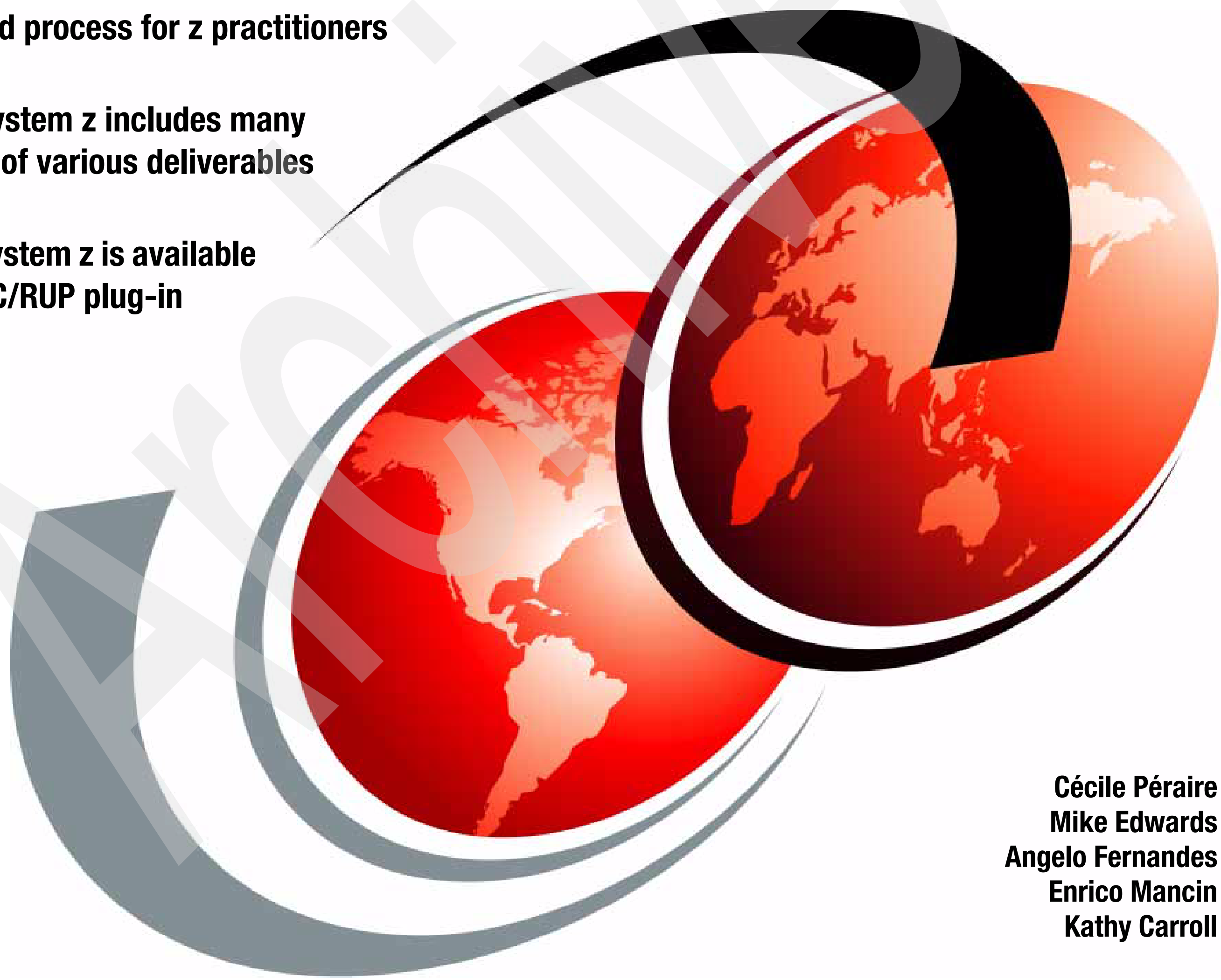
Rational software

IBM

The IBM Rational Unified Process for System z

- RUP for System z includes a succinct end-to-end process for z practitioners
- RUP for System z includes many examples of various deliverables
- RUP for System z is available as an RMC/RUP plug-in



Cécile Péraire
Mike Edwards
Angelo Fernandes
Enrico Mancin
Kathy Carroll

Redbooks

ibm.com/redbooks

International Technical Support Organization

The IBM Rational Unified Process for System z

July 2007

SG24-7362-00

Note: Before using this information and the product it supports, read the information in "Notices" on page vii.

First Edition (July 2007)

This edition applies to the IBM Rational Method Composer Version 7.1

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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites 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

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Redbooks (logo) ®
developerWorks®
iSeries®
z/OS®
zSeries®
CICS®
Electronic Service Agent™
IBM®
IMS™
Rational Summit®
Rational Unified Process®
Rational®
Redbooks®
RequisitePro®
RUP®
SoDA®
Summit Ascendant™
Summit®
System i™
System z™
WebSphere®

The following terms are trademarks of other companies:

Java, JavaScript, JavaServer, JSP, J2EE, RSM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Pentium, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication describes the new Rational® Unified Process® (RUP®) for System z™ method, which has been especially created for use by organizations that are involved in developing application software in the System z environment.

Developing application software in the System z environment has been going on for many decades and generally during this time, traditional development lifecycle methodologies have been applied to the development process. With the current environment of businesses needing to be more agile, on demand, and flexible to user needs, pressure is on IT organizations to respond in as agile and flexible a manner as possible in order to satisfy user needs with precision and quality.

RUP is based on proven development principles and contains best practices for developing software. This specific adaptation of a modern best-of-breed methodology, that is, RUP for System z, provides you with a development process that has already yielded much valued benefits to software development practitioners in other platform environments.

This IBM Redbooks publication demonstrates the use of the RUP for System z method by using a case study of an application development example. It provides you with actual example work products produced during the various lifecycle iterations and phases, so that you are able to more easily understand the iterative and incremental nature of application development and its associated benefits.

The new RUP for System z is also available as a Web site for easy reference, through a Rational Method composer (RMC) plug-in. This IBM Redbooks publication shows you how to download and install the new RUP for System z plug-in. Furthermore, it helps you configure it if necessary, to suit your own application development environment in order to enable your team to derive the utmost benefit and add value to your development activities.

This IBM Redbooks publication is intended for the whole of the System z application development community from beginners to advanced practitioners, for roles ranging from project managers, architects and designers, to programmers and testers alike, because it covers the full end-to-end development lifecycle for the System z environment. In addition, System z Development Managers and Method Designers in particular might find this book useful as a ready reference guide.

For convenience and to suit different levels of user expertise, this book is broken down into the following parts:

Part 1. Introduction to the IBM Rational Unified Process for System z

Part 2. The IBM Rational Unified Process for System z for Beginners

Part 3. The IBM Rational Unified Process for System z for Advanced Practitioners

Part 4. The IBM Rational Unified Process for System z for Method Designers and Project Managers

Part 5. Appendixes

The team that wrote this IBM Redbooks publication

This IBM Redbooks publication was produced by a team of specialists from around the world working at the International Technical Support Organization Raleigh Center, in San Jose, California. Figure 0-1 and Figure 0-2 show the IBM Rational Unified Process for System z team.

Figure 0-1 From left, Enrico Mancin, Cécile Péraire, Angelo Fernandes, and Mike Edwards

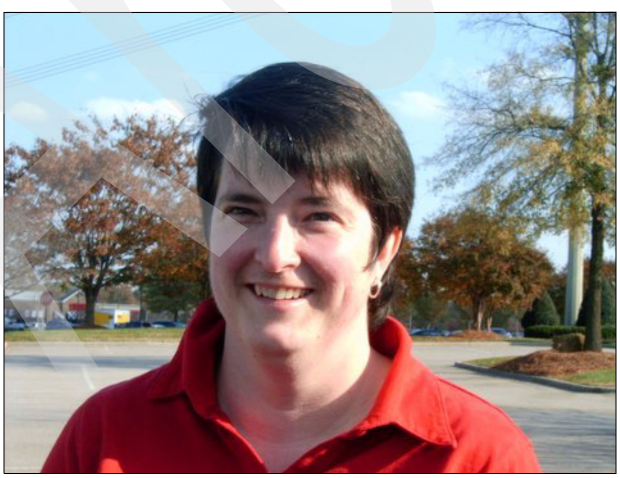


Figure 0-2 Kathy Carroll

Enrico Mancin is a Master Certified IT Architect on the Technical Sales Rational Team for IBM SWG. He joined IBM in 2003 as a result of the Rational Software company acquisition. He is the author of some articles on RUP, SOA, and Open Source. Before joining IBM, Enrico worked as a senior software specialist with Rational, assisting clients adopting Rational methods and tools. He also covered the role of Chief Architect in important Italian companies and has about 20 years of experience in the field of software engineering.

Cécile Péraire is a method architect and author with IBM US, contributing to the definition of IBM software development methods, including the Rational Unified Process (RUP) and some of its most significant extensions. Before joining the IBM Rational Unified Process team, Cécile worked as a senior consultant with Rational, assisting clients adopting Rational methods and tools. She has about 16 years of experience in the field of software engineering. Cécile holds a Ph.D. in software testing from the Swiss Federal Institute of Technology in Lausanne (EPFL).

Angelo Fernandes is a Quality and Support Lead for System z development at the Australian Development Laboratory in IBM Australia. He has more than 28 years of System z development and support experience, having worked in both client and IBM software development environments. Over the years, he has fulfilled roles spanning the complete end-to-end lifecycle of software development and has project managed a number of IBM System z software product development and support projects.

Mike Edwards is a Software Engineer on the development team for Electronic Service Agent™ for zSeries® in IBM Canada. He holds a Master's Degree in Engineering from the University of Toronto and has worked in the IT Industry for more than 20 years. Before joining IBM, Mike worked in a number of industries as an independent consultant.

Kathy Carroll is a Software Engineer in Research Triangle Park, North Carolina. She has more than 15 years experience with application development software from development to consulting. Her current role is the lead developer for the VAGen to EGL Migration team. She holds a Computer Science degree from Wake Forest University.

Thanks to the following people for their contributions to this project:

Per Kroll, STSM, Manager RUP/RMC, IBM Software Group, Rational, Raleigh.

Bruce MacIsaac, Manager RUP/OpenUP Content, IBM Software Group, Rational, San Jose.

Bob Cancilla, Product Market Manager - Rational Tools for System i™ and System z, IBM Software Group, Rational, Costa Mesa.

Alessandro di Bari, IBM Rational, Italy.

Gregory Hodgkinson, IBM Business Partner 7irene, UK.

Joe Vincens, IBM Rational, Raleigh

Dan Bruce, IBM Rational, Raleigh

Bob Haimowitz, IBM International Technical Support Organization, Raleigh

Jonathan Sayles, IBM Rational, Raleigh

Mark Evans, IBM Rational, Raleigh

Sanjay Chandru, IBM Rational, Raleigh

Alex Lui, IBM Rational, Raleigh

James Conover, IBM SWG-AIM, Buffalo

William Deason, IBM SWG-AIM, San Jose

Chris Rayns, IBM International Technical Support Organization, Poughkeepsie

Richard M. Conway, IBM International Technical Support Organization, Poughkeepsie

and

Joe DeCarlo, Manager for Special Projects, International Technical Support Organization, Raleigh Center via San Jose, California.

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbooks Publication dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll have the opportunity to team with IBM technical professionals, IBM Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

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

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

  ibm.com/redbooks

- Send your comments in an e-mail 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

Part 1

Introduction to the IBM Rational Unified Process for System z

This part introduces the IBM Rational Unified Process for System z (RUP for System z).

1

Introduction

In this chapter, we discuss the purpose, the target audience, the rationale for the book, and the scope of the method. We also provide an overview of the contents, including case study examples we used during the researching and writing of this book.

1.1 Purpose

The purpose of this book is to introduce the System z software development community to the newly developed Rational Unified Process (RUP) for the System z environment.

Its aim is to describe the key fundamental principles and best practices of RUP and to demonstrate the value of applying these very same principles and practices to development activities within the System z environment by using Rational Unified Process for System z.

By using a real System z CICS® TS application written in COBOL as a case study, the aim is to demonstrate the key elements and steps involved in adopting the RUP for System z process to application development in the System z environment.

The book also describes how to obtain and install the new RUP for System z plug-in created for the Rational Method Composer (RMC), so that you can publish the method as a Web site and further customize the method, if necessary, to suit your own organization's needs and preferences.

1.2 Audience

This IBM Redbooks publication is intended for the whole of the System z application development community from project managers, architects and designers, to programmers and testers alike, because it covers the full end-to-end development lifecycle for the System z environment.

In addition, System z Development Managers and Method Designers in particular, that is, people who are responsible for implementing methods, standards, and procedures within their teams or organizations, might find this book more useful as a reference guide to assist them in adapting RUP for System z to their own development environments. The intent is for you to be able to implement RUP for System z in a manner that is most appropriate to your own specific development environment, so that you might reap the vast benefits that are associated with it.

1.3 Rationale

Although in the recent past since the 1990s, there have been several changes in methods and processes used in the System z application development environment, they have all largely been related to the traditional waterfall development lifecycle model. So far, it has been a common belief that modern development methodologies, such as RUP, are applicable only to the Object-Oriented programming world.

However, given the current frequently changing, on demand business climate in which we live, businesses are required to be nimble, flexible, and responsive to ever changing business needs. Therefore, we thought it timely to investigate and document how a modern, popular, and integrated development methodology, such as RUP, can be applied to application development in the System z environment in order to add value to the applications and products developed for System z.

The result is this IBM Redbooks publication. This book leverages the best elements of RUP with a specific focus on the System z development environment.

1.4 Scope

The RUP for System z method addresses *green field development* and *system evolution* with *architectural changes* (including turning an existing capability into a Web service, for instance) or with *significant impact on existing user business processes*.

Pure maintenance is out of our scope. For more information about maintenance, refer to 4.6, "Note on maintenance projects" on page 49 for a brief discussion about maintenance projects and refer to the RUP for Maintenance Projects plug-in at:

http://www.ibm.com/developerworks/rational/downloads/06/plugins/rmc_prj_mnt/

The RUP for Maintenance Projects plug-in provides a delivery process, tasks, and guidance for avoiding pitfalls during a maintenance cycle and successfully delivering a product with higher quality than the previous release.

1.5 Overview

The main topics of this IBM Redbooks publication are:

- Introduction to RUP and its extension to Service-Oriented Architecture (SOA)
- Why RUP for System z
- RUP for System z roadmap
- RUP for System z process essentials
- RUP for System z end-to-end lifecycle
- RUP for System z content elements
- Catalog Manager case study
- Enterprise Generation Language (EGL)
- RUP for System z Work Breakdown Structure (WBS)
- How to customize RUP for System z

The main topics are followed by an appendix, which contains work products of the Catalog Manager application development case study that were generated during various iterations of the RUP development phases. There is an appendix that provides a terminology mapping between RUP and System z terms and another that provides information about where to download the RUP for System z Rational Method Composer (RMC) plug-in.

Introduction to RUP and its extension to Service-Oriented Architecture

This chapter introduces you to the key underlying principles of RUP and its framework of reusable method content and process building blocks. It provides an overview of the RUP lifecycle, describing its various phases, iterations, and the purpose and goal behind each of the phases. This chapter also describes a roadmap through the RUP when developing service-oriented solutions.

Why RUP for System z

The System z environment has been around for a long time. Over the years, its developers have been pioneers in formulating and using various application development methodologies. So why RUP for System z? This chapter provides you with compelling reasons for why we undertook this project of producing a RUP for System z and the value proposition that comes with it. The RUP key principles are commercially proven approaches to software development, obtained from industry experts and from thousands of clients and development projects. So why not expose RUP to benefit the System z environment too?

RUP for System z roadmap

This chapter provides a roadmap, walking through each phase (inception, elaboration, construction, and transition) of a typical System z development project.

RUP for System z process essentials

This chapter provides the process essentials: A brief definition of each project phase (inception, elaboration, construction, and transition) in terms of the main goals, activities, and milestones. For each activity, the chapter lists the corresponding key roles, tasks, output work products, and available examples from the Catalog Manager case study. The corresponding section of the RUP for System z Web site provides advanced System z practitioners with all the links necessary to perform specific activities or tasks.

RUP for System z end-to-end lifecycle

This chapter describes the RUP for System z process from an end-to-end lifecycle perspective. The end-to-end lifecycle can be used as a template for planning and running a project. It provides a complete model with predefined phases, iterations, activities, and tasks.

RUP for System z content elements

The RUP for System z includes a large number of content elements (roles, tasks, and artifacts). Most of these elements come from the Rational Unified Process (RUP) and its Service-Oriented Architecture (SOA) extension. However, some content elements have been added to the RUP for System z because they are specific to the System z environment. This chapter presents these new content elements.

Catalog Manager case study

It is common knowledge that the use of a new technology is best shown by real working examples. In this chapter, you will find a real-life case study as an example of how we put RUP for System z into practice. The case study walks you through our development of a COBOL CICS application, showing you actual work products and deliverables at various levels of incremental progress and achievement, as derived during the different phases and iterations of the method. Reading this chapter will allow you to visualize how RUP for System z can be put into practice during application development projects in your own organization.

Enterprise Generation Language (EGL)

This chapter introduces the Enterprise Generation Language (EGL) and the value that this programming language can bring to you and your organization. EGL is a high-level procedural language that developers unfamiliar to Java™ can use to quickly develop Web, TUI, and batch applications with data-driven business logic. EGL can also be used to generate COBOL for your System z. EGL was designed for developers who need to focus on the business logic of an application rather than the technology or platform on which the application needs to run. The result is higher productivity. We used EGL in this IBM Redbooks publication project to develop a Web client application that consumes COBOL CICS Web Services.

RUP for System z Work Breakdown Structure (WBS)

The RUP for System z includes a Work Breakdown Structure that covers the whole development lifecycle from beginning to end. This Work Breakdown Structure can be used as a template for planning and running a project. This chapter presents the Work Breakdown Structure for each project phase (inception, elaboration, construction, and transition).

How to customize RUP for System z

Finally, for any method to be practical and applicable to your own organization's environment, it needs to be flexible and customizable. This chapter shows you how to customize RUP for System z to suit your own organization's needs and preferences in order to allow you to implement the method or parts of the method in a manner that helps you derive the most benefit out of it.

Part 2

The IBM Rational Unified Process for System z for Beginners

This part includes learning material related to the IBM Rational Unified Process for System z (RUP for System z). This part is targeted toward beginners.

2

Introduction to the IBM Rational Unified Process and its extension to Service-Oriented Architecture

The IBM Rational Unified Process for System z (RUP for System z) is based on the IBM Rational Unified Process (RUP) and its Service-Oriented Architecture (SOA) extension (RUP for SOA). This chapter introduces RUP and RUP for SOA.

Most of the content in this chapter comes directly from RUP and RUP for SOA where you can obtain additional introductory information.

2.1 Overview

The IBM Rational Unified Process (RUP) is a software engineering process framework. It provides best practices and guidance for successful software development and a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its users within a predictable schedule and budget. Figure 2-1 illustrates the overall architecture of RUP.

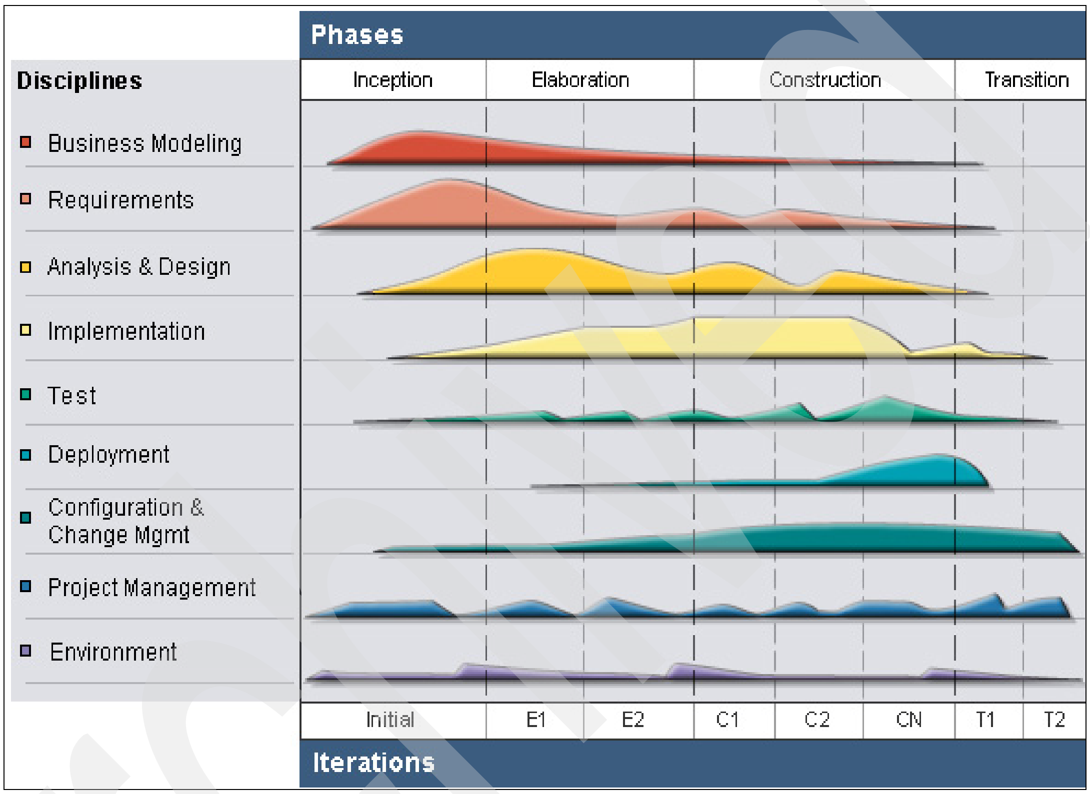


Figure 2-1 Overall Architecture of RUP

As shown in Figure 2-1, RUP has two dimensions:

- The horizontal axis represents time and shows the lifecycle aspects of the process as it unfolds. The lifecycle is divided into four phases: inception, elaboration, construction, and transition. Each phase is divided into one or more iterations. For instance, in Figure 2-1, inception has one iteration, elaboration has two iterations, construction has *n* iterations, and transition has two iterations. The right number of iterations per phase varies from project to project.
- The vertical axis represents disciplines, such as requirements, analysis, and design, or implementation, which logically group activities by nature.

The graph shows that most iterations cover all disciplines; however, the emphasis varies over time. For example, in early iterations you spend more time on requirements; in later iterations, you spend more time on implementation.

This introduction to RUP and its SOA extension includes the following content:

- **Introduction to RUP**
  This section answers fundamental questions about the nature and purpose of RUP.

- **Key principles for successful software development**
  This section presents key principles characterizing the industry's best practices in the creation, deployment, and evolution of software-intensive systems. RUP is based on these principles.
- **RUP lifecycle**
  This section describes the phases and milestones of a typical RUP project lifecycle.
- **Developing service-oriented solutions**
  This section describes a roadmap through RUP when developing service-oriented solutions.

2.2 Introduction to RUP

This section introduces the IBM Rational Unified Process (RUP) by describing the heart of RUP and the IBM Rational Method Composer (RMC) platform.

2.2.1 The heart of RUP

At its heart, the IBM Rational Unified Process (RUP) is about successful software development. There are three central elements that define RUP:

- **An underlying set of philosophies and principles for successful software development**
  These philosophies and principles are the foundation on which RUP has been developed. See 2.3, "Key principles for successful software development" on page 14 for more on the topic.
- **A framework of reusable method content and process building blocks**
  Defined and improved on an ongoing basis by Rational Software, the RUP family of method plug-ins defines a method framework from which you create your own method configurations and tailored processes.
- **The underlying method and process definition language**
  Underlying it all is a unified method architecture meta-model. This model provides a language for describing method content and processes. This new language is a unification of different method and process engineering languages, such as the SPEM extension to the Unified Modeling Language (UML) for software process engineering, the languages used for RUP v2003, Unified Process, IBM Global Services Method, as well as IBM Rational Summit® Ascendant.

One of the core practices behind RUP is iterative and incremental development. This practice is also good to keep in mind as you start with RUP: Do not try to "do" all of RUP at once. Adopt an approach to implementing, learning, and using RUP that is itself iterative and incremental. Start by assessing your existing process and selecting one or two key areas that you want to improve. Begin using RUP to improve these areas first and then, in later iterations or development cycles, make incremental improvements in other areas.

2.2.2 The IBM Rational Method Composer (RMC) platform

Over many years of development effort, RUP has evolved into a rich process engineering platform called IBM Rational Method Composer (RMC). RMC enables teams to define, configure, tailor, and practice a consistent process.

The key elements of the platform are:

- **Method delivery tool**
  RUP is delivered to practitioners as an interactive Web site using industry-standard browser technology. A RUP Web site is a Rational Method Composer-published process presentation configured for your project and tailored to your specific needs. The Web site is created using dynamically generated HTML pages, which RMC enables you to publish in the form of multiple RUP Web sites, each representing a configured and tailored process definition.
- **Method configuration tool**
  IBM Rational Method Composer (RMC) supports the fine-grained publish-time configuration of method content and processes to meet the varied needs of different projects and users. RMC allows the optional inclusion of method and process extensions using Method Composer's plug-in technology. It also allows you to configure variants on processes, which are published differently depending on user-specific selections.
- **Method authoring tool**
  The IBM Rational Method Composer (RMC) tool is specifically designed for method content management and process authoring with functions, such as form-based authoring, breakdown structure-based authoring, content browsing, content search, and import and export of method content. RMC also provides mechanisms for rapid process assembly using process patterns and reusable method elements. It supports the creation of method plug-ins that provide powerful ways of extending and modifying existing content, simplifying method content, process management, and maintenance.
- **A marketplace for process extensions**
  The RMC/RUP section of the developerWorks® Rational Web site provides a place for process engineers in the software development community to share their method extensions as consumable plug-ins and provides a rich source of method extensions for the project manager. The RMC/RUP section of the developerWorks Rational Web site can be found at:

  http://www.ibm.com/developerworks/rational/products/rup/

More information about RMC can be found at:

http://www.ibm.com/software/awdtools/rmc/

2.3 Key principles for successful software development

This section presents key principles characterizing the industry's best practices in the creation, deployment, and evolution of software-intensive systems. RUP is based on these principles, and they are the following:

- Adapt the process.
- Balance competing stakeholder priorities.
- Collaborate across teams.
- Demonstrate value iteratively.
- Elevate the level of abstraction.
- Focus continuously on quality.

Each principle is presented through:

- *The benefits* derived from applying the principle.
- *The pattern* of behavior that best embodies the principle.

- The most recognizable *"anti-patterns"* or behaviors contrary to the principle that can harm software development projects.

2.3.1 Adapt the process

This principle states that it is critical to rightsize the development process to the needs of the project. More is not better, less is not better: Instead, the amount of ceremony, precision, and control present in a project must be tailored according to a variety of factors, including the size and distribution of teams, the amount of externally imposed constraints, and the phase the project is in.

Benefits:

- Lifecycle efficiency
- Increased project agility
- Realistic plans and estimates

Pattern:

- Rightsize the process to project needs, including:
  - The size and distribution of the project team
  - The complexity of the application
  - The need for compliance
- Adapt process ceremony to the lifecycle phase (allow formality to evolve from light to heavy as uncertainties are resolved).
- Improve the process continuously.
- Balance plans and estimates with the level of uncertainty.

Anti-patterns:

- Always see more process and more detailed up front planning as better:
  - Force early estimates and stick to those estimates.
  - Develop precise plans, and manage the project by tracking against a static plan.

2.3.2 Balance competing stakeholder priorities

This principle articulates the importance of balancing often conflicting business and stakeholder needs, as well as balancing custom development against asset reuse in the satisfaction of these needs.

Benefits:

- Align applications with business and user needs.
- Reduce custom development.
- Optimize business value.

Pattern:

- Define, understand, and prioritize business and user needs.
- Prioritize projects and requirements and couple the needs with the software capabilities.
- Understand what assets we can leverage.
- Balance asset reuse with user needs.

Anti-patterns:

- Thoroughly document precise requirements at the outset of the project and force stakeholder acceptance of requirements:
  - Negotiate any changes to the requirements where each change can increase the cost or duration of the project.
  - Lock down requirements up front, thereby reducing the ability to leverage existing assets.
  - Primarily perform custom development.
- Architect a system only to meet the needs of the most vocal stakeholders.

2.3.3 Collaborate across teams

This principle stresses the importance of fostering optimal project-wide communication. This is achieved through proper team organization and the setting up of effective collaborative environments.

Benefits:

- Team productivity
- Better coupling between business needs and the development and operations of software systems

Pattern:

- Motivate people to perform at their best.
- Create self-managed teams.
- Encourage cross-functional collaboration (for example, analysts, developers, and testers).
- Provide effective collaborative environments.
- Manage evolving artifacts and tasks to enhance collaboration, progress, and quality insight with integrated environments.
- Integrate business, software, and operation teams.

Anti-patterns:

- To nurture heroic developers willing to work extremely long hours, including weekends
- Have highly specialized people equipped with powerful tools for doing their jobs, with limited collaboration between different team members, and limited integration between different tools. The assumption is that if just everybody does his or her job, the end result will be good.

2.3.4 Demonstrate value iteratively

This principle explains why software development greatly benefits from being iterative. An iterative process makes it possible to easily accommodate change, to obtain feedback and factor it into the project, to reduce risk early, and to adjust the process dynamically.

Benefits:

- Early risk reduction

- Higher predictability throughout the project
- Trust among stakeholders

Pattern:

- Enable feedback by delivering incremental user value in each iteration.
- Adapt your plans using an iterative process.
- Embrace and manage change.
- Attack major technical, business, and programmatic risks early.

Anti-patterns:

- Plan the whole lifecycle in detail and track variances against plan (can actually contribute to project failure).
- Assess status in the first two thirds of the project by relying on reviews of specifications, rather than assessing status of test results and demonstrations of working software.

2.3.5 Elevate level of abstraction

Complexity is a central issue in software development. Elevating the level of abstraction helps reduce complexity as well the amount of documentation required by the project. This can be achieved through reuse, the use of high-level modeling tools, and stabilizing the architecture early.

Benefits:

- Productivity
- Reduced complexity

Pattern:

- Reuse existing assets.
- Use higher-level tools and languages to reduce the amount of documentation produced.
- Focus on architecture first.
- Architect for resilience, quality, understandability, and complexity control.

Anti-patterns:

- To go directly from vague, high-level requirements to custom-crafted code:
  - Because few abstractions are used, a lot of the discussions are made at the code level compared to a more conceptual level, which misses many opportunities for reuse, among other things.
  - Informally captured requirements and other information require decisions and specifications to be revisited repeatedly.
  - Limited emphasis on architecture causes major rework late in the project.

2.3.6 Focus continuously on quality

This principle emphasizes that to achieve quality, it must be addressed throughout the project lifecycle. An iterative process is particularly adapted to achieving quality, because it offers many measurement and correction opportunities.

Benefits:

- Higher quality
- Earlier insight into progress and quality

Pattern:

- Ensure team ownership of quality for the product.
- Test early and continuously in step with integration of demonstrable capabilities.
- Incrementally build test automation.

Anti-patterns:

- To peer-review all artifacts and complete all unit testing before integration testing
- To conduct in-depth peer review of all intermediate artifacts, which is counterproductive because it delays application testing and hence identification of major issues
- To complete all unit testing before doing integration testing, again delaying identification of major issues

2.4 RUP lifecycle

This section describes the phases of a typical RUP project lifecycle.

2.4.1 Inception Phase

The overriding goal of the Inception Phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. The Inception Phase is of significance primarily for new development efforts, in which there are significant business and requirement risks, which must be addressed before the project can proceed. For projects focused on enhancements to an existing system, the Inception Phase is shorter but is still focused on ensuring that the project is both worth doing and possible to do.

Objectives

The primary objectives of the Inception Phase include:

- Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria, and what is intended to be in the product and what is not
- Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design trade-offs
- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios
- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the Elaboration Phase)
- Estimating potential risks (the sources of unpredictability)
- Preparing the supporting environment for the project

Essential activities

The essential activities of the Inception Phase include:

- **Formulating the scope of the project**. This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.
- **Planning and preparing a business case**. Evaluating alternatives for risk management, staffing, the project plan, and cost, schedule, and profitability trade-offs.
- **Synthesizing a candidate architecture**, evaluating trade-offs in design, and in make, buy, and reuse, so that cost, schedule, and resources can be estimated. The aim here is to demonstrate feasibility through a proof of concept. This might take the form of a model, which simulates what is required or an initial prototype, which explores what are considered to be the areas of high risk. The prototyping effort during inception needs to be limited to gaining confidence that a solution is possible and that the solution is realized during elaboration and construction.
- **Preparing the environment for the project**, assessing the project and the organization, selecting tools, and deciding which parts of the process to improve.

A typical iteration in inception is illustrated in Figure 2-2 on page 20.

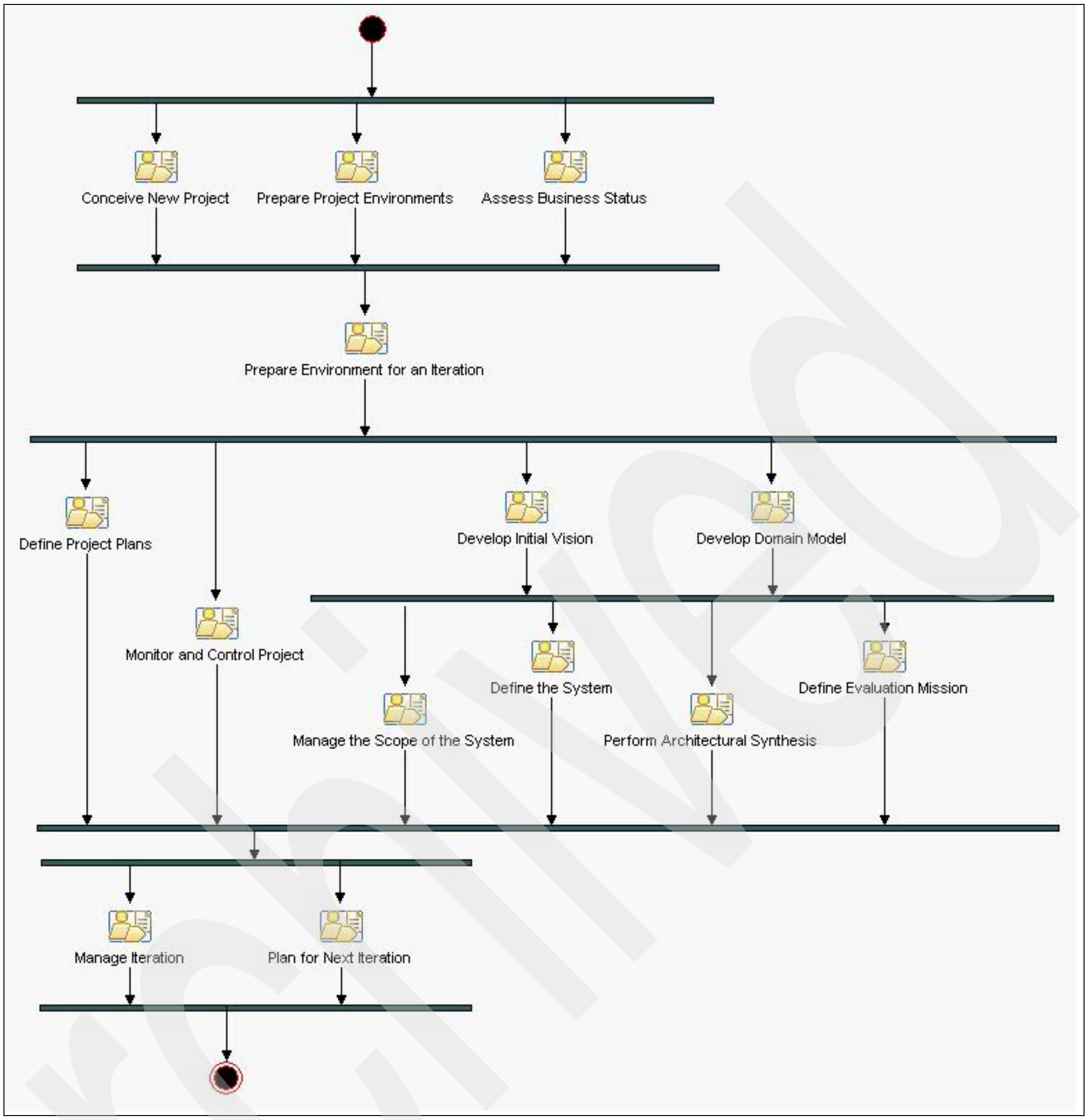


Figure 2-2 Typical iteration in the Inception Phase

Milestone

At the end of the Inception Phase is the first major project milestone or *Lifecycle Objectives Milestone*. At this point, you examine the lifecycle objectives of the project and decide either to proceed with the project or to cancel it.

Evaluation criteria:

- Stakeholder concurrence on scope definition and cost/schedule estimates.
- Agreement that the right set of requirements has been captured and that there is a shared understanding of these requirements.
- Agreement that the cost/schedule estimates, priorities, risks, and development process are appropriate.
- All risks have been identified and a mitigation strategy exists for each risk.

The project might be canceled or considerably rethought if it fails to reach this milestone.

2.4.2 Elaboration Phase

The goal of the Elaboration Phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the Construction Phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risk. The stability of the architecture is evaluated through one or more architectural prototypes.

Objectives

The primary objectives of the Elaboration Phase include:

- To ensure that the architecture, requirements, and plans are stable enough, and the risks sufficiently mitigated to be able to predictably determine the cost and schedule for the completion of the development. For most projects, passing this milestone also corresponds to the transition from a light-and-fast, low-risk operation to a high cost, high risk operation with substantial organizational inertia.
- To address all architecturally significant risks of the project.
- To establish a baseline architecture derived from addressing the architecturally significant scenarios, which typically expose the top technical risks of the project.
- To produce an evolutionary prototype of production-quality components, as well as possibly one or more exploratory, throwaway prototypes to mitigate specific risks, such as: design/requirement trade-offs, component reuse, and product feasibility or demonstrations to investors, clients, and users.
- To demonstrate that the baseline architecture will support the requirements of the system at a reasonable cost and in a reasonable time.
- To establish a supporting environment.

In order to achieve these primary objectives, it is equally important to set up the supporting environment for the project. This includes tailoring the process for the project, preparing templates, guidelines, and setting up tools.

Essential activities

The essential activities of the Elaboration Phase include:

- **Defining, validating, and baselining the architecture** as rapidly as practical.
- **Refining the vision**, based on new information obtained during the phase, establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions.
- **Creating and baselining detailed iteration plans for the Construction Phase**.
- **Refining the development process and putting in place the development environment**, including the process, tools, and automation support required to support the construction team.
- **Refining the architecture and selecting components**. Potential components are evaluated and the make, buy, and reuse decisions sufficiently understood to determine the Construction Phase cost and schedule with confidence. The selected architectural components are integrated and assessed against the primary scenarios. Lessons learned from these activities might well result in a redesign of the architecture, taking into consideration alternative designs or reconsideration of the requirements.

A typical iteration in elaboration is illustrated in Figure 2-3 on page 22.

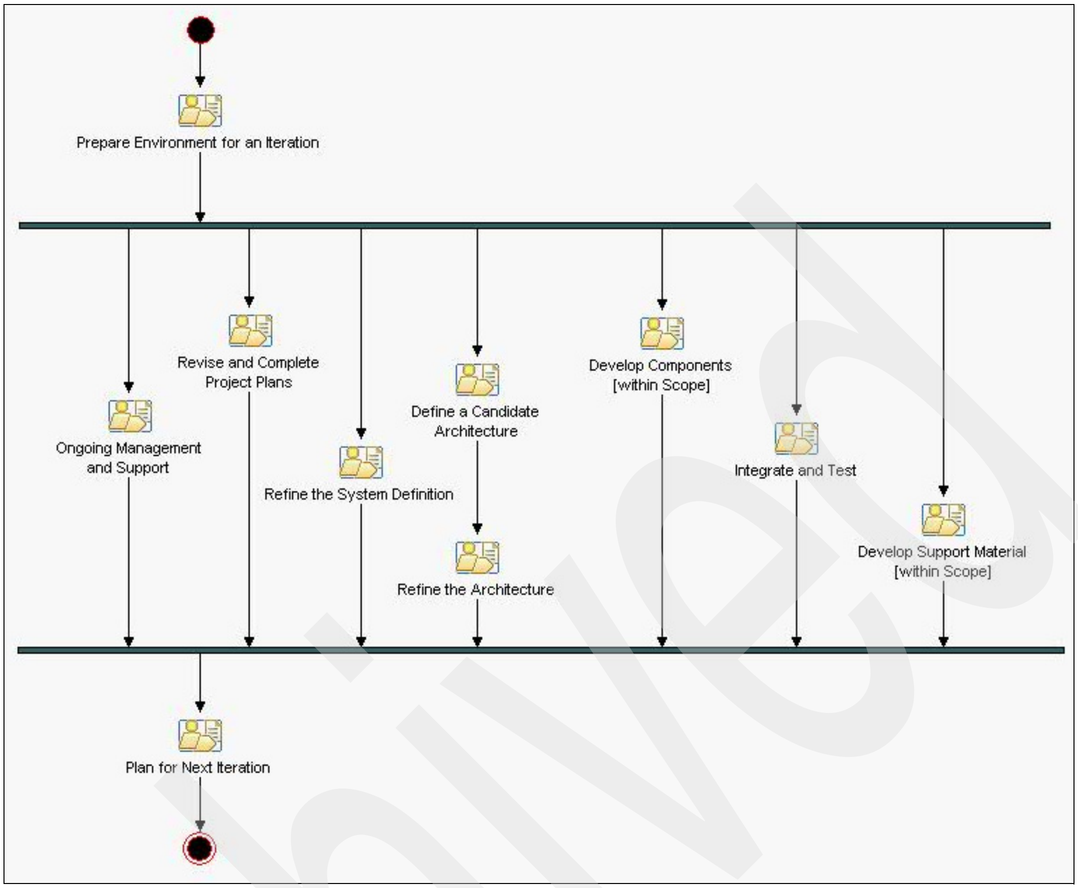


Figure 2-3 Typical iteration in Elaboration Phase

Milestone

At the end of the Elaboration Phase is the second important project milestone, the *Lifecycle Architecture Milestone*. At this point, you examine the detailed system objectives and scope, the choice of architecture, and the resolution of the major risks.

Evaluation criteria:

- The product vision and requirements are stable.
- The architecture is stable.
- The key approaches to be used in test and evaluation are proven.
- Test and evaluation of executable prototypes have demonstrated that the major risk elements have been addressed and have been credibly resolved.
- The iteration plans for the Construction Phase are of sufficient detail and fidelity to allow the work to proceed.
- The iteration plans for the Construction Phase are supported by credible estimates.
- All stakeholders agree that the current vision can be met if the current plan is executed to develop the complete system in the context of the current architecture.
- Actual resource expenditure as opposed to planned expenditure is acceptable.

The project may be aborted or considerably rethought if it fails to reach this milestone.

2.4.3 Construction Phase

The goal of the Construction Phase is clarifying the remaining requirements and completing the development of the system based upon the baseline architecture. The Construction Phase is in some sense a manufacturing process, where emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality. In this sense, the management mindset undergoes a transition from the development of intellectual property during inception and elaboration, to the development of deployable products during construction and transition.

Objectives

The primary objectives of the Construction Phase include:

- Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework.
- Achieving adequate quality as rapidly as practical.
- Achieving useful versions (alpha, beta, and other test releases) as rapidly as practical.
- Completing the analysis, design, development, and testing of all required functionality.
- To iteratively and incrementally develop a complete product that is ready to transition to its user community. This implies describing the remaining use cases and other requirements, fleshing out the design, completing the implementation, and testing the software.
- To decide if the software, the sites, and the users are all ready for the application to be deployed.
- To achieve some degree of parallelism in the work of development teams. Even on smaller projects, there are typically components that can be developed independently of one another, allowing for natural parallelism between teams (resources permitting). This parallelism can accelerate the development activities significantly, but it also increases the complexity of resource management and workflow synchronization. A robust architecture is essential if any significant parallelism is to be achieved.

Essential activities

The essential activities of the Construction Phase include:

- Resource management, control, and process optimization
- Complete component development and testing against the defined evaluation criteria
- Assessment of product releases against acceptance criteria for the vision

A typical iteration in construction is illustrated in Figure 2-4 on page 24.

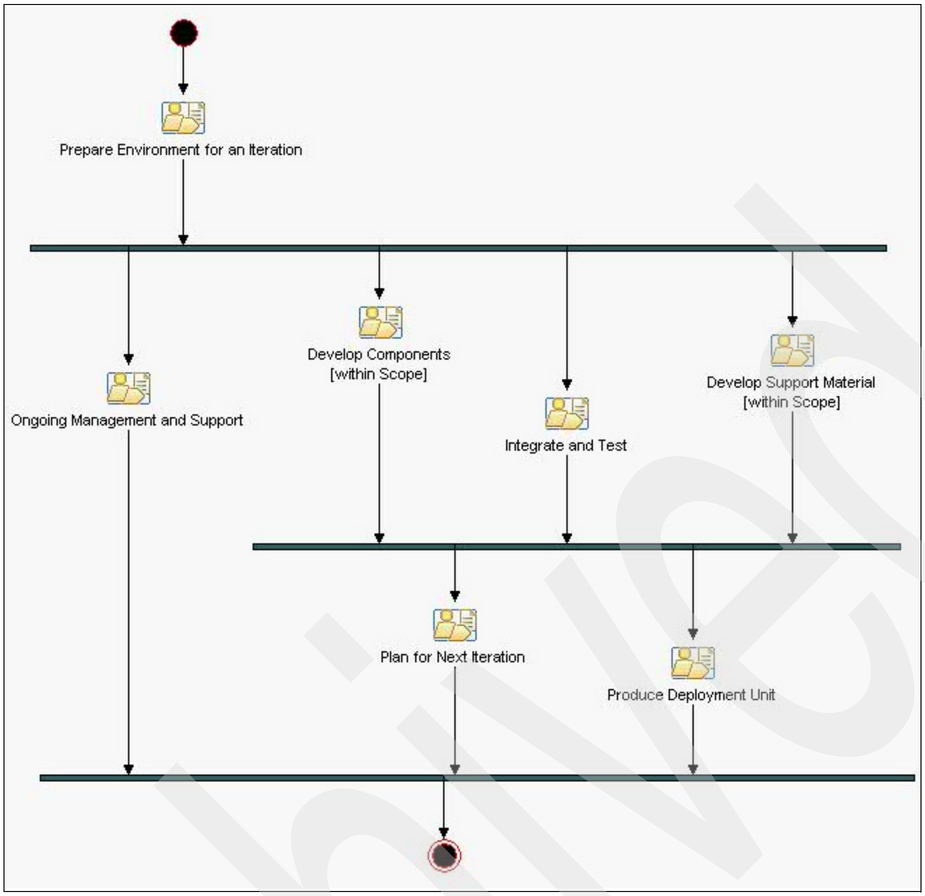


Figure 2-4 Typical iteration in Construction Phase

Milestone

At the *Initial Operational Capability Milestone*, the product is ready to be handed over to the Transition Team. All functionality has been developed and all *alpha* testing (if any) has been completed. In addition to the software, a user manual has been developed, and there is a description of the current release.

Evaluation criteria

The evaluation criteria for the Construction Phase involve the answers to these questions:

- Is this product release stable and mature enough to be deployed in the user community?
- Are all the stakeholders ready for the transition into the user community?
- Are actual resource expenditures as opposed to planned still acceptable?

Transition might have to be postponed by one release if the project fails to reach this milestone.

2.4.4 Transition Phase

The focus of the Transition Phase is to ensure that software is available for its users. The Transition Phase can span several iterations and includes testing the product in preparation for release and making minor adjustments based on user feedback. At this point in the lifecycle, user feedback needs to focus mainly on fine-tuning the product, configuring, installing, and usability issues, all the major structural issues need to have been worked out much earlier in the project lifecycle.

Objectives

By the end of the Transition Phase, lifecycle objectives should have been met and the project should be in a position to be closed out. In some cases, the end of the current lifecycle might coincide with the start of another lifecycle on the same product, leading to the next generation or version of the product. For other projects, the end of transition can coincide with a complete delivery of the artifacts to a third party, who might be responsible for operations, maintenance, and enhancements of the delivered system.

This Transition Phase ranges from being very straightforward to extremely complex, depending on the kind of product. A new release of an existing desktop product might be very simple, whereas the replacement of a nation's air-traffic control system can be exceedingly complex.

Activities performed during an iteration in the Transition Phase depend on the goal. For example, when fixing bugs, implementation and test are usually enough. If, however, new features have to be added, the iteration is similar to one in the Construction Phase requiring analysis, design, and so forth.

The Transition Phase is entered when a baseline is mature enough to be deployed in the user domain. This typically requires that some usable subset of the system has been completed with acceptable quality level and user documentation so that transitioning to the user provides positive results for all parties.

The primary objectives of the Transition Phase include:

- Beta testing to validate the new system against user expectations
- Beta testing and parallel operation relative to an existing system that it is replacing
- Converting operational databases
- Training users and those who will maintain the new system
- Rollout to the marketing, distribution, and sales forces
- Deployment-specific engineering, such as cutover, commercial packaging and production, sales rollout, and field personnel training
- Tuning activities, such as bug fixing, enhancement for performance, and usability
- Assessment of the deployment baselines against the complete vision and the acceptance criteria for the product
- Achieving user self-supportability
- Achieving stakeholder concurrence that deployment baselines are complete
- Achieving stakeholder concurrence that deployment baselines are consistent with the evaluation criteria of the vision

Essential activities

The essential activities of the Transition Phase include:

- Executing deployment plans
- Finalizing user support material
- Testing the deliverable product at the development site
- Creating a product release

- Getting user feedback
- Fine-tuning the product based on feedback
- Making the product available to users

A typical iteration in transition is illustrated in Figure 2-5.

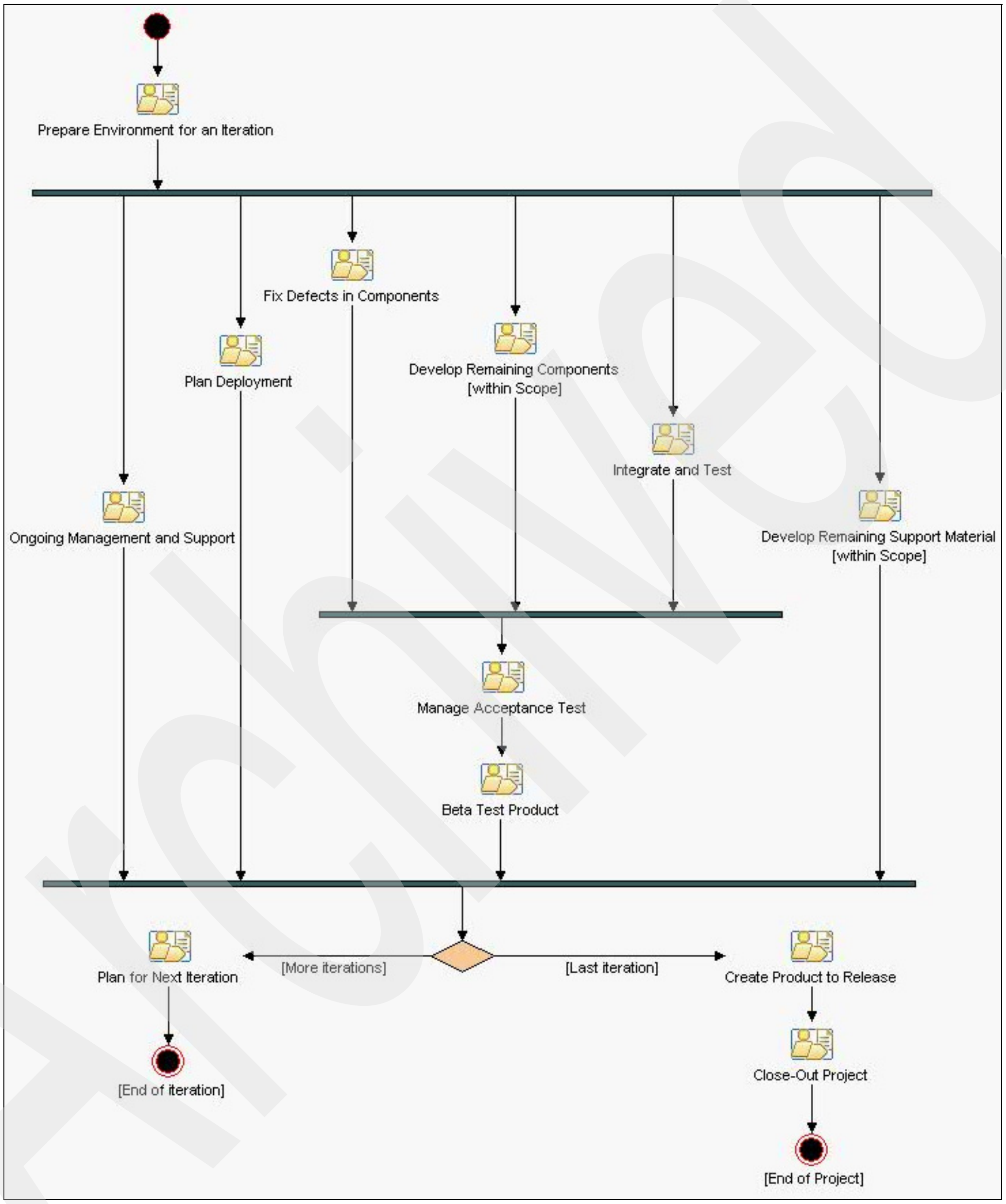


Figure 2-5 Typical iteration in the Transition Phase

Milestone

At the end of the Transition Phase is the fourth important project milestone, the *Product Release Milestone.* At this point, you decide if the objectives were met, and if you need to start another development cycle. In some cases, this milestone might coincide with the end of the Inception Phase for the next cycle. The Product Release Milestone is the result of the customer reviewing and accepting the project deliverables.

The primary evaluation criteria for the Transition Phase involve the answers to these questions:

- Is the user satisfied?
- Are actual resource expenditures compared to planned expenditures acceptable?

At the Product Release Milestone, the product is in production and the post-release maintenance cycle begins. This might involve starting a new cycle or additional maintenance release.

2.5 Developing service-oriented solutions

This section describes a roadmap through RUP when developing service-oriented solutions as defined in RUP for SOA. The presented method is called RUP/SOMA.

The SOMA (Service-Oriented Modeling and Architecture) method was developed as an engagement model within the IBM Global Business Services group, and while public papers and descriptions were available, it was primarily a method used by consultants in the field and not available to IBM clients. However, the RUP is a commercial product offering from IBM that clients use to develop their own software development processes. This integrated method offering, RUP/SOMA, has been developed to bring the unique aspects of SOMA to the RUP commercial method and make these available to commercial clients.

The framework for RUP/SOMA is described in Figure 2-6, which demonstrates the key phases of the method, including the influences driving each phase and the artifacts produced. Note that the key artifact manipulated by the method is the Service Model in 2.5.4, "Service Model" on page 29.

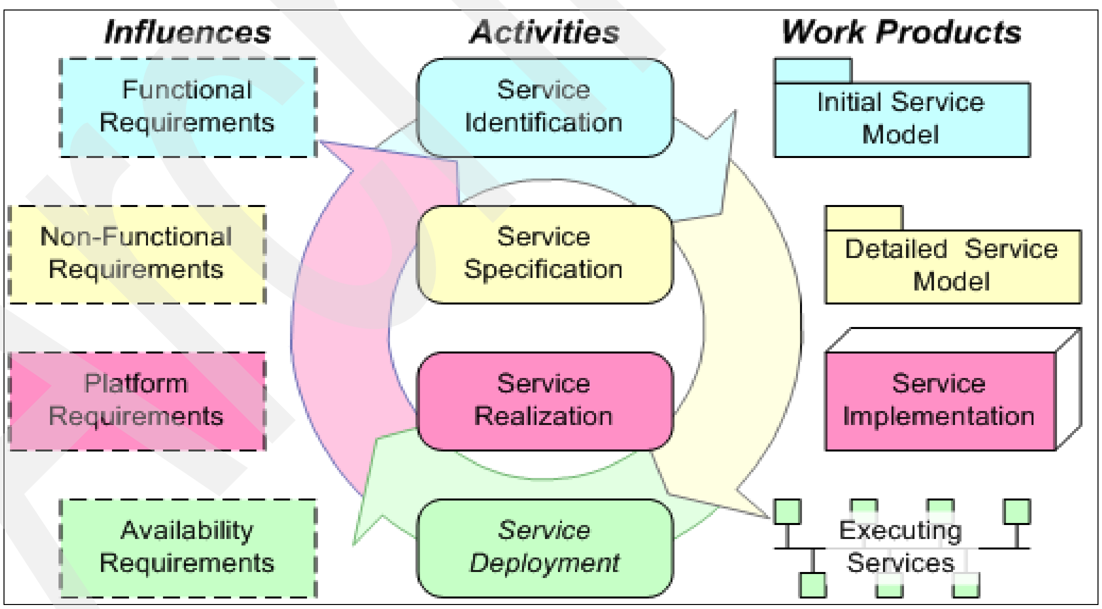


Figure 2-6 The RUP/SOMA framework

2.5.1 Service Identification

Service Identification is primarily an Elaboration time set of activities, focused on the identification of candidate services from the set of assets from both business and IT. The workflow for Service Identification is shown in Figure 2-7 on page 28.

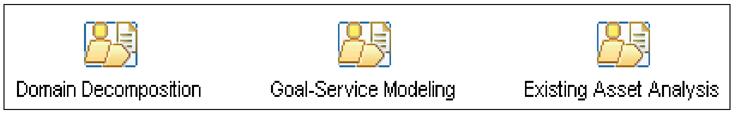


Figure 2-7 Service Identification workflow

The tasks identified within this set of activities are:

- Task: Functional Area Analysis
- Task: Refine a Business Use Case
- Task: Business Process Analysis
- Task: Business Use-Case Analysis (SOA)
- Task: Identify Business Goals and key performance indicators (KPIs)
- Task: Identify and Associate Services to Goals
- Task: Existing Asset Analysis
- Task: Data Model Analysis
- Task: Business Rule Analysis
- Task: Construct Architectural Proof-of-Concept (SOA)

2.5.2 Service Specification

Service Specification is primarily an Elaboration time set of activities, focused on the selection of candidate services that will be developed into full services. These services are then allocated to subsystems also identified above and then decomposed into sets of components for implementation. The workflow for Service Specification is shown in Figure 2-8.

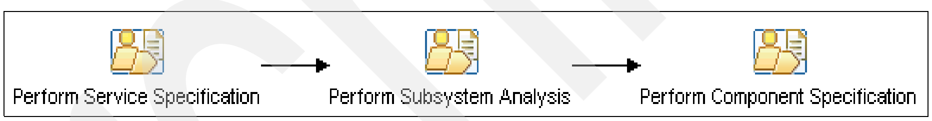


Figure 2-8 Service Specification workflow

The tasks identified within this set of activities are:

- Task: Apply Services Litmus Tests
- Task: Service Specification
- Task: Message Design
- Task: Identify Security Patterns
- Task: Subsystem Design (SOA)
- Task: Component Specification (SOA)

2.5.3 Service Realization

Service Realization is primarily a Construction time set of activities, focused on the completion of component design being ready for component implementation. The workflow for Service Realization is shown in Figure 2-9 on page 29.

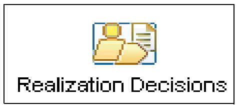


Figure 2-9 Service Realization workflow

The tasks identified within this activity are:

- Task: Document Service Realization Decisions
- Task: Component Specification (SOA)
- Task: Construct Architectural Proof-of-Concept (SOA)

2.5.4 Service Model

In SOMA, the Service Model is described using Figure 2-10; it is a single, document-based, work product that encompasses the different technical and lifecycle views of the services identified and specified during a project. The different sections of the service model are listed in more detail in the Artifact: Service Model in RUP/SOMA.

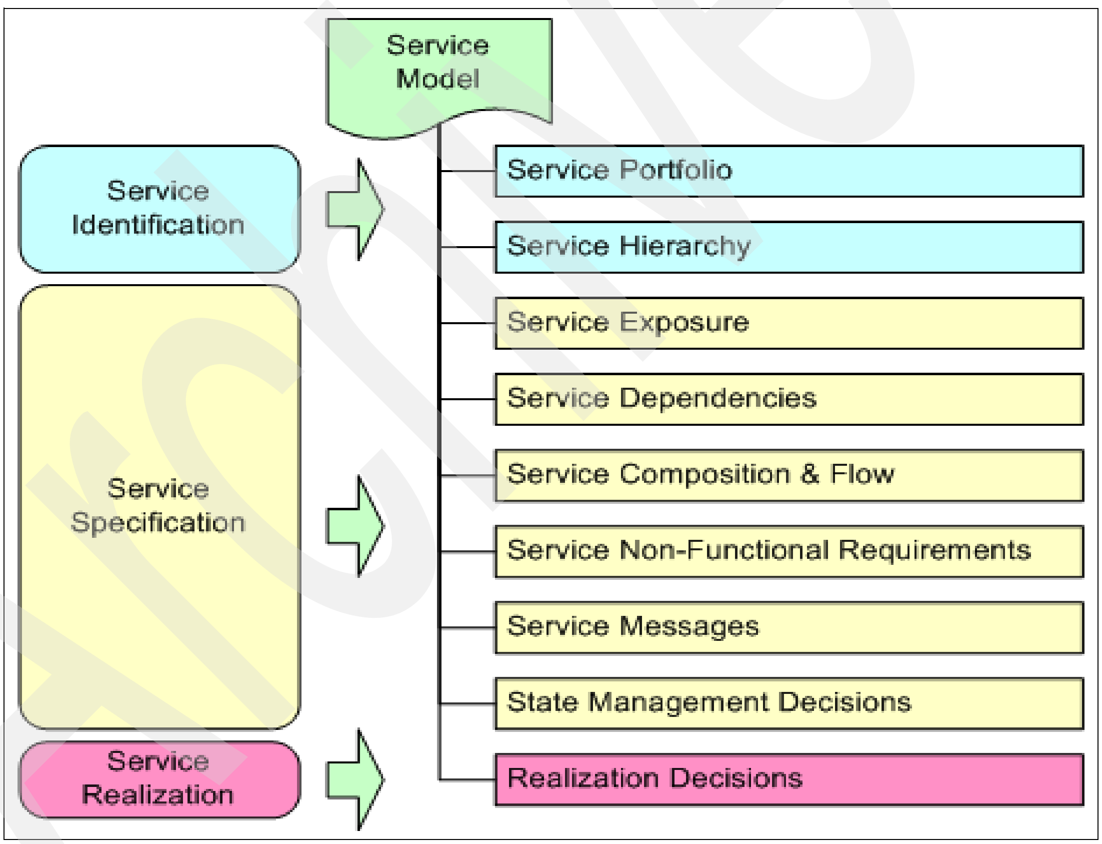


Figure 2-10 Service Model

The RUP Artifact: Service Model is described in both a document form and a Unified Modeling Language (UML) form (Template: Service Model in Word and Template: Service Model in UML) though it is more likely that a project will use elements of both of these forms in presenting the results of their work.

3

Why the IBM Rational Unified Process for System z

In this chapter, we discuss the reasons and the rationale behind the creation of a RUP for System z. We also describe the main differences between the older waterfall development model and the RUP iterative development model. And finally, we describe the evolution of RUP for System z.

3.1 Mainframe software development: A key business capability

More and more businesses today rely on mainframe servers to power their transformation into on demand enterprises. System z provides business integration and business resilience: much desired and necessary capabilities in today's on demand world. It provides businesses the capability to dynamically respond to changing business conditions by being more flexible and responsive to change.

With the demand for flexibility and responsiveness comes the need and responsibility to provide supportive applications and systems to meet user needs. Delivery of applications and systems must be both timely and of a consistent quality. In addition, applications that we build must accurately satisfy requirements, so that the users get precisely what they need to be able to achieve their business objectives optimally.

3.2 System z application development: A tradition

System z and its application developers have been around for arguably more years than their counterparts in other environments. System z and its application developers have long been pioneers in creating and following application development methodologies, recognizing that process is a required and important element to producing quality software consistently. Traditionally, and even today, methods and processes used in the System z development environment are generally based on a *waterfall* lifecycle model. For example, we generally adopt a workflow that progresses sequentially from Requirements gathering, on to Analysis, and then through Design, Code and Unit Testing, through to Testing and Deployment, for example, General Availability (GA).

The waterfall model, as the name suggests, is derived from the cascading effects of a waterfall with a distinct start and end, with an ordered number of phases in between, from requirements gathering and analysis, design, implementation and integration, concluding with testing at the very end. A phase is not started until a prior stage is completed.

With the waterfall model approach, there are generally only two user interaction points: the first during the Requirements gathering stage and the other at the final Deployment stage when the application solution or product is handed over to the users. Obviously, there can be and generally there are surprises at the end in that user needs have been accurately met. Figure 3-1 on page 33 shows a traditional waterfall model diagram.

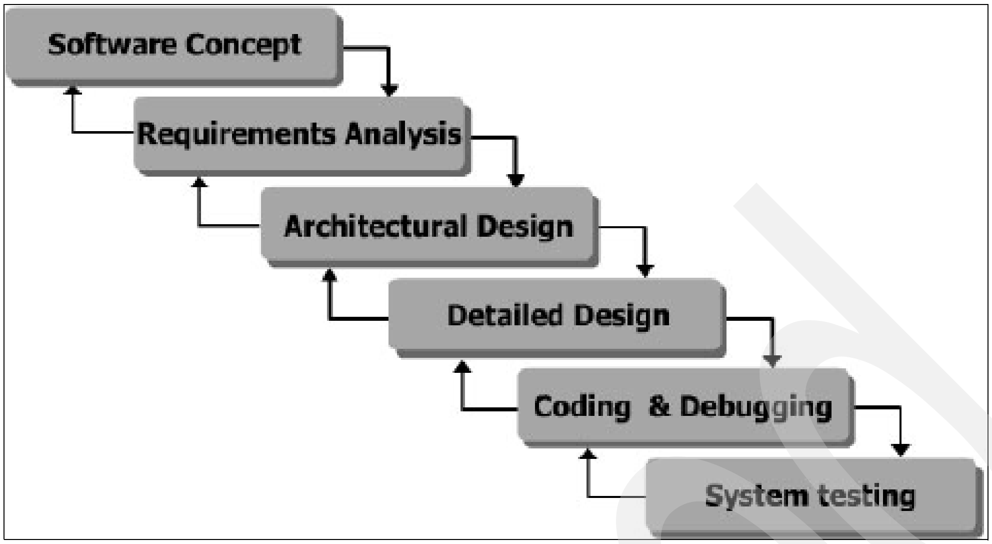


Figure 3-1 Waterfall development model

So is it time for the System z development world to be introduced to a more modern and thoroughly tested development approach? We think so.

Today, as software development is becoming a key business capability, our best practices are maturing within the larger context of business-driven development. We think RUP for System z is the method to adopt to take System z development and its resulting application products to the next higher level of quality and consumerability.

3.3 What is different

With the passage of time and changing business demands, older methods and processes are beginning to show shortcomings. Several of the main deficiencies of the waterfall model are in the areas of requirements gathering and product deployment. Both deficiencies are related to the fact that the waterfall model is a linear model, which means that when one phase or discipline is completed, the project moves on to the next phase.

The problem with the requirements discipline in the waterfall model is that it is usually performed one time at the beginning of the project based on which product or application is built. A considerable amount of time elapses between interaction and input from the user toward deriving the requirements, to the time the product is built, tested, and delivered. By then, more likely than not, user needs have changed in order to keep pace with the current climate of frequently changing business conditions. The primary problem here is the inability to easily adapt to changing user requirements.

