IBM

Problem Determination Using Self-Managing Autonomic Technology

Add self-managing autonomic capabilities to components

Common Base Events best practices

Practical problem determination scenarios

Edson Manoel
Morten Jul Nielsen
Abdi Salahshour
Sai Sampath K.V.L.
Sanjeev Sudarshanan

Redbooks

ibm.com/redbooks

International Technical Support Organization

Problem Determination Using Self-Managing Autonomic Technology

June 2005

SG24-6665-00

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

First Edition (June 2005)

This edition applies to Version 2, of IBM Autonomic Computing Toolkit, which can be obtained at http://www.ibm.com/developerworks/autonomic.

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 illustrates 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. 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 IBM's application programming interfaces.

Trademarks

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

AIX®
Cloudscape™
DB2 Universal Database™
DB2®
developerWorks®
ibm.com®
IBM®
MQSeries®
Rational®
Redbooks (logo) ™
Redbooks™
Tivoli Enterprise Console®
Tivoli Enterprise™
Tivoli®
WebSphere®

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Preface

This IBM Redbook provides practical information related to problem determination using IBM Self-Managing Autonomic Technology, with an emphasis on the components provided by the IBM Autonomic Computing Toolkit.

The autonomic computing initiative from IBM defines a variety of capabilities that enable information technology systems and solutions to be self-configuring, self-healing, self-optimizing, and self-protecting. The technologies required to enable this capability across a complex, multivendor, distributed environment are still evolving. The process of implementing and deploying systems capable of autonomic behavior is an evolutionary one.

The primary objective of this book is to empower software developers to provide facilities to enable applications or other system components to participate in an autonomic environment and focuses on the problem determination capabilities provided by the IBM Autonomic Computing Toolkit and other Self-Managing Autonomic Technology.

The tools, components, and techniques used and demonstrated in this book provide the base for automating problem determination and self-healing of applications and components using Self-Managing Autonomic Technology. As with the technologies themselves, components of the IBM Autonomic Computing Toolkit will evolve as new capabilities and tools become available. Software developers are encouraged to check the latest releases of both IBM Autonomic Computing Toolkit and other tools provided by IBM at:

http://www.ibm.com/developerworks/autonomic

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Edson Manoel is a software engineer at IBM working in the ITSO, Austin Center, as a Senior IT Specialist in the systems management area. Prior to joining the ITSO, Edson worked in the IBM Software Group, Tivoli® Systems, and in IBM Brazil Global Services Organization. He was involved in numerous projects in designing and implementing systems management solutions for IBM clients and Business Partners. Edson holds a Bachelor of Science degree in applied mathematics from Universidade de Sao Paulo, Brazil.

Morten Jul Nielsen is a Advisory IT Specialist with IBM Global Services in Denmark. He has nine years of experience in the IT field, of which the last four years are within Java™ and J2EE application development. He has worked at IBM for nine years. He holds a Master of Science in Engineering in Informatics from Technical University of Denmark (DTU). His areas of expertise include analysis, design, and development of Java and J2EE application. He has practical experience in problem determination and resolution. He has written extensively about the Autonomic Management Engine and resource models.

Abdi Salahshour is a Senior Software Engineer and problem determination (PD) architect at IBM Corporation, Autonomic Computing (AC) Technology and Enablement, working as AC Technology team leader. He started with IBM in 1982 and served in many key roles in design and development of database diagnostic tools, system management, and self-healing architecture and enablement in heterogeneous and distributed environments, served as member of the IBM PD Council focused on PD artifacts modeling and architecture, one of the authors of the IBM Common Base Event, design and implementation of the Generic Log Adapter, joint work with IBM Research to exploit technologies and data mining in an autonomic environment. Abdi holds several patents in these areas and is co-author of several IBM technical bulletins and papers to conferences, such as IEEE International Conference on Autonomic Computing and IBM Academy of Technology for Proactive Problem Prediction, Avoidance and Diagnosis.

Sai Sampath K.V.L. is a Senior Software Engineer with IBM Software Labs for IBM in Bangalore, India. He has more than eight years of experience in the IT field, of which the last six years working for IBM. Sai works for the IBM JVM development and support team with a particular focus on plug-ins and GC component for Linux®, IBM AIX®, and Microsoft® Windows®. His area of expertise include design and development of solutions for customer requirements using IBM WebSphere® and IBM Tivoli security products.

Sanjeev Sudarshanan is a Staff Software Engineer at IBM Software Labs, Bangalore, India. He has a total of nine years of IT experience, with five years at IBM. Sanjeev is a member of the WebSphere MQ Level 3 support team at IBM, Bangalore. His area of interests are middleware solutions, databases, problem determination, and autonomic computing. He holds an MBA in Systems from MK University, India and he is an IBM Certified MQSeries® Specialist.

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

Brent Miller is a Senior Technical Staff Member in the IBM Tivoli Autonomic Computing Division. He is the lead architect for self-healing. He has led several software development teams in printers, mobile computers, and network computing. His expertise includes adaptivity, printing, service discovery, pervasive computing, and autonomic computing. He has also been involved with and chaired several mobile computing consortia and task forces, including chairing the Service Discovery task force of the Software Working Group for the Bluetooth Special Interest Group (SIG). Brent has published several papers and holds several patents in these areas. He is the principal author of the book, Bluetooth Revealed and a contributing author to the Internet Encyclopedia.

Dave Smith is a Staff Software Developer on the IBM Autonomic Computing (AC) Tools and Technologies project and a committer to the IBM Eclipse Test and Performance Tools Platform (TPTP) open-source project (formally Hyades). He is currently responsible for the development of the Generic Log Adapter technology of TPTP. He has worked on the Profiling Agent and Agent Controller components of TPTP, as well as other profiling and distributed computing technologies during his 18 years with IBM.

Mike Wamboldt is a Senior Software Engineer in the IBM Tivoli Autonomic Computing Division. He is responsible for managing the integration of the autonomic computing self-healing initiatives into IBM products. He has worked for most of his career in software development, primarily in network computing and problem determination, and led several IBM-wide initiatives, including adoption of common problem determination standards, development of integrated hardware and software offerings, and management tools for network computing. Mike has been involved in several industry task forces and organizations and published several patents and industry papers and IBM white papers and Redbooks™.

Become a published author

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

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

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM® Corporation, International Technical Support Organization
  Dept. JN9B Building 905
  11501 Burnet Road
  Austin, Texas 78758-3493

Part 1

Introduction

In this part, we provide an overview of autonomic computing.

1

Autonomic computing overview

Autonomic computing is the ability of an IT infrastructure to adapt to change in accordance with business policies and objectives. Quite simply, it is about freeing IT professionals to focus on higher-value tasks by making technology work smarter, with business rules guiding systems to be self-configuring, self-healing, self-optimizing, and self-protecting.

The focus of this IBM Redbook is to introduce the problem determination aspects of the IBM Autonomic Computing initiative. In this chapter, we provide an overview of the IBM Autonomic Computing initiative and important concepts about autonomic computing in general.

1.1 What is autonomic computing?

The term “autonomic” comes from an analogy to the autonomic central nervous system in the human body, which adjusts to many situations automatically without any external help. We walk up a flight of stairs and our heart rate increases. If it is hot, we perspire. If it is cold, we shiver. We do not tell ourselves to do these things, they just happen.

Similarly, the way to handle the problem of managing a complex IT infrastructure is to create computer systems and software that can respond to changes in the IT (and ultimately, the business) environment, so the systems can adapt, heal, and protect themselves.

Guiding principles

The cost of technology continues to decrease, yet overall IT costs do not. With the expensive challenges that many companies face, IT managers are looking for ways to improve the return on investment of IT by:

- Reducing total cost of ownership
- Improving quality of service
- Accelerating time to value
- Managing IT complexity

The IBM autonomic computing vision is for intelligent, open systems that:

- Manage complexity
- “Know” themselves
- Continuously tune themselves
- Adapt to unpredictable conditions
- Prevent and recover from failures
- Provide a secure environment

Autonomic computing systems consist of four attributes. As illustrated in the following 4-quadrant chart shown in Figure 1-1 on page 5, they are:

- Self-configuring (able to adapt to changes in the system)
- Self-healing (able to recover from detected errors)
- Self-optimizing (able to improve use of resources)
- Self-protecting (able to anticipate and cure intrusions)

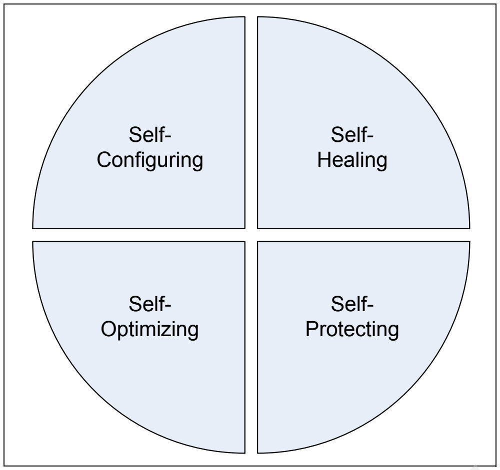


Figure 1-1 Autonomic computing attributes

Self-configuring

With the ability to dynamically configure itself, an IT environment can adapt immediately, with minimal intervention, to the deployment of new components or changes in the IT environment.

Self-healing

Self-healing IT environments can detect problematic operations (either proactively through predictions or otherwise) and then initiate corrective action without disrupting system applications. Corrective action could mean that a product alters its own state or influences changes in other elements of the environment. Day-to-day operations do not falter or fail because of events at the component level. The IT environment as a whole becomes more resilient as changes are made to reduce or help eliminate the business impact of failing components.

Self-optimizing

Self-optimization refers to the ability of the IT environment to efficiently maximize resource allocation and utilization to meet end users' needs with minimal intervention. In the near term, self-optimization primarily addresses the complexity of managing system performance. In the long term, self-optimizing components might learn from experience and automatically and proactively tune themselves in the context of an overall business objective.

Self-protecting

A self-protecting environment allows authorized people to access the right data at the right time and can take appropriate actions automatically to make itself less vulnerable to attacks on its runtime infrastructure and business data. A self-protecting IT environment can detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-service attacks, and general failures.

1.2 Autonomic computing concepts

In an autonomic environment, components work together, communicating with each other and with high-level management tools. They can manage or control themselves and each other.

Components can manage themselves to some extent, but from an overall system standpoint, some decisions need to be made by higher-level components that can make the appropriate trade-offs based on policies that are in place.

Let us start by looking at how a single entity is managed in an autonomic environment. Figure 1-2 represents the control loop that is the core of the autonomic architecture.

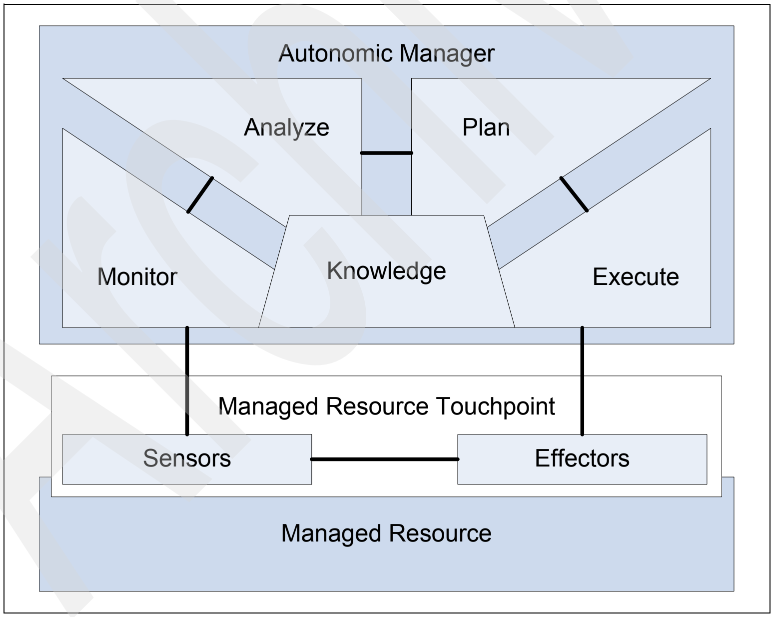


Figure 1-2 Autonomic computing control loop

1.2.1 Autonomic manager

The autonomic manager is a component that implements the control loop, also known as the MAPE loop. The architecture dissects the loop into four parts that share knowledge: monitor, analyze, plan, and execute:

- The monitor part provides the mechanisms that collect, aggregate, filter, manage, and report details (metrics and topologies) collected from an element.
- The analyze part provides the mechanisms that correlate and model complex situations. These mechanisms enable the autonomic manager to learn about the IT environment and help predict future situations.
- The plan part provides the mechanisms that structure the action needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.
- The execute part provides the mechanisms that control the execution of a plan with considerations for on-the-fly updates.

The four parts work together to provide the control loop function. Figure 1-2 on page 6 shows a structural arrangement of the parts, not a control flow. The bold line that connects the four parts should be thought of as a common messaging bus rather than a strict control flow. In other words, there can be situations where the plan part might ask the monitor part to collect more or less information. There could also be situations where the monitor part might trigger the plan part to create a new plan. The four parts collaborate using asynchronous communication techniques, like a messaging bus.

This architecture does not prescribe a particular management protocol or instrumentation technology. The architecture needs to work with the various computing technologies and standards that exist in the industry today, such as SNMP, Java Management Extensions (JMX), Distributed Management Task Force, Inc. (DMTF), Common Information Model (CIM), and vendor-specific APIs or commands, as well as with new technologies that will emerge in the future.

Given the diversity of the approaches that already exist in the IT industry, this architecture endorses Web services techniques for sensors and effectors. These techniques encourage implementers to leverage existing approaches and support multiple binding techniques and multiple marshalling techniques.

Multiple levels of autonomic managers

As implied by the previous references to complex scenarios, environments will consist of many managed resources and many autonomic managers. For instance, in an ideal environment, each component of an application might be a managed resource with its own autonomic manager. However, the application as a whole could also be seen as a managed resource with its own autonomic manager. In this case, the component-specific managers would have their own sensors and effectors that would enable them to report their status to and be controlled by the application's autonomic manager. In this way, the application could provide the intelligence or guidance to the individual components related to situations that affect the application as a whole. Policies would dictate what information and or actions would be controlled solely within the component and what should be shared with or controlled by the application's autonomic manager.

Likewise, the application might be a part of a business solution consisting of many related applications. The capability for this architecture to scale through multiple levels of autonomic managers enables it to be implemented in a phased approach with benefits accruing at every level.

1.2.2 Managed resources

The managed resource is a controlled system component. The managed resource can be a single resource or a collection of resources. The managed resource is controlled through its sensors and effectors.

The sensors provide mechanisms to collect information about the state and state transitions of an element. Sensors can be implemented using a set of **`get`** operations to retrieve information about the current state, or a set of management events (unsolicited, asynchronous messages, or notifications) that flow when the state of the element changes in a significant way, or both.

The effectors are mechanisms that change the state (configuration) of an element. In other words, the effectors are a collection of **`set`** commands or application programming interfaces (APIs) that change the configuration of the managed resource in some way.

The combination of sensors and effectors form the manageability interface (referred to as the managed resource touchpoint; see Figure 1-3 on page 9) that is available to an autonomic manager.

The manageability interface between the autonomic manager and a managed resource is organized into sensor and effector operations, which are defined as follows:

Sensor operations — Used to transmit events or properties to an autonomic manager.

Effector operations — Used to perform actions on the managed resource determined by an autonomic manager.

The architecture encourages the idea that sensors and effectors are linked together. For example, a configuration change that occurs through effectors should be reflected as a configuration change notification through the sensor interface.

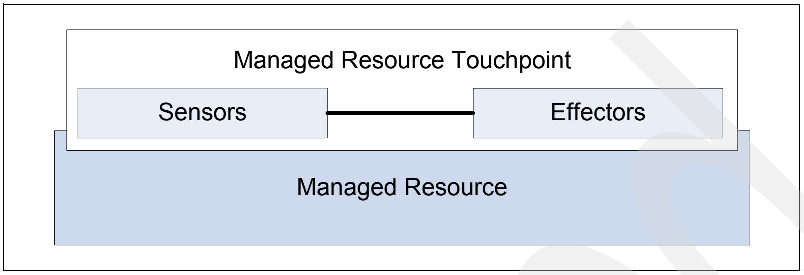


Figure 1-3 Sensors and effectors

Sensor and effector operations are organized into a set of interaction styles that define how managed resources and autonomic managers interact. Interaction styles are differentiated by whether the autonomic manager or the managed resource initiates the contact. The interaction styles defined for the sensor and effector operations are provided in Table 1-1.

Table 1-1 Sensor and effector operations and interaction styles

Operation	Interaction style
Sensor	retrieve-state Initiated by the autonomic manager.
Sensor	receive-notification Initiated by the managed resource.
Effector	perform-operation Initiated by the autonomic manager.
Effector	call-out-request Initiated by the managed resource.

Web services can (and will) be used to implement sensor-effector functions. By using a Web services architecture for communication to the managed resource touchpoint, current approaches to resource management can be reused and wrapped with a Web service.

1.2.3 Autonomic manager collaboration

The numerous autonomic managers in a complex IT system must work together to deliver autonomic computing to achieve common goals. For example, a database system needs to work with the server, storage subsystem, storage management software, Web server, and other elements of the system in order for the IT infrastructure as a whole to become a self-managing system. The sensors and effectors provided by the autonomic manager facilitate collaborative interaction with other autonomic managers.

In addition, autonomic managers can communicate with each other in both peer-to-peer and hierarchical arrangements. Figure 1-4 shows an example of a simple IT system that includes two business applications: a customer order application and a vendor relationship application. Separate teams manage these applications. Each of these applications depends on a set of IT resources—databases and servers—to deliver its function. Some of these resources (DB 3, DB 4, Server B, and Server C) are shared between the applications, which can be managed separately. There is a minimum of four management domains (decision-making contexts) in this example. Each of the applications (customer order and vendor relationship) has a domain, focused on the business system it implements. In addition, there is a composite resource domain for managing the common issues across the databases and a composite resource domain for managing common issues for the servers.

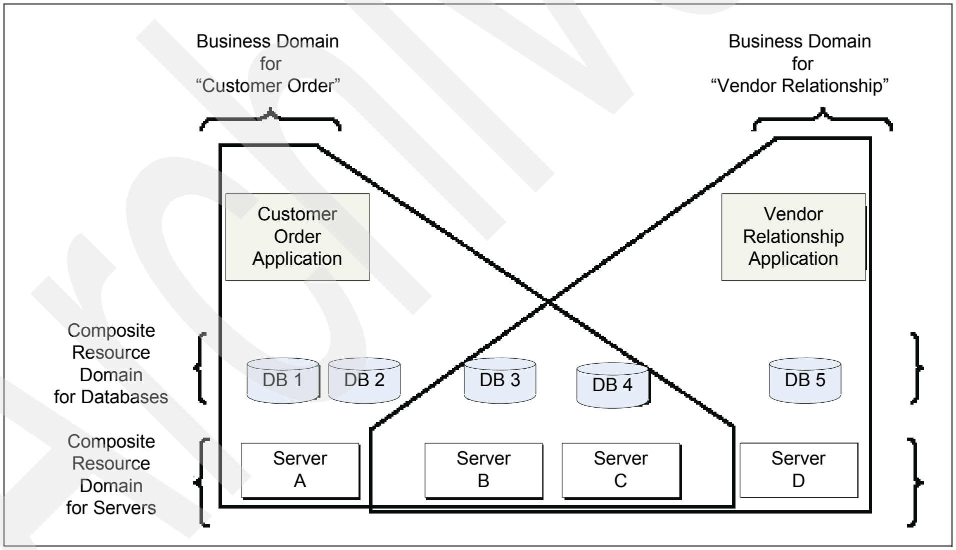


Figure 1-4 IT systems can share resources to increase efficiency

Applying the autonomic computing architecture to this example, Figure 1-5 on page 11 illustrates some of the autonomic managers that either directly or indirectly manage DB 3 and some of the interaction between these autonomic managers. There are six autonomic managers in Figure 1-5 on page 11: one for each of the management domains, one embedded in the DB 3 resource, and one dedicated to the specific database resource.

Because the decision-making contexts for these autonomic managers are interdependent and self-optimizing, the autonomic managers for the various contexts will need to cooperate. This is accomplished through the sensors and effectors for the autonomic managers, using a matrix management protocol.

This protocol makes it possible to identify situations in which there are multiple managers and enables autonomic managers to electronically negotiate resolutions for domain conflicts, based on a system-wide business and resource optimization policy.

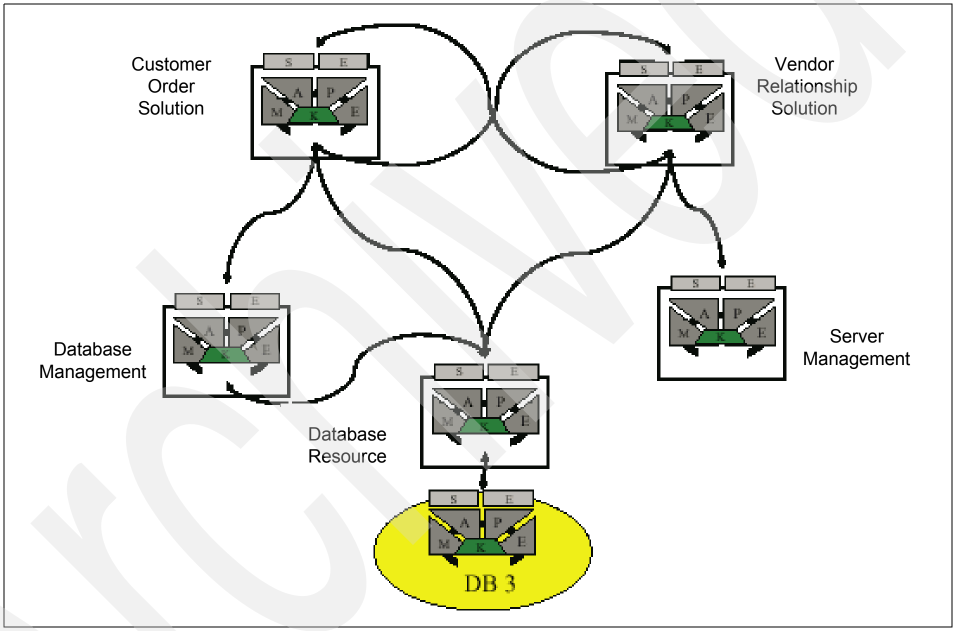


Figure 1-5 Direct and indirect management of DB 3 by six autonomic managers

1.2.4 Autonomic manager knowledge

Data used by the autonomic manager's four components are stored as shared knowledge. The shared knowledge includes things such as topology information, system logs, performance metrics, and policies. The knowledge used by a particular autonomic manager could be created by the monitor part, based on the information collected through sensors, or passed into the autonomic manager through its effectors. An example of the former occurs when the monitor part creates knowledge based on recent activities by logging the notification it receives from a managed resource into a system log. An example of the latter is policy. A policy consists of a set of behavioral constraints or preferences that influence the decisions made by an autonomic manager. Specifically, the plan part of an autonomic manager is responsible for interpreting and translating policy details. The analysis part is responsible for determining if the autonomic manager can abide by the policy, now and subsequently.

1.2.5 Policies for autonomic managers

Although a detailed discussion of how policies would be implemented is outside the scope of this book, for architectural completeness, we provide a brief explanation.

An autonomic computing system requires a uniform method for defining the policies that govern the decision making for autonomic managers. A policy specifies the criteria that an autonomic manager uses to accomplish a course of action. Policies are a key part of the knowledge used by autonomic managers to make decisions, essentially controlling the planning portion of the autonomic manager. By defining policies in a standard way, they can be shared across autonomic managers to enable entire systems to be managed by a common set of policies.

Policies must be specified consistently for an autonomic system to behave cohesively. The autonomic computing blueprint published by IBM (see `http://www.ibm.com/autonomic/pdfs/ACwpFinal.pdf`) is currently defining the specifications and capabilities for policy-based autonomic managers. This definition includes:

- Specification of canonical configuration parameters for management elements
- Format and schema used to specify user requirements or criteria
- Mechanisms used, including wire formats, for sharing and distributing policies
- Schema used to specify and share policy between autonomic managers

One of the key things that an autonomic system must do is to share policies among autonomic managers, so this capability will leverage and extend the policy standards.

1.3 Evolving to autonomic computing

Autonomic computing will not be an instant transformation for enterprise class infrastructures. The transition will be gradual, with new technologies (both hardware and software) being implemented at various stages and levels. The five levels, or transition steps, of autonomic maturity are described in the following sections.

Level 1: Basic

The starting point where most systems are today, this level represents manual computing in which all system elements are installed and managed as separate entities. These environments require an extensive, highly skilled IT staff who must aggregate and analyze multiple sources of system generated data and manage the IT environment from a broad spectrum of individual consoles with multiple interfaces. Highly skilled staff sets up, monitors, and eventually replaces system elements.

Level 2: Managed

Customers achieve this level through consolidation of data and actions from disparate systems onto fewer consoles, using management tools such as those offered by the IBM Tivoli portfolio. The IT staff continues to analyze this data and initiate management actions; benefits include greater system awareness and improved productivity.

Level 3: Predictive

At this level, the system monitors and correlates data to recognize patterns and recommends actions that are approved and initiated by the IT staff. At the predictive level, management integration across multiple components begins to occur. With the implementation of predictive capabilities, benefits include the possibility to reduce dependency on deep skills.

Level 4: Adaptive

At the adaptive level, not only does the system monitor, correlate, and develop action plans, the system also takes actions according to established policies. This level enables the staff to manage performance against service-level objectives. This assists an organization in establishing a balance between human and system interactions and helps the IT infrastructure to be more responsive to changing business conditions and improves resiliency.

Level 5: Autonomic

At this final level, the components of the infrastructure are well integrated and dynamically manage themselves according to business rules and policies. The autonomic level allows staff to focus on enabling business requirements. Business policy becomes the primary driver behind IT management, and the business benefits from the improved business agility and resiliency.

Figure 1-6 on page 14 gives an overview of the autonomic maturity index.

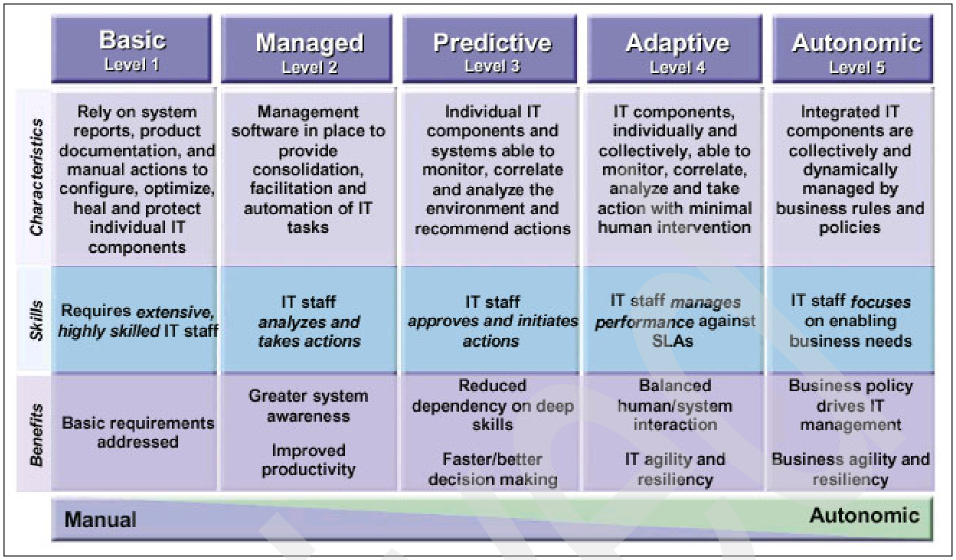


Figure 1-6 Autonomic maturity index

Architectural levels of maturity

The earlier discussion about autonomic maturity levels demonstrated that self-managing capabilities will not be incorporated in one quick step. Rather, they constitute a concept that permeates all aspects of a system. Figure 1-7 on page 15 reinforces this observation by showing a possible relationship between the maturity levels, the three decision making contexts (resource element context, composite resource context, and business solution context), and the parts of the autonomic manager. This mapping results in two important observations:

- As the maturity levels increase, the decision-making context for the autonomic manager changes. The pyramid on the right side summarizes the three different decision-making contexts in which autonomic managers can implement self-managing capabilities.

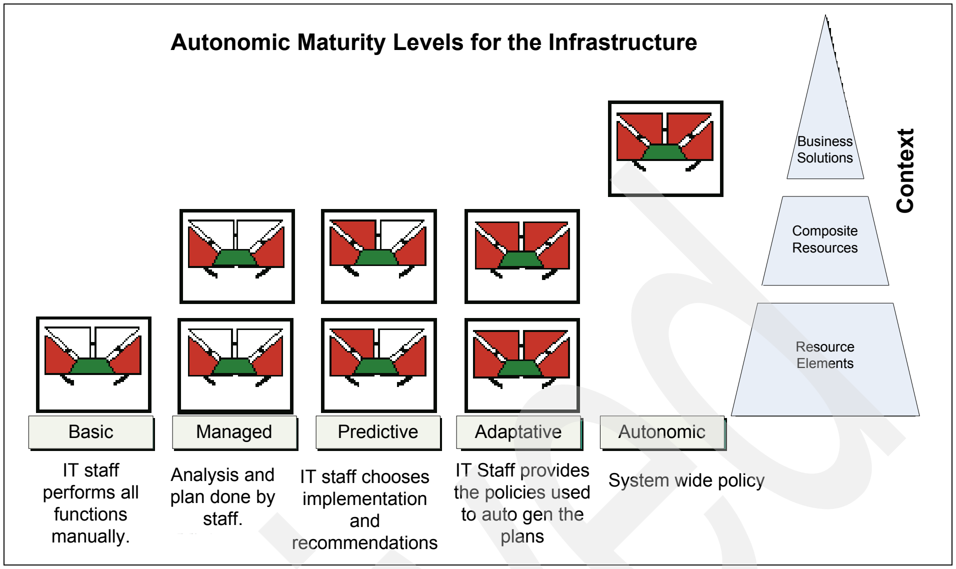


Figure 1-7 Autonomic maturity levels for the IT infrastructure

- The different parts of the autonomic manager are implemented at each maturity level. The monitor and execute part of the autonomic manager are implemented at the basic and managed levels. So, at these two levels, IT personnel are responsible for performing the function of the analyze and plan parts. The analyze part of the autonomic managers is supplied at the predictive maturity level. At this level, the IT professional is responsible for the plan function. In the adaptive and the autonomic level, all of the parts of the autonomic manager are working, so the IT personnel can delegate the work to the system. The difference between these two maturity levels is the decision-making context. The adaptive maturity level supports either the resource element or the composite element context, and the autonomic level supports the business solution context.

