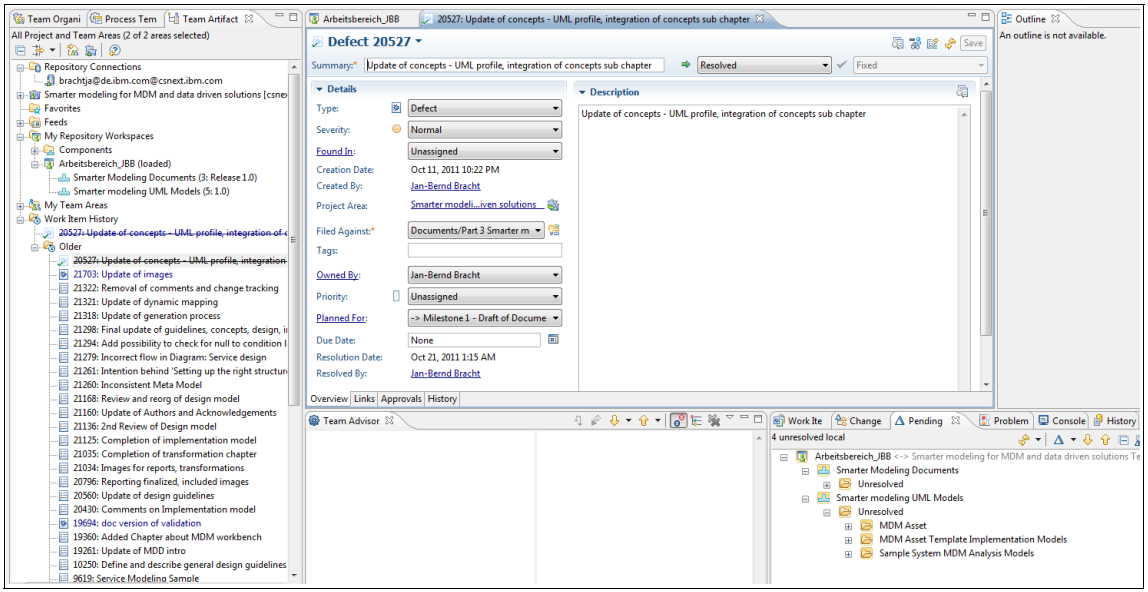

Figure 10-13 Work item management based in IBM Rational Team Concert

The following key features of IBM Rational Team Concert can help support your methodology:

- Project collaboration

  Project collaboration is an important aspect in projects. In fact, project collaboration is one of the most critical success factors. Communication must take place on all levels and concepts, and domain insights must be conveyed to these collaborators to whom they matter, that is, the developers who are implementing the system. Any additional communication helps achieve results better and faster.

- Source control

  Source control is required to keep control of the created artifacts, which is true for any documentation, models, and source code. Any source control system, such as IBM Rational ClearCase® or Subversion, can do the job. However, our example here limits the number of additional tools, even if integrated, to a minimum, so we use the source control system built into IBM Rational Team Concert.

- Planning and tracking

  Planning and tracking, and the reporting that accompanies it, is a core ingredient in any project. Too often, only a few people on a project hold this information and track it in internal documents, while sharing it with the team infrequently. So what could be better than embedding this information where it matters, with the modeling and development IDE, and allowing the developers and designers to set the status of tasks themselves? Project management can then extract the information and concentrate on creating the correct reports for the audience.

- Risk and change management

  Risk and change management are only two representative processes that play a role in a project. IBM Rational Team Concert capabilities have successfully been used to execute our defined processes. The key part of this feature is its ability to easily and dynamically specify and configure the workflows that are required to support these processes in the realm of an individual project.

InfoSphere MDM Server Workbench

InfoSphere MDM Server Workbench is an IBM Rational Software Architect or IBM Rational Application Developer plug-in. It provides a tool set to support development tasks that are related to the customization of InfoSphere MDM Server.

A wizard is available for setting up the development environment, and a user interface generator, a data maintenance user interface generator, and extension tools are included as well. Our example focuses on extension tools only. See *InfoSphere MDM Server Workbench User Guide* (included with the product) for information about other features.

Extension tools follow a model-based approach to define extensions to the data model and transactions provided by InfoSphere MDM Server. Based on this model, deployable artifacts are generated, which you can then customize to meet your own requirements, such as adding business logic or validations. You can apply changes to the model easily at any time. Regenerating from the models creates the necessary updates without overwriting any manually added custom code. For this reason, custom code must be tagged.

Figure 10-14 shows an example of a Hub Module model.