The problem with deployment in the waterfall model is that there is generally only one deliverable handed over to the user in the form of the final product, which is delivered at the very end of the development cycle. This practice, as alluded to earlier, creates cause for surprises because user needs are not accurately met.

RUP addresses these same deficiencies and provides many more other benefits. Rather than prescribing a plan-build-assemble sequence of activities for a software project, RUP is an *iterative,* incremental process that steers development teams toward results more quickly.

An *iteration,* indicated by the circular arrows in Figure 3-2 is defined as “executing the same set of activities a certain number of times or until a specified result is obtained.”

But what in fact are these activities, how many times (iterations) must these activities be executed, and how long must each iteration be?

Well, the activities as applied to the iterative model are related to the development disciplines: requirements analysis, design, implementation, integration, and testing, all together comprising a single iteration in the RUP paradigm as illustrated in Figure 3-3.

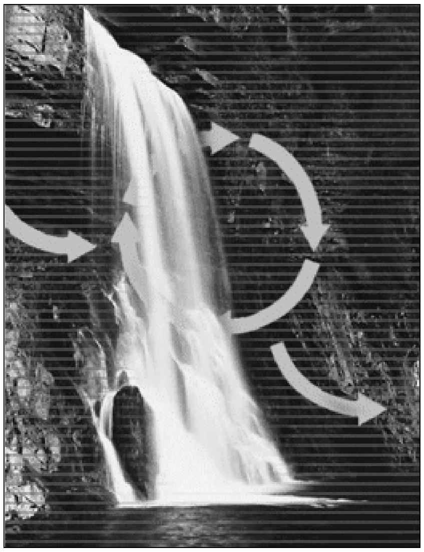


Figure 3-2 Waterfall and Iterative process

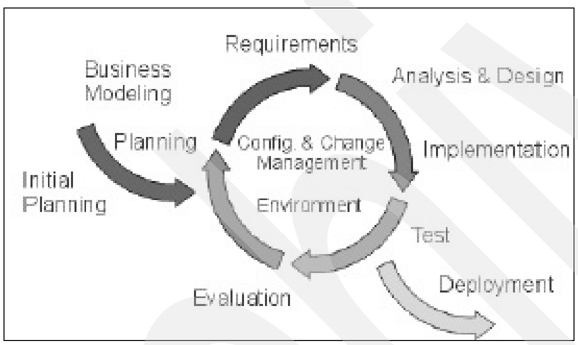


Figure 3-3 What is an iteration

An iteration typically spans two to three weeks, but the number and size of the iterations are project specific. The iteration interval after it has been decided must remain constant throughout the project. Furthermore, the iterations are divided into four phases: Inception, Elaboration, Construction, and Transition as illustrated in Figure 3-4 on page 35.

Kurt Bitnner, IBM Communities of Practice Architect, in his paper *Driving Iterative Development With Use Cases,* March 2006, states that each iteration concludes in a minor milestone being met and each phase concludes in a major milestone being met.

Furthermore, each iteration produces a partial working implementation of the final system with each implementation building on the previous implementation until the final product is complete, states Per Kroll, Manager of Methods, IBM Rational, in *Transitioning from waterfall to iterative development*, April 2004.

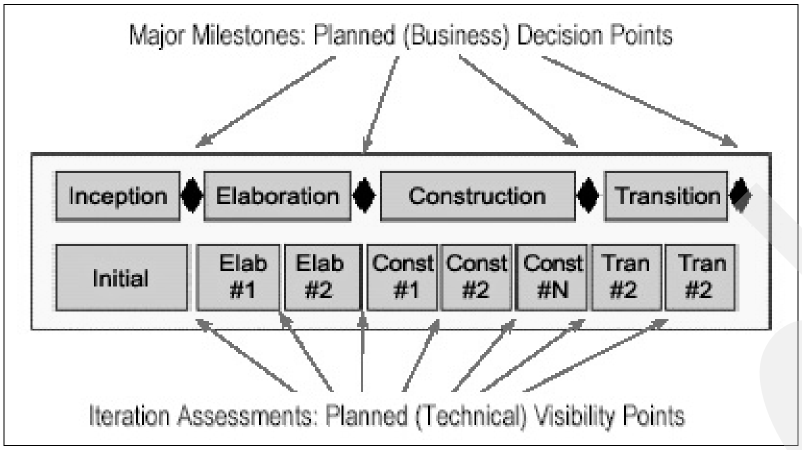


Figure 3-4 Iterations and Phases

3.4 Iterative compared to waterfall: Differences and benefits

With the practices advocated by RUP, software requirements management is a more user-interactive activity. Product requirements are continually and frequently tracked and validated against stakeholder needs. Also, the product is built incrementally, with the most complex and riskiest components designed and built first in order to validate with the user that their requirements and the product being built are in sync. Besides, because of the iterative practice of RUP, other development activities, such as test and product documentation are implemented from early on in the development, thereby ensuring quality and documentation support of the early deliverable.

So, in summary, the key differences between the waterfall and RUP's iterative process are:

- **Requirements** is done not only at the beginning but continues throughout the process, because requirements by nature change over time and development efforts need to be aligned to stakeholder needs.
- **Implementation** starts earlier to enable early stakeholder feedback, which is key to ensure that we build the right system.
- **Test** starts earlier. Because the later the discovery of defects, the more costlier it is to put things right.
- **Project** plans are refined throughout the project based on a continuous re-evaluation (at least one time per iteration) of risks and priorities.

3.5 Evolution of RUP for System z

The evolution of the activities involved in software development has occurred over the last several decades, and it has evolved as per the demands of the times. However, in recent times, there have been significant changes in the scope and speed of developing software, as well as the tools and languages used to develop software. The RUP, being a proven method based on principles that characterize the industry's best practices in the creation, deployment, and evolution of software-intensive systems, is recognized as being the modern day method to address and cope with all the current demands and pressures placed on software development organizations.

RUP by itself is a vast repository or knowledge base of best practices. Its method content is comprised of tasks, roles, and work products that pertain to software development in general and is applicable to all development environments.

RUP is an all-encompassing modern process framework. It is generic in nature, because it applies to any software development environment: big or small, old or new. However, there is a common belief that its practices are applicable to more modern and newer technologies and their associated programming languages, for example, Object-Oriented development and programming JAVA, and so forth.

As we all know, the System z environment is still the powerhouse of the industry, powering and driving all the mission-critical systems and applications that keep the largest modern day enterprises up and running, ready to adapt quickly and efficiently to the next round of changes that future innovation will bring.

For this reason, we thought it necessary to produce a development method specifically for System z practitioners, a method that depicts software development practices currently in use in the System z environment while still leveraging some of the modern best practices encompassed in RUP.

RUP for System z provides practitioners with specific software development guidance and a succinct end-to-end process dedicated to the System z environment. RUP for System z includes a large set of work product examples taken from an application created in CICS Cobol and turned into Web services. The end-to-end lifecycle is available in the form of a Work Breakdown Structure (WBS).

4

IBM Rational Unified Process for System z roadmap

The IBM Rational Unified Process for System z (RUP for System z) roadmap walks through each phase (inception, elaboration, construction, and transition) of a typical system z development project.

The RUP for System z roadmap addresses green field development and system evolution with architectural changes (including turning an existing capability into a Web service for instance) or with significant impact on existing user business processes. Refer to 4.6, “Note on maintenance projects” on page 49 for a discussion about pure maintenance projects.

4.1 Introduction

The IBM Rational Unified Process for System z (RUP for System z) roadmap is illustrated in Figure 4-1. The roadmap provides an overview for each of the elements in the figure.

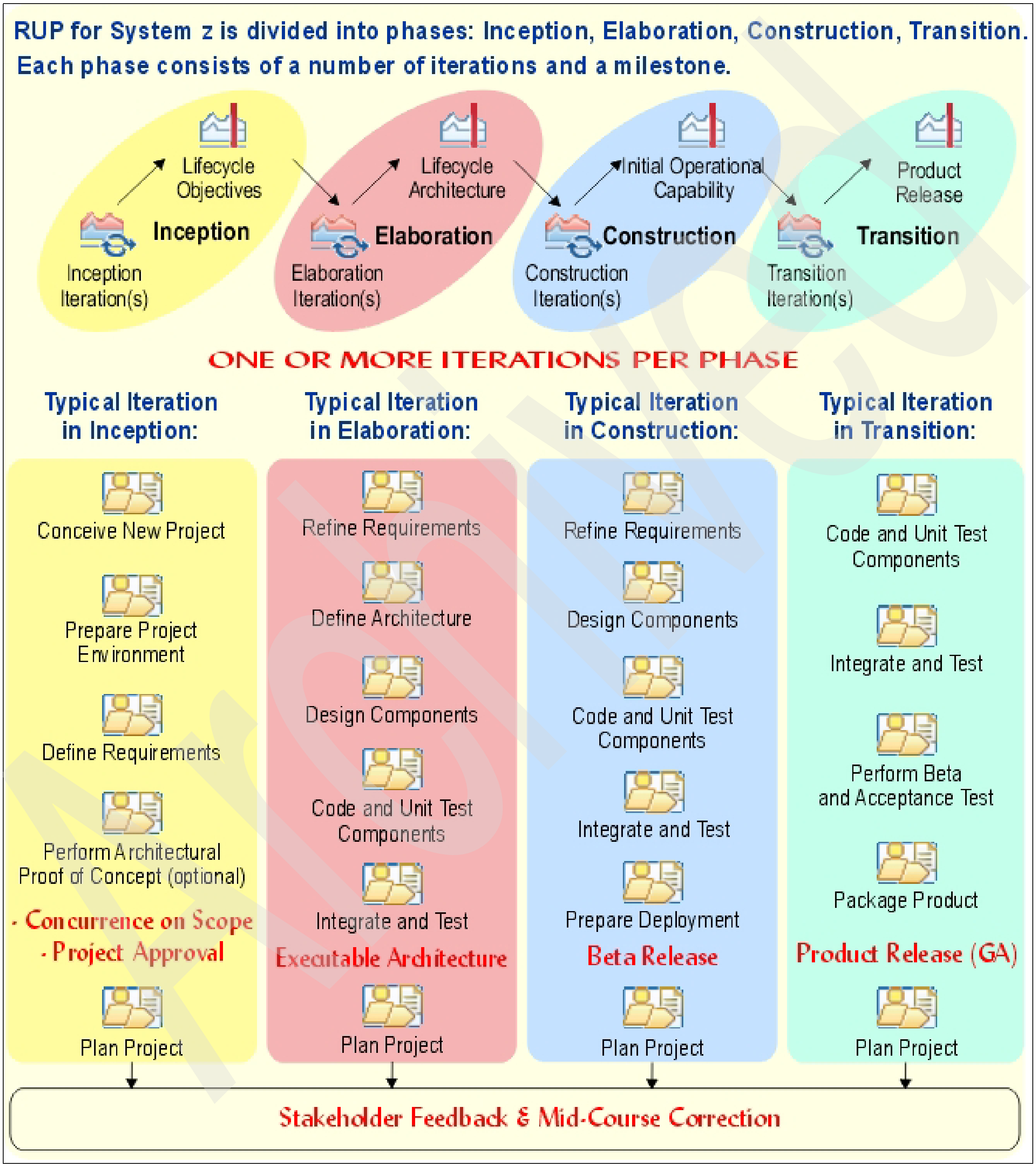


Figure 4-1 RUP for System z lifecycle

The activities forming each iteration in Figure 4-1 on page 38 can be executed in sequence or in any order. Indeed, in RUP, an *iteration* is not necessarily a sequence of activities but a more complex combination of activities, including possible parallelism among activities.

4.2 Inception Phase overview

The overriding goal of the Inception Phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. The Inception Phase is of significance primarily for new development efforts, in which there are significant business and requirement risks, which must be addressed before the project can proceed. For projects focused on enhancements to an existing system, the Inception Phase is brief but is still focused on ensuring that the project is both worth doing and possible to do. The Inception Phase consists of a number of iterations culminating in the Lifecycle Objectives Milestone.

4.2.1 Inception objectives

The primary objectives of the Inception Phase include:

- Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria, what is intended to be in the product, and what is not.
- Discriminating the critical use cases of the system, which are the primary scenarios of operation that will drive the major design trade-offs.
- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios.
- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the Elaboration Phase).
- Estimating potential risks, which are the sources of unpredictability.
- Preparing the supporting environment for the project. This can include tailoring the process for the project, preparing templates, guidelines, and setting up tools if necessary.

4.2.2 Typical inception iteration

This section provides an overview of the activities performed in a typical iteration of the Inception Phase, as illustrated in Figure 4-2.

Figure 4-2 Typical iteration in the Inception Phase

The Conceive New Project activity brings a project from the initial germ of an idea to a point at which a reasoned decision can be made to continue or abandon the project. During this activity, an economic analysis, the Business Case, is produced and the risks are assessed. The Business Case, Risk List, and initial Vision are reviewed. If found satisfactory, the project is formally set up and is given limited sanction (and budget) to begin the planning effort. An initial draft of the Software Development Plan is created.

The Prepare Project Environment activity prepares the development environment for a project, where the development environment includes both process and tools. This activity includes establishing an environment where the overall product can be developed, built, and made available for stakeholders.

The Define Requirements activity covers the definition of the project Vision. It gains agreement on the scope of the system and outlines the key requirements. The requirements can be described in terms of a Use-Case Model, which includes Use Cases and Actors. The primary purpose of a *use case* is to capture the required system behavior from the perspective of the user in achieving one or more desired goals. A use case represents one or more sequences of actions that a system performs that yield an observable result of value to a particular actor, such as a user. In inception, the main use cases are identified and briefly described. The requirements (functional and non-functional), which do not fit appropriately within the use cases, must be documented in the Supplementary Specifications. After several use cases and supplementary requirements are identified, they are prioritized so that their order of development can be decided. For instance, use cases that represent some significant functionality, have a substantial architectural coverage (that exercises many architectural elements), or stress or illustrate a specific and delicate point of the architecture, will be developed first. In addition, the project terms must be defined in a Glossary, which will be maintained throughout the life of the project. This activity also kicks off the test effort by providing a first draft of the Test Plan.

The Perform Architectural Proof-of-Concept activity aims at demonstrating the solution feasibility by building an Architectural Proof-of-Concept and assessing the viability of this architectural proof-of-concept. An architectural proof-of-concept can take many forms, such as a sketch of a conceptual model of a solution using a notation, such as Unified Modeling Language (UML), a simulation of a solution, or an executable prototype. The architectural proof-of-concept is assessed against the architecturally significant requirements. Requirements that are typically architecturally significant include performance, scaling, process and thread synchronization, and distribution.

The Plan the Project activity starts with an assessment of the current iteration and a reevaluation of the risks. It refines the Software Development Plan (covering all project phases and iterations) and creates a fine-grained Iteration Plan for the next iteration or iterations. This activity also acquires the necessary resources (including staff) to perform the coming iteration or iterations.

4.2.3 Lifecycle objectives milestone

At the end of the Inception Phase is the first major project milestone or *Lifecycle Objectives Milestone*. At this point, you examine the lifecycle objectives of the project and decide either to proceed with the project or to cancel it. At the end of the Inception Phase, the project is evaluated against the following criteria:

- Stakeholder concurrence on scope definition and cost/schedule estimates.
- Agreement that the right set of requirements has been captured and that there is a shared understanding of these requirements.
- Agreement that the cost/schedule estimates, priorities, risks, and development process are appropriate.
- All risks have been identified and a mitigation strategy exists for each risk.

The project might be canceled or considerably rethought if it fails to reach this milestone.

A summary of essential work products and their state at the end of the Inception Phase:

- Business Case (100% complete)
- Vision (about 100% complete)
- Glossary (about 40% complete)
- Software Development Plan (about 80% complete)
- Iteration Plan for the first elaboration iteration (about 100% complete)
- Risk List (about 25% complete)
- Use-Case Model (about 20% complete)
- Supplementary Specifications (about 20% complete)
- Test Plan (about 10% complete)
- Software Architecture Document (about 10% complete)
- Architectural Proof-of-Concept (one or more proof of concept prototypes available to address very specific risks)

4.3 Elaboration Phase overview

The goal of the Elaboration Phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the Construction Phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risk. The stability of the architecture is evaluated through one or more architectural prototypes. The Elaboration Phase consists of a number of iterations culminating in the Lifecycle Architecture Milestone.

4.3.1 Elaboration objectives

The primary objectives of the Elaboration Phase include:

- Ensuring that the architecture, requirements, and plans are stable enough and the risks are sufficiently mitigated to be able to predictably determine the cost and schedule for the completion of the development. For most projects, passing this milestone also corresponds to the transition from a light-and-fast, low-risk operation to a high cost, high risk operation with substantial organizational inertia.
- Addressing all architecturally significant risks of the project.
- Establishing a baseline architecture derived from addressing the architecturally significant scenarios, which typically expose the top technical risks of the project.
- Producing an evolutionary prototype of production-quality components, as well as possibly one or more exploratory, throw-away prototypes to mitigate specific risks such as: design/requirements trade-offs, component reuse, and product feasibility or demonstrations to investors, clients, and users.
- Demonstrating that the baseline architecture will support the requirements of the system at a reasonable cost and in a reasonable amount of time.
- Refining the supporting environment.

4.3.2 Typical elaboration iteration

This section provides an overview of the activities performed in a typical iteration of the Elaboration Phase, as illustrated in Figure 4-3.

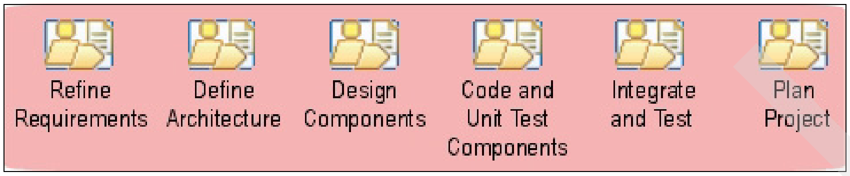


Figure 4-3 Typical iteration in the Elaboration Phase

The Refine Requirements activity addresses detailing the requirements of the system in terms of its use cases. Only the use cases that are in the scope of the current iteration are detailed in order to reach the goal of the iteration. The remaining use cases will be detailed in later iterations. Detailing a use case involves describing its flow of events (see Guideline: Use Case for guidance about how to detail a use-case flow of events). The requirements (functional and non-functional) that do not fit appropriately within the use cases need to be detailed in the supplementary specifications. Use cases and supplementary requirements continue to be prioritized, so that their order of development can be decided. The project terms continue to be defined or refined in the Glossary.

The Define Architecture activity starts by creating an initial sketch of the software architecture in Define Candidate Architecture. This sub-activity defines a candidate architecture (initial system organization), leverages existing assets, defines the architectural patterns, identifies the architecturally significant use cases, and perform a use-case analysis (also called a *use-case realization*) on each candidate architecture. This includes identifying the Analysis Elements necessary to express the behavior of each use case. These analysis elements will be later refined into design elements, such as Modules and Design Classes, and source code. After a candidate architecture has been defined, the Define Architecture activity completes the architecture for an iteration in Refine the Architecture. This sub-activity provides the natural transition from analysis activities to design activities by identifying appropriate design elements from analysis elements. It also describes the organization of the system's run-time and deployment architecture and maintains the consistency and integrity of the architecture. The activity ends with a review of the resulting architecture, as documented in the Software Architecture Document.

The Design Components activity addresses the detailed design of one or more components within the scope identified in the iteration plan. The main goals are (see Perform Component Design):

- Refine the definition of each component behavior with interactions of design elements (modules, classes, interfaces, events, and so forth) or remaining Analysis Elements.
- Identify new design elements by analyzing these interactions.
- Partition the design elements into subsystems and document the internal structure and behavior of the subsystems, their interfaces, and their dependencies. These subsystems can now be refined and implemented separately from each other.
- Refine the definitions of design elements by working out the "details" of how they realize the behavior required of them.

When services are involved, the Design Components activity refines the design with service elements (see Design Services). When databases are involved, this activity identifies the design elements to be persisted in a database and designs the corresponding database structures (see Design Databases). When a user-interface is involved, this activity models and prototypes the user interface (see Design User Interface). This activity ends with a review of the resulting design (see Review the Design), as documented in the Design Model, and optionally other supporting models, such as the Service Model.

The Code and Unit Test Components activity completes a part of the implementation so that it can be delivered for integration. This activity implements the elements in the design model by writing source code, adapting existing source code, and compiling, linking, and performing unit tests. If defects in the design are discovered, rework feedbacks on the design are submitted. The activity also involves fixing code defects and performing unit tests to verify the changes. The code is reviewed to evaluate quality and compliance with the programming guidelines.

The Integrate and Test activity covers the integration and test of the product. The Integrate sub-activity integrates changes from multiple implementers to create a new consistent version of an Implementation Subsystem (this is done for any implementation subsystem within the scope of the iteration) and integrates implementation subsystems to create a new consistent version of the overall system when appropriate. The integrator integrates the system, in accordance with the Integration Build Plan, by adding the delivered implementation subsystems into the system integration workspace and creating Builds. Each build is then integration-tested by a tester. After the last increment, the build can be completely system-tested. The Test sub-activity includes the tasks required for testing within a particular scope. The scope could be a specific level or type of test, such as Functional Verification Test (FVT) or System Verification Test (SVT). It might also be limited to the components, or portions thereof, that have been implemented or are planned to be implemented during the iteration. The Test sub-activity includes the refinement of the Test Plan, the definition of Test Cases and their implementation into Test Scripts (a *test script* is a step-by-step instruction enabling the execution of a test case), the execution and evaluation of the tests (by a group of tests called Test Suite created to exercise a category of tests, such as FVT or SVT), and the corresponding reporting of incidents that are encountered. It also includes the definition and implementation of the Installation Verification Procedures (IVPs).

The Plan the Project activity starts with an assessment of the current iteration and a re-evaluation of the risks. It refines the Software Development Plan (covering all project phases and iterations) and creates a fine-grained Iteration Plan for the next iteration or iterations. This activity also acquires the necessary resources (including staff) to perform the coming iteration or iterations.

4.3.3 Lifecycle architecture milestone

At the end of the Elaboration Phase is the second important project milestone, the *Lifecycle Architecture Milestone*. At this point, you examine the detailed system objectives and scope, the choice of architecture, and the resolution of the major risks. At the end of the Elaboration Phase, the project is evaluated against the following criteria:

- The product vision and requirements are stable.
- The architecture is stable.
- The key approaches to be used in test and evaluation are proven.
- Test and evaluation of executable prototypes have demonstrated that the major risk elements have been addressed and have been credibly resolved.
- The iteration plans for the Construction Phase are of sufficient detail and fidelity to allow the work to proceed.
- The iteration plans for the Construction Phase are supported by credible estimates.

- All stakeholders agree that the current vision can be met if the current plan is executed to develop the complete system in the context of the current architecture.
- The actual resource expenditure compared to the planned expenditure is acceptable.

The project might be canceled or considerably rethought if it fails to reach this milestone.

Summary of essential work products and their state at the end of the Elaboration Phase:

- Glossary (about 80% complete)
- Software Development Plan (about 95% complete)
- Iteration Plans for the construction iterations (about 100% complete, at least for first iteration)
- Risk List (about 50% complete)
- Use-Case Model (about 80% complete)
- Supplementary Specifications (about 80% complete)
- Software Architecture Document (about 100% complete)
- Design Model (about 60% complete)
- Service Model (about 60% complete)
- Test Plan (about 30% complete)
- Test Cases (about 40% complete)
- Test Scripts (about 40% complete)
- Implementation Elements, including source code (about 40% complete)
- Builds are available (one or more per iteration, for instance)
- One or more executable architectural prototypes are available (to explore critical functionality and architecturally significant scenarios)
- Installation Verification Procedures (IVPs) (about 80% complete)

4.4 Construction Phase overview

The goal of the Construction Phase is completing the development of the system based upon the baseline architecture. The Construction Phase is in some sense a manufacturing process, where emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality. In this sense, the management mind-set undergoes a transition from the development of intellectual property during inception and elaboration, to the development of deployable products during construction and transition. The Construction Phase consists of a number of iterations culminating in the *Initial Operational Capability Milestone.*

4.4.1 Construction objectives

The primary objectives of the Construction Phase include:

- Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework.
- Achieving adequate quality as rapidly as practical.
- Achieving useful versions (alpha, beta, and other test releases) as rapidly as practical.
- Completing the analysis, design, development, and testing of all required functionality.

- Iteratively and incrementally developing a complete product that is ready to transition to its user community. This implies describing the remaining use cases and other requirements, filling out the design, completing the implementation, and testing the software.
- Deciding if the software, the sites, and the users are all ready for the application to be deployed.
- Achieving some degree of parallelism in the work of development teams. Even on smaller projects, there are typically components that can be developed independently of one another, allowing for natural parallelism among teams (resources permitting). This parallelism can accelerate the development activities significantly, but it also increases the complexity of resource management and workflow synchronization. A robust architecture is essential if any significant parallelism is to be achieved.

4.4.2 Typical construction iteration

This section provides an overview of the activities performed in a typical iteration of the Construction Phase, as illustrated in Figure 4-4.

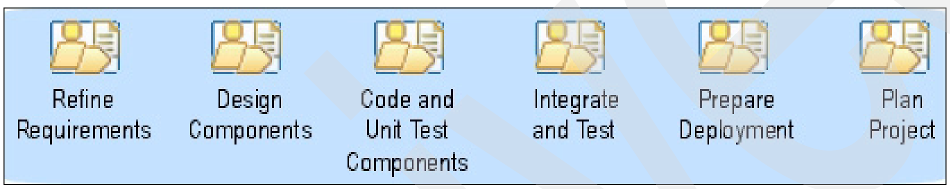


Figure 4-4 Typical Iteration in the Construction Phase

The Refine Requirements activity completes detailing the requirements of the system in terms of Use Cases. The requirements (functional and non-functional) that do not fit appropriately within the use cases must be detailed in the Supplementary Specifications. The project terms continue to be defined or refined in the Glossary.

The Design Components activity completes the detailed design of the components within the scope identified in the iteration plan. This activity ends with a review of the resulting design (see Review the Design) as documented in the Design Model and optionally other supporting models, such as the Service Model.

The Code and Unit Test Components activity completes most parts of the implementation so that they can be delivered for integration. This activity implements the elements in the design model by writing source code, adapting existing source code, compiling, linking, and performing unit tests. The code is reviewed to evaluate quality and compliance with the programming guidelines.

The Integrate and Test activity covers the integration and test of the product. The Integrate sub-activity integrates changes from multiple implementers to create a new consistent version of an Implementation Subsystem (this is done for any implementation subsystem within the scope of the iteration) and integrates implementation subsystems to create a new consistent version of the overall system when appropriate. The Test sub-activity includes the tasks required for testing within a particular scope, such as Functional Verification Test (FVT) of the components implemented during the iteration or System Verification Test (SVT). It includes the refinement of the Test Plan, definition of Test Cases and their implementation into Test Scripts (a test script is a step-by-step instruction enabling the execution of a test case), the execution and evaluation of the tests (by a group of tests called the Test Suite created to exercise a category of tests, such as FVT or SVT), and the corresponding reporting of incidents that are encountered. It also includes the definition and implementation of the Installation Verification Procedures (IVPs).

The Prepare Deployment activity defines the Deployment Plan. Its purpose is to ensure that the system successfully reaches its users. The Deployment Plan provides a detailed schedule of events, persons responsible, and event dependencies required to ensure successful cutover to the new system. The activity also includes the definition of a first draft of User Support Materials and other collateral materials covering the full range of information required by the user to learn, install, operate, use, and maintain the system.

The Plan the Project activity starts with an assessment of the current iteration and a reevaluation of the risks. It refines the Software Development Plan (covering all project phases and iterations) and creates a fine-grained Iteration Plan for the next iteration or iterations. This activity also acquires the necessary resources (including staff) to perform the coming iteration or iterations.

4.4.3 Initial operational capability milestone

At the *Initial Operational Capability Milestone*, the product is ready to be handed over to the Transition Team. All functionality has been developed and all alpha testing (if any) has been completed. In addition to the software, a user manual has been developed, and there is a description of the current release. The evaluation criteria for the Construction Phase involve the answers to these questions:

- Is this product release stable and mature enough to be deployed in the user community?
- Are all the stakeholders ready for the transition into the user community?
- Are the actual resource expenditures compared to the planned resource expenditures still acceptable?

Transition might have to be postponed by one release if the project fails to reach this milestone.

Summary of essential work products and their state at the end of the Construction Phase:

- Glossary (about 100% complete)
- Software Development Plan (about 100% complete)
- Iteration Plans for the transition iterations (about 100% complete, at least for first iteration)
- Risk List (about 75% complete)
- Use-Case Model (about 100% complete)
- Supplementary Specifications (about 100% complete)
- Design Model (about 100% complete)
- Service Model (about 100% complete)
- Test Plan (about 90% complete)
- Test Cases (about 80% complete)
- Test Scripts (about 80% complete)
- Implementation Elements, including source code (about 90% complete)
- Builds are available (one or more per iteration, for instance)
- The executable system is available
- Installation Verification Procedures (IVPs) (about 90% complete)
- User Support Material (about 40% complete)

4.5 Transition Phase overview

The focus of the Transition Phase is to ensure that software is available for its users. The Transition Phase can span several iterations and includes testing the product in preparation for release and making minor adjustments based on user feedback. At this point in the lifecycle, user feedback needs to focus mainly on fine-tuning the product, configuration, installation, and usability issues. All the major structural issues must have been worked out much earlier in the project lifecycle. The Transition Phase consists of a number of iterations culminating in the *Product Release Milestone*.

4.5.1 Transition objectives

The Transition Phase is entered when a baseline is mature enough to be deployed in the user domain. This deployment typically requires that some usable subset of the system has been completed with acceptable quality level and user documentation so that transitioning to the user provides positive results for all parties. The primary objectives of the Transition Phase include:

- Beta testing to validate the new system against user expectations.
- Beta testing and parallel operation relative to a system that it is replacing.
- Converting operational databases.
- Training users and those who will maintain the system.
- Rolling out to the marketing, distribution, and sales forces.
- Deployment-specific engineering, such as cutover, commercial packaging and production, sales rollout, and field personnel training.
- Tuning activities, such as fixing bugs and enhancing for performance and usability.
- Assessing the deployment baselines against the complete vision and the acceptance criteria for the product.
- Achieving user self-supportability.
- Achieving stakeholder concurrence that deployment baselines are complete.
- Achieving stakeholder concurrence that deployment baselines are consistent with the evaluation criteria of the vision.

The Transition Phase ranges from being straightforward to extremely complex, depending on the type of product. A new release of an existing desktop product might be simple, whereas the replacement of a nation's air-traffic control system might be exceedingly complex. Activities performed during an iteration in the Transition Phase depend on the goal. For example, when fixing bugs, implementation and test are usually enough. If, however, new features have to be added, the iteration is similar to one in the Construction Phase requiring analysis and design and so forth.

4.5.2 Typical transition iteration

This section provides an overview of the activities performed in a typical iteration of the Transition Phase, as illustrated in Figure 4-5 on page 48.

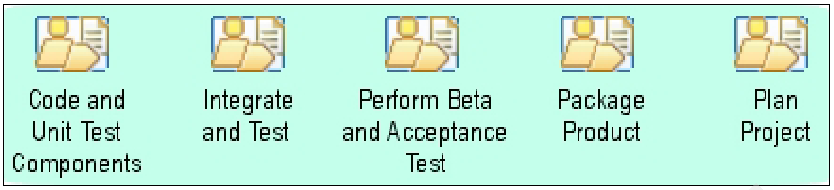


Figure 4-5 Typical Iteration in Transition

In transition, in some cases, it might be necessary to update the system requirements and design. Any significant changes, however, should be deferred to a future generation of the solution to maintain the stability necessary to field capability that is useful to the users and establish a foundation for building future solutions (or decisions made at the Initial Operational Capability Milestone might have to be revisited). Refer to Refine Requirements and Design Components for more information about how to refine the requirements and design if necessary.

The Code and Unit Test Components activity completes all remaining parts of the system implementation so that they can be delivered for integration. This activity implements the elements in the design model by writing source code, adapting existing source code, compiling, linking, and performing unit tests. The code is reviewed to evaluate quality and compliance with the programming guidelines.

The Integrate and Test activity completes the integration and test of the product. The Integrate sub-activity integrates changes from multiple implementers to create a new consistent version of an Implementation Subsystem and integrates implementation subsystems to create a new consistent version of the overall system. The Test sub-activity includes the tasks required for testing within a particular scope, such as System Verification Test (SVT). It includes the refinement of the Test Plan, definition of Test Cases and their implementation into Test Scripts (a test script is a step-by-step instruction enabling the execution of a test case), the execution and evaluation of the tests (by a group of tests called Test Suite created to exercise a category of tests, such as SVT), and the corresponding reporting of incidents that are encountered. It also completes the definition and implementation of the Installation Verification Procedures (IVPs).

The Perform Beta and Acceptance Test activity covers beta and acceptance testing of the product. The Perform Beta Test sub-activity solicits feedback on the product from a subset of the intended users while it is still under active development. A beta test gives the product a controlled, real-world test, so that feedback from potential users can be used to shape the final product. It also provides a preview of the next release to interested clients. The Perform Acceptance Test sub-activity ensures that the product is deemed acceptable to the client prior to its general release.

The Package Product activity builds and packages the product for release. It produces any remaining User Support Material and any artifact needed to effectively deploy the product to its users, such as Training Materials or Release Notes. It also produces a Deployment Unit that enables the software product to be effectively installed and used. The Deployment Unit package consists of a Build (an executable collection of components), documents, such as User Support Material, and Installation Verification Procedures (IVPs). A Deployment Unit is sufficiently complete to be downloaded and run on a node. This definition fits the cases where the product is available over the Internet and the Deployment Unit can be downloaded directly and installed by the user. In the case of “shrinkwrap” software, the Deployment Unit is adorned with distinct packaging consisting of artwork and messaging (Product Artwork) and sold as a Product.

The Plan the Project activity starts with an assessment of the current iteration and a reevaluation of the risks. When more iterations are coming, it refines the fine-grained Iteration Plan for the next iteration or iterations. During the last project iteration, a final Status Assessment is prepared for the Project Acceptance Review, which, if successful, marks the point at which the client formally accepts ownership of the software product. The project manager then completes the closeout of the project by disposing of the remaining assets and reassigning the remaining staff.

4.5.3 Product release milestone

At the end of the Transition Phase is the fourth important project milestone, the *Product Release Milestone*. At this point, you decide if the objectives were met, and if you should start another development cycle. The Product Release Milestone is the result of the client reviewing and accepting the project deliverables. The primary evaluation criteria for the Transition Phase involve the answers to these questions:

- Is the user satisfied?
- Are the actual resource expenditures compared to planned expenditures acceptable?

At the Product Release Milestone, the product is in production and the post-release maintenance cycle begins. This can involve starting a new cycle or an additional maintenance release.

Summary of essential work products completed during the Transition Phase:

- Risk List
- Test Plan
- Test Cases
- Test Scripts
- Implementation Elements, including source code
- Deployment Unit
- Build
- User Support Material
- Installation Verification Procedures (IVPs)
- Product

By the end of the Transition Phase, the project should be in a position to be closed out. In some cases, the end of the current lifecycle might coincide with the start of another lifecycle on the same product, leading to the next generation or version of the product. For other projects, the end of the Transition Phase might coincide with a complete delivery of the artifacts to a third party, who might be responsible for operations, maintenance, and enhancements of the delivered system.

4.6 Note on maintenance projects

The RUP for System z roadmap presented in this chapter addresses green field development and system evolution with architectural changes (including turning an existing capability into a Web service, for instance) or significant impact on existing user business processes. Pure maintenance is out of the scope of this book.

The main characteristics of a maintenance product cycle compared to a regular development product cycle are:

- The Inception and Elaboration phases are merged into a single iteration called Inception/Elaboration.
- There are no architectural changes in the product cycle, or the changes have a certifiably trivial impact on existing design and user business processes.
- The process is driven by change requests rather than by requirements or new product scope.
- Attention is paid to re-factoring code, design, and requirements to reduce long-term increases in system complexity. This is known as *making perfective changes*.
- The product scope is not changed or increased.
- The product cycle has the same business drivers as the previous product cycle.
- The lifecycle is informal, particularly in regard to project artifacts.

For more information about maintenance, refer to the RUP for Maintenance Projects plug-in at:

http://www.ibm.com/developerworks/rational/downloads/06/plugins/rmc_prj_mnt/

The RUP for Maintenance Projects plug-in provides a delivery process, tasks, and guidance for understanding the purpose of a maintenance development cycle, avoiding pitfalls during a maintenance cycle, and successfully delivering a product with higher quality than the previous release.

Part 3

The IBM Rational Unified Process for System z for Advanced Practitioners

This part includes material handy to implement the IBM Rational Unified Process for System z (RUP for System z) on your project. This part is targeted toward advanced practitioners.

5

Process essentials

This chapter provides the process essentials: A brief definition of each project phase (inception, elaboration, construction, and transition) in terms of main goals, activities, and milestones. For each activity, the chapter lists the corresponding key roles, tasks, output work products, and available examples from the Catalog Manager Case Study. The corresponding section of the RUP for System z Web site provides advanced System z practitioners with all of the links (underlined terms) necessary to perform specific activities or tasks.

5.1 Inception essentials

The primary goal of the Inception Phase is to achieve concurrence among all stakeholders on the project scope and to ensure that the project is both worth doing and possible to do. The Inception Phase consists of a number of iterations culminating in the Lifecycle Objectives Milestone. A typical inception iteration includes the activities presented in Table 5-1. The milestone is described right after the table.

Table 5-1 Activities of a typical inception iteration

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Conceive New Project	- Project Manager - Management Reviewer	- Develop Business Case - Identify and Assess Risks - Initiate Project - Project Approval Review	- Business Case - Software Development Plan - Risk List - Review Record	- Business Case
Prepare Project Environment	- Process Engineer - Tool Specialist - Configuration Manager	- Tailor the Development Process for the Project - Select and Acquire Tools - Set Up Tools - Set Up Configuration Management (CM) Environment	- Development Process - Tools - Project Repository	None
Define Requirements	- System Analyst - Software Architect - Test Designer	- Develop Vision - Find Actors and Use Cases - Develop Supplementary Specifications - Capture a Common Vocabulary - Prioritize Use Cases - Define Test Approach	- Vision - Use-Case Model - Supplementary Specifications - Glossary - Software Architecture Document - Test Strategy - Test Plan - Test Environment Configuration	- Vision - Use-Case Model - Supplementary Specification - Glossary - Software Architecture Document - Test Plan
Perform Architectural Proof-of-Concept (optional)	- Software Architect	- Architectural Analysis - Construct Architectural Proof-of-Concept - Assess Viability of Architectural Proof-of-Concept	- Software Architecture Document - Analysis Model - Design Model - Deployment Model - Architectural Proof-of-Concept - Review Record	None

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Plan the Project	- Project Manager	- Assess Iteration - Identify and Assess Risks - Plan Phases and Iterations - Develop Iteration Plan - Acquire Staff	- Iteration Assessment - Risk List - Software Development Plan - Iteration Plan	- Risk List - Software Development Plan - E1 Iteration Plan

Lifecycle Objectives Milestone

At the end of the Inception Phase, the project is evaluated against the following criteria:

- Stakeholder concurrence on the scope definition.
- Agreement that the right set of requirements has been captured.
- Agreement that the cost/schedule estimates, priorities, risks, and development process are appropriate.
- All risks have been identified and a mitigation strategy exists for each risk.

The state of several essential work products at the Inception Phase milestone are:

- Business Case (100% complete)
- Vision (about 100% complete)
- Glossary (about 40% complete)
- Software Development Plan (about 80% complete)
- Iteration Plan for the first elaboration iteration (about 100% complete)
- Risk List (about 25% complete)
- Use-Case Model (about 20% complete)
- Supplementary Specifications (about 20% complete)
- Test Plan (about 10% complete)
- Software Architecture Document (about 10% complete)
- Architectural Proof-of-Concept (one or more proof of concept prototypes available to address very specific risks)

5.2 Elaboration essentials

The primary goal of the Elaboration Phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the Construction Phase. The stability of the architecture is evaluated through one or more architectural prototypes. The Elaboration Phase consists of a number of iterations culminating in the Lifecycle Architecture Milestone. A typical elaboration iteration includes the activities presented in Table 5-2 on page 56. The milestone is described right after the table.

Table 5-2 Activities of a typical elaboration iteration

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Refine Requirements	- Requirements Specifier - System Analyst - Software Architect	- Detail a Use Case - Develop Supplementary Specifications - Capture a Common Vocabulary - Prioritize Use Cases	- Use Case - Supplementary Specifications - Glossary - Software Architecture Document	- Use-Case Specifications - Unified Modeling Language (UML) Models - Supplementary Specification
Define Architecture	- Software Architect - Designer - Technical Reviewer	- Architectural Analysis - Service Analysis - Existing Asset Analysis - Use-Case Analysis - Identify Design Elements - Describe the Run-time Architecture - Describe Distribution - Review the Architecture	- Software Architecture Document - Analysis Model - Design Model - Design Class - Design Subsystem - Design Package - Interface - Module - Signal - Event - Deployment Model - Service Model - Service Component - Review Record	- Software Architecture Document - Unified Modeling Language (UML) Models
Design Components	- Designer - Software Architect - Database Designer - User-Interface Designer - Technical Reviewer	- Use-Case Design - Identify Design Elements - Subsystem Design - Module Design - Class Design - Subsystem Design (SOA) - Component Specification (SOA) - Database Design - Design the User Interface - Prototype the User-Interface - Review the Design	- Design Model - Design Class - Design Subsystem - Design Package - Interface - Module - Signal - Event - Service Model - Service Component - Data Model - Navigation Map - User-Interface Prototype - Review Record	- Unified Modeling Language (UML) Models

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Code and Unit Test Components	- Implementer - Technical Reviewer	- Implement Design Elements - Implement Developer Test - Execute Developer Tests - Review Code	- Implementation Subsystem - Implementation Element - Developer Test - Test Log - Review Record	None
Integrate and Test	- Integrator - Test Designer - Test Analyst - Tester	- Integrate Subsystem - Integrate System - Define Test Approach - Define Test Details - Implement Test - Define Installation Verification Procedures (IVPs) - Implement Installation Verification Procedures (IVPs) - Execute Test Suite - Analyze Test Failure	- Build - Implementation Subsystem - Test Strategy - Test Plan - Test Environment Configuration - Test Case - Test Script - Test Log - Installation Verification Procedures (IVPs) - Change Request	- Test Plan - Test Cases - Installation Verification Procedures (IVPs)
Plan the Project	- Project Manager	- Assess Iteration - Identify and Assess Risks - Plan Phases and Iterations - Develop Iteration Plan - Acquire Staff	- Iteration Assessment - Risk List - Software Development Plan - Iteration Plan	- Risk List - Software Development Plan

Lifecycle Architecture Milestone

At the end of the Elaboration Phase, the project is evaluated against the following criteria:

- The product requirements and architecture are stable.
- The key approaches to be used in test and evaluation are proven.
- Test and evaluation of executable prototypes have demonstrated that the major risk elements have been addressed and have been credibly resolved.
- The iteration plans for the Construction Phase are of sufficient detail to allow the work to proceed and are supported by credible estimates.
- All stakeholders agree that the vision can be met if the current plan is executed to develop the complete system in the context of the current architecture.
- Actual resource expenditures compared to planned expenditures are acceptable.

State of several essential work products at the Elaboration Phase milestone:

- Glossary (about 80% complete)
- Software Development Plan (about 95% complete)
- Iteration Plans for the construction iterations (about 100% complete, at least for the first iteration)
- Risk List (about 50% complete)
- Use-Case Model (about 80% complete)
- Supplementary Specifications (about 80% complete)
- Software Architecture Document (about 100% complete)
- Design Model (about 60% complete)
- Service Model (about 60% complete)
- Test Plan (about 30% complete)
- Test Cases (about 40% complete)
- Test Scripts (about 40% complete)
- Implementation Elements, including source code (about 40% complete)
- Builds are available (one or more per iteration, for instance)
- One or more executable architectural prototypes are available (to explore critical functionality and architecturally significant scenarios)
- Installation Verification Procedures (IVPs) (about 80% complete)

5.3 Construction essentials

The main goal of the Construction Phase is to complete the development of the system based upon the baseline architecture. The Construction Phase consists of a number of iterations culminating in the Initial Operational Capability Milestone. A typical construction iteration includes the activities presented in Table 5-3. The milestone is described right after the table.

Table 5-3 Activities of a typical construction iteration

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Refine Requirements	- Requirements Specifier - System Analyst - Software Architect	- Detail a Use Case - Develop Supplementary Specifications - Capture a Common Vocabulary - Prioritize Use Cases	- Use Case - Supplementary Specifications - Glossary - Software Architecture Document	- Use-Case Specifications - Unified Modeling Language (UML) Models - Supplementary Specification

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Design Components	- Designer - Software Architect - Database Designer - User-Interface Designer - Technical Reviewer	- Use-Case Design - Identify Design Elements - Subsystem Design - Module Design - Class Design - Subsystem Design (SOA) - Component Specification (SOA) - Database Design - Design the User Interface - Prototype the User Interface - Review the Design	- Design Model - Design Class - Design Subsystem - Design Package - Interface - Module - Signal - Event - Service Model - Service Component - Data Model - Navigation Map - User Interface Prototype - Review Record	- Unified Modeling Language (UML) Models
Code and Unit Test Components	- Implementer - Technical Reviewer	- Implement Design Elements - Implement Developer Test - Execute Developer Tests - Review Code	- Implementation Subsystem - Implementation Element - Developer Test - Test Log - Review Record	- Source Code
Integrate and Test	- Integrator - Test Designer - Test Analyst - Tester	- Integrate Subsystem - Integrate System - Define Test Approach - Define Test Details - Implement Test - Define Installation Verification Procedures (IVPs) - Implement Installation Verification Procedures (IVPs) - Execute Test Suite - Analyze Test Failure	- Build - Implementation Subsystem - Test Strategy - Test Plan - Test Environment Configuration - Test Case - Test Script - Test Log - Installation Verification Procedures (IVPs) - Change Request	- Test Plan - Test Cases

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Prepare Deployment	- Deployment Manager - Technical Writer - Implementer - Course Developer - Graphic Artist	- Develop Deployment Plan - Define Bill of Materials - Develop Support Materials - Develop Installation Work Products - Develop Training Materials - Create Product Artwork	- Deployment Plan - Bill of Materials - User Support Material - Installation Artifacts - Training Materials - Product Artwork	None
Plan the Project	- Project Manager	- Assess Iteration - Identify and Assess Risks - Plan Phases and Iterations - Develop Iteration Plan - Acquire Staff	- Iteration Assessment - Risk List - Software Development Plan - Iteration Plan	- Risk List - Software Development Plan

Initial Operational Capability Milestone

The evaluation criteria for the Construction Phase involve the answers to these questions:

- Is this product release stable and mature enough to be deployed in the user community?
- Are all the stakeholders ready for the transition into the user community?
- Are actual resource expenditures compared to planned resource expenditures still acceptable?

The state of several essential work products at the Construction Phase milestone are:

- Glossary (about 100% complete)
- Software Development Plan (about 100% complete)
- Iteration Plans for the transition iterations (about 100% complete, at least for first iteration)
- Risk List (about 75% complete)
- Use-Case Model (about 100% complete)
- Supplementary Specifications (about 100% complete)
- Design Model (about 100% complete)
- Service Model (about 100% complete)
- Test Plan (about 90% complete)
- Test Cases (about 80% complete)
- Test Scripts (about 80% complete)
- Implementation Elements, including source code (about 90% complete)
- Builds are available (one or more per iteration, for instance)

- The executable system is available
- Installation Verification Procedures (IVPs) (about 90% complete)
- User Support Material (about 40% complete)

5.4 Transition essentials

The primary goal of the Transition Phase is to ensure that software is available for its users. It includes testing the product in preparation for release, making minor adjustments based on user feedback, and focusing mainly on fine-tuning the product, configuration, installation, and usability issues. The Transition Phase consists of a number of iterations culminating in the Product Release Milestone. A typical transition iteration includes the activities presented in Table 5-4. The milestone is described right after the table.

Table 5-4 Activities of a typical transition iteration

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Code and Unit Test Components	- Implementer - Technical Reviewer	- Implement Design Elements - Implement Developer Tests - Execute Developer Tests - Review Code	- Implementation Subsystem - Implementation Element - Developer Test - Test Log - Review Record	- Source Code
Integrate and Test	- Integrator - Test Designer - Test Analyst - Tester	- Integrate Subsystem - Integrate System - Define Test Approach - Define Test Details - Implement Test - Define Installation Verification Procedures (IVPs) - Implement Installation Verification Procedures (IVPs) - Execute Test Suite - Analyze Test Failure	- Build - Implementation Subsystem - Test Strategy - Test Plan - Test Environment Configuration - Test Case - Test Script - Test Log - Installation Verification Procedures (IVPs) - Change Request	- Test Plan - Test Cases - Installation Verification Procedures (IVPs)

Activities	Roles	Tasks	Output work products	Catalog Manager examples
Perform Beta and Acceptance Test	- Deployment Manager - Project Manager - Tester	- Develop Product Acceptance Plan - Execute Test Suite - Analyze Test Failure - Manage Beta Test - Manage Acceptance Test	- Product Acceptance Plan - Test Log - Change Request	None
Package Product	- Deployment Manager - Technical Writer - Implementer - Course Developer - Graphic Artist - Configuration Manager	- Write Release Notes - Define Bill of Materials - Develop Support Materials - Develop Installation Work Products - Develop Training Materials - Create Product Artwork - Create Deployment Unit - Release to Manufacturing - Verify Manufactured Product - Provide Access to Download Site	- Release Notes - Bill of Materials - User Support Material - Installation Artifacts - Training Materials - Product Artwork - Deployment Unit - Product	None
Plan the Project	- Project Manager - Management Reviewer	- Assess Iteration - Identify and Assess Risks - Plan Phases and Iterations - Develop Iteration Plan - Acquire Staff - Prepare for Project Closeout - Project Acceptance Review	- Iteration Assessment - Risk List - Software Development Plan - Iteration Plan - Issues List - Status Assessment - Review Record	- Risk List

Product release milestone

The primary evaluation criteria for the Transition Phase involve the answers to these questions:

- Is the user satisfied?
- Are actual resource expenditures compared to planned resource expenditures acceptable?

At the Product Release Milestone, the product is in production and the post-release maintenance cycle begins.

Summary of several essential work products completed at the Transition Phase milestone:

- Risk List
- Test Plan
- Test Cases
- Test Scripts
- Implementation Elements, including source code
- Deployment Unit
- Build
- User Support Material
- Installation Verification Procedures (IVPs)
- Product

6

End-to-end lifecycle

The IBM Rational Unified Process for System z (RUP for System z) includes a delivery process that covers the whole development lifecycle from beginning to end. This delivery process can be used as a template for planning and running a project. It provides a complete lifecycle model with predefined phases, iterations, activities, and tasks. It includes a Work Breakdown Structure.

The RUP for System z delivery process is available on the RUP for System z Web site, as illustrated in Figure 6-1 on page 66. Refer to the RUP for System z Web site for a detailed presentation of the delivery process and to Chapter 10, “IBM RUP for System z Work Breakdown Structure” on page 201 for a presentation of the Work Breakdown Structure.

The RUP for System z Web site can be generated out of the RUP for System z RMC plug-in from IBM developerWorks at:

http://www.ibm.com/developerworks/rational/downloads/06/rmc_plugin7_1/

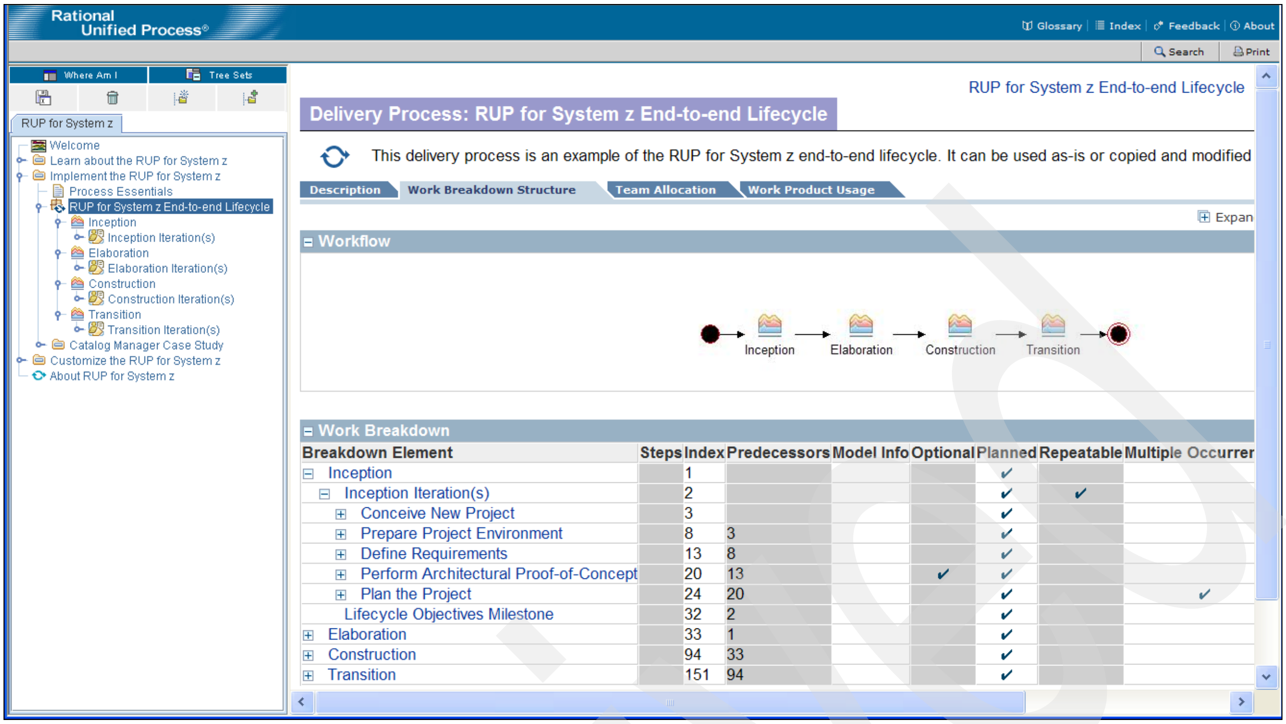


Figure 6-1 The RUP for System z End-to-end Lifecycle

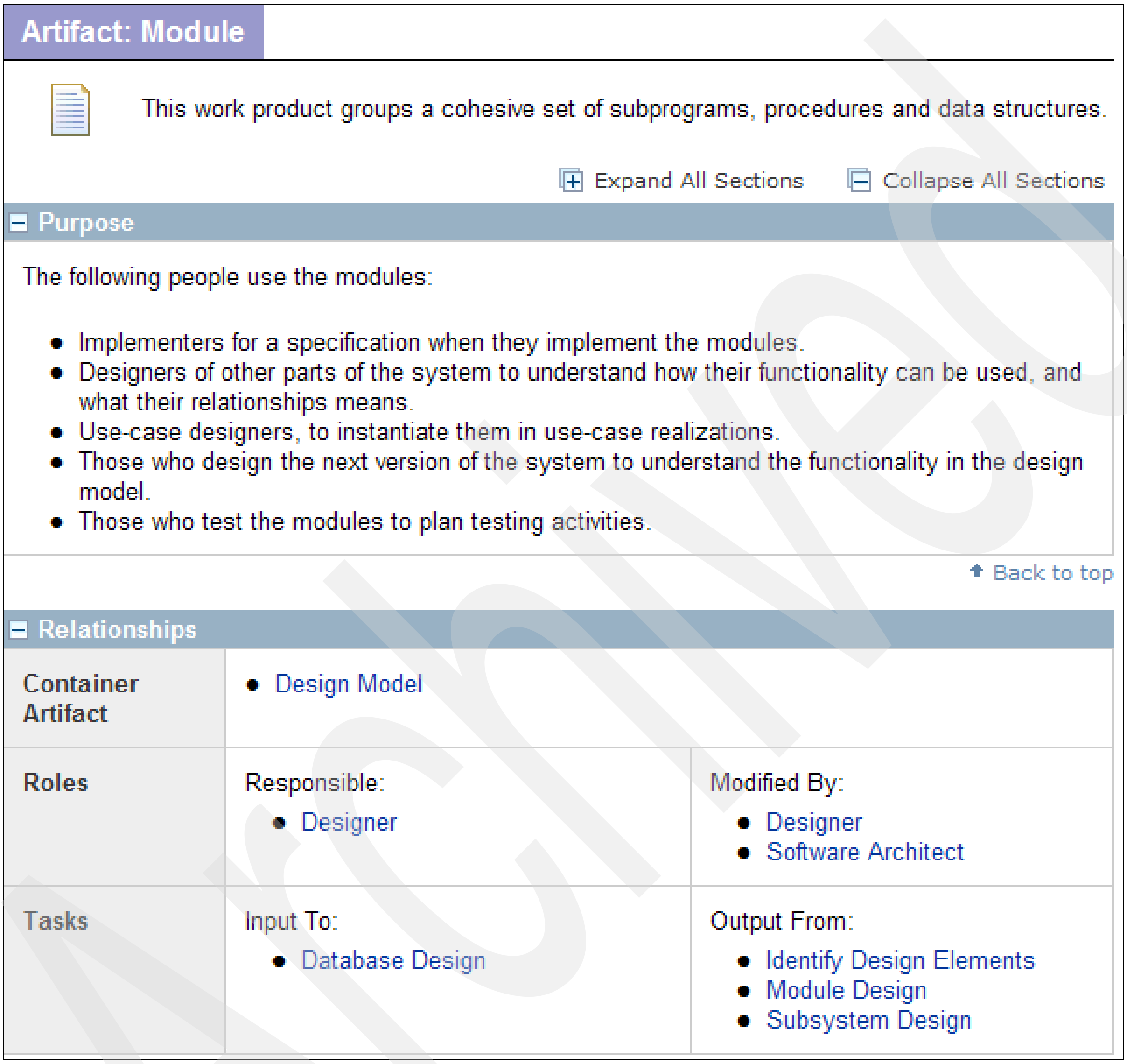
7

Content elements

The IBM Rational Unified Process for System z (RUP for System z) includes a large number of content elements (roles, tasks, and artifacts) that form the core of the method. Most of these elements come from the Rational Unified Process (RUP) and its Service-Oriented Architecture (SOA) extension. However, several content elements have been added to RUP for System z, because they are specific to the System z environment. This chapter presents these new content elements.

7.1 Artifact: Module

Figure 7-1, Figure 7-2 on page 69, and Figure 7-3 on page 70 show the Module artifact as defined in the RUP for System z Web site.



Artifact: Module

This work product groups a cohesive set of subprograms, procedures and data structures.

Expand All Sections   Collapse All Sections

Purpose

The following people use the modules:

- Implementers for a specification when they implement the modules.
- Designers of other parts of the system to understand how their functionality can be used, and what their relationships means.
- Use-case designers, to instantiate them in use-case realizations.
- Those who design the next version of the system to understand the functionality in the design model.
- Those who test the modules to plan testing activities.

Back to top

Relationships

Container Artifact	• Design Model	
Roles	Responsible: • Designer	Modified By: • Designer • Software Architect
Tasks	Input To: • Database Design	Output From: • Identify Design Elements • Module Design • Subsystem Design

Figure 7-1 Module Artifact (Purpose and Relationships)

⊟ Description

Main Description

A module is a software element that groups a cohesive set of subprograms, procedures and data structures. Modules are separate software reusable units that can be authored, edited and compiled separately and simultaneously. Modules also promote modularity and encapsulation (i.e. information hiding), both of which can make complex programs easier to understand.

Modules provide a separation between specification, realization and implementation. A module specification expresses the elements that are provided and required by the module realization first and implementation later. The elements defined during specification for the interface are visible to other modules. The implementation contains the working code that corresponds to the elements declared in the specification and realization.

In the object oriented world we talk about classes. Classes are a category of objects. A class defines the common properties and the common behaviors of the different objects that belong to it. In other words, a class is a description of a set of objects that share the same attributes, operations, methods, relationships and semantics. Typically in UML you have a native representation for classes (the class element) and you are able to specify them providing information like name, attributes, methods, relationships and some other general descriptions. Since a lot of these specifications are common between classes and modules and since a class can also be described through a source code module (i.e. MyClass.java), you can leverage the UML class element in order to model your modules. However you need to be careful about their differences.

The differences between classes and modules are:

- Classes can inherit properties and behavior from another class.
- Classes are instantiated to create objects.
- Polymorphism allows dynamic relationships (class instances can change at run-time) while relations between modules are static.

The similarities between classes and modules are:

- Modules and classes can form a hierarchy.
- Modules and classes can leverage encapsulation to hide implementation details.

So, as described above, you may represent a module through a UML class element, but in order to avoid misunderstanding between classes and modules you also have to provide specific instructions for the stakeholder who needs to read your model. The suggested approach is to leverage the stereotypes technique that enables you to extend the UML language providing the necessary instructions for the readers; for example, you might want to use the stereotype <<Module>> to specify that the model element in your diagram is representing a module instead of a class.

Figure 7-2 Module Artifact (Description)

Tailoring

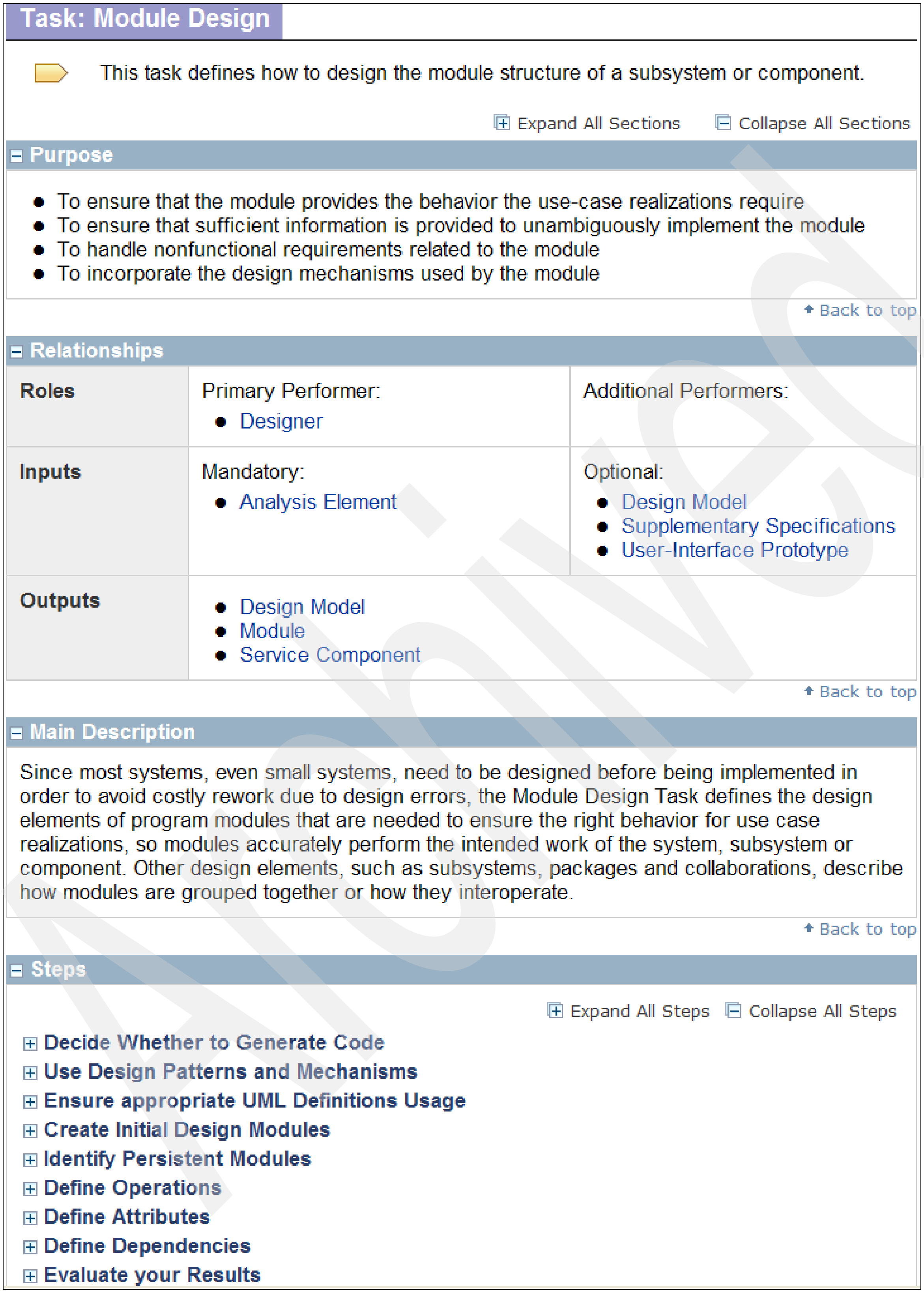
Representation Options	**UML Representation: Stereotyped Class <<Module>>** A module may have the following properties:

Property Name	Brief Description	UML Representation
Name	The name of the module.	The attribute "Name" on model element.
Brief Description	Brief description of the role and purpose of the module.	Tagged value, of type "short text".
Relationships	The relationships, such as associations and dependencies, in which the module participates.	Owned by an enclosing package.
Operations	The operations defined by the module.	The operation compartment.
Attributes	The attributes defined by the module.	The attributes compartment.
Special Requirements	A textual description that collects all requirements, such as non-functional requirements, that are not considered in the design model, but that need to be taken care of during implementation.	Tagged value, of type "short text".
Diagrams	Any diagrams local to the module, such as interaction diagrams, class diagrams, or state chart diagrams.	Owned by an enclosing package.

Figure 7-3 Module Artifact (Tailoring)

7.2 Task: Module Design

Figure 7-4 on page 71 shows the Module Design task as defined in the RUP for System z Web site.



Task: Module Design

This task defines how to design the module structure of a subsystem or component.

Expand All Sections   Collapse All Sections

Purpose

- To ensure that the module provides the behavior the use-case realizations require
- To ensure that sufficient information is provided to unambiguously implement the module
- To handle nonfunctional requirements related to the module
- To incorporate the design mechanisms used by the module

Back to top

Relationships

Roles	Primary Performer: • Designer	Additional Performers:
Inputs	Mandatory: • Analysis Element	Optional: • Design Model • Supplementary Specifications • User-Interface Prototype
Outputs	• Design Model • Module • Service Component	

Back to top

Main Description

Since most systems, even small systems, need to be designed before being implemented in order to avoid costly rework due to design errors, the Module Design Task defines the design elements of program modules that are needed to ensure the right behavior for use case realizations, so modules accurately perform the intended work of the system, subsystem or component. Other design elements, such as subsystems, packages and collaborations, describe how modules are grouped together or how they interoperate.

Back to top

Steps

Expand All Steps   Collapse All Steps

- **Decide Whether to Generate Code**
- **Use Design Patterns and Mechanisms**
- **Ensure appropriate UML Definitions Usage**
- **Create Initial Design Modules**
- **Identify Persistent Modules**
- **Define Operations**
- **Define Attributes**
- **Define Dependencies**
- **Evaluate your Results**

Figure 7-4 Module Design Task

Figure 7-5 to Figure 7-19 on page 82 show the Module Design task steps as defined in the RUP for System z Web site.