As Figure 1-7 shows, the progressive implementation of the architecture occurs for each of the three contexts. This is due to the fact that to deliver a self-managing capability in a business system context requires a self-managing capability in the lower context.

1.4 Value of autonomic computing

By enabling computer systems to have self-configuring, self-healing, self-optimizing, and self-protecting features, autonomic computing is expected to have many benefits for business systems, such as reduced operating costs, lower failure rates, more security, and the ability to have systems that can respond more quickly to the needs of the business within the market in which they operate.

1.4.1 Software vendors

The primary business imperatives for enterprises today include driving down the total cost of ownership of the infrastructure while at the same time increasing user productivity.

Autonomic computing is a core part of enabling automation, which is one of the three major characteristics of the On Demand Operating Environment, the other two being:

- Integrated, core business systems are linked within the business and across different enterprises.
- Virtualized data and applications are managed centrally, through a grid-like infrastructure, which optimizes the use of computing capacity and delivers better performance.

The implications for an autonomic, on demand business approach are immediately evident: A network of organized, smart computing components can give clients what they need, when they need it, without a conscious mental or even physical effort.

Independent software vendors (ISVs) will play a key role through building automation and autonomic functionality into their product sets, both hardware and software.

After systems and networks begin to feature these attributes, IT professionals will be able to work at a higher level. Managers could set business goals, and computers would automatically set the IT actions needed to deliver them. For example, in a financial-trading environment, a manager might decide that trades have to be completed in less than a second to realize service and profitability goals. It would be up to software tools to configure the computer systems to meet those metrics.

As enterprises build infrastructures that support the dynamic needs of on demand computing, software products that conform to standards and are enabled for autonomic computing will be more attractive to the corporate decision makers choosing the components for their IT infrastructure.

1.4.2 Enterprises

A few examples of the results delivered by implementing autonomic computing solutions with self-management characteristics include:

- Operational efficiency

  As the IT infrastructure becomes more autonomic, executing business policy becomes the focus of IT management. Management of the business and of IT will no longer be separate, and possibly conflicting, activities.

  Self-configuring and self-optimizing technologies drive efficiency in running and deploying new processes and capabilities.

- Supporting business needs with IT

  The actualization of self-configuring systems speeds the deployment of new applications required to support emerging business requirements.

  Self-healing capabilities help deliver the 24x7 availability required to keep businesses running.

- Workforce productivity

  Workforce productivity is enhanced when the focus is on management of business processes and policies, without the need to translate these needs into actions that separately manage the supporting technology.

Systems that are self-managing free up IT resources, which then can move from mundane system management tasks to focusing on working with users to solve business problems.

1.5 Core capabilities

In order for the autonomic managers and the managed resources in an autonomic system to work together, the developers of these components need a common set of capabilities. This section conceptually describes an initial set of core capabilities that are needed to build autonomic systems. These core capabilities include:

- Solution installation
- Common systems administration
- Problem determination
- Autonomic monitoring
- Complex analysis
- policy-based management
- Heterogeneous workload management

These capabilities are enabled through a series of tools, technologies, and facilities that are described in Part 2, "Self-Managing Autonomic Technology" on page 23.

1.5.1 Solution installation

Today, there are several installation, configuration, and maintenance mechanisms for software solutions. Having various mechanisms creates difficulties for customers installing software in complex systems environments due to the differences and idiosyncrasies of many system administration tools and distribution packaging formats. These problems are further compounded in a Web services environment, where application functionality can be composed dynamically. From an autonomic systems perspective, lack of solution knowledge inhibits important elements of self-configuring, self-healing, self-optimizing, and self-protecting.

A common solution knowledge capability eliminates the complexity introduced by many formats and many installation tools. By capturing installation and configuration information in a consistent manner, autonomic managers can share the facilities and information regarding the installed environment in contexts beyond installation, such as problem determination or optimization.

1.5.2 Common systems administration

Autonomic systems require common console technology to create a consistent human interface for the autonomic managers of the IT infrastructure. The common console capability provides a framework for reuse and consistent presentation for other autonomic core technologies.

The primary goal of a common console is to provide a single platform that can host all of the administrative console functions in server, software, and storage products in a manner that enables users to manage solutions rather than managing individual systems or products. Administrative console functions range from setup and configuration to solution runtime monitoring and control.

The value to the customer is reduced cost of ownership, attributable to more efficient administration, and reduced learning curves as new products and solutions are added to the autonomic system environment. By enabling increased consistency of presentation and behavior across administrative functions, the common console creates a familiar user interface that promotes reusing learned interaction skills versus learning new and different product-unique interfaces.

1.5.3 Problem determination

Autonomic managers take actions based on problems or situations they observe in the managed resource. Therefore, one of the most basic capabilities is being able to extract high-quality data to determine whether or not a problem exists in managed resources. In this context, a problem is a situation in which an autonomic manager needs to take action. A major cause of poor quality information is the diversity in the format and content of the information provided by the managed resource.

To address the diversity of the data collected, a common problem determination architecture normalizes the data collected, in terms of format, content, organization, and sufficiency. To do this, it defines a base of data that must be collected or created when a situation or event occurs. This definition includes information about both the kinds of data that must be collected and the format that must be used for each field collected. The problem determination architecture categorizes the collected data into a set of situations. Situations represent the state of an application, such as starting or stopping. The technologies used to collect the autonomic data must be capable of accommodating existing data sources (such as logs and traces) and data that is supplied using the standard format and categorization. To accommodate this existing data, the architecture defines an adapter/agent infrastructure that will provide the ability to plug in an adapter to transform data from a component-specific format to the standard format and sensors to control data collection.

The scope of this book is to introduce the problem determination aspects of the IBM Autonomic Computing initiative through practical example scenarios. The remainder of this book goes into further details about the tools and techniques used in the example scenarios and is meant to be used as a reference to developers creating self-managing autonomic capabilities in their applications.

1.5.4 Autonomic monitoring

Autonomic monitoring is a capability that provides an extensible runtime environment for an autonomic manager to gather and filter data obtained through sensors. Autonomic managers can use this capability as a mechanism for representing, filtering, aggregating, and performing a range of analysis of sensor data. An autonomic manager using this autonomic monitoring functionality can help manage certain applications or resources more effectively through:

- Multiple source data capture: Enables processing of data from industry standard APIs and from any custom data interfaces that a particular application uses.

- Local persistence checking: Links corrective actions or responses to the repeated occurrences of a problem condition so that a single point-in-time threshold exception does not immediately trigger a costly and unnecessary troubleshooting response. It is important to ensure that individual events or conditions are not monitored in isolation, but rather within context of other related events around the same time period. For instance, if a monitor senses that CPU utilization is over 80% for a monitoring cycle, that could reflect a momentary spike and not an on-going condition. But if over a period of time, it is recognized that the CPU is consistently utilized at over 80%, this could indicate a condition that should be analyzed.
- Local intelligent correlation: Recognizes a number of metrics in aggregate as a *problem signature*, enabling root cause identification and response to problems rather than symptoms. For example, the fact that a client cannot access a database server could indicate that the database server has failed. However, if at the same time, it is recognized that a network router is down that would prevent clients from accessing the server, it might not make sense to spend resources looking into a possible problem with the server.
- Local data store and reporting: Provides a real-time *heart monitor* that determines whether the application environment and individual applications are functioning properly.

Refer to Chapter 2, "IBM Autonomic Computing Toolkit overview" on page 25, Chapter 6, "Autonomic Management Engine and resource models" on page 193, and Chapter 7, "Advanced Autonomic Management Engine topics" on page 231 for details about the Autonomic Management Engine.

1.5.5 Complex analysis

Autonomic managers need to have the capability to perform complex data analysis and reasoning on the information provided through sensors. The analysis will be influenced by stored knowledge data.

In April 2003, IBM introduced the autonomic computing blueprint, which is based on open standards and is designed to develop self-managing systems that use intelligent control loops to collect information from the system, make decisions, and adjust the system as necessary. The autonomic computing blueprint defines complex analysis technology building blocks that autonomic managers can use to represent knowledge, perform analysis, and do planning.

An autonomic manager's ability to quickly analyze and make sense of this data is crucial to its successful operation. Common data analysis tasks include classification, clustering of data to characterize complex states and detect similar situations, prediction of anticipated workload and throughput based on past experience, and reasoning for causal analysis, problem determination, and optimization of resource configurations.

1.5.6 Policy-based management

An autonomic computing system requires a uniform method for defining the policies that govern the decision-making for autonomic managers. Policy specifies the criteria that an autonomic manager uses to accomplish a definite goal or course of action. Policies are a key part of the knowledge used by autonomic managers to make decisions, essentially controlling the planning portion of the autonomic manager. By defining policies in a standard way, they can be shared across autonomic managers to enable entire systems to be managed by a common set of policies.

1.5.7 Heterogeneous workload management

Heterogeneous workload management includes the capability to instrument system components uniformly to manage workflow through the system. Business workload management is a core technology that monitors end-to-end response times for transactions or segments of transactions, rather than at the component level, across the heterogeneous infrastructure.

1.5.8 Summary

Autonomic computing systems require deployment of autonomic managers throughout the IT infrastructure, managing resources that include other autonomic managers from a diverse range of suppliers. These systems, therefore, must be based on open industry standards.

The architecture reinforces the fact that self-managing implies intelligent control loop implementations that will monitor, analyze, plan, and execute using knowledge of the environment. In addition, the loops can be embedded in resource runtime environments, or delivered in management tools. These control loops collaborate using a matrix management protocol.

This chapter presented an introduction to autonomic computing, along with an overview of the autonomic blueprint to assist in understanding the self-managed autonomic technology and phases of maturity in implementation.

Part 2

Self-Managing Autonomic Technology

This part of the book includes chapters about the IBM Autonomic Computing Toolkit, Common Base Events, the Generic Log Adapter, the Log and Trace Analyzer, the Autonomic Management Engine and resource models, and finally a chapter that describes advanced Autonomic Management Engine topics.

2

IBM Autonomic Computing Toolkit overview

The IBM Autonomic Computing Toolkit is a collection of technologies, tools, examples, scenarios, and documentation that is designed for users wanting to learn about, adapt, and develop autonomic behavior in their products and systems. At the time of writing this book, the current version of the IBM Autonomic Computing Toolkit is 2.0. This toolkit is currently available for download from the IBM developerWorks® Web site at:

http://www.ibm.com/developerWorks/autonomic

This chapter provides a brief overview of the IBM Autonomic Computing Toolkit and its various components. This toolkit will continue to be enhanced to provide additional and more advanced capabilities.

Although there are facilities to allow for development of autonomic managers, our primary focus in this book is to use the resources provided by the IBM Autonomic Computing Toolkit for problem determination, enabling applications and other resources to be managed. However, even this includes facilities to create simple control loops so that specific resources can be self-configuring, self-healing, self-optimizing, and self-protecting, and therefore generating real benefits today. As higher-level autonomic managers become available in the future, resources enabled today will be manageable by those managers and will reap even more benefits for the enterprise.

2.1 Key component areas

The IBM Autonomic Computing Toolkit provides content that enables the development of self-managing autonomic capabilities. The content of the IBM Autonomic Computing Toolkit can be divided into four main categories:

- Technologies
- Tools
- Scenarios
- Information and documentation

Technologies

Product and system capabilities that can take advantage of technologies in the IBM Autonomic Computing Toolkit include problem determination, common systems administration, and solution installation and deployment.

Problem determination capabilities can be enhanced with the Autonomic Management Engine (AME), the Generic Log Adapter, the Log and Trace Analyzer, and Common Base Events. The Integrated Solutions Console is used to build effective common systems administration capabilities. The Solution Install technologies provide capabilities for solution deployment and configuration.

Tools

In addition to delivering these technologies, the IBM Autonomic Computing Toolkit provides the tooling necessary to customize the technologies so that solutions can be created to meet specific needs. Tools such as the Integrated Solutions Console Toolkit, Resource Model Builder, Adapter Rule Editor tool, and other Eclipse plug-ins are included to assist in the creation of custom solutions.

Scenarios

Scenarios are also provided that show how the technologies work together and how they can be used in realistic situations. The scenarios are meant to be testing environments that allow for investigation. All scenarios are demonstrated using the technologies and tools available in the IBM Autonomic Computing Toolkit. The IBM Autonomic Computing Toolkit includes a Problem Determination scenario performing self-healing and two automated installation scenarios performing self-configuring tasks.

Information and documentation

The IBM Autonomic Computing Toolkit also focuses on educating users about autonomic computing. Detailed individual technology and tooling documentation is provided along with documentation to assist with developing autonomic solutions.

2.2 IBM Autonomic Computing Toolkit tools and technologies

The technologies and tools contained in the current version of the IBM Autonomic Computing Toolkit are intended to assist product developers develop self-managing autonomic capabilities in their products, thus increasing the autonomic maturity level of their applications.

The IBM Autonomic Computing Toolkit provides the following tools and technologies.

2.2.1 Autonomic Management Engine

The Autonomic Management Engine components of the IBM Autonomic Computing Toolkit can be used to develop an adaptive level of self-management, autonomic maturity level 4.

The Autonomic Management Engine provides an example implementation of an autonomic manager and includes built-in representations of the four parts of the control loop (monitor, analyze, plan, and execute) presented in Figure 2-1 on page 32.

Chapter 6, "Autonomic Management Engine and resource models" on page 193 and Chapter 7, "Advanced Autonomic Management Engine topics" on page 231 describes the Autonomic Management Engine in great detail.

Autonomic Management Engine Version 1.1 is shipped with the IBM Autonomic Computing Toolkit Version 2.0.

2.2.2 Resource Model Builder

Effective self-management and monitoring capabilities are performed by a *resource model* and developed using the IBM Tivoli *Resource Model Builder* tool (an Eclipse plug-in) provided by the IBM Autonomic Computing Toolkit. Resource models contain specific metrics, events, thresholds, and parameters that are used to determine the health of IT resources along with specifications for corrective actions in the event of detected failures and error conditions. The

Autonomic Management Engine provides services for installing and managing resource models.

Refer to Chapter 6, “Autonomic Management Engine and resource models” on page 193 for details about the Resource Model Builder. In addition, the case study scenario chapters in this book provide details about the creation, installation, and management of resource models.

Resource Model Builder Version 1.3 is shipped with the IBM Autonomic Computing Toolkit Version 2.0.

2.2.3 Generic Log Adapter

Several technologies and tools are provided to help developers create touchpoints that enable managed resources to communicate with autonomic managers. The Generic Log Adapter is an example of such a technology and is included in the IBM Autonomic Computing Toolkit to translate product log messages into a Common Base Event data format.

The Common Base Event is an XML structure that can be consumed by an autonomic manager. For information about Common Base Events, usage, and creation best practices, refer to Chapter 3, “Common Base Events” on page 35. In addition, additional information can be found in the Eclipse Test and Performance Tools Platform (TPTP) Project from the Eclipse organizations at:

`http://www.eclipse.org/tptp/index.html`

The Generic Log Adapter (GLA) provides the ability to take a product log file and convert the messages into a Common Base Event format so that the product can become a managed resource. The IBM Autonomic Computing Toolkit includes the GLA as a technology to help products adapt to the autonomic reference architecture without requiring the product to change the way it creates its log files.

The IBM Autonomic Computing Toolkit provides the *Adapter Configuration Editor* tool to be used in conjunction with the Generic Log Adapter, providing the development environment for specific parser rules to create Common Base Events. The Adapter Configuration Editor tool is provided as an Eclipse plug-in. The toolkit also supplies a Generic Log Adapter Runtime and Rule Sets bundle, providing the ability to run adapter configuration files created by the Adapter Configuration Editor.

Chapter 4, “Generic Log Adapter” on page 77 provides a great deal of information about the Generic Log Adapter, including usage, configuration, and deployment examples.

Generic Log Adapter Version 3.0.0 is shipped with the IBM Autonomic Computing Toolkit Version 2.0.

2.2.4 Log and Trace Analyzer

The Log and Trace Analyzer can be used to consume Common Base Event messages, analyze them, and present a correlated view of the log events. The Log and Trace Analyzer is thus an example of a partial implementation of an autonomic manager, covering the monitor and analyze parts of the control loop. The Log and Trace Analyzer can be used to achieve autonomic maturity levels 2, managed, and 3, predictive self-healing attributes.

The Log and Trace Analyzer includes a facility to import any type of log file remotely. This facility is provided by the Remote Agent Controller component.

Refer to Chapter 5, “Log and Trace Analyzer” on page 137 for information and a complete usage example of the Log and Trace Analyzer tool.

Log and Trace Analyzer Version 3.0.0 is shipped with the IBM Autonomic Computing Toolkit Version 2.0.

2.2.5 Agent Controller

The Agent Controller bundle contains the Remote Agent Controller (RAC) for multiplatforms to allow processing of remote log files using the Log and Trace Analyzer tool.

Refer to Chapter 5, “Log and Trace Analyzer” on page 137 for information about the Remote Agent Controller.

Agent Controller Version 3.0.0 is available with the IBM Autonomic Computing Toolkit Version 2.0.

2.2.6 Integrated Solutions Console

The Integrated Solutions Console component provides a user environment to host self-management capabilities. It is a Web-based infrastructure based on industry-standard technologies. This console can include administrative console functions for managing multiproduct customer IT environments. The IT environments can include IBM hardware and software, Business Partner applications, client applications, all of these. Administrative console functions range from setup and configuration to runtime monitoring and control and can be used to develop more effective managed systems with maturity level 2 and higher.

Integrated Solutions Console Version 5.0.2 is shipped with the IBM Autonomic Computing Toolkit Version 2.0.

Integrated Solutions Console Toolkit

The Integrated Solutions Console Toolkit is a development environment for creating Integrated Solutions Console plug-ins. It includes a Integrated Solutions Console run time, developers' information center, and sample components, including an Integrated Solutions Console plug-in for IBM WebSphere.

The Integrated Solutions Console Toolkit is included in the Integrated Solutions Console Version 5.0.2 bundle.

2.2.7 Eclipse tooling package

The Generic Log Adapter Configuration Editor and Log and Trace Analyzer tooling package and the Resource Model Builder package are provided as plug-ins to an Eclipse base. The Eclipse Version 3.0 or later environment installed provides the necessary base tooling support for the IBM Autonomic Computing Toolkit.

2.3 IBM Autonomic Computing Toolkit scenarios

The scenarios contained in the current version of the IBM Autonomic Computing Toolkit are intended to assist product developers with practical usage examples of the various tools and technologies provided by the toolkit.

The IBM Autonomic Computing Toolkit provides the following tools and technologies.

2.3.1 Problem Determination scenario

The Problem Determination scenario represents a simple self-managing system that uses an intelligent control loop to collect system information, analyze it, plan appropriate responses, and then make necessary adjustments to resolve problems. The Problem Determination scenario demonstrates an autonomic control loop operating on two sample IBM products, IBM WebSphere Application Server - Express and IBM Cloudscape™ database management, each of which outputs product information to a log file. After an event has been detected, the autonomic manager issues corrective action on the appropriate managed resource. Administration and control of the scenario are handled by a custom plug-in component to the Integrated Solutions Console that was created for the Problem Determination scenario.

The Problem Determination scenario is included in the Problem Determination Scenario bundle; it provides the customized components to establish the specific scenario and demonstration support code to control the demonstration. The following components are required prior to installing the Problem Determination Scenario bundle and are available in the IBM Autonomic Computing Toolkit:

- Autonomic Management Engine
- Generic Log Adapter Runtime and Rule Sets
- Integrated Solutions Console
- Eclipse tooling package

The Problem Determination Scenario bundle, all used technologies, and prerequisites can be downloaded from the IBM developerWorks Web site at:

http://www.ibm.com/developerworks/autonomic/probdet1.html

2.3.2 Solution Install and Deployment scenario

The Solution Install and Deployment scenario is also part of the IBM Autonomic Computing Toolkit and demonstrates another example of the interaction between an autonomic manager and managed resources to achieve a self-management behavior, in this case, self-configuring. An autonomic maturity level 3 can be achieved by developing installation processes incorporating Solution Install and Deployment technologies.

The components of the Solution Install and Deployment scenario of the IBM Autonomic Computing Toolkit can be downloaded from the IBM developerWorks Web site at:

http://www.ibm.com/developerworks/autonomic/solins1.html

2.4 Supporting the autonomic computing architecture

Looking at the control loop shown in Figure 2-1 on page 32, let us consider the technologies and capabilities provided by the IBM Autonomic Computing Toolkit needed for each stage. This will allow us to better understand the specific facilities provided by the toolkit and their intended usage.

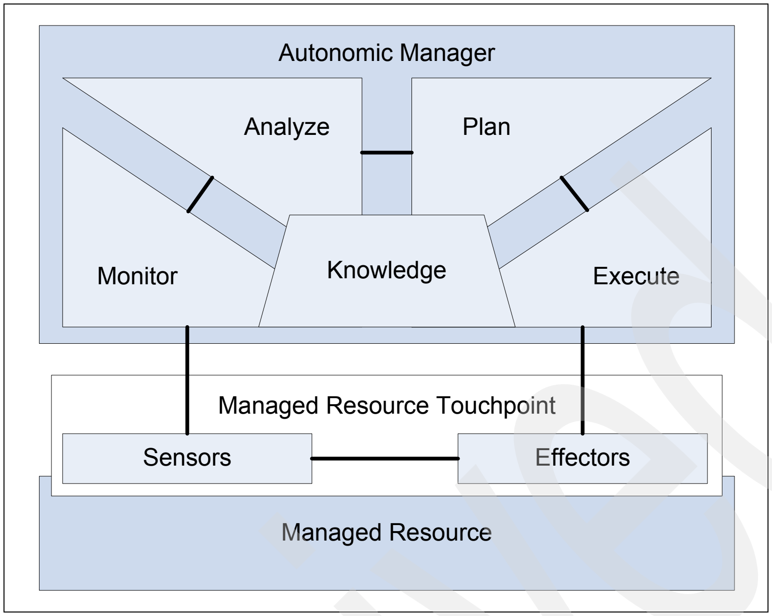


Figure 2-1 Autonomic computing control loop

If we start with the managed resource, the first step might be to consider how information is passed to the autonomic manager. This would be accomplished through the sensor. In the case of unsolicited events, a current application or resource might generate its own log file. In this case, the Generic Log Adapter (GLA) facility would be used to convert log entries to Common Base Events. It is also possible (and probably more efficient) for an application to generate Common Base Events directly. Both of these methods are described in detail in Chapter 3, "Common Base Events" on page 35.

The Generic Log Adapter is an example of a facility that helps adapt a product to participate in the autonomic computing architecture by generating Common Base Events that can be consumed by an autonomic manager.

GLA provides the ability to take a product log file and convert the messages into the Common Base Event data format. Log files are first parsed using a rule-based parser and are then translated into the Common Base Event format.

A single GLA Runtime can be used to parse the log files of multiple products as long as the rules have been defined for each log message format. The adapter includes a handler that can pass the Common Base Event information to the autonomic manager using the sensor and effector interfaces.

On the top of the loop is the autonomic manager. It consists of the monitoring, analysis, plan, and execute components. The IBM Autonomic Computing Toolkit currently ships a relatively simple implementation of these components called Autonomic Management Engine (AME). The AME is primarily intended to provide an implementation that can be used to test the interfaces and other components of the autonomic environment.

AME monitors system resources using resource models, sends aggregated events, and performs corrective actions for problems. AME constantly monitors the system looking for events to handle. AME is available in the IBM Autonomic Computing Toolkit in the AME bundle.

Moving clockwise around the loop, at the autonomic manager, the next component is the monitor. This component can either consume unsolicited Common Base Events or it can request specific information from the sensor. Resource models provide the building blocks for monitoring Common Base Events in an autonomic computing environment through automated best practices.

The Resource Model Builder provides a standard, Eclipse-based interface providing a step-by-step wizard to build resource models. See Chapter 6, "Autonomic Management Engine and resource models" on page 193 for details about using the Resource Model Builder.

The Resource Model Builder uses out-of-the box, predefined resource models to specify which resource data is accessed from the system at run time and how this data is processed. For example, the Process resource model obtains data related to processes running on the system. Performance data is automatically collected by the resource model and processed by an appropriate algorithm to determine whether or not the system is performing to expectations.

When a resource model is run, it gathers data at regular intervals, known as cycles; the duration of a cycle is the cycle time. A resource model with a cycle time of 60 seconds gathers information every 60 seconds. The data collected is a snapshot of the status of the resources specified in the resource model. Each of the supplied resource models has a default cycle time, which can be modified as required.

Each resource model defines one or more thresholds. A threshold is a named property of the resource with a default value that can be modified in the customization phase.

Common Base Events are then analyzed by the resource models, which contain specific thresholds and metrics used to determine if corrective actions are required or additional information needs to be gathered. This implements the analyze, plan, and execute components of the autonomic manager.

The Log and Trace Analyzer (LTA) is an example of a partial implementation of the autonomic manager, covering the monitor and analyze parts of the control loop.

The LTA enables viewing, analysis, and correlation of log files. This tool makes it easier and faster to debug and resolve problems within multi-tier systems by consuming data in the Common Base Event format and providing specialized visualization and analysis of the data.

The LTA contains a log analysis engine. The role of this engine is to provide an algorithm that takes an incident that is recorded in a log file as an input parameter, match this incident based on predefined rules against the symptoms of an available symptom database, and return an array of objects representing the solutions and directives for the matched symptoms. The LTA provides a default implementation of an analysis engine and a set of instruments that could be used to implement a custom analysis engine.

The LTA enables the writing of Java code-based parsers. Parsers written in Java code are commonly used when the individual log messages are very long and complex.

The IBM Autonomic Computing Toolkit provides numerous out-of-the-box parsers and rules for several existing IBM products.

2.5 Summary

The IBM Autonomic Computing Toolkit is available to provide technologies, tools, samples, and documentation to allow software developers to begin to enable resources to be managed in an autonomic environment.

The current toolkit includes facilities to take advantage of a browser-based common interface for system administration. It also provides technologies to enable common solution installation and configuration. However, in this book, we are not focusing on either of these facilities. Instead, we focus on facilities around problem determination, with specific emphasis on enabling resources to achieve self-healing capabilities. To do this, we use a series of case study scenarios using technology and tools provided by the IBM Autonomic Computing Toolkit and other self-managing autonomic technologies. We present the case study scenarios in Part 3, “Case study scenarios” on page 257.

3

Common Base Events

To resolve the issue of disparate data available for problem determination and self-healing systems, there is a need to adopt a formal, disciplined, and standard approach to the creation of problem data reported by IT components.

The development of a common message format, called Common Base Event, addresses the issue of structuring log files in a standard way so that multiple log files can be analyzed and correlated seamlessly.

This chapter describes the structure of Common Base Events, best practices about how they can be created, and how they can be generated.

3.1 What is a Common Base Event?

Before going into detail about Common Base Events, two elements must be defined, situations and events:

Situation
: This is the state change in a resource that causes data to be reported. Situations are defined by the category of the state change they represent, such as a START situation or a STOP situation.

Event
: This can be defined as any significant change in the state of an IT resource and can be generated for various reasons, such as for reporting a problem, a resolution to a problem, or the completion of a task. Events encapsulate message data sent as the result of a problem or situation in an IT environment.

The Common Base Event (CBE) format for log and trace information defines the structure of an event in a consistent and common format. Its foundation is the fact that messages are represented by events that are indicatives of a situation.

The Common Base Event facilitates the effective intercommunication among disparate enterprise components, which support problem determination and autonomic computing functions in an enterprise. It is emerging to be the *de facto* standard for reporting events in many enterprise applications. Common Base Events are adopted by the Eclipse project. For more information about the Eclipse Test and Performance Tools Platform (TPTP) Project, see:

`http://www.eclipse.org/tptp/index.html`

Different kinds of events, such as logging, tracing, management, and business events, can be mapped to Common Base Events. The Common Base Event model ensures consistency of format and the necessary scope for a consistent message content of such events through an XML schema definition. For those unfamiliar with Extensible Markup Language (XML), it is described in detail at:

`http://www.w3.org/TR/2000/REC-xml-20001006`

An XML schema introduction is available at:

`http://www.w3.org/TR/2001/REC-xmlschema-0-20010502`

In addition to ensuring consistency of format and content, the Common Base Event model inspires completeness. Creating meaningful and useful messages is a difficult task for developers, because many details of the event, such as the situation and the context in which the situation occurred, should be placed into a limited space. The Common Base Event schema simplifies definitions and contains prescribed elements that guarantee that adequate information will be logged.

Important: Following a standard such as the Common Base Event model is just a starting point for making your application problem determination friendly. More importantly than following the standard is carefully planing the content to be included in the event message. For example, even simple things such as date and time format mismatches can render the entire content of the event message useless to the interpretation of its receiver when correlating it with several other events.

The current release of the Autonomic Computing Toolkit uses the Common Base Event schema Version 1.0.1. The schema is shipped with the Generic Log Adapter and can be found in the directory <GLA>\dev\eclipse\plugins\org.eclipse.hyades.logging.adapter.ui_1.2.0\commonbaseevent1_0_1.xsd, where <GLA> is the installation directory of the Generic Log Adapter. A newer schema, Version 2.0, was submitted to the OASIS Web Services Distributed Management Technical Committee to go through the standardization process.

This schema describes the structure of the generated Common Base Event. It also defines the mandatory fields and how many times each entry can appear in the Common Base Event. The following section provides an introduction to the Common Base Event structure and best practices for defining events.

3.2 Common Base Event structure

To ensure the completeness of the data, any Common Base Event consists of three aspects by providing the following information for each event:

1. The identification of the component that is *reporting* the situation. This is known as the *observing* component.
2. The identification of the component that is *affected* by the situation (could be the same as the observing component). This is the *impacted* component.
3. The *situation* itself.

All properties defined in the Common Base Event model apply to the categories described earlier. These categories are referred to as the *3-tuple* structure, as depicted in Figure 3-1 on page 38.