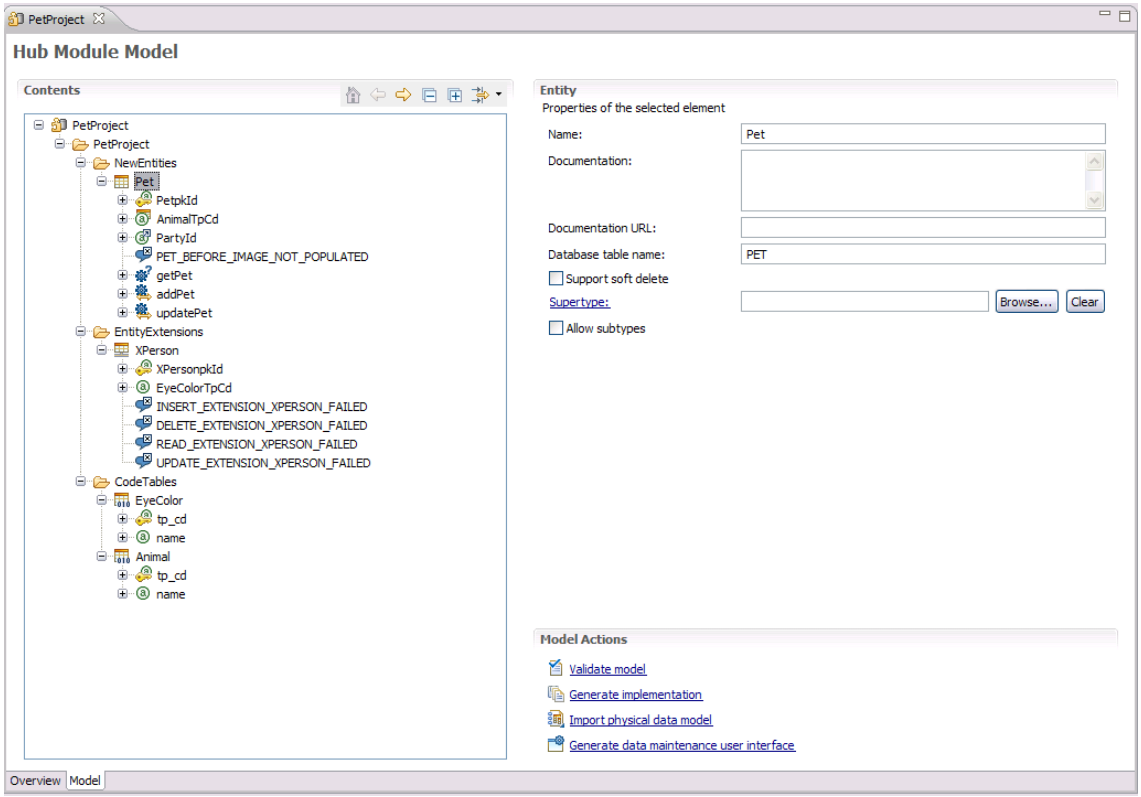

Figure 10-14 Hub Module model based in InfoSphere MDM Server Workbench

The basis for the extensions is the Hub Module model. This model defines the data and the associated transactions for a new domain. The modeler provides a tree view user interface to define the extensions, which results in an Ecore model that keeps this information. Ecore is the Eclipse implementation of Eclipse Meta Object Facility (MOF).

This Ecore model, in combination with the GenModel, is then transformed into artifacts such as Java code, SQL queries, or XML Schema Definition (XSD). It generates the following information:

- Data additions, which are new entities
- Entity subtypes in terms of inheritance of data additions
- Data extensions to enrich an existing entity by adding further attributes
- Code tables to add new code tables

- Behavior extensions for adding or changing the functionality of a transaction or the action within a transaction
- Query extension to modify how the data is accessed
- Custom inquiries for new queries
- Metadata specifications used in a manner similar to data extensions, but by describing the extension in the form of XSD
- Product type hierarchy models to create a product type hierarchy
- Composite services to define new services

Generating data additions, their subtypes, code tables, and data extensions normally does not require additional coding, as the generation contains the services to create, read, and update. Only the delete services must be manually added, because InfoSphere MDM Server does not provide deletion mechanisms for most of its entities, but relies on deactivation by updating the appropriate business effective dates. With respect to behavior extensions, query extensions, custom inquiries, and composite services, only the skeletons from the workbench generation process are included. These skeletons require manual coding to create the wanted behavior.

Business logic or workflow support: InfoSphere MDM Server Workbench does not support business logic or workflow. The modeling chapters in Part 2, “Smarter modeling” on page 129 provide insights and techniques for applying business logic or workflow.