⊟ Decide Whether to Generate Code

The way you do design differs depending on whether you generate code from the design model or not. If you generate code, the design needs to be very detailed and should be synchronized with the code. On the other hand, if code is not generated then there is no need for a detail design model. So you need to pay attention to the different levels of abstraction of your models and to the modeling tools adopted. According to the adopted modeling tools you can run transformations that apply a set of rules to UML models to transform them into code or text in general. For example, after your design model is good enough and depending on the modeling tools used, you can generate application code from the UML model by running a transformation against the source UML model. When you realize that it's possible to gain value from development processes that leverage transformations usage, you may consider developing your own transformations that apply to your paradigm. Furthermore, you could use transformations to easily generate test cases for your modules, starting from a stubbed version of your code, which can be generated, for example via a UML-to-Cobol transformation. Based on this skeleton of the code, you can create the test cases for your test.

In conclusion, transformation usage can accelerate code development providing both quality and control and allows analysts and architects to focus more on business needs and less on code semantics.

Figure 7-5 Module Design Task - Step: Decide Whether to Generate Code

⊟ Use Design Patterns and Mechanisms

A design pattern is a repeatable solution for a recurring problem in software design. The use of design patterns and mechanisms can be a great asset in class, module or capability design. Today, many software development organizations have their own standards and architecture customizations. The use of patterns allows them to enforce those standards, ensuring better productivity and quality.

Figure 7-6 Module Design Task - Step: Use Design Patterns and Mechanisms

Ensure appropriate UML definitions usage

In order to better understand the model elements used in Design Model diagrams, we need to extend the UML language to better support the module concept beyond the embedded class definition. For instance, you may consider using simple stereotypes or fully detailed UML profiles. Both of these mechanisms enable the architect to expressively design the modules that participate in the use case realization or in other realizations included in the design model. The figure below shows an example of <<Module>> stereotype for supporting modules in UML.

Legacy systems typically comprise a number of programs, each fulfilling some system functionality. One or more source files are used to implement each program through a programming language. In using UML to model a legacy system in a System z application, it is important to understand both programming language constructs, and the programming style (i.e., the way the language is used), because both language and program-specific constructs are necessary to abstract the existing programs. As a number of different languages might be used the specifics of each language must be taken into account.

During examination of each language, you identify significant language constructs. This step is fundamental in order to define appropriate UML profiles and stereotypes that allow you to abstract flows in a program. For instance the table below describes the most significant language constructs for Cobol:

Significant Language Construct	Description
COBOL Program	This is the physical unit of System Z that can be modeled
COBOL Working Storage Sections	This represents the information that is manipulated by the program.
COBOL Paragraph	This represents the decomposition of the COBOL program.
COBOL Statements	This represents each step in the flow of the COBOL program and is the next level decomposition of the COBOL paragraph.
COBOL Perform Statements	This describes the flow within the COBOL program by relating different COBOL paragraphs to each other.
COBOL Conditional Statements	This represents decisions described in the program: they are often a physical implementation of some business or technical rules.

Figure 7-7 Module Design Task - Step: Ensure Appropriate UML Definition Usage (Part 1)

Once you have identified the most significant language constructs, a specific focus on language grammar is needed because the formal representation of language constructs typically used in the System Z environment tends to be much more complicated than classical Object Oriented languages like Java or C#. The identification of the most important language constructs allows us to concentrate on important aspects of the grammar that we need to map to stereotyped UML elements. During mapping definition it's important to set a one-to-one relation in order to avoid misunderstanding and confusion. So, a possible suggested mapping from COBOL to UML is summarized in the table below. In this table you can find the list of important grammar elements, the UML mapping and the corresponding stereotype where applicable, and finally a description that makes clear the reason of the mapping itself.

Grammar Element	UML mapping	Description
Program	Component <<program>>	Physical implementation of the program design.
Program	Class <<program>>	This is the design of the program.
Program	Use case	Since a program is written to fulfill some functionality, it can be mapped to a use case
Program	Use case realization	The realization of functional requirements described in a use case is in the program itself.
Record Description	Attribute	If the record description has an associated PIC declaration, it is an attribute
Record Description	Nested class stereotyped <<record>>	If the record description has no associated PIC declaration is a nested class belonging to the program
Paragraph	Private operation	A paragraph is a private collection of steps not callable by external programs
Statement	Activity	A statement is a step within a paragraph
Perform Statement	Reflexive message	It calls another paragraph in the program

Figure 7-8 Module Design Task - Step: Ensure Appropriate UML Definition Usage (Part 2)

The distinction between Record Description attribute and Record Description nested class is illustrated by the following figure:

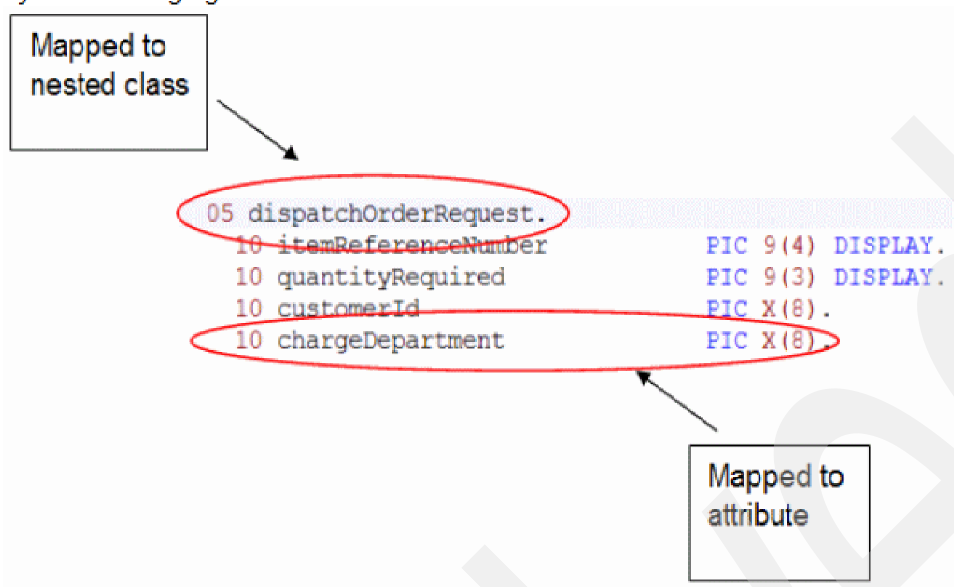
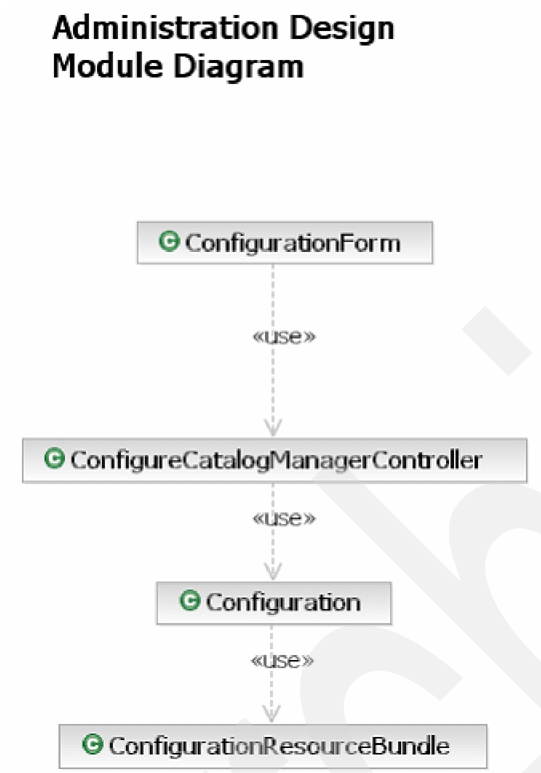


Figure 7-9 Module Design Task - Step: Ensure Appropriate UML Definition Usage (Part 3)

⊟ Create Initial Design Modules

Create one or several initial design modules for the Artifact: Analysis Elements given as input to this task and assign trace dependencies. The design modules are representations of program modules in the overall design model. The initial design modules created in this step will be refined, adjusted, split, or merged in subsequent steps when assigned various design properties-such as operations, methods, subroutines, procedures. Depending on the type of the analysis element (boundary, entity, or control) being designed, there are specific strategies you can use to create initial design classes/modules. The figure below shows an initial design of dependencies between classes/modules.



You need to keep in mind the difference between Classes and Modules. Typically Classes can be instantiated to create objects, can inherit behavior and data from another class. In addition polymorphism allows relationships between class instances to change at run-time, while relations between modules are static. Analysis Elements are used in deriving the Module Design Task. During Module Design Task: Use-Case Analysis specifies elements of an early conceptual model for "things in the system which have responsibilities and behavior". They represent the prototypical classes and/or modules of the system, and are a 'first-pass' at the major abstractions that the system must handle. So, analysis elements give rise to the major abstractions of the system design: the design classes, design modules and subsystems of the system.

Figure 7-10 Module Design Task - Step: Create Initial Design Modules (Part 1)

Designing Boundary Modules

Boundary modules either represent interfaces to users or to other systems.

Typically, boundary modules that represent interfaces to other systems are modeled as subsystems, because they often have complex internal behavior. If the interface behavior is simple (perhaps acting as only a pass-through to an existing API to the external system), you might choose to represent the interface with one or more boundary modules.

Boundary classes/modules that represent interfaces to users generally follow the rule of one boundary module for each window or one for each form or map in the user interface. Consequently the responsibilities of the boundary modules can be on a fairly high-level, and need to be refined and detailed in this step. Additional models or prototypes of the user interface can be another source of input to be considered in this step.

Figure 7-11 Module Design Task - Step: Create Initial Design Modules (Part 2)

Designing Control Modules

A control module is responsible for managing the flow of a use case and, therefore, coordinates most of its actions; control modules encapsulate logic that is not particularly related to user interface issues (boundary modules) or to data engineering issues (entity data). This logic is sometimes called application logic or business logic.

Take the following issues into consideration when control modules are designed:

1. **Complexity** - You can handle uncomplicated controlling or coordinating behavior using boundary or entity modules. As the complexity of the application grows, however, reconsider to extend Control Modules responsibility. If you omit to do that, you might face significant issues such as:
   - The use-case coordinating behavior becomes embedded in the UI, making it more difficult to change the system.
   - The same UI cannot be used in different use-case realizations without difficulty
   - The UI becomes burdened with additional functionality, degrading its performance
   - The entity modules might become burdened with use-case specific behavior, reducing their generality

   To avoid these problems, control modules are introduced to provide behavior related to coordinating flows-of-events.

1. **Change probability** - If the probability of changing flows of events is low or the cost is negligible, the extra expense and complexity of additional control classes/modules might not be justified.
2. **Distribution and performance** - The need to run parts of the application on different nodes, or in different process spaces, introduces the need to specialize design model elements. This specialization is often accomplished by adding control modules and distributing behavior from the boundary and entity classes/modules onto the control classes/modules. In doing this, the boundary classes/modules migrate toward providing purely UI services, the entity classes/modules move toward providing purely data services, and the control classes/modules provide the rest.
3. **Transaction management** - Managing transactions is a classic coordination activity. Without a framework to handle transaction management, one or more transaction manager modules would have to interact to ensure that you maintain the integrity of the transactions.

Figure 7-12 Module Design Task - Step: Create Initial Design Modules (Part 3)

Designing Entity modules

During analysis, entity elements or classes represent manipulated units of information. They are often passive and persistent, and might be identified and associated with the analysis mechanism for persistence. The details of designing a database-based persistence mechanism are covered in Task: Database Design. Performance considerations could force some refactoring of persistent classes and modules, causing changes to the Design Model that is discussed jointly between the Database Designer and the Designer.

A broader discussion of design issues for persistent analysis elements is presented later under the heading Identify Persistent Modules.

Figure 7-13 Module Design Task - Step: Create Initial Design Modules (Part 4)

Identify Persistent Modules

Analysis Elements that need to store their state on a permanent medium are referred to as persistent. The need to store their state might be for permanent recording of module information, for backup in case of system failure, or for exchange of information.

Incorporate design mechanisms corresponding to persistency mechanisms found during analysis. For example, depending on what is required by the class, the analysis mechanism for persistency might be realized by one of these design mechanisms:

- In-memory storage
- Flash card
- Binary file (ex: VSAM file)
- Database Management System (DBMS)

Persistent data might not be derived from entity classes only; persistent data could also be needed to handle nonfunctional requirements in general. Examples are persistent data needed to maintain information relevant to process control or to maintain state information between transactions.

The figure below shows design elements that incorporate design mechanism according to the directions provided by the Analysts in the Analysis Model.

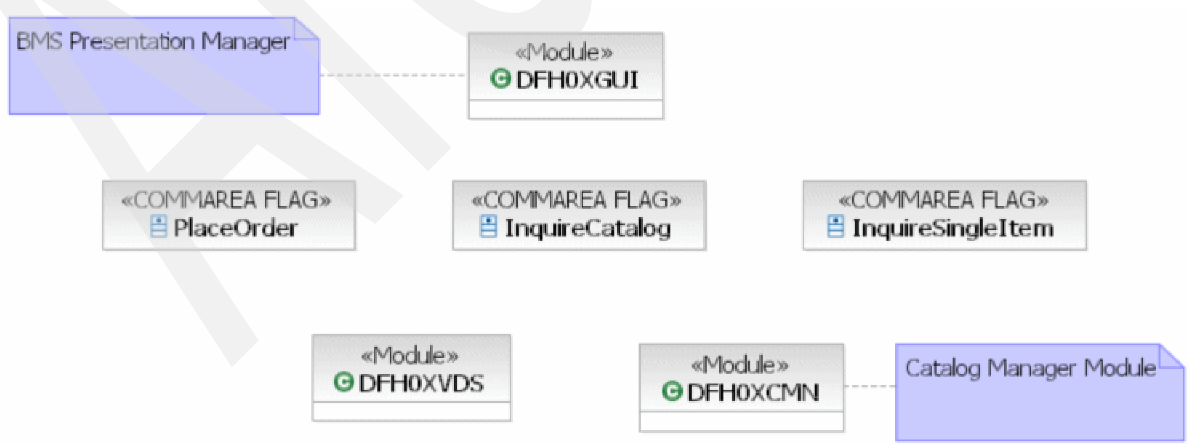


Figure 7-14 Module Design Task - Step: Identify Persistent Modules

Define Operations

Identifying operations

To identify operations on design modules:

- Study the responsibilities of each corresponding analysis class, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
- Study the use-case realizations in the participating classes view to see how the operations are used by the use-case realizations. Extend the operations, one use-case realization at a time, refining the operations, their descriptions, return types, and parameters. Each use-case realization's requirements pertaining to classes are described textually in the Flow of Events of the use-case realization.
- Study the Special Requirements use case to be sure that you do not miss implicit requirements on the operation that might be stated there.

Operations are required to support the messages that appear on sequence diagrams because message specifications that have not yet been assigned to operations, describe the behavior the class is expected to perform. The figure below illustrates an example of a sequence diagram.

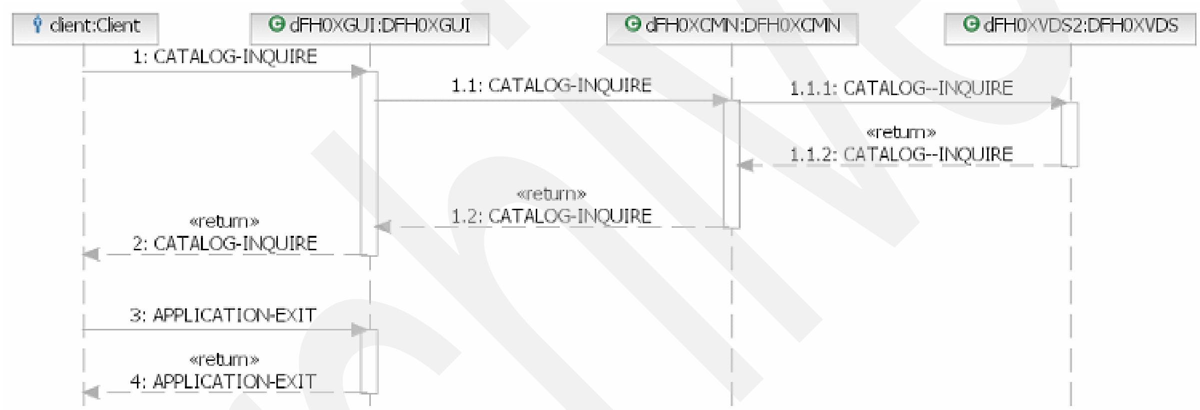


Figure 7-15 Module Design Task - Step: Define Operations (Part 1)

Naming and describing the operations

Use naming conventions for the implementation language when you're naming operations, return types, and parameters and their types. These are usually described in the Project-Specific Guidelines.

For each operation, you should at least define the following:

1. The operation name - keep the name short and descriptive of the result the operation achieves.

   - The names of operations should follow the syntax of the implementation language. Example: PLACE-ORDER() would be acceptable for COBOL; a better name that fits for all languages would be placeOrder().
   - Avoid names that imply how the operation is performed. For example, in module DF0XCMN the COST() operation is better than CALCULATE-COST(), since the latter implies a calculation is performed. The operation might simply return a value in VSAM file or in a database; a better name would be GET-COST().
   - The name of an operation should clearly show its purpose. Avoid unspecific names, such as GET-DATA, that are not descriptive about the result they return. Use a name that shows exactly what is expected, such as GET-ADDRESS. Better yet, simply let the operation name be the name of the property that is returned or set.
   - Operations that are conceptually the same should have the same name even if different Modules define them, if they are implemented in entirely different ways, or if they use a different number of parameters or variables.

3. A short description - As meaningful as you try to make it, the name of the operation is often only vaguely useful when trying to understand what the operation does. Give the operation a short description consisting of a couple of sentences written from the operation user's perspective.

The figure below shows some examples of operation name.

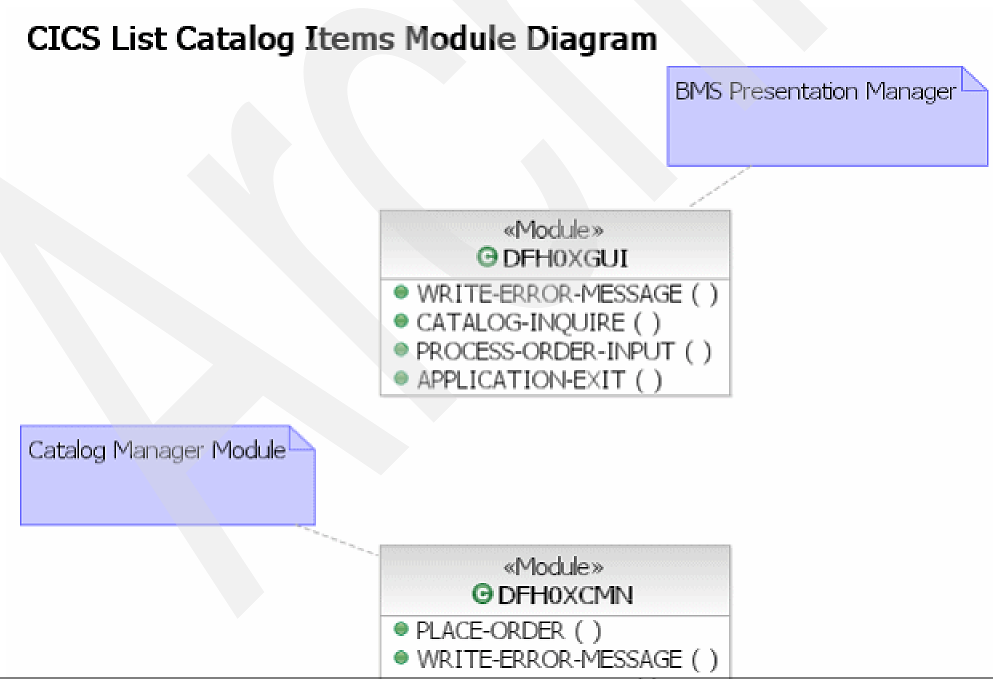


Figure 7-16 Module Design Task - Step: Define Operations (Part 2)

Define Attributes

During the definition of operations, attributes needed by the module to carry out its operations are identified. Attributes provide information storage according to the mapping definition you provided in the earlier step. For each attribute, define:

- its name, which should follow the naming conventions of both the implementation language and the project
- its type, which will be an elementary data type supported by the implementation language
- its default or initial value, to which it is initialized
- its visibility, which will take one of those defined during Ensure appropriate UML definitions usage step
- persistent data, whether the attribute is persistent (the default) or transient.

Check to make sure all attributes are needed. Attributes should be justified-it's easy for attributes to be added early in the process and survive long after they're no longer needed due to shortsightedness.

Figure 7-17 Module Design Task - Step: Define Attributes

Define Dependencies

For each case where the communication between modules is required, ask these questions:

- Is some data or information dependent on other module?
- Is the receiver of some data or information a global one?
- Is the receiver of some data responsible for temporary objects created and destroyed during the operation itself?

If the answer to any of the above questions is yes, establish a dependency between the modules in a class diagram containing the two modules.

The following figure shows some examples of dependencies.

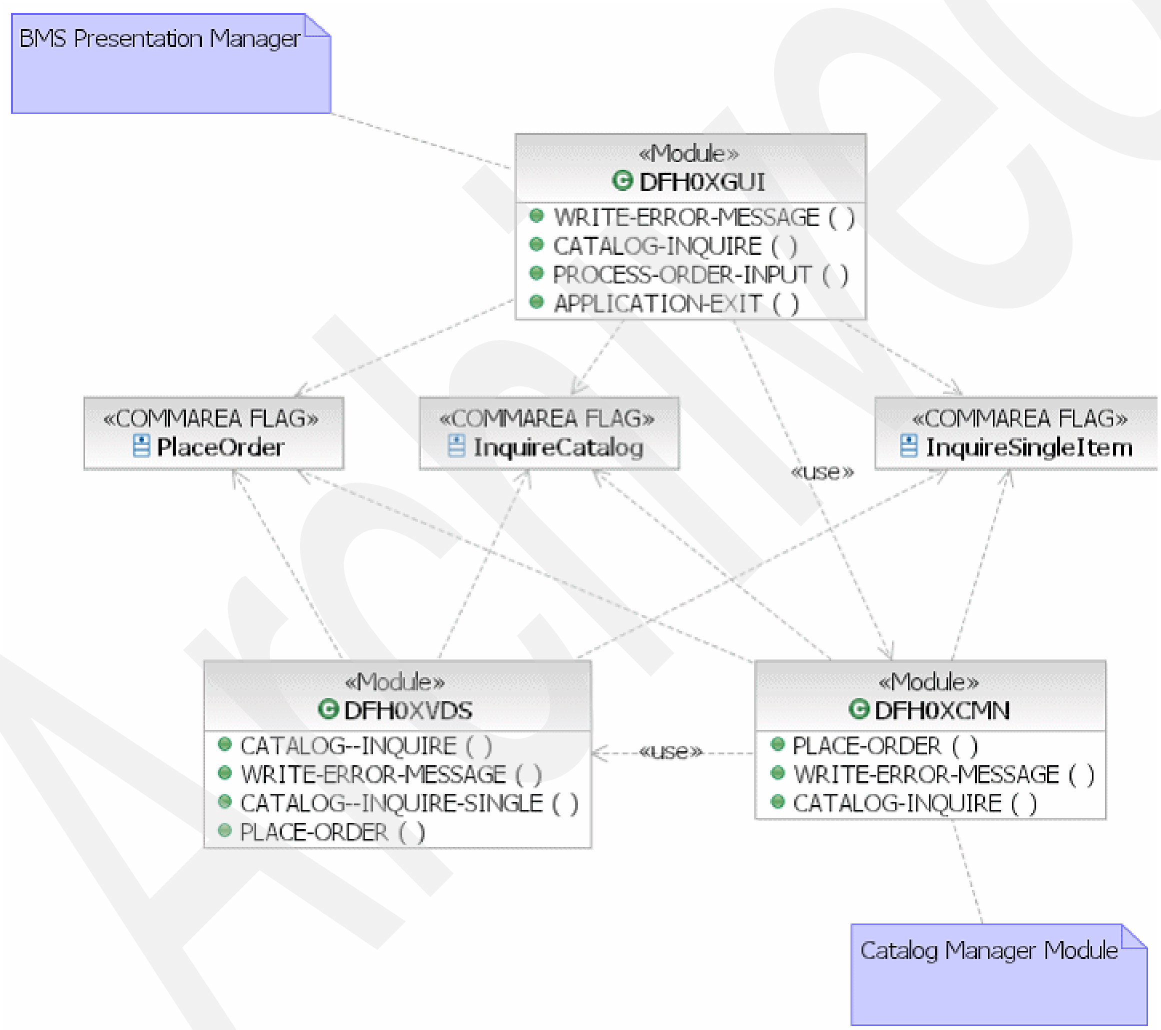


Figure 7-18 Module Design Task - Step: Define Dependencies

Evaluate your Results

Check the design model at this stage to verify that your work is headed in the right direction. There is no need to review the model in detail, but you should consider the Checklist: Design Model.

Figure 7-19 Module Design Task - Step: Evaluate your Results

7.3 Artifact: Installation Verification Procedures (IVPs)

Figure 7-20 on page 84 shows the Installation Verification Procedures (IVPs) artifact as defined in the RUP for System z Web site.

Artifact: Installation Verification Procedures (IVPs)

This work product is one or more programs or scripts (manual or automated) that are run at the end of installation of a software application.

Expand All Sections Collapse All Sections

Purpose

The purpose of the IVPs is to verify that the installed program or application functions correctly in the environment in which it is installed.

The following people use an IVP:

- Implementers who may be integrating a program with other assets and want to ensure that there are no obvious problems.
- Testers who test the software, use it to plan testing activities.
- Users of the program or application who want to ensure that the basic functionality is in place.

Back to top

Relationships

Container Artifact	• Product	
Roles	Responsible: • Tester	Modified By: • Tester
Tasks	Input To: • Implement Installation Verification Procedures (IVPs) • Create Deployment Unit • Define Installation Verification Procedures (IVPs)	Output From: • Define Installation Verification Procedures (IVPs)

Back to top

Illustrations

Examples	• Catalog Manager Installation Verification Procedures (IVPs) - Elaboration Phase • Catalog Manager Installation Verification Procedures (IVPs) - Transition Phase

Back to top

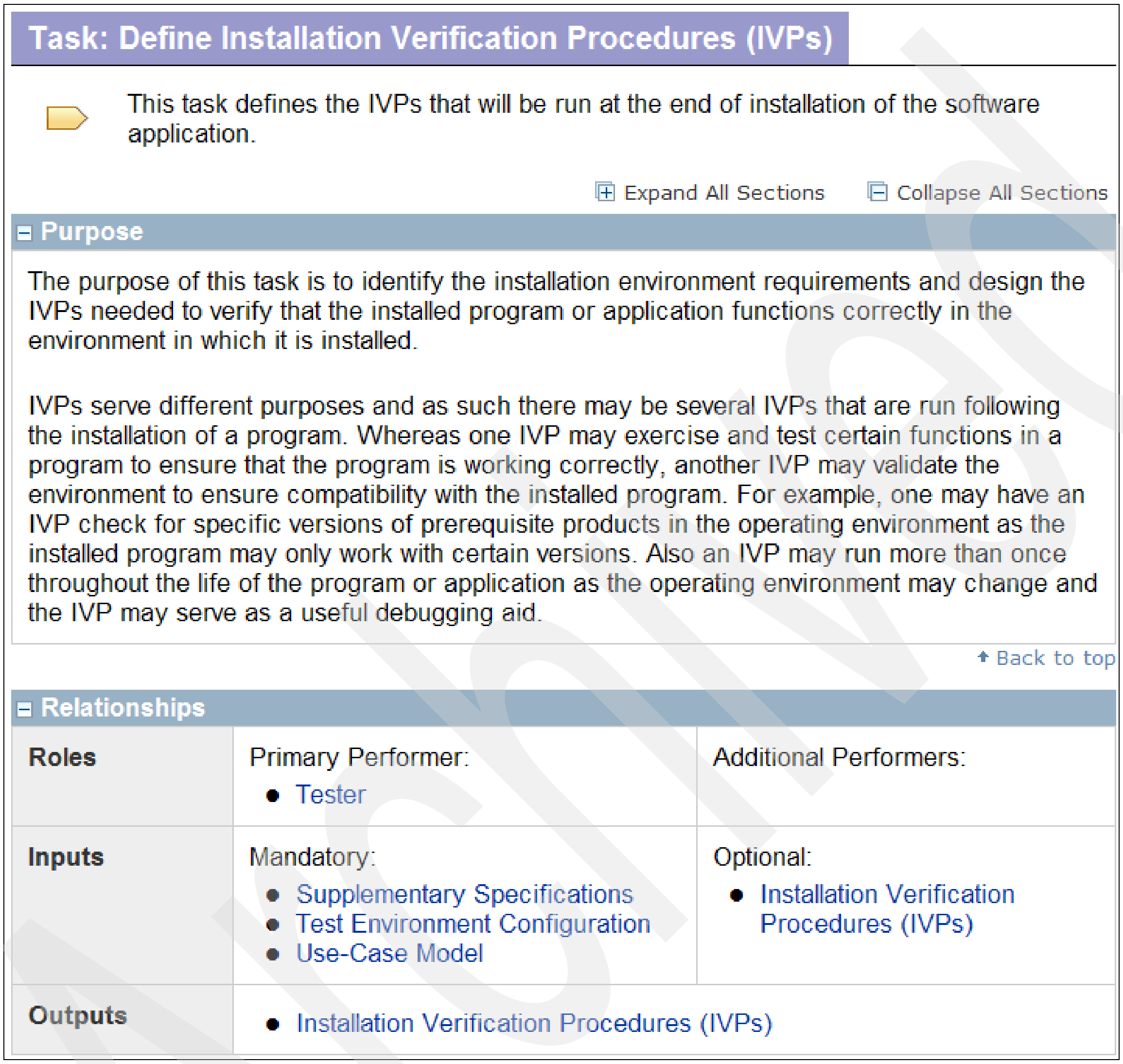
Key Considerations

It is recommended that you use IVPs especially in a complex program or application that may be running in a variety of environments. In the case of a simple application, just exercising the functions of the application may suffice.

Figure 7-20 Installation Verification Procedures (IVPs) Artifact

7.4 Task: Define Installation Verification Procedures (IVPs)

Figure 7-21 and Figure 7-22 on page 86 show the Define Installation Verification Procedures (IVPs) task as defined in the RUP for System z Web site.



Task: Define Installation Verification Procedures (IVPs)

This task defines the IVPs that will be run at the end of installation of the software application.

Expand All Sections   Collapse All Sections

Purpose

The purpose of this task is to identify the installation environment requirements and design the IVPs needed to verify that the installed program or application functions correctly in the environment in which it is installed.

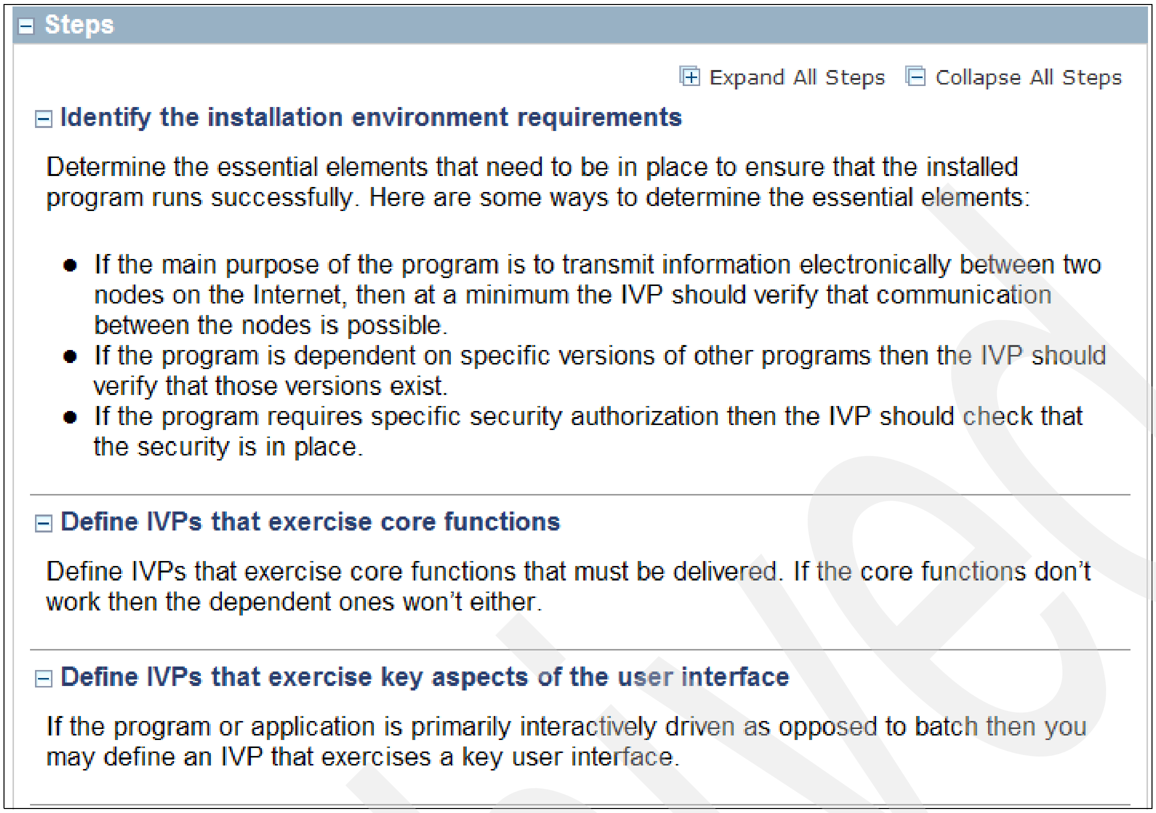
IVPs serve different purposes and as such there may be several IVPs that are run following the installation of a program. Whereas one IVP may exercise and test certain functions in a program to ensure that the program is working correctly, another IVP may validate the environment to ensure compatibility with the installed program. For example, one may have an IVP check for specific versions of prerequisite products in the operating environment as the installed program may only work with certain versions. Also an IVP may run more than once throughout the life of the program or application as the operating environment may change and the IVP may serve as a useful debugging aid.

Back to top

Relationships

Roles	Primary Performer: • Tester	Additional Performers:
Inputs	Mandatory: • Supplementary Specifications • Test Environment Configuration • Use-Case Model	Optional: • Installation Verification Procedures (IVPs)
Outputs	• Installation Verification Procedures (IVPs)	

Figure 7-21 Define Installation Verification Procedures Task (Purpose and Relationships)


Steps

Expand All Steps Collapse All Steps

Identify the installation environment requirements

Determine the essential elements that need to be in place to ensure that the installed program runs successfully. Here are some ways to determine the essential elements:

- If the main purpose of the program is to transmit information electronically between two nodes on the Internet, then at a minimum the IVP should verify that communication between the nodes is possible.
- If the program is dependent on specific versions of other programs then the IVP should verify that those versions exist.
- If the program requires specific security authorization then the IVP should check that the security is in place.

Define IVPs that exercise core functions

Define IVPs that exercise core functions that must be delivered. If the core functions don’t work then the dependent ones won’t either.

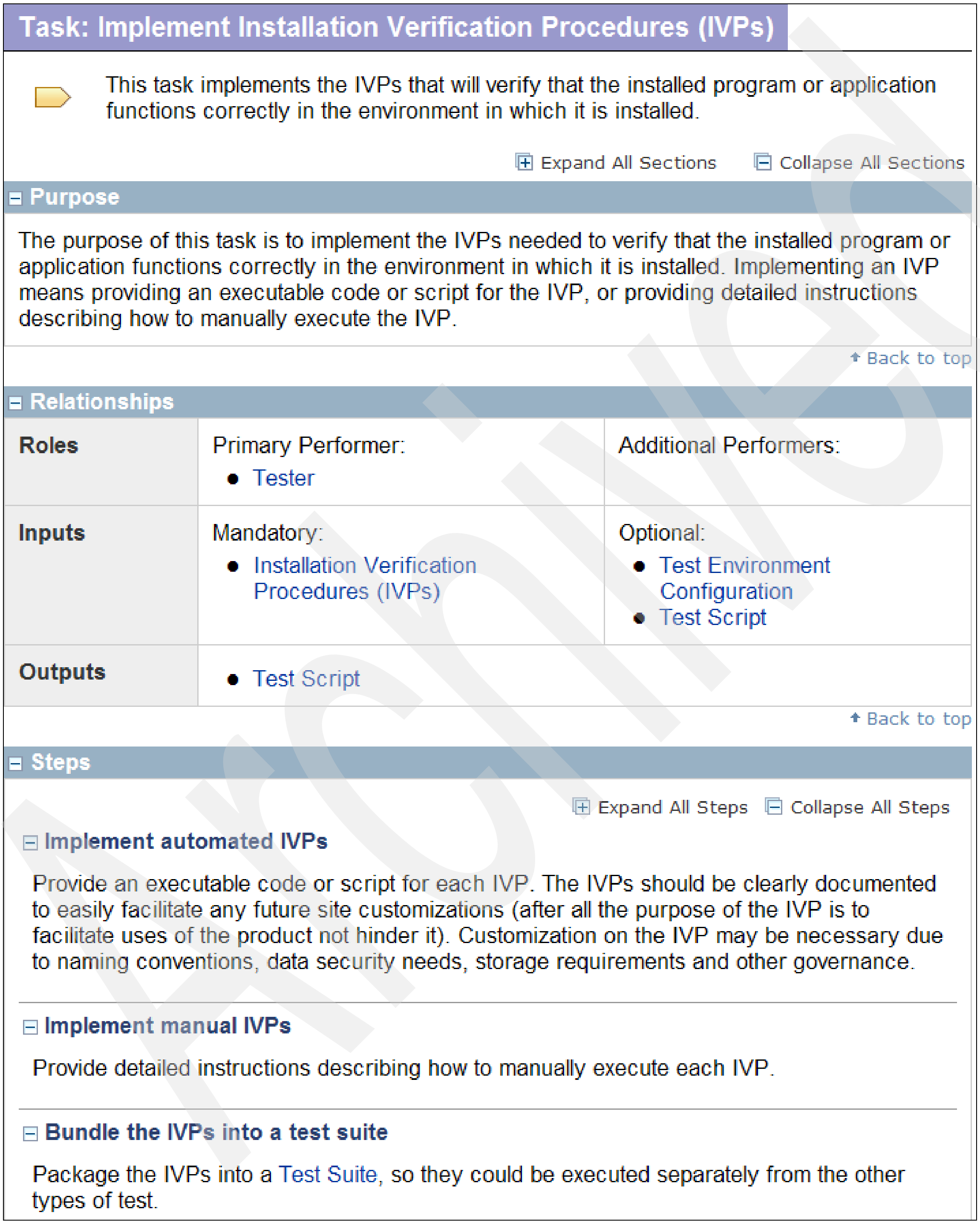
Define IVPs that exercise key aspects of the user interface

If the program or application is primarily interactively driven as opposed to batch then you may define an IVP that exercises a key user interface.

Figure 7-22 Define Installation Verification Procedures Task (Steps)

7.5 Task: Implement Installation Verification Procedures (IVPs)

Figure 7-23 shows the Implement Installation Verification Procedures (IVPs) task as defined in the RUP for System z Web site.



Task: Implement Installation Verification Procedures (IVPs)

This task implements the IVPs that will verify that the installed program or application functions correctly in the environment in which it is installed.

Expand All Sections   Collapse All Sections

Purpose

The purpose of this task is to implement the IVPs needed to verify that the installed program or application functions correctly in the environment in which it is installed. Implementing an IVP means providing an executable code or script for the IVP, or providing detailed instructions describing how to manually execute the IVP.

Back to top

Relationships

Roles	Primary Performer: • Tester	Additional Performers:
Inputs	Mandatory: • Installation Verification Procedures (IVPs)	Optional: • Test Environment Configuration • Test Script
Outputs	• Test Script	

Back to top

Steps

Expand All Steps   Collapse All Steps

Implement automated IVPs

Provide an executable code or script for each IVP. The IVPs should be clearly documented to easily facilitate any future site customizations (after all the purpose of the IVP is to facilitate uses of the product not hinder it). Customization on the IVP may be necessary due to naming conventions, data security needs, storage requirements and other governance.

Implement manual IVPs

Provide detailed instructions describing how to manually execute each IVP.

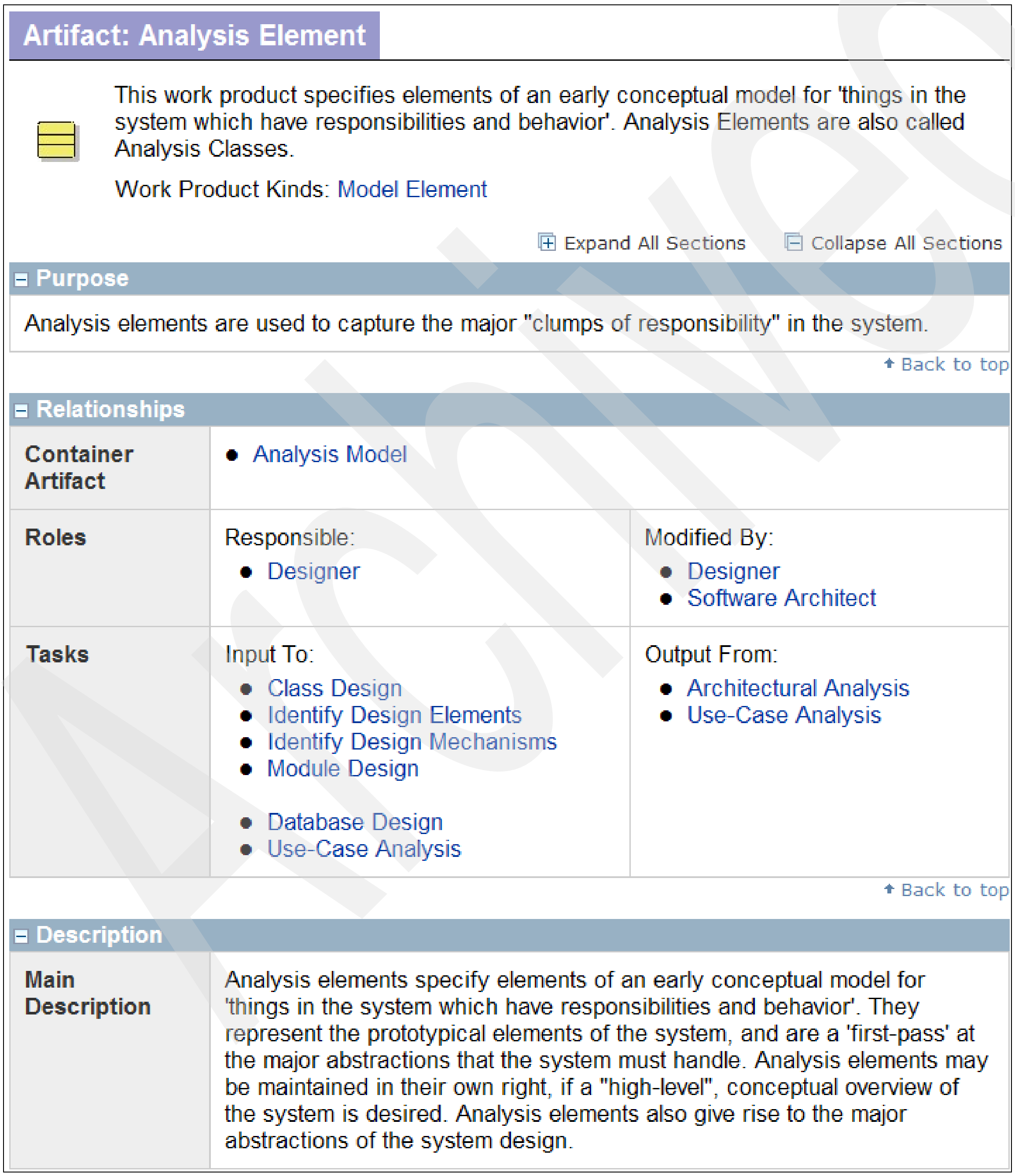
Bundle the IVPs into a test suite

Package the IVPs into a Test Suite, so they could be executed separately from the other types of test.

Figure 7-23 Implement Installation Verification Procedures Task

7.6 Artifact: Analysis Element

RUP analysis and design activities start by identifying conceptual classes, which are called *analysis classes*. Analysis classes specify early conceptual “things” in the system, which have responsibilities and behavior. They are refined later on into detailed design classes or other design elements. In order to generalize this approach to non object-oriented development environments in RUP for System z, *analysis class* is renamed *analysis element* so that an analysis element can be used to identify conceptual things that can be turned later on into modules, classes, or any other design element. Figure 7-24 shows the Analysis Element artifact as defined in the RUP for System z Web site.



Artifact: Analysis Element

This work product specifies elements of an early conceptual model for 'things in the system which have responsibilities and behavior'. Analysis Elements are also called Analysis Classes.

Work Product Kinds: Model Element

Expand All Sections   Collapse All Sections

Purpose

Analysis elements are used to capture the major "clumps of responsibility" in the system.

Back to top

Relationships

Container Artifact	• Analysis Model	
Roles	Responsible: • Designer	Modified By: • Designer • Software Architect
Tasks	Input To: • Class Design • Identify Design Elements • Identify Design Mechanisms • Module Design • Database Design • Use-Case Analysis	Output From: • Architectural Analysis • Use-Case Analysis

Back to top

Description

Main Description	Analysis elements specify elements of an early conceptual model for 'things in the system which have responsibilities and behavior'. They represent the prototypical elements of the system, and are a 'first-pass' at the major abstractions that the system must handle. Analysis elements may be maintained in their own right, if a "high-level", conceptual overview of the system is desired. Analysis elements also give rise to the major abstractions of the system design.

Figure 7-24 Analysis Element Artifact

7.7 Task: Service Analysis

Figure 7-25 and Figure 7-26 on page 90 show the Service Analysis task as defined in the RUP for System z Web site.

Task: Service Analysis

This task identifies the design elements of a service-oriented solution in terms of services and partitions and documents the initial specification of those services.

Expand All Sections   Collapse All Sections

Purpose

- To identify the design elements of a service-oriented solution in terms of services and partitions.
- To document the initial specification of services.
- To determine the initial dependencies and communication between services.

Back to top

Relationships

Roles	Primary Performer: • Software Architect	Additional Performers:
Inputs	Mandatory: • Business Case • Software Architecture Document • Vision	Optional: • Supplementary Specifications • Use-Case Model
Outputs	• Service Model	

Back to top

Main Description

Service Analysis is the process of identifying and validating candidate services, components and flows. These candidate services may yet require additional refinement; however the steps included here provide an effective manner in which to produce an initial set of Artifact: Service Specification.

Figure 7-25 Service Analysis Task (Purpose, Relationships, and Main Description)

⊟ Adopt a Service Identification Approach

Select the Service Identification approach among a number of approaches available to support you in identifying the services that will be part of the solution that you are creating. Approaches are top-down from requirements to design (Task: Business Process Analysis, Task: Business Use Case Analysis (SOA)), data-driven (Task: Data Model Analysis), rule-driven (Task: Business Rule Analysis), bottom-up (Task: Existing Asset Analysis).

Learn and understand these approaches in order to select the most appropriate approach based on the circumstances of your project.

⊟ Identify service partitions

Specify a set of logical partitions that will be used to organize the services that will be a part of the solution that you are creating. A partition represents some logical or physical boundary of the system.

For example partitions could be used to represent the web, business and data tiers of a traditional n-tier application. Partitions might also be used to denote more physical boundaries (such as my primary data center, secondary site, customer site, partners and so on), in which case the crossing of partitions may have particular constraints for security, allowed protocols, bandwidth and so on.

⊟ Analyze existing assets

Existing systems such as packaged or custom applications as well as industry standards, models and assets are the primary source to leverage in order to fulfill the realization of services. Rarely does an application get developed entirely from newly developed components. Existing services or components must be evaluated as we map out the services to be used by the solution. The process for the identification of candidate services that resides in reusable assets is described in Task: Existing Asset Analysis.

⊟ Identify services

Specify the candidate services that will be used in the solution. The actions of service identification step have many references according to the approach selected above.

Summarizing the steps above, SOA architects and designers have to address the issues related to the requirements translation into services definitions, they have to find services in existing systems, and then to refine these services using available tools and techniques to make them work in the real world. Once candidate services have been selected and documented in the (categorized) Service Portfolio, then architects and designers need to determine which ones should be exposed as services. Therefore, some criteria are needed to help decide whether to expose a service and most importantly, whether to fund the creation of the service component that will provide the functionality of the service as well as the maintenance, monitoring, security, performance and other service level agreements of the service.

⊟ Develop initial service specification

Specify the composition and collaboration of the candidate services that comprise the solution. SOA Architects and Designers can also specify how services can be combined to create composite services. We could say that previous step Identify Services is about the analysis of the Artifact: Service Model; service specification can be seen as the design of the Artifact: Service Model. SOA team needs just to outline the early Service Model. It is useful to remind that during the later service specification, the service model is fully specified in Task: Service Specification.

Figure 7-26 Service Analysis Task (Steps)

8

Catalog Manager case study

The purpose of this chapter is to apply the knowledge acquired from the previous chapters in this IBM Redbooks publication and use it to develop a sample application.

This chapter provides a reference to assist practitioners in developing an application iteratively in the System z environment. By identifying the similarities between the case study and a development exercise in your environment, you can use this chapter as a guide to estimate time intervals, identify tasks involved, and better understand the development methodology with a concrete example.

The case study uses a CICS catalog manager application to provide an implementation example of Rational Unified Process for System z (RUP for System z). This application is a working COBOL application that is shipped with CICS TS 3.1 and is designed to illustrate best practices for exposing CICS applications as Web services.

We will review the RUP for System z steps employed in the development of this sample application to demonstrate the agility of iterative development as compared to the traditional *waterfall* model used ubiquitously in development environments.

We realize that for any z/OS® development effort, there is normally a group of professionals spanning the software development dispersion. All information provided by this chapter must, therefore, only be used as a project guide and not an official solution.

8.1 Overview of the Catalog Manager application

The Catalog Manager application is a catalog-management, purchase order style application that accesses an order catalog stored in a VSAM file.

It provides the following functions:

- List the details of an item in a catalog.
- Order a quantity of a certain item.
- Replenish items in a catalog that have been depleted.

After an item is ordered, the catalog is automatically updated to reflect the new stock levels. Replenished items in the catalog are reset to a stock amount of 100.

The application is accessed from both a 3270 terminal interface and a Web-based interface using a commercially available Internet browser as illustrated in Figure 8-1. SOAP and Web services are used to expose the CICS-controlled information (catalog manager functions) as service-oriented architecture (SOA) service providers, which in turn are accessed using SOA service requestors from a browser.

Because the focus of this chapter is on the iterative development process applied to specific functions of the Catalog Manager application, much of the details concerning the CICS TS configuration and Web services are omitted. You can find additional information about these topics in *Application Development for CICS Web Services,* SG24-7126-00.

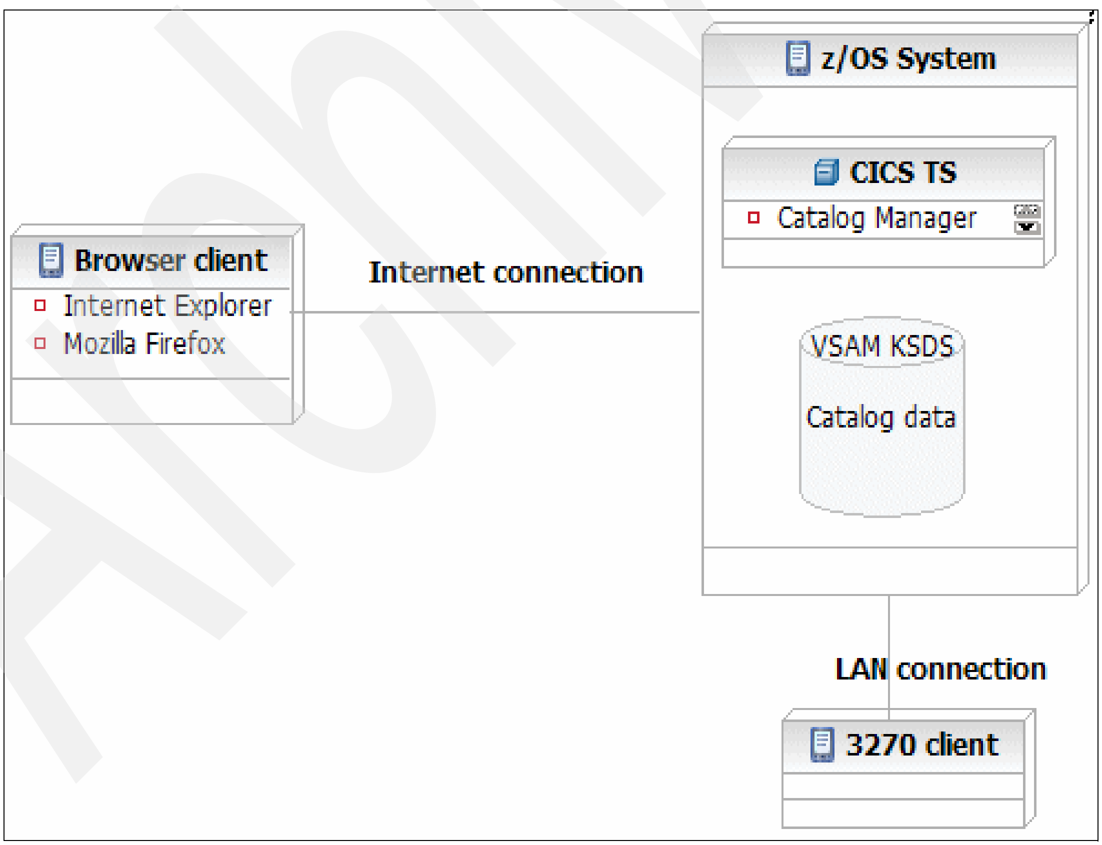


Figure 8-1 Catalog Manager System Architecture overview

8.2 Catalog Manager iterative development process

Applying the iterative process to our case study, the Catalog Manager development schedule is divided into eight iterations with each iteration spanning three weeks. The iteration cycle, as discussed in Chapter 3, “Why the IBM Rational Unified Process for System z” on page 31, typically spans two to three weeks with the total number of iterations dispersed among the four RUP phases: Inception, Elaboration, Construction, and Transition. The *iteration cycle* is the heartbeat of the project and after it has been selected, it remains constant for the duration of the project. Both the iteration cycle and the number of iterations are project dependent with the latter directly relating to the complexity of the project.

The Catalog Manager iterative development plan has one iteration in the Inception Phase, three iterations in the Elaboration Phase, two iterations in the Construction Phase, and two iterations in the Transition Phase. We assess and mitigate the risks during each iteration with each iteration culminating in a minor milestone for the project and facilitating successful achievement of the phase’s objectives.

Table 8-1 illustrates the Catalog Manager project plan broken down into iterations and phases. It depicts the relationship between the iterations and their respective phase, the relationship of the phases to each other, and the major milestone that marks the conclusion of each phase.

Table 8-1 Catalog Manager project plan

Task Name	Days	Start	Finish
Catalog Manager Project Plan	125	Mon 10/02/06	Fri 03/23/07
Inception	15	Mon 10/02/06	Fri 10/20/06
Inception Iteration I1- Preliminary Iteration	15	Mon 10/02/06	Fri 10/20/06
Lifecycle Objectives Milestone	0	Fri 10/20/06	Fri 10/20/06
Elaboration	45	Mon 10/23/06	Fri 12/22/06
Elaboration Iteration E1 - Architectural Prototype for CICS Application	15	Mon 10/23/06	Fri 11/10/06
Elaboration Iteration E2 - Architectural Prototype for Web Services Connectivity	15	Mon 11/13/06	Fri 12/01/06
Elaboration Iteration E3 - Architectural Prototype for Web Services Catalog Access	15	Mon 12/04/06	Fri 12/22/06
Lifecycle Architecture Milestone	0	Fri 12/22/06	Fri 12/22/06
Construction	29	Tue 01/02/07	Fri 02/09/07
Construction Iteration C1 - Develop Ordering Capability	14	Tue 01/02/07	Fri 01/19/07
Construction Iteration C2 - Develop Replenish Inventory Capability and Beta Release	15	Mon 01/22/07	Fri 02/09/07
Initial Operational Capability Milestone	0	Fri 02/09/07	Fri 02/09/07
Transition	30	Mon 02/12/07	Fri 03/23/07
Transition Iteration T1 - R1 Release	15	Mon 02/12/07	Fri 03/02/07
Transition Iteration T2 - R2 Release	15	Mon 03/05/07	Fri 03/23/07
Product Release Milestone	0	Fri 03/23/07	Fri 03/23/07

8.3 Catalog Manager RUP phases

The activities in each phase primarily focus on addressing a specific set of risks with the aim of reducing the risks and ensuring that the project is moving forward. The milestones at the end of each phase serve a dual process:

- The milestone serves to act as a driving force to attain a specific target by providing development status to our stakeholders, whose decisions are key to moving to the next phase.
- The milestones are checkpoints for the project as a whole, because they allow developers and management to track the progress of the work as they complete key points in the project lifecycle.

Table 8-2 further describes these phases in more detail as well as the associated major milestone that concludes the phase.

Table 8-2 Phases and milestones

Phase	Description	Milestone
Inception Phase	The Inception Phase will develop the product requirements and establish the business case for the system. The major use cases will be identified and a high level Software Development Plan will be developed. At the end of the Inception Phase, we will decide whether to fund and proceed with the project based upon the business case.	The *Lifecycle Objectives Milestone* at the end of the Inception Phase and marks the Go/No Go decision for the project.
Elaboration Phase	The Elaboration Phase will refine the requirements and develop a stable architecture. At the completion of the Elaboration Phase, all high risk use cases will have been analyzed and designed. An executable system called an *architectural prototype* will test the feasibility and performance of the architecture.	The *Lifecycle Architectural Milestone* marks the end of the Elaboration Phase. The major architectural components are in place and stable.
Construction Phase	During the Construction Phase, the remaining use cases will be analyzed and designed. The Beta release will be developed and distributed for evaluation. The implementation and test activities to support the R1.0 and R2.0 releases will be completed.	The *Initial Operational Capability Milestone* (completion of the beta) marks the end of the Construction Phase.
Transition Phase	The Transition Phase will prepare the R1.0 and R2.0 releases for distribution. It provides the required support to ensure a smooth installation.	The *Product Release Milestone* (completion of the R2.0 release) marks the end of the Transition Phase. At this point, all capabilities defined in the Vision document are installed and available for the users.

8.3.1 Catalog Manager Inception Phase

The Inception Phase addresses business risks so that we focus on mitigating the risk that the project might be either economically undesirable or technically infeasible. During this phase, it is crucial that we discuss with stakeholders their needs and the problems that our solution is attempting to solve. We scrutinize all aspects of the project as well as identify the major use cases.

One of the risks identified in building the Catalog Manager application is the software development team's unfamiliarity with Web services architecture and technology, which might preclude them from delivering the Web services component on time. To mitigate this, we decide to provide early training on Web services to the team members in Iteration one of the Elaboration Phase, prior to developing the Web services component.

Identifying the major use cases for a catalog order system, such as the Catalog Manager, entails getting everyone's agreement that a client needs to be able to list the items in a catalog as well as order a certain quantity of a specific item. Moreover, a customer service representative must be able to replenish (restock) depleted items in the catalog. The Catalog Manager application is illustrated in Figure 8-2.

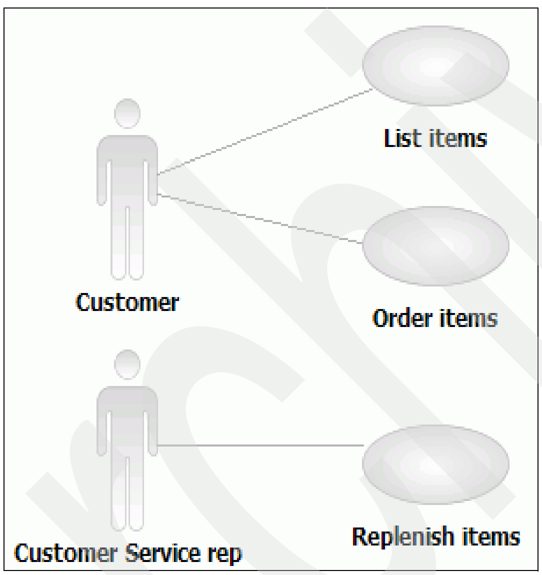


Figure 8-2 Catalog Manager application

Concluding the Inception Phase

The Inception Phase for the Catalog Manager concludes with the Lifecycle Objective Milestone, which indicates whether to proceed or abandon the project. At this stage, we propose a single solution that:

- Solves the right problem
- Is technically feasible
- Is economically viable

All stakeholders agree on these points prior to taking the project to the next step, that is, developing the architecture approach in the Elaboration Phase. If all stakeholders do not agree on these points, a decision to cancel the project is made. This can in fact be a desirable outcome of the Inception Phase, because terminating a project at this stage is the least expensive option of all the phases.

The following sections outline the Catalog Manager Inception Phase iteration details, work product deliverables, and the use of different tools to develop the project.

Iterations in the Inception Phase

The inception Phase has only one iteration that is summarized as shown in Table 8-3.

Table 8-3 Inception iteration

Inception iteration	Description	Risks addressed
I1 Iteration **(Preliminary Iteration)** (weeks 1-3)	**Define and approve Business Case.** **Define high-level product requirements.** The Vision document contains key features and constraints. **Define project scope.** A Use-Case Diagram includes key Actors and Use Cases. Only a brief description is provided for each Actor and Use Case. **Plan the overall project and next iteration.** A high-level Software Development Plan, a Risk List, and an Iteration Plan for the first elaboration iteration are created. **Create a very first draft of the Test Plan.** **Define application-specific terminology.** Important terms are defined in the Glossary.	Clarifies user requirements up front. Develops realistic Software Development Plans and scope. Determines feasibility of the project from a business point of view.

Work products produced in the Inception Phase

Table 8-4 summarizes the work products produced during the Inception Phase and their state of completion. After the decision is made to move on to the next Phase, in our case, the Elaboration Phase, then we must also create the plan for the first iteration of the Elaboration Phase (E1 Iteration Plan).

Table 8-4 Inception Phase work products

Inception Phase work products	Percent completion
Business Case	100
Vision	100
Glossary	40
Software Development Plan	80
Risk List	25
Use-Case Model	20
Supplementary Specification	20
Software Architecture Document	10
Catalog Manager Test Plan	10
E1 Iteration Plan	100

Tools used in the Inception Phase

The following tools are used to develop the work products in the Inception Phase:

- IBM Rational Software Architect/Modeler
  We use the Rational Software Architect (RSA) tool to create the Unified Modeling Language (UML) models, but the Rational Software Modeler (RSM™) tool can just as easily be used.
- IBM Rational Software Architect/Modeler and IBM Rational SoDA®
  We generate a use-case model survey report from RSA by employing a Rational SoDA template.
- IBM Rational RequisitePro®
  We manage our requirements by using Rational RequisitePro.
- IBM Rational Method Composer and IBM Rational Portfolio Manager
  We use Rational Method Composer to export the RUP for System z work breakdown structure into a Rational Portfolio Manager Software Development Plan. We then use Rational Portfolio Manager to tailor the Software Development Plan for the Catalog Manager Project and to detail the E1 Iteration Plan.
- IBM Rational Clear Case
  We use Rational Clear Case for configuration management.

8.3.2 Catalog Manager Elaboration Phase

The Elaboration Phase addresses architectural and technical risks. It spans three iterations culminating in a Lifecycle Architectural Milestone, which is an executable architecture. This is a partial implementation of the system to verify that we have a stable architecture to support the significant Functionality, Usability, Reliability, Performance, and Scalability (FURPS) requirements.

During this phase, we outline the basic and alternate flows of each use case as well as identify the most critical (important) use cases. For each critical use case, we identify the architecturally significant (most important) scenarios for the Catalog Manager and use them to create the executable architecture; that is, we design, implement, and test these scenarios. We also document these architectural scenarios in our Software Architecture Document.

For each of these scenarios:

1. We create a *use-case realization*, which is a sequence or interaction diagram identifying the components to fulfill the behavior specified by the scenario.
2. We develop test cases that validate the scenarios so that we know what the desired behavior is. Test cases during the Elaboration Phase focus more on identifying problem areas, such as load testing and performance, rather than validating that the desired behavior is correct.

But how do we determine these scenarios? Per Kroll, Manager of Methods, IBM Rational, suggests the following approaches:

- *The functionality is the core of the application, or it exercises key interfaces*.
  The system's key functionality must determine the architecture. Analyze factors such as: redundancy management strategies, resource contention risks, performance risks, data security strategies, and so on.
- *Choose use cases describing the functionality that must be delivered.*
  Delivering an application without its key functionality is fruitless.

- *Choose use cases describing functionality for an area of the architecture not covered by another critical use case.*
  Even if a certain area of the architecture does not appear to be high risk, it might conceal technical difficulties that can be exposed only by designing, implementing, and testing some of the functionality within that area.

Iterations in the Elaboration Phase

The Elaboration Phase has three iterations that can be summarized as shown in Table 8-5.

Table 8-5 Elaboration iterations

Elaboration iteration	Description	Risks addressed
E1 Iteration **Architectural Prototype for CICS Application** (week 4-6)	**Complete analysis and design for high risk requirements related to CICS.** Create Use-Case Specification for the *List Catalog Items* use case, derive an Analysis Model, and refine it into a Design Model. Document the architecture (high-level design) in the Software Architecture Document. **Develop the architectural prototype for CICS application.** Code the part of the application implementing the *List Catalog Items* use case. **Demonstrate feasibility and performance through testing.**	Architectural issues related to CICS clarified. Technical risks related to CICS mitigated. Early prototype for user review. Performance risks related to high volume of requests mitigated on the CICS side.
E2 Iteration **Architectural Prototype for Web Services Connectivity** (week 7-9)	**Train the team on Web Services.** **Complete analysis and design for high risk requirements related to Web Services.** Create Use-Case Specification for the *Configure Catalog* use case, derive analysis elements, and refine the Design Model. Refine the architecture (high-level design) in the Software Architecture Document. **Refine the architectural prototype for Web Services, so it establishes the connectivity between CICS and Web Services.** Code the Web service elements related to the *Configure Catalog* use case. **Demonstrate feasibility through testing (integrate as necessary).**	Risks of low skills related to Web Services and unknown technology mitigated. Architectural issues related to Web Services partially clarified. Technical risks related to Web Services partially mitigated.

Elaboration iteration	Description	Risks addressed
E3 Iteration **Architectural Prototype for Web Services Catalog Access** (week 10-12)	**Complete analysis and design for all remaining high risk requirements related to Web Services.** Derive analysis elements from the *List Catalog Items* use-case specification in the context of Web services and refine the Design Model. Refine the architecture (high-level design) in the Software Architecture Document. **Develop the architectural prototype for Web Services, so the Catalog Manager is available as a Web service.** Code the Web services elements related to the *List Catalog Items* use case. **Demonstrate feasibility and performance through testing (integrate as necessary).** **Define and Implement Installation Verification Procedures (IVPs) for the *List Catalog Item* use case.**	Architectural issues related to Web Services fully clarified. Technical risks related to Web Services fully mitigated. Early prototype for user review. Performance risks related to high volume of requests mitigated on the Web Services side. Browser Incompatibility Risk mitigated.

Iteration one of the Elaboration Phase - E1

The first iteration, E1, of the Elaboration Phase focuses on implementing an architectural prototype for the Catalog Manager application on the z/OS host. Consequently, we chose to implement the *List Items* use case for the 3270 interface. This is a core use case, because it validates the 3270 interface of the Catalog Manager application. It is also critical from a performance and load perspective, because it accesses the VSAM repository that lists all the items retrieved from the catalog.