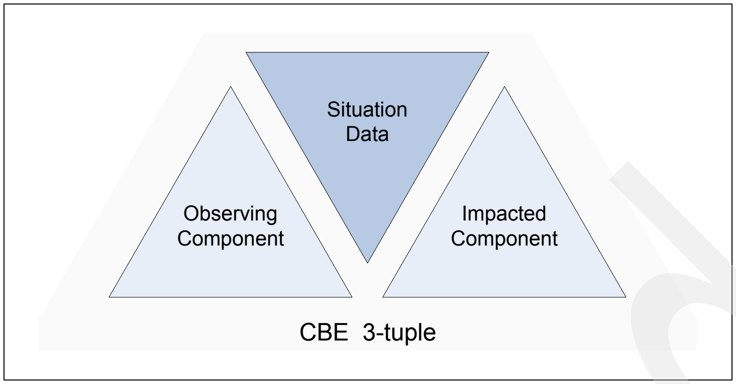


Figure 3-1 The Common Base Event 3-tuple structure

3.3 Common Base Event properties

Common Base Event (CBE) properties represent the data that can be collected for the CBE 3-tuple previously described. For each of the properties of the CBE 3-tuple structure, a series of elements are used as presented in Table 3-1.

Table 3-1 CBE 3-tuple properties

Observing component	Impacted component	Situation data
reporterComponentId[a]	sourceComponentId[b]	version localInstanceId globalInstanceId creationTime[b] severity priority situation[b] contextDataElements msgDataElement msg repeatCount elapsedTime associatedEvents extensionName extendedDataElements sequenceNumber associationEngine

a. Conditionally required element
b. Required element

In the Common Base Event, the sourceComponentId element entry refers to the component that is experiencing the situation (affected by the situation) and is a required field. The reporterComponentId element entry refers to the component that reports the situation. In the case where the reporting and affected component are the same (reporterComponentId and the sourceComponentId are the same), the reporterComponentId becomes optional; otherwise both of them are required entries in the Common Base Event.

In the Common Base Event, the *situation element* entry includes the event information in detail. It includes a collection of information and is flexible enough to include extra properties and attributes to accommodate component-specific entries. For component-specific attributes that are not included in the Common Base Event data model, we recommend that you use the ExtendedDataElement element to describe it.

As seen in Table 3-1 on page 38, the set of required elements for a Common Base Event is small to allow for a wide variety of system events to be represented in a compact manner.

The required CBE elements are:

- reporterComponentId
- sourceComponentId
- creationTime
- situation

However, certain applications that use Common Base Events might require optional elements to be included. For example, Common Base Events used for problem determination typically need some of the optional elements to be completed so that they can be used effectively during root cause analysis.

The optional CBE elements are:

- version
- localInstanceId
- globalInstanceId
- severity
- priority
- msg
- extensionName
- repeatCount
- elapsedTime
- sequenceNumber
- contextDataElement
- extendedDataElement
- msgDataElement
- associatedEvents

- associationEngine

The following sections go into detail about each of the CBE required and optional elements listed and provide best practices information when defining them.

3.4 Common Base Event required elements

At the heart of every Common Base Event are several elements that must always be present. This section describes the best practices for the required CBE elements.

3.4.1 The sourceComponentId and reporterComponentId elements

The sourceComponentId element identifies the component that was affected by the event or situation. The reporterComponentId element identifies the component that reported the event or situation on behalf of the affected component. Both sourceComponentId and reporterComponentId are described by the *ComponentIdentification* element type, and therefore, both are of the type ComponentIdentification.

The sourceComponentId element is required, and after a value is set, it must not be changed. The event producer must provide the sourceComponentId element. If the reporter component and the affected component are the same, the reporterComponentId element must not be present. Therefore, the reporterComponentId element is required only if the reporting component is not the same as the source component. After a value is set for the reporterComponentId element, it must not be changed.

A component generating a Common Base Event about itself specifies its ComponentIdentification information for sourceComponentId. A component that generates a Common Base Event on behalf of another component specifies its own ComponentIdentification information for reporterComponentId; for sourceComponentId, the reporter specifies the ComponentIdentification information for the component on whose behalf it is generating the Common Base Event.

For example, if a hosting environment detects a situation in one of the resources that it hosts, the hosting environment should generate a Common Base Event, specifying its own information for reporterComponentId and the information about the managed resource that it hosts for sourceComponentId.

The ComponentIdentification element type

The intent of the ComponentIdentification element type is to describe and identify an instance of a component at run time. It provides detailed information about the components that are identified by the sourceComponentId and reporterComponentId elements using a collection of properties such as:

- location

  This property specifies the physical address corresponding to the location of a component, such as IP address, host name, and MAC address. Examples are:

Fully qualified host name	`myserver.ibm.com`
IPv4 address	`10.0.0.1`
MAC address	`09:01:00:03:71:CE`

  The format of the value of location is determined by the locationType property. The best practice for a location address is to use the fully qualified host name.

  This is a required property.

- locationType

  This property provides details about the format and meaning of the value in the location property. Valid values include IPv4, IPv6, host name, and IPX. For a complete list, refer to the *Autonomic Computing Toolkit Developer's Guide*, SC30-4083.

  This is a required property.

- application

  This property defines the human-readable common name of an application. The value for this property is usually the name of a business application that is the business logic and business data used to address a set of specific business requirements. A business application typically consists of several components of multiple types combined in an unique manner to provide the functions and resources needed to address specific requirements. The version of the application can also be incorporated into the application name, separated by a # sign. We recommend that the vendor name be prepended to the application name, for example, `IBM HR Appl#4.3.1`.

  This is an optional property.

- executionEnvironment

  The executionEnvironment property specifies the hosting environment of the application that is specified in the application property. It identifies the immediate environment within which the application is operating.

  For example, a Java 2 Platform Enterprise Edition (J2EE) application might be hosted by an IBM WebSphere Application Server named `BaseApplicationServerCell\nodeABC\server1` that specifies the name, cell, node, and server name. An application might be directly hosted by an operating system named `RedHatLinux`.

  The executionEnvironment version information optionally can be appended to the end of the component separated by a # character, for example, `BaseApplicationServerCell\mynode\server1#5.1`.

  This is an optional property.

- component

  The component property specifies the logical identity of a component. This property must contain the name of a particular component of an application, product, or subsystem, for example, `IBM WebSphere Application Server Platform#5.0`. This value should be unique within the scope specified by the reporter location property. The type of the information specified in component is described in the componentIdType property, described later.

  This is a required property.

- subComponent

  The subComponent property specifies a further distinction for the logical component property of the event. It should contain the identity of the subcomponent of the component property and it should be the most granular definition specified in the event, for example, a module name, a class name, a class, and a method.

  This is a required property.

- componentIdType

  The componentIdType property specifies the format and meaning of the component property. Valid values include:

  – ProductName
  – DeviceName
  – SystemName
  – ServiceName
  – Process
  – Application
  – Unknown

For a complete list, refer to the *Autonomic Computing Toolkit Developer's Guide*, SC30-4083.

This is a required property.

> **Tip:** The component, subcomponent, and componentIdType properties form a logical group. The interrelationships among these properties should be taken into account by producers and consumers of Common Base Events.

- instanceId

  The instanceId property specifies an identifier for the instance of the component that is specified in the component property. This is useful when more than one occurrence of a component is running in a system. This property is intended as an identifier for use with correlation so that management components can ensure that the information contained in the Common Base Event pertains to the same instance of a component. For example, the instanceId property can take the form of a `Java EJBHandle`.

  This is an optional property.

- processId

  The processId property is a string that specifies the process identifier of the component or subcomponent that generated the event. The value is platform specific. The threadId property can be used in conjunction with processId to further delineate the component or subcomponent.

  This is an optional property.

- threadId

  The threadId property is a string that specifies the thread identifier of the component or subcomponent that is specified by the processId property. Both threadId and processId can be obtained from the hosting environment or platform.

  This is an optional property.

- componentType

  The componentType property characterizes all instances of a given kind of component. A componentType is defined as an XML QName that contains a namespace qualified name, for example:

  ```
  http://www.ibm.com/xmlns/prod/autonomic/J2EE_RT/IBMWebSphereApplicationServer
  ```

  This example could be shortened using standard XML namespace declaration when Common Base Event instances are created.

This property is not meant to be used for unique identification of the components. Instead, it identifies the type of component (or resource) that can be used as an additional qualifier to identify and establish a scope for the instance name of the component. As a general rule, componentType should specify the most granular identifier for the component that produces the Common Base Event.

This is a required property.

3.4.2 The creationTime element

This element specifies the time at which this Common Base Event was created. It uses a type called dateTime as specified in XML schema. The lexical representation for this element is as follows:

```
CCYY-MM-DDThh:mm:ss
```

Where `CC` represents the century, `YY` the year, `MM` the month, and `DD` the day. The letter `T` is the date and time separator and `hh`, `mm`, and `ss` represent hour, minute, and second, respectively.

An example value for this field, indicating June 28, 2004 at 3:32 Eastern Standard Time which is 5 hours behind UTC, is `2004-06-28T03:32:00-05:00`. Additional indicators can be present in a dateTime type, including Z to indicate the current time zone, century coding, and so on.

For details, refer to the XML schema at:

http://www.w3.org/TR/2001/REC-xmlschema-0-20010502

3.4.3 The situation element

When an event is generated, the 3-tuple format of the Common Base Event data must be reported: the observing component, the impacted component, and situation data. The situation element is one of the most important parts of a Common Base Event. This element describes the category of event that was detected, providing autonomic managers with information to perform appropriate actions.

As described earlier in this chapter, a situation represents a categorization of a state change into predefined categories. The lack of standards regarding both the format and content of problem logs makes it difficult to write software that automates problem resolution in large, complex systems. For example, in order to write a simple management function such as `If any component stops, then start it`, a consistent way of identifying the component is needed. This issue is resolved by the conversion of messages into a finite set of categories to adhere to a common log format.

The situation element is represented by the following required properties:

- categoryName
- situationType

The categoryName property

The Common Base Event model defines a set of categories describing the type of situation. This property categorizes the type of the situation that caused the event to be reported and can be set to one of the following values:

- startSituation
- stopSituation
- connectSituation
- configureSituation
- requestSituation
- featureSituation
- dependencySituation
- createSituation
- destroySituation
- reportSituation
- availableSituation
- otherSituation

We explain these situation values in detail in the following section.

The situationType property

The situationType property further describes the situation by providing additional details about the categorization of the event. The situationType property is an abstract element that is used to specify all of the supported situation types defined by the categoryName property.

Every situationType property definition contains a *reasoningScope* subproperty definition. The reasoningScope defines whether the impact of this situation is *internal* or *external*. Valid values for this subproperty are INTERNAL or EXTERNAL. If reasoningScope is defined as INTERNAL, the impact of the situation is contained within the component that is originating the situation. If reasoningScope is defined as EXTERNAL, the impact of the situation might extend outside the component that is originating the situation.

In addition to the reasoningScope subproperty, for every situation we create, there are some required subelements. These required subelements change from one situation to another. For example, if situationType is defined as a startSituation, it requires successDisposition and situationQualifier, while if situationType is defined as a reportSituation it requires reportCategory.

For each of the categoryName properties previously defined, the following items represent the best practices for defining the respective situationType subproperties.

startSituation

The startSituation describes the start process for a component. The startSituation includes the following properties:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCESSFUL. This is a required property for a start situation.

- situationQualifier

  This further describes the start situation. The situationQualifier values for startSituation are START INITIATED, RESTART INITIATED, or START COMPLETED. This is a required property for a start situation.

Messages that indicate that a component has begun the startup process, that it has finished the startup process, or that it has aborted the startup process all fall into this category. Existing messages typically include words such as starting, started, initializing, and initialized, for example:

```
DIA3206I The TCP/IP protocol support was started successfully.
DIA3000I "%1S" protocol support was successfully started.
DIA3001E "%1S" protocol support was not successfully started.
WSVR0037I: Starting EJB jar: {0}
```

A single Common Base Event can represent an entire start situation, for example, a start situation that includes the following information to indicate that the entire startup process has successfully completed:

```
SituationQualifier = START COMPLETED , successDisposition = SUCCESSFUL
```

A start situation also could be specified with finer granularity, using multiple Common Base events, for example, several start situations to indicate that the first attempt at starting was unsuccessful, so a restart was attempted, which was successful, followed by an indication that startup process has successfully completed, such as:

```
SituationQualifier = START INITIATED , successDisposition = UNSUCCESSFUL
SituationQualifier = RESTART INITIATED , successDisposition = SUCCESSFUL
SituationQualifier = START COMPLETED , successDisposition = SUCCESSFUL
```

Example 3-1 on page 47 shows the relevant parts of a Common Base Event with a startSituation.

Example 3-1 startSituation CBE

```
<CommonBaseEvent creationTime="2005-01-25T11:19:49.766000000Z"
                 extensionName="CommonBaseEvent"
                 globalInstanceId="CE82BBB98E65751F5ABCB6206EF511D9"
                 version="1.0.1">

.......................

   <sourceComponentId application="PDWebApp" component="IBM WebSphere
Application Server Platform 5.0 [embeddedEXPRESS 5.0.2 ptf2M0325.01]"
                      componentIdType="APPLICATION"
                      executionEnvironment="DefaultNode\DefaultNode\server1"
                      location="192.168.192.168" locationType="IPV4"
                      processId="2052"
                      subComponent="ApplicationServer:com.ibm.ws.runtime.compon
ent.ApplicationMgrImpl" threadId="2839eb19"
                     componentType="WebSphereApplicationServer"/>

.......................

   <situation categoryName="StartSituation">
      <situationType xsi:type="StartSituation"
                     reasoningScope="INTERNAL"
                     successDisposition="SUCCESSFUL"
                     situationQualifier="START INITIATED"/>
   </situation>
</CommonBaseEvent>
```

stopSituation

The stopSituation describes the stop process for a component. The stopSituation includes the following properties:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCESSFUL. This is a required property for a stop situation.

- situationQualifier

  This further describes the stop situation. The situationQualifier values for stopSituation are STOP INITIATED, ABORT INITIATED, PAUSE INITIATED, or STOP COMPLETED. This is a required property for a stop situation.

Messages that indicate that a component has begun to stop, that it has stopped, or that the stopping process has failed all fall into this category. Existing messages typically include words such as stop, stopping, stopped, completed, and exiting, for example:

```
WSVR0220I: Application stopped: {0}
WSVR0102E: An error occurred stopping, {0}
MSGS0657I: Stopping the MQJD JMS Provider
```

In a manner similar to the start situation, a single Common Base Event can represent an entire stop situation or a stop situation also could be specified with finer granularity using multiple Common Base events.

connectSituation

The connectSituation describes situations that identify aspects about a connection to another component. The connectSituation includes the following properties:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCCESSFUL. This is a required property for a connect situation.

- situationDisposition

  This further describes the connect situation. The situationDisposition values for connectSituation are INUSE, FREED, CLOSED, or AVAILABLE. This is a required property for a connect situation

Messages that indicate that a connection failed, that a connection was created, or that a connection has ended all fall into this category. Existing messages typically include words such as connection reset, connection failed, and failed to get a connection, for example:

```
DBMN0015W: Failure while creating connection {0}
DBMN0033W: connection close failure {0}
DBMN0023W: Failed to close a connection {0}
```

In a manner similar to other situations, a single Common Base Event can represent an entire connect situation or a connect situation also could be specified with finer granularity using multiple Common Base events.

Example 3-2 on page 49 shows the relevant parts of a Common Base Event with a connectSituation.

Example 3-2 connectSituation CBE

```
<CommonBaseEvent creationTime="2005-01-25T11:27:44.141000000Z"
                 extensionName="CommonBaseEvent"
                 globalInstanceId="CE82BBB98E65751F750C97B06EF611D9"
                 severity="50" version="1.0.1">

...............

   <sourceComponentId application="UNKNOWN" component="IBM WebSphere
Application Server Platform 5.0 [embeddedEXPRESS 5.0.2 ptf2M0325.01]"
componentIdType="APPLICATION"
                      executionEnvironment="DefaultNode\DefaultNode\server1"
                      location="192.168.192.168" locationType="IPV4"
                      processId="2052" subComponent="Application
Server:com.ibm.ws.webcontainer.srt.WebGroup" threadId="4d8b6b04"
                      componentType="WebSphereApplicationServer"/>
   <msgDataElement>
      <msgCatalogTokens value="SRVE0026E: [Servlet Error]-[WorkWithDB]:
javax.servlet.ServletException: com.ibm.db2.jcc.c.DisconnectException: A
communication error has been detected. Communication protocol being used: {0}.
Communication API being used: {1}. Location where the error was detected: {2}.
Communication function detecting the error"/>
      <msgId>SRVE0026E</msgId>
      <msgIdType>IBM4.4.1</msgIdType>
   </msgDataElement>

...............


   <situation categoryName="ConnectSituation">
      <situationType xsi:type="ConnectSituation"
                     reasoningScope="INTERNAL"
                     successDisposition="UNSUCESSFUL"
                     situationDisposition="CLOSED"/>
   </situation>
</CommonBaseEvent>
```

configureSituation

The configureSituation describes situations in which components identify information about their configuration data and changes to that data. The configureSituation includes the following property:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCESSFUL. This is a required property for a configure situation

Any changes made to a component's configuration data should be represented with this category. Additionally, messages that describe the current configuration state fall into this category. Because many different names and types exist for configuration properties, some analysis is required to determine if existing messages fall into this category. Existing messages might include words such as configure, configured, configuration, or set to generally in combination with port number, name, address, directory, path, and so on. For example:

```
ADFS0134I: File transfer is configured with host="9.27.11.13", port="9090",
securityEnabled="false".
```

Configure situations could be reported in conjunction with other situations; for example, a resource might report its current configuration state along with a start situation. Configure situations also should be reported when configuration state data changes.

requestSituation

The requestSituation describes situations in which a component identifies the completion status of a request. The requestSituation includes the following properties:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCESSFUL. This is a required property for a request situation

- situationQualifier

  This property further describes the request situation. The situationQualifier values for requestSituation are REQUEST INITIATED or REQUEST COMPLETED. This is a required property for a request situation.

Request situations typically relate to complex management tasks or transactions that a component undertakes on behalf of a requestor, rather than the detailed processing associated with carrying out the transaction. Messages that indicate that a request was made to perform a transaction or operation or that such a request has completed fall into this category. Existing messages typically include words such as configuration synchronization started or backup procedure complete, for example:

```
ADMS0003I: Configuration synchronization completed
```

featureSituation

The featureSituation describes situations that announce that a feature of a component is now installed and present for service requests. The featureSituation includes the following property:

- featureDisposition

  The featureDisposition values are AVAILABLE or NOT AVAILABLE. This is a required property for a feature situation, and after a value is set, it must not change. The string length for this property must not exceed 64 characters

The feature situation applies to the presence of individual features of a component. The operations and operational state of the component itself should be reported using the available situation.

Example 3-3 shows the relevant parts of the Common Base Event with the featureSituation.

Example 3-3 featureSituation CBE

```
<CommonBaseEvent creationTime="2005-01-25T11:19:50.984000000Z"
                 extensionName="CommonBaseEvent"
                 globalInstanceId="CE82BBB98E65751F5B6D1A606EF511D9"
                 version="1.0.1">

.....

   <sourceComponentId application="UNKNOWN"
      component="IBM WebSphere Application Server Platform 5.0
[embeddedEXPRESS 5.0.2 ptf2M0325.01]"
      componentIdType="APPLICATION"
      executionEnvironment="DefaultNode\DefaultNode\server1"
      location="192.168.192.168" locationType="IPV4" processId="2052"
      subComponent="ApplicationServer:com.ibm.ws.webcontainer.http.HttpTranspor
t"
      threadId="2839eb19" componentType="WebSphereApplicationServer"/>

.............

   <situation categoryName="FeatureSituation">
      <situationType xsi:type="FeatureSituation"
                     reasoningScope="INTERNAL"
                     featureDisposition="AVAILABLE"/>
   </situation>
</CommonBaseEvent>
```

createSituation

The createSituation describes situations in which a component creates some entity. The createSituation includes the following property:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCESSFUL. This is a required property for a create situation.

Messages that indicate that a document, file, Enterprise JavaBeans, component instance, or other entity was created all fall into this category. Existing messages typically include words such as create, created, and now exists, for example:

```
ADMR0009I: Document cells/flatfootNetwork/applications/Dynamic Cache
Monitor.ear/Dynamic Cache Monitor.ear was created
```

destroySituation

The destroySituation describes situations in which an entity was removed or destroyed. The destroySituation includes the following property:

- successDisposition

  The successDisposition values are SUCCESSFUL or UNSUCESSFUL. This is a required property for a destroy situation.

Messages that indicate that a document, file, Enterprise JavaBeans, component instance, or other entity was destroyed all fall into this category. Existing messages typically include words such as destroy, destroyed, deleted, and no longer exists, for example:

```
CONM6007I: The connection pool was destroyed for data source
(UDDI.Datasource.techs8.server1).
```

reportSituation

The reportSituation describes situations in which the component is conveying general information, such as the status, heartbeat, or performance information. The reportSituation includes the following property:

- reportCategory

  The reportCategory values are PERFORMANCE, SECURITY, HEARTBEAT, STATUS, TRACE, DEBUG, or LOG. This is a required property for a report situation.

Metric data, such as current CPU utilization and current memory heap size, falls into this category. In addition, supplemental error information, such as trace or debug data, as well as general status information that is not related to another defined situation, can be conveyed in a report situation. Existing messages typically include words such as utilization, rate (or rate indications such as values per unit), buffer size, number of processes, and number of threads, for example:

```
IEE890I WTO Buffers in console backup storage = 1024
```

However, not all such indicators necessarily involve metric data, so not all seemingly similar messages are metric situations. Consider, for example, configuration properties, which are distinct from metric properties. Configuration property state changes should be reported with a configure situation (described earlier), rather than with a report situation. In addition, existing messages might contain words such as debug or trace.

Care should be taken so that the report situation is not treated as a default or catch-all category. In some respects, all Common Base Events can be considered to be reporting something. However, the report situation is intended to address specific types of data reporting, such as metric, trace and debug data. See Example 3-4.

Example 3-4 reportSituation CBE

```
<CommonBaseEvent creationTime="2004-02-12T10:19:53Z"
                 globalInstanceId="DEB3EB119D9474DC99360C7464D811D9"
                 msg="[Thu Feb 12 10:19:53 CST 2004] ITSO434W Resource
utilization above 80%"
                 severity="30" version="1.0.1">
   <sourceComponentId application="ITSOSimpleApp1"
                      component="ITSO Simple App#1" componentIdType="Name"
                      location="server1.itso.ibm.com" locationType="IPV4"
                      subComponent="ITSOSubComponent"/>
   <msgDataElement>
      <msgId>ITSO434W</msgId>
   </msgDataElement>
<situation categoryName="ReportSituation">
   <situationType xsi:type="ReportSituation"
                  reasoningScope="INTERNAL"
                  reportCategory="STATUS"/>
   </situation>
</CommonBaseEvent>
```

availableSituation

The availableSituation describes a component's operational state and availability. The availableSituation includes the following properties:

- operationDisposition

  The operationDisposition values are STARTABLE or NONSTARTABLE. This is a required property for an available situation.

- availabilityDisposition

  The availabilityDisposition values are AVAILABLE or NOT AVAILABLE. This is a required property for an available situation.

- processingDisposition

  The processingDisposition values are:

FUNCTION_PROCES	Indicates that a functional operation was processed.
FUNCTION_BLOCK	Indicates that a functional operation was blocked (not processed).
MGMTTASK_PROCESS	Indicates that a management operation was processed.
MGMTTASK_BLOCKED	Indicates that a management operation was blocked (not processed).

  This is a required property for an available situation and provides a context for operations that can be performed on the component by establishing whether a product is installed, operational, and ready to process functional requests, or operational and ready (or not ready) to process management requests. Existing messages typically include words such as available, unavailable, ready, online, and offline, for example:

```
ADMC0013I: SOAP connector available at port 8880
ADMC0026I: RMI Connector available at port 2809
```

An available situation might be generated when a component's availability state changes, or it could be generated as a result of some other action. For example, a component might generate an available situation that indicates NONSTARTABLE as a result of a management request to start the component.

dependencySituation

The dependencySituation describes a dependency relationship of one component to another. This can include situations in which a component indicates that it cannot find some other component or capability on which it depends. The dependencySituation includes the following property:

- dependencyDisposition

  The dependencyDisposition values are MET or NOT MET. This is a required property for a dependency situation

This category includes messages that indicate that some dependency was met or not met. Such messages could take the form of indications that a required component, file, resource, capability, or feature is not available or was found (or not found) or the version of a component matches (or does not match) what was expected. Existing messages typically include words such as dependency, find, found, required, no such, or mismatch, for example:

```
WSVR0017E: Error encountered binding the J2EE resource, Pet Store JMS Queue
Connection Factory, as jms/queue/QueueConnectionFactory from resources.xml no
resource binder found
```

Dependency situations are distinguished from available and feature situations by the existence of a dependency relationship between the component and some other resource. A component might use an available situation to indicate that one of its own features has become available or unavailable, while a component should report the availability or unavailability of a feature that it depends on using a dependency situation.

otherSituation

The otherSituation category describes situations that cannot be represented in any of the defined situation categories. This category has no additional properties.

The defined situation categories are intended to provide the ability to express most situations that are encountered by a managed resource that required a Common Base Event to be generated.

There might be cases in which a situation does not absolutely fall into one of the defined categories, requiring otherSituation to be specified. However, otherSituation generally is not useful to autonomic managers that need to analyze Common Base Events and perform operations on the managed resource based on those events, because otherSituation does not indicate a specific, analyzable, actionable situation. Therefore, the use of the otherSituation category is not encouraged.

3.5 Common Base Event optional elements

Now that we have addressed the required elements of a Common Base Event, we turn our attention to the optional elements. These optional elements can be just as important as the required elements in some applications.

The following sections describe the optional CBE elements.

3.5.1 The version element

The version element is a string that is used to identify the version of the Common Base Event specification to which the event conforms so that the event can be recognized and managed appropriately by event consumers. The version element is intended to facilitate compatibility and migration as the Common Base Event specification evolves. This field is optional.

This property can be used to assist receivers with parsing and migration of Common Base Event V1.0 data that did not include version, by providing a marker for backward-compatibility-processing procedures.

This version element *must* stay consistent with the version specified in the Common Base Event schema header. The reason that version is repeated here is to allow Common Base Event XML fragments, when it might not be desired to include the full XML document headers, to exchange the events.

If the version element is not specified, it is assumed to be 1.0. However, because the current version of the Common Base Event specification is 1.0.1, this element should be treated as though it were required, and its value should be set to 1.0.1.

3.5.2 The localInstanceId element

The localInstanceId element is a string that is used to locally identify instances of an event. There is no implied guarantee that this value is globally unique; it needs to be unique only within the scope of the execution process that generates the event. The localInstanceId element can be assigned by the component that generates the event or by the consumer of the event. The value of localInstanceId might be a multipart value, such a time stamp, location, offset, or message identifier, and might use other application-defined techniques to ensure the uniqueness of the values. For example, the value might be set to the string concatenation of the local host IP address, the absolute path of the access.log file, the local fully qualified host name, a time stamp, and the sequenceNumber element (described later), as follows:

```
10.10.3.56mycomputer.lab.ibm.com2002100902534.002000-240
```

This is an optional element, and after it is set, it must not be changed.

3.5.3 The globalInstanceId element

The globalInstanceId element is a complex data type that represents the primary identifier for the event. This element uniquely identifies the event; therefore, it could be used as the primary key for the event. The globalInstanceId element can be assigned by the component that generates the event or by the consumer of the event.

The value must be a globally unique identifier (GUID) that is at least 128 bits, but no more than 256 bits long, and it must begin with an alphabetic character (that is, A-Z). The GUID generation algorithm must ensure the uniqueness of this value.

One method for constructing a GUID is defined in the Internet draft *draft-leach-uuids-guids-01*. However, this method does not generate a GUID that begins with an alphabetic character, so if this method is used, the result must be prepended with a single alphabetic character.

This is an optional element, and after it is set, it must not be changed for the lifetime of the event. The globalInstanceId element is required if associations among events are to be established. If globalInstanceId is not specified, the association specified by associatedEvent (described later) cannot be used.

3.5.4 The severity element

The severity element is used to indicate the severity level of the event from the point of view of the component that reports the event. This element is intended to define the significance or gravity of the situation that was encountered so that administrators can focus on the most severe problems.

A component's domain expert typically provides the severity values. The meanings of the values that this element can contain can be described by an enumeration of common values or qualifiers that indicate the severity level of the event. For example, an enumeration of values, such as information or warning, might be used, or a set of integers that map to the intended severity levels might be used. The Common Base Event specification defines these severity values:

0 Unknown	
10 Information	This value indicates that the event contains only general information and is not reporting an error.
20 Harmless	This value indicates that the error has no effect on the normal operation of the resource.
30 Warning	This value indicates that it is appropriate to let the user decide if an action is needed in response to the event.
40 Minor	This value indicates that action is needed, but the situation is not serious at this time.
50 Critical	This value indicates that an immediate action is needed and the scope is broad (perhaps an imminent outage to a critical resource will result).
60 Fatal	This value indicates that an error occurred, but it is too late to take remedial action.

This is an optional element, and after it is set, it must not be changed. The values are 0 to 70. The reserved values start at 0 for Unknown and increase by increments of 10 to 60 for Fatal. Other severity values can be added but must not exceed 70. If no value is specified, this event is interpreted as having no severity (there is no default value for severity).

This element is provided for compatibility with management functions that require events to have a severity level. These values are usually provided by a component's domain expert, and it is set by the originator of the event. The meanings of the values that this element can contain can be described by an enumeration of common values or qualifiers that indicate the severity level of the event. For example, information, warnings, or a set of integers that map to the intended severity levels are all valid values. This document does not imply any specific implementation, but instead suggests the following values based on prior experiences with the understanding that users of this field can add additional implementation-specific values. This field is intended to define the seriousness of the kind of situation that was encountered so that administrators can focus on the most severe problems

3.5.5 The priority element

The priority element defines the importance of the event so that an event consumer can establish a relative order in which the event records should be processed.

The priority element is independent of severity, because priority is intended to be used primarily by event consumers, while severity indicates the significance of the situation as perceived by the affected component. For example, an event with a high priority but a low severity typically should be processed before an event with a low priority but a high severity.

The Common Base Event specification defines these priority values:

10 Low	For an event that does not need to be processed immediately
50 Medium	For an event of average importance
70 High	For an important event that requires immediate attention

This is an optional element and it can be changed. If no value is specified, this event is interpreted as having no priority (there is no default value for priority). The values are 0 to 100. The reserved value for Low is 10, for Medium is 50, and for High is 70. Other priority values can be added but must not exceed 100.