Looking at the MDD approach described in Chapter 3, “Model-driven development” on page 75, there is a gap between the step from the platform-independent model to the platform-specific model, because that part of the model is to be done manually using the model editor.

10.4.3 Source control

This section includes a description of how the IBM Rational Team Concert source control feature works. It also describes how to set up the correct structures in conjunction with the modeling projects maintained in IBM Rational Software Architect.

Figure 10-15 shows the general functionality of the source control system.

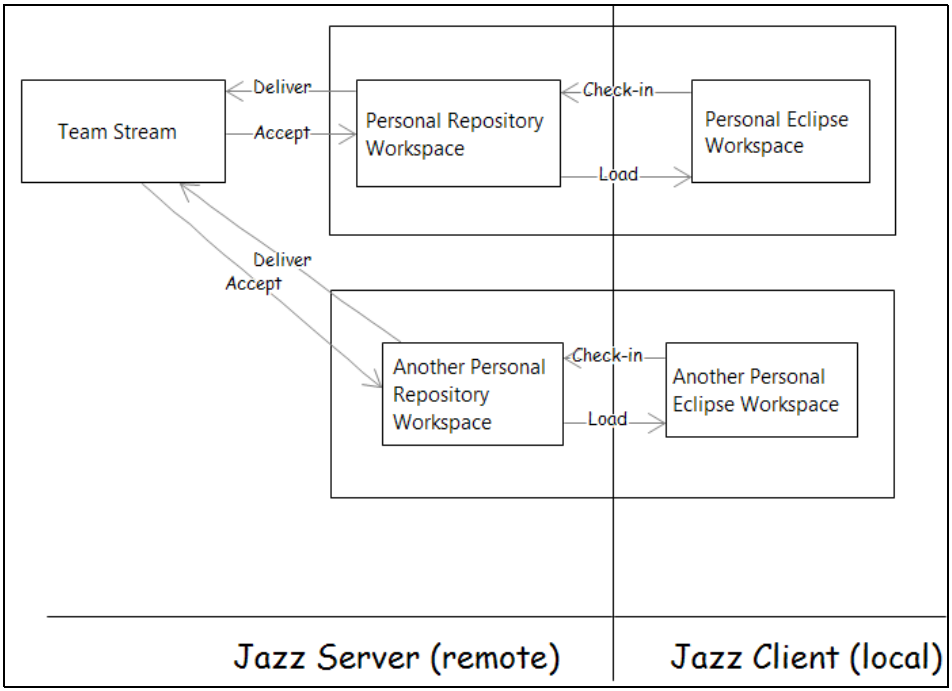

Figure 10-15 Source control based in IBM Rational Team Concert

The source control system includes the following main components and interactions:

- *Streams* for sharing and merging artifacts in a team
- *Repository workspaces*, which are personal spaces on the IBM Rational Team Concert repository server
- *Eclipse workspaces*, where you work on your local workstation
- *Check-In* to transfer changes to the Repository Workspace from the Eclipse Workspace
- *Deliver* to share Repository Workspace Changes to the Stream to share it with the team

- *Accept* to provide access to the components of the stream in the Repository Workspace and load it into the Repository Workspace
- *Load* to load the components from a repository workspace into a local workspace (the local workspace is created if it does not exist)

Now, for the real structures that you want to store, apply some structure and divide the stream into components. The examples in this book used the IBM Rational Team Concert source control mechanism divided into the following components:

- Documents that were created and collaborated on
- IBM Rational Software Architect based models that provided the verification for the written documentation and the models that contained the diagrams

With these activities for team collaboration and source control complete, the IBM Rational Team Concert component is complete. Next, populate the project in the workspace with data. This example uses UML models in addition to corresponding documents. So, you must create the folder structure and the draft documents for those folders. For example, the folder structure might be as follows:

- Part 1
  - Chapter 1
  - Chapter 2
  - ...
  - Chapter *n*
- Part 2
  - Chapter 1
  - Chapter 2
  - ...
  - Chapter *n*

The outline of the structure is such that chapters are defined for each part of the book, and a work item is defined for each chapter, where all chapters have a parent relationship to a part. To define this structure manually would be more time consuming.

Next, begin working with the artifacts. When any changes are made and saved, they display in the Jazz Source Control/Pending Changes view in IBM Rational Software Architect. From here, you can check in, check out, and deliver the changes.

Work items that refer to models are created accordingly. The example for this book consists of numerous projects, fragments, and models that are related to each other. One example for this relationship is the separation of analysis and design models into projects to achieve some separation of concerns among business and IT. However, both model types are related to the same system, and thus are directly related. Still, other aspects are spread across some models, such as the physical data models for multiple source systems. In this example, we decide to set up an identical model structure for each system separately to achieve a certain degree of separation of concerns.