Figure 8-3 shows the Catalog Manager user interface, which is a basic interface to fulfill the *List Catalog Items* use case and to exit the application. Options 2 and 3 are not implemented at this time but serve as placeholders for functionality to be implemented during later phases.

```
CICS EXAMPLE CATALOG APPLICATION  - Main Menu

Select an action, then press ENTER

Action . . . .     1. List Items
                   2. Order Item Number
                   3. Replenish Inventory
                   4. Exit




F3=EXIT    F12=CANCEL
```

Figure 8-3 Catalog Manager 3270 user interface

After the **List Items** function is selected, a list of items, descriptions, and costs displays (see Figure 8-4 on page 100).

```
CICS EXAMPLE CATALOG APPLICATION  - Inquire Catalog


Item    Description                             Cost
-----------------------------------------------------------------
0010    Ball Pens Black 24pk                    2.90
0020    Ball Pens Blue 24pk                     2.90
0030    Ball Pens Red 24pk                      2.90
0040    Ball Pens Green 24pk                    2.90
0050    Pencil with eraser 12pk                 1.78
0060    Highlighters Assorted 5pk               3.89
0070    Laser Paper 28-lb 108 Bright 500/ream   7.44
0080    Laser Paper 28-lb 108 Bright 2500/case 33.54
0090    Blue Laser Paper 20-lb 500/ream         5.35
0100    Green Laser Paper 20-lb 500/ream        5.35
0110    IBM Network Printer 24 - Toner cart   169.56
0120    Standard Diary: Week to view 8 1/4x5 3/4 25.99
0130    Wall Planner: Erasable 36x24           18.85
0140    70 Sheet Hard Back wire bound notepad   5.89
0150    Sticky Notes 3x3 Assorted Colors 5pk    5.35


F3=EXIT    F7=BACK   F8=FORWARD   F12=CANCEL
```

Figure 8-4 Catalog Manager List Items

In order to fulfill the behavior of the *List Catalog* use case, we designed and implemented the modules in Figure 8-5 on page 101. We followed best practices by employing a Model View Controller (MVC) design pattern separating the data (model) and user interface (view) concerns.

The data (*Model*) in the VSAM file is handled by module DFH0XVDS, the user interface (*View*) is handled by module DFH0XGUI, and the *Controller* module is DFH0XCMN.

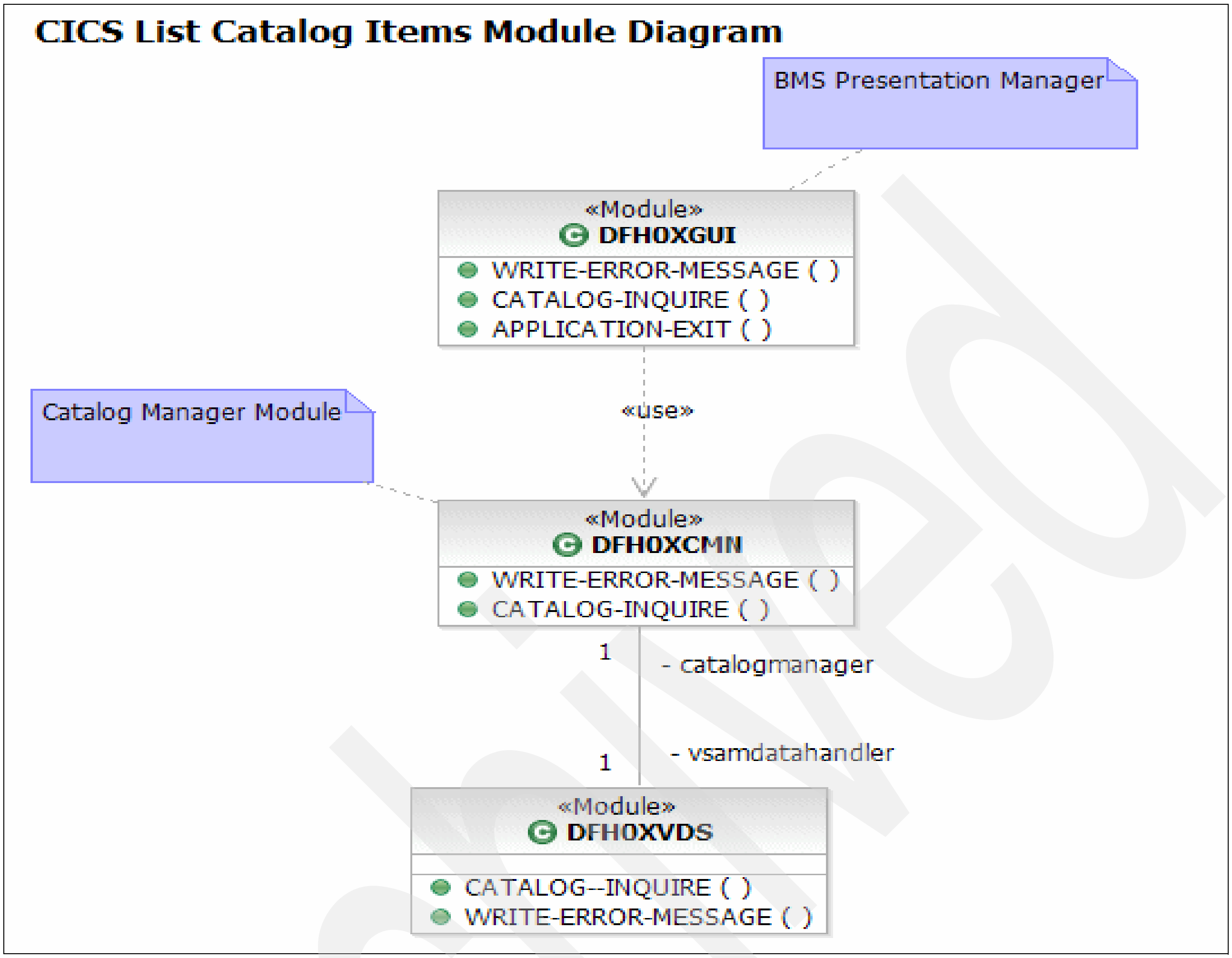


Figure 8-5 List Catalog Items module diagram

Iteration two of the Elaboration Phase - E2

The second iteration, E2, of the Elaboration Phase focuses on implementing an architectural prototype for the Catalog Manager Web services component. We chose to first implement the *Configure Catalog* use case for reasons similar to those we used for the *List Items* use case in iteration E1 of the Elaboration Phase. This is a core use case for the Web services component in that it exercises the Web services interface for the Catalog Manager and allows us to configure the Web services interface to communicate with the Catalog Manager Web services components on the z/OS host.

The Catalog Manager application on the z/OS host is really unaware that it is communicating with a Web service, because this communication is handled by the CICS Transaction Server (CICS TS) housing the Catalog Manager.

The *Configure Catalog* use case is implemented by first invoking the **CONFIGURE** option from a Web services client welcome panel as seen in Figure 8-6 on page 102.

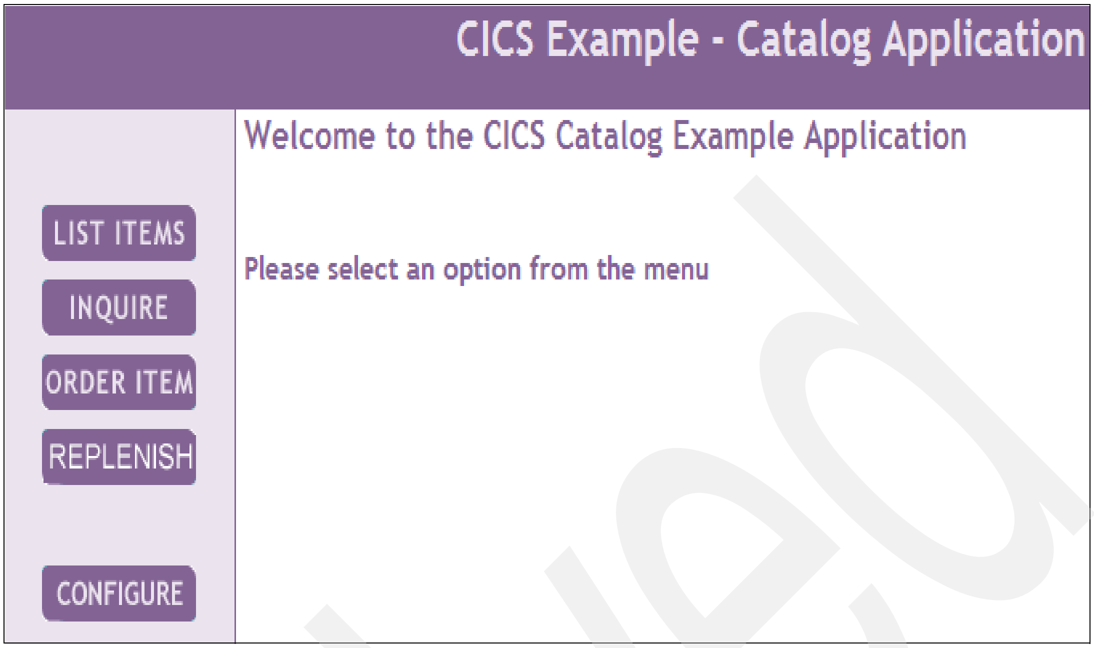


Figure 8-6 Catalog Manager Web services client welcome panel

Figure 8-7 shows the Configure Application panel that displays after selecting the **Configure** option from the panel in Figure 8-6. This panel allows you to specify the connection endpoints of the Catalog Manager service provider on the z/OS host. The other options (LIST ITEMS, INQUIRE, ORDER ITEM, and REPLENISH) have not been implemented at this stage. They are placeholders to allow for additional Web services functionality during a later iteration and phase.

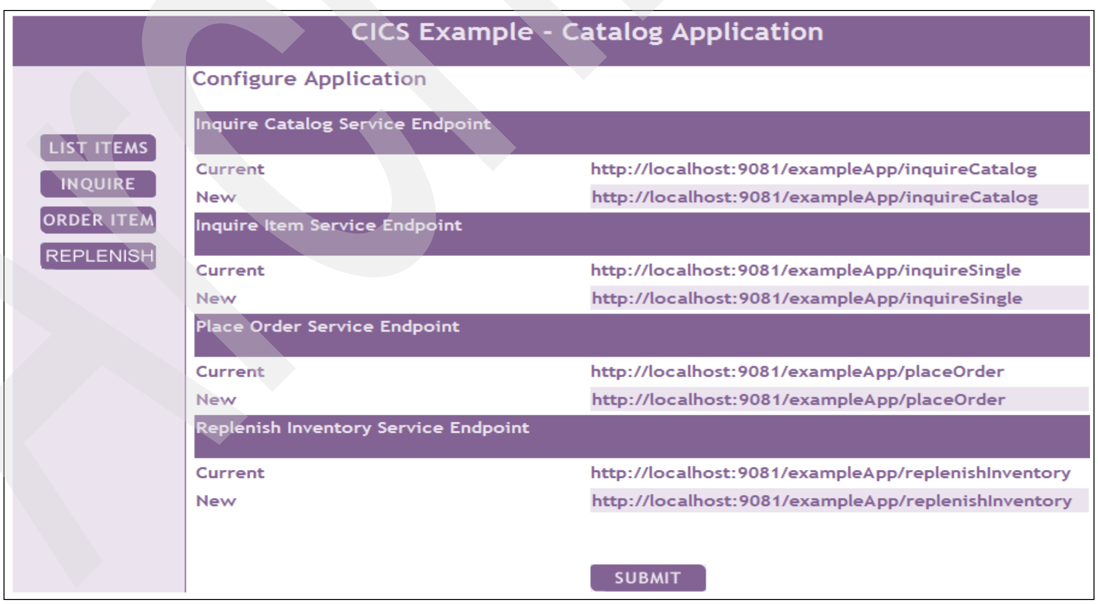


Figure 8-7 Catalog Manager Web services client configure panel

We again use an MVC design pattern to implement getting and setting the Web services configuration endpoints as seen in Figure 8-8 on page 103.

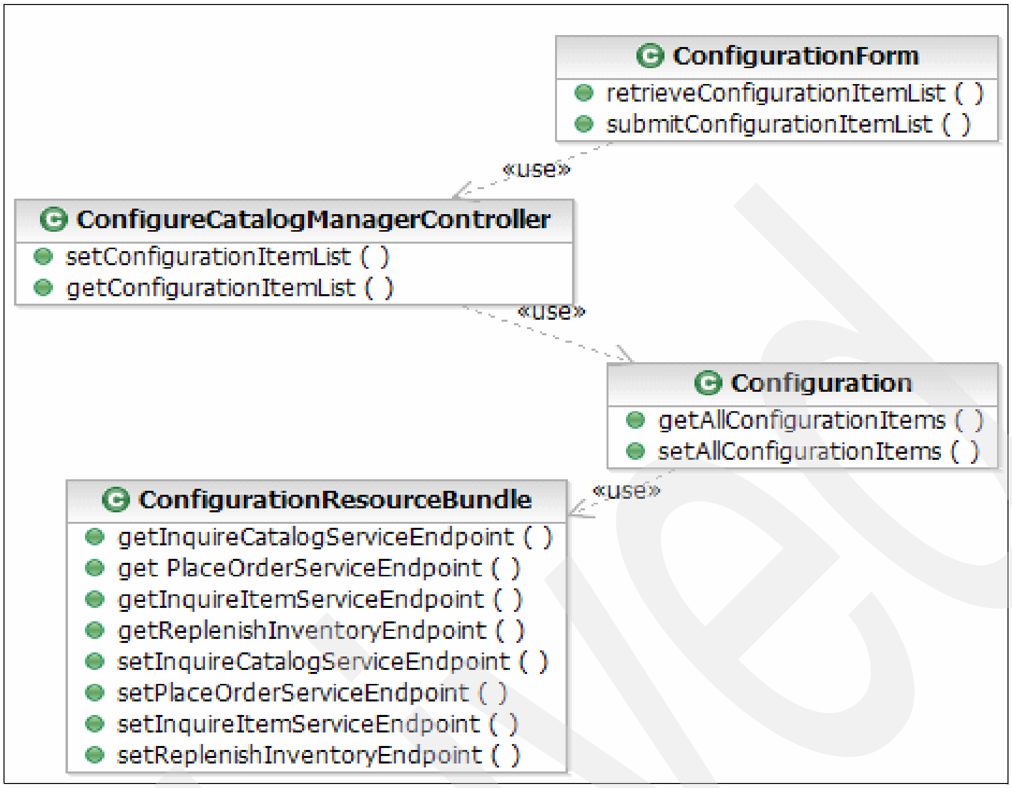


Figure 8-8 Configure Catalog Manager class diagram

Iteration three of the Elaboration Phase - E3

Iteration three, E3, is the final iteration in the Elaboration Phase. Our purpose here is to complete analysis and design for all remaining high risk requirements relating to Web services. If you recall, the *List Catalog* use case is implemented in iteration E1 of the Elaboration Phase for the Catalog Manager z/OS component. Therefore, for the same reasons that we described earlier, it makes sense to also first implement the *List Catalog* use-case Web service for the Catalog Manager Web services client interface.

Because we already have a placeholder (see Figure 8-7 on page 102) for the Web services *List Items* use case, we now develop the Web services client code to support this use case. However, we also need to enable the previously implemented *List Items* function (Elaboration E1) of the Catalog Manager on the z/OS host as a Web services provider. This is a necessary requirement, because the List Items Web services requester will be communicating with the List Items Web services provider in CICS.

There are two methods of converting a CICS COBOL program into a Web service provider:

- Use the CICS TS 3.1 Web services assistant program DFHLS2WS.
- Use the Web services enablement components wizard of WebSphere® Developer for zSeries (see pages 122-133 of *Application Development for CICS Web Services,* SG24-7126-00).

For our purposes, we utilize the CICS Web Services assistant program DFHLS2WS. There are, of course, a number of resource definition and configuration steps that we perform in the CICS environment to allow the Catalog Manager through CICS to act as a service provider. For a discussion about this topic, refer to *Application Development for CICS Web Services,* SG24-7126-00. We will discuss this topic in more detail during the implementation of the *Replenish Inventory* use case in the Construction Phase.

After we have enabled the List Items function as a Web services provider in CICS, we concentrate on developing the List Items Web services requester to invoke this function.

The implementation of this use case will enable our Web services client to request the Catalog Manager on the z/OS host to list the items in the catalog, just as though it were performing this function natively on a 3270 workstation. In this capacity, the Catalog Manager utilizing the services of CICS TS 3.1 is acting as a Web services provider. The items are retrieved and returned to the Web services client to be displayed.

After selecting the **List Items** option in Figure 8-6 on page 102, Figure 8-9 is displayed. On submitting this request, it sends the result of the List Items back to the Web services client interface to be displayed. The results are exactly the same as in Figure 8-4 on page 100.

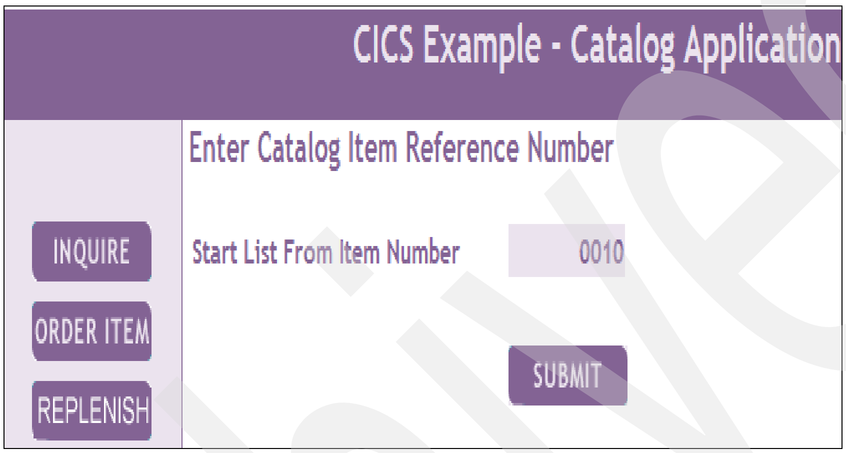


Figure 8-9 Catalog Manager Web service client List Items panel

Figure 8-10 depicts the model elements that we use to implement the *List items* use case. As before, we use a MVC design pattern for the implementation. For the CICSProvider interface, we only implement the inquireCatalog() function to support the use case, but we include the other placeholder functions for additional use cases in later iterations and phases.

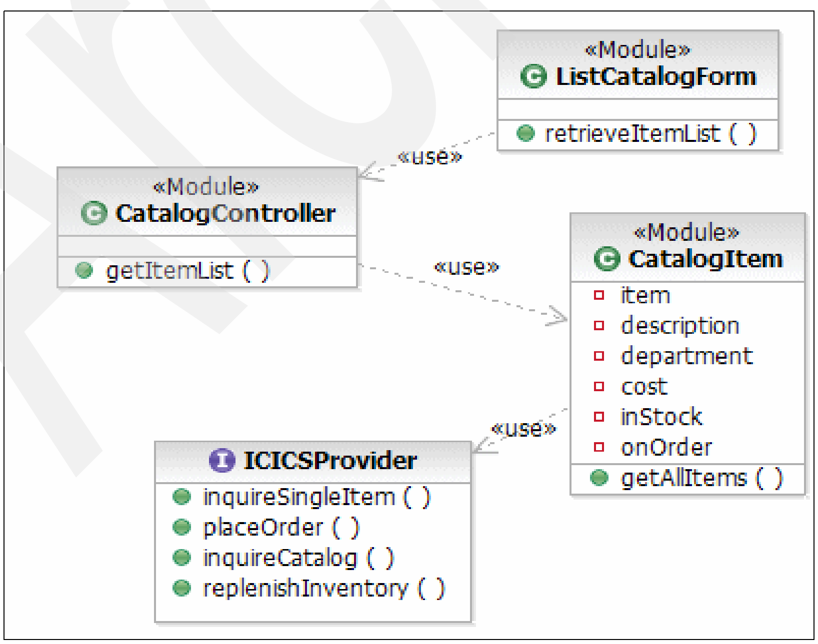


Figure 8-10 Catalog Manager List Items Class/Module Diagram

Concluding the Elaboration Phase

The Elaboration Phase for the Catalog Manager concludes with the Lifecycle Architectural Milestone. Our aim at this point is to:

- Bring architectural and technical risks under control.
- Establish and demonstrate a sound architectural foundation.
- Establish a credible plan for developing the product.

This means that we need to have a stable, proven architecture to handle the technical risks that we identify. In other words, the test scenarios for the use cases developed in the Elaboration Phase have not caused the system to fail. At this point, we have a partially completed design model, test cases, and executable code.

The decision to proceed to the Construction Phase is made based on us mitigating the technical risks that we identify.

The following sections outline the Catalog Manager Elaboration Phase iteration details, work product deliverables, and the use of different tools to develop the project.

Work products produced in the Elaboration Phase

Table 8-6 summarizes the work products produced during the Elaboration Phase and their state of completion.

Table 8-6 Elaboration Phase work products

Elaboration Phase work products	Percent completion
Glossary	80
Software Development Plan	95
E2, E3, and C1 Iteration Plans	100
C2 Iteration Plan	80
Risk List	50
Use-Case Model, including Use-Case specifications	80
Supplementary Specification	80
Software Architecture Document	100
Analysis Model	50
Design Model	60
Service Model	60
Test Plan	30
Test Cases, including Test Scripts	40
Test Evaluation Summary	Created
Source Code	40
Builds for E1, E2, and E3	Created
Installation Verification Procedures (IVPs)	80

Tools used in the Elaboration Phase

The following tools are used to develop the work products in the Elaboration Phase:

- IBM Rational Software Architect/Modeler
  We use the Rational Software Architect (RSA) tool to create the Unified Modeling Language (UML) models, but the Rational Software Modeler (RSM) tool can just as easily be used.
- IBM Rational Software Architect/Modeler and IBM Rational SoDA
  We generate a use-case model survey report from RSA by employing a Rational SoDA template.
- IBM Rational RequisitePro
  We manage our requirements by using Rational RequisitePro.
- IBM WebSphere Developer for zSeries (WDz)
  We use this tool to test the Catalog Manager Web service client.
- IBM Rational Portfolio Manager
  We use Rational Portfolio Manager to refine the Software Development Plan and to detail Iteration Plans.
- IBM Rational Clear Case
  We use Rational Clear Case for configuration management.

8.3.3 Catalog Manager Construction Phase

The Construction Phase addresses logistical risks, that is, completing the remaining work in the allotted time. It spans two iterations culminating in an Initial Operational Capability Milestone, which is assessing that the product is suitable to be delivered to the users.

During this phase, we do most of the work and implement all functionality. The remaining scenarios are detailed, designed, implemented, and tested, following a pattern not unlike that of the Elaboration Phase.

Up until this point, our testing has been focused on proving the suitability (Inception) and technical feasibility (Elaboration) of the solution. We switch gears now to concentrate more on testing the user interface of the solution, but we also need to ensure that prior architectural tests continue to work as the new functionality is implemented. Because the number of test cases has now grown, we make use of an automation tool to alleviate the manual testing effort.

Iterations in the Construction Phase

The Construction Phase has two iterations that can be summarized as shown in Table 8-7 on page 107.

Table 8-7 Construction iterations

Construction iteration	Description	Risks addressed
C1 Iteration **Develop Ordering Capability** (week 13-15)	**Implement and test key user requirements.** Create Use-Case Specification for the *Order Catalog Item* use case, derive analysis elements, refine the Design Model, and implement. **Integrate and Test.**	All important features from a user perspective are implemented.
C2 Iteration **Develop Inventory Replenishing Capability** **Beta Release** (week 16-18)	**Implement and test remaining requirements.** Create Use-Case Specification for the *Replenish Inventory* use case, derive analysis elements, refine the Design Model, and implement. **Integrate and Test.** **Prepare deployment.** **Assess if release is ready to go for beta testing.**	All required features are implemented in the Beta.

Iteration one of the Construction Phase - C1

The first iteration, C1, of the Construction Phase focuses on implementing the *Order Item* use case for both the 3270 interface and the Web client interface. Because we planned ahead and included placeholders for these functions (see Figures 8-8 and 8-11) on the main menu of the 3270 interface and the Web interface, we now concentrate on writing the supporting code.

The order item selection (for the 3270 interface) is made from the main menu as seen in Figure 8-11.

```
CICS EXAMPLE CATALOG APPLICATION  - Main Menu

Select an action, then press ENTER

Action . . . .   2 1. List Items
                   2. Order Item Number  20
                   3. Replenish Inventory
                   4. Exit
```

Figure 8-11 3270 interface order item

We develop the PLACE-ORDER() function to support the *order item* use case as seen in Figure 8-12 on page 108. Also, we have to Web enable this function in CICS using the CICS TS Web services assistant program DFHLS2WS, as we did before for the *List Catalog* function.

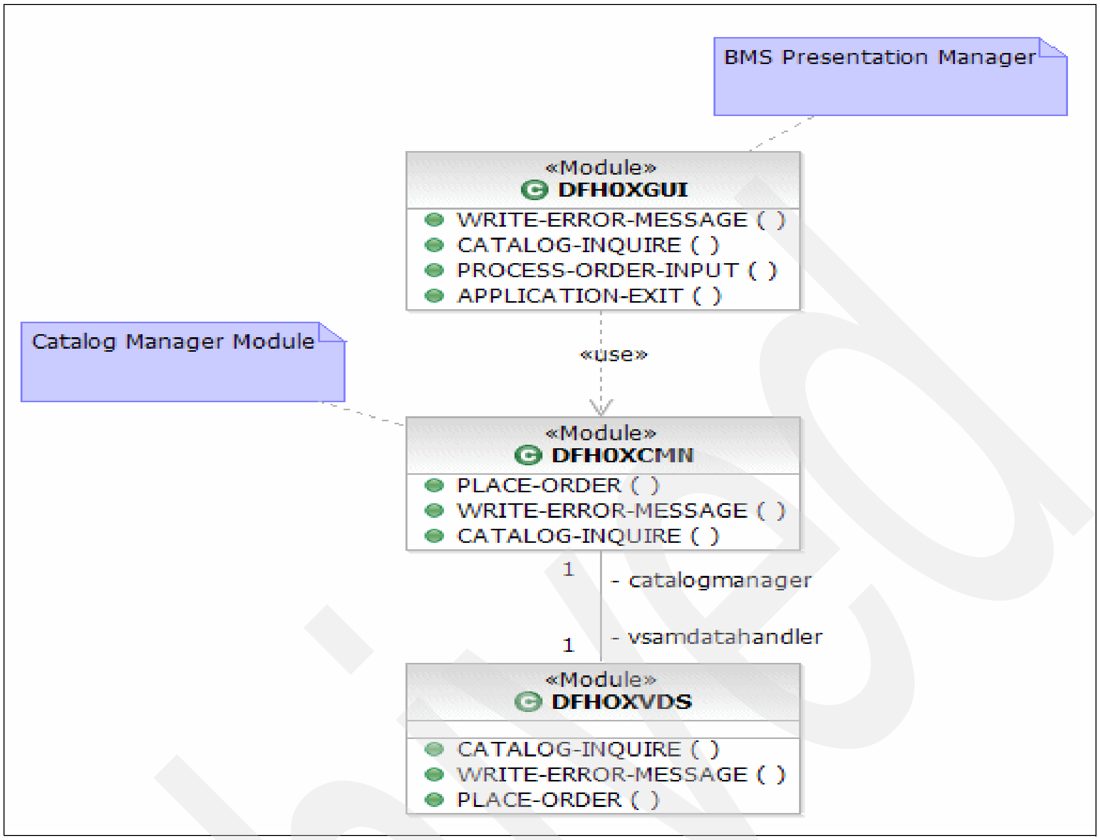


Figure 8-12 Catalog Manager order item module diagram

However, for the Web services interface, another panel is provided from the main menu as shown in Figure 8-13 that allows you to specify additional attributes, such as user name and department name, when placing the order.

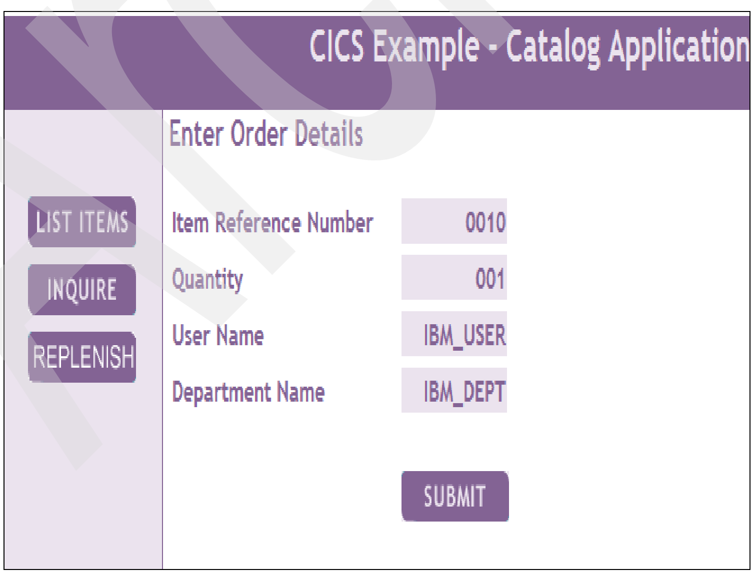


Figure 8-13 Catalog Manager Web services client order item panel

In Figure 8-14 on page 109, we add functionality in the Web services interface for the *order item* use case.

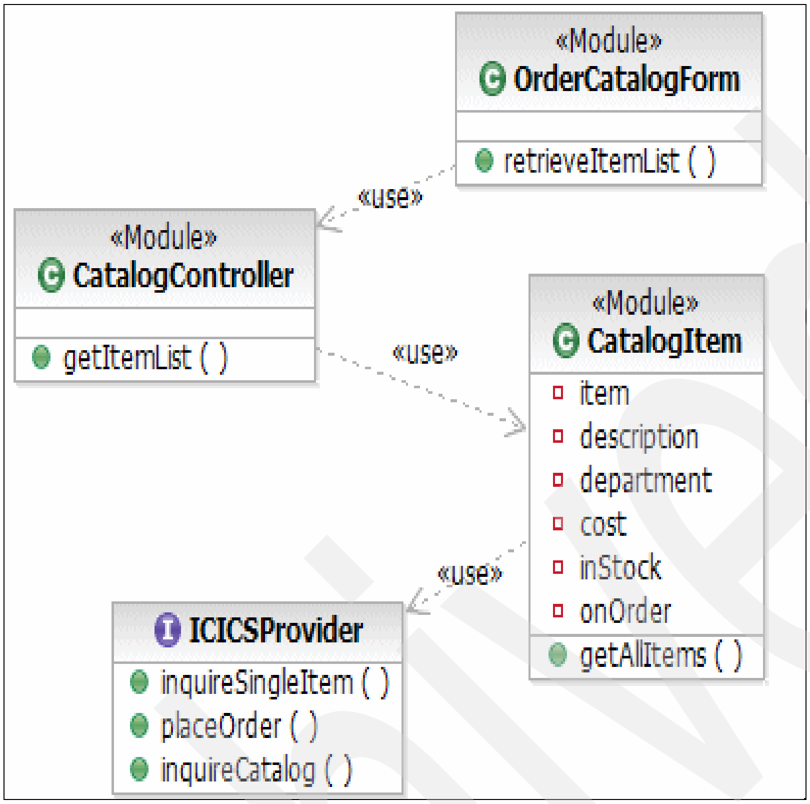


Figure 8-14 Catalog Manager order item class/module diagram

Iteration two of the Construction Phase - C2

The second iteration, C2, of the Construction Phase focuses on implementing the *Replenish Inventory* use case for both the 3270 interface and the Web client interface. Because we planned ahead and included placeholders for these functions (see Figure 8-3 on page 99 and Figure 8-1 on page 92) on the main menu of the 3270 interface and the Web interface, we now concentrate on writing the supporting code.

The **Replenish Inventory** selection (for the 3270 interface) is made from the main menu as seen in Figure 8-15, and the supporting functionality is illustrated in Figure 8-16 on page 110.

```
CICS EXAMPLE CATALOG APPLICATION  - Main Menu

Select an action, then press ENTER

Action . . . .    3 1. List Items
                    2. Order Item Number
                    3. Replenish Inventory
                    4. Exit
```

Figure 8-15 Catalog Manager Replenish Inventory

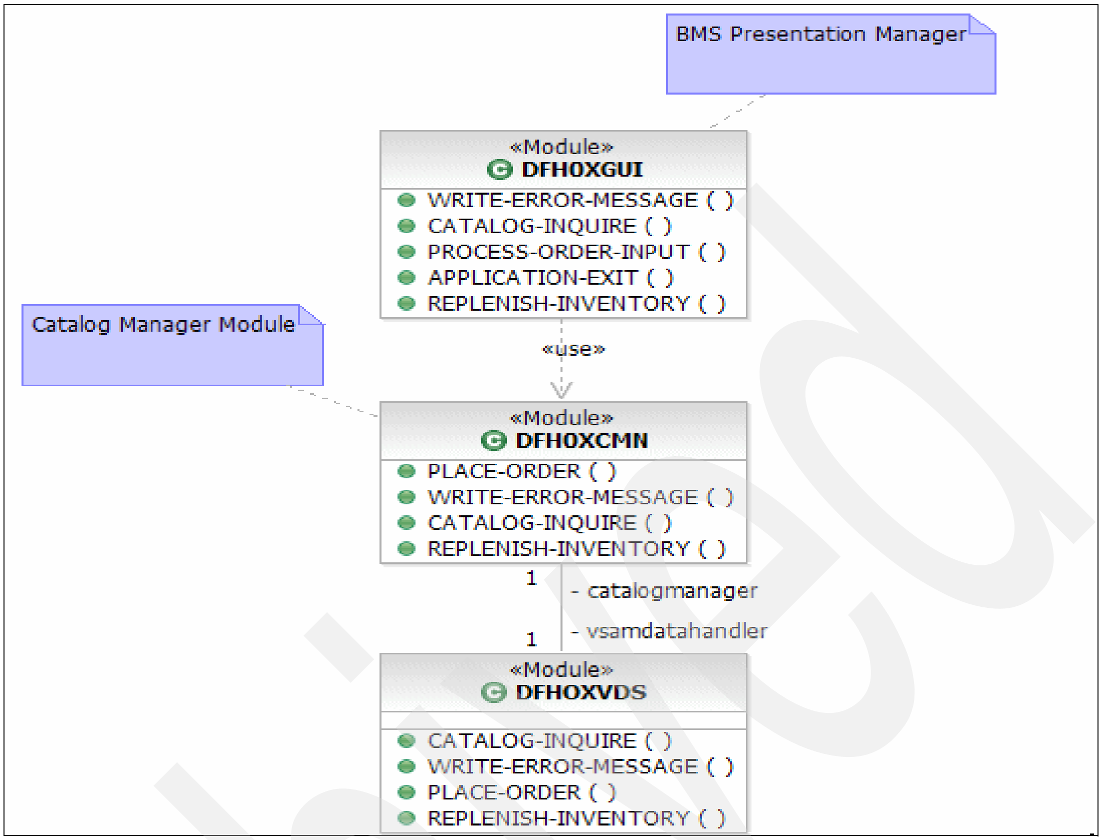


Figure 8-16 Catalog Manager Replenish Inventory module diagram

For the Web services interface, we also select the **REPLENISH** option from the Web interface main menu as shown in Figure 8-6 on page 102. The supporting modules/classes to implement this function are illustrated in Figure 8-17 on page 111. Again, we also need to Web enable the Replenish function in CICS as a Web services provider to communicate with our Replenish Web services requester client. The next section discusses this approach in more detail.

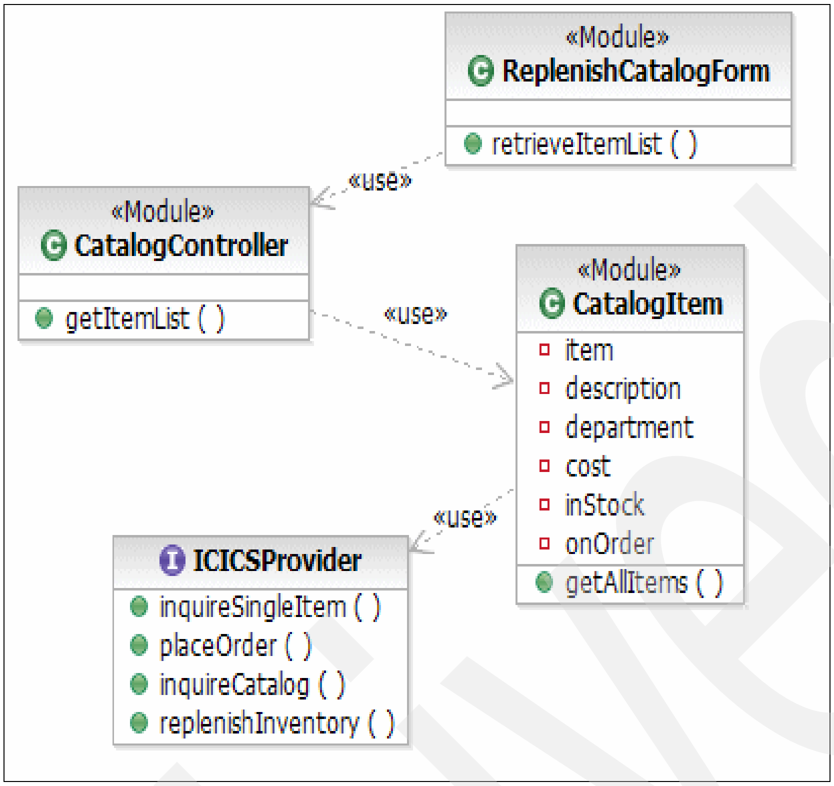


Figure 8-17 Catalog Manager Replenish Inventory class/module diagram

Enabling Replenish function as a Web service provider in CICS

There are three development approaches for creating Web services in CICS:

- **Top-down approach**: Creates a service from an existing Web Services Description Language (WSDL) and is used for new applications with existing WSDL or new WSDL. WSDL uses eXtensible Markup Language (XML) to specify the characteristics of a Web service: name of the Web service, what it can do, and how it is invoked.
- **Bottom-up approach**: Creates a WSDL from an existing application and is used for an existing application.
- **Meet-in-the-middle-approach**: Used for an existing application with existing WSDL.

Because we already have an existing COBOL application for the Replenish function, we employ the Bottom-up-approach.

We now execute the following steps to enable the Replenish function as a Web service provider in CICS:

1. Run CICS Web Services assistant DFHLS2WS passing it as input; our replenish function data structure as shown in Figure 8-18 on page 112. This does the following tasks:
   a. Creates a WSDL for the replenish function.
   b. Creates a WSBIND file. The WSBIND file is used by CICS to:
      i. Transform Simple Object Access Protocol (SOAP) messages to application data on input.

         SOAP is the protocol that is used to communicate among the three actors in an SOA, as shown in Figure 8-19 on page 112: the service provider (Catalog Manager via CICS), the service requester (Web services client), and the service broker. The service broker (also known as a *service registry*) makes the Web service access and interface information available to any potential service requester. A service broker is not used in the Catalog Manager example.

ii. Transform application data to SOAP messages on output.

```
Catalogue COMMAREA structure
03 CA-REQUEST-ID            PIC X(6).
03 CA-RETURN-CODE           PIC 9(2) DISPLAY.
03 CA-RESPONSE-MESSAGE      PIC X(79).
Fields used in Replenish Inventory
03 CA-ORDER-REQUEST.
    05 CA-USERID                PIC X(8).
    05 CA-CHARGE-DEPT           PIC X(8).
    05 CA-ITEM-REF-NUMBER       PIC 9(4) DISPLAY.
    05 CA-QUANTITY-REQ          PIC 9(3) DISPLAY.
    05 FILLER                   PIC X(888).
```

Figure 8-18 Replenish Inventory function data structure

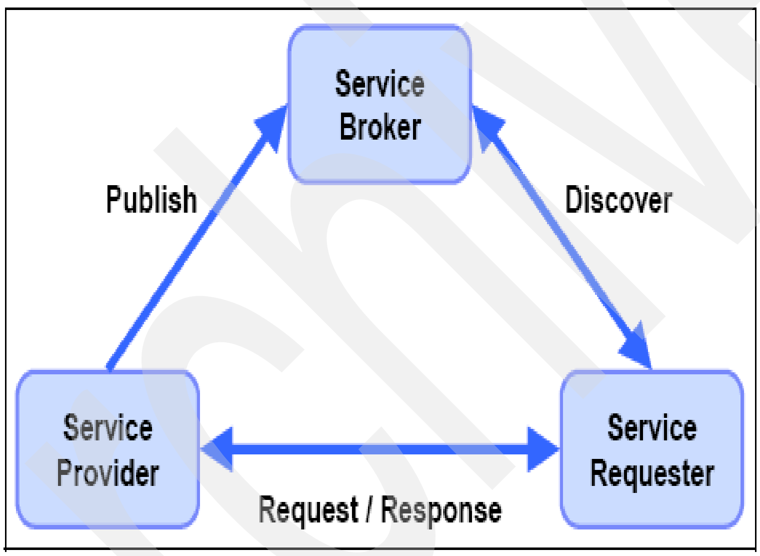


Figure 8-19 Service-oriented architecture (SOA) components and operations

Example 8-1 on page 113 shows the input parameters passed to DFHLS2WS for the Replenish Inventory function.

Example 8-1 Input parameters

```
//LS2WS     EXEC DFHLS2WS,
//  JAVADIR='/usr/lpp/java/J1.4/',
//  USSDIR='cicsts31',
//  PATHPREF=''
//INPUT.SYSUT1 DD *
PDSLIB=//RUP4Z.SDFHSAMP
PGMNAME=DFH0XCMN
LANG=COBOL
PGMINT=COMMAREA
REQMEM=DFHRUP4Z
RESPMEM=DFHRUP4Z
LOGFILE=/u/rup4z/provider/wsbind/replenishInventory.log
WSBIND=/u/rup4z/provider/wsbind/replenishInventory.wsbind
WSDL=/u/rup4z/provider/wsdl/replenishInventory.wsdl
URI=exampleApp/replenishInventory
```

The input parameters have the following meanings:

PDSLIB The library containing the Replenish Inventory program that is exposed as a Webservice.

PGMNAME The name of the program for the CICS Catalog Manager example application DFH0XCMN.

LANG Specifies the language in which the program is written (in this example, COBOL).

PGMINT Describes the program input. DFH0XCMN uses a COMMAREA.

REQMEN and RESPMEM Define the copybooks for request and response.

LOGFILE, WSBIND, and WSDL Specify the fully qualified hierarchical file system (HFS) file names of the files to be generated.

URI Stands for the URIMAP that is used to map a Web services request to a Web service.

2. Copy the generated WSBIND file to a UNIX® directory on z/OS that will act as a Web service pickup directory for the PIPELINE (see next step). For our example, we define a pickup directory called `/u/rup4z/provider/wsdir`.

3. Define a service provider PIPELINE in CICS using the following CICS transaction:

   CEDA DEFINE PIPELINE(RUP4ZPIP)

   The output of this transaction is illustrated in Figure 8-20 on page 114.

```
DEFINE PIPELINE(RUP4ZPIP)
OVERTYPE TO MODIFY                                       CICS RELEASE = 0640
 CEDA  DEFine PIpeline( RUP4ZPIP )
  PIpeline     ==> RUP4ZPIP
  Group        ==> SOADEVWS
  Description  ==> RUP4Z PIPELINE for Catalog Manager Replenish Web services
  STatus       ==> Enabled          Enabled | Disabled
  Configfile   ==> /usr/lpp/cicsts/cicsts31/samples/pipelines/basicsoap11prov
  (Mixed Case) ==>  ider.xml
               ==>
               ==>
               ==>
  SHelf        ==> /var/cicsts/
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  Wsdir          :/u/rup4z/provider/wsdir
  (Mixed Case)   :
                 :
```

Figure 8-20 PIPELINE definition for Replenish Web services provider

A *PIPELINE* is a sequence of programs arranged so that the output from one program is used as input to the next program. There are pipelines that support service providers and pipelines that support service requesters.

In our example, we are creating a service provider PIPELINE, which is a pipeline of user-provided and system-provided programs that receives an inbound SOAP message, processes the contents, and sends a response.

4. Install the PIPELINE in CICS, which will subsequently create the WEBSERVICE definition in CICS:

   CEDA INSTALL PIPELINE(RUP4ZPIP) GROUP(SOADEVWS)

5. We check if the WEBSERVICE is installed. Note that the Web service definition is required to map the incoming SOAP body to the COMMAREA interface of the program.

   CEMT INQUIRE WEBSERVICE

   The replenishInventory Web service definition is displayed in Figure 8-21 on page 115.

```
INQUIRE WEBS
RESULT - OVERTYPE TO MODIFY
  Webservice(replenishInventory)
  Pipeline(RUP4ZPIP)
  Validationst( Novalidation )
  State(Inservice)
  Urimap($309050)
  Program(DFH0XCMN)
  Pgminterface(Commarea)
  Container()
  Datestamp(20061102)
  Timestamp(13:09:05)
  Wsdlfile()
  Wsbind(/u/rup4z/provider/wsdir/replenishInventory.wsbind)
  Endpoint()
  Binding(DFH0XCMNHTTPSoapBinding)
```

Figure 8-21 Catalog Manager replenishInventory Web service

6. The URIMAP $309050 in Figure 8-21 is created dynamically by CICS. It is used to map an incoming request to the associated Web service and pipeline. CICS bases the definition on the URI specified in the input to DFHLS2WS in step 1 and stored by DFHLS2WS in the WSBIND file. We list the contents of the URIMAP by issuing the following command:

   CEMT I URIMAP($309050)

   The output is displayed in Figure 8-22.

```
INQUIRE URIMAP($309050)
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Uri($309050 ) Pip Ena     Http
    Host(*                        ) Path(/exampleApp/replenishInventory  )
```

Figure 8-22 URIMAP for replenishInventory

Notice that the PATH attribute is set to the URI that will be found in the HTTP request issued by our Web client.

7. We use WebSphere Developer for zSeries to import our WSDL /u/rup4z/provider/wsdl/replenishInventory/wsdl into our Software Development Platform (SDP) as shown in Figure 8-23 on page 116.

```
1. Start up WebSphere Developer for zSeries v6.
2. Create a new project as follows: File, New, Project, “My-WSDL”
3. Copy WSDL file to workstation: File, Import, FTP
4. Host=myzOS or 9.9.9.9
(CEMT I TCPIPS, then expand output to get your IP address)
5. Folder=/u/rup4z/provider/wsdl/
6. login=userlogon
7. password=userpass
8. finish
```

Figure 8-23 Importing our replenishInventory wsdl file

8. We are ready to test our Web services, so we utilize the WSDL editor in WDz as shown in Figure 8-24:
   a. In WDz, right-click your WSDL file and select **Open With WSDL Editor**.
   b. In the WSDL editor, click the **Graph** tab.
   c. In the Services pane, expand **services** → **port** → **soap:address.**
   d. Click the **Properties** tab to specify the Web service endpoint.

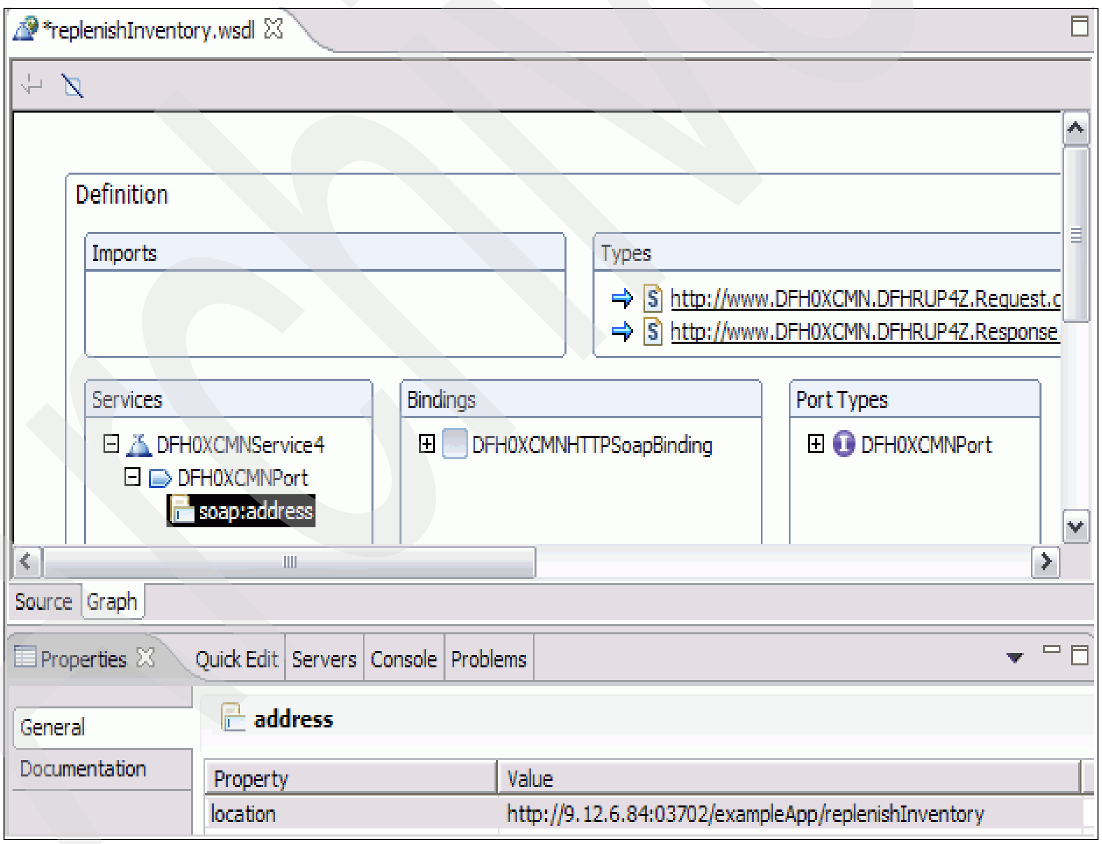


Figure 8-24 replenishInventory Web services endpoint

Notice that the Web services address is:

http://*host ip address*:*host port*/*URI for replenishInventory*.

This URI corresponds to the URI path defined earlier in CICS for the replenishInventory Web services provider.

9. We now test the Web service by:
   a. Right-clicking on the WSDL file
   b. Selecting **Web Services Test with Web Services Explorer**
   c. Right-clicking on the DFH0XCMNOperation link

   Web Services Explorer provides a form where you enter your request-specific data. All values must be entered according to the WSDL specification as shown in Figure 8-25.

   a. Enter `01REPL` for ca_request_id.
   b. Enter 0 for other fields.
   c. Click **Go.**

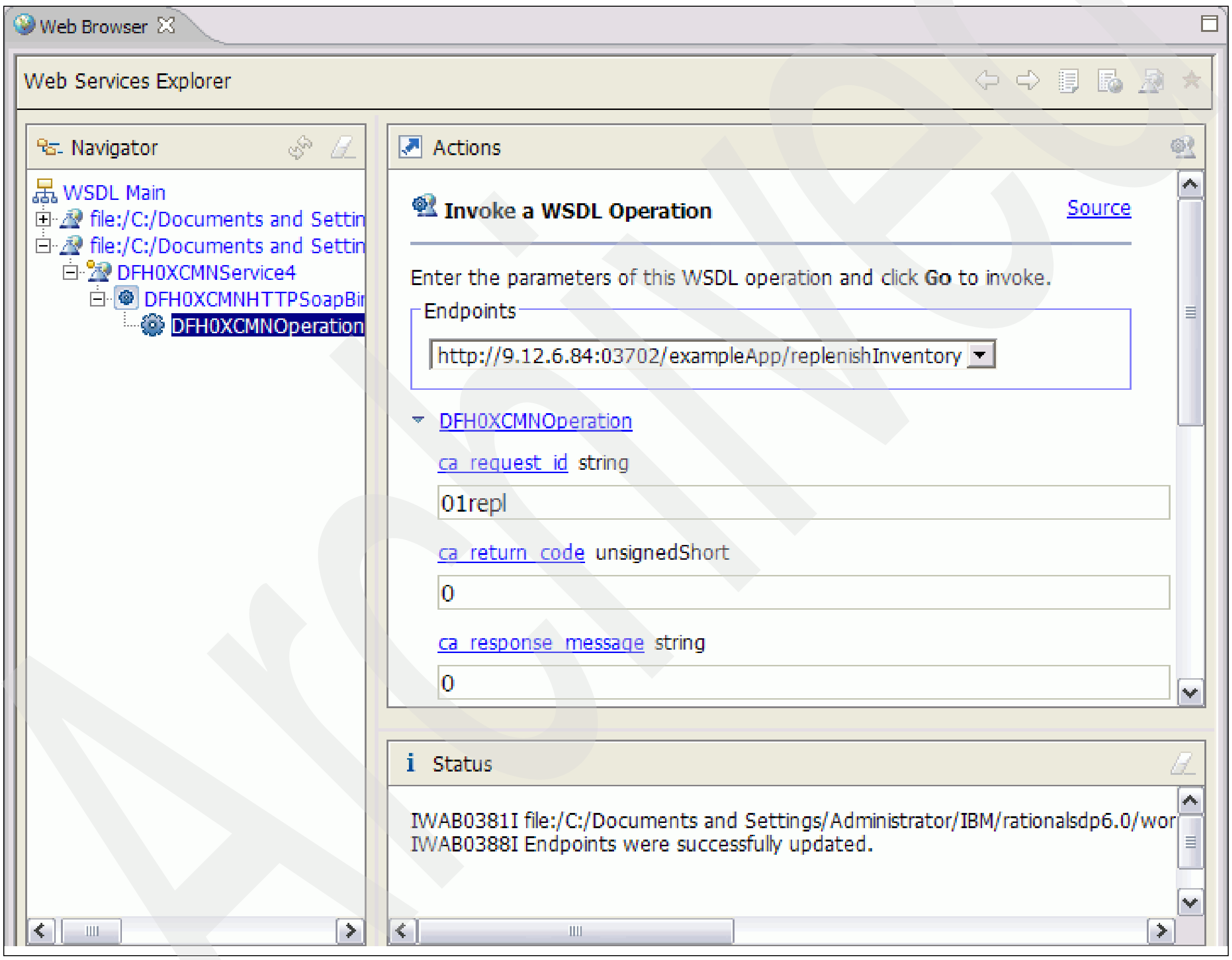


Figure 8-25 Invoking the replenishInventory Web service in WdZ

10. The results of the replenishInventory request are displayed in the status panel as shown in Figure 8-26 on page 118.

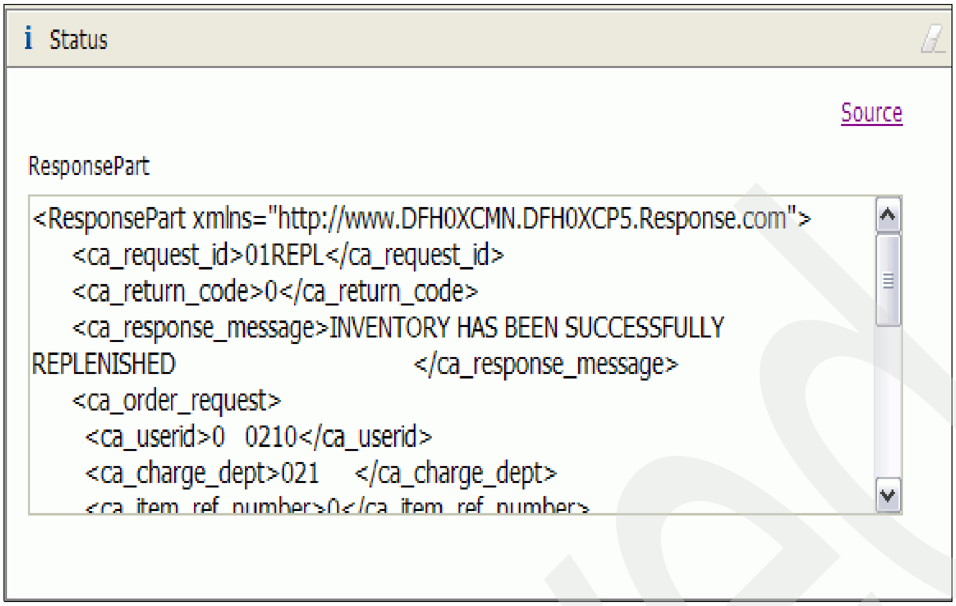


Figure 8-26 Response from replenishInventory Web service request

Concluding the Construction Phase

The Construction Phase for the Catalog Manager concludes with the Initial Operational Capability Milestone. Our aim at this point is to:

- Ensure that the solution is developed according to the requirements.
- Ensure that the solution is ready to be delivered to the users and stakeholders.
- Achieve adequate quality as rapidly as possible.

We are now ready to deploy the solution as a beta release to be evaluated by users and stakeholders. This takes us to the Transition Phase.

The following sections outline the Catalog Manager Construction Phase iteration details, work product deliverables, and the use of different tools to develop the project.

Work products produced in the Construction Phase

Table 8-8 on page 119 summarizes the work products produced during the Construction Phase and their state of completion.

Table 8-8 Construction Phase work products

Construction Phase work products	Percent completion
Glossary	90
Software Development Plan	100
C2 and T1 Iteration Plans	100
T2 Iteration Plan	80
Risk List	75
Use-Case Model, including Use-Case specifications	100
Supplementary Specification	100
Analysis Model	100
Design Model	95
Service Model	95
Test Plan	90
Test Cases, including Test Scripts	80
Deployment Plan	Created
Source Code	95
Builds for C1, C2, and Beta	Created
Installation Verification Procedures (IVPs)	90

Tools used in Construction

The following tools are used to develop the work products in the Construction Phase:

- IBM Rational Software Architect/Modeler
  We use the Rational Software Architect (RSA) tool to create the Unified Modeling Language (UML) models, but the Rational Software Modeler (RSM) tool can just as easily be used.
- IBM Rational Software Architect/Modeler and IBM Rational SoDA
  We generate a use-case model survey report from RSA by employing a Rational SoDA template.
- IBM Rational RequisitePro
  We manage our requirements by using Rational RequisitePro.
- IBM WebSphere Developer for zSeries (WDz)
  We use this tool to test the Catalog Manager Web services client.
- IBM Rational Manual Tester
  We use this tool to exercise our test cases.
- IBM Rational Functional Tester
  We use this tool to automate our testing suites.
- IBM Rational Portfolio Manager
  We use Rational Portfolio Manager to refine the Software Development Plan and to detail Iteration Plans.

- IBM Rational Clear Case
  We use Rational Clear Case for configuration management.

8.3.4 Catalog Manager Transition Phase

The Transition Phase addresses solution rollout (delivery) risks and brings these risks under control. It spans two iterations culminating in a Product Release Milestone, which marks the product completion. During this phase, we are mostly concerned about deployment and fixing defects identified in the released product. We decide to deliver the Catalog Manager as two releases: R1 containing only the 3270 components and R2 containing the Web services components.

Iterations in Transition

The Transition Phase has two iterations that can be summarized as shown in Table 8-9.

Table 8-9 Transition Phase iterations

Transition iteration	Description	Risks addressed
T1 Iteration **R1 Release** (week 12-21)	Refine the Installation Verification Procedures (IVPs). Deploy Beta. Fix defects from Beta and incorporate feedback. Package, distribute, and install R1 Release at our business partner sites.	User feedback prior to release of R1. High Product quality. Defects minimized. Quality of service (QOS) improved. R1 fully reviewed by user community.
T2 Iteration **R2 Release** (week 22-24)	Fix defects from R1 and incorporate feedback. Package and distribute R2 Release through the Web.	Two-stage release minimizes defects and provides easier transition for users.

Iteration one of the Transition Phase - T1

The first iteration, T1, of the Transition Phase focuses on deploying the beta release and fixing the associated defects. After these defects are addressed and our users and stakeholders are satisfied, we package and deploy R1 of the product, which contains only the 3270 components.

The span between R1 and R2 allows us to address any interdependent defects in the Web services components that might arise from user testing of R1 of the product.

Iteration two of the Transition Phase - T2

The second iteration, T2, of the Transition Phase focuses on deploying the second release, R2, which contains only the Web services components. We ensure that sufficient training material is provided, because this release is more comprehensive. It requires prerequisite products on the workstation as well as the host.

Concluding the Transition Phase

The Transition Phase for the Catalog Manager concludes with the Product Release Milestone. Our aim at this point is to:

- Deliver the solution to its users.
- Achieve user self-sufficiency.

Successful deployment of the product indicates that the Product Release Milestone has been achieved. Of course, Product Release milestone assessment is based on the satisfaction of our users and stakeholders.

The following sections outline the Catalog Manager Transition Phase iteration details, work product deliverables, and the use of different tools to develop the project.

Work products produced in the Transition Phase

The table in Table 8-10 summarizes the work products produced during the Transition Phase and their state of completion.

Table 8-10 Transition Phase work products

Transition Phase work products	Percent completion
Glossary	100
Iteration Plans T2	100
Risk List	100
Design Model	100
Service Model	100
Test Plan	100
Test Cases	100
Test Scripts	100
Source Code (Implementation Elements)	100
Installation Verification Procedures (IVPs)	100
Builds for R1 and R2	Created

Tools used in the Transition Phase

The following tools are used in the Transition Phase:

- IBM Rational Software Architect/Modeler
  We use the Rational Software Architect (RSA) tool to create the Unified Modeling Language (UML) models, but the Rational Software Modeler (RSM) tool can just as easily be used.
- IBM Rational Manual Tester
  We use this tool to exercise our test cases.
- IBM Rational Functional Tester
  We use this tool to automate our testing suites.
- IBM Rational Portfolio Manager
  We use Rational Portfolio Manager to detail Iteration Plans.
- IBM Rational Clear Case
  We use Rational Clear Case for configuration management.

9

EGL Web Service consumption case study

The purpose of this chapter is to provide an introduction to the Enterprise Generation Language (EGL) and illustrate how quickly and simply it can be used to develop the Web interface of the Catalog Manager case study application introduced in the previous chapter. This chapter does not attempt to explain Web development concepts. For an introduction to these concepts in the context of EGL, refer to the excellent EGL tutorials on developerWorks:

http://www-128.ibm.com/developerworks/rational/products/egl/egldoc.html

9.1 Introduction to Enterprise Generation Language

Enterprise Generation Language (EGL) is a highly productive and intuitive high level programming language, which allows the developer to focus on business-logic rather than target platform runtime nuances. For example, a System z developer can quickly develop a Web client without extensive knowledge of middleware programming or JAVA/J2EE™. The EGL source code is generated into either COBOL or Java depending on the desired target execution environment. The same source code can be deployed to various execution platforms. The target environment specifics are limited to the build descriptor files, which control the generation process. Build descriptor files and record definitions also isolate datastore specifics allowing the EGL developer to use the simplified coding constructs to access data regardless of the underlying datastore, such as relational database, DL/I database, MQ Series, and serial file. As a result, the EGL developer has to handcraft very little EGL code in order to deploy code that is optimally built for the target datastores and target execution platform.

EGL is a feature that is bundled with the following version 6.0.*x* WebSphere and Rational design and construction products:

- Rational Software Developer Platform (RAD)
- Rational Software Architect (RSA)
- WebSphere Developer for zSeries (WDz)
- WebSphere Developer Studio Client (for iSeries®)

Both platforms integrate rapid development technologies, such as Java Server Faces (JSF), within the Eclipse framework, which produce a highly productive development environment. JSF is a server-side user interface component framework, which is graphical, consistent, and easy to use. The framework provides a simple model for the development of Java-based dynamic Web pages. EGL integration into these IBM and Rational products exploits drag and drop development (for example, Web page development) and declarative programming to specify properties (for example, data element properties). These simplified development styles result in high quality code generated and compiled by EGL rather than crafted by the developer. The EGL perspective provides EGL specific editors and a debugger, which provide a common look and feel for both Web-centric and data-intensive programs. The EGL editor has the following functionality:

- Standard editing operations
- 4GL macro statements
- Drag and drop rapid development techniques
- Context-based Content Assist
- Colorized language elements
- Code Templates
- Code Snippets
- Editor view preferences (colors, fonts, hide/show line numbers, and so forth)
- Integration with the syntax compiler to display compile errors

These features of the EGL editor contribute to EGL's goal to provide a simplified approach to application development. The EGL and JSF aware editors facilitate the ease of learning the language syntax and programming paradigm. The technology neutral specification of the EGL language, EGL code generation, and EGL-based debugging provides complete, end-to-end isolation from the complexity of the deployment environment. Therefore, existing System z procedural programmers can quickly utilize the EGL and JSF framework to develop a Web application without extensive J2EE training. The RAD EGL/JSF tooling quickly produces production quality functionality, thereby shifting more development cycles from construction to analysis and design.

9.2 Development approach

The Catalog Manager case study involved using EGL to develop a Web interface to request CICS Web Services. The pervious chapter concentrates on the application of RUP for Z development during the case study. This chapter emphasizes the construction of the code. For details concerning the CICS TS configuration and Web services, refer to *Application Development for CICS Web Services,* SG24-7126-00. Note, EGL could have easily been used to implement the Web services and the 3270 client as well. However, it was decided to exploit the existing CICS code in order to concentrate on the Web client development in EGL.

System requirement to recreate and execute this sample code is version 6.0.*x* of Rational Software Developer, Rational Software Architect, or WebSphere Developer for zSeries at the latest level of maintenance with features EGL and IBM WebSphere Application Server V6.0 Test Environment installed. Configuration of the development workspace will be covered in the elaboration section.

Parallel development was used to develop this Web client application. The page template was developed independently of the Elaboration Phase use cases (that is, the List Catalog Items and the Configure Catalog use cases). An experienced Web developer was assigned the page template, because it required HTML and JavaScript™ knowledge. The page template has a navigational link to all the pages in the application. JavaScripts were used to suppress the navigational link to the currently rendered page. Thus, the replenish page will not have the replenish navigational link. The new EGL Web developer was able to concentrate on how to achieve the use-case functionality rather than be obsessed with the appearance of the pages. The key concepts utilized during the Elaboration Phase were:

- Web Services Explorer to test the WSDL file
- Project Explorer view **Create EGL Interfaces and Binding Library**. Menu option to EGL artifacts from WSDL files
- Page Data View **Insert New control for select objects** menu option to get visual components on the page and bind it to data in the pageHandler
- EGL pageHandler onPageLoad function to initialize page values
- EGL forward statement to transfer control to another page
- EGL system functions j2eeLib.getSessionAttr and j2eeLib.setSessionAttr to cache session data
- EGL system function serviceLib.setWebEndpoint to dynamically alter the Web service endpoint
- EGL system function mathLib.stringAsDecimal to format the cost column
- EGL record/dataItem specification to assist with data formatting

During the Construction Phase, the EGL developer incorporated the Catalog Manager Page Template to unify the page layouts and navigation. To facilitate development, an EGL code template was defined to provide a shortcut for the functions required to invoke a Web service. The most significant decision of the Construction Phase was where to invoke the Web service. The page that gathered the request data could request the Web service then determine whether the response page or the error page is the next page. An alternate approach is to send the request parameter to the response page, which would request the service in the onPageLoad function. However, the onPageLoad function can neither forward control to another page nor cause an error message to be displayed when the page is first presented to the user. As a result, invocation of Web services will occur on the page that gathers the request data, so the program can handle error conditions.

9.3 Inception Phase

In accordance with the Catalog Manager Software development plan, no EGL specific activities occurred during the Inception Phase.

9.4 Elaboration Phase

This section gives a detailed account of how to recreate the executable architecture developed during the Elaboration Phase. The main activities are invoking a Web service, caching session data, and formatting data.