3.5.6 The msg element

The msg element is the text that accompanies the event. Because the message is intended to provide information to the end users of an application, message text is typically translated into different national languages and displayed according to a selected locale. The locale of the msg element is specified by the msgLocale element of the MsgDataElement element (described later).

This element is optional, but if the msgCatalogId and msgCatalog properties of the MsgDataElement do not specify any values, we recommend that the msg element have a value. The string length for the msg element must not exceed 1024 characters.

3.5.7 The extensionName element

The extensionName element contains the name of an event class that this event represents (for example, Trace or CommonBaseEvent). The event class name is indicative of any additional properties that are expected to be present within this particular event. Only the component that reports the event can specify the extensionName element. Some applications making use of Common Base Events, for example, for problem determination, might define well-known event classes that are used with extensionName.

This is an optional element, although if the ExtendedDataElement element (described later) is used, we recommend that the extensionName element have a value. After this value is set, it must not be changed. If the value is null, the extensionName element is assumed to indicate the *CommonBaseEvent* event class.

3.5.8 The repeatCount element

The repeatCount element specifies the number of occurrences of identical events within a specified time interval. The time interval is specified by the elapsedTime element, described next. The definition of identical events is application specific, and repeatCount can be assigned by the component that generates the event or by the consumer of the event. The use of repeatCount and elapsedTime allows a reduction in the number of Common Base Events that are generated. When a number of identical events occur within the specified time interval, a single Common Base Event can be generated to indicate that multiple identical events have occurred.

This element is optional, and it can be changed. There is no default value. A value of 0 or no value indicates no repeated occurrences of the event.

3.5.9 The elapsedTime element

The elapsedTime element indicates the time interval during which some number of identical events occurred. The number of occurrences is specified by the value of the repeatCount element, just described. The elapsedTime value indicates the duration of time within which the repeated events were observed. The units for elapsedTime are microseconds. The elapsedTime element can be assigned by the component that generates the event or by the consumer of the event, but it must be assigned by the same component that assigns the repeatCount element.

This element is optional, and it can be changed. However, if the repeatCount element has a value, the elapsedTime element must also have a value. There is no default value for elapsedTime.

3.5.10 The sequenceNumber element

The sequenceNumber element is a source-defined number that enables multiple messages to be sent and processed in a logical order that might be different from the order in which they arrived at the consumer's location (for example, with event servers or management tools). The sequence number helps consumers to sort messages when they arrive.

This element is optional, and after a value is set, it must not be changed.

3.5.11 The contextDataElement element

The contextDataElement element defines the context or contexts to which this event refers. This element contains data that is used to assist with problem diagnostics by correlating events that are generated during the execution of a unit of work.

The following properties make up the contextDataElement:

- type
- name
- contextValue
- contextId

The type property

This type property specifies the data type of this contextDataElement, specifically its contextValue property (described later). This type should enable the consumer of the event to recognize the format of the contextValue property. The type is application specific, for example, `PD_LogRecordCorrelator`.

This property is required, and after a value is set, it must not be changed. The string length for this property must not exceed 64 characters.

The name property

This property specifies the name of the application that created this contextDataElement, for example, `Correlation engine`.

This property is required, and after a value is set, it must not be changed. The string length for this property must not exceed 64 characters.

The contextValue property

This is the value of the context with respect to the implementation of the context.

This property and the contextId property (described next) are mutually exclusive; contextValue is required unless contextId specifies a value. Otherwise (if contextId does specify a value), contextValue should not be specified. However, if both properties have a value, contextId is ignored. After a value is set, it must not be changed. The string length for this property must not exceed 1024 characters.

The contextId property

This property is the reference to an element that contains a product-specific context. The value must be a globally unique identifier (GUID) that is at least 128 bits, but no more than 256 bits long, and it must begin with an alphabetic character (that is, A-Z). The GUID generation algorithm must ensure the uniqueness of this value.

This property and the contextValue property are mutually exclusive; contextId is required unless contextValue specifies a value. Otherwise (if contextValue does specify a value), contextId must not be specified. However, if both properties have a value, contextId is ignored. After a value is set, it must not be changed.

3.5.12 The ExtendedDataElement element

The base information included in a Common Base Event might not be sufficient to represent all of the information captured by a component when creating a Common Base Event. The Common Base Event provides several methods for including this additional data, including extending the Common Base Event schema or supplying one or more ExtendedDataElement elements within the Common Base Event. An ExtendedDataElement element is used to represent a single data item, and a Common Base Event can contain more than one of these elements (essentially one for each additional data item).

The properties that make up the ExtendedDataElement element are described next.

The name property

The name property provides a name used to identify and qualify the data contained in the ExtendedDataElement element. The name uniquely identifies a particular type or data element, but that data element can be included multiple times in a single Common Base Event. This property is required within the ExtendedDataElement element.

The type property

The type property specifies the data type of the values associated with the ExtendedDataElement element, specifically the format of the values or hexValue property (described later). The Common Base Event specification describes the following types:

- byte, short, int, long, float, double
- string, dateTime, boolean
- byteArray, shortArray, intArray, longArray, floatArray, doubleArray
- stringArray, dateTimeArray, booleanArray, hexBinary
- noValue

These data types are the only valid data types for the ExtendedDataElement element. The default value is string. The type noValue is a reserved string to identify cases where an ExtendedDataElement element contains only children and no data. This property is required within the ExtendedDataElement element.

The values property

This provides the value or values associated with the ExtendedDataElement element. The value can be a single value or a list of values (represented as an array). The data type of the value is defined by the type property. This property and the hexValue property (described next) are mutually exclusive; but one of the properties must be specified. This value must be provided if the value for the type property is anything other than hexBinary.

The hexValue property

This provides the value associated with the ExtendedDataElement element when the value type (as defined by the type property) is hexBinary. This property and the values property (previously described) are mutually exclusive; but one of the properties must be specified. This value must be provided if the value for the type property is hexBinary and must not be specified if the data type is any other value.

The children property

The Common Base Event can represent extended data as a hierarchy of related data items, using a tree or structured list of ExtendedDataElement elements. The children property refers to other related ExtendedDataElement elements to specify the structured list of data elements. This property is optional within the ExtendedDataElement element.

3.5.13 The msgDataElement element

The msgDataElement element specifies information associated with the message that is contained in this Common Base Event. The msgDataElement information is provided by the component that generates the event.

Often, the text for a set of messages is stored in one or more message files and is accessed by a key. This mechanism allows the message text to be separate from the program that uses it, making translation into different national languages easier. Many messages also contain one or more substitution parameters that are added at the time the message is rendered. By providing information about the message as specified in the msgDataElement properties (described next), it possible to render the message in a different language at a later time.

This is an optional property, and after a value is set, it must not be changed.

The following properties make up the msgDataElement element:

- msgId
- msgIdType
- msgLocale
- msgCatalogId
- msgCatalogTokens
- msgCatalog
- msgCatalogType

The msgId property

The msgId property specifies the message identifier for the event. It should be provided by the component that generates the event. This identifier should be a unique value represented as a string of alphanumeric or numeric characters. It can be as simple as string of numeric characters identifying a message in a message catalog or a multipart string of alphanumeric characters, for example, `DBT1234E`. The format for msgId is specified by the msgIdType property, described next. Because msgId is not translated, it provides a consistent way of identifying the message for an event across multiple locales.

This is an optional property, and after it is set, it must not be changed. The string length for the msgId property must not exceed 256 characters.

The msgIdType property

The msgIdType property specifies the format of the msgId property. It should be provided by the component that generates the event. The msgIdType property can represent a standard or a well-known convention for message formats. For example, `IBM3.4.1` specifies a message that consists of a 3-part, 8-character string identifier, with 3 alphabetic characters that represent a component, followed by 4 numeric characters, followed by a suffix of 1 alphabetic character.

The reserved keywords defined in the Common Base Event Version 1.0.1 specification are:

- IBM* (* is as just described)
- JMX
- DottedName
- Unknown (when the message format is not specified or does not conform to one of the well-known formats specified here)

This is an optional property; however, if msgId is specified, msgIdType must have a value. After a value is set, it must not be changed. The string length for the msgIdType property must not exceed 32 characters.

The msgLocale property

The msgLocale property specifies the locale for which the message is rendered. Its value is a locale code that conforms to the IETF RFC 1766 specifications. For example, `en-US` is the value for United States English.

This property is optional, and after it is set, it must not be changed. If msgLocale is not specified, the consumer of the event must determine the locale. The string length for the msgLocale property must not exceed 11 characters.

The msgCatalogId property

The msgCatalogId property is the index, or identifier, for a message that is used to resolve the message text from a catalog of messages.

This property is optional; however, the msgCatalogId, msgCatalog, and msgCatalogType are mutually dependent; that is, when any one of these properties has a value, the other two properties also should contain values. After a value is set, it must not be changed. The string length for this property must not exceed 64 characters.

The msgCatalogTokens property

The msgCatalogTokens property consists of an array of string values that hold substitution data that is used to resolve an internationalized message into a fully formatted text. The order of the values is implied by the implicit order of the array elements. The locale of the tokens should be the same as the locale of the message text, defined by the msgLocale property (described earlier).

This property is optional, and after it is set, it must not be changed. If there are no substitution values, this property does not need to be specified. The string length of the msgCatalogTokens property must not exceed 256 characters per token.

The msgCatalog property

The msgCatalog property is the qualified name of the message catalog that contains the translated message specified by the msgCatalogId property.

This property is optional; however, the msgCatalogId, msgCatalog, and msgCatalogType are mutually dependent; that is, when any one of these properties has a value, the other two properties also should contain values. After a value is set, it must not be changed. The string length for the msgCatalog property must not exceed 128 characters.

The msgCatalogType property

The msgCatalogType property specifies the format of the msgCatalog property. The format defines the substitution identifier syntax for the msgCatalogTokens property. The reserved keywords defined in the Common Base Event Version 1.0.1 specification are:

- Java
- XPG

This property is optional; however, the msgCatalogId, msgCatalog, and msgCatalogType are mutually dependent; that is, when any one of these properties has a value, the other two properties also should contain values. Furthermore, if the msgCatalog property contains a value, the msgCatalogType property also must a value. After a value is set, it must not be changed. The string length for the msgCatalogType property must not exceed 32 characters.

3.5.14 The associatedEvent element

The associatedEvent element allows events that are associated with each other to be grouped together and coupled with a corresponding association engine.

The associatedEvent element has the following properties:

- associationEngine
- associationEngineInfo
- resolvedEvents

The associationEngine property

This property identifies the application that generated this event that is to be associated with other events.

This property and the associationEngineInfo property (described next) are mutually exclusive; associationEngine is required unless associationEngineInfo specifies a value. Otherwise (if associationEngine does specify a value), associationEngineInfo must not be specified. After a value is set, it must not be changed.

The associationEngineInfo property

This property refers to an associationEngine element (described later) that identifies the application (association engine) that establishes the association among related events and the type of association.

This property and the associationEngine property are mutually exclusive; associationEngineInfo is required unless associationEngine specifies a value. Otherwise (if associationEngineInfo does specify a value), associationEngine must not be specified. After a value is set, it must not be changed.

The resolvedEvents property

This property is an array of globalInstanceId values that corresponds to those events that are associated with this event.

This is a required property, an array of NMTOKENS with a minimum of one element. The values can be changed. The values are provided by the application that is specified in the name property of the associationEngine element.

3.5.15 The associationEngine element

The associationEngine element identifies the application (association engine) that establishes the association among related events. This element also defines properties that describe the type of association.

The associationEngine element is a stand-alone entity in the XML schema and the associatedEvents created by the application that is identified by the associationEngine refer to it. This will eliminate the need to repeat the same data in every associated event.

The associationEngine element has the following properties:

- name
- type
- id

The name property

This property specifies the name of the application that creates the association, for example, `my association engine name`.

This property is required, and after a value is set, it must not be changed. The string length for this property must not exceed 64 characters.

The type property

This property describes the type of association created by this association engine. The association types defined in the Common Base Event Version 1.0.1 specification are:

Contains	This association type represents events that are contained within a root event.
CausedBy	This association type represents a causality relationship in which an event can refer to the cause of the situation.
Cleared	This association type represents a relationship in which an event refers to another event that can correct the situation or results in the situation becoming irrelevant.
MultiPart	This association type represents a collection of events that, taken together, make up a single event.
Correlated	This association type represents a relationship between a child event and a parent event, based on a correlation algorithm that is specified in the name property (just described).

This property is required, and after a value is set that corresponds to a particular name property value, it must not be changed. The string length for this property must not exceed 64 characters.

The id property

The id property specifies the primary identifier for the element. This property value must be globally unique. The recommend value for this property is a globally unique identifier (GUID) that is at least 128 bits, but no more than 256 bits long, represented as a hex string.

This property is required, and after a value is set, it must not be changed.

3.6 Generating Common Base Events

The Autonomic Computing Toolkit provides two ways to generate Common Base Events. The first is using the Common Base Event APIs provided in the org.eclipse.hyades.logging.event.cbe package. This Java package is located in the hlcbe101.jar file and is available from the Eclipse Web site at:

http://www.eclipse.org/tptp/index.html

It is also shipped with the Autonomic Computing Toolkit as part of the Generic Log Adapter component.

The second way is by using the Generic Log Adapter that is suitable for existing applications that already generate text-based log files.

Generating Common Base Events using APIs

In some cases, we want to generate Common Base Events directly. For example, if we are writing our new application to provide the application's tracing as Common Base Events, we might also want an application to generate Common Base Events and send them directly to a resource manager or log them into a file. To accomplish this, there are APIs provided to generate and configure a Common Base Event object that we can use for each event we need to report (see Figure 3-2).

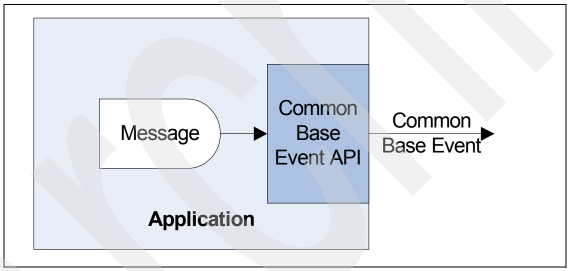


Figure 3-2 Generating Common Base Events directly from the application

The Common Base Event API provides the CommonBaseEvent class that implements the interface called ICommonBaseEvent. The interface ICommonBaseEvent is part of the package org.eclipse.hyades.logging.core. This package is provided by the Eclipse project at:

http://www.eclipse.org/tptp/index.html

It is also provided by the Autonomic Computing Toolkit in the Generic Log Adapter component and can be found in the <GLA_inst_dir>\lib\hlcore.jar file, where <GLA_inst_dir> is the Generic Log Adapter component installation directory.

The process for generating Common Base Events can be summarized by the following actions:

1. Create an event factory.
2. Create an empty Common Base Event.
3. Define the source component.
4. Define the reporter component.
5. Define the situation data.
6. Obtain the creation time.
7. Define additional information for the event as needed.

See 3.7, “Generating Common Base Events using APIs” on page 70 for more details about each of these steps.

Generating Common Base Events using GLA

Currently, most enterprise applications provide log files for tracing the application behavior. The Generic Log Adapter (GLA) provides a way to convert existing log files into the Common Base Event format (see Figure 3-3). The GLA engine takes an adapter file and the log file as inputs and converts the entries in the log file to Common Base Events. The adapter configuration file describes the conversion rules from the log file to the Common Base Event fields. The Autonomic Computing Toolkit provides a plug-in to Eclipse for writing and editing adapter configuration files that specifically generate the output in a Common Base Event format and testing the adapter configuration file before deploying it.

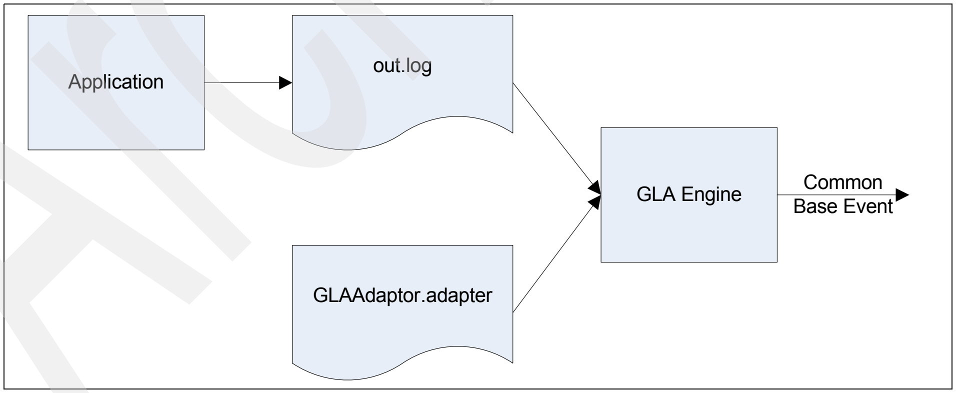


Figure 3-3 Using the Generic Log Adapter to generate Common Base Events

Chapter 4, “Generic Log Adapter” on page 77 describes how to generate Common Base Events using the Generic Log Adapter in more detail.

3.7 Generating Common Base Events using APIs

To make an application generate Common Base Events natively, we can use the Common Base Event Java library provided by Eclipse. There are some examples for generating Common Base Events using this Java library available at:

http://www.eclipse.org/tptp/index.html

In our Java project, we need to include the library files that include the Common Base Event API. These libraries can be located in the following files, which need to be part of the CLASSPATH:

- hlcore.jar
- hlcbe101.jar
- hgla.jar
- ecore.jar
- common.jar

The latest Javadocs and Common Base Event API documentation can be found at the following Web site:

http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/hyades-home/docs/components/common_base_event/cbe101/index.html

3.7.1 Creating the Common Base Event

Using the ICommonBaseEvent interface, we can create a Common Base Event by completing the following steps:

1. Obtain an instance of the Common Base Event factory. For example:
   - Retrieve the singleton instance of the event factory context:

     ```
     EventFactoryContext eventFactoryContext =
     EventFactoryContext.getInstance();
     ```

   - Set the simple event factory home type for all callers:

     ```
     eventFactoryContext.setEventFactoryHomeType("org.eclipse.hyades.logging.
     events.cbe.impl.SimpleEventFactoryHomeImpl");
     ```

   - Retrieve the preconfigured simple event factory home instance from the event factory context:

     ```
     EventFactoryHome eventFactoryHome =
     eventFactoryContext.getEventFactoryHome();
     ```

   - Retrieve an event factory instance from the event factory home:

     ```
     EventFactory eventFactory =
     eventFactoryHome.getEventFactory("com.ibm.itso.sg246665.cbeTutorial");
     ```

2. Create an empty Common Base Event, for example:

```
CommonBaseEvent cbe = eventFactory.createCommonBaseEvent();
```

3. Now, we have an empty Common Base Event. We need to fill it with the needed information.

3.7.2 Filling in the Common Base Event

The objective of this phase is to set the data for the properties represented in the CBE 3-tuple: the component affected by the situation, the component reporting the situation, and the situation itself. Complete the following steps:

1. Define the component affected by the situation. This is done by setting the source component using the IComponentIdentification interface.

   The following sample shows an example implementation:

```
private ComponentIdentification getSourceComponent(
      EventFactory eventFactory, String subComponent) {

   ComponentIdentification sourceComp = eventFactory
         .createComponentIdentification();
   try {
      sourceComp.setLocation(Inet4Address.getLocalHost().getHostName());
   } catch (UnknownHostException e) {
      sourceComp.setLocation("127.0.0.1");
   }
   sourceComp.setLocationType("IPV4");
   sourceComp.setComponent("My Component");
   sourceComp.setSubComponent(subComponent);
   sourceComp.setComponentIdType("Application");
   sourceComp.setComponentType("My Application");

   return sourceComp;
}
```

2. Assign the source component to the Common Base Event, for example:

```
cbe.setSourceComponentId(getSourceComponent(eventFactory,"MyDatabase"));
```

3. Define the component reporting the situation. This is done by setting the reporter component that senses the problem using the IComponentIdentification interface. The following sample shows an example implementation:

```
private ComponentIdentification getReporterComponent(
      EventFactory eventFactory) {

   ComponentIdentification reporterComp = eventFactory
         .createComponentIdentification();
   try {
      reporterComp.setLocation(Inet4Address.getLocalHost().getHostName());
   } catch (UnknownHostException e) {
      reporterComp.setLocation("127.0.0.1");
   }
   reporterComp.setLocationType("IPV4");
   reporterComp.setComponent("My Component");
   reporterComp.setSubComponent("Main Program");
   reporterComp.setComponentIdType("Application");
   reporterComp.setComponentType("My Application");

   return reporterComp;
}
```

4. Assign the reporter component to the Common Base Event, for example:

```
cbe.setReporterComponentId(getReporterComponent(eventFactory));
```

5. Define the situation data. For example, if the event represents a connectSituation, the following properties are required:

   – reasoningScope (required for all the situation types)

   – successDisposition

   – situationDisposition

   The following sample show an example of a connectSituation:

```
Situation situation = eventFactory.createSituation();
ConnectSituation connectSituation = eventFactory.createConnectSituation();
connectSituation.setReasoningScope("EXTERNAL");
connectSituation.setSuccessDisposition("SUCCESSFUL");
connectSituation.setSituationDisposition("CLOSED");
situation.setSituationType(connectSituation);
cbe.setSituation(situation);
```

   The previous connectSituation can be alternatively defined as follows:

```
Situation conSituation = eventFactory.createSituation();
conSituation.setConnectSituation("EXTERNAL","CLOSED","SUCCESSFUL");
cbe.setSituation(conSituation);
```

6. Fill in the creation time, for example:

```
java.text.SimpleDateFormat simpDateFormat = new
   java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ");
cbe.setCreationTime(simpDateFormat.format(new Date()));
```

7. Fill in other optional information, such as the message, severity and so on as needed. For example:

```
// Set the message
cbe.setMsg("All Connections to My Database has been closed");

// Set the severity
cbe.setSeverity((short) 20);
```

8. After the Common Base Event contains the needed information, it is ready to be sent to the assigned manager. The example scenarios presented in this book describe in more detail how to generate Common Base Events natively from the application. Refer to Part 3, "Case study scenarios" on page 257.

Example 3-5 shows the complete listing of the Java class discussed in this section.

Example 3-5 CBE creation Java class

```
/*
 * Created on Feb 1, 2005
 *
 */
package com.ibm.itso.sg246665.cbe;

import java.net.Inet4Address;
import java.net.UnknownHostException;
import java.util.Date;

import org.eclipse.hyades.logging.events.cbe.CommonBaseEvent;
import org.eclipse.hyades.logging.events.cbe.ComponentIdentification;
import org.eclipse.hyades.logging.events.cbe.ConnectSituation;
import org.eclipse.hyades.logging.events.cbe.EventFactory;
import org.eclipse.hyades.logging.events.cbe.EventFactoryHome;
import org.eclipse.hyades.logging.events.cbe.Situation;
import org.eclipse.hyades.logging.events.cbe.impl.EventFactoryContext;

class CBECreationExample {

  public CBECreationExample() {
  }

  public static void main(String args[]) {
    System.out.println("Starting CBE Creation Example");
    System.out.println("Example CBE: "
```

```
        + (new CBECreationExample()).exampleCBE());

  }

  private CommonBaseEvent exampleCBE() {

    // Create an Event factory
    //   - Retrieve the singleton instance of the Event Factory Context:
    EventFactoryContext eventFactoryContext =
EventFactoryContext.getInstance();

    //   - Set the simple Event Factory Home type for all callers:
    eventFactoryContext.setEventFactoryHomeType(

"org.eclipse.hyades.logging.events.cbe.impl.SimpleEventFactoryHomeImpl");

    //   - Retrieve the preconfigured simple Event Factory Home instance from
    //    the Event Factory Context:
    EventFactoryHome eventFactoryHome =
eventFactoryContext.getEventFactoryHome();

    //   - Retrieve an Event Factory instance from the Event Factory Home:
    EventFactory eventFactory = eventFactoryHome.getEventFactory(
       "com.ibm.itso.sg246665.cbeTutorial");


    // Create an empty CBE
    CommonBaseEvent cbe = eventFactory.createCommonBaseEvent();

    // Defines the Source Component
    cbe.setSourceComponentId(getSourceComponent(eventFactory, "MyDatabase"));

    // Defines the Reporting component
    cbe.setReporterComponentId(getReporterComponent(eventFactory));

    // Define the Situation
    Situation situation = eventFactory.createSituation();
    ConnectSituation connectSituation = eventFactory.createConnectSituation();
    connectSituation.setReasoningScope("EXTERNAL");
    connectSituation.setSuccessDisposition("SUCCESSFUL");
    connectSituation.setSituationDisposition("CLOSED");
    situation.setSituationType(connectSituation);
    cbe.setSituation(situation);

    // The situation can alternatively be defined using the following
    Situation conSituation = eventFactory.createSituation();
    conSituation.setConnectSituation("EXTERNAL", "CLOSED", "SUCCESSFUL");
    cbe.setSituation(conSituation);
```

```
    // Sets the Creation time
    java.text.SimpleDateFormat simpDateFormat = new java.text.SimpleDateFormat(
        "yyyy-MM-dd'T'HH:mm:ssZ");
    cbe.setCreationTime(simpDateFormat.format(new Date()));

    // Set the message
    cbe.setMsg("All Connections to My Database has been closed");

    // Set the severity
    cbe.setSeverity((short) 20);

    return cbe;
  }

  /**
   * @param eventFactory
   * @param subComponent
   * @return
   */
  private ComponentIdentification getSourceComponent(
      EventFactory eventFactory,
      String subComponent) {

    ComponentIdentification sourceComp = eventFactory
        .createComponentIdentification();
    try {
      sourceComp.setLocation(Inet4Address.getLocalHost().getHostName());
    } catch (UnknownHostException e) {
      sourceComp.setLocation("127.0.0.1");
    }
    sourceComp.setLocationType("IPV4");
    sourceComp.setComponent("My Component");
    sourceComp.setSubComponent(subComponent);
    sourceComp.setComponentIdType("Application");
    sourceComp.setComponentType("My Application");

    return sourceComp;
  }

  /**
   * @param eventFactory
   * @return
   */
  private ComponentIdentification getReporterComponent(
      EventFactory eventFactory) {

    ComponentIdentification reporterComp = eventFactory
        .createComponentIdentification();
    try {
```

```
      reporterComp.setLocation(Inet4Address.getLocalHost().getHostName());
    } catch (UnknownHostException e) {
      reporterComp.setLocation("127.0.0.1");
    }
    reporterComp.setLocationType("IPV4");
    reporterComp.setComponent("My Component");
    reporterComp.setSubComponent("Main Program");
    reporterComp.setComponentIdType("Application");
    reporterComp.setComponentType("My Application");

    return reporterComp;
  }

}
```

4

Generic Log Adapter

The Generic Log Adapter is a tool provided by the IBM Autonomic Computing Toolkit to convert log messages from a text-based log file format to the Common Base Event format. The generated Common Base Event messages can be consumed by different autonomic managers. In this chapter, we discuss how to convert existing log files into Common Base Events that can be delivered to autonomic managers.

We discuss the following topics in this chapter:

- Generic Log Adapter overview
- Creating a simple adapter
- Creating a multiple contexts adapter
- Deploying and running the adapter configuration file
- Writing a custom outputter

4.1 Generic Log Adapter overview

In order to facilitate problem determination and take advantage of the self-healing capabilities of autonomic computing, existing applications must report their log and trace files using the Common Base Event format. This can be achieved either by:

- Changing the application's log producer components to produce Common Base Events format output messages
- Developing a log converter that reads the existing log and trace files and transform log and trace entries into the Common Base Event format

The latter process can be simplified by the Generic Log Adapter tool.

The Generic Log Adapter tool provided by the IBM Autonomic Computing Toolkit uses Eclipse as the base development environment. It is a set of plug-ins that are incorporated into the Eclipse development environment. The Generic Log Adapter tool is made up of the following elements:

- Generic Log Adapter Configuration Editor

  The Generic Log Adapter Configuration Editor is also known as the Generic Log Adapter Rule Builder. It enables you to write the Generic Log Adapter configuration file, or simply the adapter, that will determine how the log entries will be transformed into the Common Base Event format. It allows the use of Java, script fragments using regular expressions, or a combination of both to describe the mapping of log entries to the Common Base Event format.

- Generic Log Adapter Runtime environment

  The Generic Log Adapter Runtime is needed to produce the Common Base Events based on the rules and output format defined in the adapter configuration file.

Figure 4-1 on page 79 describes the flow of events from the application log to the generated Common Base Events. For more information about the Generic Log Adapter and the Generic Log Adapter Configuration Editor, see *Generic Log Adapter Getting Started* provided with the Generic Log Adapter.

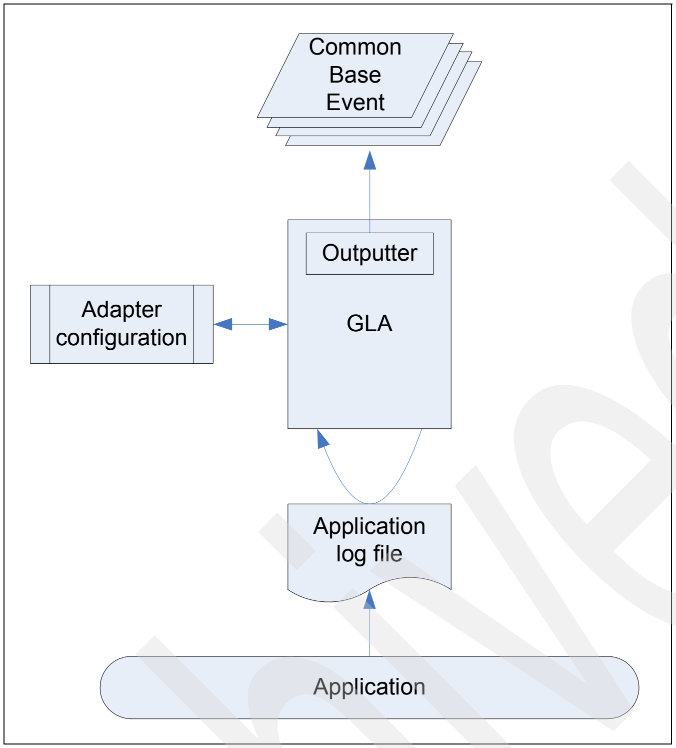


Figure 4-1 Using the Generic Log Adapter to generate Common Base Events

Generic Log Adapter components

In order to perform its job, the Generic Log Adapter uses the information defined in the adapter configuration file. A single Generic Log Adapter Runtime can handle several log files. Each log file is defined in the adapter configuration file by a *context*. Each context is responsible for describing the rules of how to convert a certain log file to Common Base Events.