How do we correlate these aspects to IBM Rational Team Concert? In this example, one story was prepared for each subsystem (two upstream and one downstream). Because each of these systems has the model structure that distinguishes between analysis and design models, corresponding work items are added that spread across these projects to the stories.

There is a large degree of freedom when setting up these structures. In essence, it is your own style and preferences that prevail.

10.4.4 Alternative modeling environment

The previous sections provide some insights into the suggested modeling environment, consisting of various tools from the IBM Rational and InfoSphere solutions. These tools are chosen because they provide the highly flexible and integrated working environment that meets your project needs. The strict enforcement of UML standards can be a challenge. However, it provides a good foundation for a model-driven approach, which requires some guidance and adherence to metamodels and underlying definitions.

The information presented in this book is independent from the tools, even though there are a number of tool-specific design artifacts contained in our models. If there is a different tool selection, you must adapt and slightly vary your guidelines before you can continue following the same approach. The core principles of the methodology remain the same, however, ensuring a consistent set of models that is aimed at the same goal of describing the master data management system.

Although there are various tools available, this section outlines some of the areas to consider when adopting this methodology to your tool choice.

Consider the following common factors:

- Enforcement of UML standards and formalities versus simplicity of modeling

  You must enforce standards and formalities to some degree, but also contend with the issues around the adoption rate. You can use a simplified modeling environment to express what you think can contribute to a higher adoption rate.

- Integration with version control

  Provides a ready-to-use versus third-party component, granularity of change tracking, or the ability to establish links between design changes and code changes.

- Integration with project management

  Integration with project management refers to the ability to measure workload and report efforts on design and code level, and to track and plan against project plans.

- Traceability

  Traceability includes requirements and changes from requirement through design to code.

- Quality of a one-size-fits-all tool

  Using just one tool might be convenient. However, one tool frequently does not have the functionality to meet all project needs.

- Tool usability

  Do not underestimate tools visualization capabilities. Throughout this book, we emphasize the importance of a clean and traceable model. However, this model must be built, which can be simple or complex, based on the editor used.

- Flexibility

  Flexibility refers to adding additional modeling aspects, customizing the UML, and adding small scripts that aid in the design. The Rational Tool Suite is built on Eclipse, which is known for its extensibility and interoperability, and has become a *de facto* standard. Other tools might rely on proprietary technology and languages, which is not necessarily bad, because these languages can be tailored to perfectly fit the underlying model. However, you might build a model silo here.

- Code generation, transformations, and extensive reporting

  These aspects are the core aspects of the approach that we describe in this book. Most enterprise modeling environments offer some of these capabilities with similar flexibility. They typically vary in standardization and available skills.

Abbreviations and acronyms

AOP	aspect-oriented programming
ASI	Adaptive Service Interface
AST	abstract syntax tree
BI	Business Intelligence
BIRT	Business Intelligence and Reporting Tools
CDI	Customer Data Integration
CIM	computation independent model
CMS	content management systems
CRM	customer relationship management
CSS	cascading style sheets
DBMS	database management system
DDD	domain-driven design
DDL	Data Definition Language
DMS	document management systems
EA	Enterprise Architect
EAI	Enterprise Application Integration
EMF	Eclipse Modeling Framework
EObj	entity object
ER	entity relationship
ERP	enterprise resource planning
ESB	enterprise service bus
ETL	extract, transform, and load
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
IBM	International Business Machines Corporation
IDE	integrated development environment
IIOP	Internet Inter-ORB Protocol
ITSO	International Technical Support Organization
JET	Java Emitter Template
JSP	JavaServer Pages
KYC	know your customer
LDM	Logical Data Model
LOB	lines of business
M1	model layer
M2	metamodel layer
M2M	model-to-model
M2T	model-to-text
M3	meta-metamodel layer
MDA	MDD and model-driven architecture
MDD	model-driven development
MDSD	model-driven software development
MOF	Meta Object Facility
NFR	non-functional requirements
OCL	Object Constraint Language
OMG	Object Management Group
PDF	Portable Document Format
PEP	politically exposed person
PIM	platform-independent model
PSM	platform-specific model
REST	Representational State Transfer
RMI	remote messaging interface
RUP	Rational Unified Process
SCM	supply chain management

SME	subject matter expert
SOA	service-oriented architecture
SQL	Structured Query Language
T2M	text-to-model
TOGAF	The Open Group Architecture Framework
UI	user interface
UML	Unified Modeling Language
WBS	work breakdown structure
WSDL	Web Services Description Language
XMI	XML Metadata Interchange
XPATH	XML Path
XSD	XML Schema Definition

Related publications

The publications listed in this section are considered suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks publications

The following IBM Redbooks publications provide additional information about the topic in this document. Some publications referenced in this list might be available in softcopy only.