9.4.1 Web Service invocation

The development environment is WebSphere Developer for zSeries Version 6.0.1.1 with Fix 4. The workbench capabilities for EGL Developer and Web Service Development must be enabled. The **EGL Developer** capability provides the EGL development perspective and its associated development tools. The **Web Service Developer** capability enables validation of wsdl files through the Web Services Explorer. The menu option **Window** → **Preferences** launches the dialog in Figure 9-1 on page 126.

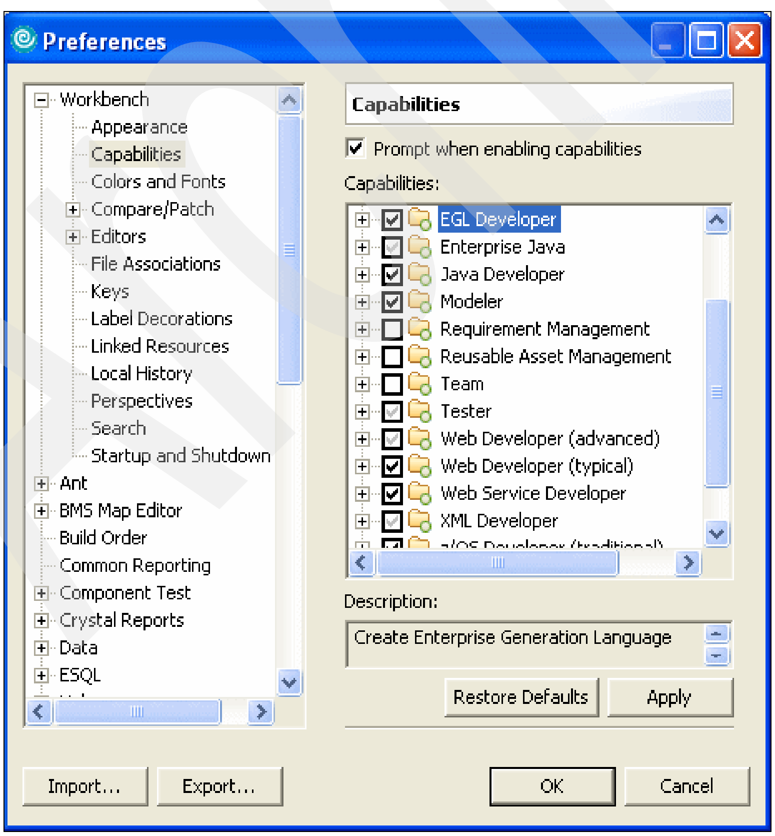


Figure 9-1 WDz Workbench Preference View on Capabilities

Expand **Workbench**, select **Capabilities**, and ensure **EGL Developer** and **Web Service Developer** are checked. Select **OK** and reopen the preference dialog to modify the Default EGL Web Project Feature Choices to include **EGL support with JSF Component Interfaces**. This project feature will enable direct manipulation of the user interface elements using EGL server-side logic rather than client-side JavaScripts. Development will be done in EGL and Web perspectives. Use the menu option **Window** → **Open Perspective** → **Other** and select **EGL** to change the current perspective. Figure 9-2 on page 127 shows the default layout of the EGL perspective.

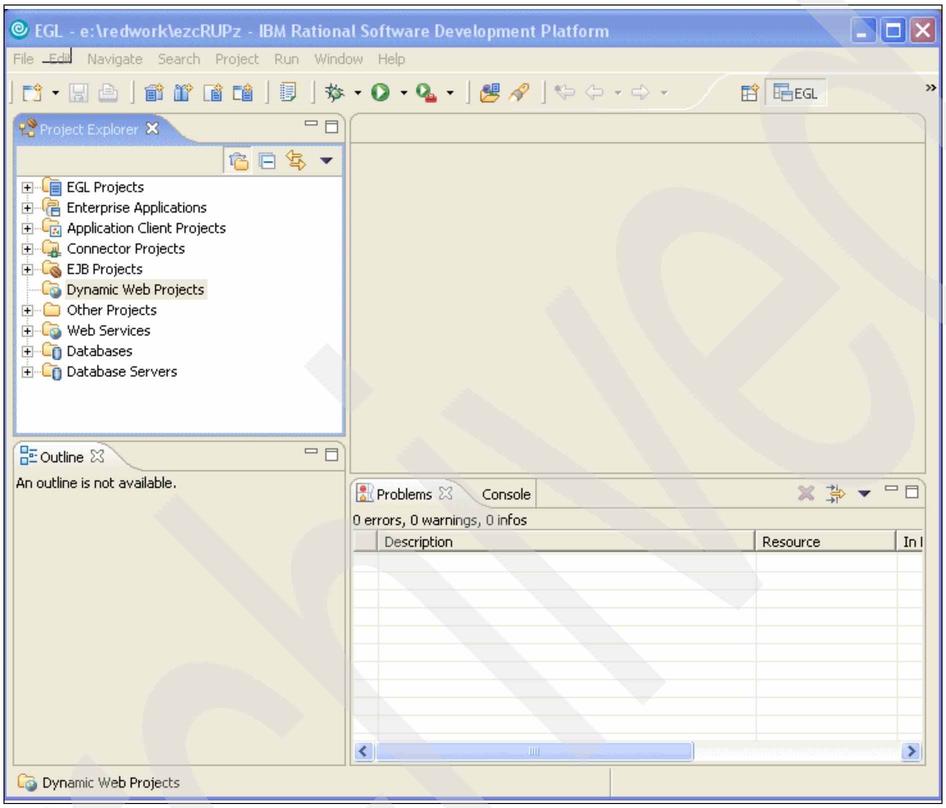


Figure 9-2   EGL Perspective

The default perspective layout has the *Project Explorer* view in the upper left corner. It provides the means to navigate to development artifacts using a hierarchical containment structure. This view shows the relationships among the EGL projects, packages, and source files. All other open views are updated with content when an element is selected or opened in an editor. The *Console* view, in the lower right, shows the output of your program execution and enables you to enter data for a running program. The Console view can show standard output text, standard error text, and standard input text in three colors: by default, standard output text is blue, standard error text is red, and standard input text is green. The *Outline* view, in the lower left, displays an outline of the structured file that is currently open in the editor area in the upper right. The content of the Outline view is editor specific. Syntax errors will be detailed in the Problems view in the lower right.

These are the high level development steps that we followed to inquire about the catalog:

1. Create EGL Web Project.
2. Import WSDL file.
3. Test WSDL file.
4. Generate EGL artifacts from WSDL file.

5. Create the JSP™ page.
6. Customize the EGL pageHandler.
7. Start the server.
8. Test page.

The detailed steps are:

1. On the Project Explorer view, select **New** → **Other** and select **EGL Web Project** on the dialog shown in Figure 9-3.

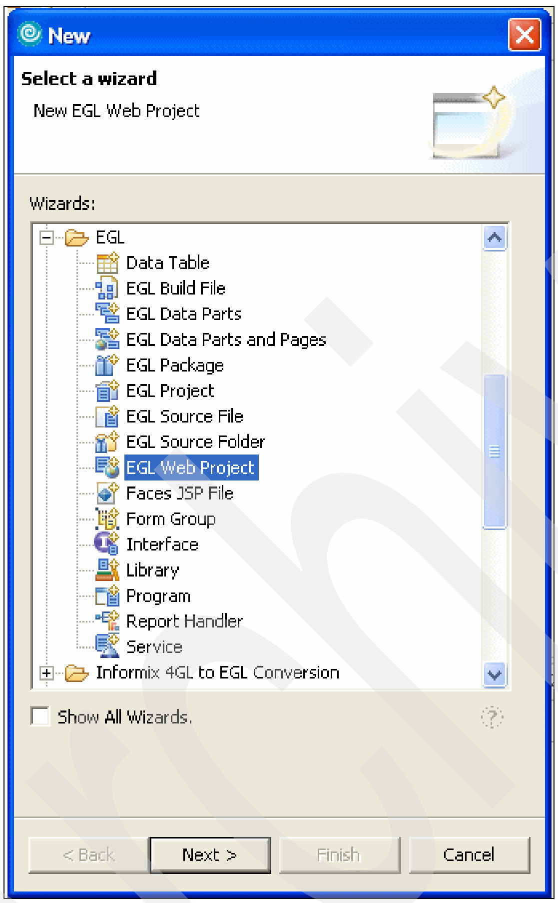


Figure 9-3 Selection page for the New Wizard

2. Select **Next** and enter the project name `EzcCatalogWebProj`. The EGL case study application acronym is *ezc* (*E*GL RUP for *z* *C*atalog Manager). Every development artifact will have this acronym as a prefix to its name. Figure 9-4 on page 129 shows the appropriate settings for the New EGL Web Project Wizard.

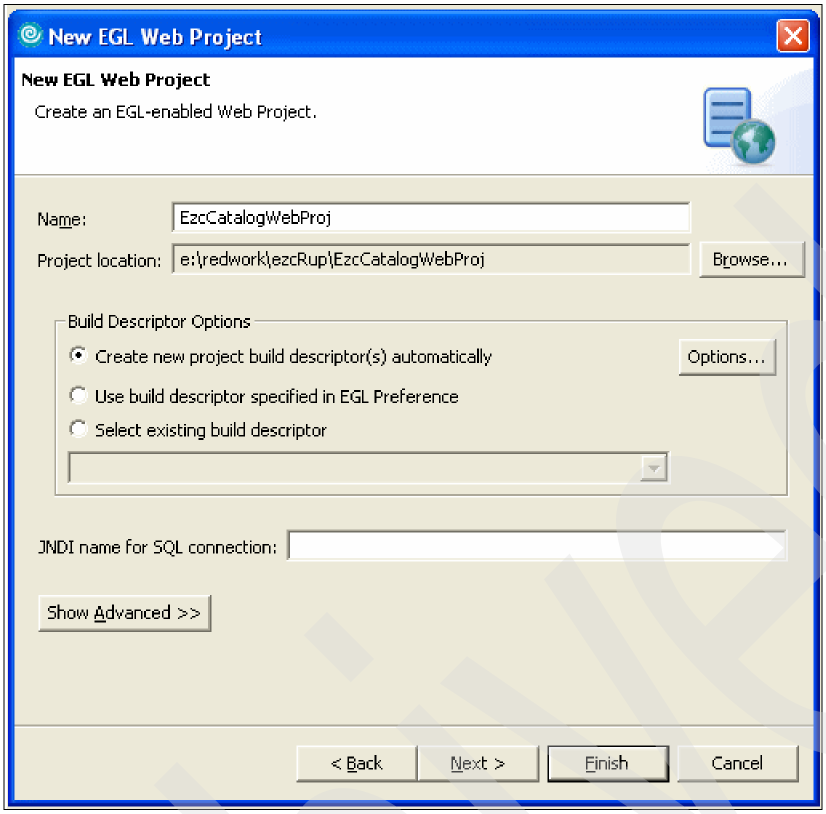


Figure 9-4 New EGL Web Project Wizard

3. On the first page of the new EGL Web Project wizard, specify the project name and ensure that **Create new project build descriptor(s) automatically** is selected. Use **Next** to advance to the next page shown in Figure 9-5 on page 130.

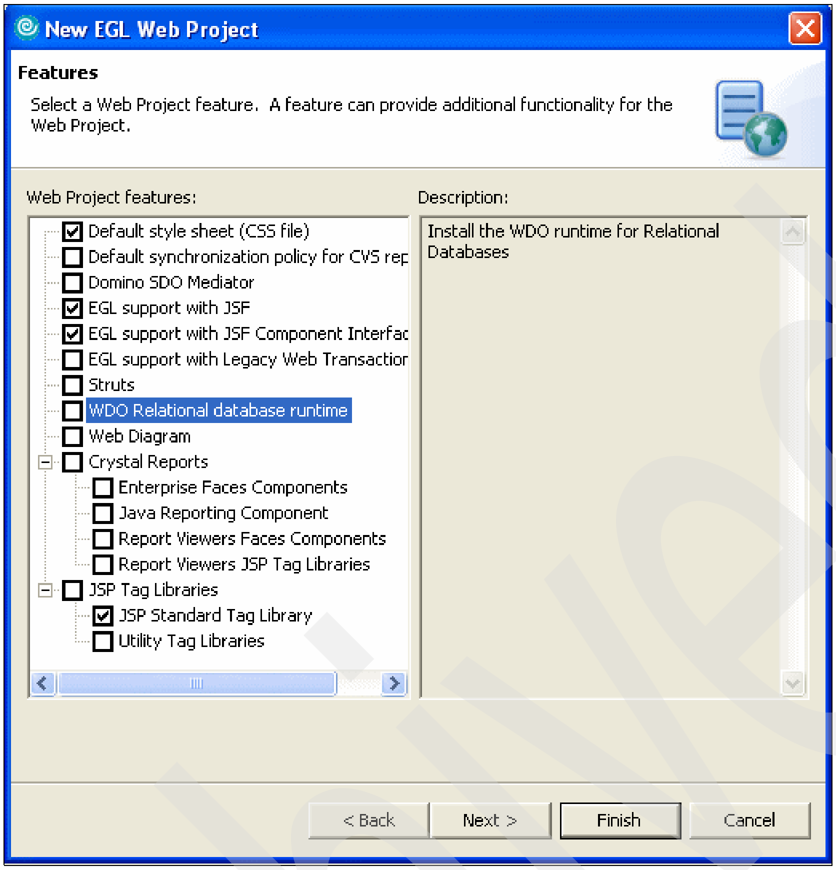


Figure 9-5 New EGL Web Project Wizard's Features page

4. Deselect the **WDO Relational database runtime** Web project feature, because the data access will occur using the Web services. Select **Finish** to create the Web project. Accept the Confirm Perspective Switch prompt.
5. Navigate to the WEB-INF source folder under the newly created Web project's WebContent folder, select the folder, then select **New** → **Folder** from the Project Explorer context menu. If that option is not available, select **New** → **Other** and expand **Simple** on the New Wizard. Name the folder `wsdl`. Import the wsdl files from the workspace where you imported the sample application. Figure 9-6 on page 131 illustrates the file Import dialog.

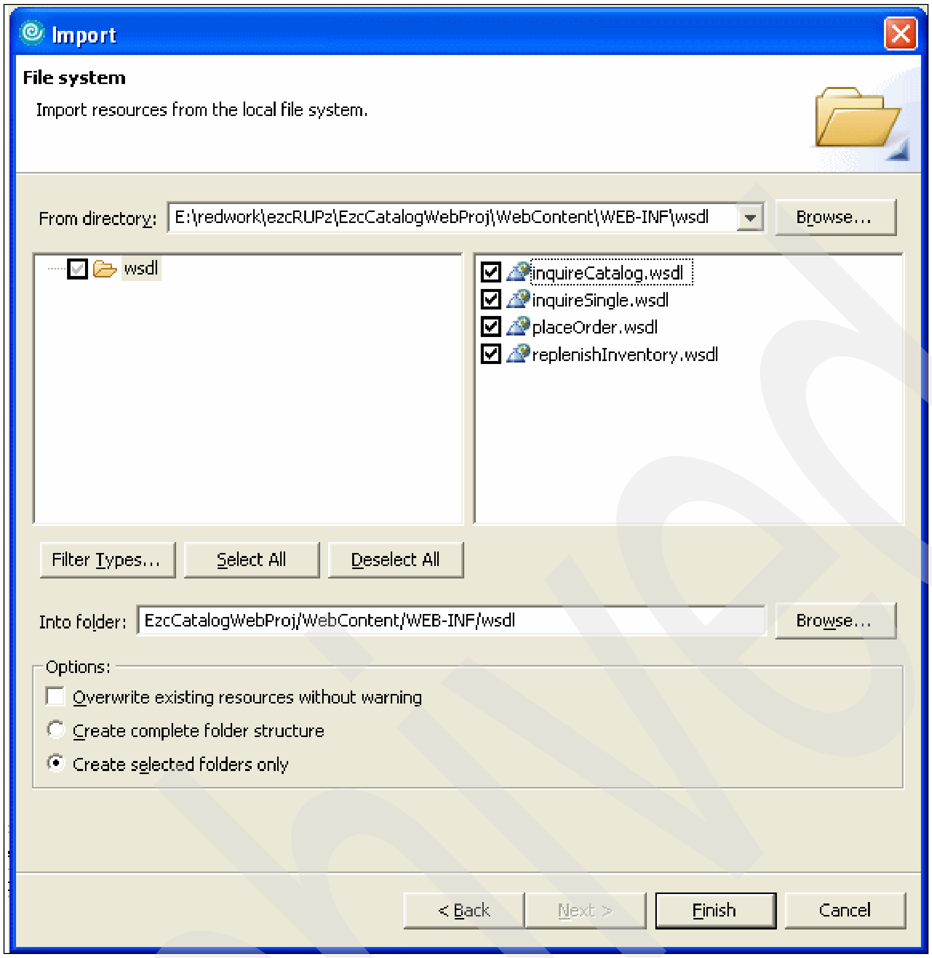


Figure 9-6 File System File Wizard

6. Locate the inquireCatalog.wsdl file in the Project Explorer view and use the context menu option **Web Services** → **Test with Web Services Explorer**. If this menu option is missing, you do not have the Web Services Developer Capability selected in the workbench preferences. Figure 9-7 on page 132 shows the resulting browser.

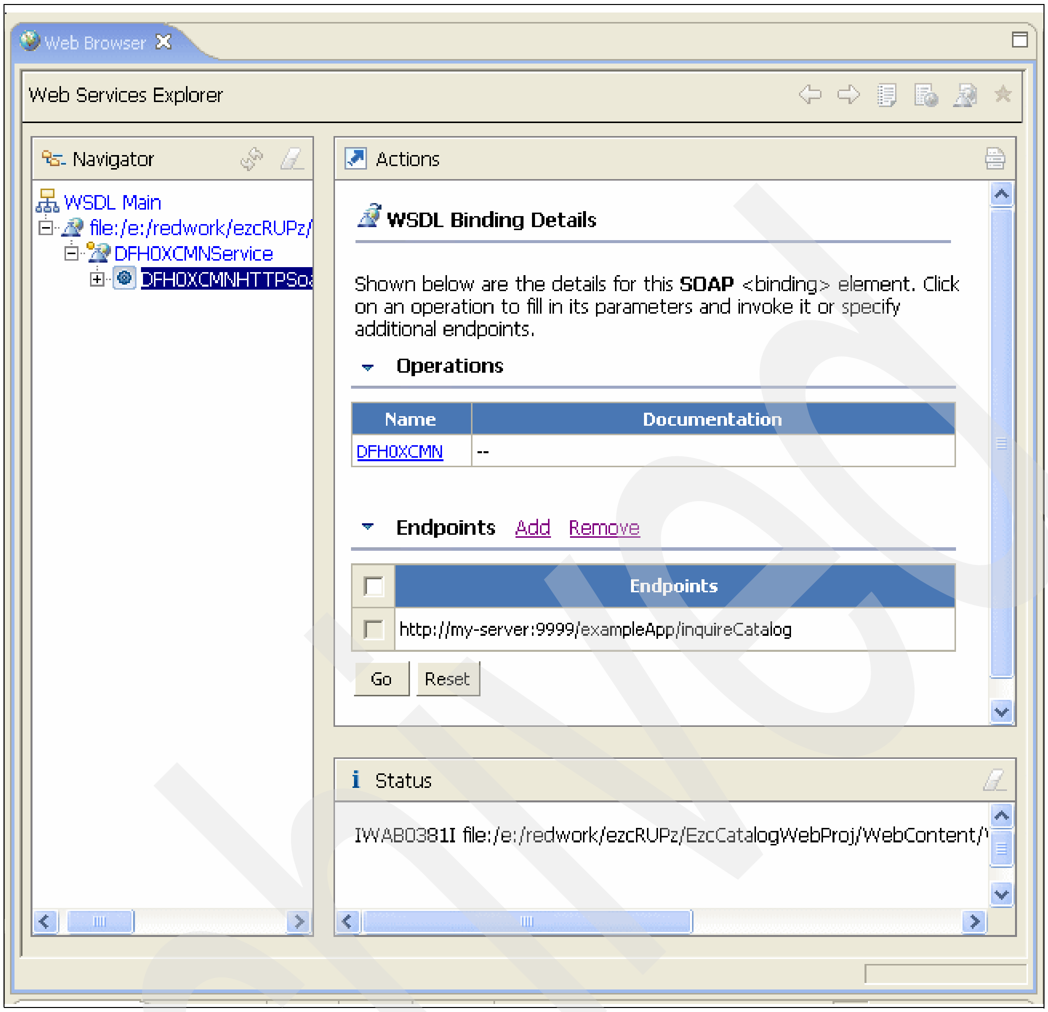


Figure 9-7 Web Services Explorer WSDL Binding Details

7. On the Web Services Explorer view, add a new endpoint and modify it to reference the location of your Web services. Select the check box beside the new service endpoint and click **Go**. Recheck the new endpoint and select the Operation **DFH0XCMN**. The Web Explorer will show the request parameters for the Web service. Ensure the status message says the endpoints were updated successfully. Initialize `ca_request_id` to `01INQC`, `ca_list_start_ref` to `50` (or any integer value less than 70), and all the remaining fields to zero. You might want to validate your connectivity to the Web services by using `inquireSingle.wsdl`, because it has fewer fields to initialize. Figure 9-8 on page 133 shows the Web Services Explorer with entry fields for the request data.

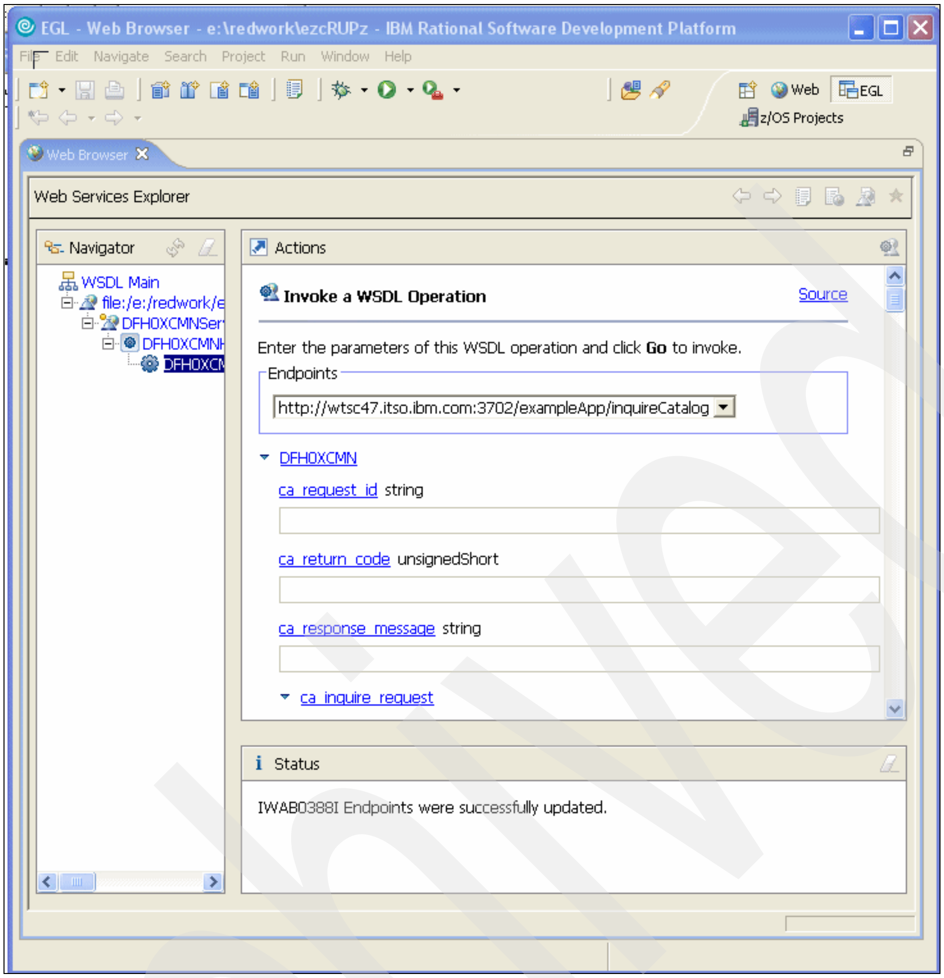


Figure 9-8 Web Services Explorer: Invoke a WSDL Action

After all the data fields have been initialized, click **Go**. If the status message is "`IWAB0383E Error validate RequestPart`", look for any request fields marked with a red asterisk (*). Most likely, it is an unsignedShort field that was not initialized to zero. If successful, the status looks like Figure 9-9 on page 134.

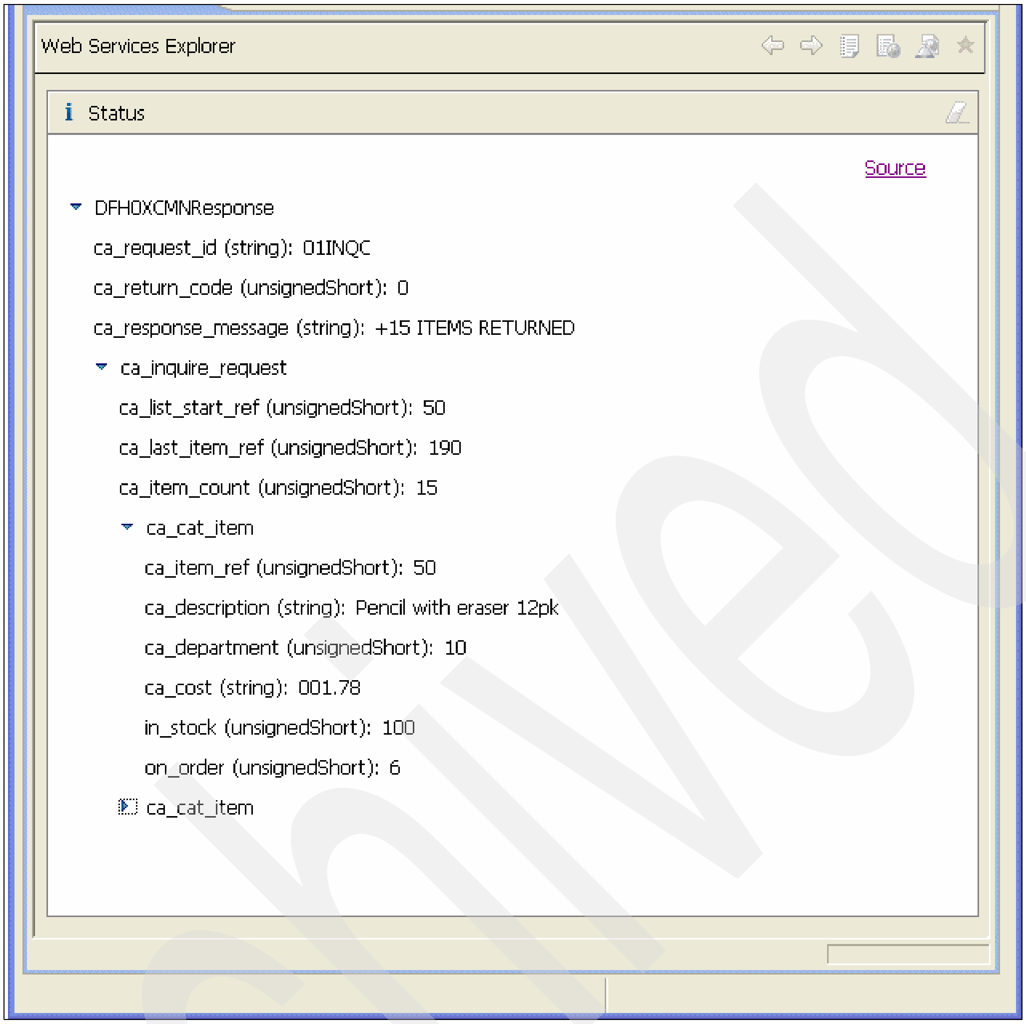


Figure 9-9 Web Services Explorer Status content for successful WSDL action

If you modify the `ca_list_start_ref` to `75` and request the service, you get the soap error shown in Figure 9-10 on page 135.

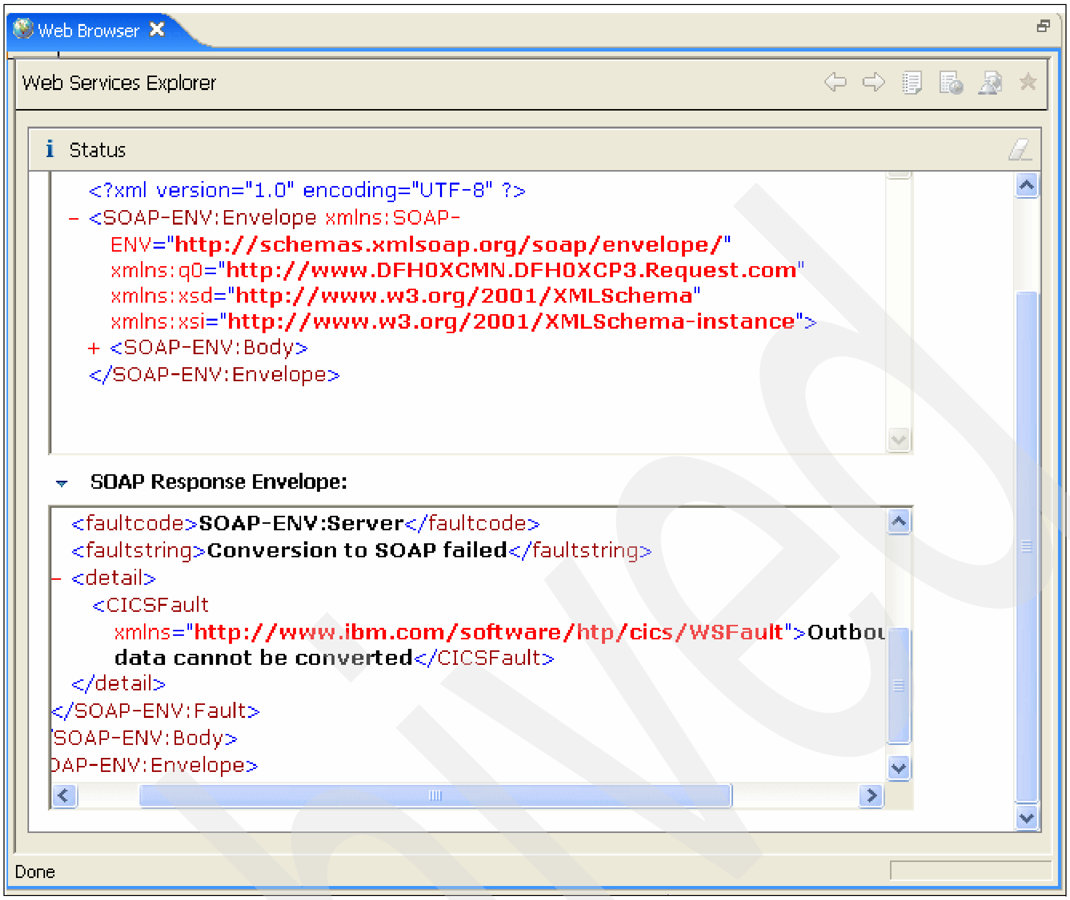


Figure 9-10 Web Services Explorer Status content for SOAP error

Using the Web Services Explorer quickly lets you validate that you have access to the service, lets you validate that the service is up and running, and allows you to validate the request data.

8. The next step is to create the EGL service binding library and the EGL interfaces that are necessary to invoke the inquire catalog Web service. Select the **inquireCatalog.wsdl** file in the Project Explorer view, select the menu option **Create EGL Interfaces and Binding Library**, and accept the wizard defaults by selecting **Finish**. Figure 9-11 on page 136 shows the EGL perspective after the EGL service components have been generated.

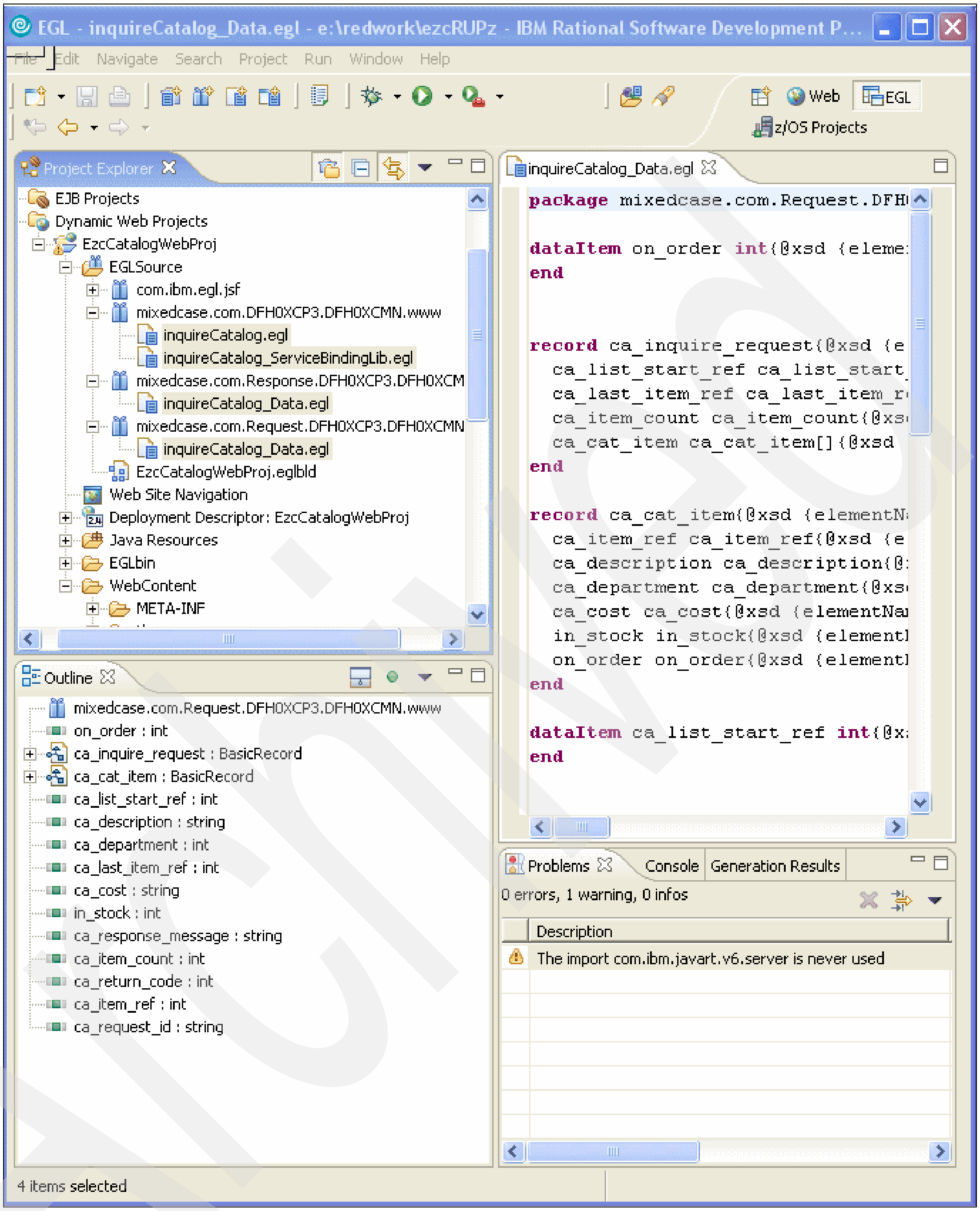


Figure 9-11 EGL Perspective after generation from the WSDL file

9. The EGL editor is opened on the file that defines the Web service request variables. The Outline view shows the structural list of the data elements and records defined to represent the Web service request. Switch to the Web perspective and create a Java Server Page (JSP) to gather request data and invoke the service. From the Project Explorer view, highlight the **WebContent** folder and select **New** → **Faces JSP File.** If that option is not available, select **New** → **Other** and expand **EGL** on the New Wizard. On the new Faces JSP wizard, name the file `ezcInquirePrototypePage` and accept the default properties. Verify that you are editing the JSP file in the Web perspective. Figure 9-12 on page 137 shows the development environment.

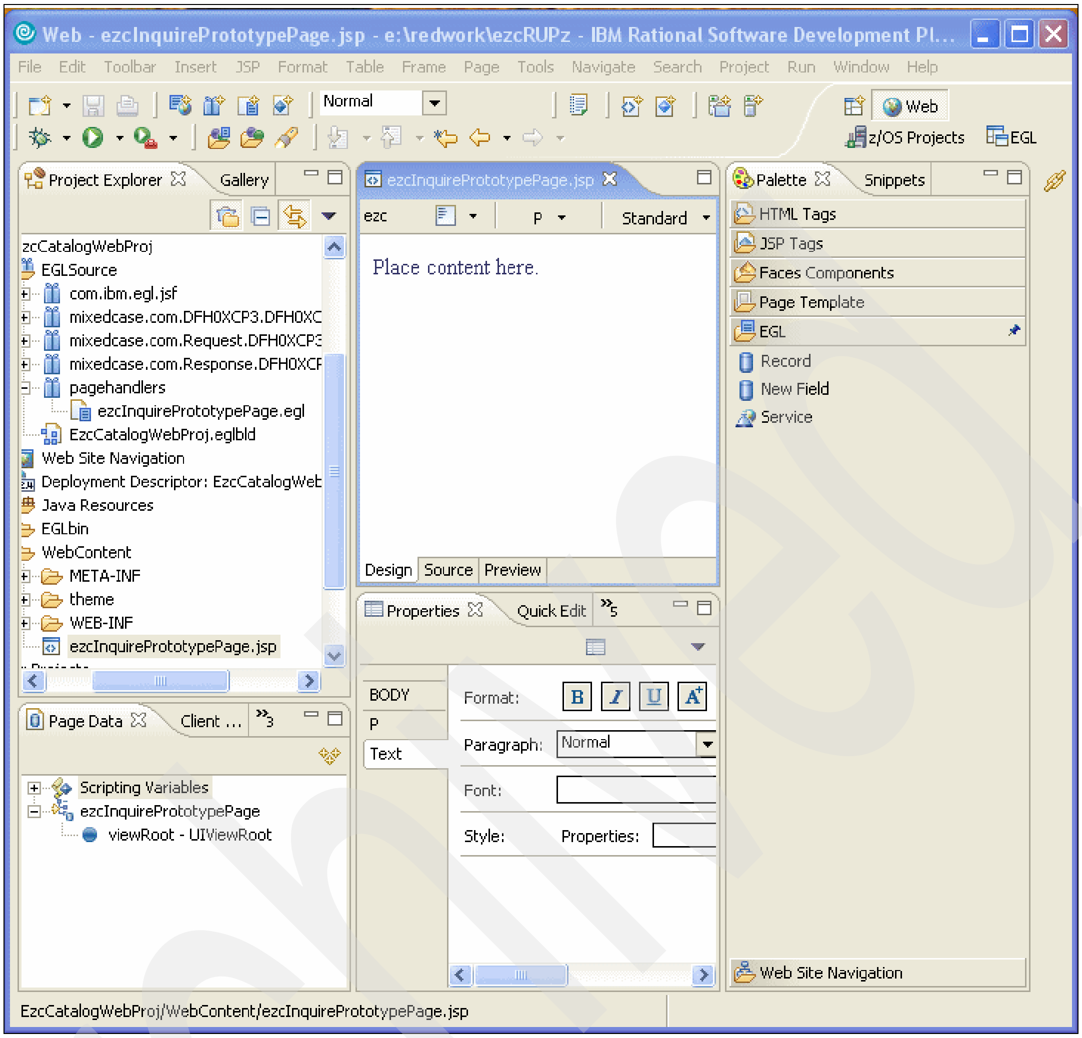


Figure 9-12 Web Perspective editing ezcInquirePrototypePage.jsp

The new Faces JSP file wizard will create the JSP file and an associated EGL pageHandler file of the same name. The pageHandler is written in EGL and controls a user's runtime interaction with a Web page. From a pageHandler, you can assign data values for submission to a JSP file, change the data returned from the user or from a called program, and forward it to another JSP file. The PageHandler includes:

- An OnPageLoad function, which is invoked the first time that the JSP renders the Web page
- A set of event handler functions, each of which is invoked in response to a specific user action, such as clicking a button
- Optionally, validation functions that are used to validate Web page input fields
- Private functions that can be invoked only by the PageHandler functions

It is important to note that the OnPageLoad function can neither forward control to another page nor cause an error message to be displayed when the page is first presented to the user. For more details, use the **Help** → **Search** toolbar menu option and search on "PageHandler runtime scenarios".

In Figure 9-12, the content area shows the Page Designer editor opened on the newly created JSP file in design mode. The *Page Data* view, the *Properties* view, and the *Palette* view are the most commonly used auxiliary views to declaratively code the JSP. The Page Data view shows all the data objects available on the JSP. The Properties view is used to update the properties of the selected object. The Palette view contains drawers of items that can be dragged and dropped onto the Page Designer. In Figure 9-12 on page 137, the EGL drawer is open to reveal that EGL provides record, new field, and service items for a JSP. Use “Page Designer support for EGL” as the help search key for a detailed introduction to the Page Designer.

10. To continue developing the prototype, select the Service item and drag it over the **Design** tab of the Page Designer. Save the modified JSP, which triggers an automatic generation of its pageHandler. The generation launched the Generation Results view to display the results of each generated file. Unfortunately, the pageHandler failed to generate cleanly. Figure 9-13 has the generation errors associated with the pageHandler.

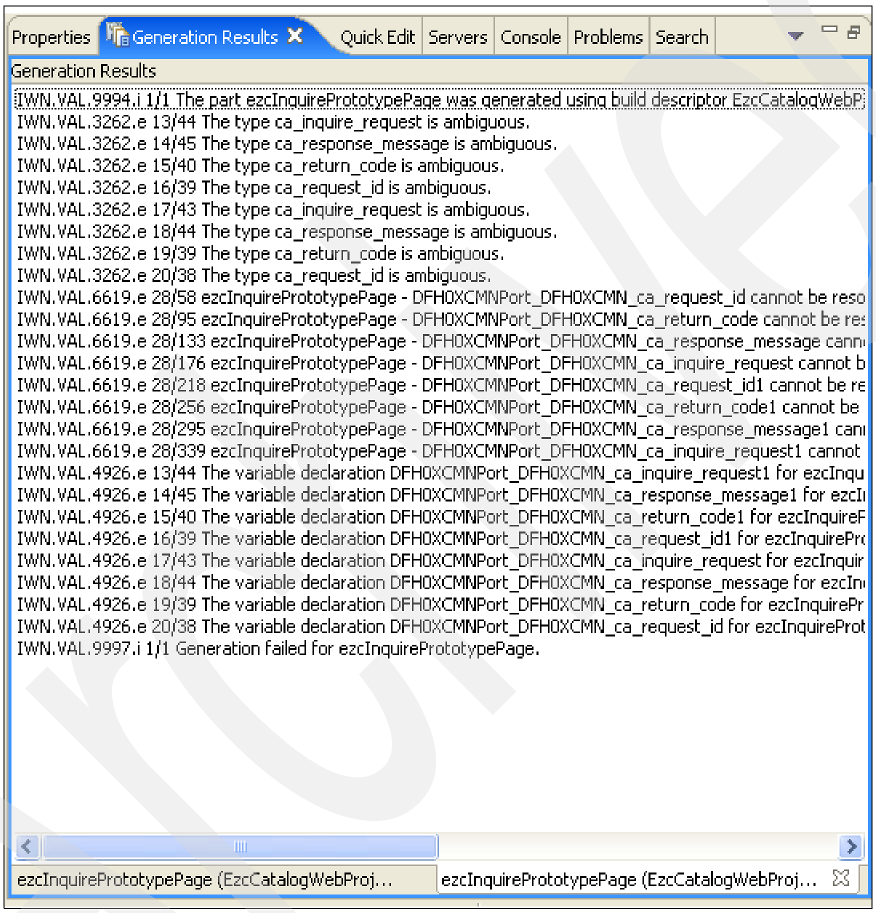


Figure 9-13 Generation Errors for ezcInquirePrototypePage.egl

11. Double-click on any error message in the generation results. The offending file will be opened to the line of source code that produced the problem. Double-click on the message “`The type ca_request_id is ambiguous`”, which opens the pageHandler view with the reference to `ca_request_id` highlighted. Press F3 to open the definition of the type. Because the type is ambiguous, you will get the dialog in Figure 9-14 on page 139.

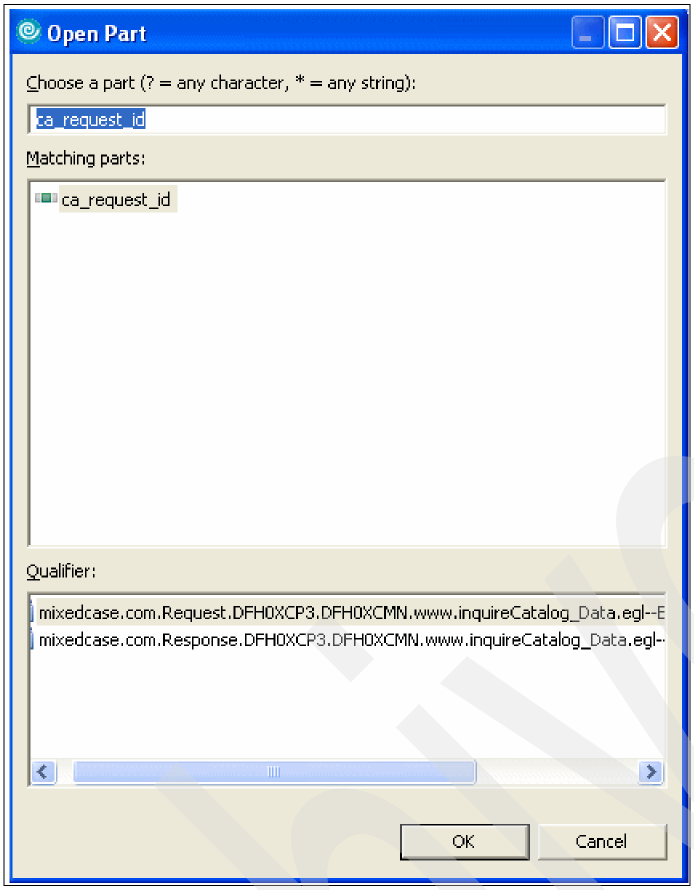


Figure 9-14 Open Part selection dialog for ca_request_id

12. Figure 9-14 shows that ca_request_id is defined in the request and the response packages. The pageHandler must use both the request and the response definition of ca_request_id. Without fully qualifying the type definitions, the pageHandler will have compiler errors as shown in Figure 9-15 on page 140.

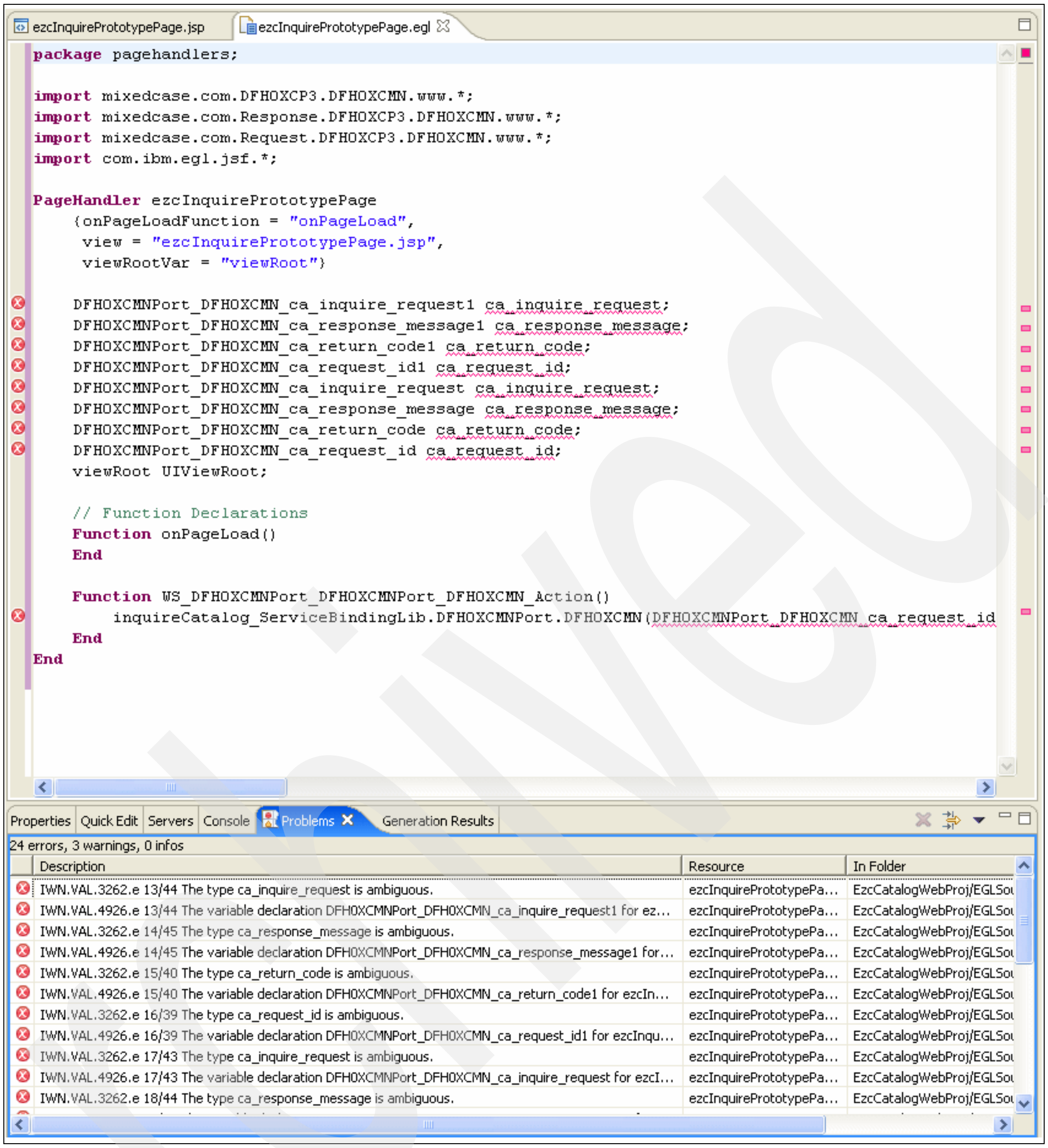


Figure 9-15 Source code and compiler messages for ezcInquirePrototypePage.egl

13.Now you have enough information to realize that the Web service used the same data structure for the request and the response. The generated code produces a request package and a response package with the same data item names. Therefore, the types need to be fully qualified in the pageHandler. For clarity and simplicity, the generated variable names have been altered to start with `input_` and `output_` rather than `DFHOXCMNPort_DFHOXCMN`. The name of the generated function has also been modified to reflect the Web service that is being invoked. Figure 9-16 on page 141 shows the modified pageHandler, which resolves the ambiguous references.

```
package pagehandlers;

import mixedcase.com.DFHOXCP3.DFHOXCMN.www.*;
import mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.*;
import mixedcase.com.Request.DFHOXCP3.DFHOXCMN.www.*;
import com.ibm.egl.jsf.*;

PageHandler ezcInquirePrototypePage
    (onPageLoadFunction = "onPageLoad",
     view = "ezcInquirePrototypePage.jsp",
     viewRootVar = "viewRoot")

    // Web Service request/input variables
    input_ca_inquire_request mixedcase.com.Request.DFHOXCP3.DFHOXCMN.www.ca_inquire_request;
    input_ca_response_message mixedcase.com.Request.DFHOXCP3.DFHOXCMN.www.ca_response_message;
    input_ca_resturn_code mixedcase.com.Request.DFHOXCP3.DFHOXCMN.www.ca_return_code;
    input_ca_request_id mixedcase.com.Request.DFHOXCP3.DFHOXCMN.www.ca_request_id;

    // Web Service response/output varaiables
    output_ca_inquire_request mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.ca_inquire_request;
    output_ca_response_message mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.ca_response_message;
    output_ca_return_code mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.ca_return_code;
    output_ca_request_id mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.ca_request_id;

    viewRoot UIViewRoot;

    // Function Declarations
    Function onPageLoad()
    End

    // V1.0 invoke the Web Service
    Function invokeInquireCatalogServiceAction()
            inquireCatalog_ServiceBindingLib.DFHOXCMNPort.DFHOXCMN(
                input_ca_request_id,
                input_ca_resturn_code,
                input_ca_response_message,
                input_ca_inquire_request,
                output_ca_request_id,
                output_ca_return_code,
                output_ca_response_message,
                output_ca_inquire_request);
    End
End
```

Figure 9-16 Source code for ezcInquirePrototypePage.egl without ambiguous references

14.Set the initial values for the required input fields in the OnPageLoad function. Take advantage of code assist to get correct names without typographical errors. To invoke code assist, start typing a variable name, press Ctrl+Spacebar, and code assist prompts you with possible completions. If code assist does not offer suggestions, you might have syntax errors. Add a new function, `requestInquireCatalogAction`, that will set the endpoint and request the Web service. The new function and the modified onPageLoad function are shown in Figure 9-17.

```
    // Function Declarations
    Function onPageLoad()
        input_ca_request_id = "01INQC";
        input_ca_inquire_request.ca_list_start_ref = 50;
    End

    // V1.0 Set the endpoint and invoke service ... assuming happy path only
    Function requestInquireCatalogAction()
        // Verify that hardcoded endpoint is correct
        ezcEndpoint string = "http://wtsc47.itso.ibm.com:3702/exampleApp/inquireCatalog";
        sysLib.writeStdout("Setting endpoint ... " + ezcEndpoint);
        serviceLib.setWebEndpoint(inquireCatalog_serviceBindingLib.DFHOXCMNPort, ezcEndpoint);
        sysLib.writeStdout("Calling web service");
        invokeInquireCatalogServiceAction();
        sysLib.writeStdout("Done calling web service");
    End
```

Figure 9-17 New source code for ezcInquirePrototypePage.egl to invoke the Web service

15. The first version of this function will have the endpoint hardcoded with the appropriate endpoint. The function uses the system library function named sysLib.writeStdout to write out information to the console. This provides a simple means to get trace and debug information without the time required to run the Web page in debug mode. EGL provides the system function, serviceLib.setWebEndpoint, to dynamically alter the Web service endpoint. Note the application is making no attempt to handle any error conditions. The remaining steps are to update the JSP with the input data, invoke the Web service, and display the results. Open the Page Designer on the JSP file; the editor needs to be on the Design view in order to drag and drop elements onto the editor. Select the text **Place content here**, replace it with a header `Inquire Catalog Prototype`, and press Enter. To add the list start reference input field, a label, and a submit button, go to the **Page Data** view (usually found in the lower left corner of the Web perspective) and select the necessary data object as illustrated in Figure 9-18.

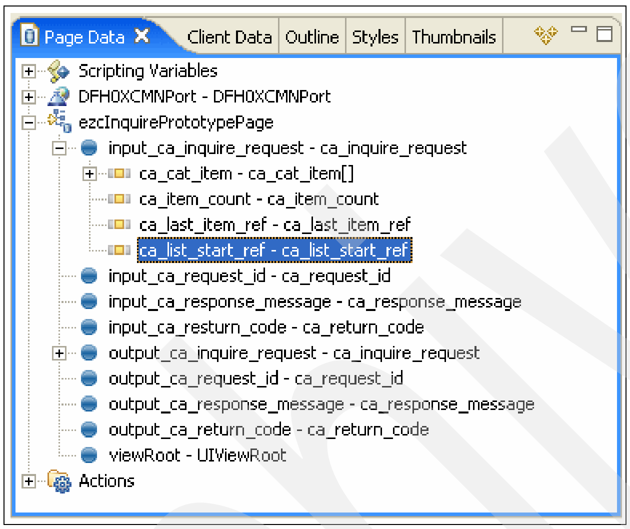


Figure 9-18 Page Data View with the service request variable selected

16. Select the **Insert New Controls for "ca_list_start_ref"** from the Page Data view context menu. On the insert control wizard, you can change labels and the order of the components. Select **Creating a new record** to render a submit button. Verify that the options dialog has **Create submit button** checked and append a colon to each label not selected. Figure 9-19 on page 143 shows the wizard settings used to create the Catalog Manager EGL example.

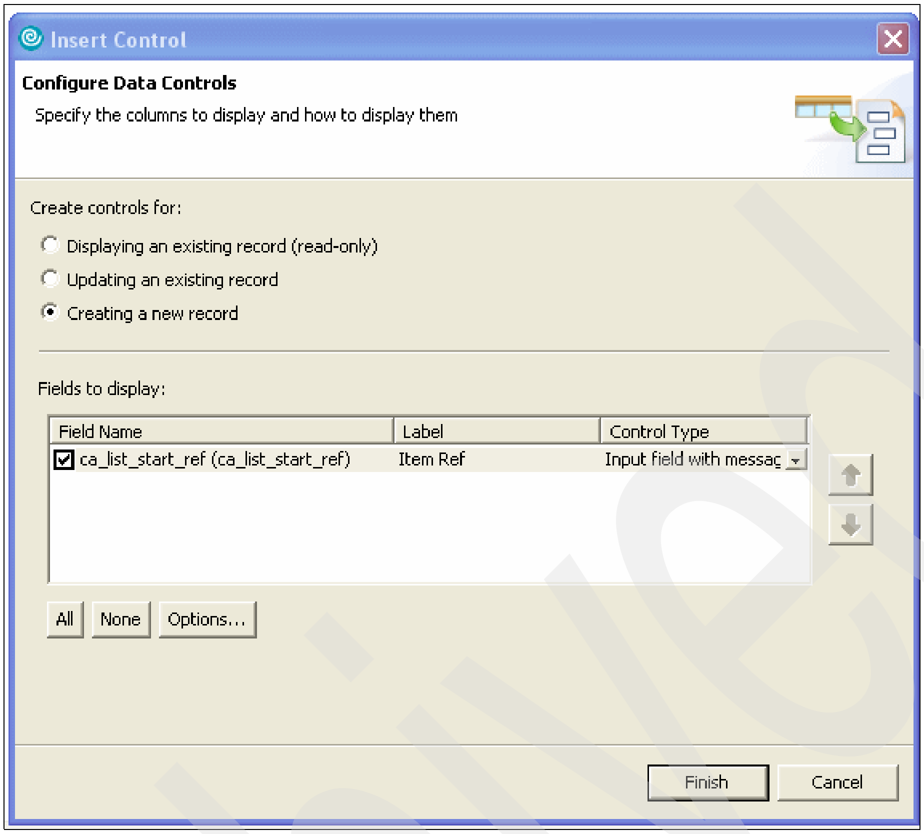


Figure 9-19 Insert Control Wizard for service request variable

17.This action will create an HTML table to control the layout of the JSP user-interface components themselves and the EGL bindings, which are relationships between components and data or logic. In this case, one input text field is added to the page and it has been bound to the originally selected EGL data variable. Figure 9-20 shows the Design tab of the Page Designer.

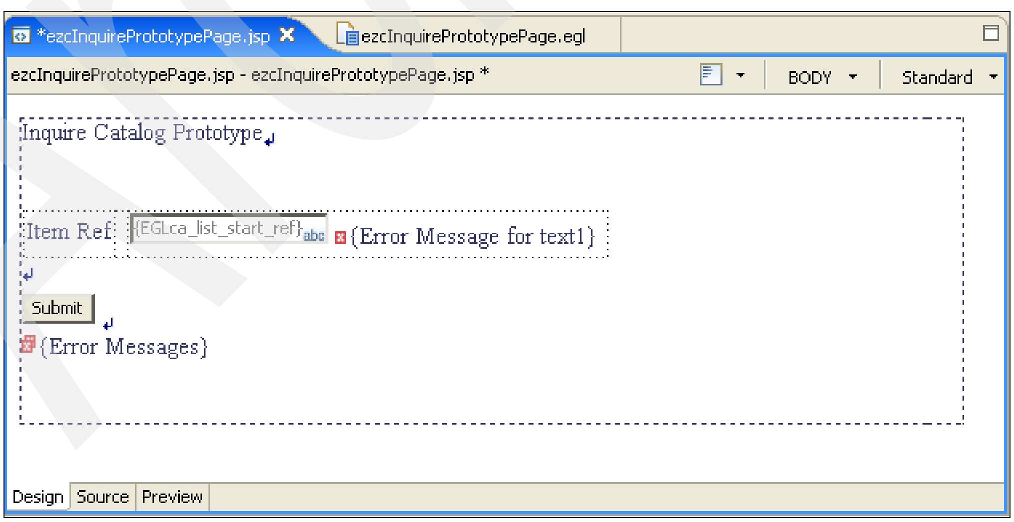


Figure 9-20 Page Designer's Design tab for ezcInquirePrototypePage.jsp

18. Select the **Preview** tab to preview what the page will look at run time. In Figure 9-21, the dashed lines around the user-interface components no longer appear, because they are an editing convenience.

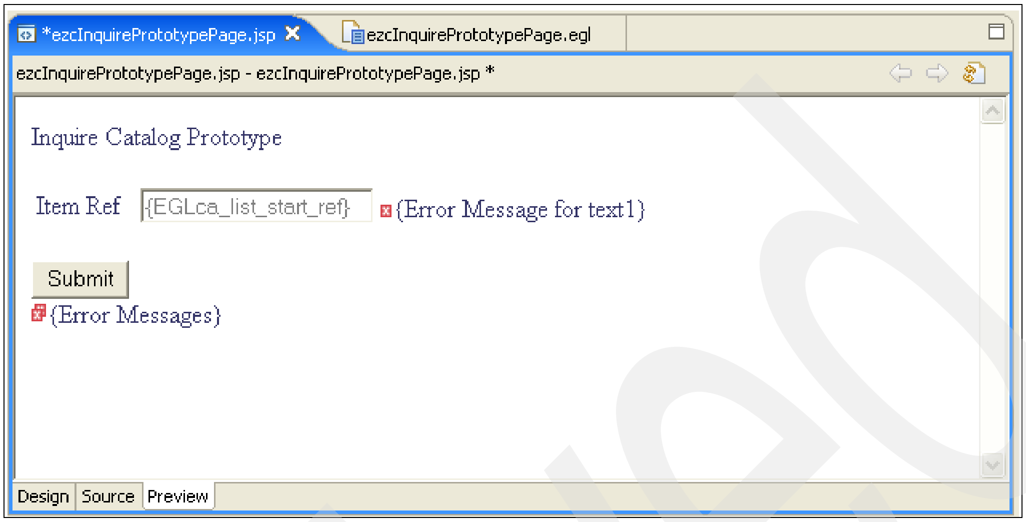


Figure 9-21 Page Designer's Preview tab for ezcInquirePrototypePage.jsp

19. Select the **Source** tab to see the raw source code as shown in Figure 9-22.

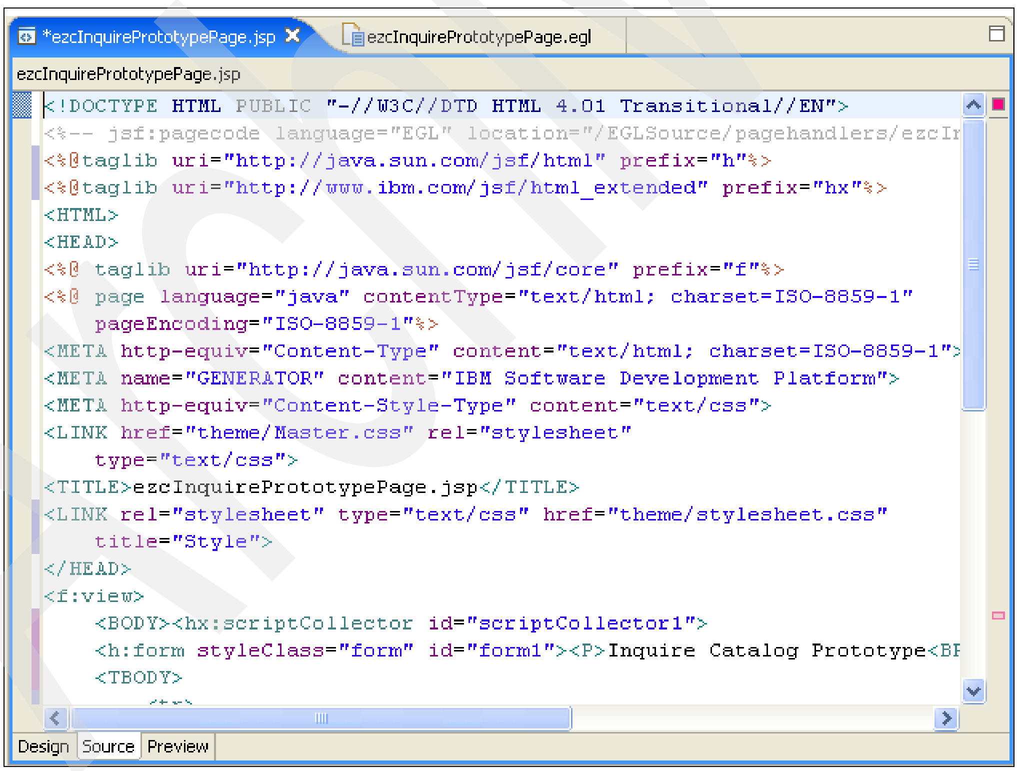


Figure 9-22 Page Designer's Source tab for ezcInquirePrototypePage.jsp

20. The submit button needs to be bound to an action. This is done by highlighting the function in the Page Data view and dropping it on the submit button. Select **requestInquireCatalogAction** from the Page Data view and drag it to the **Submit** button and drop it. Now, when the button is selected by the user, the bound pageHandler function

will be executed. Use the **insert controls** menu option to add the output data on the page. Figure 9-23 shows the **ca_cat_item** and **output_ca_response_message** selected in the Page Data view.

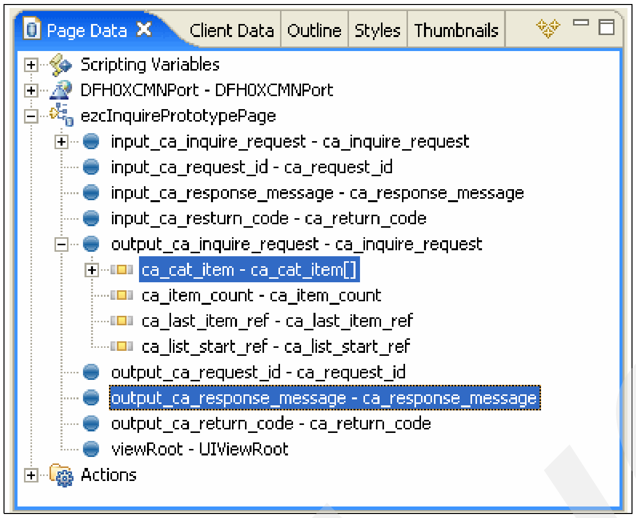


Figure 9-23 Page Data View with the service response variables selected

21. Make sure that **Displaying an existing record (read-only)** is selected, so that all fields have the control type of output. Figure 9-24 shows the setting for the Insert Control dialog for the response variables.