Each context in the adapter configuration file consists of a series of components that describe the conversion rules for the associated log file. Each context will run as a separate thread independent of the other contexts in the same adapter configuration file. The following components need to be defined for each log file to be handled by the Generic Log Adapter:

- Context
- Sensor
- Extractor
- Parser

- Formatter
- Outputter

Figure 4-2 describes the structure of the adapter configuration file.

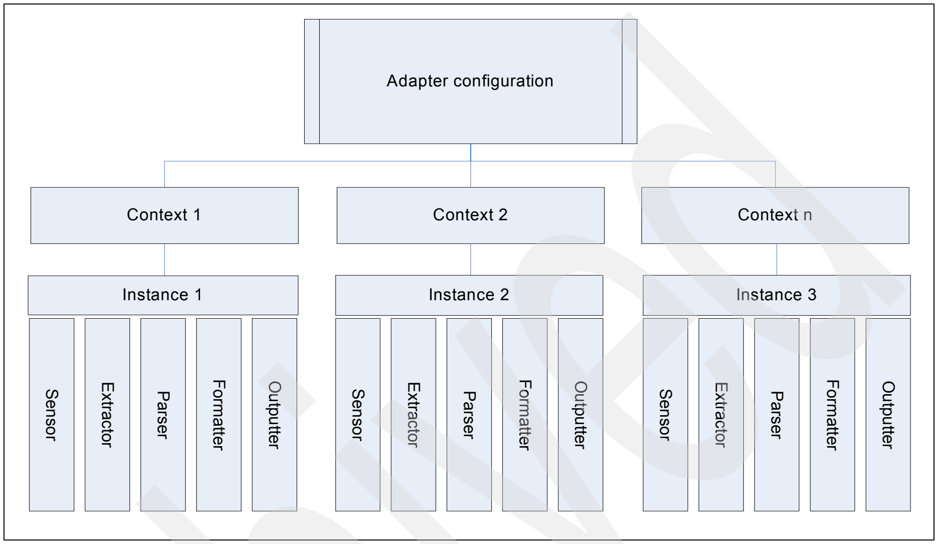


Figure 4-2 Structural view of the adapter configuration file

Context

The context describes the grouping of components that will perform log processing tasks. As described previously, if the Generic Log Adapter needs to process multiple log files, there can be more than one context defined in the adapter configuration file. The package that contains the context is org.eclipse.hyades.logging.adapter.impl. The Generic Log Adapter provides one context type: BasicContext. This context runs each of its components in a single thread. The executable class is org.eclipse.hyades.logging.adapter.impl.BasicContext.

Sensor

The *sensor* portion defines the mechanism that reads the log content to be processed. The package that contains these sensors is org.eclipse.hyades.logging.adapter.sensors. Currently, there are three available sensor types provided by the Generic Log Adapter:

- SingleFileSensor

  This sensor reads local files. The executable class is org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor.

- StaticParserSensor

  This sensor integrates external parsers written in Java. It directly invokes the Java parser and exports the Common Base Event produced by the Java parser. The executable class is org.eclipse.hyades.logging.adapter.config.sensors.StaticParserSensor.

  When defining a StaticParserSensor, the name of the Java class that implements the parser is mandatory. Also, the Java class must implement the org.eclipse.hyades.logging.parsers.IParser interface.

- AdapterCBESensor

  This sensor is used internally by the Generic Log Adapter to monitor the execution of the adapter's execution. The executable class is org.eclipse.hyades.logging.adapter.internal.util.AdapterSensor.

Extractor

The *extractor* provides a mechanism to receive the lines from the sensor and separate the event messages. Simply put, it defines the rules to recognize the message boundaries. The package that contains these extractors is org.eclipse.hyades.logging.adapter.extractors. There are two extractors provided by the Generic Log Adapter:

- RegularExpressionExtractor

  This extractor processes the log file using regular expressions. The executable class is org.eclipse.hyades.logging.adapter.extractors.RegularExpressionExtractor.

- SimpleExtractor

  This extractor processes log files using simple string comparisons and should only be used with log files that contain unchanging message delimiters. The executable class is org.eclipse.hyades.logging.adapter.extractors.SimpleExtractor.

Parser

The *parser* defines a set of string mappings to convert the message received from the extractor to Common Base Event entries.

The parser processes the message in two phases:

1. Global Processing Phase

   A set of global regular expressions are executed against the message provided by the extractor.

2. Attribute Processing Phase

   Specific sets of substitution rules are executed to determine the attribute values.

The parser can process the message into a series of name-value pairs during the global processing phase, and then refer to these pairs by name during the attribute processing phase. There is only one parser provided by the Generic Log Adapter: Parser. The executable class is org.eclipse.hyades.logging.adapter.parsers.Parser

Formatter

The *formatter* takes attributes and their values from the parser and then creates the Common Base Event Java object instance. The generated Common Base Event instance complies with the Common Base Event Version 1.0.1 specifications. There is only one formatter provided by the Generic Log Adapter: CBEFormatter. The executable class is org.eclipse.hyades.logging.adapter.formatters.CBEFormatter.

Outputter

The *outputter* provides a way to wrap the formatted Java object provided by the formatter in a form suitable for storing. For example, it can convert the Common Base Event Java object to an XML format that can be stored in a file. The package that contains these formatters is org.eclipse.hyades.logging.adapter.outputters. There are four outputters provided by the Generic Log Adapter:

- SingleFileOutputter

  This outputter receives the Common Base Events from the formatter and writes them into a file. The executable class is org.eclipse.hyades.logging.adapter.outputters.CBEFileOutputter.

  If the Generic Log Adapter processes multiple log files and stores the Common Base Events in a file, the required outputter class is org.eclipse.hyades.logging.adapter.outputters.CBEConvergentFileOutputter.

  The directory and fileName are the required properties for this outputter.

- StandardOutOutputter

  This outputter receives the Common Base Events from the formatter and writes them to standard output. The executable class is org.eclipse.hyades.logging.adapter.outputters.CBEStdoutOutputter. There is no required property for this outputter.

- LoggingAgentOutputter

  This outputter receives the Common Base Events from the formatter, creates a logging agent, and writes the Common Base Event to the logging agent. The executable class is org.eclipse.hyades.logging.adapter.outputters.CBELogOutputter

  The required property for this outputter is agentName. The optional property is watiUntilLoggingTime, which will wait for the user to attach and start monitoring the agent before it starts writing to the agent. If this property is not specified, the outputter will not wait for the agent to be monitored before writing to it. The wait time needs to be specified in milliseconds.

The Generic Log Adapter also provides an alternative LoggingAgentOutputter that should only be included in adapters that will be used to import logs into the Log and Trace Analyzer tool. The executable class to be used is org.eclipse.hyades.logging.adapter.config.outputters.StaticParserOutputter. There is no required property for this outputter class.

- NotificationOutputter

  This outputter receives Common Base Event instances and sends the externalized records through a Notification Source to a Notification Sink. The executable class is com.ibm.etools.logging.adapters.outputters.CBENotifyOutputter. This outputter class is available in the outputters.jar file located in the <Eclipse_DIR>/plugins/com.ibm.etools.logging.adapters.outputters_6.0.0, where <Eclipse_DIR> is the Eclipse installation directory.

  The required properly is resourceURL, where the URL of the Notification Sink needs to be provided.

4.2 Creating a simple adapter

In this scenario, we use the Generic Log Adapter to create an adapter for a sample application that generates log information using a text file. The application named SampleApplication generates the following log information on a file named sample.log:

```
ITSO434W Resource utilization above 80%
ITSO945E Cannot connect to data source
ITSO259I Successful connection
ITSO314C Critical internal error-process exiting
ITSO001I SampleApplication starting...
ITSO900I SampleApplication stopping
ITSO901I SampleApplication stopped!!!
ITSO999I SampleApplication restarting
```

In order to create an adapter configuration file for the sample application, the following tasks need to be performed:

1. Creating the adapter configuration file
2. Configuring the Common Base Event elements
3. Configuring the adapter outputter
4. Testing the adapter configuration file

5. Deploying and running the adapter configuration file

We describe these tasks in detail in the following sections.

4.2.1 Creating the adapter configuration file

The Generic Log Adapter provides a plug-in to Eclipse. This plug-in is called Generic Adapter UI and enables the Generic Log Adapter Configuration Editor. The adapter configuration file can be easily edited using the Adapter Configuration Editor that runs under Eclipse. To generate a new adapter configuration file from scratch using the Eclipse Generic Log Adapter Configuration Editor, complete the following steps:

1. Start Eclipse by running <Eclipse_Install>\eclipse\StartEclipse.bat, where <Eclipse_Install> is the install directory of the Eclipse development kit. Typically, it will be C:\IBM\AutonomicComputingToolkit\Developmentkit.
2. Open a Generic Log Adapter Perspective by selecting **Window** → **Open Perspective** → **Other** → **Generic Log Adapter**.
3. Create a new project to work with by selecting **File** → **New** → **Other** → **Simple** → **Project**, and clicking **Next**. A window similar to the one shown in Figure 4-3 opens.

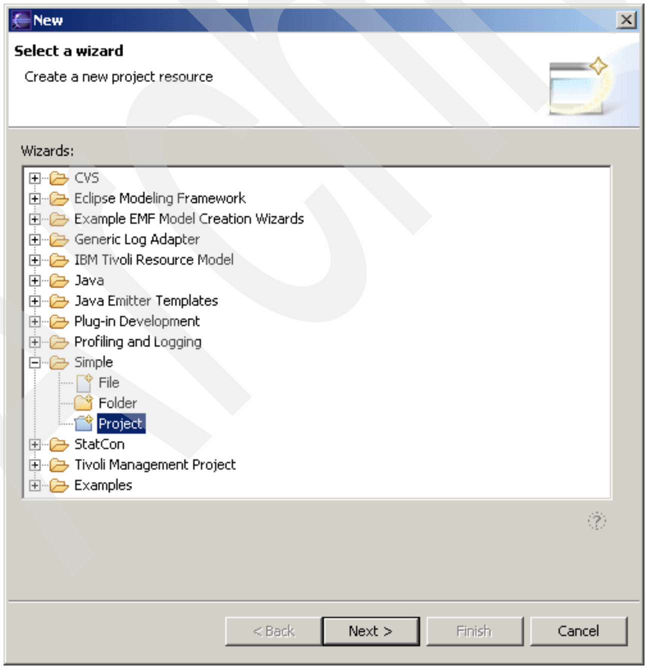


Figure 4-3 Create a new simple project

4. Give it a project name, such as `SimpleGLAProject`, and click **Finish**. This is shown in Figure 4-4. Optionally, a different location for the project can be selected.

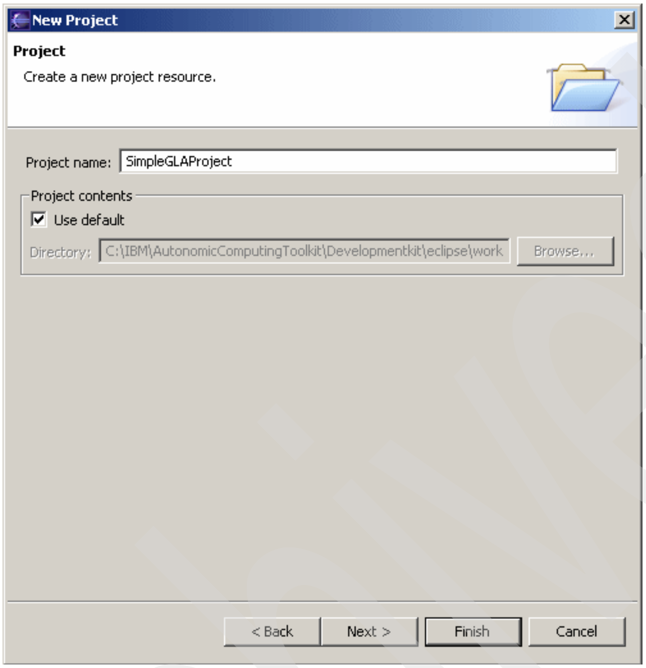


Figure 4-4 Specify the project name and location

5. Create a new Generic Log Adapter file by selecting **File** → **New** → **Other** → **Generic Log Adapter** → **Generic Log Adapter File**, and clicking **Next**. A window similar to the one shown in Figure 4-5 opens.

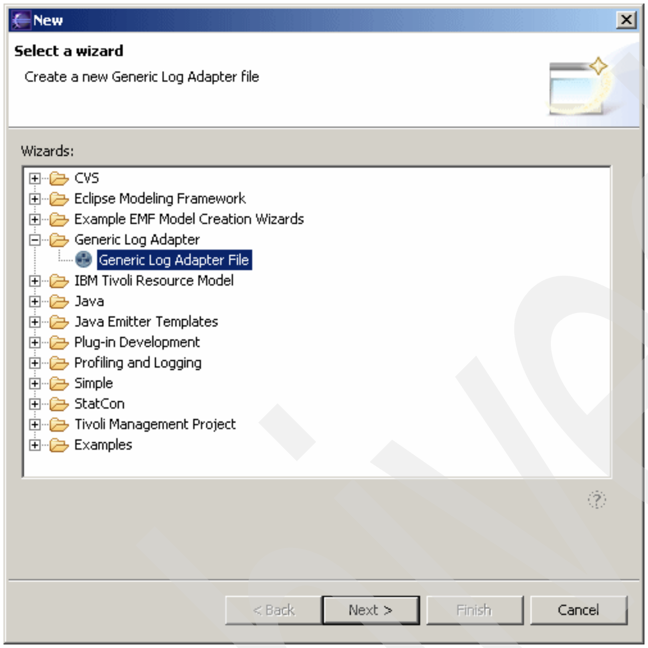


Figure 4-5 Create a new Generic Log Adapter file

6. Choose the project in which we want to include this adapter configuration file, give it a name with an ".adapter" extension, and click **Next**, as shown in Figure 4-6 on page 87.

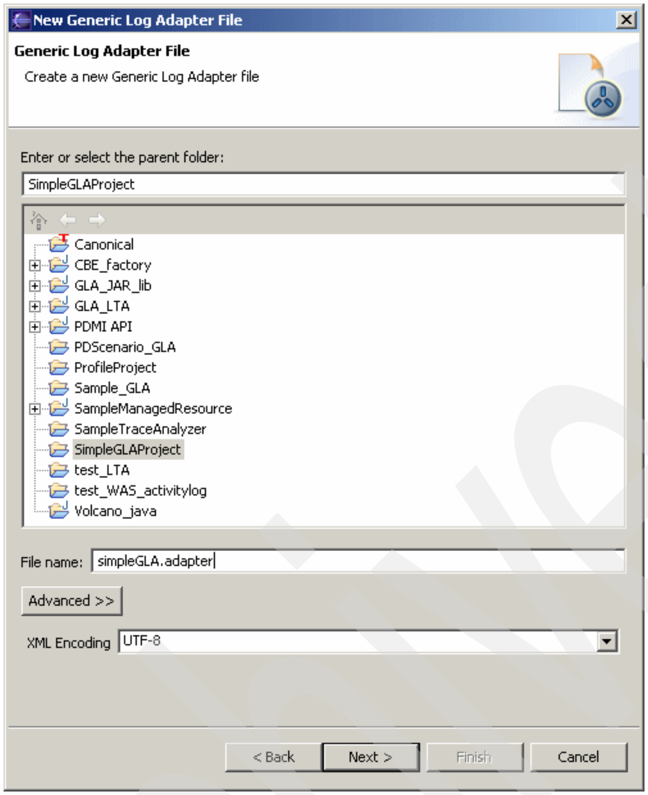


Figure 4-6 Specify the Generic Log Adapter File name

7. Specify the template log file that will be used to define the parser rules and input log file for the Generic Log Adapter engine, as shown in Figure 4-7, and click **Finish**. Our sample application uses the sample.log file.

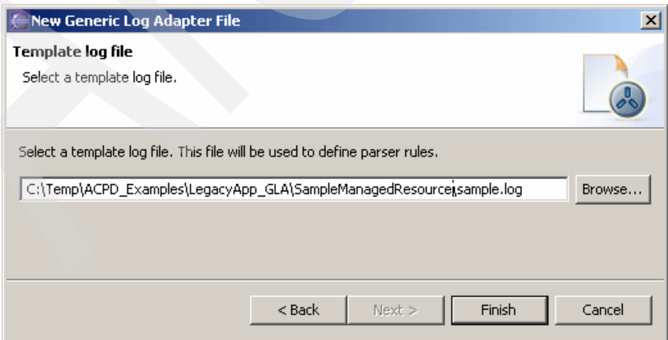


Figure 4-7 Provide sample log file that will be used as input for the GLA

8. Expand the adapter configuration file hierarchy to be able to see the full tree view, as shown in Figure 4-8.

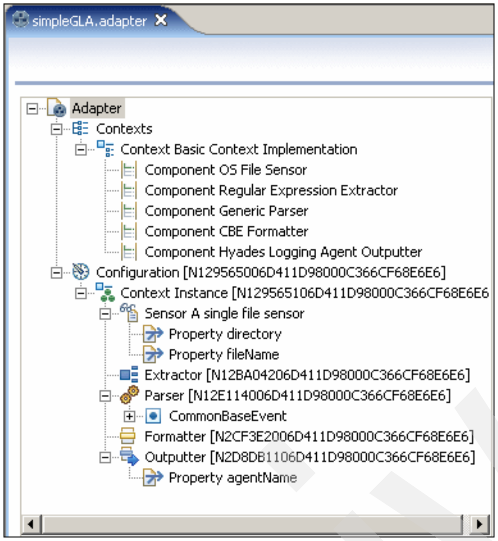


Figure 4-8 The GLA adapter file opened using the Configuration Editor

The newly created adapter configuration file contains the following characteristics:

- Context

  It uses one context that relates to the single log file created by the sample application sample.log file. This context uses the org.eclipse.hyades.logging.adapter.impl.BasicContext class.

- Sensor

  This sensor will read our example application's log file: sample.log. It implements the SingleFileSensor sensor type with the executable class org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor.

- Extractor

  The extractor type used in this example is the RegularExpressionExtractor with the executable class org.eclipse.hyades.logging.adapter.extractors.RegularExpressionExtractor.

- Parser

  It uses the default parser that will further process the log messages provided by the extractor based on specific rules defined in 4.2.2, "Configuring the Common Base Event elements" on page 89.

- Formatter

  It uses the default CBEFormatter formatter.

- Outputter

  We can use a custom outputter class to forward the generated Common Base Events to a certain destination or event sink instead of writing it to a file using the CBEFileOutputter class. For an example of writing a custom outputter, refer to 4.5, “Writing a custom outputter” on page 124.

Now, we are ready to use the Adapter Configuration Editor to edit our adapter configuration file.

4.2.2 Configuring the Common Base Event elements

Now, we have an empty adapter configuration file. In the following steps, we describe how to write the substitute rules to convert the log file message to the Common Base Event format and show how to provide all the required elements of the 3-tuple structure, as described in 3.3, “Common Base Event properties” on page 38. They are:

- sourceComponentID
- reporterComponentID
- creationTime
- situation category

In this example scenario, the reporterComponentID will not be required, because the reporter component and the affected component represent the same component.

We will also fill out additional optional elements that are particular to the example provided in this chapter. The resulting sample adapter is provided with this book. For details, refer to Appendix A, “Additional material” on page 379.

To edit the newly created adapter file, perform the following steps:

1. Open the project in Eclipse.
2. Open the adapter configuration file in the Configuration Editor by double-clicking the adapter configuration file. In our example, the adapter file is named simpleGLA.adapter.
3. Expand the adapter tree, as shown in Figure 4-8 on page 88. Expand **Adapter** → **Configuration** → **Context Instance** → **Parser** → **CommonBaseEvent**, which defines the structure of the generated Common Base Event.

Defining the sourceComponentID element

In the following steps, we show how to add the sourceComponentID element and how to configure the component property. All the remaining required properties must be defined following similar steps. We also show how to add an optional property.

To add the sourceComponentID element and define the component property to it, perform the following tasks:

1. Expand **Adapter** → **Configuration** → **Context Instance** → **Parser** → **CommonBaseEvent**, right-click, and select **Add** → **sourceComponentId**, as shown in Figure 4-9.

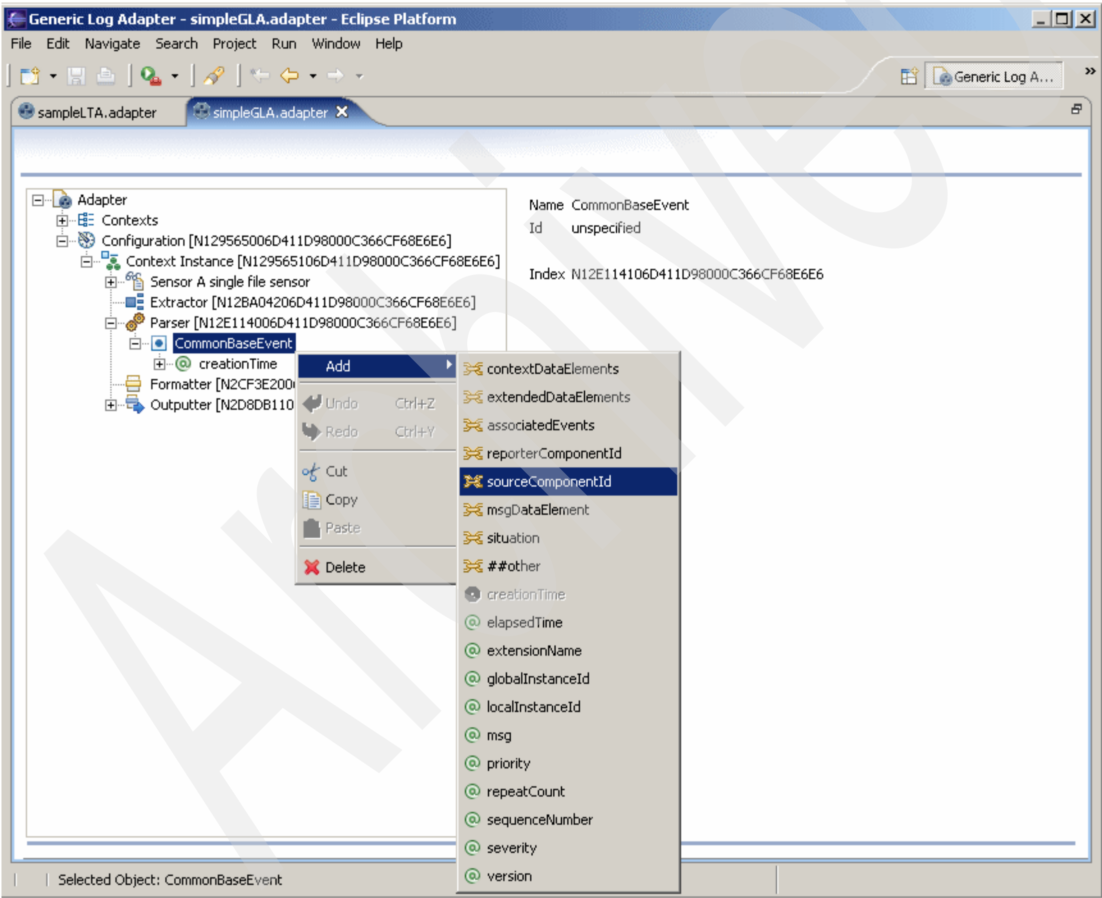


Figure 4-9 Add the sourceComponentID to the Common Base Event

2. The sourceComponentID element defines the following required properties:

   – location
   – locationType
   – component
   – subComponent
   – componentIdType
   – componentType

   To configure the component property, select **sourceComponentId** and expand **component.** Select **Substitution Rule** to specify what will be placed in the component element of the Common Base Event. This is shown in Figure 4-10.

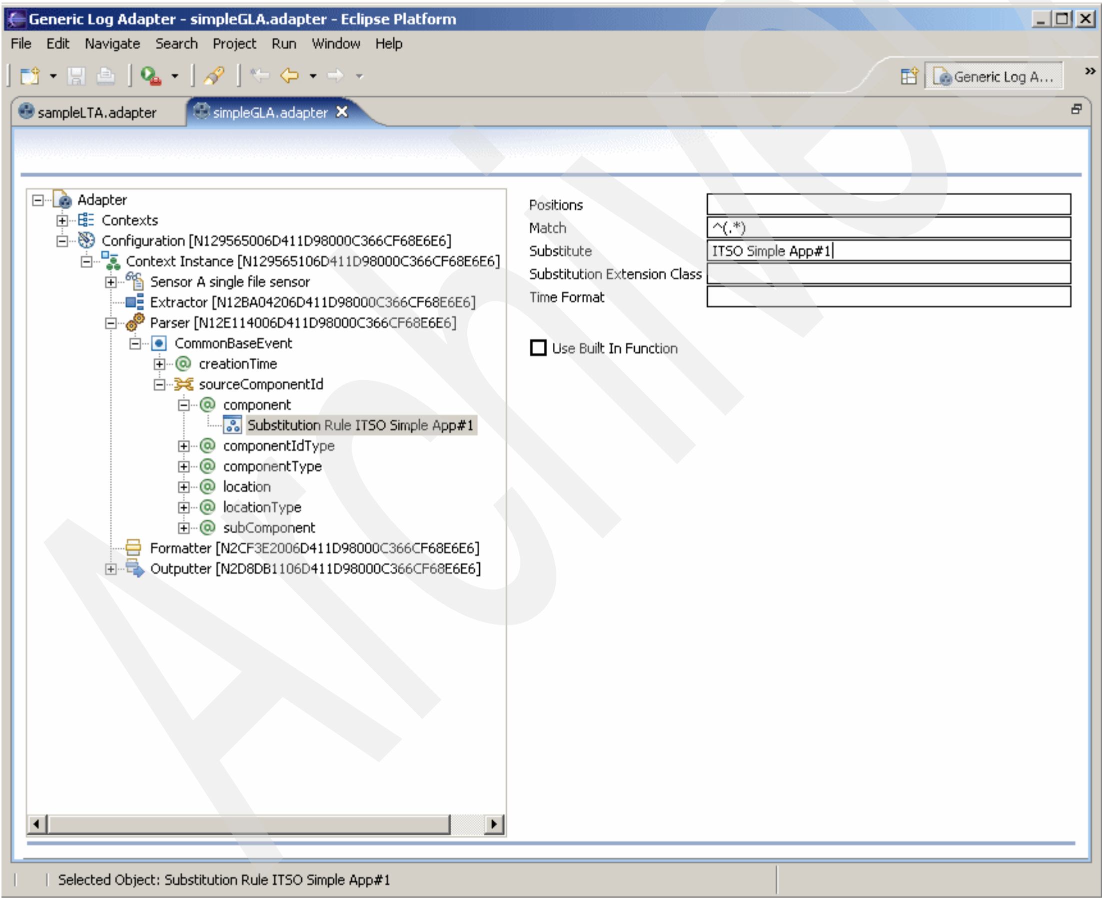


Figure 4-10 Set the component property

All the remaining required elements must be set similarly.

3. To add an optional attribute to the sourceComponentId, select **sourceComponentId**, right-click, and select, for example, **Add** → **application**, as shown in Figure 4-11.

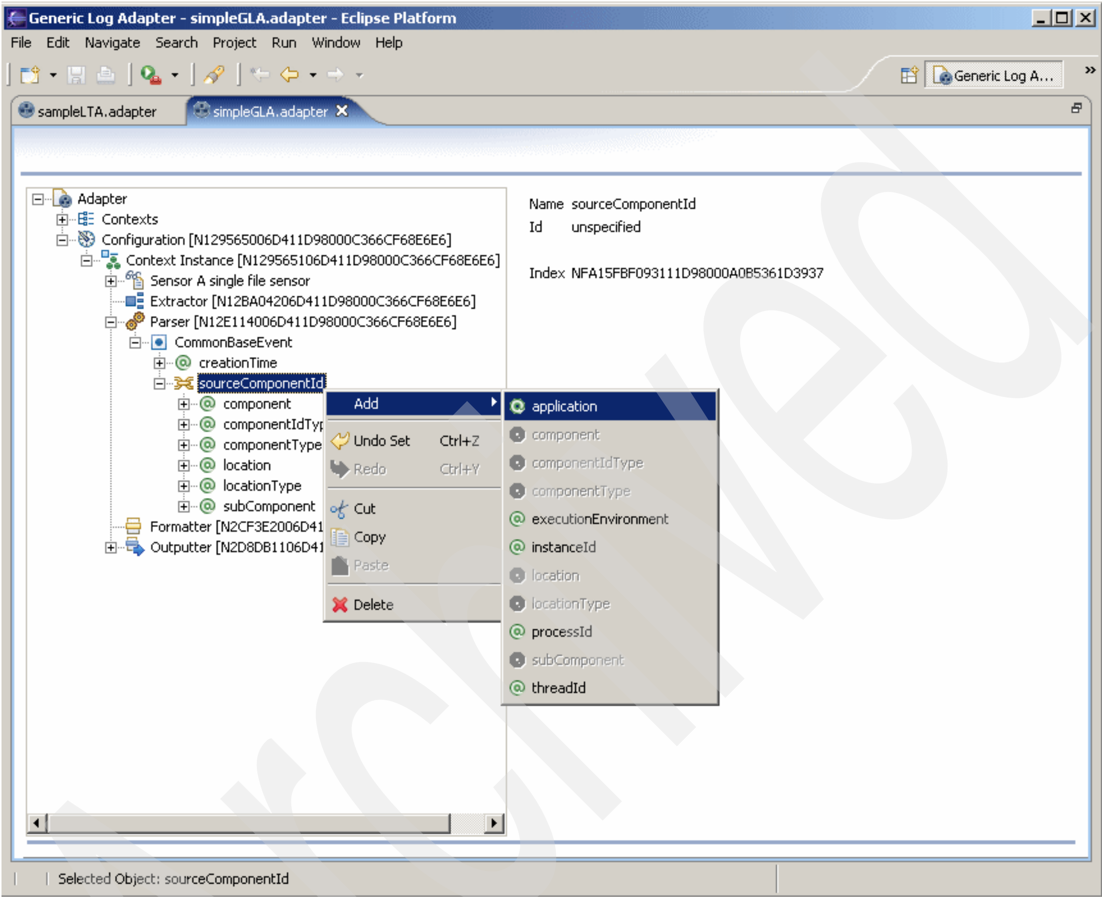


Figure 4-11 Add an application to sourceComponentId

4. To specify the substitution rule of what will be written in the application attribute, select **application**, right-click, and select **Add** → **Substitution Rule**.
5. Now, define the substitution rule to define what will be written in the generated Common Base Event. The substitution rule has a Match field that defines the portion of the incoming text to replace. In our case, we choose the complete incoming message `^(.*)` and replace it with `ITSOSimpleApp1`, as shown in Figure 4-12 on page 93.