- *IBM WebSphere Information Analyzer and Data Quality Assessment*, SG24-7508
- *Patterns: Model-Driven Development Using IBM Rational Software Architect*, SG24-7105

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Other publications

The following publications are also relevant as further information sources:

- Berson, et al., *Master Data Management and Customer Data Integration for a Global Enterprise*, Mcgraw-Hill Osborne Media, 2007, ISBN 0072263490
- Dreibelbis, et al., *Enterprise Master Data Management: An SOA Approach to Managing Core Information*, IBM Press, 2008, ISBN 0132366258
- Evans, et al., *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003, ISBN 0321125215
- Fowler, *Domain Specific Languages*, Addison-Wesley, 2010, ISBN 0321712943
- Fowler, et al., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002, ISBN 0321127420

- Godinez, et al., *The Art of Enterprise Information Architecture: A Systems-Based Approach for Unlocking Business Insight*, Pearson Education, 2010, ISBN 0137035713
- Peppers, et al., *Customer Data Integration: Reaching a Single Version of the Truth*, Wiley, John & Sons, 2006, ISBN 0471916978

Online resources

These websites are also relevant as further information sources:

- Data Mapping in UML:

  http://community.sparxsystems.com/whitepapers/data-modeling/data-mapping-uml

- Design SOA services with Rational Software Architect, Part 1: Get started with requirements, process, and modeling:

  http://www.ibm.com/developerworks/rational/tutorials/r-soaservices1/

- Design SOA services with Rational Software Architect, Part 2: Design services with the UML Profile for Software Services:

  http://www.ibm.com/developerworks/rational/tutorials/r-soaservices2/

- Design SOA services with Rational Software Architect, Part 3: Use assets and patterns in your design:

  http://www.ibm.com/developerworks/rational/tutorials/r-soaservices3/

- Design SOA services with Rational Software Architect, Part 4: Generate and test web services from UML models:

  https://www.ibm.com/developerworks/rational/tutorials/rt-soaservices4/

- The information perspective of SOA design, Part 5: The value and use of Rational Data Architect in SOA:

  http://www.ibm.com/developerworks/data/library/techarticle/dm-0803sauter2/index.html

- Enterprise Data Modeling: 7 Mistakes You Can't Afford to Make:

  http://www.information-management.com/newsletters/enterprise_architecture_data_model_ERP_BI-10020246-1.html?pg=2

- Getting from use cases to code, Part 1: Use-Case Analysis:

  http://www.ibm.com/developerworks/rational/library/5383.html

- Getting from use cases to code: Part 2: Use Case Design:

  http://www.ibm.com/developerworks/rational/library/content/RationalEdge/aug04/5670.html

- IBM InfoSphere MDM Server v9.0.2 Information Center:

  http://publib.boulder.ibm.com/infocenter/mdm/v9r0m2/index.jsp

- Reverse engineering UML class and sequence diagrams from Java code with IBM Rational Software Architect:

  http://www.ibm.com/developerworks/rational/library/08/0610_xu-wood/index.html

- Running a successful project with the InfoSphere Master Data Management Workbench:

  http://www.ibm.com/developerworks/data/library/techarticle/dm-1008mdmworkbenchproject/index.html

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Smarter Modeling of IBM InfoSphere Master Data Management Solutions

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Smarter Modeling of IBM InfoSphere Master Data Management Solutions

Overcome the challenges in master data management integration

Understand the importance of a methodology

Gain insights into consistent and comprehensive modelling

This IBM Redbooks publication presents a development approach for master data management projects, and in particular, those projects based on InfoSphere MDM Server.

The target audience for this book includes Enterprise Architects, Information, Integration and Solution Architects and Designers, Developers, and Product Managers.

Master data management combines a set of processes and tools that defines and manages the non-transactional data entities of an organization. Master data management can provide processes for collecting, consolidating, persisting, and distributing this data throughout an organization.

IBM InfoSphere Master Data Management Server creates trusted views of master data that can improve applications and business processes. You can use it to gain control over business information by managing and maintaining a complete and accurate view of master data. You also can use InfoSphere MDM Server to extract maximum value from master data by centralizing multiple data domains. InfoSphere MDM Server provides a comprehensive set of prebuilt business services that support a full range of master data management functionality.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7956-00 ISBN 0738436801