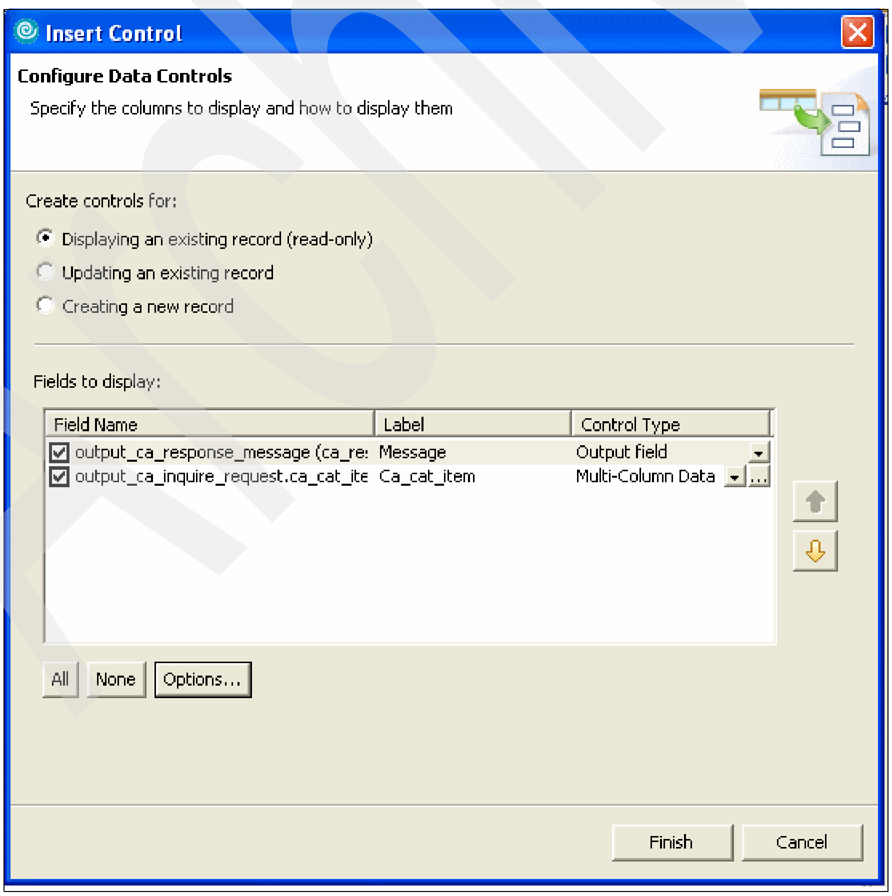


Figure 9-24 Insert Control Wizard for service response variable

Figure 9-25 shows the current layout of the ezcInquirePrototypePage.

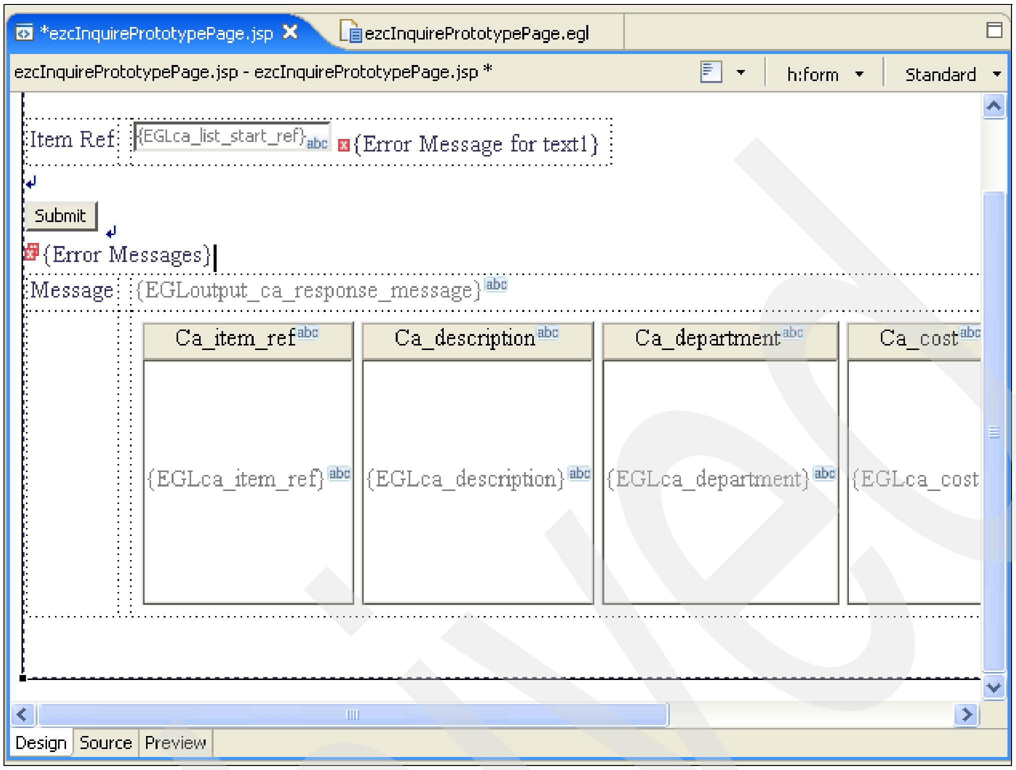


Figure 9-25 Completed ezcInquirePrototypePage.jsp

22.Save your changes and test the page. Select the JSP in the Project Explorer view and select **Run on Server**. Select **Finish** on the Select Server dialog if you are prompted. If you are required to define a server, you do not have the integrated WebSphere test environment feature installed.

23.Be patient while the server starts. Watch the status of the server in the Servers view. The starting status looks like Figure 9-26.

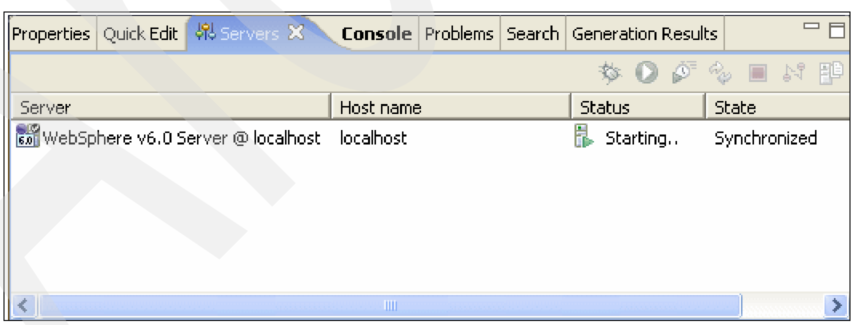


Figure 9-26 Servers view while starting a server

24.Do not attempt to do anything else in WDz until the server has started. Figure 9-27 on page 147 shows the started status.

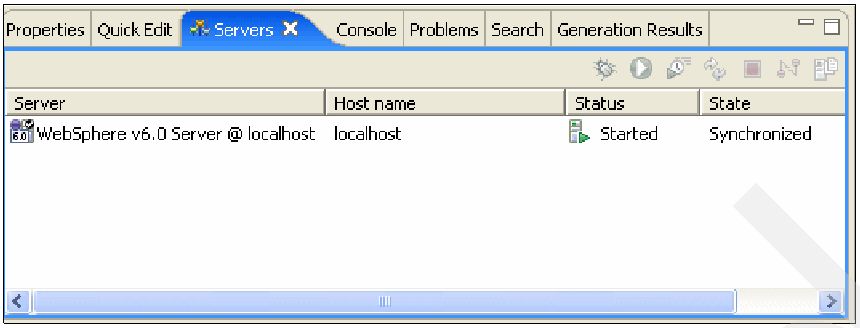


Figure 9-27 Servers view with server in Started state

25. Here is the content of the Console view after the server starts. The console has informational (blue) messages as well as error (red) messages. This is the first place to look if your server does not start. If your server status is started, do not worry about red messages. Figure 9-28 shows the Console content.

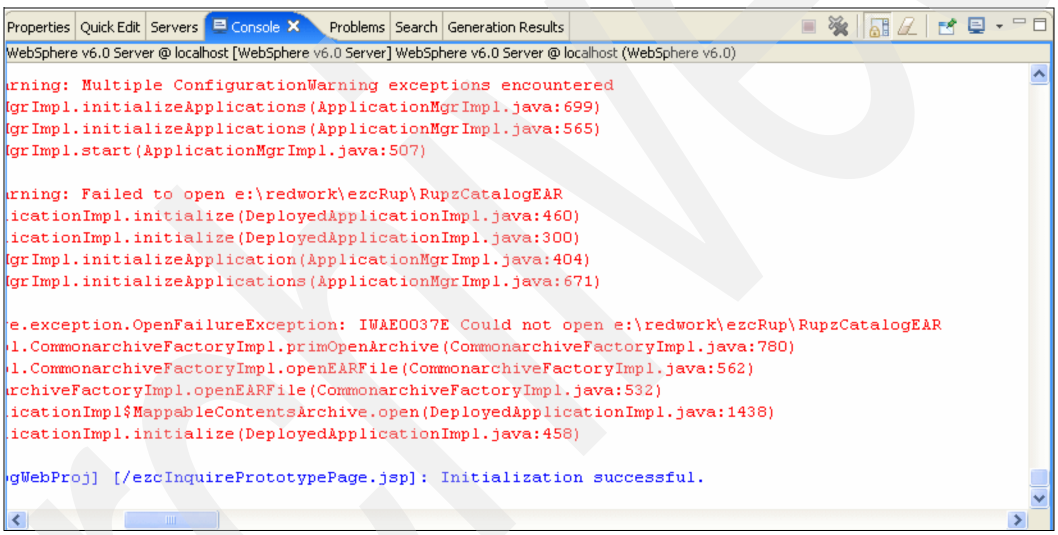


Figure 9-28 Console messages while starting the server

26. It is helpful to clear the console before using the Web page so that the console content is limited to the execution of the application. Recall that the writeStdout statements included in the pageHandler show here. Figure 9-29 show the runtime rendering of the JSP page.

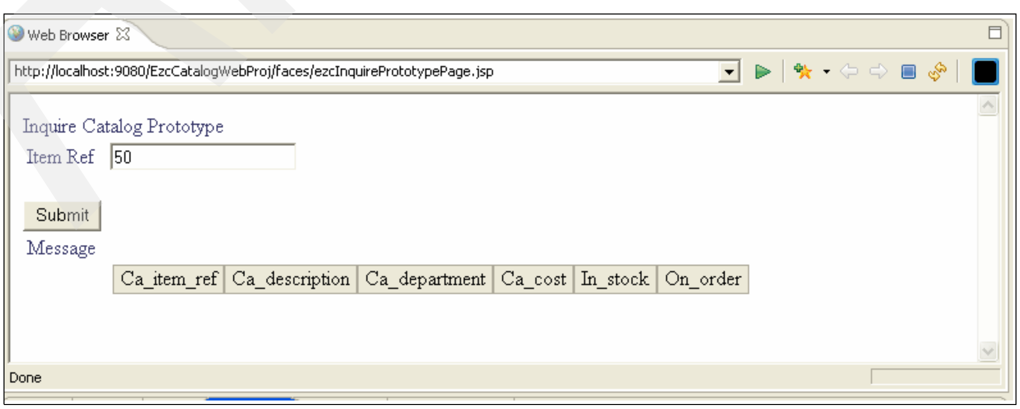


Figure 9-29 Runtime rendering of ezcInquirePrototypePage.jsp

27. The data item initialized in the pageHandler has been rendered onto the page. Test with the default value 50 by selecting **Submit**. Figure 9-30 is the state of the page after a successful invocation of the Web service.

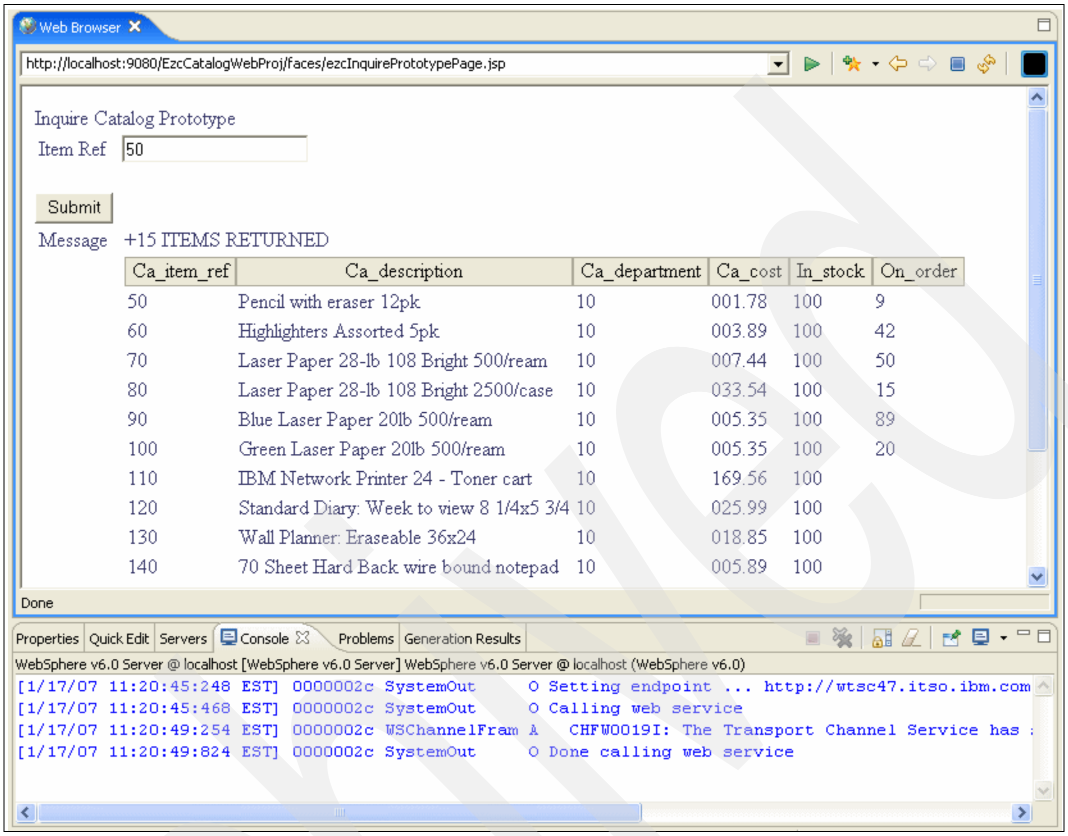


Figure 9-30 Response data and console output for inquire catalog service

28. Now, enter 75 as the next test. The invocation of the Web service results in an exception that EGL logs into the console, and the page is rendered in its initial state as shown in Figure 9-31 on page 149.

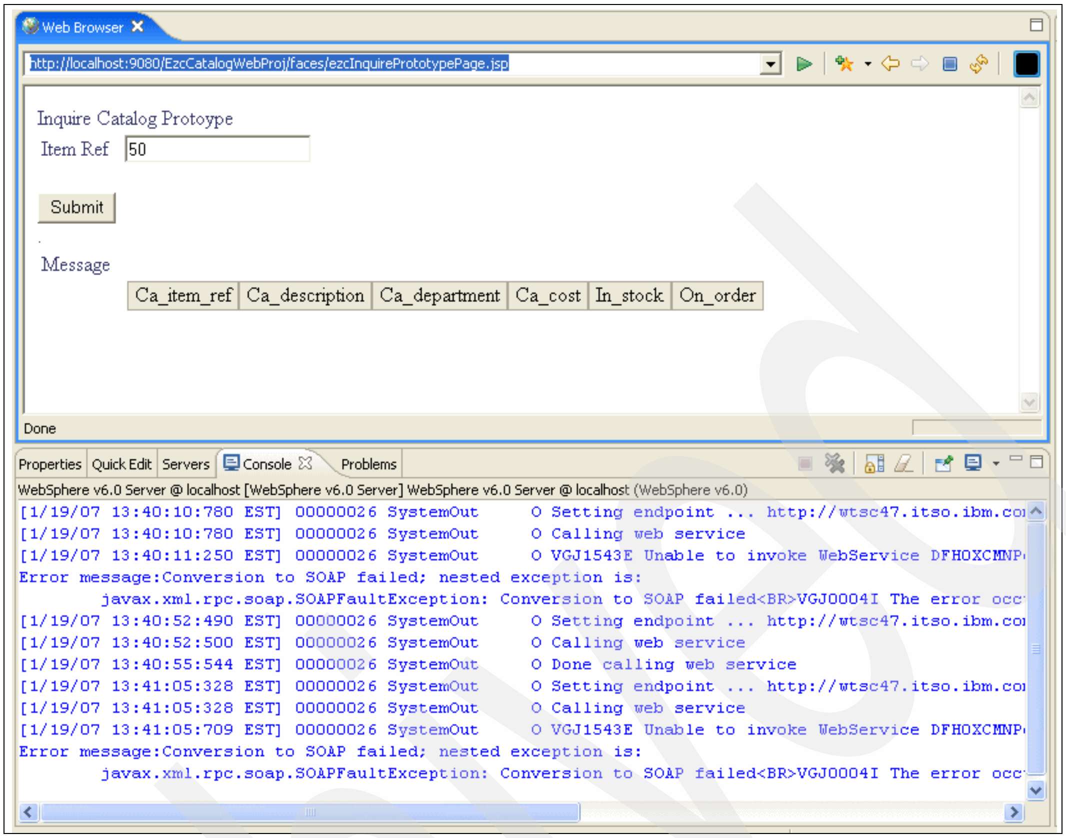


Figure 9-31 Inquire catalog response and console output with exception

Figure 9-30 on page 148 and Figure 9-31 illustrate the default rendering of the Web service data to the page and tracing messages to the console. EGL did the majority of the work. The JSP and pageHandler need to be tweaked to handle error conditions and to resolve cosmetic issues, specifically the format of the cost data and zero numeric data. With very little development effort, the prototype validates that the Web application can gather request parameters, change the Web service endpoint, and display the response data. Error handling, data movement between pages, and data formatting need to be resolved in order to complete the elaboration prototype use cases.

9.4.2 Error handling

At this point, the configured endpoint is hardcoded in the inquire catalog pageHandler and the user interface is limited to a single page. The prototype needs to handle error and exception conditions, to be extended to handle basic navigation between pages, and to cache the user-provided Web service endpoint. These are the steps:

1. For error handling, you need to create a simple JSP page to display application errors and one page to show system exception details. Create a JSP page named `ezcErrorDetailsPrototypePage` with the header `An Error Has Occurred`. Add a string variable named `errorDetails` to the pageHandler. Modify the onPageLoad function to accept a string parameter named `inError` and initialize the pageHandler variable `errorDetails` to the parameter. Figure 9-32 on page 150 is the content of the pageHandler.

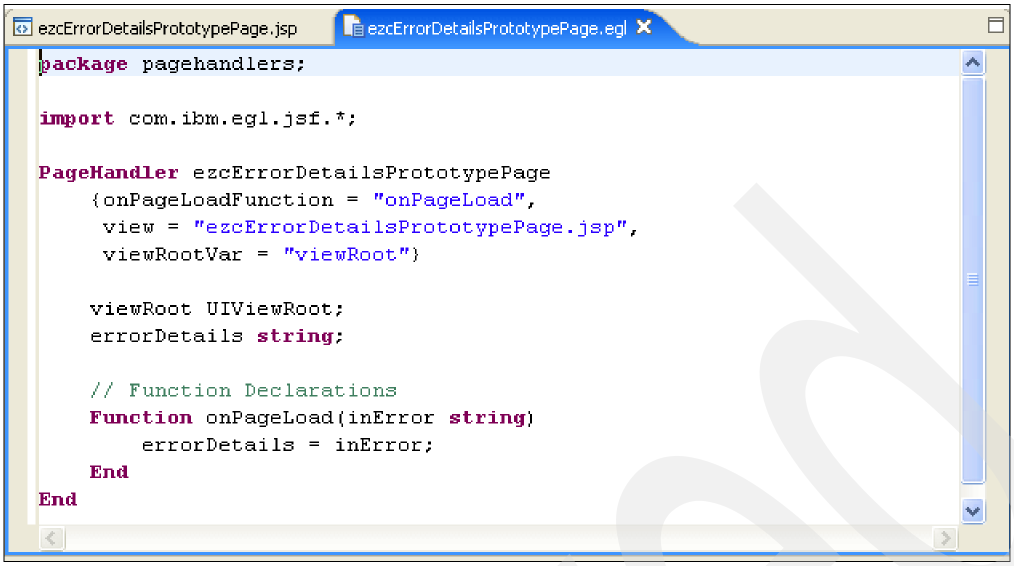


Figure 9-32 Source code for ezcErrorDetailsPrototypePage.egl

2. The layout of the page must be modified to emphasize the header. The Properties view is the primary means for modifying the elements on the page. Figure 9-33 shows the Properties view when the header text is highlighted (that is, the static text: `An Error Has Occurred`).

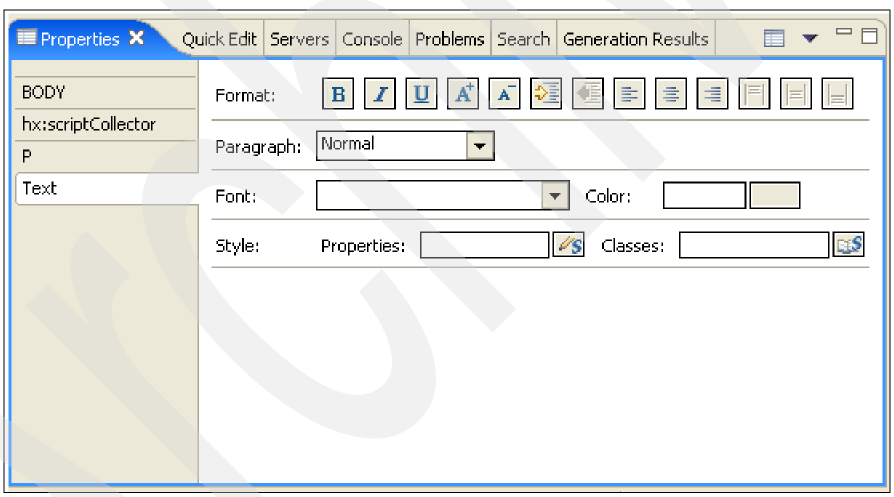


Figure 9-33 Properties view for text on a JSP page

3. Use the first and fourth Format buttons to make the header font larger and bolder. The optional format toolbar can also be used to make these font changes. When editing a JSP page, you will have a Toolbar menu. Ensure that the Format menu has a check mark beside it. The format toolbar will appear below the workspace menu bar. The toolbars are readily available and easy to use. The "hover help" makes it easy to determine what the tools do.

4. Use the Page Data view **Insert Controls for "errorDetails"** menu options. Now, you need to make the output field for the error message as large as possible. Modify the width of the HTML table that contains the field. An easy way to find this table is to use the Outline view, which is usually found in the lower left corner of the EGL or Web perspective. Figure 9-34 on page 151 is the Outline view of the prototype configure page with the HTML table selected.

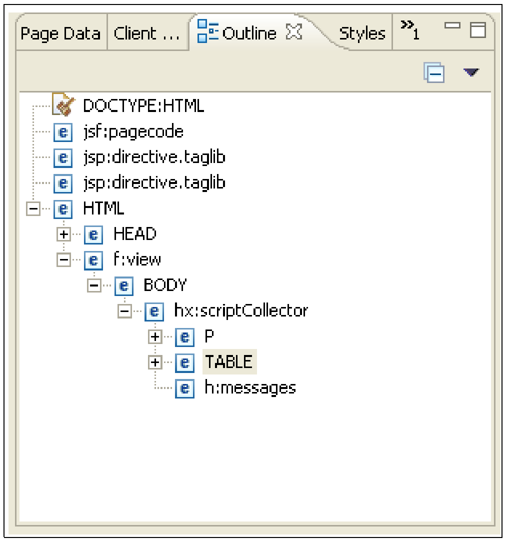


Figure 9-34 Page Outline View for ezcErrorDetailsPrototypePage

5. After the HTML table is selected, open the Properties view, which looks like Figure 9-35.

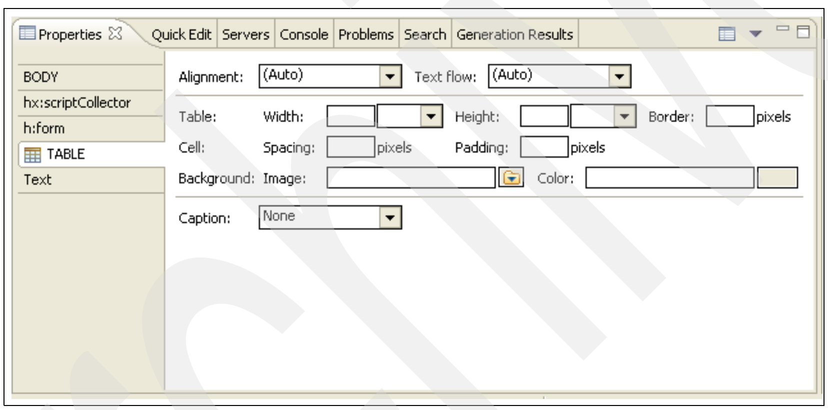


Figure 9-35 HTML Table properties

6. On the Properties view, change the width to `100` percent of its container. Set the width of the first cell to `104` pixels. Make the text in this field bold. Figure 9-36 on page 152 shows the element selected in the Outline view and the Design tab.

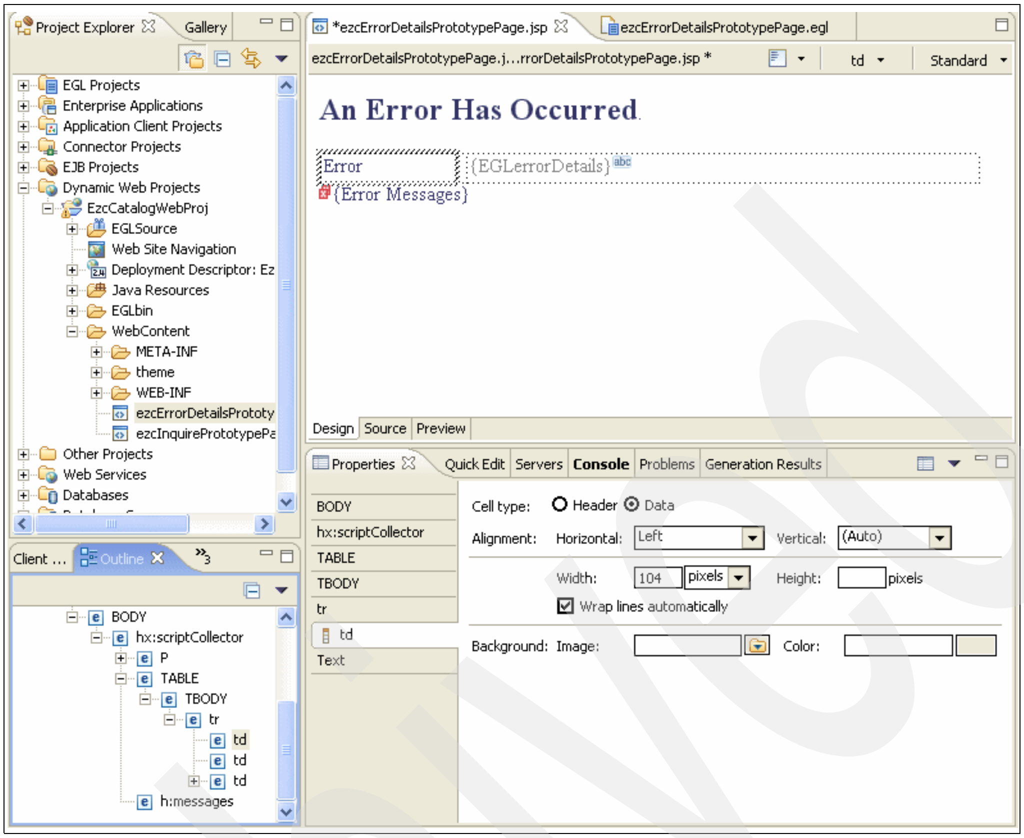


Figure 9-36 Web Perspective with an HTML cell selected

7. Because the first two columns have a specified width, only the third column grows when the table grows with the page. The error pages also need a backward navigational link. To add the BACK link, open the HTML Tags palette drawer and drag and drop a link on the bottom of the page. Select the type **Others**, set the URL to `javaScript:history.back()`, and set the Link text to **Back**. The completed dialog for this hyperlink is shown in Figure 9-37.

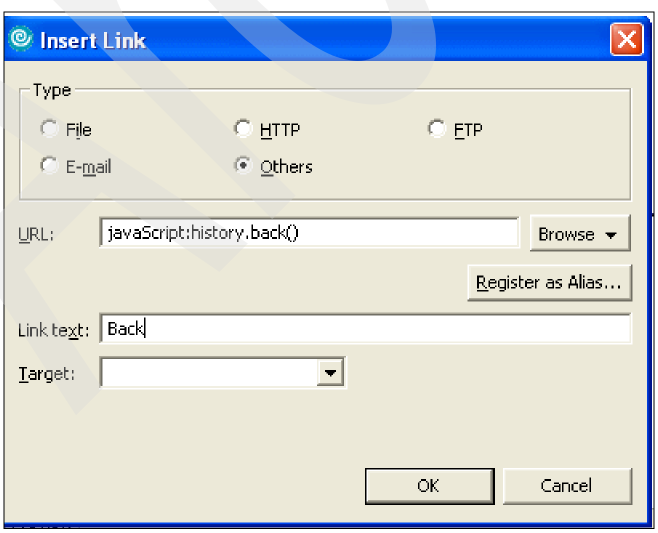


Figure 9-37 Insert Link dialog for backwards link

8. Instead of a hardcoded URL, JavaScript is used to determine the last page rendered. The runtime version of this error page is shown in Figure 9-38.

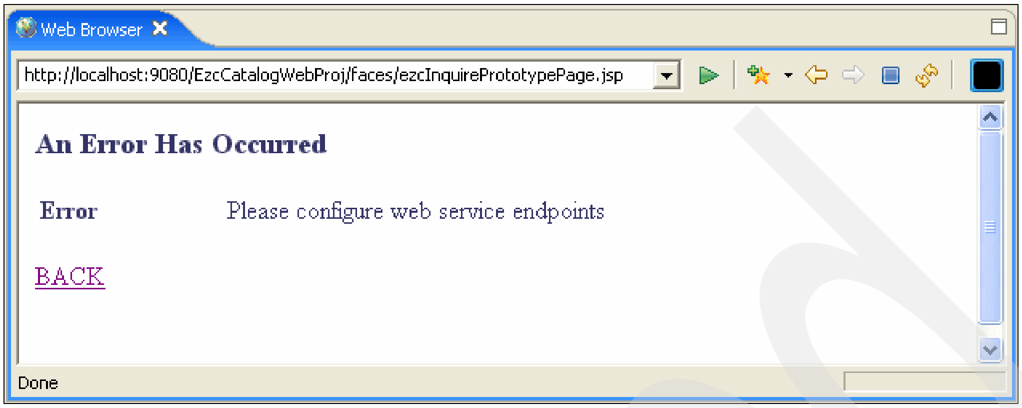


Figure 9-38 Application error page at run time

The EGL online documentation indicates that the runtime processing of a Web service can generate an EGL system exception, notably SysLib.ServiceInvocationException. The documentation reveals that all exceptions have a code and a description, and the service invocation exception has several more descriptive fields. It is helpful to have a separate error page for the details of an EGL system exception.

9. Create a separate error page to show the details of an EGL system exception detail. Name the page `ezcExceptionDetailsPrototypePage` with the header `EGL System Exception Details`. Adjust the header text so that it has a larger, bolder font than the default. Select **Edit Page Code** from the Page Designer Design tab context menu to modify the pageHandler. Define an EGL record, `eglExceptionDetailsPrototypeRec`, in the pageHandler to hold all six fields associated with a service invocation exception. Add the variable declaration for **details** of type `eglExceptionDetailsPrototypeRec`. Modify the onPageLoad function to accept an eglExceptionDetailsPrototypeRec parameter named `inDetails` and initialize the pageHandler variable details to the parameter. Figure 9-39 on page 154 shows the content of the exception pageHandler.

*ezcExceptionDetailsPrototypePage.jsp | *ezcExceptionDetailsPrototypePage.egl

```
package pagehandlers;

import com.ibm.egl.jsf.*;

record eglExceptionDetailsPrototypeRec
    code string { displayName = "Code" };
    description string { displayName = "Description"};
    faultCode string { displayName = "FaultCode"};
    source string { displayName = "Source"};
    location string { displayName = "Location"};
    diagnostic string { displayName = "Diagnostic"};
end // eglExceptionDetailsPrototypeRec

PageHandler ezcExceptionDetailsPrototypePage
    {onPageLoadFunction = "onPageLoad",
     view = "ezcExceptionDetailsPrototypePage.jsp",
     viewRootVar = "viewRoot"}

    viewRoot UIViewRoot;
    details eglExceptionDetailsPrototypeRec;

    // Function Declarations
    Function onPageLoad(inDetails eglExceptionDetailsPrototypeRec)
        details = inDetails;
    End
End
```

Figure 9-39 Source code for ezcExceptionDetailsPrototypePage.egl

10. The displayName property for the record items is used for labels and column headings when the record is dropped on the JSP page. Now, drop the details data object onto the JSP page. Modify the HTML table, so its width is 100%. Make all the labels bold and adjust the first cell to a pixel width of 104. Add the BACK hyperlink.

11. This exception details page will be launched from a try-onException block, which needs to surround any Web service invocation. The EGL system function, sysLib.currentException, provides access to the exception fields. The forward statement transfers control to the named page. It is helpful to verify that the JSP name was entered correctly, because no runtime error message is issued if the page does not exist. Highlight the page name in the forward statement and press F3. If the editor opens on the page, the name was typed correctly. Figure 9-40 on page 155 shows the exception handling in the modified invokeInquireCatalogServiceAction function.

```
// V2.0 invoke the Web Service and handle exceptions
Function invokeInquireCatalogServiceAction()
    details eglExceptionDetailsPrototypeRec;
    try
        inquireCatalog_ServiceBindingLib.DFHOXCMNPort.DFHOXCMN(
            input_ca_request_id,
            input_ca_resturn_code,
            input_ca_response_message,
            input_ca_inquire_request,
            output_ca_request_id,
            output_ca_return_code,
            output_ca_response_message,
            output_ca_inquire_request);
    onException
        set details empty;
        details.code = sysLib.currentException.code;
        details.description = sysLib.currentException.description;
        case (sysLib.currentException.code)
            when ("com.ibm.egl.ServiceInvocationException")
                details.faultCode = sysLib.currentException.faultCode;
                details.source = sysLib.currentException.source;
                details.location = details.location + sysLib.currentException.location;
                details.diagnostic = sysLib.currentException.diagnostic;
        end // case
        forward details to "ezcExceptionDetailsPrototypePage";
    end // try/onException statement
End  // invokeInquireCatalogServiceAction
```

Figure 9-40 Service invocation exception handling code in ezcInquirePrototypePage.egl

12. Retest the inquire catalog function with item ref value 75. Now, the exception is handled and the runtime exception page shows in Figure 9-41.

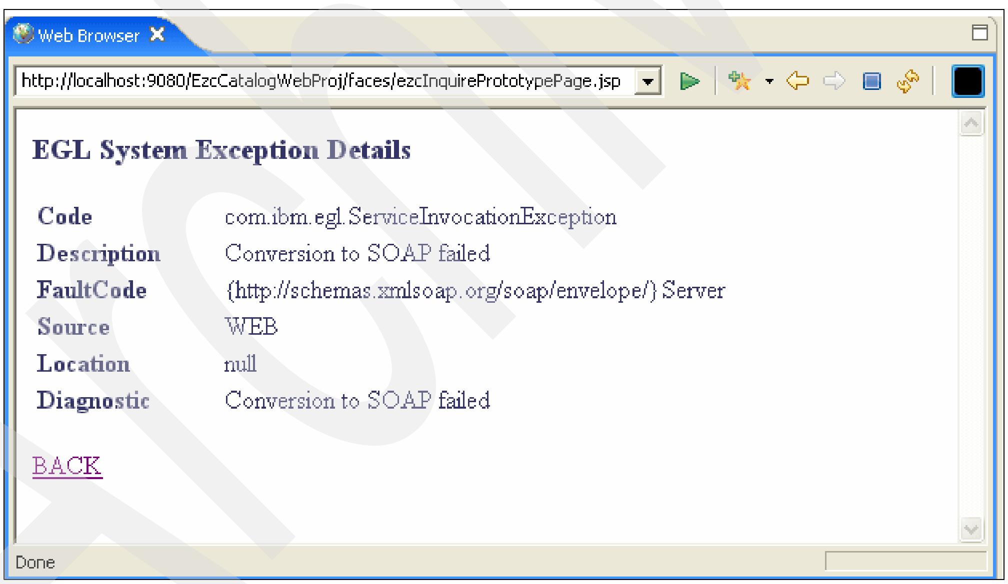


Figure 9-41 EGL System Exception Details page at run time

13. Do not forget to try out the **BACK** hyperlink. Now that the application can handle system exceptions, it will be easier to test the configure application use case.

9.4.3 Configure application prototype

To configure the application prototype:

1. Create the prototype configure page `ezcConfigurePrototypePage` by selecting **New** → **Faces JSP File** with the WebContent folder selected. Watch the lower right corner of the WDz window for the status of the workbench. From the Page Designer's Design tab,

modify the default contents of the page by changing **Place your page content here.** to show a page header `Configure Prototype`. Open the context menu and select **Edit Page Code** to open the EGL editor on the associated pageHandler. Modify the pageHandler to have two string variables: one string variable for the current endpoint variable name and one string variable to hold the new endpoint. Modify the onPageLoad function to properly initialize the two variables. If the session variable has not been set, the system library is used to get the endpoint name from the service binding library. You have to add an import statement to this library, so your code can reference it. You need a function to save the new endpoint and a second function to navigate to the inquire page. The pageHandler code looks like Figure 9-42.

```
package pagehandlers;

import mixedcase.com.DFHOXCP3.DFHOXCMN.www.inquireCatalog_ServiceBindingLib;
import com.ibm.egl.jsf.*;

PageHandler ezcConfigurePrototypePage
    (onPageLoadFunction = "onPageLoad",
     view = "ezcConfigurePrototypePage.jsp",
     viewRootVar = "viewRoot")

    viewRoot UIViewRoot;

    currentEndpoint string;
    newEndpoint string;

    // Function Declarations
    Function onPageLoad()
        j2eeLib.getSessionAttr("allEP", currentEndpoint);
        if (currentEndpoint == "")
            currentEndpoint = serviceLib.getWebEndpoint(inquireCatalog_ServiceBindingLib.DFHOXCMNPort
        end
        newEndpoint = currentEndpoint;
    End

    //  cache the endpoint into session object
    Function requestConfigureEndpointAction()
        message String;
        sysLib.writeStdout("Caching endpoint " + newEndpoint);
        j2eeLib.setSessionAttr("allEP", newEndpoint);

        // setWebEndpoint successful so update currentEndpoint
        currentEndpoint = newEndpoint;
    End // requestConfigureEndpointAction()

    // goto the inquire catalog page
    Function requestInquireCatalogAction()
        forward to "ezcInquirePrototypePage";
    End // requestInquireCatalogAction()

End
```

Figure 9-42   Source code for ezcConfigurePrototypePage.egl

2. Modify ezcInquireCatalogPrototype.egl's requestInquireCatalogAction to retrieve the cached endpoint variable from the session object. If the variable has not been set, issue an application error requesting configuration of the application. An application error message will be issued if the Web service returns a nonzero return code. Figure 9-43 on page 157 has the second version of this function.

```
// V2.0 Retrieve endpoint from session object, set endpoint, and invoke request
Function requestInquireCatalogAction()
    ezcEndpoint string;

    j2eeLib.getSessionAttr("allEP", ezcEndpoint);
    if (ezcEndpoint == "")
        forward "Please configure web service endpoints" to "ezcErrorDetailsPrototypePage";
    end // if

    sysLib.writeStdout("Setting endpoint ... " + ezcEndpoint);
    serviceLib.setWebEndpoint(inquireCatalog_ServiceBindingLib.DFHOXCMNPort, ezcEndpoint);
    sysLib.writeStdout("Calling web service");
    invokeInquireCatalogServiceAction();

    // service was successful ... display results to user
    if (output_ca_return_code == 0)
        setError(output_ca_response_message);
    else
        forward output_ca_response_message to "ezcErrorDetailsPrototypePage";
    end // if-else
    sysLib.writeStdout("Done calling web service");
End // requestInquireCatalogAction()
```

Figure 9-43 Code to retrieve cached endpoint

3. Switch back to the Page Designer on the prototype configure page. Add the currentEndpoint and newEndpoint pageHandler variables to the JSP page using the **Insert New Controls for selected objects** context menu option on the Page Data tab. Ensure that the **Update an existing record** radio button is selected so that a submit button can be added. The currentEndpoint needs a control type of Output, and the newEndpoint needs a control type of Input. Bind the requestConfigureEndpointAction to the submit button. Open the faces component drawer on the palette and drop a command button after the submit button. Use the Properties view to change the Button label to `Inquire` as shown in Figure 9-44.

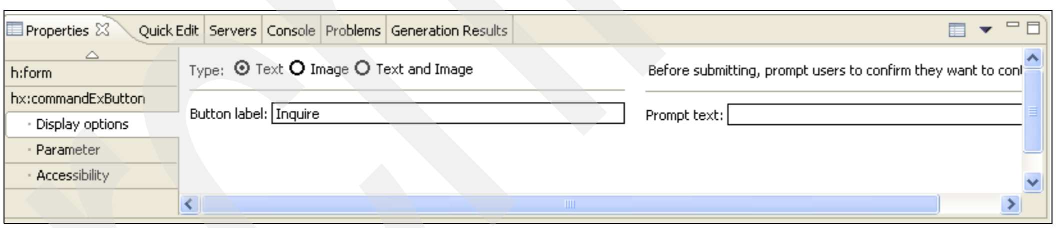


Figure 9-44 Command button Properties view on the Display options tab

4. Bind each action function to its appropriate command button. Add another command button to the inquire page and a function to its pageHandler to invoke the configure page. Figure 9-45 on page 158 shows a preview of the configure page.

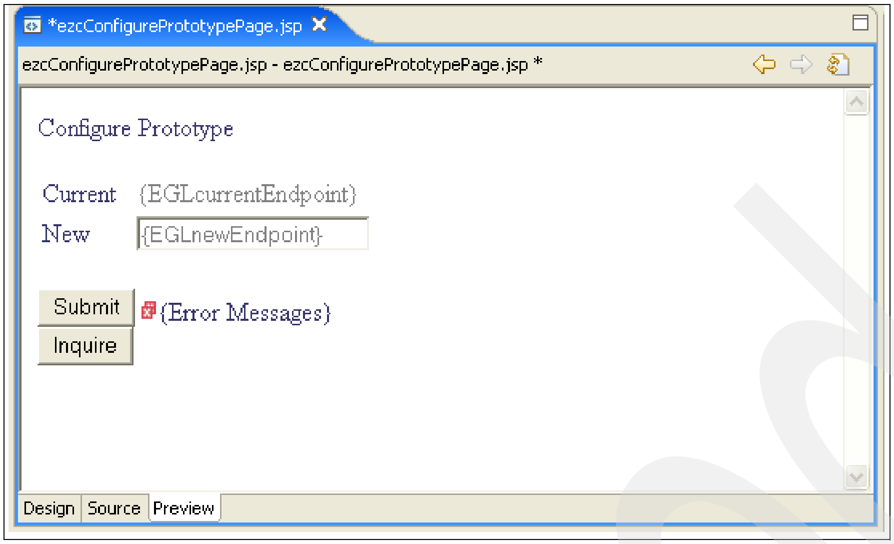


Figure 9-45 Page Designer's Preview of the ezcConfigurePrototypePage.jsp

5. The default field size does not show the entire value of the Web service endpoint. The layout of the page must be modified to emphasize the header and make the endpoint text field large enough to show the entire value just like the error pages (see Figure 9-33 on page 150). Make sure the HTML consumes 100% of the panel (see Figure 9-35 on page 151). Adjust the width of the first column to 104 pixels (see Figure 9-36 on page 152). To ensure that the input text field will take the majority of its column, adjust the width of the input text field to a percentage. Figure 9-46 shows the Properties view of the input text field, and it is not obvious how to modify its width.

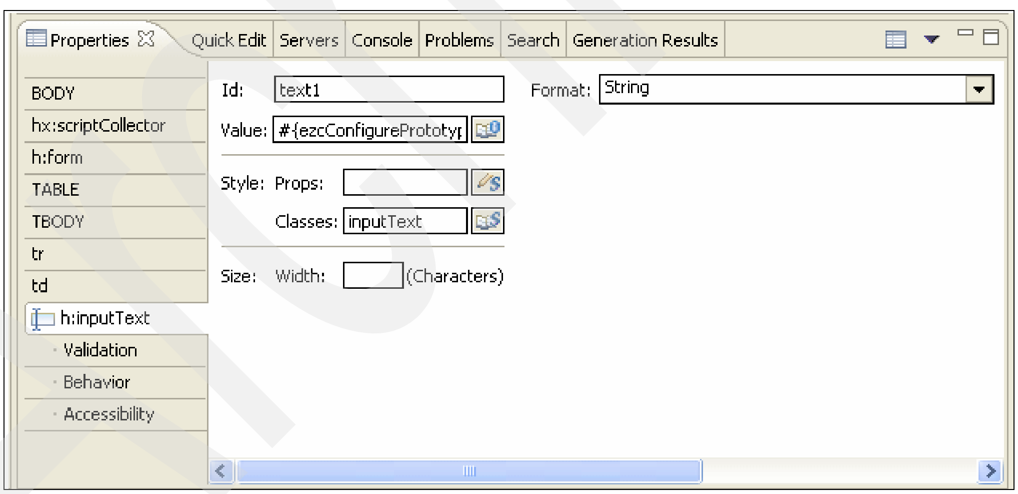


Figure 9-46 Input text field Properties view

6. Select the push button after the Style: Props: field and modify the dialog to resemble Figure 9-47 on page 159.

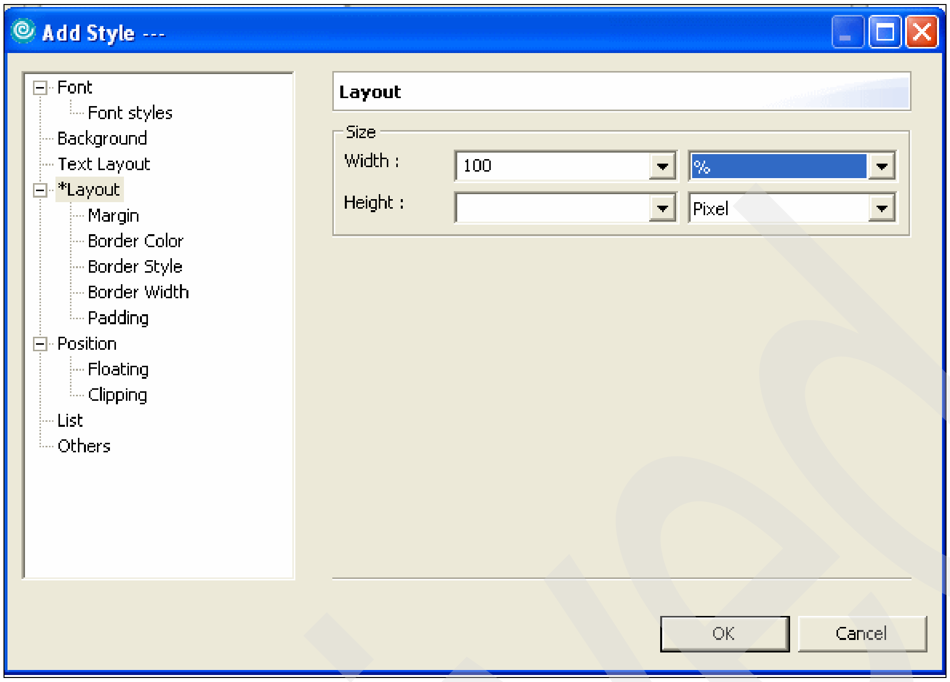


Figure 9-47 Add Style Properties Dialog

7. The modified layout width attribute will allow the entire text of the endpoint to be visible. This information is required to invoke the Web service. To make the field required, select the new endpoint value input text field and modify its validation properties to make it a required field (check **Value is required**) as shown in Figure 9-48.

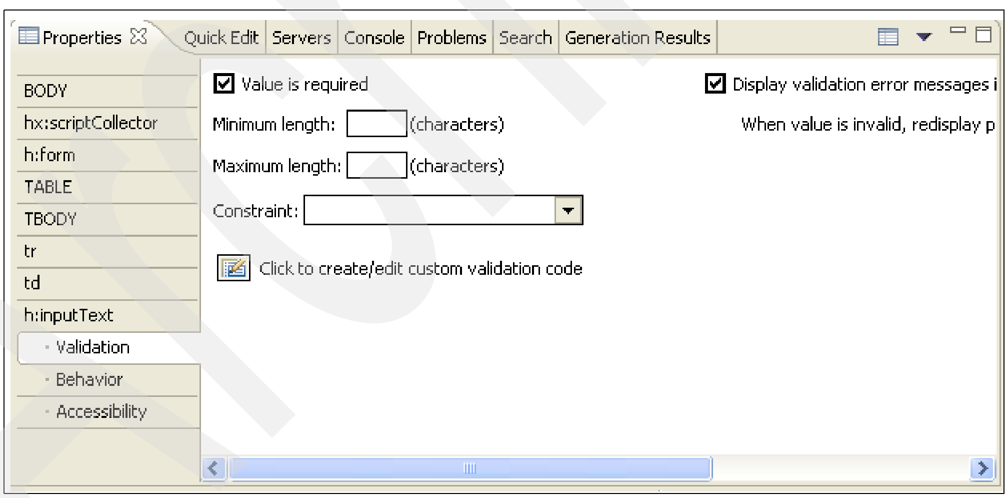


Figure 9-48 Validation Properties for input text field

8. Now, you can test the ezcConfigurePrototypePage page. Start the server and then select **Run on Server** for ezcConfigurePrototypePage.jsp. Figure 9-49 on page 160 shows the runtime version of the prototype configure page.

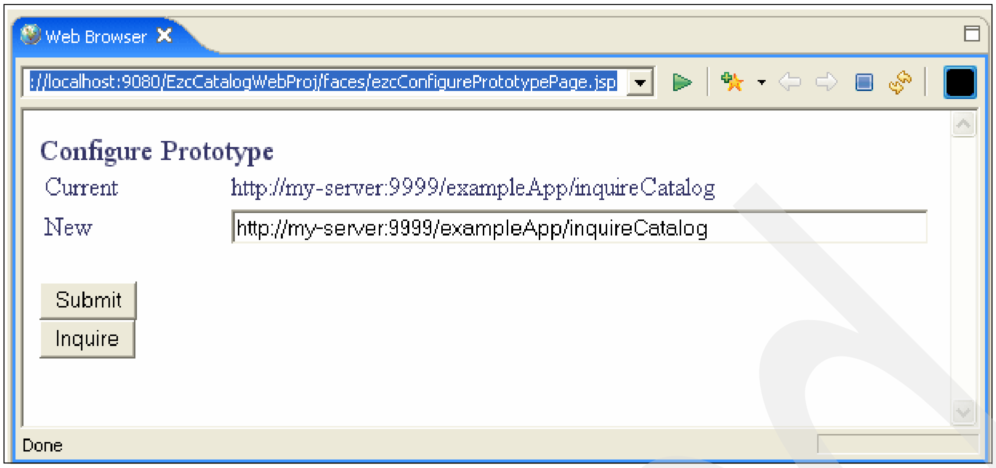


Figure 9-49 Runtime rendering of ezcConfigurePrototypePage.jsp

9. Replace the new endpoint input field with the appropriate endpoint, select **Submit**, then select **Inquire**, and select the **Submit** button on the inquire page. Figure 9-50 shows the result of this test scenario.

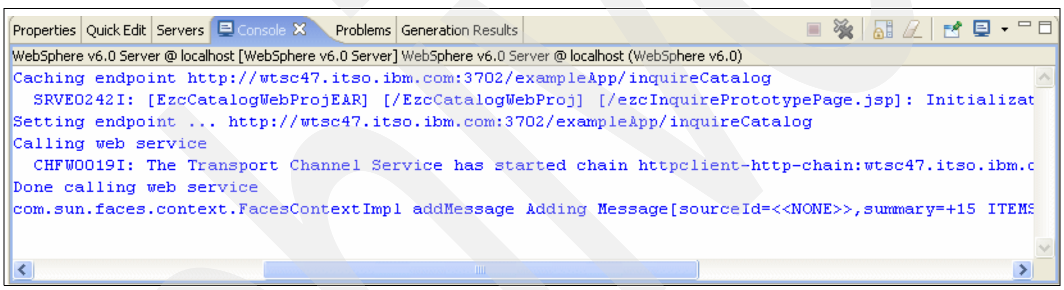


Figure 9-50 Console content after successful execution of prototype

Even if you do not code specific trace entries to the console, you need to review the console frequently.

9.4.4 Data formatting

The final activity of the elaboration stage is to clean up the data format issues. Figure 9-51 on page 161 shows the default data table.

Ca_item_ref	Ca_description	Ca_department	Ca_cost	In_stock	On_order
50	Pencil with eraser 12pk	10	001.78	100	6
60	Highlighters Assorted 5pk	10	003.89	100	42
70	Laser Paper 28-lb 108 Bright 500/ream	10	007.44	100	50
80	Laser Paper 28-lb 108 Bright 2500/case	10	033.54	100	4
90	Blue Laser Paper 20lb 500/ream	10	005.35	100	89
100	Green Laser Paper 20lb 500/ream	10	005.35	100	20
110	IBM Network Printer 24 - Toner cart	10	169.56	100	
120	Standard Diary: Week to view 8 1/4x5 3/4	10	025.99	100	
130	Wall Planner: Eraseable 36x24	10	018.85	100	
140	70 Sheet Hard Back wire bound notepad	10	005.89	100	
150	Sticky Notes 3x3 Assorted Colors 5pk	10	005.35	100	54
160	Sticky Notes 3x3 Assorted Colors 10pk	10	009.75	100	32
170	Sticky Notes 3x6 Assorted Colors 5pk	10	007.55	100	102
180	Highlighters Yellow 5pk	10	003.49	100	10
190	Highlighters Blue 5pk	10	003.49	100	20

Figure 9-51 Results from inquire catalog Web service with default format

The production version of this data table needs better column labels, a properly formatted monetary value, center or right alignment for numeric values, and zeros showing for numeric values rather than blanks. The easiest way to achieve these changes is by using an EGL Record.

The steps are:

1. Define a new EGL record in the inquire pageHandler. Use the response record as the starting template. Locate and copy the record definition for `mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.ca_cat_item`. Paste the record definition below the import statements in the inquire pageHandler. Make the following changes to the record definition:
   a. Change the record name to `catalogItemPrototypeRec`.
   b. Rename the dataItems and remove `ca_department`.
   c. Change the dataItem types to `int`, `string`, and `money{6,2}`.
   d. Delete all the old dataItem properties.
   e. For each dataItem, add the displayName property, which will be used as the column label for each dataItem.
   f. For each numeric dataItem, add the zeroFormat property set to `yes`.
   g. For the money dataItem, add the currency and numericSeparator properties set to `yes`.

The resulting record definition is shown in Figure 9-52.

```
// Record definition to help format catalog data
record catalogItemPrototypeRec
  item_ref int {displayName = "Item Ref", zeroFormat = yes};
  description string {displayName = "Description" };
  in_stock int {displayName = "In Stock", zeroFormat = yes};
  on_order int {displayName = "On Order", zeroFormat = yes};
  money_cost money(6,2){displayName = "Cost", zeroFormat = yes, currency = yes, numericSeparator = yes };
end
```

Figure 9-52 EGL record definition for catalogItemPrototypeRec

2. Additional modifications need to be made to the pageHandler to get the correctly formatted data to the Web page. Add a variable, `prettyList`, declared as an array of

catalogItemPrototypeRecs to the pageHandler. Add a new function to the inquire pageHandler named `formatResults` to move the data values from the response record to the formatting EGL record. Invoke this function if the Web service request was successful. Figure 9-53 shows the final version of the function necessary to display the formatted data.

```
prettyList catalogItemPrototypeRec[];

// Get results into an EGL record for easy formatting
Function formatResults()
    index int = 1;
    mySize int = output_ca_inquire_request.ca_cat_item.getSize();
    oldOne mixedcase.com.Response.DFHOXCP3.DFHOXCMN.www.ca_cat_item;
    newOne catalogItemPrototypeRec;
    while (mySize >= index)
        oldOne = output_ca_inquire_request.ca_cat_item[index];
        newOne.item_ref = oldOne.ca_item_ref;
        newOne.description = oldOne.ca_description;
        newOne.in_stock = oldOne.in_stock;
        newOne.on_order = oldOne.on_order;
        newOne.money_cost = mathLib.stringAsDecimal(oldOne.ca_cost);
        prettyList.appendElement(newOne);
        index = index + 1;
    end // while
End // formatResults()

// V3.0 Retrieve endpoint from session object, set endpoint, and invoke request
// Get the results properly formatted
Function requestInquireCatalogAction()
    ezcEndpoint string;

    j2eeLib.getSessionAttr("allEP", ezcEndpoint);
    if (ezcEndpoint == "")
        forward "Please configure web service endpoints" to "ezcErrorDetailsPrototypePage";
    end // if

    sysLib.writeStdout("Setting endpoint ... " + ezcEndpoint);
    serviceLib.setWebEndpoint(inquireCatalog_ServiceBindingLib.DFHOXCMNPort, ezcEndpoint);
    sysLib.writeStdout("Calling web service");
    invokeInquireCatalogServiceAction();

    // service was successful ... display results to user
    if (output_ca_return_code == 0)
        formatResults();
    else
        forward output_ca_response_message to "ezcErrorDetailsPrototypePage";
    end // if-else
    sysLib.writeStdout("Done calling web service");
End // requestInquireCatalogAction()
```

Figure 9-53 Source code for formatResults function in ezcInquirePrototypePage.egl

3. Several EGL-provided functions are utilized to get the data into the new record. The function getSize returns how many catalog items are in the ca_cat_item field. The EGL system library mathLib handles the conversion of the string ca_cost field to the decimal value required by the money field. The remaining formatting issue is whether to have right or center alignment on the numeric fields. The alignment needs to be modified on the visual component. Use the **Insert New Controls for "prettyList"** menu option to add visual components for the newly added prettyList array. Change the HTML table properties, so that its width is `100%` of the page width (reference Figure 9-35 on page 151). Modify the horizontal alignment property (Alignment: Horizontal) to **Right** for the cost column. The modified Properties view is shown in Figure 9-54 on page 163.

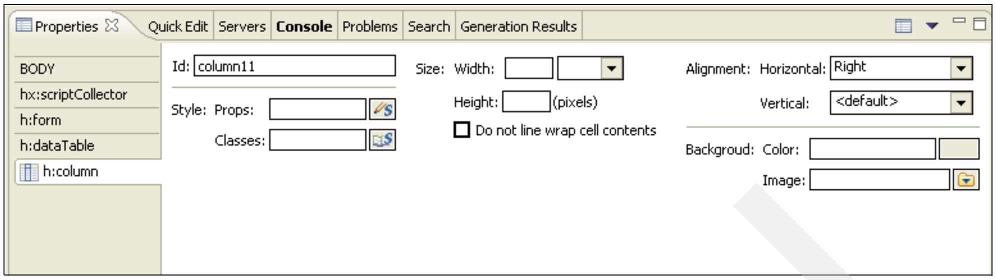


Figure 9-54 Properties view for table column

4. Change the ItemRef**,** In Stock, and On Order columns to have the **Alignment: Horizontal** property set to **Center**. Test the application and verify the data formatting. Figure 9-55 illustrates the data formatting capabilities of the EGL dataItem declarations.

Item Ref	Description	In Stock	On Order	Cost
50	Pencil with eraser 12pk	100	9	$1.78
60	Highlighters Assorted 5pk	100	42	$3.89
70	Laser Paper 28-lb 108 Bright 500/ream	100	50	$7.44
80	Laser Paper 28-lb 108 Bright 2500/case	100	15	$33.54
90	Blue Laser Paper 20lb 500/ream	100	89	$5.35
100	Green Laser Paper 20lb 500/ream	100	20	$5.35
110	IBM Network Printer 24 - Toner cart	100	0	$169.56
120	Standard Diary: Week to view 8 1/4x5 3/4	100	0	$25.99
130	Wall Planner: Eraseable 36x24	100	0	$18.85
140	70 Sheet Hard Back wire bound notepad	100	0	$5.89
150	Sticky Notes 3x3 Assorted Colors 5pk	100	55	$5.35
160	Sticky Notes 3x3 Assorted Colors 10pk	100	32	$9.75
170	Sticky Notes 3x6 Assorted Colors 5pk	100	102	$7.55
180	Highlighters Yellow 5pk	100	10	$3.49
190	Highlighters Blue 5pk	100	20	$3.49

Figure 9-55 Formatted results from inquire catalog Web service

The prototype provides an executable architecture that validates the invocation of Web services, storage and retrieval of session data, error handling, passing data between pages, and page to page navigation. Due to the ease of construction, the elaboration was a throwaway prototype rather an executable architecture that gets extended during the Construction Phase.

9.5 Construction Phase

In the Construction Phase, the EGL developers incorporated the results of the Web developers' Elaboration Phase. The Web developers determined the look-and-feel as well as the navigation of the application by specifying the page template, style sheets, and graphics. These artifacts are imported into the WebContent\theme folder. If you are recreating the Catalog Manager Web client, you will need to import the content of this directory from the sample code. The *navigationShell.jtpl* page template will be used to create the pages in the application. The rendering of the navigational buttons depends on the value of the page ID variable. The Page Designer Source tab should be used to modify this line of code: <c:set var="pageID">*xxx*</c:set> where *xxx* is the page identifier. It is important to use the same pageID and JSP file names used in navigationShell.jtpl so that the navigation will work correctly. Figure 9-56 has the name-sensitive parts of the page handler in bold.

```
<c:if test='${pageID != "inquireALL"}'>
    <A href="/EzcCatalogWebProj/faces/ezcInquireCatalogPage.jsp"></A>
</c:if>
<c:if test='${pageID != "inquireONE"}'>
    <A href="/EzcCatalogWebProj/faces/ezcInquireSinglePage.jsp"></A>
</c:if>
<c:if test='${pageID != "placeOrder"}'>
     <A href="/EzcCatalogWebProj/faces/ezcPlaceOrderPage.jsp"></A>
</c:if>
<c:if test='${pageID != "replenish"}'>
     <A href="/EzcCatalogWebProj/faces/ezcReplenishPage.jsp"></A>
</c:if>
<c:if test='${pageID != "welcome"}'>
    <A href="javascript:history.back()"></A>
</c:if>
<c:if test='${pageID != "configure"}'>
    <A href="/EzcCatalogWebProj/faces/ezcConfigurePage.jsp"></A>
</c:if>
```

Figure 9-56 Name-sensitive sections of the page template

The simplest pages are built first, then you move in complexity to the layout intensive pages.

9.5.1 Simple response pages

Build the simple response pages first, remembering to use the navigationShell.jtpl as the page template.

To create the welcome page:

1. Verify you have the page template navigationShell.jtpl in your WebContent\theme folder.
2. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
3. Create a new Faces JSP File named `ezcWelcomePage`, select the option **Create with page template** and select **User-defined page template** and **navigationShell.jtpl** on the second page.
4. Switch to the Page Designer's **Source** tab.
5. Change the page title to `Welcome to RUPz Catalog Application`.
6. Change the page ID to `welcome`.
7. Switch to the Page Designer's **Design** tab.
8. Change the header text to `Welcome to the CICS RUPz Catalog Example Application`.
9. Change the body content to static text: `Please select an option from the menu.`

10. Save the JSP file.

To create the application error details page, you follow the same steps:

1. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
2. Create a new Faces JSP File named `ezcErrorDetailsPage` using navigationShell.jtpl.
3. Switch to the Page Designer's **Source** tab.
4. Change the page title to `EGL RUPz Catalog Application Error`.
5. Change the page ID to `appError`.
6. Switch to the Page Designer's **Design** tab.
7. Change the header to `An Error Has Occurred`.
8. Select the **Edit Page Code** context menu option.
9. Add variable declaration `errorDetails string` to the pageHandler.
10. Add parameter `inDetails string` to the function onPageLoad and use it to initialize the pageHandler variable.
11. Save the modifications to the pageHandler.
12. Add controls for errorDetails, check the option **Display an existing record (read-only)**, and change the label to `Error` using page designer.
13. Change the first column cell size to `104` pixels as referenced in Figure 9-36.
14. Make the text in the first column cell bold.
15. Make the HTML table grow with the page as referenced in Figure 9-35.
16. Save the JSP file.

Figure 9-57 on page 166 is the completed application error page defined with the page template.

Figure 9-57 Completed Application Error page

To create the system exception details page, you follow the similar steps. Reference 9.4.2, “Error handling” on page 149 for more details:

1. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
2. Creates Faces JSP File named `ezcExceptionDetailsPage` using navigationShell.jtpl.
3. Switch to the Page Designer’s **Source** tab.
4. Change the page title to `EGL RUPz System Exception`.
5. Change the page ID to `sysError`.
6. Switch to the Page Designer’s **Design** tab.
7. Change the header to `An EGL System Exception Has Occurred`.
8. Select the **Edit Page Code** context menu option.
9. Add the EGL record definition for the details as shown in Figure 9-39 on page 154.
10. Add variable declaration `errorDetails eglExceptionDetailsRec` to the pageHandler.
11. Add parameter `inDetails eglExceptionDetailsRec` to the function onPageLoad and use it to initialize the pageHandler variable.
12. Save the modifications to the pageHandler.
13. Add controls for errorDetails and check the option **Display an existing record (read-only)** using the page editor.
14. Change the first column cell size to `104` pixels as referenced in Figure 9-36 on page 152.
15. Make all the label text bold.
16. Make the HTML table grow with the page as referenced in Figure 9-35 on page 151.
17. Save the JSP file.

The order response and the replenish Inventory response pages are the same as the application error page:

1. Select folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
2. Create a new Faces JSP File named `ezcOrderResponsePage` using the template.
3. Switch to the Page Designer's **Source** tab.
4. Change the page title to `EGL RUPz Order Results`.
5. Change the page ID to `orderResponse`.
6. Switch to the Page Designer's **Design** tab.
7. Change the header to `Order Placed`.
8. Select the **Edit Page Code** context menu option.
9. Add variable declaration `orderDetails string` to the pageHandler.
10. Add parameter `inDetails string` to the function onPageLoad and use it to initialize the pageHandler variable.
11. Save the modifications to the pageHandler.
12. Add controls for orderDetails, check the option **Display an existing record (read-only)**, and change the label to `Order Details` using the page editor.
13. Change the first column cell size to `104` pixels as referenced in Figure 9-36 on page 152.
14. Make all label text bold. Toggle off the automatic wrap option.
15. Make the HTML table grow with the page as referenced in Figure 9-35 on page 151.
16. Save the JSP file.

To create the replenish inventory response page:

1. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
2. Create a new Faces JSP File named `ezcReplenishResponsePage` using the template.
3. Switch to the Page Designer's **Source** tab.
4. Change the page title to `EGL RUPz Replenish Inventory Results`.
5. Change the page ID to `replenishResponse`.
6. Switch to the Page Designer's **Design** tab.
7. Change the header to `Replenish Inventory`.
8. Select the **Edit Page Code** context menu option.
9. Add variable declaration `replenishDetails string` to the pageHandler.
10. Add parameter `inDetails string` to the function onPageLoad and use it to initialize the pageHandler variable.
11. Save the modifications to the pageHandler.
12. Add controls for orderDetails, check the option **Display an existing record (read-only)**, and change the label to `Replenish Details` using the page editor.
13. Change the first column cell size to `120` pixels as referenced in Figure 9-36 on page 152.
14. Make all label text bold. Toggle off the automatic wrap option.
15. Make the HTML table grow with the page as referenced in Figure 9-35 on page 151.
16. Save the JSP file.

9.5.2 Web Service request pages

Next, create the pages that request a Web service and display either the error page or a simple response page depending on the results. To improve productivity, define an EGL template to abbreviate the process of coding the Web service invocation. To define an EGL template, open the preferences dialog and navigate to **EGL** → **Editor** → **Templates** and select **new**. Figure 9-58 shows the creation dialog for the ezcWebCall template.