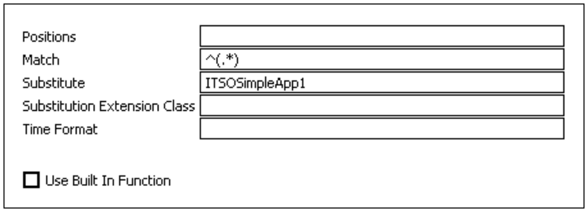


Figure 4-12 The substitution rule for the application attribute

Figure 4-13 presents the adapter configuration file hierarchy after completing the configuration of the sourceComponentID element.

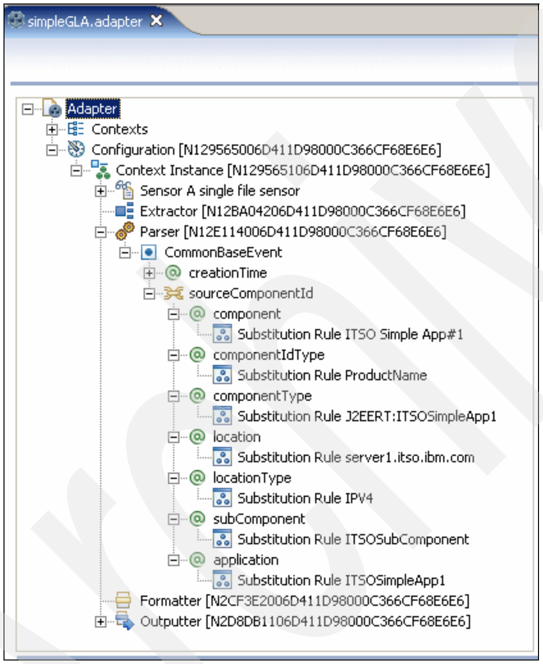


Figure 4-13 The complete sourceComponentID element

Defining the creationTime element

Our SampleApplication generates log entries using the following format for date and time:

```
[<Day of the week> <Month> <Day> HH:MM:SS <Time zone> <Year>]
```

For example:

```
[Fri Feb 04 11:14:37 CST 2005]
```

As described in 3.3, “Common Base Event properties” on page 38, the creationTime element must be defined by a type called dateTime as specified in XML schema. The format for the creationTime element must be `CCYY-MM-DDThh:mm:ss`, where `CC` represents the century, `YY` the year, `MM` the month, and `DD` the day. The letter `T` is the date and time separator and `hh`, `mm`, and `ss` represent hour, minute, and second, respectively.

Our adapter must convert the date and time from `[<Day of the week> <Month> <Day> HH:MM:SS <Time zone> <Year>]` to the `CCYY-MM-DDThh:mm:ss` format. In our example, we will make use of regular expressions to achieve that.

To configure the creationTime element, expand **Adapter** → **Configuration** → **Context Instance** → **Parser** → **CommonBaseEvent** → **creationTime**. Select **Substitution Rule** to specify what will be placed in the field. In our case, we use the following substitution rules:

Match `(.*) (MMM) (\d{2}) (\d{2}:\d{2}:\d{2})(.*)(\d{4})(.*)`
where `MMM` represents the first three letter of the month

Substitute `$6-MM-$3T$4Z`
where `MM` represents the month in numerical format

We have to define a substitution rule for each month of the year. Figure 4-14 on page 95 shows the complete creationTime element for our example adapter.

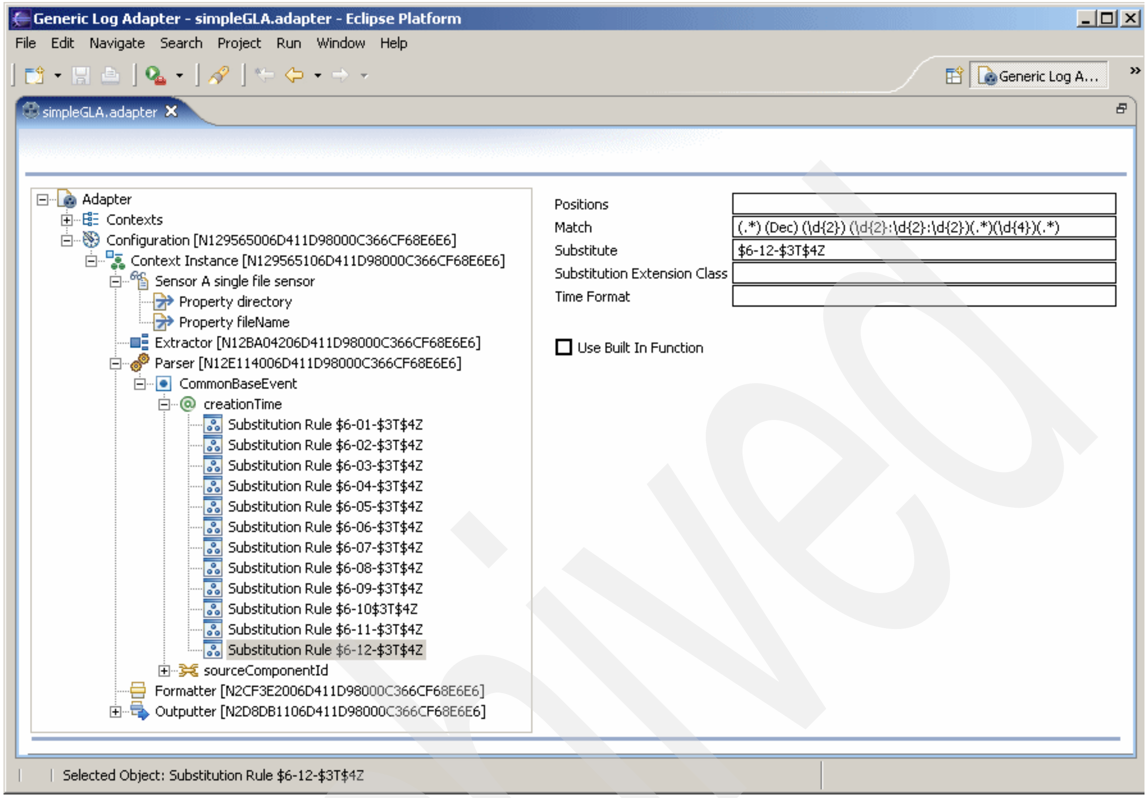


Figure 4-14 Set the creationTime element

Defining the situation element

In this section, we define the situation element for our case study sample application. The application SampleApplication generates the following log information:

```
ITSO434W Resource utilization above 80%
ITSO945E Cannot connect to data source
ITSO259I Successful connection
ITSO314C Critical internal error-process exiting
ITSO001I SampleApplication starting...
ITSO900I SampleApplication stopping
ITSO901I SampleApplication stopped!!!
ITSO999I SampleApplication restarting
```

Our first step is to identify the situation elements to be defined by classifying each message generated by the SampleApplication and matching them to a valid situation category. For a list of valid situations, refer to Chapter 3, “Common Base Events” on page 35.

Table 4-1 presents the matching between the messages generated by the SampleApplication and the situation types.

Table 4-1 Situation types

Log message	Situation
ITSO001I SampleApplication starting... ITSO999I SampleApplication restarting	startSituation
ITSO945E Cannot connect to data source ITSO259I Successful connection ITSO314C Critical internal error-process exiting	connectSituation
ITSO900I SampleApplication stopping ITSO901I SampleApplication stopped!!!	stopSituation
ITSO434W Resource utilization above 80%	reportSituation

Next, for each situation that has been identified, a series of subelements are required. All the subelements must be defined under the proper situationType definition. For example, the StartSituation requires reasoningScope, successDisposition, and situationQualifier. For each situation type identified earlier, these subelements must be defined by taking into account the meaning of each message and matching the message with a valid value for the subelement.

Table 4-2 provides a break-down of the situation elements and their respective subelement definitions for our case study scenario.

Table 4-2 Situation subelements definition

Subelement definitions			
Situation: StartSituation			
Message	**Reasoning Scope**	**SituationQualifier**	**SuccessDisposition**
ITSO001I	INTERNAL	START INITIATED	SUCCESSFUL
ITSO999I	INTERNAL	RESTART INITIATED	SUCCESSFUL
Situation: ConnectSituation			
Message	**Reasoning Scope**	**SituationDisposition**	**SuccessDisposition**
ITSO945E	INTERNAL	CLOSED	UNSUCCESSFUL
ITSO259I	INTERNAL	CLOSED	SUCCESSFUL
ITSO314C	INTERNAL	CLOSED	UNSUCCESSFUL

Subelement definitions			
Situation: StopSituation			
Message	**Reasoning Scope**	**SituationQualifier**	**SuccessDisposition**
ITSO900I	INTERNAL	STOP INITIATED	SUCCESSFUL
ITSO901I	INTERNAL	STOP COMPLETED	SUCCESSFUL
Situation: ReportSituation			
Message	**Reasoning Scope**	**ReportCategory**	
ITSO434W	INTERNAL	STATUS	

The last step is to define the situations in the adapter configuration file. This can be done by performing the following steps:

1. Expand **Adapter** → **Configuration** → **Context Instance** → **Parser** → **CommonBaseEvent**, right-click, and select **Add** → **situation**, as shown in Figure 4-15 on page 98.

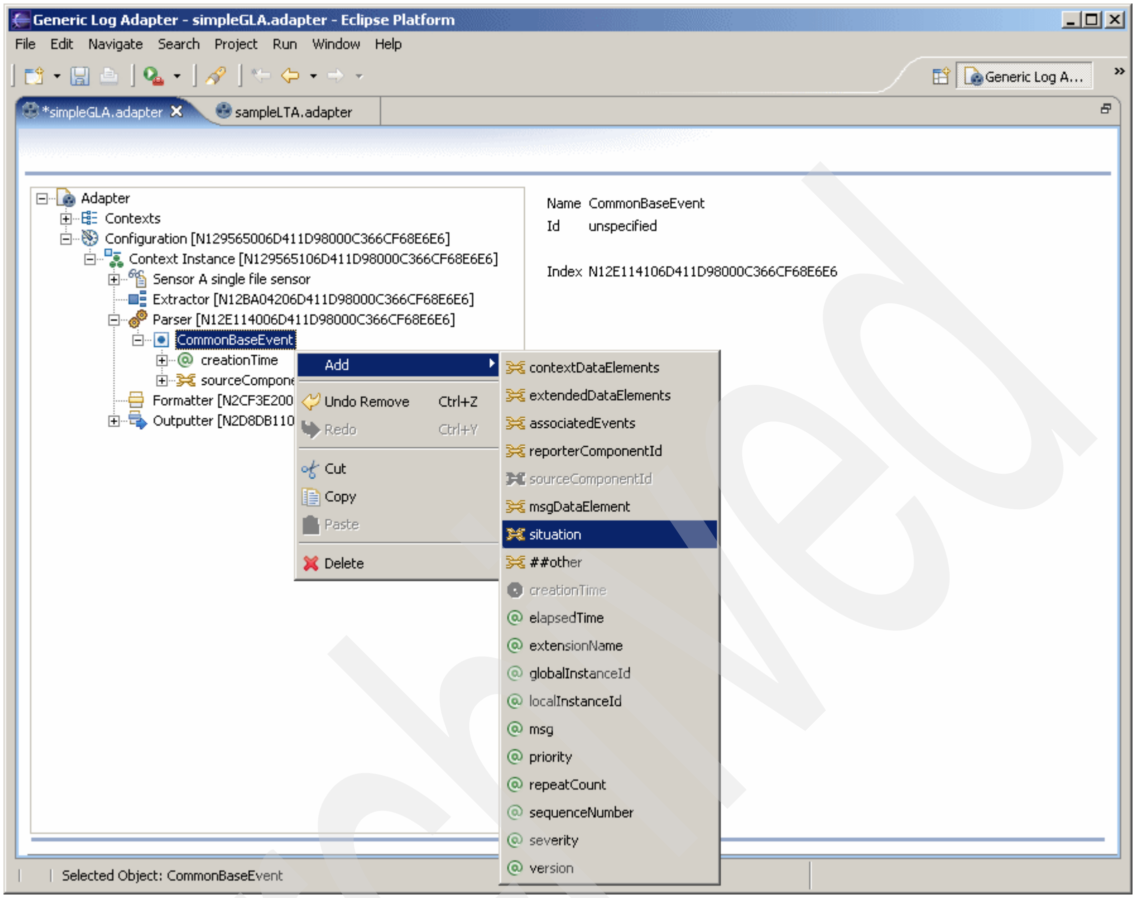


Figure 4-15 Define a situation

2. To add a situation type, select **situation**, right-click, and select **Add** → **situationType**.

3. All the situations identified for the application will be defined under the situationType definition performed in the previous step. For example, to define the startSituation, select **situationType**, right-click, and select **Add** → **startSituation**.

4. All the required subelements of each defined situation need to be configured. For our case study example, we use the information provided in Table 4-2 on page 96 to define them. For example, to define the subelement reasoningScope of the startSituation element, expand **startSituation** → **reasoningScope**, and select **Substitution Rule.** Now, we define the substitution rule to define what will be written in the Common Base Event for this situation. In our case, we choose to apply the `.*ITS0001I.*|.*ITS0999I.*` regular expression on the incoming message and define reasoningScope as `INTERNAL`, as shown in Figure 4-16.

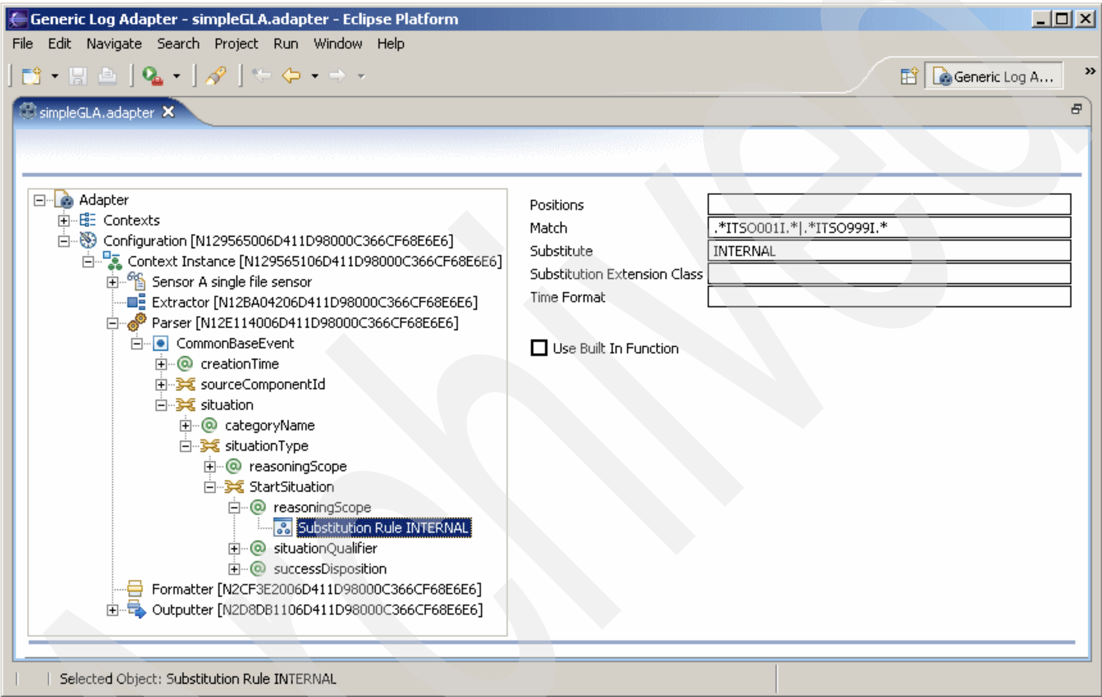


Figure 4-16 Define the reasoningScope subelement

Repeat this step to define the remaining subelements for startSituation using the information provided in Table 4-2 on page 96.

5. A categoryName for the startSituation must also be defined. Expand **Adapter** → **Configuration** → **Context Instance** → **Parser** → **situation** → **categoryName**, and select **Substitution Rule.** Now, we define the substitution rule to define what will be written in the Common Base Event for this categorySituation. In our case, we choose to apply the `.*ITS0001I.*|.*ITS0999I.*` regular expression on the incoming message and substitute it with `StartSituation`, as shown in Figure 4-17 on page 100.

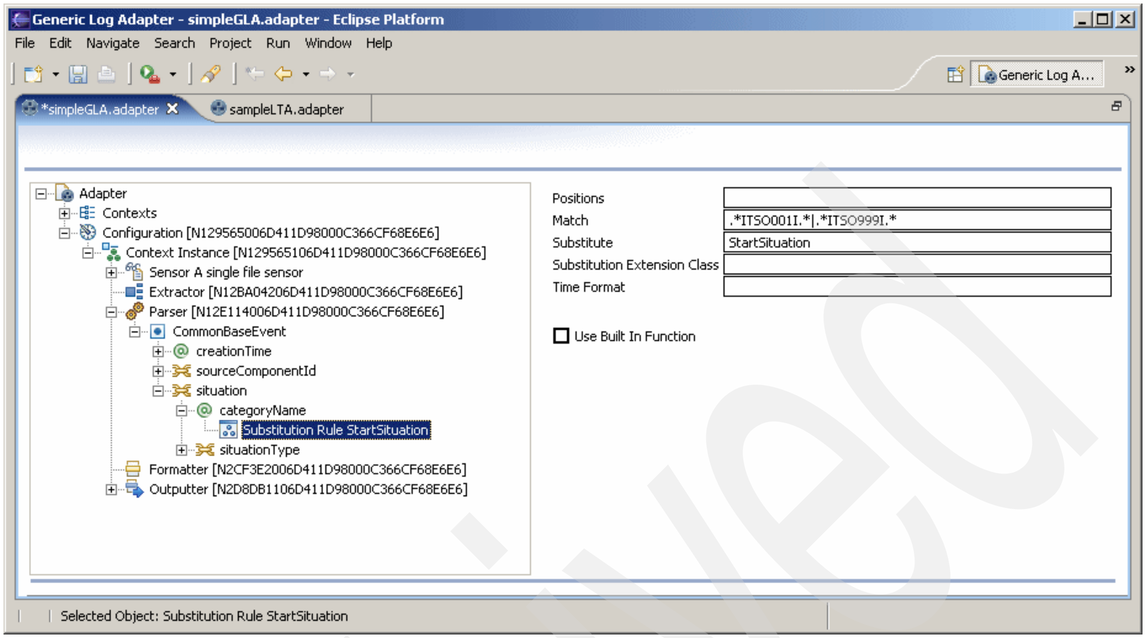


Figure 4-17 Define categoryType

6. The remaining identified situations for our case study scenario, stopSituation, connectSituation, and reportSituation, must be defined by repeating the previous four (2 on page 98-5 on page 99) steps.
7. Repeat the previous steps to generate any other optional Common Base Event elements that are needed. For our case study, we define the following optional elements in our adapter configuration file:
   – severity

     The severity level for our messages needs to be defined based on the indication of severity of the event. Based on the guidelines provided by the *Autonomic Computing Toolkit Developer's Guide*, SC30-4083, the following levels are defined for our case study sample application:

20 - Harmless	Mapped to the informational messages: `.*] ITSO\d{3}(I).*`
30 - Warning	Mapped to the application's warning messages: `.*] ITSO\d{3}(W).*`
50 - Critical	Mapped to the application's error messages: `.*] ITSO\d{3}(E).*`
60 - Fatal	Mapped to the application's critical messages: `.*] ITSO\d{3}(C).*`

– msg

This field is the informative attribute of the message. Although optional, we highly recommend that it is defined on all Common Base Events.

For more information about the different optional Common Base Event elements, check the *Autonomic Computing Toolkit Developer's Guide*, SC30-4083.

4.2.3 Configuring the adapter outputter

As described in "Generic Log Adapter components" on page 79, the outputter component externalizes the resulting Common Base Event. In this section, we show how to place all the events generated by the SampleApplication to a file. In 4.5, "Writing a custom outputter" on page 124, we provide examples to change the behavior of the outputter.

When the adapter file is created, the Generic Log Adapter Configuration Editor defines the outputter to a default configuration. The default configuration uses the LoggingAgentOutputter with executable class org.eclipse.hyades.logging.adapter.outputters.CBELogOutputter. Because we want the outputter to write the events to a log file, we need to change the outputter configuration as follows:

1. Identify the outputter context configuration. This can be done by selecting **Adapter → Contexts → Context Basic Context Implementation → Component hyades Logging Agent Outputter**. (This component has **Outputter** for the Role.) See Figure 4-18 on page 102.

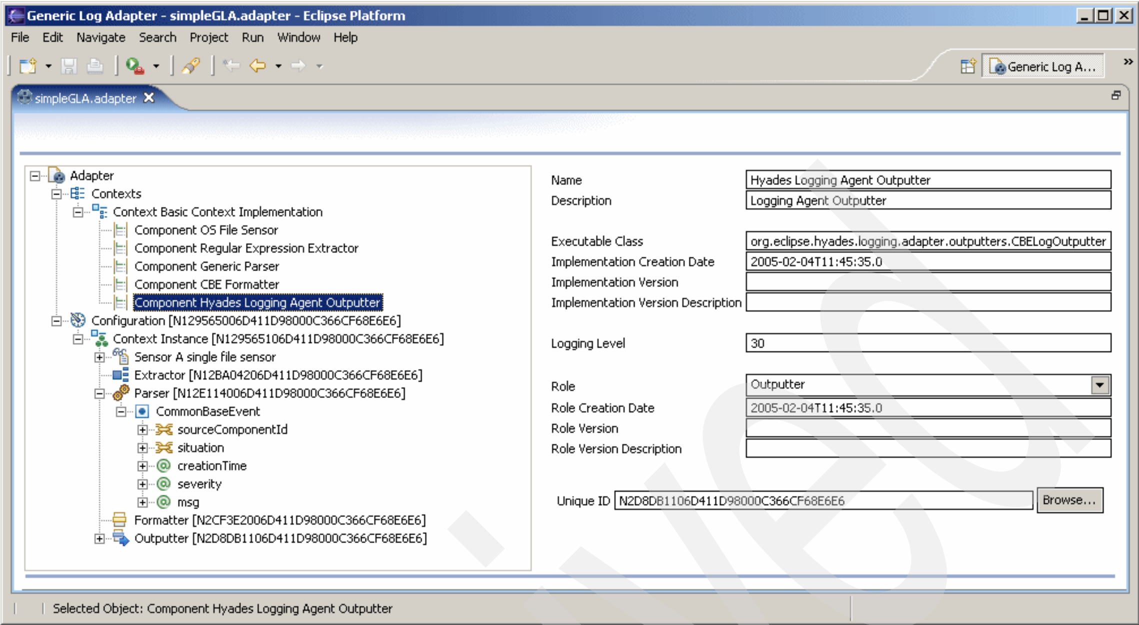


Figure 4-18 Default outputter context configuration

2. Change the outputter context configuration with the following information:

   Name `Single File Outputter`

   Description `Writes Common Base Events to a file`

   Executable Class

   `org.eclipse.hyades.logging.adapter.outputters.CBEFileOutputter`

   Figure 4-19 on page 103 shows the modified outputter context configuration.

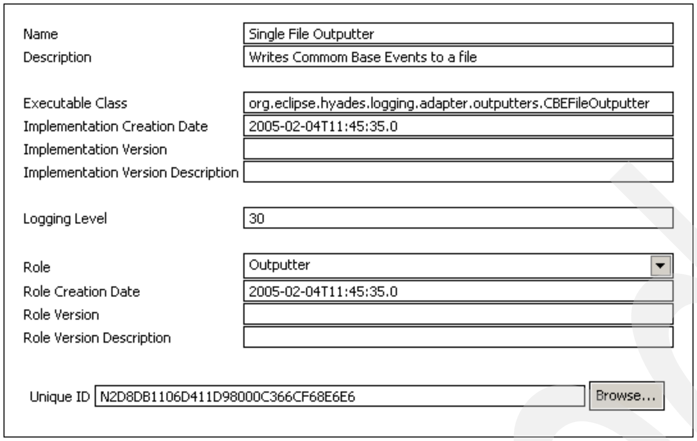


Figure 4-19 Modified outputter context configuration

3. Change the outputter context instance configuration with the required properties for the outputter type SingleFileOutputter. They are fileName and directory. This can be done by selecting **Adapter** → **Configuration** → **Context instance** → **Outputter**.
4. Delete the existing property named agentName by right-clicking the agentName property and selecting **Delete**.
5. Add a new property to the outputter by right-clicking the Outputter and selecting **Add** → **Property**. Configure the Property as follows:

   Property Name — For example: `fileName`

   Property Value — The file name of the Common Base Event repository file, for example, `CBEOut.log`
6. Repeat the previous step to add a second property to the outputter as follows:

   Property Name — For example: `directory`

   Property Value — The directory hosting the Common Base Event repository file

4.2.4 Testing the adapter configuration file

The Generic Log Adapter perspective within Eclipse enables us to test the adapter configuration file we are creating for our SampleApplication in an easy way. Import the simpleGLA.adapter file into a simple Eclipse project. This sample adapter file is provided with this book as an example of an adapter configuration file. For more details, refer to Appendix A, “Additional material” on page 379. Use the following steps to test the adapter configuration file:

1. Open the project you are working with in Eclipse.
2. Open the adapter configuration file using Eclipse in the Generic Log Adapter perspective.
3. Select the **Context Instance** in the editor. This is required in case the adapter contains more then one context instance. Click the **Rerun adapter** icon, as shown in Figure 4-20.

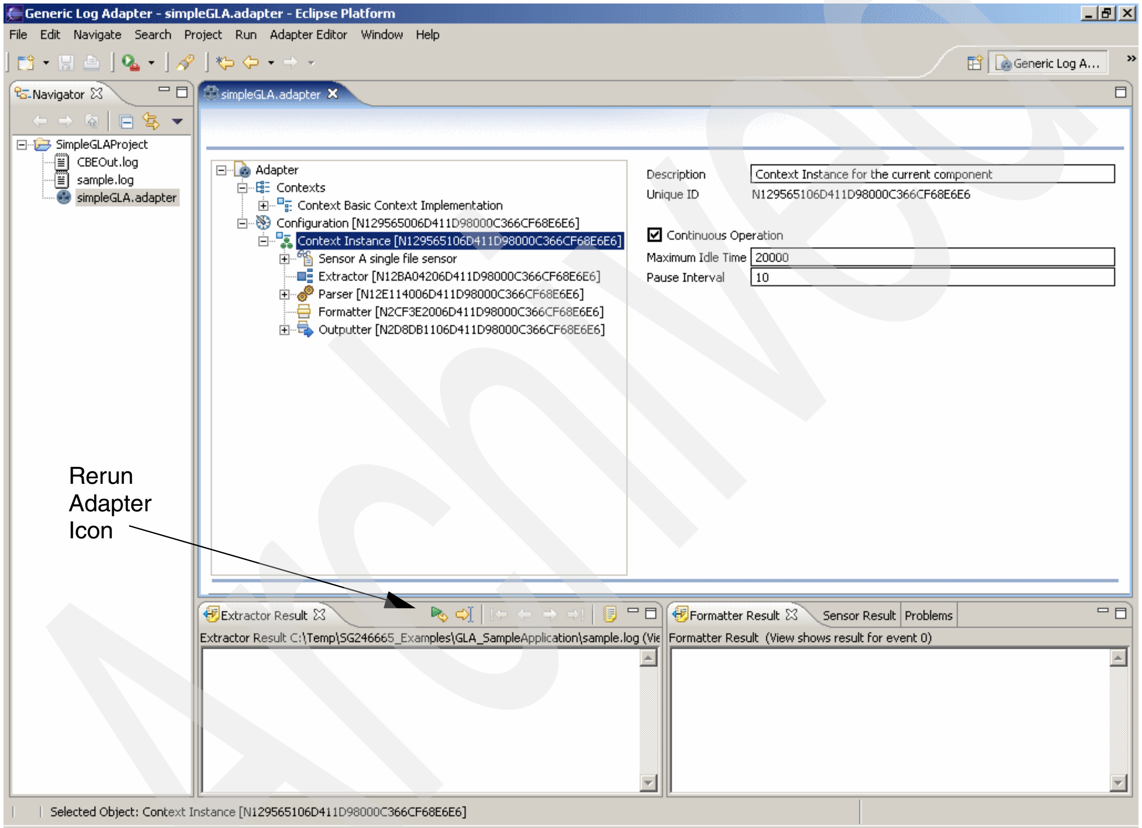


Figure 4-20 Select the Context Instance in the adapter file and rerun the adapter

4. Now, we can run, refresh, and step through the log template file and see the generated Common Base Event in the Formatter Result tab, as shown in Figure 4-21 on page 105.