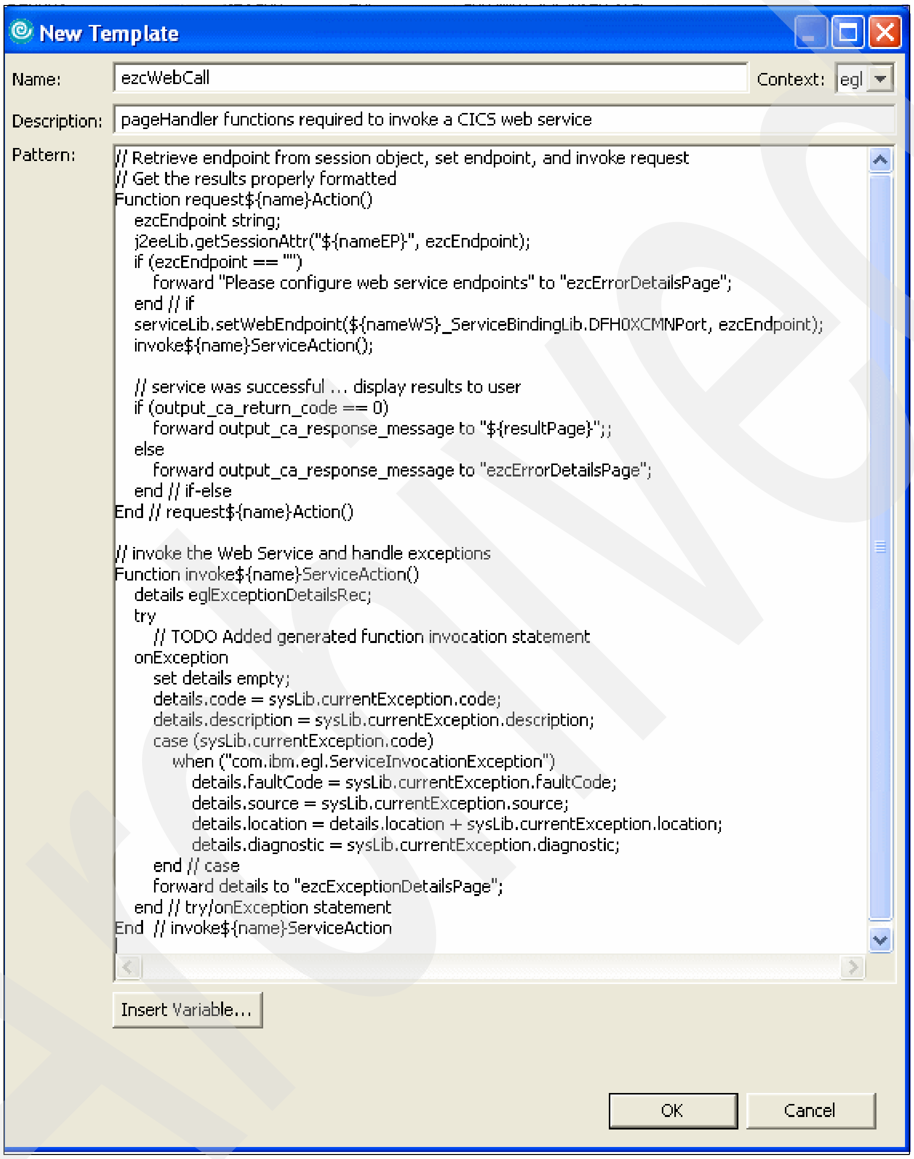


Figure 9-58 EGL Template creation dialog for ezcWebCall template

This template will provide the functions required to retrieve the configured endpoint, set the Web service endpoint, invoke the service, and handle the results. After the onPageLoad function, type `ezc` and hold down the Ctrl+Spacebar keys (which invoke content assist). Select **ezcWebCall** from the pop-up list and press Enter. Figure 9-59 on page 169 is the source code that is added to the pageHandler.

```
// Retrieve endpoint from session object, set endpoint, and invoke request
// Get the results properly formatted
Function requestnameAction()
    ezcEndpoint string;
    j2eeLib.getSessionAttr("nameEP", ezcEndpoint);
    if (ezcEndpoint == "")
        forward "Please configure web service endpoints" to "ezcErrorDetailsPage";
    end // if
    serviceLib.setWebEndpoint(nameWS_ServiceBindingLib.DFHOXCMNPort, ezcEndpoint);
    invokenameServiceAction();

    // service was successful ... display results to user
    if (output_ca_return_code == 0)
        forward output_ca_response_message to "resultPage";;
    else
        forward output_ca_response_message to "ezcErrorDetailsPage";
    end // if-else
End // requestnameAction()

// invoke the Web Service and handle exceptions
Function invokenameServiceAction()
    details eglExceptionDetailsRec;
    try
        // Added generated function invocation statement

    onException
        set details empty;
        details.code = sysLib.currentException.code;
        details.description = sysLib.currentException.description;
        case (sysLib.currentException.code)
            when ("com.ibm.egl.ServiceInvocationException")
                details.faultCode = sysLib.currentException.faultCode;
                details.source = sysLib.currentException.source;
                details.location = details.location + sysLib.currentException.location;
                details.diagnostic = sysLib.currentException.diagnostic;
        end // case
        forward details to "ezcExceptionDetailsPage";
    end // try/onException statement
End  // invokenameServiceAction
```

Figure 9-59   Code added by the ezcWebCall EGL template

The variable parts of the code will be contained in a blue box. The Tab key will navigate between the variables. When the text is highlighted, you can type over the value. After the four variables are customized, copy the generated function invocation into the try block, and then the code to invoke the Web service is completed. Figure 9-60 on page 170 show the complete code to invoke the place order Web service.

```
// Retrieve endpoint from session object, set endpoint, and invoke request
// Get the results properly formatted
Function requestPlaceOrderAction()
    ezcEndpoint string;
    j2eeLib.getSessionAttr("orderEP", ezcEndpoint);
    if (ezcEndpoint == "")
        forward "Please configure web service endpoints" to "ezcErrorDetailsPage";
    end // if
    serviceLib.setWebEndpoint(placeOrder_ServiceBindingLib.DFHOXCMNPort, ezcEndpoint);
    invokePlaceOrderServiceAction();

    // service was successful ... display results to user
    if (output_ca_return_code == 0)
        forward output_ca_response_message to "ezcOrderResponsePage";;
    else
        forward output_ca_response_message to "ezcErrorDetailsPage";
    end // if-else
End // requestPlaceOrderAction()

// invoke the Web Service and handle exceptions
Function invokePlaceOrderServiceAction()
    details eglExceptionDetailsRec;
    try
        placeOrder_ServiceBindingLib.DFHOXCMNPort.DFHOXCMN(
            input_ca_request_id,
            input_ca_return_code,
            input_ca_response_message,
            input_ca_order_request,
            output_ca_request_id,
            output_ca_return_code,
            output_ca_response_message,
            output_ca_order_request);
    onException
        set details empty;
        details.code = sysLib.currentException.code;
        details.description = sysLib.currentException.description;
        case (sysLib.currentException.code)
            when ("com.ibm.egl.ServiceInvocationException")
                details.faultCode = sysLib.currentException.faultCode;
                details.source = sysLib.currentException.source;
                details.location = details.location + sysLib.currentException.location;
                details.diagnostic = sysLib.currentException.diagnostic;
        end // case
        forward details to "ezcExceptionDetailsPage";
    end // try/onException statement
End  // invokePlaceOrderServiceAction
```

Figure 9-60 Customized ezcWebCall template code to invoke place order service

To create a Web service requesting page, you start with the basic steps outlined in the previous section and then take additional steps to interface with the Web service.

To create the place order page:

1. From the Project Explorer view, select the **placeOrder.wsdl** file and select menu choice **Create EGL Interfaces and Binding Library.**
2. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
3. Create a new Faces JSP File named `ezcPlaceOrderPage` using the template.
4. Switch to the Page Designer's **Source** tab.
5. Change the page title to `EGL RUPz Place Order`.
6. Change the page ID to `placeOrder`.
7. Switch to the Page Designer's **Design** tab.
8. Change the header to `Enter Order Details`.
9. Open the EGL palette drawer and drop service on the page. Select the service with the binding **placeOrder_ServiceBindingLib**.
10. Save the JSP file.

11. Remove the compiler errors by modifying the pageHandler variable declarations to resemble Figure 9-16 on page 141. Qualify the type definitions with the package name and shorten the variable name with the input/output prefixes.

12. Add the parameter `inDetails int` to the onPageLoad function and initialize the request parameters:

    – input_ca_request_id = “01ORDR;

    – input_ca_order_request.ca_quantity_req = 1;

    – input_ca_order_request.ca_item_ref_number = inDetails;

    – if (inDetails == 0) input_ca_order_request.ca_item_ref_number = 10; end

13. Use the ezcWebCall EGL template with the following customizations:

    – PlaceOrder

    – orderEP

    – placeOrder

    – ezcOrderResponsePage

14. Replace the comment // Added generated function invocation statement with the appropriate code reference (see Figure 9-60 on page 170).

15. Save the pageHandler file.

16. Edit the JSP to remove the default body content from the page.

17. In the Page Data view, expand **ezcPlaceOrderPage** and **input_ca_order_request**. Then, add controls for ca_item_ref_number**,** ca_quantity_req**,** ca_userid, and ca_charge_dept. On the Insert Controls dialog, check **Create a new record**, order the fields, and change the labels to `Reference Number`; `Quantity`; `User name`; and `Department name`, and ensure the Submit button checkbox is checked on the Options dialog.

18. Reformat the resulting HTML table containing the new visual components:

    a. Select one of the text fields in the third column in the fourth row.

    b. Use the context menu **Table** → **Add Column to Right.**

    c. Use the context menu **Table** → **Add Row Below.**

    d. Drag and drop each display error text field to the column to its right.

    e. Drag and drop the submit button to third column, fifth row.

    f. Set the horizontal alignment property for the Submit button and add its image.

    g. Update the width of all the input fields to `15` characters and require input.

    h. Change the integer fields’ text layout horizontal alignment property to **Right-justified**.

    i. Select the table cell at the first column, fifth row. Use the Properties view to change its width to `200` pixels.

19. From the Page Data view, expand **Actions** and drag **requestPlaceOrderAction** to the **Submit** button.

20. Save the JSP file.

To create the replenish inventory page:

1. From the Project Explorer view, select the **replenishInventory.wsdl** file and select menu choice **Create EGL Interfaces and Binding Library.**

2. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.

3. Create a new Faces JSP File named `ezcReplenishPage` using the template.

4. Switch to the Page Designer's **Source** tab.
5. Change the page title to `EGL RUPz Replenish Inventory`.
6. Change the page ID to `replenish`.
7. Switch to the Page Designer's **Design** tab.
8. Change the header to `Enter Replenish Details`.
9. Open the EGL palette drawer and drop service on the page. Select the service with the binding **replenishInventory_ServiceBindingLib**.
10. Save the JSP file.
11. Remove the compiler errors by modifying the pageHandler variable declarations to resemble Figure 9-16. Qualify the type definitions with the package name and shorten the variable name with the input/output prefixes.
12. Initialize the request parameters in the onPageLoad function:
    - input_ca_request_id = "01REPL";
    - input_ca_order_request.ca_quantity_req = 100;
    - input_ca_order_request.ca_item_ref_number = 10;
13. Use the ezcWebCall EGL template with the following customizations:
    - ReplenishInventory
    - replenishEP
    - replenishInventory
    - ezcReplenishRepsonsePage
14. Replace the comment // Added generated function invocation statement with the appropriate code reference (see Figure 9-60 on page 170).
15. Save the pageHandler file.
16. Edit the JSP to remove the default body content from the page.
17. In the Page Data view, expand **ezcPlaceOrderPage** and **input_ca_order_request**. Then, add controls for ca_item_ref_number, ca_quantity_req, ca_userid, and ca_charge_dept. On the Insert controls dialog, check **Create a new record**, order the fields, and change the labels to `Reference Number`; `Quantity`; `User name`; and `Department name`, and ensure the Submit button checkbox is checked on the options dialog.
18. Reformat the resulting HTML table containing the new visual components:
    a. Select one of the text fields in the third column in the fourth row.
    b. Use the context menu **Table → Add Column to Right.**
    c. Use the context menu **Table → Add Row Below.**
    d. Drag and drop each display error text field to the column to its right.
    e. Drag and drop the submit button to third column, fifth row.
    f. Set the horizontal alignment property for the Submit button and add its image.
    g. Update the width of all the input fields to `15` characters and require input.
    h. Change the integer fields' text layout horizontal alignment property to **Right-justified**.
    i. Select the table cell at the first column, fifth row. Use the Properties view to change its width to `200` pixels.
19. From the Page Data view, expand **Actions** and drag **requestReplenishInventoryAction** to the **Submit** button.

20. Save the JSP file.

Eventually, you might need to verify that a button actually has a function bound to it. Select the button and toggle to the Source tab. The definition of the button should be highlighted. The following text indicates the function requestReplenishAction has been bound to the submit button action="#{ezcReplenishPage.EGLrequestReplenishAction}".

The inquire catalog and inquire single pages have the same general layout as the replenish and place order pages. The inquire pageHandlers need additional coding to populate the formatting EGL record, which will be passed to the inquire response page. To create the inquire single item page:

1. From the Project Explorer view, select the **inquireSingle.wsdl** file and select menu choice **Create EGL Interfaces and Binding Library.**
2. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
3. Create a new Faces JSP File named `ezcInquireSinglePage` using **navigationShell.jtpl**.
4. Switch to the Page Designer's **Source** tab.
5. Change the page title to `EGL RUPz Inquire Catalog Item`.
6. Change the page ID to `inquireONE`.
7. Switch to the Page Designer's **Design** tab.
8. Change the header to `Enter Catalog Item Reference Number`.
9. Open the EGL palette drawer and drop service on the page. Select the service with the binding **inquireSingle_ServiceBindingLib**.
10. Save the JSP file.
11. Remove the compiler errors by modifying the pageHandler variable declarations to resemble Figure 9-16. Qualify the type definitions with the package name and shorten the variable name with the input/output prefixes.
12. Initialize the request parameters in the onPageLoad function:
    - input_ca_request_id = “01INQS”;
    - input_ca_inquire_single.ca_item_ref_req = 10;
13. Use the ezcWebCall EGL template with the following customizations:
    - InquireSingle
    - oneEP
    - inquireSingle
    - formatResults()
14. Change the true block forward statement to just the invocation of formatResults.
15. Code the formatResults function as shown in Figure 9-61 on page 174.

```
// Get results into an EGL record for easy formatting
Function formatResults()
    oldOne mixedcase.com.Response.DFHOXCP4.DFHOXCMN.www.ca_single_item;
    newOne catalogItemRec;
    oldOne = output_ca_inquire_single.ca_single_item;
    newOne.item_ref = oldOne.ca_sngl_item_ref;
    newOne.description = oldOne.ca_sngl_description;
    newOne.in_stock = oldOne.in_sngl_stock;
    newOne.on_order = oldOne.on_sngl_order;
    newOne.money_cost = mathLib.stringAsDecimal(oldOne.ca_sngl_cost);
    prettyList.appendElement(newOne);
    forward prettyList to "ezcInquireResponsePage";
End // formatResults()
```

Figure 9-61 Source code formatResults function in ezcInquireSinglePage.egl

16. Define record catalogItemRec as shown in Figure 9-62.

```
/ Record definition to help format catalog data
ecord catalogItemRec
 item_ref int {displayName = "Item Ref", zeroFormat = yes};
 description string {displayName = "Description" };
 in_stock int {displayName = "In Stock", zeroFormat = yes};
 on_order int {displayName = "On Order", zeroFormat = yes};
 money_cost money(6,2){displayName = "Cost", zeroFormat = yes, currency = yes, numericSeparator = yes };
nd
```

Figure 9-62 Source code catalogItemRec record in ezcInquireSinglePage.egl

17. Add the variable definition `prettyList catalogItemRec[]`; to the pageHandler.
18. Save the pageHandler file.
19. Edit the JSP to remove the default body content from the page.
20. In the Page Data view, expand **ezcInquireSinglePage** and **input_ca_inquire_single**. Then, add controls for ca_item_ref_req. On the Insert controls dialog, check **Create a new record**, change the label to `Part Item Reference` and ensure the Submit button checkbox is checked on the Options dialog.
21. Reformat the resulting HTML table containing the new visual components:
    a. Select one of the text fields in the third column in the first row.
    b. Use the context menu **Table** → **Add Column to Right.**
    c. Use the context menu **Table** → **Add Row Below.**
    d. Drag and drop display error text field to the column to its right.
    e. Drag and drop the Submit button to third column, second row.
    f. Set the horizontal alignment property for the Submit button and add its image.
    g. Update the width of the input fields to `15` characters and require input.
    h. Change the integer field text layout horizontal alignment property to **Right-justified**.
    i. Select the table cell at the first column, third row. Use the Properties view to change its width to `200` pixels.
22. From the Page Data view, expand **Actions** and drag **requestInquireSingleAction** to the **Submit** button.
23. Save the JSP file.

The inquire catalog implementation is basically the inquire single implementation with an extra pageHandler variable to handle validation for the input parameter. Recall the

Elaboration Phase resulted in the discovery that the inquire catalog Web service produced a soap error if the part reference number is greater than 70. To create the inquire catalog page:

1. From the Project Explorer view, select the **inquireCatalog.wsdl file** and select menu choice **Create EGL Interfaces and Binding Library.**
2. Select the folder, **EzcCatalogWebProj\WebContent,** in the Project Explorer view.
3. Create a new Faces JSP File named `ezcInquireCatalogPage` using **navigationShell.jtpl**.
4. Switch to the Page Designer's **Source** tab.
5. Change the page title to `EGL RUPz Inquire Catalog` using the source tab.
6. Change the page ID to `inquireALL` using the source tab.
7. Switch to the Page Designer's **Design** tab.
8. Change the header to `Enter Catalog Item Reference Number`.
9. Open the EGL palette drawer and drop service on the page. Select the service with the binding **inquireCatalog_ServiceBindingLib**.
10. Save the JSP file.
11. Remove the compiler errors by modifying the pageHandler variable declarations to resemble Figure 9-16. Qualify the type definitions with the package name and shorten the variable name with the input/output prefixes.
12. Add pageHandler variable declaration `prettyList catalogItemRec[];`.
13. Initialize the request parameters in the onPageLoad function:
    - input_ca_request_id = “01INQC”;
    - input_ca_inquire_request.ca_list_start_ref = 10;
14. Use the ezcWebCall EGL template with the following customizations:
    - InquireCatalog
    - allEP
    - inquireCatalog
    - formatResults()
15. Change the true block forward statement to just invocation of formatResults.
16. Add the variable definition, `prettyList catalogItemRec[];`, to the pageHandler.
17. Code the formatResults function as shown in Figure 9-53. Include the forward statement to the inquire response page.
18. Save the pageHandler file.
19. Edit the JSP to remove the default body content from the page.
20. In the Page Data view, expand **ezcInquireCatalogPage** and **input_ca_inquire_request.** Then, add controls for ca_list_start_ref. On the Insert controls dialog, check **Create a new record**, change the label to `Start List from Item Reference` and ensure the Submit button checkbox is checked on the Options dialog.
21. Reformat the resulting HTML table containing the new visual components:
    a. Select one of the text fields in the third column in the first row.
    b. Use the context menu **Table** → **Add Column to Right.**
    c. Use the context menu **Table** → **Add Row Below.**
    d. Drag and drop display error text field to the column to its right.
    e. Drag and drop the Submit button to third column, second row.

f. Set the horizontal alignment property for the Submit button and add its image.

g. Update the width of the input field to `15` characters and require input.

h. Change the integer field text layout horizontal alignment property to **Right-justified**.

i. Select the table cell at the first column, third row. Use the Properties view to change its width to `200` pixels.

22. From the Page Data view, expand **Actions** and drag **requestInquireCatalogAction** to the **Submit** button.

23. Save the JSP file.

All four pages that invoke Web services are very similar. The page template and the EGL code template allow the pages to be quickly developed. The only remaining pages are the HTML intensive pages.

9.5.3 HTML intensive pages

To create the inquire response page:

1. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
2. Create a new Faces JSP File named `ezcInquireResponsePage` using **navigationShell.jtpl**.
3. Switch to the Page Designer's **Source** tab.
4. Change the page title to `EGL RUPz Inquire Results`.
5. Change the page ID to `inquireResponse`.
6. Switch to the Page Designer's **Design** tab.
7. Change the header to `Item Details - Select Item to Place Order`.
8. Select **Edit Page Code** context menu option.
9. Add the pageHandler variable declaration `catalogList catalogItemRec[];`.
10. Add the pageHandler variable declaration `selected int[] {selectFromListItem="catalogList", selectType = index};`.
11. Add the parameter `inputList catalogItemRec[]` to the onPageLoad function and initialize the pageHandler variable with the parameter:
    – catalogList = inputList;
12. Add the function `requestPlaceOrderAction` as specified in Figure 9-63 on page 177.

```
package pagehandlers;

import com.ibm.egl.jsf.*;

PageHandler ezcInquireResponsePage
    {onPageLoadFunction = "onPageLoad",
     view = "ezcInquireResponsePage.jsp",
     viewRootVar = "viewRoot"}

    viewRoot UIViewRoot;

    catalogList catalogItemRec[];
    // Bind to row Selection column
    selected int[] {selectFromListItem="catalogList", selectType = index};


    // Function Declarations
    Function onPageLoad(inputList catalogItemRec[])
        catalogList = inputList;
    End

    Function requestPlaceOrderAction()
        hit catalogItemRec;
        hitItem int;
        if (selected.getSize() > 0)
            hit = catalogList[selected[1]];
            hitItem = hit.item_ref;
            forward hitItem to  "ezcPlaceOrderPage";
        else
            forward "Select a part" to "ezcErrorDetailsPage";
        end // if-else
    End // requestPlaceOrder()
```

Figure 9-63 Source code for ezcInquireResponsePage.egl

13. Save the pageHandler file.
14. Edit the JSP to remove the default body content from the page.
15. Open the HTML Tags drawer and drop an HTML table into the body of the page. The table should have 3 rows, 1 column, 100% width, and no border.
16. In the Page Data view, expand **ezcInquireResponsePage** and add controls for catalogList into the first row of the table. EGL will use the record definition to get the column labels and column order, so just accept the defaults.
17. Change the HTML table properties, so it takes 100% of the page width.
18. Drag and drop the Submit button from the Faces Component drawer into the third row. Set the horizontal alignment property to **Right** and add its image.
19. From the Page Data view, expand **Actions** and drag **requestPlaceOrderAction** to the Submit button.
20. Select the data table, go to its Properties view, shown in Figure 9-64 on page 178, select the **Row Action** tab, and click **Add Selection Column to the table**.

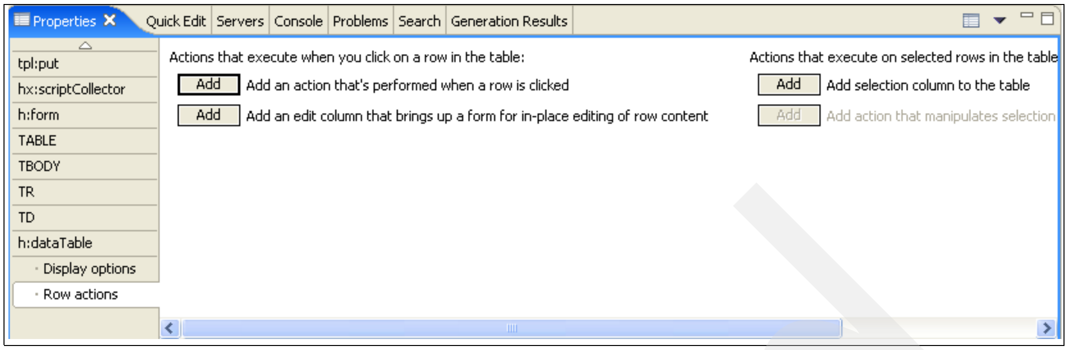


Figure 9-64 Row actions Properties view

21. Select the **h:dataTable** tab, shown in Figure 9-65, to change the order of the columns and add the **Select** label to the new column.

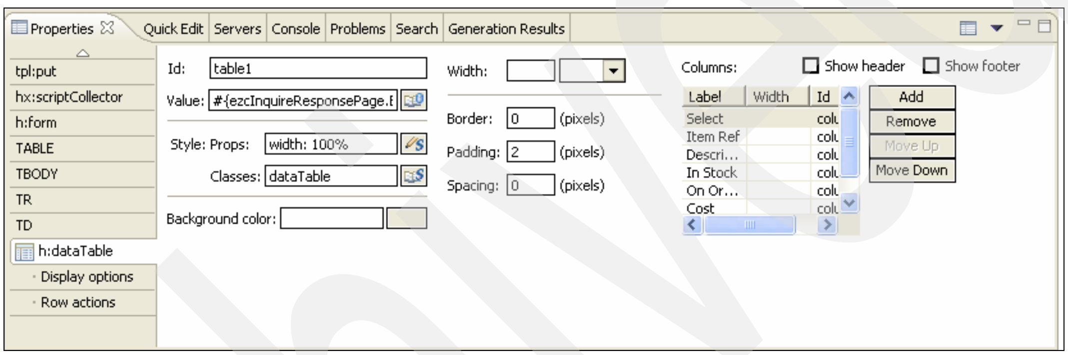


Figure 9-65 DataTable property view on the h:dataTable tab

22. In the Page Data view, expand **ezcInquireResponsePage,** select **selected**, and drag the data variable to checkbox in the newly added selected column.

23. You must modify the value property of this column from **#{ezcinquireResponsPage.EGLselected.nullAsIntegerArray** to `#{ezcInquireResponsePage.EGLselectedAsIntegerArray`. Figure 9-66 shows the view used to make this change.

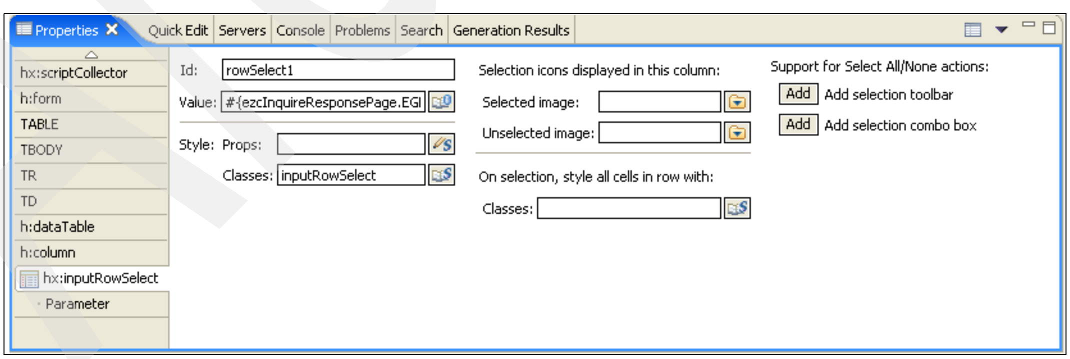


Figure 9-66 Column property view on the hx:inputRowSelect tab

24. Change the horizontal alignment of the column data. Cost is right-aligned. Item Ref, In Stock, On Order, and Select are centered.

25. Save the JSP file.

The last page to develop is the configure page. It has the most involved page layout as well as the longest pageHandler. To create the configure page:

1. Select the folder **EzcCatalogWebProj\WebContent** in the Project Explorer view.
2. Create a new Faces JSP File named `ezcConfigurePage` using **navigationShell.jtpl**.
3. Switch to the Page Designer's **Source** tab.
4. Change the page title to `EGL RUPz Catalog Management Configuration`.
5. Change the page ID to `configure`.
6. Switch to the Page Designer's **Design** tab.
7. Change the header to `Configure Application`.
8. Select the Edit Page Code context menu option.
9. Add the variable declarations specified in Figure 9-67 to the pageHandler.

```
package pagehandlers;

import com.ibm.egl.jsf.*;
import mixedcase.com.DFHOXCP3.DFHOXCMN.www.inquireCatalog_ServiceBindingLib;
import mixedcase.com.DFHOXCP4.DFHOXCMN.www.inquireSingle_ServiceBindingLib;
import mixedcase.com.DFHOXCP5.DFHOXCMN.www.placeOrder_ServiceBindingLib;
import mixedcase.com.DFHRUP4Z.DFHOXCMN.www.replenishInventory_ServiceBindingLib;

PageHandler ezcConfigurePage
    {onPageLoadFunction = "onPageLoad",
     view = "ezcConfigurePage.jsp",
     viewRootVar = "viewRoot"}

    viewRoot UIViewRoot;
    currentInquireAllEP string;
    newInquireAllEP string;
    currentInquireOneEP string;
    newInquireOneEP string;
    currentOrderEP string;
    newOrderEP string;
    currentReplenishEP string;
    newReplenishEP string;
```

Figure 9-67 Import statements and variable declarations for ezcConfigurePage

10. Add the new functions specified in Figure 9-68 on page 180 to the pageHandler.

```
//  Function requestConfigure
Function requestConfigureEndpointAction()
    j2eeLib.setSessionAttr("allEP", newInquireAllEP);
    j2eeLib.setSessionAttr("oneEP", newInquireOneEP);
    j2eeLib.setSessionAttr("orderEP", newOrderEP);
    j2eeLib.setSessionAttr("replenishEP", newReplenishEP);

    // setWebEndpoint successful so update currentEndpoint
    currentInquireAllEP = newInquireAllEP;
    currentInquireOneEP = newInquireOneEP;
    currentOrderEP = newInquireOneEP;
    currentReplenishEP = newReplenishEP;
End // requestConfigureEndpointAction

//  Reset
Function requestResetEndpointAction()
    newInquireAllEP = currentInquireAllEP;
    newInquireOneEP = currentInquireOneEP;
    newInquireOneEP = currentOrderEP;
    newReplenishEP = currentReplenishEP;
End // requestResetEndpointAction
```

Figure 9-68   Action functions for ezcConfigurePage.egl

11.Modify the onPageLoad function to initialize the endpoint variables such as Figure 9-69.

```
//look at session variable first
Function onPageLoad()
    j2eeLib.getSessionAttr("allEP", currentInquireAllEP);
    if (currentInquireAllEP == "")
        currentInquireAllEP = serviceLib.getWebEndpoint(inquireCatalog_Se
    end
    newInquireAllEP = currentInquireAllEP;

    j2eeLib.getSessionAttr("oneEP", currentInquireOneEP);
    if (currentInquireOneEP == "")
        currentInquireOneEP = serviceLib.getWebEndpoint(inquireSingle_Ser
    end
    newInquireOneEP = currentInquireOneEP;

    j2eeLib.getSessionAttr("orderEP", currentOrderEP);
    if (currentOrderEP == "")
        currentOrderEP = serviceLib.getWebEndpoint(placeOrder_ServiceBinc
    end
    newOrderEP = currentOrderEP;

    j2eeLib.getSessionAttr("replenishEP", currentReplenishEP);
    if (currentReplenishEP == "")
        currentReplenishEP = serviceLib.getWebEndpoint(replenishInventory
    end
    newReplenishEP = currentReplenishEP;
End
```

Figure 9-69   Source code for the onPageLoad function in ezcConfigurePage.egl

12.Save the pageHandler changes.

13.Open the HTML Tags drawer and drop an HTML table on the JSP page. Figure 9-70 on page 181 shows the creation parameters.

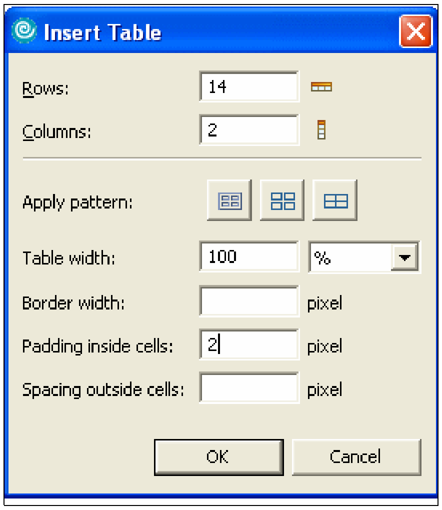


Figure 9-70 HTML table creation dialog

14. To make the labels span the entire table, you will have to join together two cells. Select two adjacent cells in the first row while holding down the Ctrl key. Figure 9-71 shows the result of this selection.

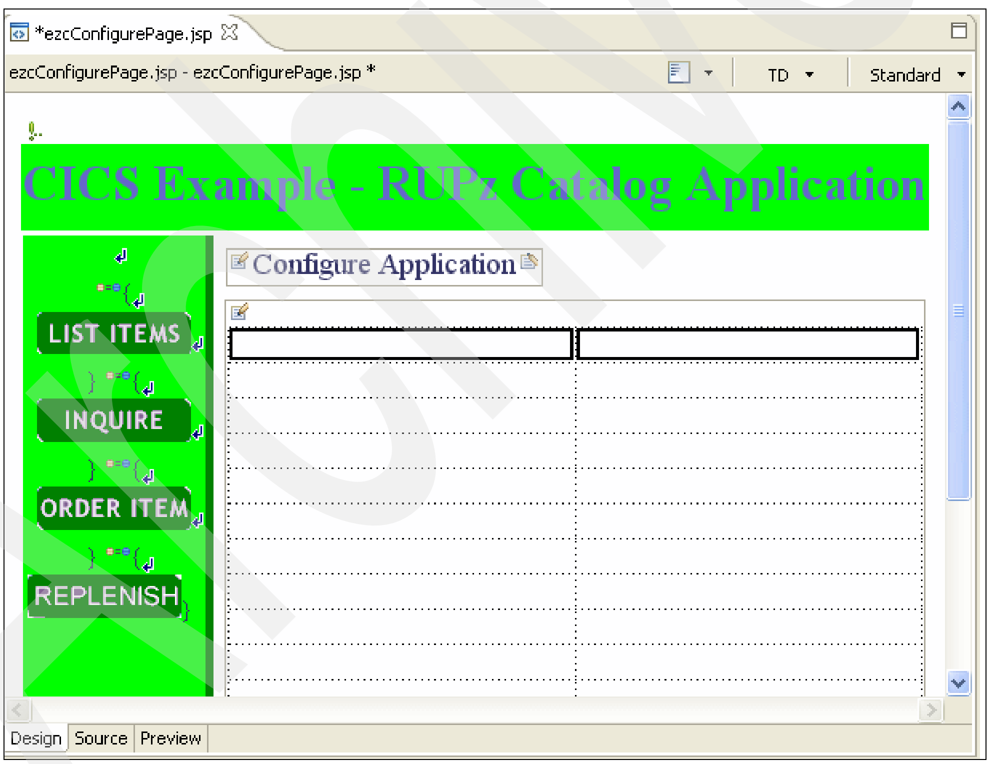


Figure 9-71 HTML table with two cells selected

15. Using the context menu, select **Table** → **Join Selected Cells** to create one cell.

16. Do this for rows 4, 7, and 10.

17. Use the Ctrl key to select rows 1, 4, 7, and 10. Modify the background color to lime on the Properties view.

18. Enter static text into these cells:
    - Inquire Catalog Service Endpoint
    - Inquire Item Service Endpoint
    - Place Order Service Endpoint
    - Replenish Inventory Service Endpoint
19. Change column width to `15%` of the second row, first column.
20. Enter static text of `Current` into column 1 of rows 2, 5, 8, and 11.
21. Enter static text of `New` into column 1 of rows 3, 6, 9, and 12.
22. Now the page looks like Figure 9-72.

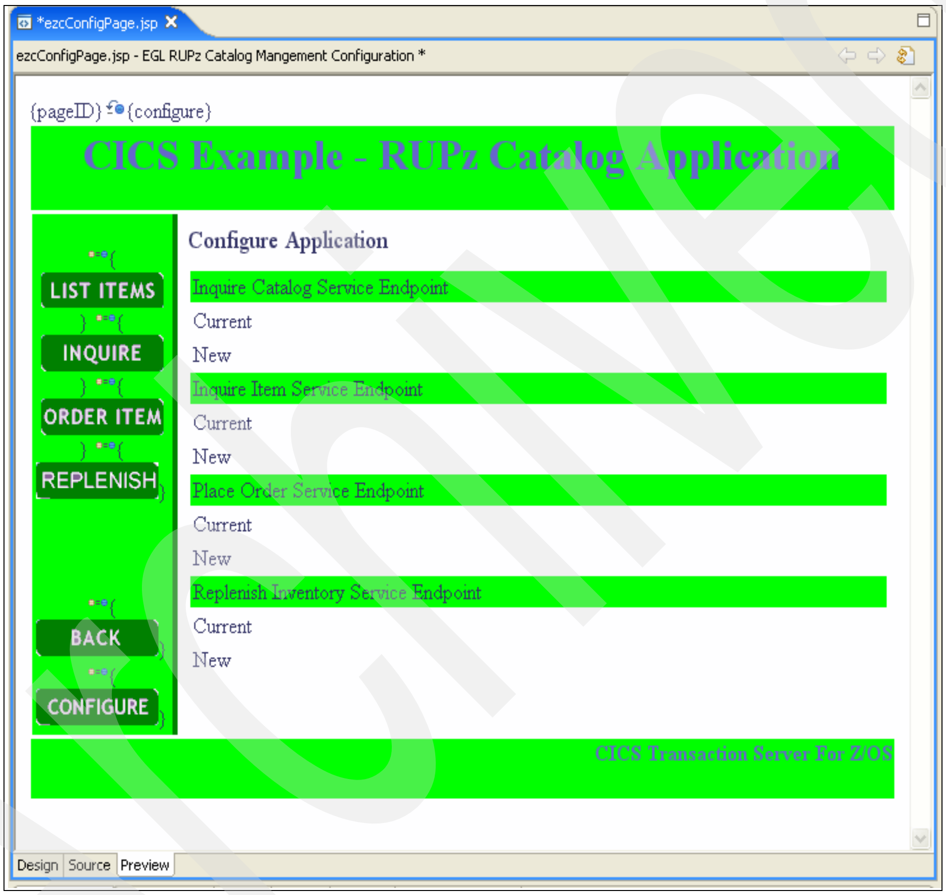


Figure 9-72 Configure Page with basic table layout

23. Open the Faces Components drawer on the palette. For column 2 on rows 2, 5, 8, and 11, drop output text.
24. In the Page Data view, expand **ezcConfigurePage**. Drag the current*XXX* page data variables and bind them to the appropriate output text field.
25. From the Faces Components drawer, drop input text. For column 2 on rows 3, 6, 9, and 12, drop input text.
26. In the Page Data view, expand **ezcConfigurePage**. Drag the new*XXX* page data variables and bind them to the appropriate input text field.
27. Also, modify the input text component to have borders of `0` thickness. Figure 9-73 on page 183 shows where to make this modification on the Set Style Properties dialog.

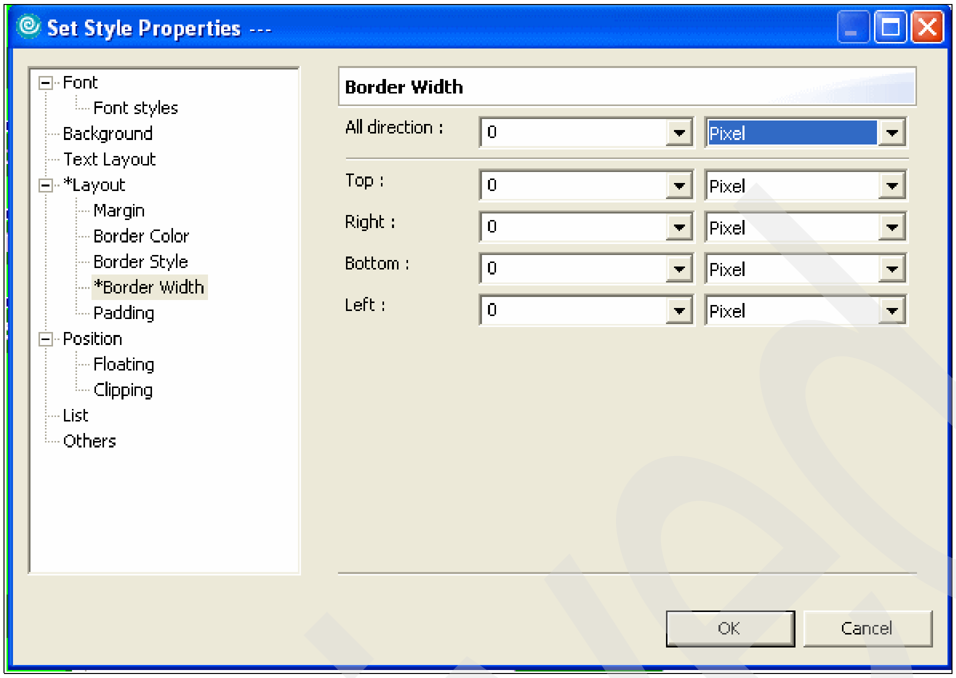


Figure 9-73 Set Style Properties dialog to modify border width

28. For each of the input fields, change the validation so that a value is required and make the cell width `100%`.
29. Drag the Command button from the Faces Component drawer to each of the last two rows, center their horizontal alignment, and add graphics. The top button is the SUBMIT. The bottom button is RESET.
30. Bind the action `requestConfigureEndpointsAction` to the top button and `requestResetEndpointsAction` to the bottom button, which results in Figure 9-74 on page 184.

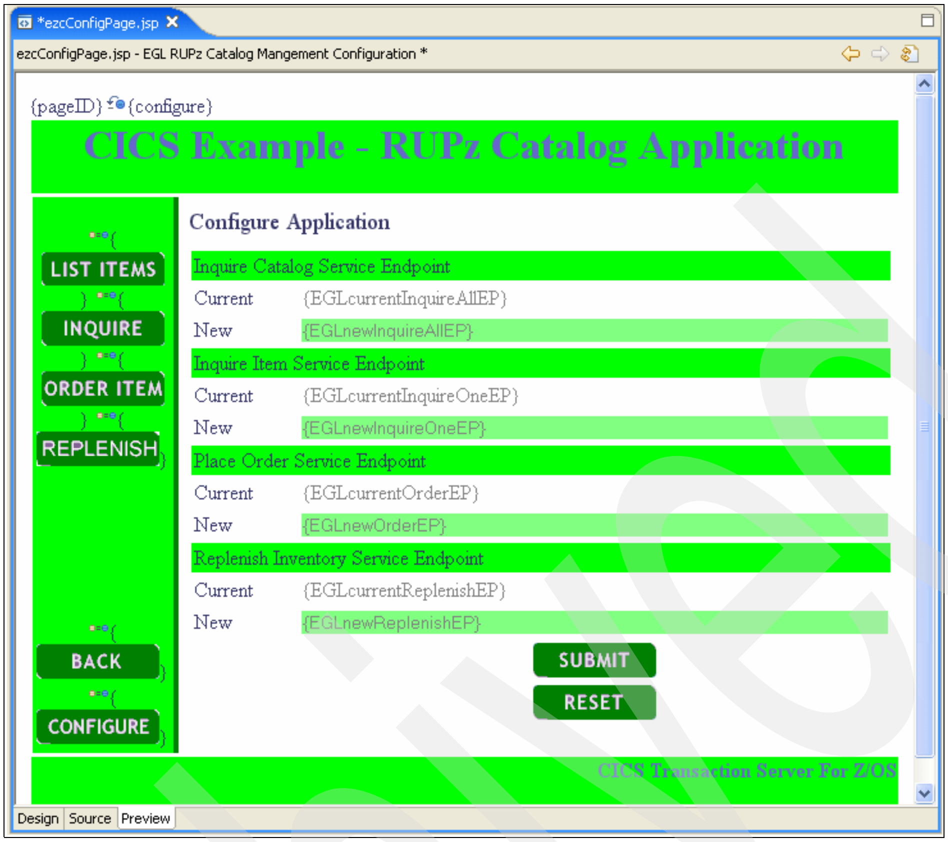


Figure 9-74 Completed preview of ezcConfigurePage.jsp

31.Save the page.

The application is complete and ready for testing.

9.5.4 Test scenario

The steps to test the scenario are:

1. Go to the Server view, select a server, and select **Start** from the context menu. After the server status is **Started**, select the **ezcWelcomePage.jsp** in the Project Explorer view and select **Run on Server**. Figure 9-75 on page 185 shows the welcome page for the application.

Figure 9-75 Welcome to the CICS RUPz Catalog Application Page

2. Before invoking any service, the Web service endpoints must be configured. Select the **CONFIGURE** button to get to the configuration page shown in Figure 9-76 on page 186.

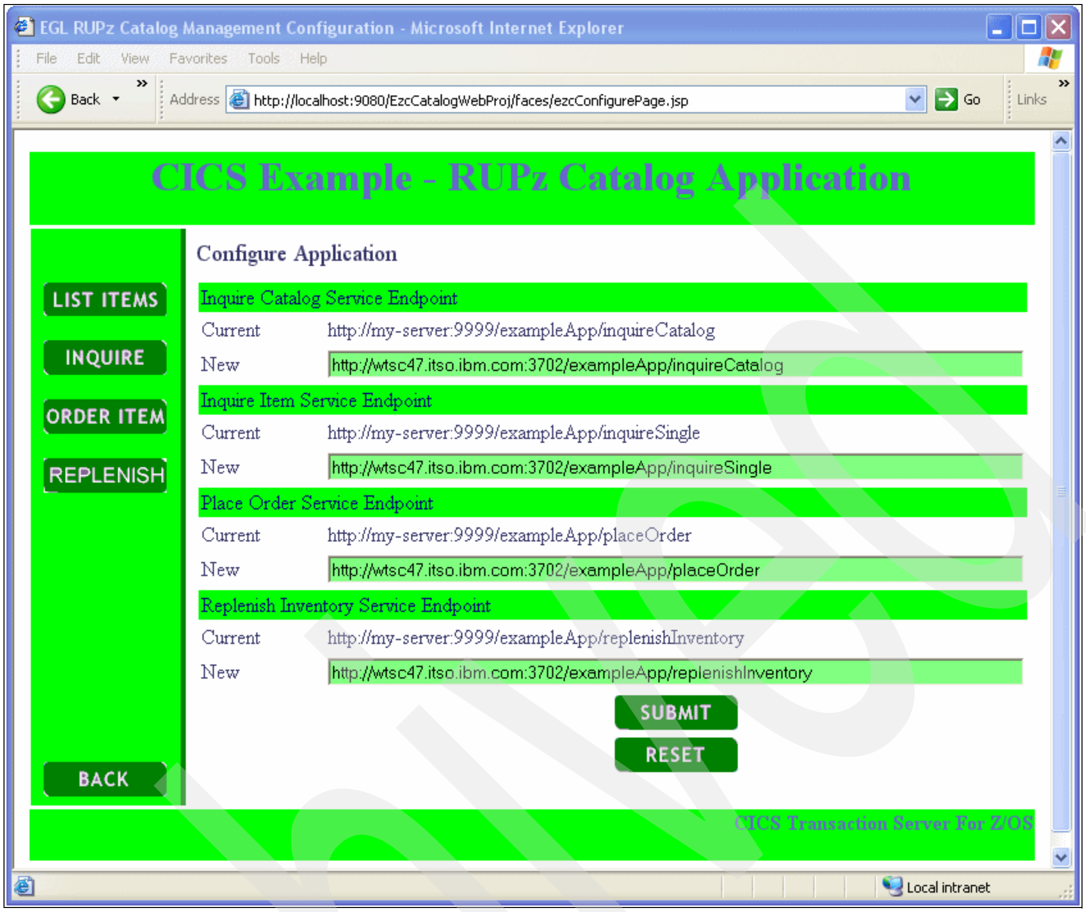


Figure 9-76 EGL RUPz Catalog Management Configuration Page

3. Notice the **CONFIGURE** navigational button is not available, because the configure page is active. Update all the **New** text fields with the location of your Web services and select the **SUBMIT** button. Test the inquire catalog service by selecting **LIST ITEMS** and enter 50 in the input field as shown in Figure 9-77 on page 187.

Figure 9-77 EGL RUPz Inquire Catalog Page

4. Verify that the **LIST ITEMS** navigation button is not available and select **Submit**. The results are shown in Figure 9-78 on page 188.

Figure 9-78 EGL RUPz Inquire Results Page for catalog inquire

5. Verify the alignment of all the column data. Verify the format of the Cost column. Select the **SUBMIT** button without selecting a part to verify that you get an application error message indicating a part must be selected. Use the **BACK** button to retry the **SUBMIT** button after selecting part item number 50. Enter values in the place order page similar to those values shown in Figure 9-79 on page 189.

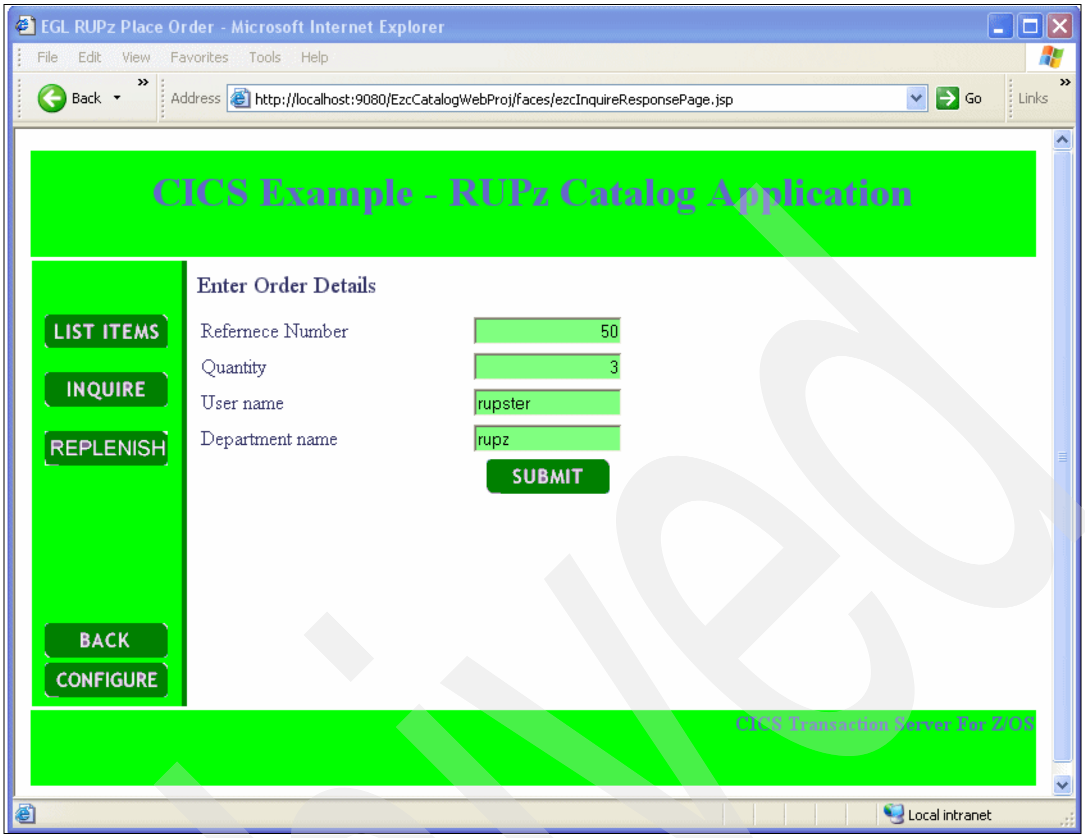


Figure 9-79 EGL RUPz Place Order Page

6. Leave any field empty to see a validation error message. Entering character data in the numeric fields will also result in a validation error. With all the fields filled with appropriately formatted data, the **SUBMIT** button will result in the informational page shown in Figure 9-80 on page 190.

Figure 9-80 EGL RUPz Order Results Page

7. Select the **INQUIRE** button to test the inquire single Web service. Figure 9-81 on page 191 shows the inquire item request page.

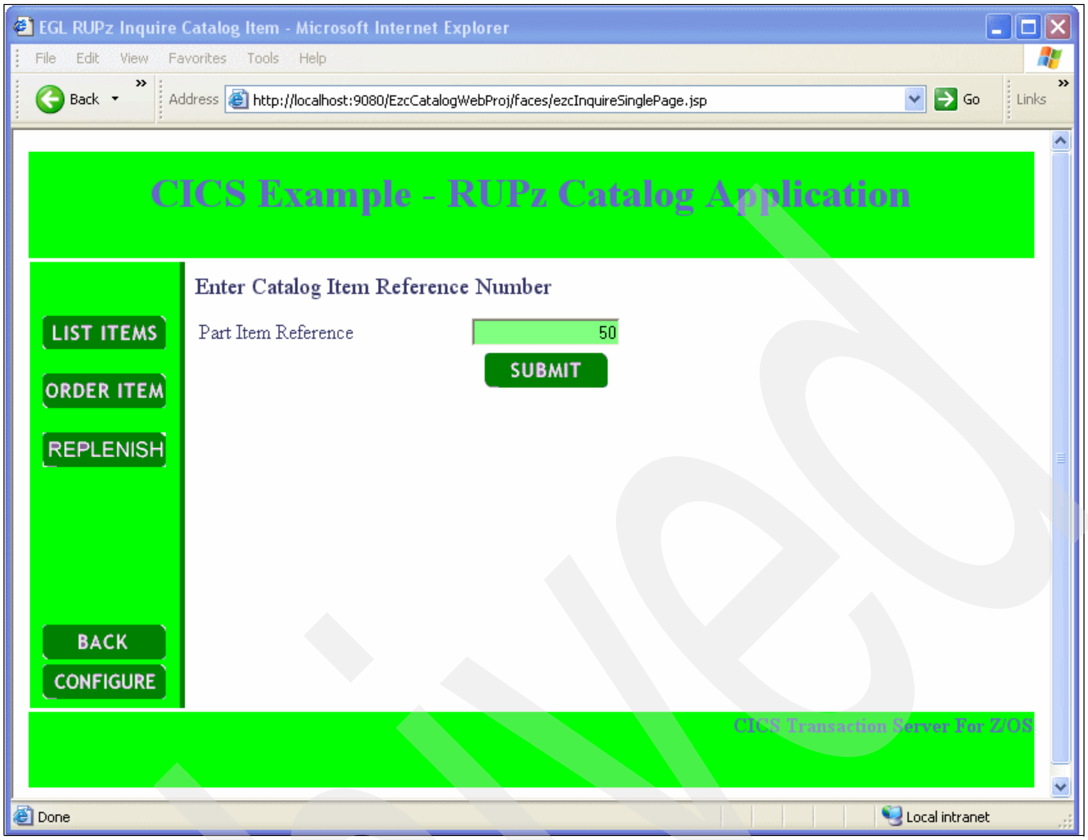


Figure 9-81 EGL RUPz Inquire Catalog Item Page

8. Verify that the **INQUIRE** button is not available. Enter `50` as the part item reference in order to verify that the in stock and on order amounts have been properly adjusted by the quantity ordered. Reference Figure 9-78 on page 188 for the original amounts and Figure 9-79 on page 189 for the quantity ordered. Figure 9-82 on page 192 shows the result of the inquire of a single part.

Figure 9-82 EGL RUPz Inquire Results Page for inquire single

9. Verify the data table content and data format. You should verify that you get an error message if you use the **SUBMIT** button without the part selected. Use the **BACK** button to try the **SUBMIT** button with the item selected to verify that the application navigates to the place order page. Select the **REPLENISH** button to get to the page shown in Figure 9-83 on page 193.

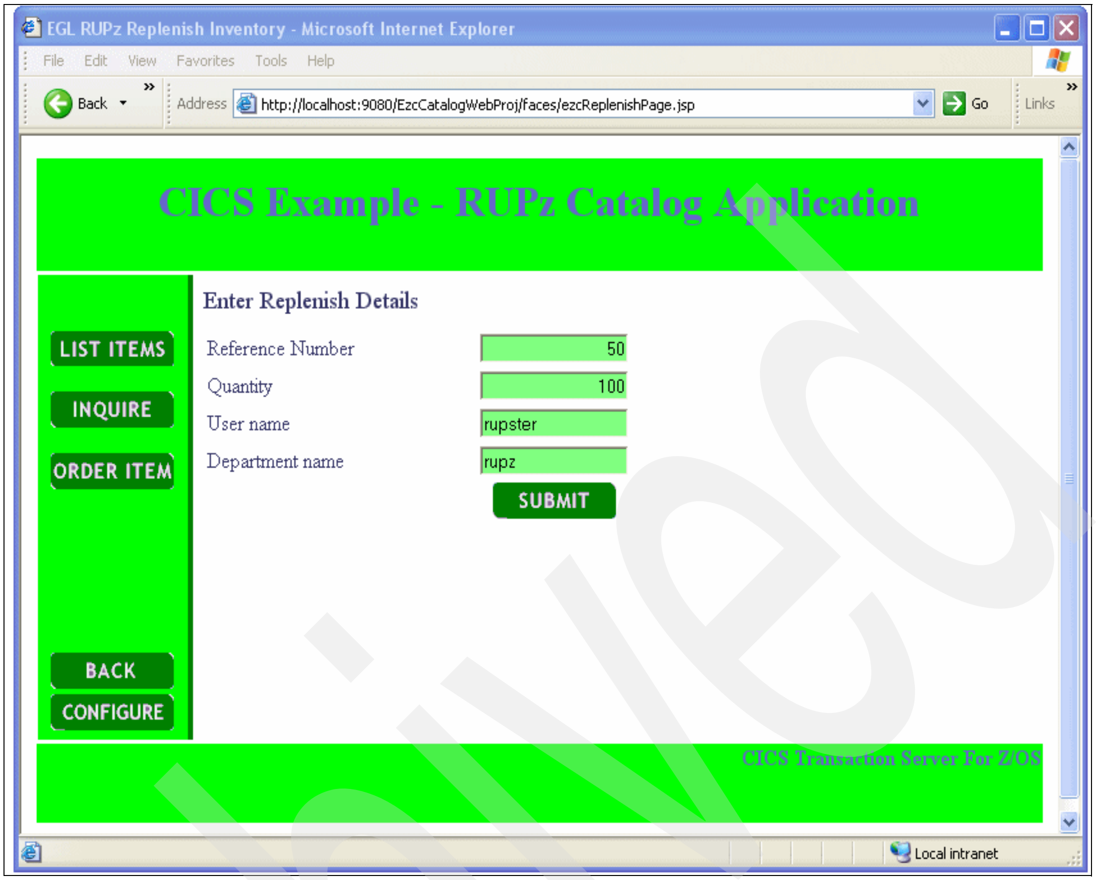


Figure 9-83 EGL RUPz Replenish Inventory Page

10. Leave any field empty to see a validation error message. Entering character data in the numeric fields will also result in a validation error. With all the fields filled with appropriately formatted data, the **SUBMIT** button will result in the informational page shown in Figure 9-84 on page 194.

Figure 9-84 EGL RUPz Replenish Inventory Results Page

11. Test the application using invalid Web service endpoints. Navigate to the configure application page and modify the inquire single Web service endpoint to have an invalid server address. Invoke the inquire single Web service to verify the system exception details page shown in Figure 9-85 on page 195.

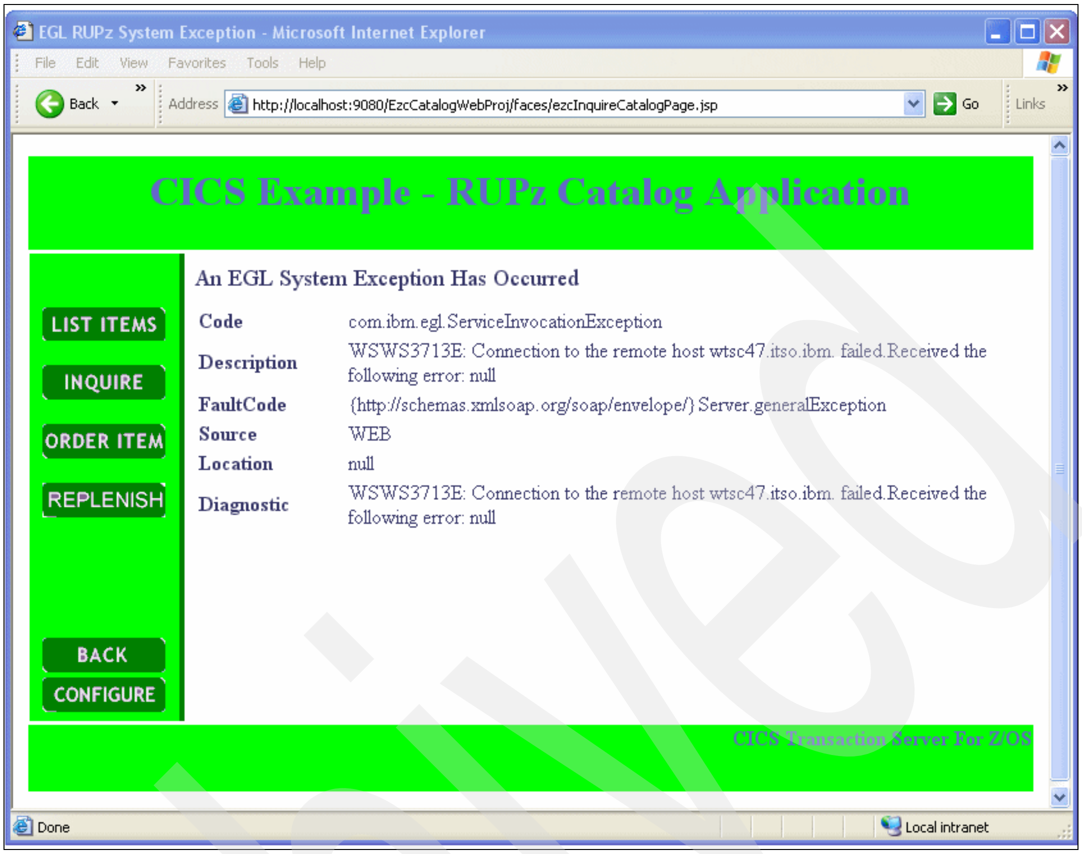


Figure 9-85 EGL RUPz System Exception Page for invalid server address

12. Go back to the configure page and set the inquire single endpoint to the valid host and strip several characters from the end of the valid endpoint. Retry the inquire single request and verify the system exception page resembles Figure 9-86 on page 196.

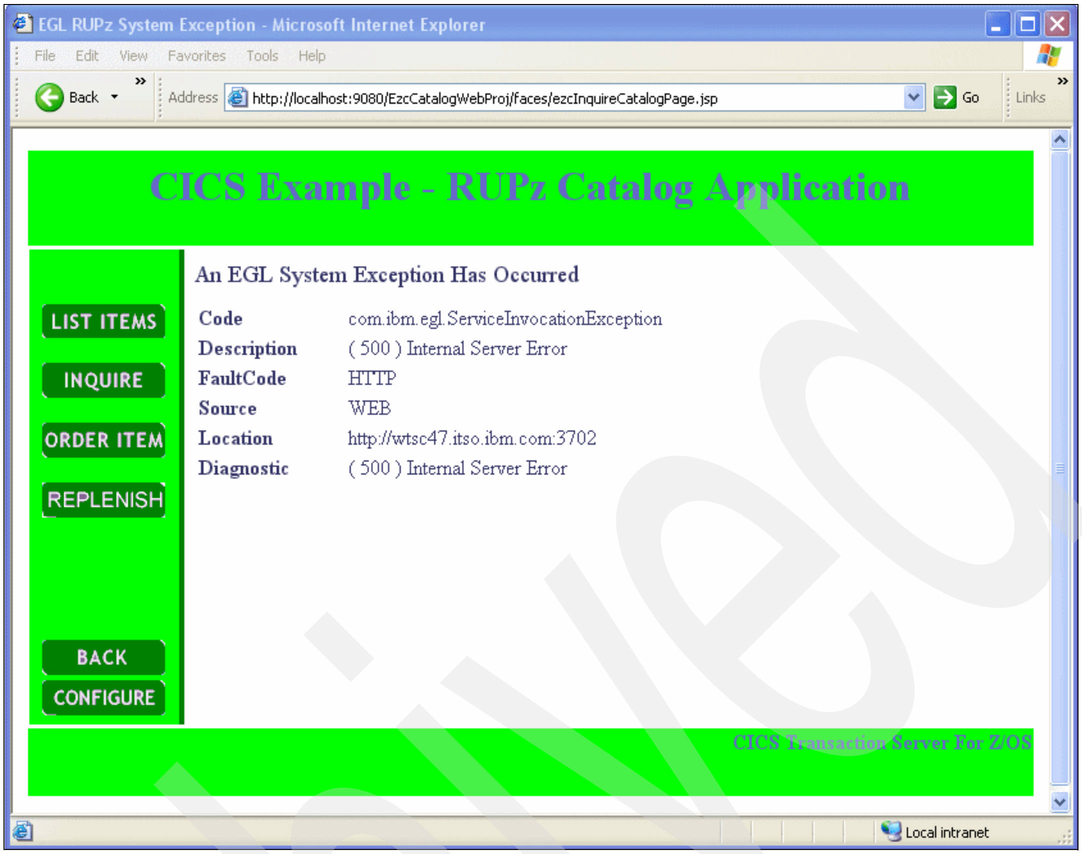


Figure 9-86 EGL RUPz System Exception for invalid service location

13. Now, verify that the replenish inventory Web service did adjust all the in stock values to 100. Use 10 as the starting part reference number. Figure 9-87 on page 197 reveals that the in stock values were reset to 100.

Figure 9-87 EGL RUPz Inquire Results for catalog inquire after replenish inventory

14. When you can successfully invoke each Web service, you should explore the affects of stopping and starting the server as well as stopping the application. If you close the Web browser and reselect the **Run on Server** menu option, you can immediately issue an inquire Web service because the server is still holding on to the cached endpoints. To relinquish the endpoints, you must stop the server.

9.6 Transition Phase

In accordance with the Catalog Manager Software development plan, no EGL specific activities occurred in the Transition Phase.

9.7 Summary

This chapter is a step-by-step guide to help you recreate the EGL portion of the RUP for z Catalog Manager sample application. If you work through this chapter, you will discover how EGL enables you to develop a Web application very quickly.

Part 4

IBM RUP for System z for Method Designers and Project Managers

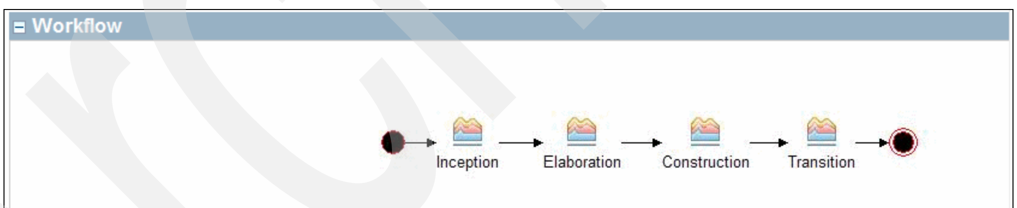
This part includes material handy to customize the IBM Rational Unified Process for System z (RUP for System z) to better address your project needs. This part is targeted toward project managers and method designers.

10

IBM RUP for System z Work Breakdown Structure

The IBM Rational Unified Process for System z (RUP for System z) includes a Work Breakdown Structure that covers the whole development lifecycle from beginning to end, as illustrated in Figure 10-1. This Work Breakdown Structure can be used as a template for planning and running a project. This chapter presents the Work Breakdown Structure for each project phase (inception, elaboration, construction, and transition). Refer to the RUP for System z Web site for more details about the topic. The RUP for System z Web site can be generated out of the RUP for System z RMC plug-in from IBM developerWorks at:

http://www.ibm.com/developerworks/rational/downloads/06/rmc_plugin7_1/



Work Breakdown

Breakdown Element	Steps	Index	Predecessors	Model Info	Optional	Planned	Repeatable	Multiple Occurrences
Inception		1				✔		
Elaboration		33	1			✔		
Construction		94	33			✔		
Transition		151	94			✔		

Figure 10-1 The RUP for System z Workflow and Work Breakdown Structure

10.1 Inception Phase

Figure 10-2 presents the Work Breakdown Structure for a typical iteration in inception. This Work Breakdown Structure includes activities and tasks with predecessors. The number of steps per task is shown with blue dots.