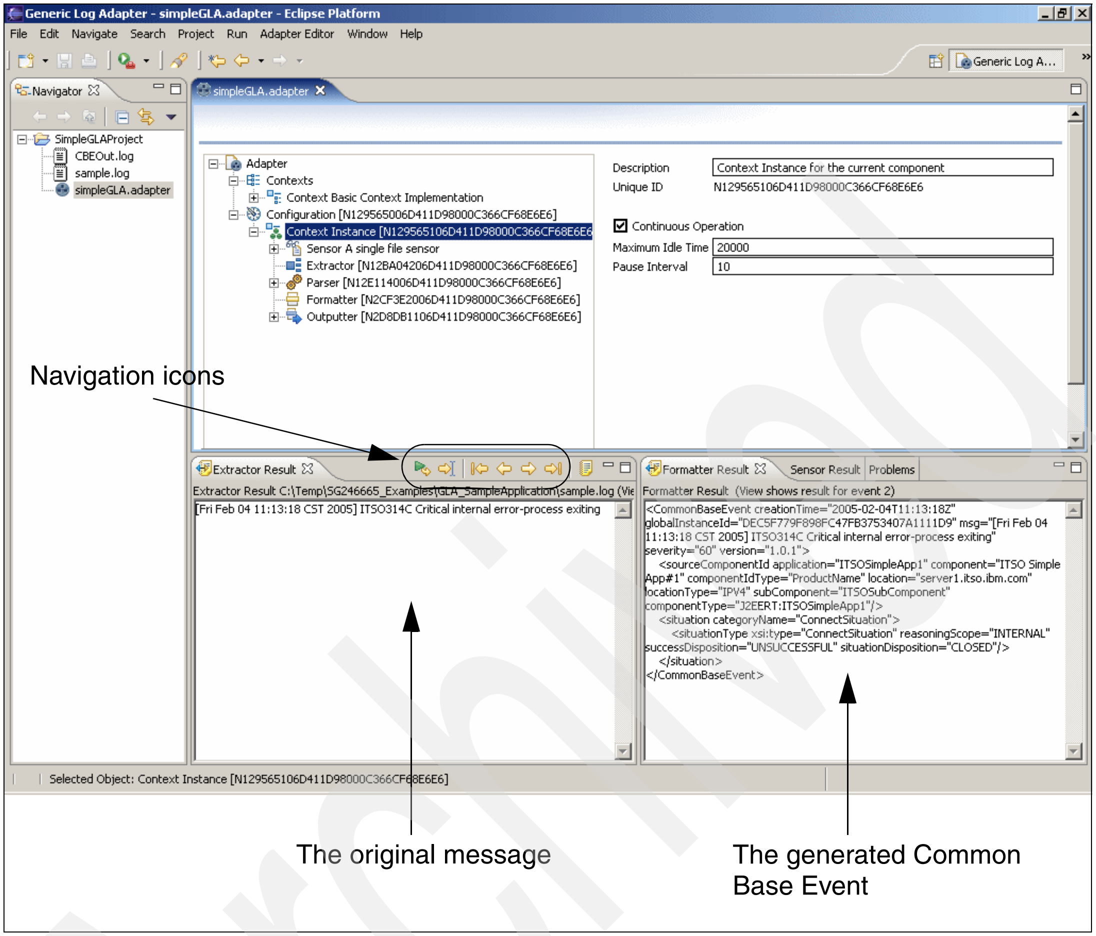


Figure 4-21 Run, refresh, and step through the template log file

5. If the result is incorrect, change the substitution rule, save the file, and rerun the adapter.
6. Also, verify that the outputter is generating the Common Base Event in the correct format. In our case, the adapter has created a CBEOut.log file in the predefined directory.

4.3 Creating a multiple contexts adapter

A single Generic Log Adapter Runtime can handle several log files. Each log file is defined in the adapter configuration file by a context. Figure 4-2 on page 80 shows the structure of the adapter configuration file using multiple contexts. In this section, we show how to create an adapter configuration file with multiple contexts.

Figure 4-22 is an illustration of the multiple context case study scenario that we discuss in this section. Here, the adapter configuration file has two contexts: context1 and context2. Both contexts use the CBEConvergentFileOutputter outputter class to write the Common Base Events to a single file named CBEOut.log. The log information is read by the Generic Log Adapter Runtime from the sample.log and sample2.log files.

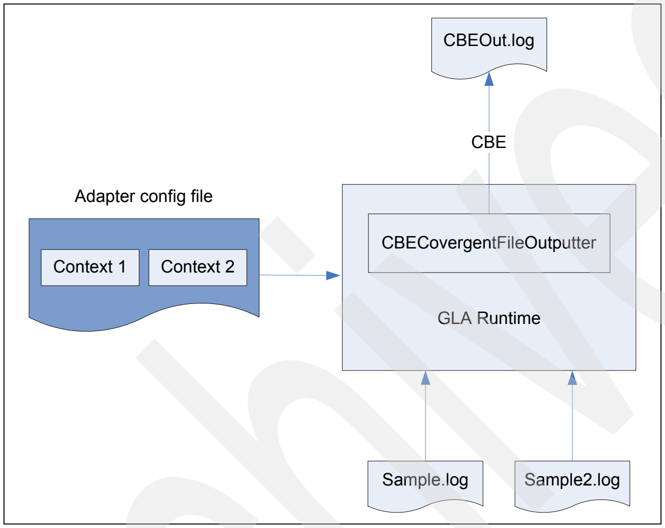


Figure 4-22 Illustration of multiple context in the case study

We already created a Generic Log Adapter configuration file with single context in 4.2, “Creating a simple adapter” on page 83. In this scenario, we create a second context in the same adapter file created in the previous example. The log file sample2.log that is created by the application name Sample2Application is defined in the adapter file by the second context. The following information is logged in the sample2.log:

```
[Thus Feb 24 11:16:35 CST 2005] ITS0001I Sample2Application starting...
[Thu Feb 24 11:16:37 CST 2005] ITS0026I RMI Connector unavailable at port 2809
[Thu Feb 24 11:16:42 CST 2005] ITS0134I File transfer is configured with
host=acpd001, port=9090, security enabled=false
[Thu Feb 24 11:17:03 CST 2005] ITS0003I Configuration synchronization completed
[Thu Feb 24 11:17:43 CST 2005] ITS0013I SOAP connector available at port 8880
[Thu Feb 24 11:23:24 CST 2005] ITS0900I Sample2Application stopping
[Thu Feb 24 11:23:26 CST 2005] ITS0901I Sample2Application stopped!!!
```

For the multiple context setup using an existing adapter configuration file, the following tasks need to be performed:

1. Adding a new context instance to the adapter file.
2. Adding a second context to the adapter file.
3. Associating the new context components.
4. Testing the multiple context instance adapter file.

We describe these tasks in detail in the following sections.

4.3.1 Adding a new context instance to the adapter file

First, we need to open the existing adapter file. Complete the following steps:

1. Open the project in Eclipse.
2. Open a Generic Log Adapter perspective by selecting **Window** → **Open Perspective** → **Other** → **Generic Log Adapter**.
3. Open the adapter configuration file in the Configuration Editor. In our example, we open previously created simpleGLA.adaper file.
4. Save the adapter configuration file as multiGLA.adapter.

Now, we modify the multiGLA.adapter file for the multiple context setup. To create a new context instance in the adapter file, expand **Adapter** → **Configuration**, right-click, and select **Add** → **Context Instance**, as shown in Figure 4-23 on page 108.

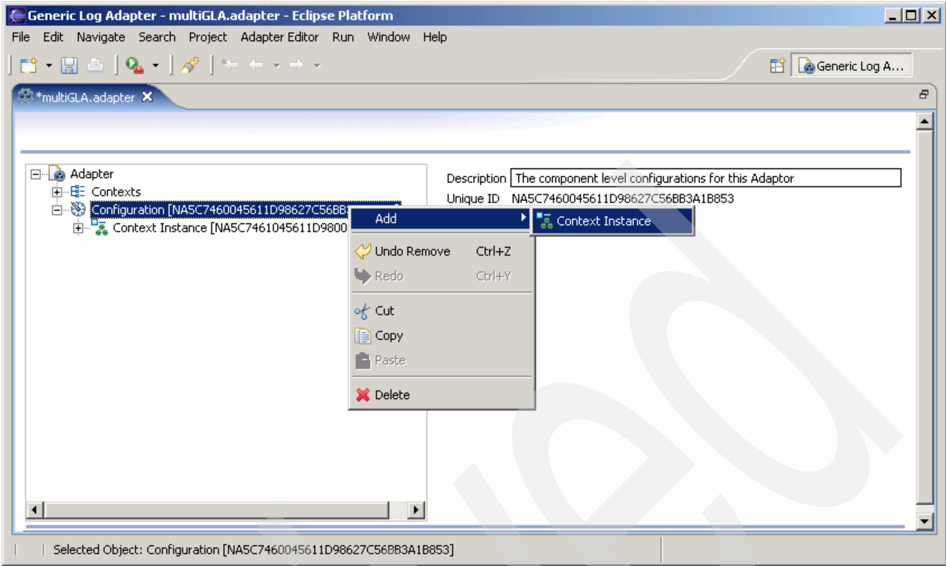


Figure 4-23 Add a new context instance

Next, we need to add the components to the context instance. The five components of the contexts are:

- Sensor
- Extractor
- Parser
- Formatter
- Outputter

To add the sensor component to the context instance, expand **Adapter** → **Configuration**, right-click the newly created context instance, and select **Add** → **Sensor**. This is shown in the Figure 4-24 on page 109.

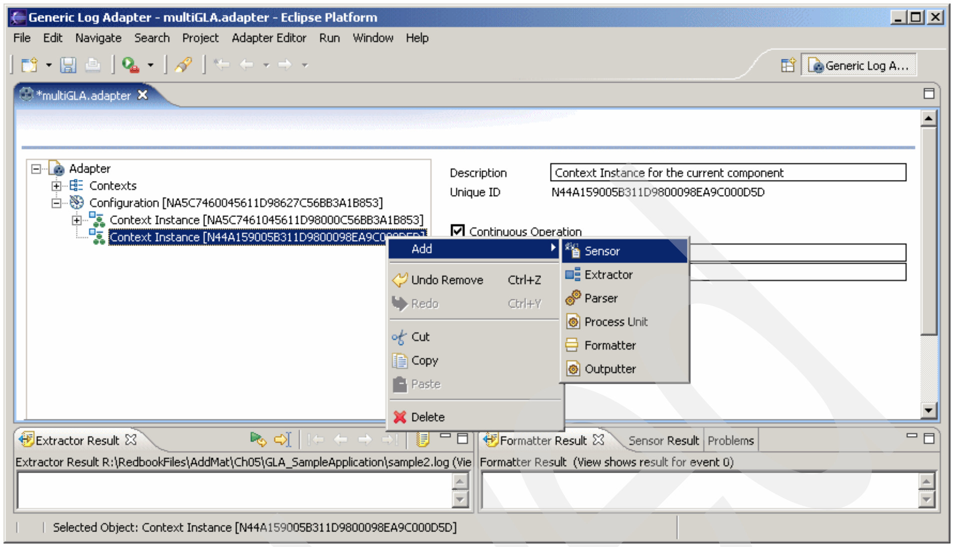


Figure 4-24 Add the component to the context instance

In a similar way, we need to add the other components such as the extractor, parser, formatter, and outputter into the context instance. Figure 4-25 shows the context instance after adding all the necessary components.

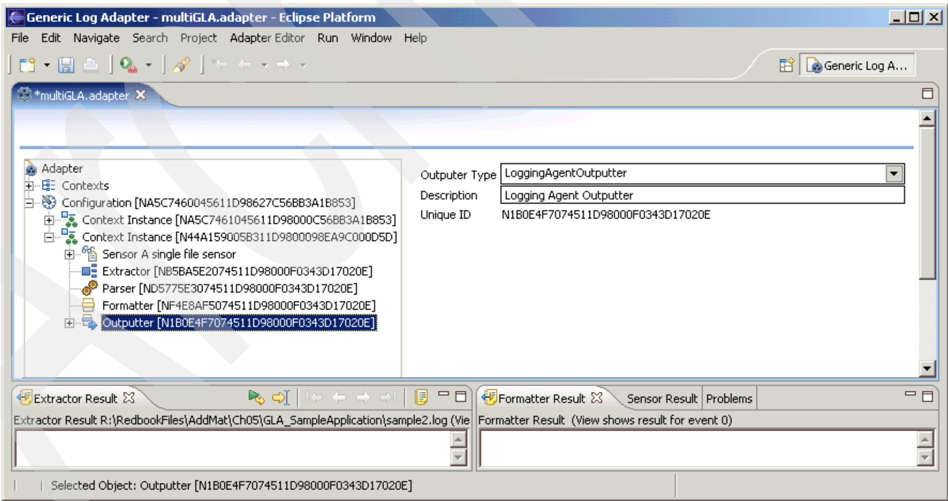


Figure 4-25 Context instance with components

Now, we need to configure the sensor, extractor, parser, and outputter components of the new context instance.

Configuring the sensor

The sensor will read our Sample2Application log file, sample2.log. It implements the SingleFileSensor sensor type with the executable class org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor. To set this, perform the following steps:

1. Expand the **Adapter** → **Configuration**, and then expand the second **Context Instance** → **Sensor**. Click **Property directory** and set the Property Value to the directory where the log exists, for example, `R:\RedbookFiles\AddMat\GLA\SimpleGLAProject`, as shown in Figure 4-26.

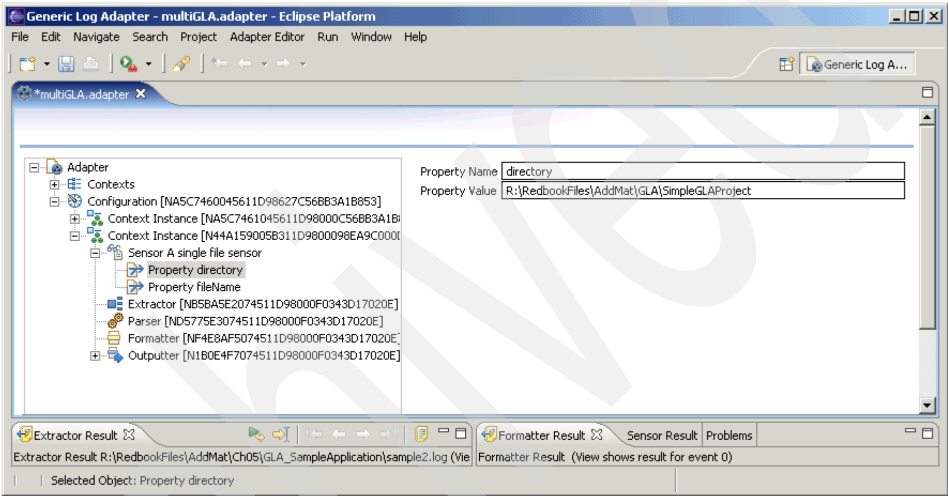


Figure 4-26 Configure the sensor

2. Next, click **Property fileName** and set the Property Value to `sample2.log`.

Configuring the extractor

We need to make sure that the properties of the extractor are set as appropriate. To do this, expand **Adapter** → **Configuration**, and then expand the second **Context Instance** and click **Extractor**. Check the properties **Contains Line Breaks**, **Include Start Pattern**, and **Include End Pattern**. Enter ^ in **Start Pattern** and $ in **End Pattern**, as shown in Figure 4-27 on page 111.

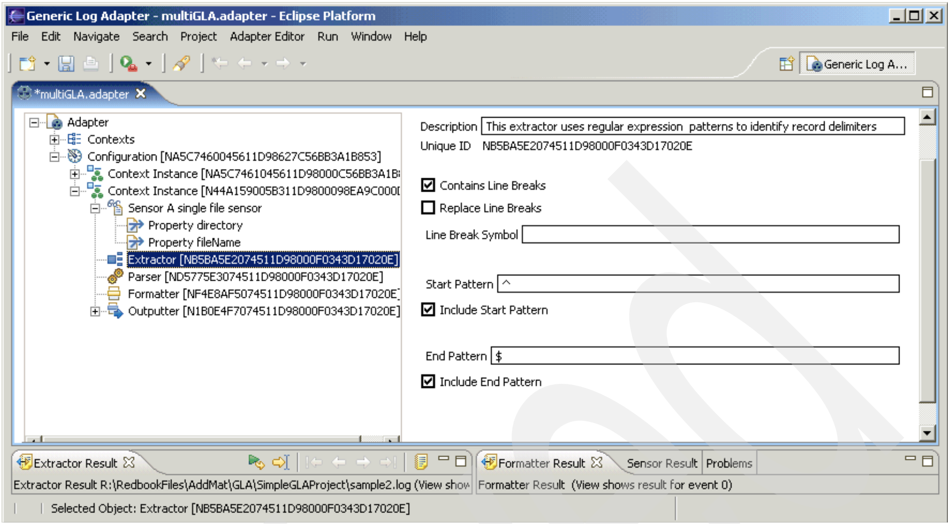


Figure 4-27 Configure the extractor

Configuring the parser

Configuring the parser is a two-step process. First, we need to add the Common Base Event into the parser. Second, we configure the Common Base Event elements. Complete the following steps:

1. To add the Common Base Event into the parser, expand **Adapter** → **Configuration**. Then expand the second **Context Instance**, right-click **Parser**, and select **Add** → **CommonBaseEvent**, as shown in Figure 4-28 on page 112.

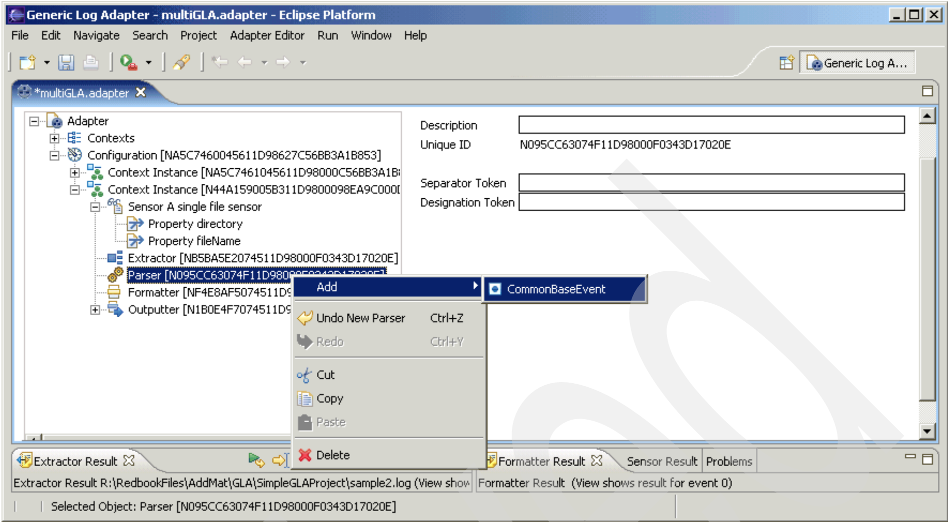


Figure 4-28 Add the Common Base Event in the parser

2. We need to configure the Common Base Event elements now. At a minimum, we need to include the following elements to the Common Base Event:

   – sourceComponentID
   – reporterComponentID
   – creationTime
   – situation

   These elements are defined in the Common Base Event following the instructions provided in 4.2.2, “Configuring the Common Base Event elements” on page 89.

Defining the situation elements

An important step when creating the adapter configuration file is to identify the situation elements for the messages generated by the Sample2Application. The application Sample2Application generates the following log messages:

```
ITSO001I Sample2Application starting...
ITSO026I RMI Connector unavailable at port 2809
ITSO134I File transfer is configured with host=acpd001, port=9090, security
enabled=false
ITSO003I Configuration synchronization completed
ITSO013I SOAP connector available at port 8880
ITSO900I Sample2Application stopping
ITSO901I Sample2Application stopped!!!
```

Table 4-3 on page 113 presents the messages generated by the Sample2Application and the matching situation for these messages.

Table 4-3 Situation types for messages in sample2.log

Log message	Situation
ITSO001 Sample2Application starting	startSituation
ITSO134I File transfer is configured with host=acpd001, port=9090, security enabled=false	configureSituation
ITSO003I Configuration synchronization completed	requestSituation
ITSO0013I SOAP connector available at port 8880 ITSO026I RMI Connector unavailable at port 2809	availableSituation
ITSO900I Sample2Application stopping ITSO901I Sample2Application stopped!!!	stopSituation

Next, we provide a break-down of the situation elements and their respective subelement definitions for the previous messages. Table 4-4 provides the break-down of the situation elements shown in Table 4-3.

Table 4-4 Situation subelements

<table>
<tr><th colspan="5">Subelement definitions</th></tr>
<tr><td colspan="5">Situation: StartSituation</td></tr>
<tr><td>Message</td><td>Reasoning Scope</td><td>SituationQualifier</td><td colspan="2">SuccessDisposition</td></tr>
<tr><td>ITSO001</td><td>INTERNAL</td><td>START INITIATED</td><td colspan="2">SUCCESSFUL</td></tr>
<tr><td colspan="5">Situation: ConfigureSituation</td></tr>
<tr><td>Message</td><td>Reasoning Scope</td><td colspan="3">SuccessDisposition</td></tr>
<tr><td>ITSO134I</td><td>INTERNAL</td><td colspan="3">SUCCESSFUL</td></tr>
<tr><td colspan="5">Situation: RequestSituation</td></tr>
<tr><td>Message</td><td>Reasoning Scope</td><td>SituationQualifier</td><td colspan="2">SuccessDisposition</td></tr>
<tr><td>ITSO003I</td><td>INTERNAL</td><td>REQUEST COMPLETED</td><td colspan="2">SUCCESSFUL</td></tr>
<tr><td colspan="5">Situation: AvailableSituation</td></tr>
<tr><td>Message</td><td>Reasoning Scope</td><td>Processing Disposition</td><td>Operation Disposition</td><td>Availability Disposition</td></tr>
<tr><td>ITSO013I</td><td>EXTERNAL</td><td>FUNCTION_PROCESS</td><td>STARTABLE</td><td>AVAILABLE</td></tr>
<tr><td>ITSO026I</td><td>EXTERNAL</td><td>FUNCTION_BLOCK</td><td>NONSTARTABLE</td><td>NOTAVAILABLE</td></tr>
<tr><td colspan="5">Situation: StopSituation</td></tr>
</table>

Subelement definitions			
Message	**Reasoning Scope**	**SituationQualifier**	**SuccessDisposition**
ITSO900I	INTERNAL	STOP INITIATED	SUCCESSFUL
ITSO901I	INTERNAL	STOP COMPLETED	SUCCESSFUL

Table 4-4 on page 113 is used as a reference to create all situation elements for the Common Base Event. All the situation elements are created for the Sample2Application following the steps described in “Defining the situation element” on page 95.

Figure 4-29 shows the Common Base Event elements defined for Sample2Application.

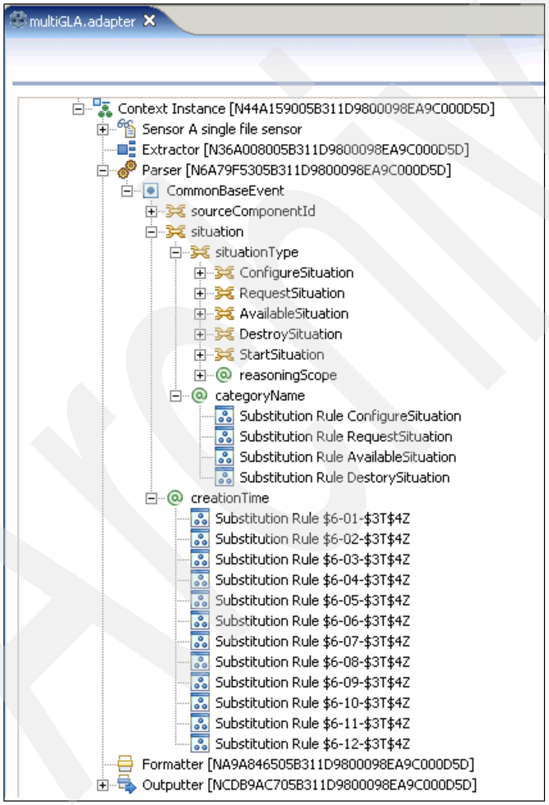


Figure 4-29 Common Base Elements set on context instance 2

Configuring the outputter in the context instance

As shown in Figure 4-22 on page 106, it is our intention to have this context instance storing Common Base Events in the same log file (CBEOut.log) as the previously created context instance. We configured the second context instance outputter in a similar way as the first context outputter, as described in 4.2.3, "Configuring the adapter outputter" on page 101.

4.3.2 Adding a second context to the adapter file

In the previous section, we created and configured a context instance. The next step is to create a context that will be associated to the new context instance.

In the example presented in this chapter, both contexts defined in the adapter configuration file need to be configured to use the same output file. Therefore, the outputter of both the contexts needs to use the CBEConvergentFileOutputter class. The executable class is org.eclipse.hyades.logging.adapter.outputters.CBEConvergentFileOutputter.

An easy way to define a a new context in our case is to make an exact copy of the existing context. Perform the following steps to configure the existing context to use the CBEConvergentFileOutputter class and to create the second context in the adapter file by copying the existing context:

1. First, we need to change the outputter class of the context to use the CBEConvergentFileOutputter class. Expand **Adapter** → **Contexts** → **Context Basic Context Implementation** and click the **Single File Outputter** component. In the details pane on the right, as shown in Figure 4-30 on page 116, change the Executable Class to use `org.eclipse.hyades.logging.adapter.outputters.CBEConvergentFileOutputter`.

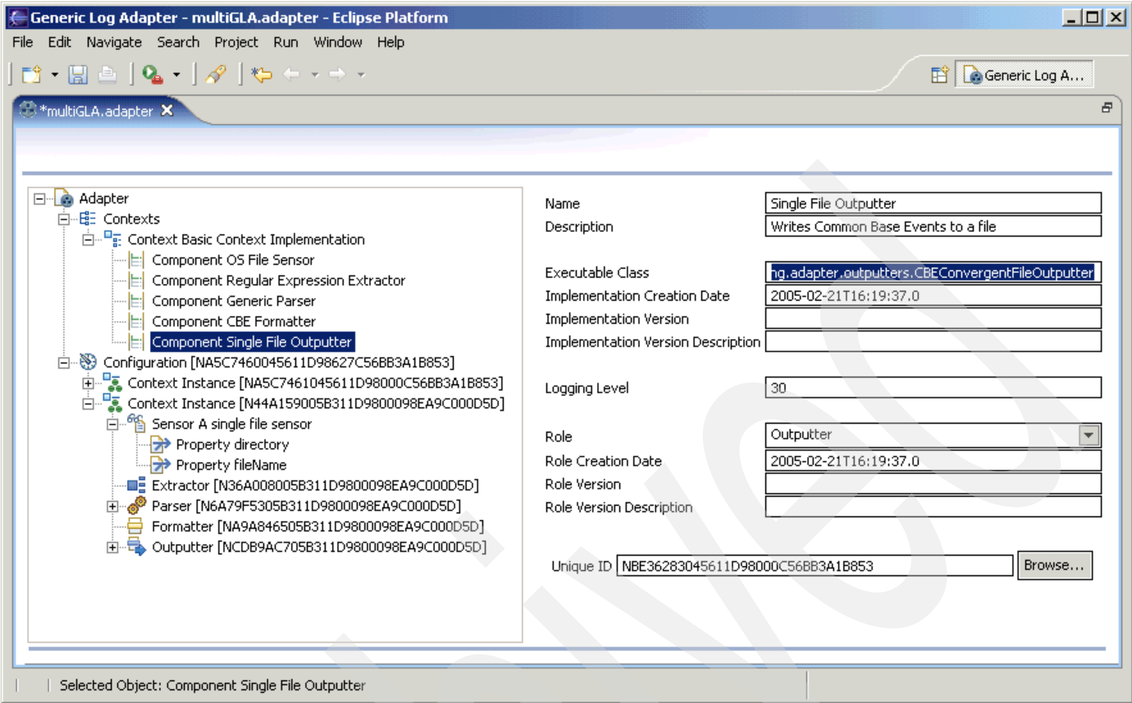


Figure 4-30 Modify context outputter to write multiple contexts to single file

2. Next, right-click **Context Basic Context Implementation** and select **Copy**, as shown in Figure 4-31 on page 117.

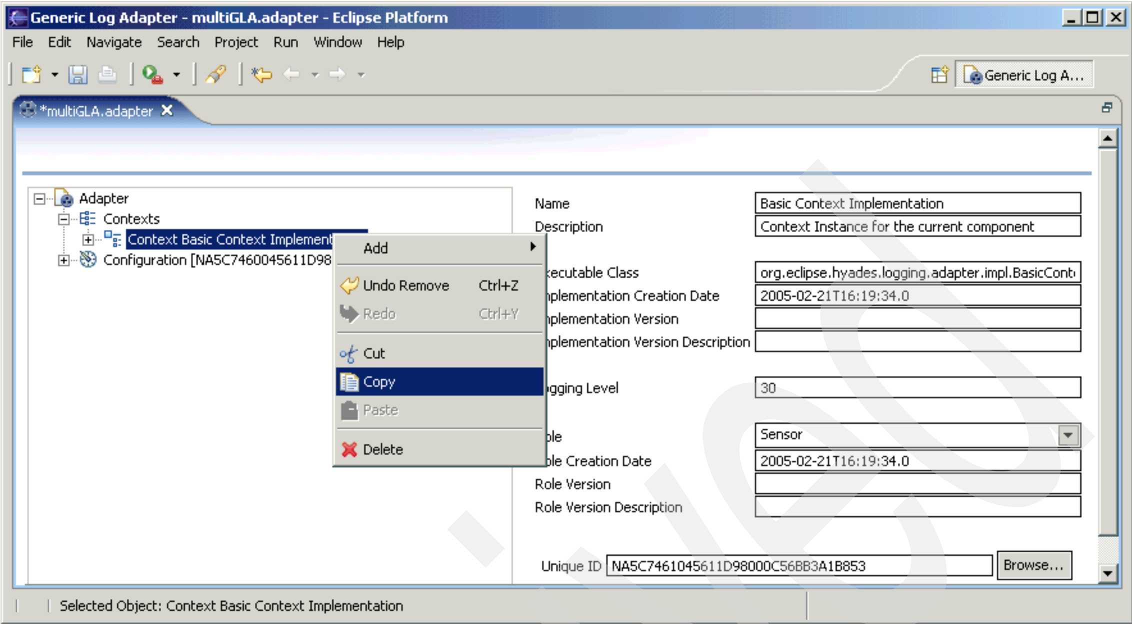


Figure 4-31 Create a new context

3. Right-click **Contexts** and select **Paste**.
4. Save the adapter configuration file. A new context is now added to the adapter file.

4.3.3 Associating the new context components

The next step is to associate the new context and each of its components with the respective elements defined in the context instance.

The association is done by the unique ID information of each element. For example, the unique ID of the context element must be associated to the unique ID of the context instance.

As for the various context components (sensor, extractor, parser, formatter, and outputter), their unique ID must be associated to the respective elements defined in the context instance (sensor, extractor, parser, formatter, and outputter). For example, the unique ID of the OS File Sensor component defined in the context, must be associated to the unique ID of the sensor element defined in the context instance.

In order to perform the associations, complete the following steps:

1. Expand and select the newly created **Context**.

2. In the details pane on the right, click the **Browse** button next to the unique ID information.

3. In the pop-up window, expand **Adapter**. Then expand **Configuration**, click the second **Context Instance**, and click **OK**.

   This is shown in Figure 4-32.