Work Breakdown

Breakdown Element	Steps	Index	Predecessors
Inception Iteration(s)		2	
Conceive New Project		3	
Develop Business Case	●●●●●●	4	
Identify and Assess Risks	●●●●●●●	5	4
Project Approval Review	●●●●	6	5
Initiate Project	●●●●	7	6
Prepare Project Environment		8	3
Tailor the Development Process for the Project	●●●●●●●	9	
Select and Acquire Tools	●●●●●	10	
Set Up Tools	●●●●●	11	10
Set Up Configuration Management (CM) Environment	●●●●	12	10
Define Requirements		13	8
Develop Vision	●●●●●●●	14	
Find Actors and Use Cases	●●●●●●●	15	14
Prioritize Use Cases	●●●	16	15
Develop Supplementary Specifications	●●●●●	17	14
Capture a Common Vocabulary	●●	18	
Define Test Approach	●●●●●●●●●●●	19	14
Perform Architectural Proof-of-Concept		20	13
Architectural Analysis	●●●●●●●●	21	
Construct Architectural Proof-of-Concept	●●●	22	21
Assess Viability of Architectural Proof-of-Concept	●●●	23	22
Plan the Project		24	20
Assess Iteration	●●●●●	25	
Identify and Assess Risks	●●●●●●●	26	25
Plan Phases and Iterations	●●●●●●●	27	
Develop Iteration Plan	●●●●	28	27
Acquire Staff	●●●●	29	
Prepare for Project Close-Out	●●●●●	30	
Project Acceptance Review	●●●●	31	30
Lifecycle Objectives Milestone		32	2

Figure 10-2 Inception Work Breakdown Structure

10.2 Elaboration Phase

Figure 10-3 presents the Work Breakdown Structure for a typical iteration in elaboration. This Work Breakdown Structure includes activities and tasks with predecessors. The number of steps per task is shown with blue dots.

⊟ Work Breakdown			
Breakdown Element	**Steps**	**Index**	**Predecessors**
⊟ Elaboration Iteration(s)		34	
⊟ Refine Requirements		35	
Detail a Use Case [within Scope]	••••••••••	36	
Prioritize Use Cases	•••	37	36
Develop Supplementary Specifications	•••••	38	
Capture a Common Vocabulary	••	39	
⊟ Define Architecture		40	35
⊞ Define Candidate Architecture		41	
⊞ Refine the Architecture		46	41
⊟ Design Components		51	40
⊞ Perform Component Design		52	
⊞ Design Services		58	
⊞ Design DataBases		61	
⊞ Design User Interface		63	
⊞ Review the Design		66	52,58,63,61
⊟ Code and Unit Test Components		68	51
Implement Design Elements	•••••	69	
Review Code	•••	70	69
Implement Developer Test	••••••	71	69
Execute Developer Tests	•••••	72	71
⊟ Integrate and Test		73	68
⊞ Integrate		74	
⊞ Test		77	
⊟ Plan the Project		85	73
Assess Iteration	•••••	86	
Identify and Assess Risks	•••••••	87	86
Plan Phases and Iterations	•••••••	88	
Develop Iteration Plan	••••	89	88
Acquire Staff	••••	90	
Prepare for Project Close-Out	•••••	91	
Project Acceptance Review	••••	92	91
Lifecycle Architecture Milestone		93	34

Figure 10-3 Elaboration Work Breakdown Structure

10.3 Construction Phase

Figure 10-4 presents the Work Breakdown Structure for a typical iteration in construction. This Work Breakdown Structure includes activities and tasks with predecessors. The number of steps per task is shown with blue dots.

Breakdown Element	Steps	Index	Predecessors
Construction Iteration(s)		95	
Refine Requirements		96	
Detail a Use Case [within Scope]	••••••••••	97	
Prioritize Use Cases	•••	98	97
Develop Supplementary Specifications	•••••	99	
Capture a Common Vocabulary	••	100	
Design Components		101	96
Perform Component Design		102	
Design Services		108	
Design DataBases		111	
Design User Interface		113	
Review the Design		116	102,108,113,111
Code and Unit Test Components		118	101
Implement Design Elements	•••••	119	
Review Code	•••	120	119
Implement Developer Test	••••••	121	119
Execute Developer Tests	•••••	122	121
Integrate and Test		123	118
Integrate		124	
Test		127	
Prepare Deployment		135	123
Develop Deployment Plan	••••••	136	
Develop Support Materials		137	
Develop Installation Work Products		138	
Develop Training Materials	••	139	
Create Product Artwork		140	
Define Bill of Materials	••	141	
Plan the Project		142	135
Assess Iteration	•••••	143	
Identify and Assess Risks	•••••••	144	143
Plan Phases and Iterations	•••••••	145	
Develop Iteration Plan	••••	146	145
Acquire Staff	••••	147	
Prepare for Project Close-Out	•••••	148	
Project Acceptance Review	••••	149	148
Initial Operational Capability Milestone		150	95

Figure 10-4 Construction Work Breakdown Structure

10.4 Transition Phase

Figure 10-5 presents the Work Breakdown Structure for a typical iteration in transition. This Work Breakdown Structure includes activities and tasks with predecessors. The number of steps per task is shown with blue dots.

⊟ Work Breakdown			
Breakdown Element	**Steps**	**Index**	**Predecessors**
⊟ Transition Iteration(s)		152	
⊟ Code and Unit Test Components		153	
Implement Design Elements	●●●●●	154	
Review Code	●●●	155	154
Implement Developer Test	●●●●●●	156	154
Execute Developer Tests	●●●●●	157	156
⊟ Integrate and Test		158	153
⊟ Integrate		159	
Integrate Subsystem	●●	160	
Integrate System	●●	161	160
⊟ Test		162	
Define Test Approach	●●●●●●●●●●●	163	
Define Test Details	●●●●●●●	164	163
Implement Test	●●●●●●●●	165	164
Define Installation Verification Procedures (IVPs)	○○○	166	163
Implement Installation Verification Procedures (IVPs)	○○○	167	166
Execute Test Suite	●●●●●●●●●●	168	165,167
Analyze Test Failure	●●●●●●●●	169	168
⊟ Perform Beta and Acceptance Test		170	158
⊞ Perform Beta Test		171	
⊞ Perform Acceptance Test		173	
⊟ Package Product		178	170
⊞ Produce Product Documentation		179	
⊞ Produce Deployment Unit		186	
⊞ Release to Manufacturing		188	179,186
⊟ Plan the Project		192	178
Assess Iteration	●●●●●	193	
Identify and Assess Risks	●●●●●●●	194	193
Plan Phases and Iterations	●●●●●●●	195	
Develop Iteration Plan	●●●●	196	195
Acquire Staff	●●●●	197	
Prepare for Project Close-Out	●●●●●	198	
Project Acceptance Review	●●●●	199	198
Product Release Milestone		200	152

Figure 10-5 Transition Work Breakdown Structure

11

How to customize the IBM Rational Unified Process for System z

One of the most common needs related to the implementation of a process is its customization. Typically, people don't like to invent a brand new process; indeed, they prefer to adopt an existing one that permits them to avoid spending time and money creating something new from scratch. Since there is no process that fits everybody, you need to customize your RUP for System z. In this chapter we discuss the purpose and the target of customization explaining how to create a project plan specific to your project and how to customize the RUP for System z using Rational Method Composer.

11.1 Introduction

The Rational Unified Process framework provide guidance on a rich set of software engineering principles. It is applicable to projects of different sizes and complexities, but this means that no single project will benefit from using all of RUP. This concept is also applicable to an already tailored process (for example, you can think about how to introduce further customization to RUP for Small Projects) that can be tailored in order to meet some specific project needs. This concept is also valid for RUP for System z, that is, you can begin to plan your project starting from activities and tasks already defined in the delivery process, but you can also realize that some specific project needs might drive you to add, modify, and customize certain process elements. In particular, the *Prepare Project Environment* activity in the Inception Phase of the RUP for System z, shown in Figure 11-1, prepares the development environment for a project, where the development environment includes both process and tools, primarily affected by the results obtained from the *Tailor the Development Process for the Project* task.

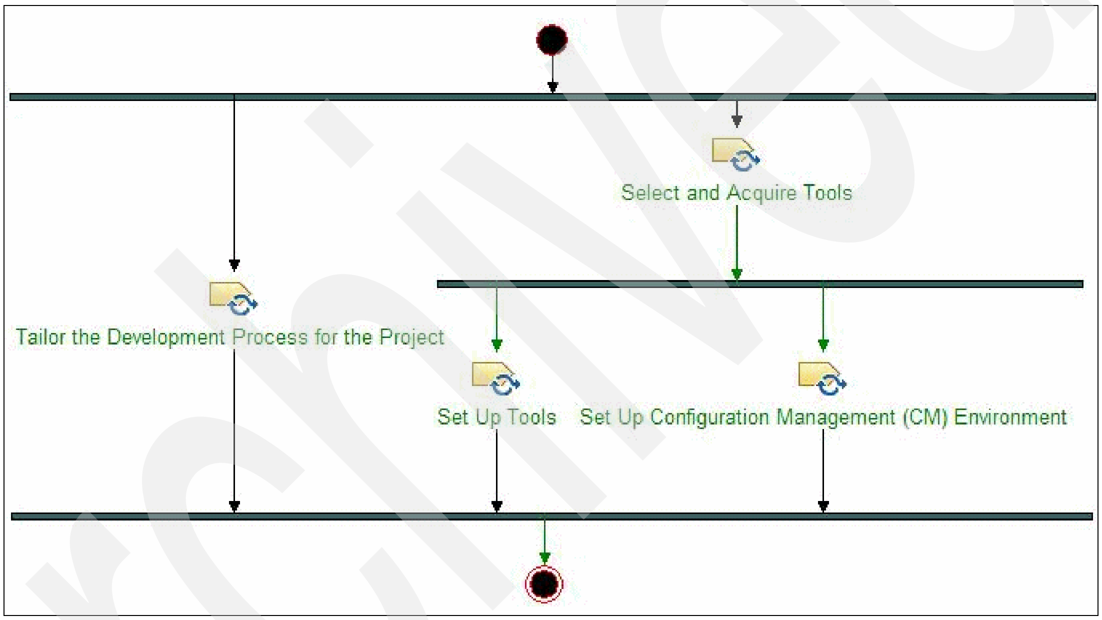


Figure 11-1 Prepare Project Environment activity in Inception Phase

It is crucial for the success of the project that the delivery process is relevant for the current project, its size, and the formality of its requirements.

Because the Rational Unified Process provides guidance on a wide range of software engineering principles, you typically need to understand which parts of the process framework can be fully adopted and which parts can be modified or even excluded. Tailoring the process is just one part of implementing a process for a project. After the process has been tailored, the project manager instantiates and executes it for the specific project. An *instantiated* process is an enactable project plan (it includes actual iterations, activities, tasks, and work products for an actual project). Instantiation is done as part of project planning.

We recommend tailoring the Rational Unified Process using Rational Method Composer (RMC). By using RMC, the resulting process Web site has the exact same functionality, look, and feel as the classic RUP Web site. Also, if RMC is used, a Delivery Process can be instantiated by exporting it from RMC and then importing it into a project management tool, such as *Rational Portfolio Manager,* where actual work products can be identified, actual resources can be assigned to roles, and so forth. Before starting a plug-in project, we highly recommend that you spend time looking at existing plug-ins on both the RMC and IBM sponsored RUP Web sites, because you might find new already available methods and processes that fit your project:

RMC:

http://www.ibm.com/developerworks/rational/downloads/06/rmc_plugin7_1/

RUP:

http://www-128.ibm.com/developerworks/rational/library/4686.html

In this chapter, we will examine two tailoring scenarios:

- Create a project plan specific to your project outside of RMC
- Customize the RUP for System z within RMC

Tip: Refer to Rational Unified Process Concept: Tailoring RUP for a detailed description of a variety of tailoring scenarios.

Note: The recommended method development process and the directions for using RMC to customize the RUP for System z are discussed in the next section.

11.2 How to create a project plan specific to your project

This section covers the creation of a project plan specific to your project by identifying the phase iterations, activities, and tasks to execute and then creating a project plan accordingly.

11.2.1 Identify the phase iterations, activities, and tasks to execute

In order to fully understand what you need to include in your Project Plan (or Iteration Plan), we recommend that you focus on which phase iterations, activities, and tasks to execute. More specifically, in order to create a development plan that enables your team to produce the appropriate work products for your project, you need to identify the following:

- How many iterations per phase are necessary on your project?
- What are the activities that need to be performed in each iteration?
- What are the tasks that need to be performed in each activity?

Tip: Refer to Rational Unified Process RUP Lifecycle page for information about how to identify how many iterations per phase are necessary for your project.

For example, the RUP for System z Inception Iteration has five activities as shown in Figure 11-2 on page 210. You have to decide if you need to perform them all. For instance, it might not be necessary to perform the Perform Architectural Proof-of-Concept activity on your project.

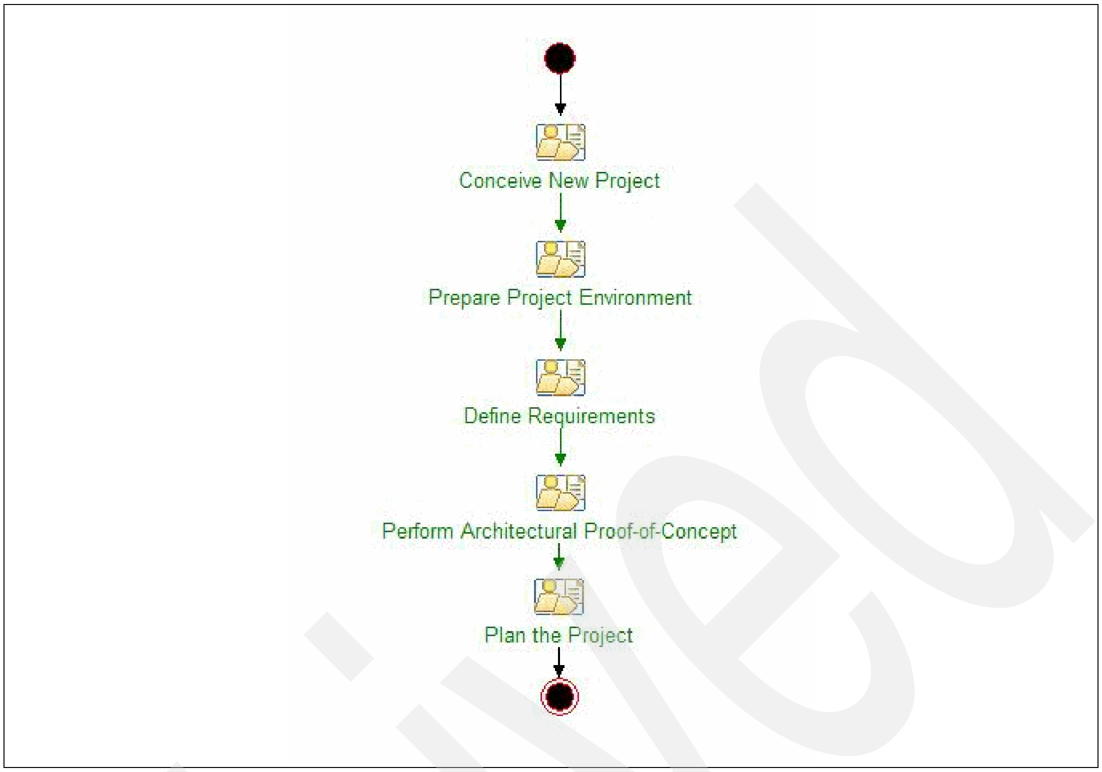


Figure 11-2 The five Inception Iteration activities of RUP for System z

By acknowledging that this is the start and end of the iteration, and not the project, we can encourage more parallel work and study the dependencies between tasks. Therefore, these activity diagrams drive iterative project planning. For example, if you look at Figure 11-3, which depicts the tasks of Define Requirements activity, you can see how most of them can be executed in parallel. Obviously, this parallelism will reflect on your project planning.

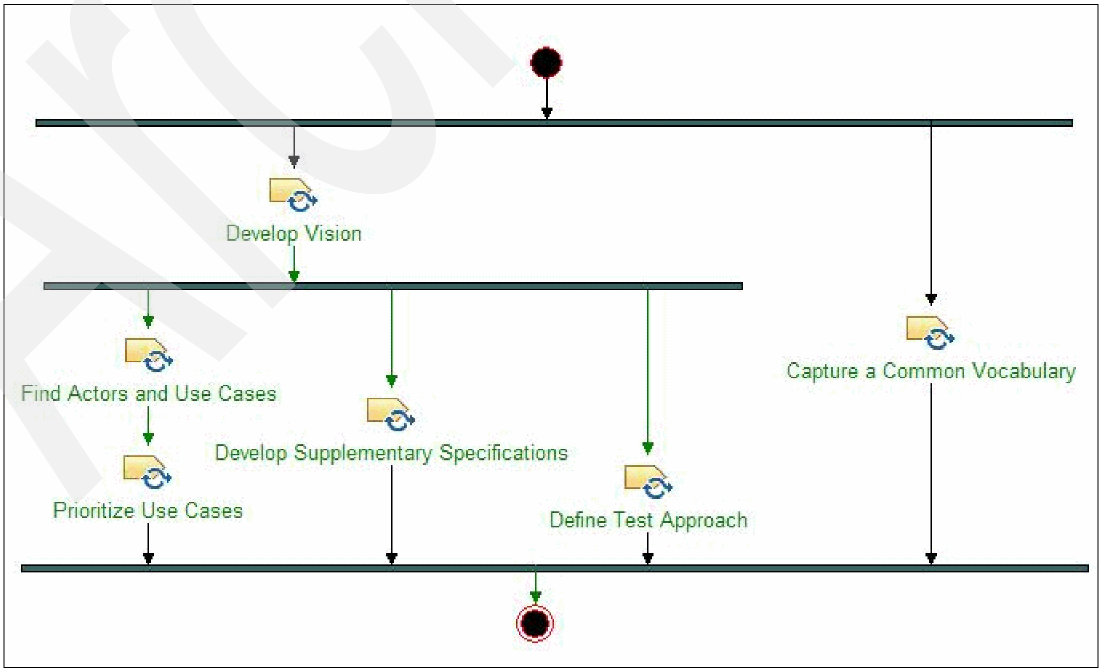


Figure 11-3 The detail of the Define Requirements activity

Let us look at the detail of the Define Requirements activity for an example. Because the Inception Phase involves establishing the problem space and designing the first version of a usage model of the solution idea but not gathering the detailed requirements, you need to ask yourself which of the tasks shown in Figure 11-3 will you need.

It is not only the project manager's job to decide which activities and tasks to perform in each iteration of the RUP Phases; it is the team's decision. Basically, the project manager takes into account the directions provided by team's members and subsequently plans, tracks, and manages the risks accordingly.

After the team decides what iterations, activities, and tasks need to be performed (by using a Development Case, for instance, or any other artifact), the project manager should create the plan accordingly. The last step after determining the activities and tasks is to include key input or output work products. The work products affect your plan in terms of milestones and the evidence of deliverables.

11.2.2 Creating a project plan

We can now assume that you have a Development Case or a well-defined process in place for your project. It is now time to create your Project (or Iteration) Plan. This can be done by exporting the RUP for System z Delivery Process from RMC, then importing it into a project management tool, such as IBM Rational Portfolio Manager or Microsoft® Project, and finally, modifying it to reflect your own specific process as defined in the previous section.

To export an existing process to a planning tool using RMC is fairly simple. You need to select **File** → **Export**, select **Project Template,** and follow the prompts for the planning tool that you use, Rational Portfolio Manager or MS Project. For step-by-step guidance to export a process using RMC, see the "Publishing and exporting Method Content" online help topic. After you export the WBS to an xml file, the next step is opening the exported file with the planning tool that you are going to use.

Using a planning tool, such as IBM Rational Portfolio Manager or Microsoft Project, you see it is simple to tailor your plan according to your own process defined in the previous section. The Gantt charts in Figure 11-4 on page 212 show an example of the RUP for System z delivery process exported into a project plan and modified to suit the needs of a particular project.

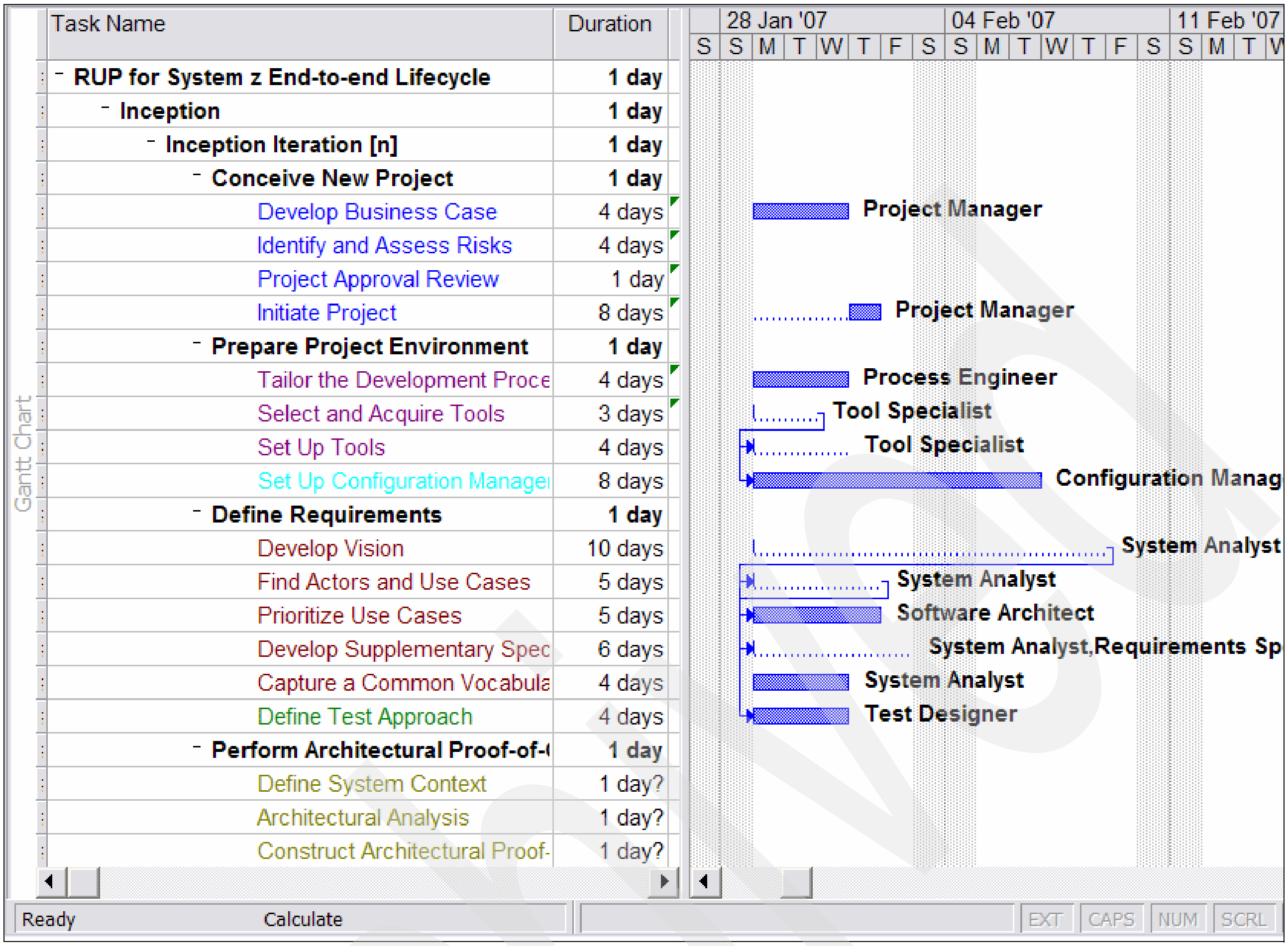


Figure 11-4 Example of Project Plan for RUP for System z

The plan in Figure 11-4 is clearly organized by time, but you can also identify disciplines by color (as coded in the overall architecture diagram of RUP in Figure 2-1 on page 12). For example, the first four tasks are in the project manager discipline, the Set Up Tools is in the environment discipline, the Develop Vision is in the requirements discipline, and finally, the Define Test Approach task is in the test discipline. In this way, all the team members can clearly identify their own tasks according the identification of the discipline, and the project manager can look for a clear single point of view of dependent tasks.

11.3 How to customize the RUP for System z using RMC

This section presents an approach to method development with IBM Rational Method Composer (RMC) that you can use to customize RUP for System z for a specific project team or organization:

http://www.ibm.com/developerworks/rational/products/rup/

This section presents and defines the essential work products produced during a method development project, as well as the typical tasks performed to produce these work products.

You can read more information about method development in the Rational Edge article "A Roadmap to Method Development" by Cécile Péraire, February 2007, at:

http://www.ibm.com/developerworks/rational/library/feb07/peraire/index.html

Specific detailed information about how to perform these tasks using RMC can be obtained in the IBM training material: *PRJ350, Essentials of IBM Rational Method Composer v7.1:*

http://www-304.ibm.com/jct03001c/services/learning/ites.wss/us/en?pageType=course_description&courseCode=RP522

11.3.1 Method development work products

A method is primarily defined in terms of Method Elements. A *Method Element* can be a *Content Element (Role, Task, Work Product,* or *Content Guidance)*, or a *Process Element (Activity, Capability Pattern, Delivery Process*, or *Process Guidance*). Refer to Table 11-1 for a definition of the types of Method Elements.

Table 11-1 The different types of Method Elements and their definition

Method Element	
Content Element	**Process Element**
► **Role**. Set of related skills, competencies, and responsibilities of an individual or individuals. ► **Task**. Assignable unit of work. Every task is assigned to a specific role. ► **Work Product**. Anything used, produced, or modified by a task. ► **Content Guidance**. Supplemental information added to a content element (such as a template, example, tool mentor, or white paper).	► **Activity**. Grouping of tasks (together with their related roles and work products). Represents the key building blocks for processes. Can be used to compose capability patterns or delivery processes. ► **Capability Pattern**. A reusable cluster of activities. Can also be used to compose delivery processes and other capability patterns. ► **Delivery Process**. End-to-end project lifecycle. ► **Process Guidance**. Supplemental information added to a process element (such as a roadmap).

A Method Element can be seen from two different and important perspectives: From the perspective of the Method Designer, who defines the method structure and hence identifies Method Elements and their relationships, or from the perspective of the Method Author, who writes the description (that is, textual and graphical content) of the Method Elements. This second perspective is particularly significant, because a large proportion of the method development effort is spent authoring Method Elements. For this reason, a separate work product called Method Element Description is used to refer to the content of a Method Element (even though the Method Element Description work product is included into the Method Element work product).

The other essential work products specific to method development are summarized in Table 11-2 on page 214 (together with the role responsible for each work product) and further presented in the paragraphs that follow.

Table 11-2 Essential work products and their responsible role

<table>
<tr><th>Work product</th><th>Role</th></tr>
<tr><td>Method sketch
Outline of the method, identifying candidate method elements, and possibly including some of their relationships and early description.</td><td rowspan="3">Method designer
Oversees the definition of the overall method.

Synonyms:
Method Architect,
Method Engineer, or
Process Engineer.</td></tr>
<tr><td>Method definition
Well-formed definition of a method in terms of its Method Elements (including their descriptions), their relationships, and characterization of one or more Method Configurations.
In RMC, a Method Definition is a composite work product encompassing all Method Plug-ins and Method Configurations relevant to the method under construction.
Method plug-in
Well-formed definition of a component of a method (or of the entire method if the method is defined using only one component) in terms of its Method Elements (including their description) and their relationships.
In RMC, a Method Plug-in is a container for Method Packages:
► Method Package
In RMC, a Method Package is a container for Method Elements (and other Method Packages).
► Method Element
Content Element (Role, Task, Work Product, or Content Guidance) or Process Element (Activity, Capability Pattern, Delivery Process, or Process Guidance). Refer to Table 11-1 for definitions.

Method configuration
Characterization of a method configuration or version of the method.
In RMC, a Method Configuration defines how to compose a Method Web site based on the Method Elements included in a selection of Method Plug-ins and Method Packages, as well as the views that will be presented in the Method Web site tree browser.</td></tr>
<tr><td>Method Web site
Main outcome of a method development project. It makes the method, or method framework, available through a set of interconnected Web pages.
In RMC, a Method Web site is automatically generated from a Method Definition (one Method Web site can be generated per Method Configuration).</td></tr>
<tr><td>Method element description
Description (that is, textual and graphical content) of a Content Element.</td><td>Method author
Writes the content of a method element.</td></tr>
</table>

Early in the project, we recommend that you outline the method using a *Method Sketch.* The goal of the Method Sketch is to help the team identify candidate Method Elements and some of their relationships and to propose an early description for some of the key elements. To do so, and depending on the project type (creation of a new method from scratch, extension of an existing method with content elements only, extension of an existing method with content and process elements, and so forth), the Method Sketch might take various forms. For instance, it might include one or more of the following elements: A walkthrough of the lifecycle; a brief description of candidate roles, tasks, and work products, and their relationships; a list of candidate method guidance (templates, examples, white papers, and so forth); an early Work Breakdown Structure (WBS); a mock-up of the method Web site. The Method Sketch can be documented on a white board, in a word document, a spreadsheet, a visual model, or using any other support. This free framework is meant to encourage creative thinking and allow the team to define and review several first sketches of the method using the content, format, notation, and support that best fit their needs and skills. After the new method, or part of the method, starts to emerge from the Method Sketch, it is time to launch Rational Method Composer (RMC), where the method is more formally and completely defined. The Method Sketch can be abandoned at this point, kept to explore other ideas, evolve into a method roadmap (process guidance), or simply evolve into the list of all the method elements to be authored, for instance. In that case, it can be used to assign responsibility and track progress.

The method is well-defined within a *Method Definition*. A Method Definition is a well-formed definition of a method in terms of its Method Elements (including their description) and their relationships. A Method Definition also characterizes one or more configurations, or versions, of the method by identifying, for each configuration, which elements are presented to the practitioners and how. In RMC, as illustrated in Figure 11-5 on page 216, a Method Definition is a composite work product encompassing all Method Plug-ins and Method Configurations relevant to the method under construction. In other words, a Method Definition corresponds to the subset of the RMC Library relevant to the method under construction.

A *Method Configuration* is a characterization of a version of the method, such as RUP for Small Projects and RUP for Large Projects. In RMC, a Method Configuration defines how to compose a Method Web site based on the Method Elements included in a selection of Method Plug-ins and Method Packages (because you might not want to publish all the Method Elements defined in the method in the context of a specific configuration). The Method Configuration also defines the views that will be presented in the Method Web site tree browser.

The *Method Web site* is the main outcome of a method development project. It makes the newly defined method, or method framework, available to the practitioners through a set of interconnected Web pages. In RMC, a Method Web site is automatically generated from the Method Definition (one Method Web site can be generated per Method Configuration). Figure 11-6 on page 217 provides an example of the Method Web site.

To summarize the role of the different constituents of a Method Definition, we can use the analogy of a bookstore specialized in selling book sets at a discounted price. A Method Plug-in can be compared to a bookstore department (travel, comics, children, and so forth). A Method Package can be compared to a set of books packaged to be sold together, the Method Element to an individual book, the Method Configuration to your shopping list, and the Method Web Site to the shopping bag full of books that you take home after shopping.

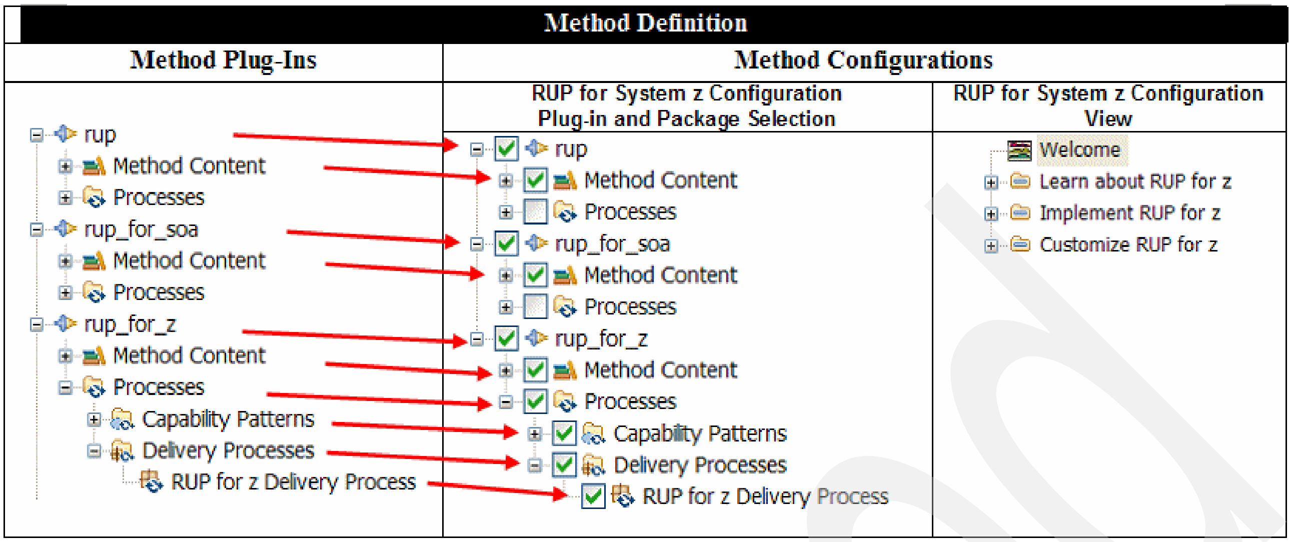


Figure 11-5 Example of Method Definition for the RUP for System z in RMC and selection of elements to be published in the RUP for System z Web site

Figure 11-5 provides a simplified example of RMC Method Definition for the RUP for System z, including three Method Plug-ins (rup, rup_for_soa, and rup_for_z, because the rup_for_z is defined by extending rup and rup_for_soa, plus other plug-ins not shown in Figure 11-5) and one Method Configuration (RUP for System z Configuration). The RUP for System z Configuration determines the content of the RUP for System z published Web site. In this simplified example, the Web site will include the content elements of the Method Content package of rup (the Processes package of rup has been filtered out), the content elements of the Method Content package of rup_for_soa (the Processes package of rup_for_soa has been filtered out), as well as all the method elements (from both Method Content and Processes packages) of rup_for_z. The practitioners will be able to navigate the Web site through the navigation tree shown on the View column of Figure 11-5.

Figure 11-6 on page 217 provides an example of the Method Web site for the RUP for System z (early version), generated out of the Method Definition presented on Figure 11-5.

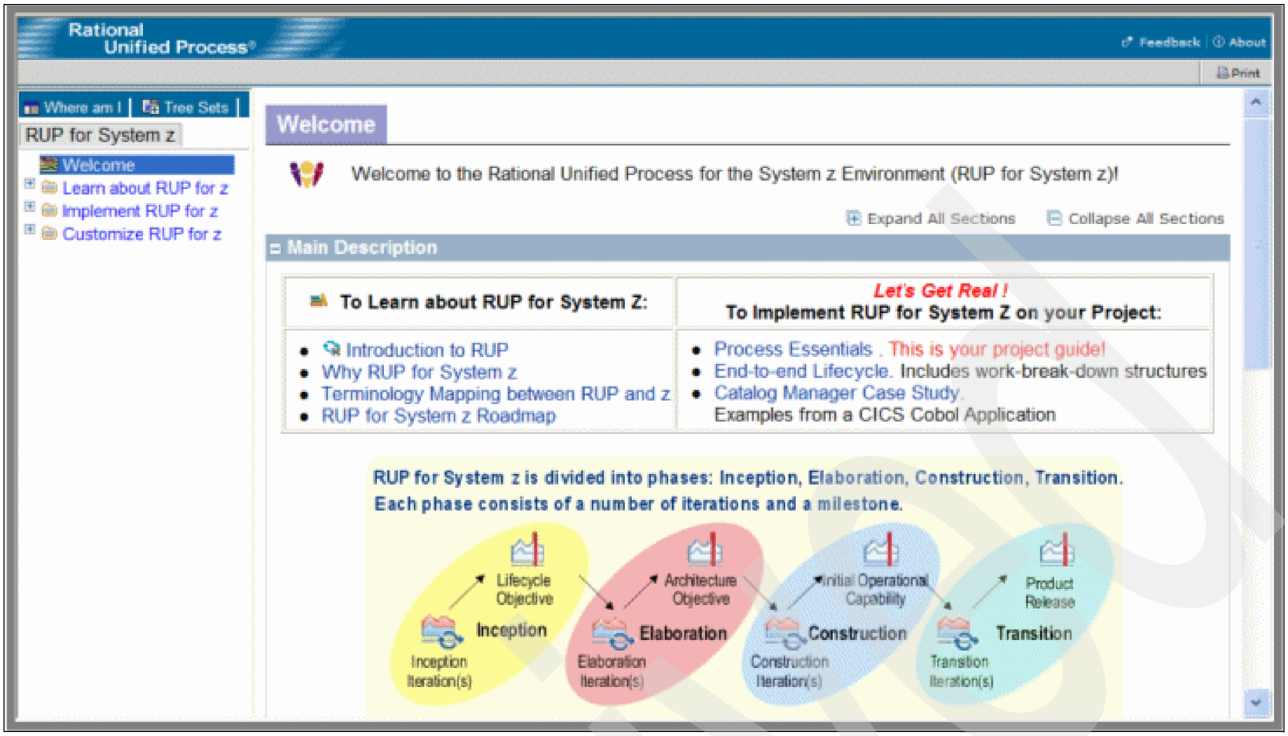


Figure 11-6 Example of Method Web site for the RUP for System z

In the case of a new method built from scratch, the Method Definition is empty at the beginning of the project. In the case of a method customization, the Method Definition already contains the plug-ins relevant to the existing method (not the existing configurations, however, because a configuration is specific to a particular method and consequently is not a reusable asset). For instance, in order to create a customized version of the RUP for System z, you start with a Method definition, including the Plug-ins shown in Figure 11-5 on page 216.

> **Note:** The complete list of plug-ins included in the RUP for System z Method Definition is: rup, base_concepts, formal_resources, informal_resources, rup_soa_plugin, rup_legacy_evolution_plugin, rup_ibm, and rup_for_z.

Then, you add at least one new Method Plug-in (referencing the existing plug-ins) in order to define the Method Elements specific to your new method and add one new Method Configuration in order to configure your new Method Web site. Your new Method Elements can be built from scratch or by leveraging elements of the existing method using variability relationships, such as contribute, extend, or replace. For more information about variability relationships, refer to the IBM training: *PRJ350, Essentials of IBM Rational Method Composer v7.1.*

11.3.2 Method development tasks

The work products specific to method development and presented in the previous section are produced by performing the tasks shown in Figure 11-7 on page 218. Note that other tasks and work products might be applicable in order to test the Web site and distribute the method, for instance, but this is the minimum recommended set of work products. The execution of the tasks in Figure 11-3 needs to be driven by a clear project vision and project plan (whatever their level of formality). The Method Reviewers include primarily peers and other Subject

Matter Experts (SMEs). However, we recommend that you involve other stakeholders in the evaluation of the method, such as practitioners and clients, to make sure that the method is consumable and meets the stakeholders' expectations.

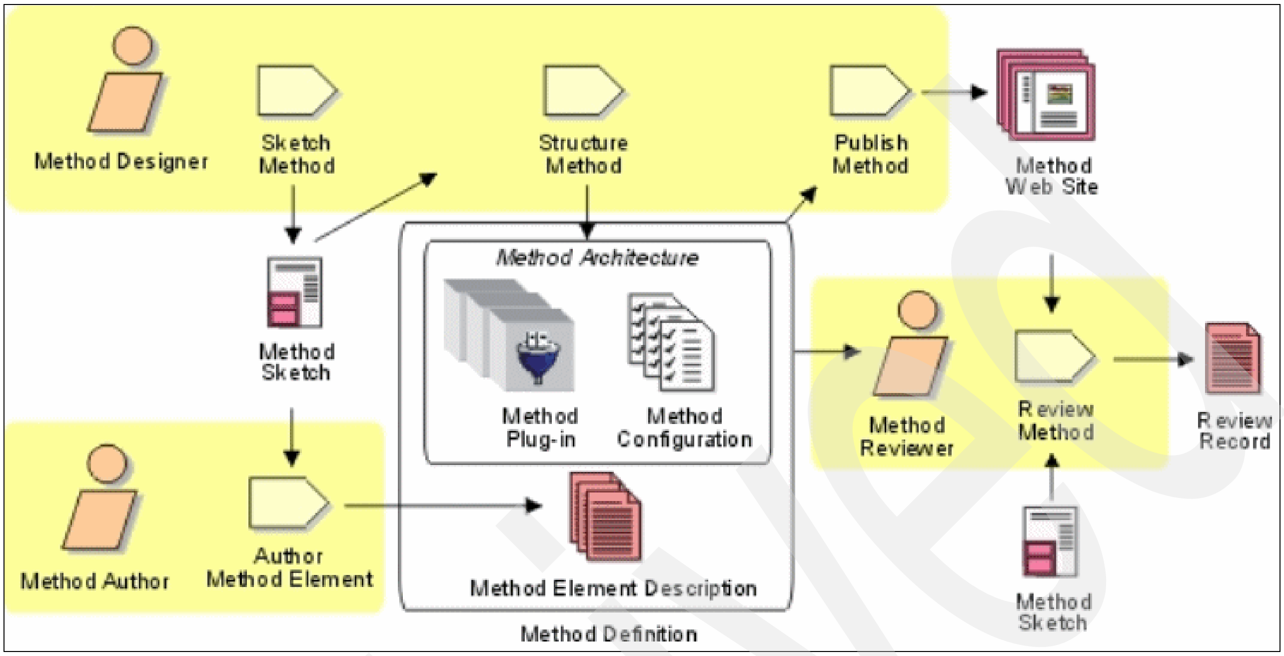


Figure 11-7 Essential tasks together with the role that performs the task and input/output work products

Table 11-3 provides the description of the *Sketch Method* task. The task is independent from the actual strategy adopted by the project team to define the method, such as top-down (the definition of the overall lifecycle in terms of phases and their objectives drives the identification of the content elements) or bottom-up (the identification of the content elements drives the definition of the lifecycle). In practice, project teams generally use a combination of the two strategies. However, whenever applicable, we recommend favoring a top-down strategy, which tends to produce more cohesive methods, because each content element is more clearly tied to one or more phase objectives.

Table 11-3 Description of the Sketch Method task

Task: **Sketch Method**	**This task outlines the method within a Method Sketch. It identifies candidate method elements and some of their relationships and proposes an early description for some of the key elements.** **Some indicative steps to perform this task are proposed below (no specific order):**
	Identify the method development strategy, such as top-down or bottom-up.
	Review existing assets, such as RUP for System z, and identify reuse opportunities.
	Identify candidate method elements (content and processes) that are relevant to the new method. If applicable, distinguish the elements that are in the project scope from the elements outside of the project scope (because they are already present in existing methods and can be reused as-is for instance).
	Identify candidate relationships between candidate method elements (for example, which role is responsible for a work product).
	Draft an early description for some key candidate method elements.

Task: **Sketch Method**	**This task outlines the method within a Method Sketch. It identifies candidate method elements and some of their relationships and proposes an early description for some of the key elements.** **Some indicative steps to perform this task are proposed below (no specific order):**
	Get feedback on the Method Sketch from stakeholders.

Table 11-4 provides the description of the Structure Method Task.

Table 11-4 Description of the Structure Method Task

Task: **Structure Method**	**This task structures the method, in terms of Method plug-ins, Method packages, Method elements, Method configurations, and the relationships among them.** **Some indicative steps to perform this task are proposed below (no specific order):**
	If applicable, review the structure of the existing Method plug-ins forming the foundation of the new method, such as the rup and rup_for_z plug-ins.
	Define Method plug-ins relevant to the new method and their dependencies to other plug-ins if applicable.
	Define Method packages within the appropriate plug-ins.
	Define Method elements within the appropriate packages and their relationships, including variability relationships to leverage existing assets.
	Logically categorize the Method elements using RMC Standard or Custom Categories.
	Define navigation views using RMC Custom Categories.
	Define Method configurations by selecting the set of Method plug-ins and Method packages to be published, as well as the navigation views to be published.
	Get feedback on the Method Structure from stakeholders.

12 Conclusions

When we first set out on the Rational Unified Process (RUP) for System z project, there were several primary questions on our minds:

- Is RUP really applicable to the System z environment?
- Are there any parts of RUP that do not apply to the System z environment?
- How is iterative development implemented in the System z environment?
- Are there real benefits for System z practitioners?

We were determined to discover the answers to these questions. The result of our quest for answers has been documented in this book.

The materials in this book have predominantly been derived from actually performing RUP prescribed functions. Applying RUP practices to the development of a real System z CICS COBOL application gave us the insight we sought.

In Chapter 1, we introduced the RUP for System z IBM Redbooks publication project: its purpose, intended audience, and the rationale behind it.

In Chapter 2, we provided a brief introduction to the Rational Unified Process and to its extension to service-oriented architecture (SOA). What it is and what it contains.

In Chapter 3, we expanded on the question, “Why RUP for z?” We explored the traditions behind System z application development and the differences between the old and new development methodologies. We summarized the key differences and associated benefits of the RUP compared to the waterfall model-based methodologies.

In Chapter 4, we provided a RUP for System z roadmap. The roadmap walks through each phase (inception, elaboration, construction, and transition) of a typical System z development project. The roadmap also describes the overriding goal of each of the four phases and its associated objectives. It describes typical iterations within each phase and provides information about phase milestones and essential work products.

In Chapter 5, we provided RUP for System z process essentials: A brief definition of each project phase (inception, elaboration, construction, and transition) in terms of main goals, activities, and milestones. For each activity, the process essentials list the corresponding key roles, tasks, output work products, and available examples from the Catalog Manager case study. The corresponding section of the RUP for System z Web site provides advanced System z practitioners with all the links necessary to perform specific activities or tasks.

In Chapter 6, we introduced an end-to-end System z development lifecycle, which is essentially a System z application delivery process example. It contains detailed descriptions of all the elements of the RUP for System z method and also includes a depiction of the lifecycle in the form of a Work Breakdown Structure (WBS).

In Chapter 7, we presented the RUP for System z content elements (roles, tasks, and artifacts) that are specific to the System z environment. The RUP for System z includes a large number of content elements. Most of these elements come from the Rational Unified Process (RUP) and its service-oriented architecture (SOA) extension. However, we had to add some content elements to the RUP for System z, because these content elements are specific to the System z environment. In this chapter, we presented these new content elements.

In Chapter 8, we provided the details of our case study: the Catalog Manager application. It introduces the CICS COBOL Catalog Manager application and provides details about the RUP practices, activities, and tasks with which we were involved during the development of the application.

In Chapter 9, we provided an overview of the Enterprise Generation Language (EGL) and its Web application development paradigm. Chapter 9 provides information about how EGL was used to develop a Web client application in order to provide a Web interface into the CICS COBOL Catalog Manager application. The Web client accesses the Web services exposed from within the CICS COBOL Catalog Manager application.

In Chapter 10, we provided the RUP for System z Work Breakdown Structure (WBS) that covers the whole development lifecycle from beginning to end. This WBS can be used as a template for planning and running a project. In this chapter, we presented the WBS for each project phase (inception, elaboration, construction, and transition).

And finally in Chapter 11, we provided a practical approach to customize RUP for System z to suit your own organization's needs if required. It illustrates the flexibility of the RUP for System z method, acknowledging the fact that your own environment might differ in some parts to the environment used in our project and therefore the importance of having the ability to customize the method to your own needs and preferences in order to derive maximum benefit.

Equally important are all the elements gathered in the Appendixes of this book. They are intended to provide you with key actual work products that were generated at various phases of our project, so that you can see and appreciate the incremental nature and benefits of the RUP for System z method.

In summary, our main conclusions are:

- By developing iteratively, we were quickly able to build and demonstrate an application architecture baseline. Because the architecture baseline was built while taking into consideration the riskiest and high priority parts of the project, we were able to demonstrate a stable application foundation framework very early to our stakeholders and to ourselves. We demonstrated an environment that we can easily improve and expand upon in later iterations. This stable framework ensures that we are building the right system.
- Furthermore with each subsequent iteration, we were able to constantly ensure and verify by actual testing and implementation of the application portions we had built so far that the application was aligned with existing or changed user needs.

- Beginning to verify work products and test application components right from the very early iterations enabled us to identify and fix defects early rather than later. As we all know, identifying and fixing defects earlier is far less costly than fixing defects later in the lifecycle. We wound up building a more robust and stable application as iterations progressed.
- The end of the iteration project reviews helped show the development team and our project sponsors precisely what we achieved and what remained to be achieved for each iteration and phase. The project planning activity of RUP helps define clear goals and assessment criteria for each iteration within a phase so that assessment of what has been achieved and what remains to be done is an easy process. Project plans were continuously reviewed and refined based on risk and priority assessments.

In exploring RUP, we found that it was certainly highly applicable to application development in the System z environment. We hope this book and its associated elements, for example, the RMC method plug-in and the example work products, assist you in utilizing RUP for System z to your organization's best advantage.

Part 5

Appendixes

The appendixes that follow present work products of the Catalog Manager case study. This case study uses a CICS Catalog Manager application to provide an implementation example of the IBM Rational Unified Process for System z (RUP for System z).

Also, the appendixes include a terminology mapping between RUP and z, which maps RUP terminology to equivalent terms used in the System z software development process, to mitigate the learning process when transitioning to RUP.

Finally, the appendixes refer to additional material that you can download from the Internet.

A

Catalog Manager case study: Inception Phase Work Products

The contents of Appendix A are contained in the Additional Material link from the IBM Redbooks publications Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG247362

and choose:

SG24-7362-00-AppendixA.zip

After you have downloaded AppendixA.zip and unpacked the file, the contents of Appendix A are:

- Catalog Manager - Business Case I1 v 1.0.doc
- Catalog Manager - Vision.doc
- Catalog Manager - Glossary.doc
- Catalog_Manager_Software_Development_Plan.doc
- Catalog_Manager_Risk_List.doc
- Catalog Manager UseCaseModelSurveyRpt.pdf
- Catalog_Manager_Supplementary_Specification_V1.1.doc
- Catalog Manager Software Architecture Document V1.1.doc
- Catalog_Manager_Test_Plan_V1.0.doc
- Catalog_Manager_Iteration_Plan_E1.doc

B

Catalog Manager case study: Elaboration Phase Work Products

The contents of Appendix B are contained in the Additional Material link from the IBM Redbooks publication Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG247362

and choose: SG24-7362-00-AppendixB.zip

After you download AppendixB.zip and unpack the file, the contents of Appendix B are:

- Catalog_Manager_Software_Development_Plan.doc
- Elaboration Catalog_Manager_Risk_List.doc
- Catalog Manager UseCaseModelSurveyRpt.pdf
- List Catalog Item.UCS
- Configure Catalog.UCS
- Replenish Inventory.UCS
- Order Item.UCS
- Supplementary Specification.SUP
- Catalog Manager Software Architecture Document V1.2.doc
- Catalog_Manager_Test_Plan_V1.1.doc
- TestScenarioMatrixListCatalog.xls
- TestScenarioMatrixConfigureCatalog.xls
- TestCaseMatrixConfigureCatalog.xls
- TestCaseMatrixListCatalog.xls
- TestEvaluationSummary.doc

C

Catalog Manager case study: Construction Phase Work Products

The contents of Appendix C are contained in the Additional Material link from the IBM Redbooks publication Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG247362

and choose:

SG24-7362-00-AppendixC.zip

After you have downloaded AppendixC.zip and unpacked the file, the contents of Appendix C are:

- Construction Catalog_Manager_Software_Development_Plan.doc
- Construction Catalog_Manager_Risk_List.doc
- Order Item.UCS
- Replenish Inventory.UCS
- Catalog_Manager_Test_Plan_V1.2.doc
- TestScenarioMatrixOrderItem.xls
- TestScenarioMatrixReplenishInventory.xls
- TestCaseMatrixOrderItem.xls
- TestCaseMatrixReplenishInventory.xls
- TestEvaluationSummary.doc

D

Catalog Manager case study: Transition Phase Work Products

The contents of Appendix D are contained in the Additional Material link from the IBM Redbooks publication Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG247362

and choose:

SG24-7362-00-AppendixD.zip

After you have downloaded AppendixD.zip and unpacked the file, the contents of Appendix D are:

- Transition Catalog_Manager_Risk_List.doc
- TestEvaluationSummary.doc

E

Terminology mapping between IBM RUP and System z

This Appendix maps RUP terminology and concepts used throughout this book to equivalent terms used in the System z software development process to mitigate the learning process when transitioning to RUP.

Table E-1 maps RUP terms to equivalent System z development terms.

Table E-1 RUP and System z terminology comparison table

RUP terminology	System z terminology
Analysis and Design Model Analysis provides a rough sketch or generalization of the solution, omitting most of the detail. Design provides the details. The Design Model is the major blueprint for the implementation of the solution. It captures the results of analysis and design into a single model.	**High Level Design** This is the second stage in the Software Development Lifecycle (SDLC). The stages are Requirements gathering, Design, Coding, and Testing. Design is further broken down into two stages: 1) High Level Design and 2) Low Level Design, which is also called Detailed Design. High Level Design is the process of designing an overall solution architecture for an application to meet both functional and non-functional requirements. It involves the following activities: 1. Design of the Solution Architecture for the project 2. Design of the technical Architecture for the project 3. Development of the Logical model for the project 4. Development of the Physical Data Model for the project

RUP terminology	System z terminology
Data Model Describes the logical and physical representations of persistent data used by the application.	**High Level Design** See definition above.
Design Elements Are part of the design model, such as design classes, interfaces, and design subsystems, that evolve from the analysis classes.	**Low Level Design** Also called Detailed design, is the process of refining and expanding the high level design of a system or component to the extent that the design is sufficiently complete to be implemented. Low level design can also involve the writing of pseudo-code.
Design Subsystem The design subsystem encapsulates behavior by packaging other model elements (classes or other design subsystems) that provide its behavior. It also exposes a set of interfaces, which defines the behavior it can perform.	**Low Level Design** See definition above.
Implementation Model Represents the physical composition of the implementation in terms of Implementation Subsystems and Implementation Elements (directories and files, including source code, data, and executable files).	**Low Level Design** See definition above.
Installation Artifacts Describes how someone should install a solution. Installation Artifacts refer to the software and documented instructions required to install the release.	**Program Directory** A document shipped with each release of a product. It contains information concerning the materials and procedures associated with the installation of the product.
Iteration Assessment Captures the results of an iteration, including the degree to which the iteration's objectives were met, lessons learned, and recommended changes.	**Project Milestones Review** A project is divided into milestones with each milestone representing important parts or steps to perform and culminating in a deliverable. Milestones are reviewed to ensure that the project is on time and on track to meeting its goals and deadlines.
Iteration Plan (one per iteration) A time-sequenced set of activities and tasks, with assigned resources, containing task dependencies for the iteration; a fine-grained plan.	**Project Schedule** A time-sequenced set of activities and tasks, with assigned resources, containing task dependencies and sequenced in a logical order. The most common representation of a project schedule is a Gantt Chart.

RUP terminology	System z terminology
Product Actual product or solution to be delivered to the client. It includes: **Product Artwork, Installation Artifacts, Deployment Unit, and Bill of Materials.**	**Product** The Product is the actual product or solution to be delivered to the customer. It includes: Installation Artifacts such as Program Directory, User Manuals, Program (A program consists of elements such as modules, macros, and other types of data). For z/OS systems, a product called System Modification Program Extended (SMP/E) is normally used to install a product, install changes (service, user modifications, or new functions) to the product and track the current status of each of the elements of the product.
Release Notes Describes a release of a solution. Release Notes identify changes and known bugs in a version of a build or deployment unit that has been made available for either internal or external use.	**Release Notes and Announcement Letter** Describes a release of a solution. Highlights new features and enhancements. Identifies known bugs and troubleshooting tips in a version of a build or deployment unit that has been made available for either internal or external use.
Run-Time Architecture Process architecture for the system in terms of active classes and their instances and the relationship of these to operating system threads and processes.	**Program Modules and their relationships** A set of rules that defines how software operates and how to interact with the software. It dictates how code and data are addressed, the form of generated code, how applications are handled, and how to enable system calls. There are several examples of run-time architectures in system z, such as CICS, IMS™, Unix System Services, Linux/z, WAS/z, HTTP Server, and so forth.
Service Model An abstraction of the IT services implemented within an enterprise and supporting the development of one or more service-oriented solutions. It is used to conceive and document the design of the software services. It is a comprehensive, composite work product encompassing all services, providers, specifications, partitions, messages, collaborations, and the relationships between them. It is needed to: - Identify candidate services and capture decisions about which services will actually be exposed - Specify the contract between the service provider and the consumer of the services (Service Specification) - Associate Services with the components needed to realize these services	**Service Model** See RUP definition.
Service Component Provide the implementation of the services identified within the Service Model.	**Service Component** See RUP definition.

RUP terminology	System z terminology
Software Architecture Document A comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system: ▸ **Use-Case View** Captures architecturally significant subset of the requirements, including use cases. ▸ **Logical View** Basis for understanding the structure and organization of the design of the system. ▸ **Implementation View** Captures the architectural decisions made for the implementation. Typically, the implementation view contains: an enumeration of all subsystems in the implementation model, component diagrams illustrating how subsystems are organized in layers and hierarchies, and illustrations of import dependencies between subsystems. ▸ **Deployment View** Captures the physical distribution of the system across a set of processing nodes. ▸ **Process View** Illustrates the distribution of processing across a set of nodes in the system, including the physical distribution of processes and threads.	**High Level Design** See definition above.
Software Development Plan A comprehensive, composite artifact that gathers all information required to manage the project. It encloses a number of artifacts developed during the Inception Phase and is maintained throughout the project.	**Project Plan** The Project Plan is a comprehensive artifact that gathers all information required to manage the project.
Task: Implement Test This task covers the development of tests that can be executed.	**Task: Code Test** This task covers the development of tests that can be executed.
Task: Execute Test This task covers the execution of tests.	**Task: Implement Test** This task covers the execution of tests.
▸ **Test Case** A *Test Case* is a set of test inputs, execution conditions, and expected results, identified for the purpose of making an evaluation of some particular aspect of a target test item. ▸ **Test Script** A Test Script is a step-by-step instruction that realizes a test, enabling its execution.	**Test Case** Test Cases in System z refer to both RUP Test Cases and RUP Test Scripts.

RUP terminology	System z terminology
Test Suite A collection of related Test Scripts. Test Scripts can be grouped together into Test Suites to perform different types of activities, such as unit test, integration test, system test, or acceptance test.	**Test Suite** Collection of related test cases. Test cases can be grouped together to perform different types of activities, such as unit test, integration test, system test, or acceptance test, for instance.
Requirements Requirements are captured in the Use-Case Model and Supplementary Specifications. The Use-Case Model is a model of the system's intended functions and its environment. It serves as a contract between the client and the developers. It is used as an essential input to activities in analysis, design, and test. The Supplementary Specifications capture system requirements that are not readily captured in behavioral requirements artifacts, such as use-case specifications.	Requirements Section in **High Level Design Document** or in the **Software Requirements Specifications (SRS) Document**. An SRS is a document that captures functional and nonfunctional requirements. It has a Requirements Traceability Matrix (RTM) that defines business rules for contingencies and responsibilities. The RTM traces requirements to their sources, such as a specific need, a use case, industry standard, or government regulation.
Vision Defines the stakeholders' view of the product to be developed, specified in terms of the stakeholders' key needs and features. It contains an outline of the envisioned core requirements, so it provides a contractual basis for the more detailed technical requirements.	**Statement Of Work (SOW)** Describes the stakeholders' view of the product to be developed. It outlines generally at a high level all work required to complete the project, the scope of work, deliverables, terms, and conditions. it provides a contractual basis for the more detailed technical requirements.

F

Additional material

This book refers to additional material that you can download from the Internet as described below.

Locating the Web material

The Web material associated with this IBM Redbooks publication is available in softcopy on the Internet from the IBM Redbooks publications Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247362`

and choose:

SG24-7362-00-CobolCode.zip
SG24-7362-00-EGLCode.zip
SG24-7362-00-RMCPlugin.zip
SG24-7362-00-AppendixA.zip
SG24-7362-00-AppendixB.zip
SG24-7362-00-AppendixC.zip
SG24-7362-00-AppendixD.zip

Alternatively, you can go to the IBM Redbooks publications Web site at:

ibm.com/redbooks

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks publication form number, SG24-7362-00.

You can also download the RUP for System z RMC plug-in from IBM developerWorks at:

`http://www.ibm.com/developerworks/rational/downloads/06/rmc_plugin7_1/`

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name	*Description*
SG24-7362-00-CobolCode.zip	Zipped COBOL code sample for the Catalog Manager application (Replenish Inventory use case).
SG24-7362-00-EGLCode.zip	Zipped EGL code sample for the Web interface of the Catalog Manager application.
SG24-7362-00-RMCPlugin.zip	Zipped RUP for System z RMC plug-in, including the entire RUP for System z method. The RUP for System z Web site can be generated out of RMC.
SG24-7362-00-AppendixA.zip	Zipped Catalog Manager Case Study Inception Phase Work Products.
SG24-7362-00-AppendixB.zip	Zipped Catalog Manager Case Study Elaboration Phase Work Products.
SG24-7362-00-AppendixC.zip	Zipped Catalog Manager Case Study Construction Phase Work Products.
SG24-7362-00-AppendixD.zip	Zipped Catalog Manager Case Study Transition Phase Work Products.

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 1 GB
Operating System: Windows® Professional 2000 or XP Professional
Processor: Pentium® 4 or higher
Memory: 2 GB or more

How to use the Web material

Create a subdirectory (folder) on your workstation and unzip the contents of the Web material zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this IBM Redbooks publication.

IBM Redbooks publications

For information about ordering these publications, see “How to get IBM Redbooks publications” on page 244. Note that some of the documents referenced here might be available in softcopy only:

- *WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects,* SG24-6361
- *Exploring WebSphere Studio Enterprise Developer V5.1.2,* SG24-6483
- *Implementing CICS Web Services*, SG24-7206
- *Application Development for CICS Web Services*, SG24-7126
- *Implementing CICS Web Services,* SG24-7206

Other publications

These publications are also relevant as further information sources:

- Per Kroll and Philippe Kruchten, The Rational Unified Process Made Easy: A Practitioner’s Guide to Rational Unified Process. Addison Wesley 2003.
- Per Kroll and Walker Royce, “Key principles for business-driven development,” The Rational Edge, October 2005:

  http://www-128.ibm.com/developerworks/rational/library/oct05/kroll/index.html
- Per Kroll, “Introducing IBM Rational Method Composer,” The Rational Edge, November 2005:

  http://www-128.ibm.com/developerworks/rational/library/nov05/kroll/index.html
- Peter Haumer, “IBM Rational Method Composer (RMC): Part 1: Key Concepts,” The Rational Edge, December 2005:

  http://www-128.ibm.com/developerworks/rational/library/dec05/haumer/
- Peter Haumer, “IBM Rational Method Composer (RMC): Part 2: Authoring Method Content and Processes,” The Rational Edge, January 2006:

  http://www-128.ibm.com/developerworks/rational/library/jan06/haumer/
- Cécile Péraire, “A Roadmap to Method Development,” The Rational Edge, February 2007:

  http://www.ibm.com/developerworks/rational/library/feb07/peraire/index.html
- Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison Wesley Longman, 1998
- Philippe Kruchten. *The Rational Unified Process*, An Introduction, Second Edition. Addison Wesley Longman. 2000
- Kurt Bittner, March 2006. “Driving Iterative Development With Use Cases”:

  http://www-128.ibm.com/developerworks/rational/library/4029.html

Online resources

These Web sites are also relevant as further information sources:

- IBM Rational Method Composer on the IBM Web site:

  http://www.ibm.com/software/awdtools/rmc/index.html

- IBM Rational Method Composer and RUP on IBM Rational developerWorks:

  http://www.ibm.com/developerworks/rational/products/rup/

- IBM Rational Method Composer (RMC) Special Interest Group (SIG) Home Page:

  http://techworks.dfw.ibm.com/rational/cop.nsf/doc/RCOP-6NHT7R?OpenDocument

- IBM Rational Method Composer (RMC) Special Interest Group (SIG) Resource Guide:

  http://techworks.dfw.ibm.com/rational/cop.nsf/doc/RCOP-6NHT7Z?OpenDocument

- IBM Rational Software Training - Process and Portfolio Management:

  http://www-304.ibm.com/jct03001c/services/learning/ites.wss/us/en?pageType=course_list&subChapter=1129&subChapterInd=S&region=us&subChapterName=Process+and+portfolio+management&country=us

- Creating and consuming Web services with EGL using WebSphere Developer for zSeries:

  http://www-128.ibm.com/developerworks/websphere/library/tutorials/0609_barosa.html/

- V6.0.1 Tutorial Exploring Enterprise Generation Language (EGL) and learn how to write business logic with EGL:

  http://www-128.ibm.com/developerworks/rational/products/egl/egldoc.html

- V6.0.1 EGL and Java Server Faces (JSF) component and JSP page development techniques:

  http://www-128.ibm.com/developerworks/rational/products/egl/egldoc.html

- EGL Web Services: Create and Consume -- A how to tutorial, V6.0.1:

  http://www-128.ibm.com/developerworks/rational/products/egl/egldoc.html

- EGL/JSPF Component tree access and manipulation -- A how to tutorial, V6.0.1:

  http://www-128.ibm.com/developerworks/rational/products/egl/egldoc.html

- RMC V7.1 Plug-Ins on IBM developerWorks:

  http://www-128.ibm.com/developerworks/rational/downloads/06/rmc_plugin7_1/

- RMC Plug-In on RUP for Maintenance Projects on IBM developerWorks:

  http://www.ibm.com/developerworks/rational/downloads/06/plugins/rmc_prj_mnt/

- IBM training: PRJ350, Essentials of IBM Rational Method Composer V7.1:

  http://www-304.ibm.com/jct03001c/services/learning/ites.wss/us/en?pageType=course_description&courseCode=RP522

How to get IBM Redbooks publications

You can search for, view, or download IBM Redbooks publications, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy IBM Redbooks publications or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

Index

D

E

F

H

I

IBM

Redbooks

The IBM Rational Unified Process for System z

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

The IBM Rational Unified Process for System z

RUP for System z includes a succinct end-to-end process for z practitioners

RUP for System z includes many examples of various deliverables

RUP for System z is available as an RMC/RUP plug-in

Faced with the growing need for the System z development community to continue to produce world class quality applications, while at the same time precisely fulfilling user needs and demands, there is a requirement now more than ever before, for System z developers to adopt more modern and agile development principles and methodologies.

The IBM world renown Rational Unified Process (RUP) is a modern software development methodology based on proven development principles and current best practices.

This IBM Redbooks publication focuses on applying the very same key principles and best practices on which RUP is based to application development in the System z environment.

No matter which facet of System z application development you are involved with, this IBM Redbooks publication is an invaluable reference that describes, in System z appropriate language, the RUP for System z method along with elements and workflows specifically created and delivered for the System z environment. In addition, it provides as a case study example, a COBOL CICS application developed using the RUP for System z methodology. The case study provides you an iteration-by-iteration walkthrough, from requirements gathering, right through to analysis, design, code, testing, and deployment of the application.

This IBM Redbooks publication also shows you how to obtain the RUP for System z plug-in for the Rational Method Composer (RMC). The RMC plug-in allows you to publish the method as a Web site for your own organization's use and to further customize the method to your own specific needs and preferences, if necessary.

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

SG24-7362-00 ISBN 073848900X