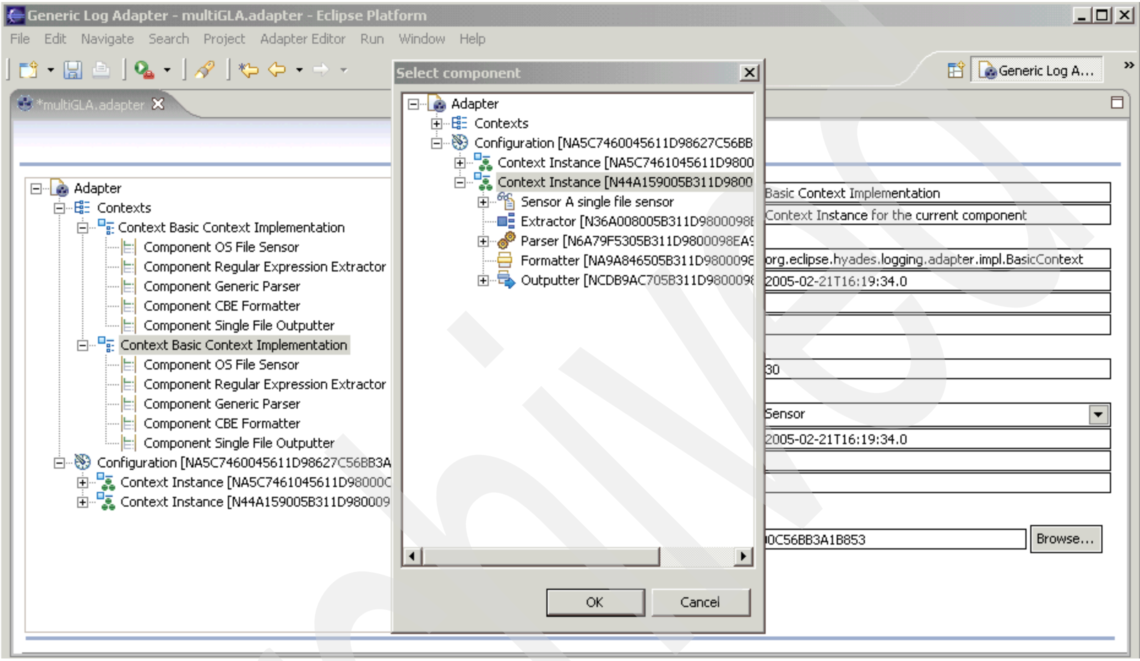


Figure 4-32 Association of context components to context instance elements

4. In a similar way as just shown, associate all the context components to its associated element in the context instance under the Configuration. Now, save the adapter file by clicking **File** → **Save**.

Restriction: The associated unique ID of the second context will not be saved when using the Generic Log Adapter Configuration Editor tool provided by Generic Log Adapter Version 3.0 shipped with the IBM Autonomic Computing Toolkit Version 2.0.

Therefore, we need to manually edit the adapter file to perform associations of context with the unique ID of context instance and its elements. Perform the following steps to manually edit the adapter configuration file:

1. Open the adapter configuration file in an external text editor.
2. Copy the unique ID of the second context instance and paste it in the unique ID of the second context, as shown in Example 4-1.
3. Make sure that all the component's unique IDs of the second context match the associated context instance element's unique ID and save the file.

Example 4-1 Associating context and context instance unique IDs

```
<-- ADAPTER CONFIGURATION FILE HEADING -->
<adapter:Adapter
xmlns:adapter="http://www.eclipse.org/hyades/schema/Adapter.xsd"
xmlns:cc="http://www.eclipse.org/hyades/schema/ComponentConfiguration.xsd"
xmlns:ex="http://www.eclipse.org/hyades/schema/Extractor.xsd"
xmlns:fmt="http://www.eclipse.org/hyades/schema/Formatter.xsd"
xmlns:hga="http://www.eclipse.org/hyades/schema/Context.xsd"
xmlns:op="http://www.eclipse.org/hyades/schema/Outputter.xsd"
xmlns:parser="http://www.eclipse.org/hyades/schema/Parser.xsd"
xmlns:pu="http://www.eclipse.org/hyades/schema/ProcessUnit.xsd"
xmlns:sensor="http://www.eclipse.org/hyades/schema/Sensor.xsd">
............................
............................

<-- ADAPTER CONFIGURATION - FIRST CONTEXT DEFINITION -->
   <hga:Context description="Context Instance for the current component"
executableClass="org.eclipse.hyades.logging.adapter.impl.BasicContext"
implementationCreationDate="2005-02-21T16:19:34.0" loggingLevel="30"
name="Basic Context Implementation" role="context"
roleCreationDate="2005-02-21T16:19:34.0"
uniqueID="NA5C7461045611D98000C56BB3A1B853">
............................
............................
   </hga:Context>

<-- ADAPTER CONFIGURATION - SECOND CONTEXT DEFINITION -->
   <hga:Context description="Context Instance for the current component"
executableClass="org.eclipse.hyades.logging.adapter.impl.BasicContext"
```

```
implementationCreationDate="2005-02-21T16:19:34.0" loggingLevel="30"
name="Basic Context Implementation" role="context"
roleCreationDate="2005-02-21T16:19:34.0"
uniqueID="N44A159005B311D9800098EA9C000D5D">
.............................
.............................
      </hga:Context>
.............................
.............................

<-- ADAPTER CONFIGURATION - FIRST CONTEXT INSTANCE DEFINITION -->
   <cc:ContextInstance continuousOperation="true" description="Context
Instance for the current component" maximumIdleTime="20000" pauseInterval="10"
uniqueID="NA5C7461045611D98000C56BB3A1B853">
.............................
.............................
      </cc:ContextInstance>

<-- ADAPTER CONFIGURATION - SECOND CONTEXT INSTANCE DEFINITION -->
      <cc:ContextInstance continuousOperation="true" description="Context
Instance for the current component" maximumIdleTime="20000" pauseInterval="10"
uniqueID="N44A159005B311D9800098EA9C000D5D">
.............................
.............................
```

4.3.4 Testing the multiple context instance adapter file

The Generic Log Adapter perspective within Eclipse enables us to test the adapter configuration file prior to deployment. In the case of multiple instances' definitions, each context instance must be tested individually. Select the appropriate **Context Instance** in the editor and click the **Rerun adapter** icon, as shown in Figure 4-20 on page 104. Navigate through the Events using the Extractor and Formatter Result panels to ensure that the parsing is being done correctly. If the result is incorrect, change the substitution rule, save the file, and rerun the adapter.

Also, verify that the outputters defined in all the context instances are generating the Common Base Event in the correct format. In our case, all context instances' outputters are logging Common Base Events in a single CBEOut.log file in the predefined directory.

For additional information about how to test the adapter configuration file, refer to 4.2.4, "Testing the adapter configuration file" on page 103.

4.4 Deploying and running the adapter configuration file

After testing the adapter configuration file, we are ready to deploy it. We need to make sure that the adapter configuration file is pointing to the log and output files at the correct locations on the machine where it will run. After making sure of the input and output file locations, we are ready to deploy the adapter configuration file.

> **Tip:** If the adapter configuration file is not pointing to the real log, we need to modify the *sensor* element in the configuration file to point to the actual location of the log file on the machine where it will run, instead of the template log file location we used while authoring. Also, if the adapter configuration file is not pointing to the real output file, in the case of using a file as an outputter, we need to edit the element singleFileOutputter to point to the actual location of the output file on the machine where it will run. This can be done in the Generic Log Adapter Configuration Editor, or by using any text editor, because the adapter configuration file is an XML file.

Deploy the adapter configuration file by copying it to the machine that will run the Generic Log Adapter Runtime. We can use a command-line argument, or we can write a small Java class to start an instance of the Generic Log Adapter Runtime. We also need to pass the adapter configuration file to the created instance of the Generic Log Adapter Runtime.

> **Restriction:** When the Generic Log Adapter Runtime is continuously monitoring a log file, it will not process the last record if there is no end pattern specified. The last record will only be processed when the Generic Log Adapter Runtime is shutting down. To ensure that the last record is processed during monitoring, provide an end pattern for the records in the adapter configuration file.

4.4.1 Starting the Generic Log Adapter Runtime from a Java class

We can use the org.eclipse.hyades.logging.adapter.Adapter class directly by creating an instance of it to start the Generic Log Adapter engine. Example 4-2 on page 122 describes how to write a small program to launch the Generic Log Adapter.

The following JAR files need to be included in the CLASSPATH before launching the Generic Log Adapter engine:

- hlcbe101.jar
- ecore.jar
- common.jar

- hlcore.jar
- hgla.jar
- glaconfigs.jar
- hexr.jar
- hglaconfig.jar
- hparse.jar
- logparsers.jar

Example 4-2 Generic Log Adapter launcher Java class

```
package com.ibm.itso.sg246665.glarunner;

import org.eclipse.hyades.logging.adapter.Adapter;
import org.eclipse.hyades.logging.adapter.AdapterException;
import java.io.*;

public class LaunchAdapter {

  public static void main(String[] args) {

    if (args.length != 1) {
      System.out.println("Please identify the adapter file."
          + " The correct syntax is:\n");
      System.out.println("java -jar glarunner.jar "
          + "\"c:\\Program Files\\sample.adapter\"");
    } else {
      String adapterFile = args[0];

      boolean separateThread = false;
      boolean daemon = false;

      Adapter glaadapter = new Adapter();
      try {
        glaadapter.setComponentConfigPath(adapterFile);
        glaadapter.setContextConfigPath(adapterFile);
        System.out.println("GLA Engine Run Time ...");
        glaadapter.start(separateThread, daemon);
        // To stop the thread
        new StopGLAThread().start();
      } catch (AdapterException ex) {
        ex.printStackTrace();
      }
    }
  }
}

class StopGLAThread extends Thread {
  String stopfile = "stop_gla";
```

```
  // This method is called when the thread runs

  public void run() {
    while (true) {
      boolean exists = (new File(stopfile)).exists();
      try {
        sleep(1000);
      } catch (InterruptedException e) {
        System.out.println(e.toString());
      }
      if (exists) {
        (new File(stopfile)).delete();
        System.exit(0);
      }
    }
  }
}
```

4.4.2 Starting the Generic Log Adapter Runtime from CLI

The Generic Log Adapter Runtime can be started through a command-line interface (CLI) using a batch file or shell script file located at <GLAinstall>\bin\gla.bat (or gla.sh), where <GLAinstall> is the location of the Generic Log Adapter installation.

The gla.bat script uses the org.eclipse.hyades.logging.adapter.Adapter class, which comes with its own main method to start the Generic Log Adapter engine. The gla.bat syntax is as follows:

```
<GLAinstall>\bin\gla.bat <adapter_name>
```

Where <adapter_name> is the file name of the adapter configuration file and must include the absolute path to the configuration file, for example, gla.bat c:\myadapter\simpleGLA.adater.

When Generic Log Adapter executes, it logs messages and errors to a file called hgla.log. The hgla.log file will be created in the directory where the Generic Log Adapter Runtime is executing. The Generic Log Adapter Runtime logs events with the logging levels 10, 20, 30, 50, and 60 defined by the Common Base Event model. The Generic Log Adapter can be configured to log specific levels of messages by specifying the minimum level in the adapter file. Each context and component within a context can be configured with a different logging level. The logging level can be changed in the adapter configuration file using the Generic Log Adapter Configuration Editor. Open the adapter file in the editor and expand **Adapter → Contexts**. Select a Context or Component and change the Logging

Level value in the details pane for that element. The value represents the minimum level of messages that the Generic Log Adapter Runtime will log.

The hgla.log file can be used to diagnose problems encountered by the Generic Log Adapter Runtime when parsing the log file.

4.5 Writing a custom outputter

This section describes how to create a custom outputter for the adapter configuration file. We develop an outputter that sends Common Base Events using Web services. These events will be consumed by a thin layer of code running on IBM WebSphere Application Server. The custom outputter shown in this section is used in the case study scenario presented later in this book. For details about how the custom outputter will be deployed, refer to Chapter 9, “Problem Determination scenario two” on page 305.

The Generic Log Adapter uses an outputter class to handle the generated Common Base Event from the log file. The outputter is responsible for delivering the Common Base Event in an expected format. For example, some of the standard outputters write the Common Base Event in an XML format to a log.

The custom outputter’s class must extend the org.eclipse.hyades.logging.adapter.impl.ProcessUnit class and implement the org.eclipse.hyades.logging.adapter.IOutputter interface. The important method in the API is the processEventItems() method. It receives an array of objects, usually CommonBaseEvent objects, and processes them. The signature of the method is:

```
public Object[] processEventItems(Object[] msgs)
```

In our case study, we are creating a custom outputter, named ITSOSHIMOutputter, that writes the Common Base Events into Web services. Figure 4-33 on page 125 describes the scenario where the Generic Log Adapter reads a log file and using the ITSOSHIMOutputter writes into a Web service component called ITSOSHIM.

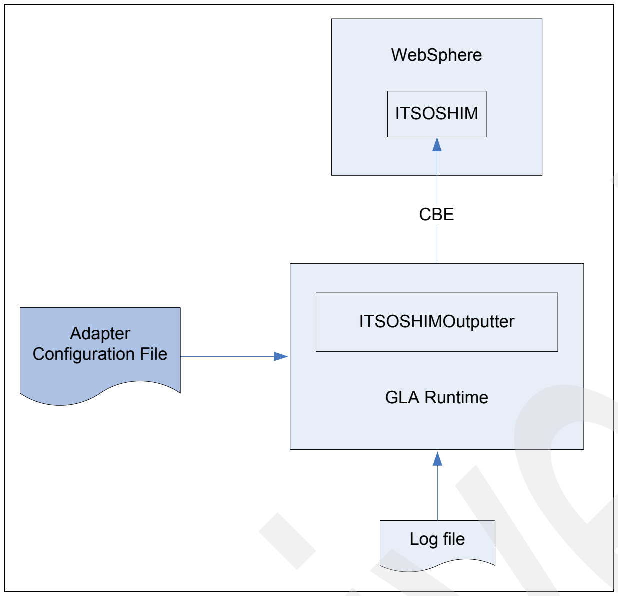


Figure 4-33 The ITSOSHIMOutputter scenario

The following sections describe the creation of a custom outputter named ITSOSHIMOutputter and its usage and testing using the Generic Log Adapter.

4.5.1 Creating a custom outputter named ITSOSHIMOutputter

In this example, we create the outputter that will write to a Web service.

The following step-by-step process describes how to create a custom outputter named ITSOSHIMOutputter.

First, we need to define a plug-in to define the custom outputter. This plug-in will be required by the Generic Log Adapter Runtime when executing the adapter. Perform the following steps:

1. Create a plug-in project using Eclipse. Select **File** → **New** → **Project**, expand **Plug-in Development**, and select **Plug-in Project**. Click **Next**.
2. In the next window, enter `GLA_CustomOutputter` as the Project name and click **Next**.
3. Now, provide the Plug-in ID, Plug-in Name, and Runtime Library values, as shown in Figure 4-34 on page 126. Also clear the **Generate the Java class that controls the plug-in's life cycle** option. Click **Finish**.

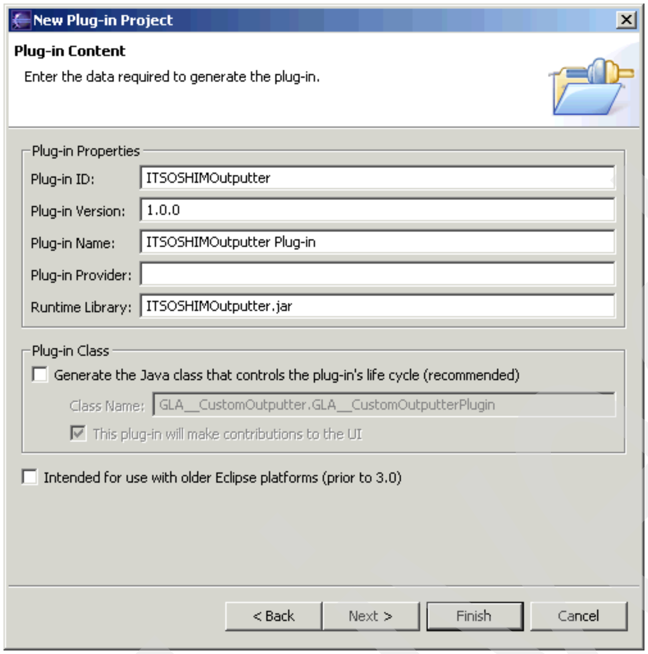


Figure 4-34 Create the ITSOSHIMOutputter plug-in project in Eclipse

Now, the project appears in the plug-in development perspective, as shown in Figure 4-35 on page 127.

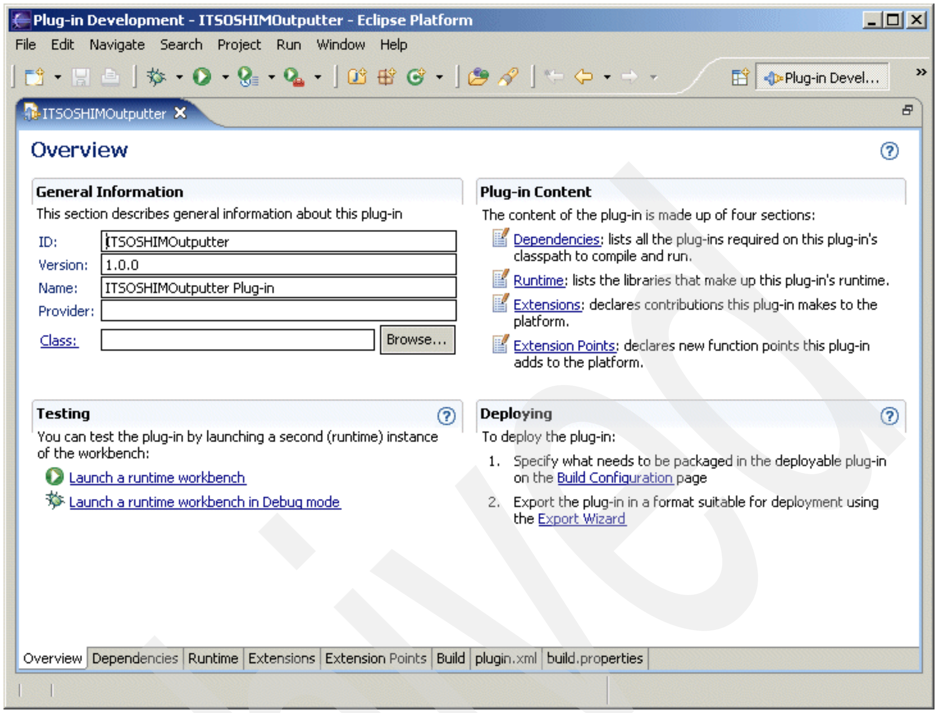


Figure 4-35 ITSOSHIMOutputter in plug-in development perspective

4. Select the **Dependencies** tab and click **Add**. Now, select the following plug-ins as dependencies:

   – org.eclipse.core.resources
   – org.eclipse.ui
   – org.eclipse.hyades.logging.adapter
   – org.eclipse.hyades.logging.core

   Click **OK**. Figure 4-36 on page 128 shows the dependencies added to the plug-in.

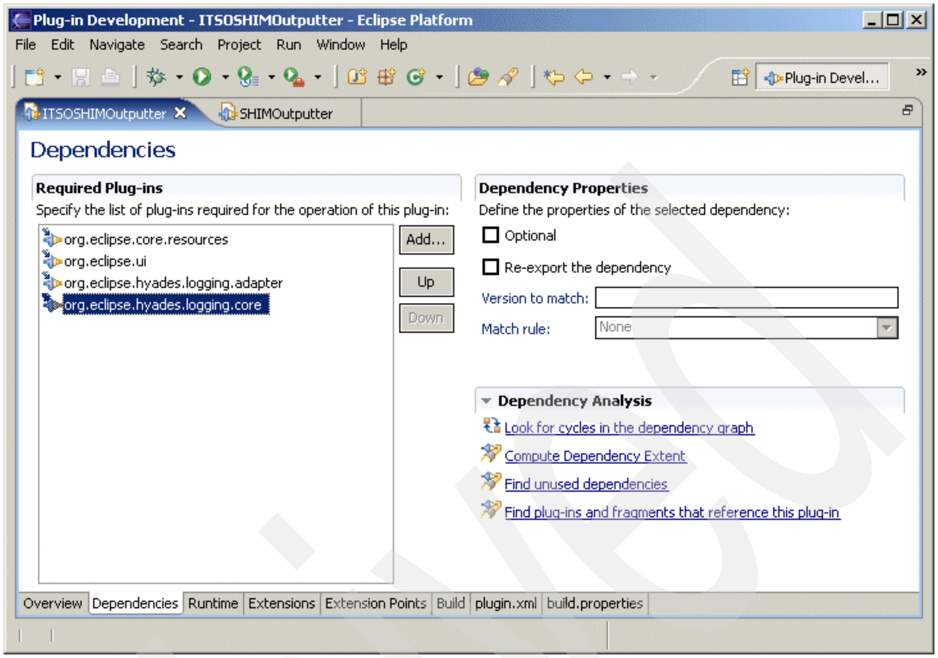


Figure 4-36 Add Dependencies to the plug-in

5. Select the **plugin.xml** tab to view the XML that defines the plug-in and to edit the XML file. Example 4-3 shows the XML for the ITSOSHIMOutputter plug-in. Note the extension tag added to the plugin.xml file. The extension tag specifies a class definition that will be used by the custom outputter defined in this section.

   The changes are highlighted in bold in Example 4-3.

Example 4-3 The plugin.xml file created for ITSOSHIMOutputter

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="ITSOSHIMOutputter"
   name="ITSOSHIMOutputter Plug-in"
   version="1.0.0"
   provider-name="">

   <runtime>
      <library name="ITSOSHIMOutputter.jar">
         <export name="*"/>
      </library>
   </runtime>
```

```
  <requires>
     <import plugin="org.eclipse.core.resources"/>
     <import plugin="org.eclipse.ui"/>
     <import plugin="org.eclipse.hyades.logging.adapter"/>
     <import plugin="org.eclipse.hyades.logging.core"/>
  </requires>
  <extension
      point="org.eclipse.hyades.logging.adapter.adapterComponent">
      <componentClassname
           name="com.ibm.itso.sg246665.outputters.ITSOSHIMOutputter">
      </componentClassname>
  </extension>

</plugin>
```

The next step is to create the implementation of the class defined during the plug-in definition. The class must extend the org.eclipse.hyades.logging.adapter.impl.ProcessUnit class and implement the org.eclipse.hyades.logging.adapter.IOutputter interface. To create the com.ibm.itso.sg246665.outputters.ITSOSHIMOutputter Java class, perform the following steps:

1. Right-click the newly created project named **GLA_CustomOutputter** and select **New** → **Class**.
2. Fill out the information for the definition of the class, as shown in Figure 4-37 on page 130. Note that the Package name must match the name of the class defined in the plugin.xml file. Also, the Superclass must be set to `org.eclipse.hyades.logging.adapter.impl.ProcessUnit`, and Interfaces must be set to `org.eclipse.hyades.logging.adapter.IOutputter`.

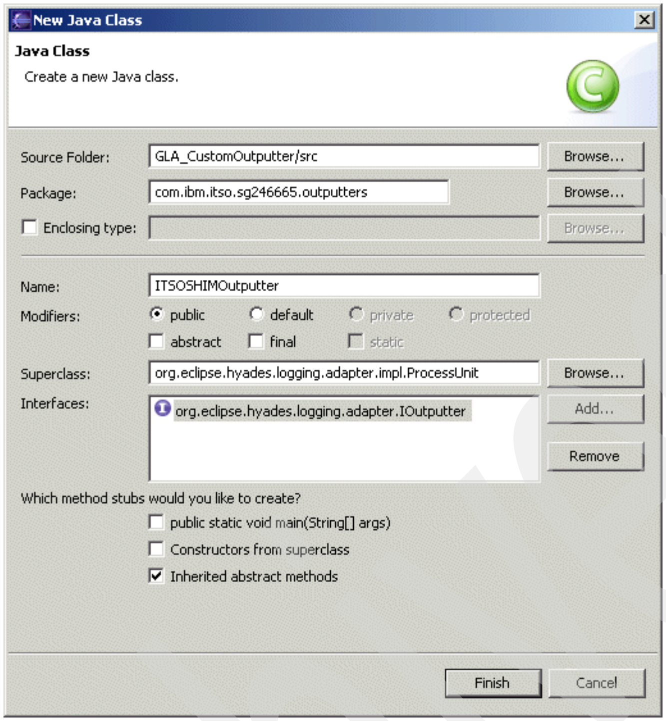


Figure 4-37 Create the ITSOSHIMOutputter class

This procedure creates a skeleton class that we need to modify to process the Common Base Events and write each event to the Web services.

We developed the ITSOSHIMOutputter Java class as follows:

- Common Base Events are processed by a method named processCBEs().
- The processCBEs method uses a Java class named ITSOSHIMProxy to send Common Base Events to Web services.

Example 4-4 on page 131 shows the complete ITSOSHIMOutputter Java class.

Example 4-4 ITSOSHIMOutputter Java class listing

```
/*
 * Created on Mar 17, 2005
 */
package com.ibm.itso.sg246665.outputters;

import java.net.MalformedURLException;
import java.net.URL;

import org.eclipse.hyades.logging.adapter.IOutputter;
import org.eclipse.hyades.logging.adapter.impl.ProcessUnit;

import org.eclipse.hyades.logging.events.cbe.CommonBaseEvent;


/**
 * @author ITSO Redbook Team
 *
 */
public class ITSOSHIMOutputter extends ProcessUnit implements IOutputter {

  private int ctr = 0;

  public Object[] processEventItems(Object[] msgs) {
    if (msgs instanceof CommonBaseEvent[]) {
      return processCBEs((CommonBaseEvent[]) msgs);
    }

    return null;
  }

  private CommonBaseEvent[] processCBEs(CommonBaseEvent[] events) {
    try {
      for (int i = 0; i < events.length; ++i) {
        CommonBaseEvent ev = events[i];
        if (ev != null) {
          ++ctr;
          //add ITSOSHIMAPI here to write the output to ITSOSHIM component
          ITSOSHIMProxy proxy = new ITSOSHIMProxy();
          // add the URL provided by the ITSOSHIM component
          URL url = new URL(

"http://acpd003.itsc.austin.ibm.com:80/awp.server.autonomic.cei.ws/servlet/rpcr
outer");
          proxy.setEndPoint(url);
          proxy.sendEvent(ev);
          System.out.println("Event Sent");
```

```
        }
      }
    } catch (MalformedURLException e) {

      e.printStackTrace();
    } catch (Exception e) {

      e.printStackTrace();
    }

    return events;
  }

}
```

The ITSOSHIMProxy Java class performs the following tasks:

1. Receives the Common Base Event from the ITSOSHIMOutputter Java class.
2. Converts the Common Base Event into XML format.
3. Makes SOAP calls to the ITSOSHIM component using the sendEvent method.

Example 4-5 shows the ITSOSHIMProxy Java class.

Example 4-5 ITSOSHIMProxy Java class listing

```
/*
 * Created on Mar 17, 2005
 *
 * Author : IBM Redbook Team
 */

package com.ibm.itso.sg246665.outputters;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Vector;

import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.Header;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.transport.http.SOAPHTTPConnection;
import org.eclipse.hyades.logging.events.cbe.CommonBaseEvent;
```

```
import com.ibm.rcp.autonomic.events.EventUtils;

public class ITSOSHIMProxy  {
    private Call call;

    private URL url = null;

    private String stringURL =
"http://localhost:9080/awp.server.autonomic.cei.ws/servlet/rpcrouter";

    private java.lang.reflect.Method setTcpNoDelayMethod;

    private Header _header;

    public ITSOSHIMProxy() {
        try {
            setTcpNoDelayMethod = SOAPHTTPConnection.class.getMethod(
                    "setTcpNoDelay", new Class[] { Boolean.class });
        } catch (Exception e) {
        }
        call = createCall();
    }

    public synchronized void setEndPoint(URL url) {
        this.url = url;
    }

    public synchronized URL getEndPoint() throws MalformedURLException {
        return getURL();
    }

    private URL getURL() throws MalformedURLException {
        if (url == null && stringURL != null && stringURL.length() > 0) {
            url = new URL(stringURL);
        }
        return url;
    }

    public synchronized void sendEvent_(org.w3c.dom.Element cbeXML)
            throws Exception {
        String targetObjectURI =
"http://tempuri.org/com.ibm.rcp.autonomic.cei.ws.CeiShim";
        String SOAPActionURI = "";

        if (getURL() == null) {
            throw new SOAPException(Constants.FAULT_CODE_CLIENT,
                    "A URL must be specified via
ITSOSHIMProxy.setEndPoint(URL).");
```

```
        }

        call.setMethodName("sendEvent");
        if (_header != null) {
            call.setHeader(_header);
        }
        call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
        call.setTargetObjectURI(targetObjectURI);
        Vector params = new Vector();
        Parameter cbeXMLParam = new Parameter("cbeXML",
                org.w3c.dom.Element.class, cbeXML,
Constants.NS_URI_LITERAL_XML);
        params.addElement(cbeXMLParam);
        call.setParams(params);
        Response resp = call.invoke(getURL(), SOAPActionURI);

        //Check the response.
        if (resp.generatedFault()) {
            Fault fault = resp.getFault();
            call.setFullTargetObjectURI(targetObjectURI);
            throw new SOAPException(fault.getFaultCode(), fault
                    .getFaultString());
        }
    }

    public void setHeader(Header header) {
        _header = header;
    }

    public synchronized void sendEvent(org.w3c.dom.Element cbeXML)
            throws Exception {

        // delegate to method sendEvent_
        sendEvent_(cbeXML);

        return;
    }

    protected Call createCall() {
        SOAPHTTPConnection soapHTTPConnection = new SOAPHTTPConnection();
        if (setTcpNoDelayMethod != null) {
            try {
                setTcpNoDelayMethod.invoke(soapHTTPConnection,
                        new Object[] { Boolean.TRUE });
            } catch (Exception ex) {
            }
        }
        Call call = new Call();
        call.setSOAPTransport(soapHTTPConnection);
```

```
            SOAPMappingRegistry smr = call.getSOAPMappingRegistry();
            return call;
        }


        public void sendEvent(CommonBaseEvent arg0) throws Exception {
            // convert the cbe to an XML element and then send that out using the
            // primary method
            sendEvent(EventUtils.cbe2Element(arg0));
        }
    }
```

4.5.2 Testing the custom outputter

To test the custom outputter, we need to create an adapter configuration file that specifies the executable class of the custom outputter.

In order to test our ITSOSHIMOutputter, we created an adapter configuration file and configured the outputter component with com.ibm.itso.sg246665.outputters.ITSOSHIMOutputter.

Figure 4-38 shows the adapter configuration file using the ITSOSHIMOutputter.

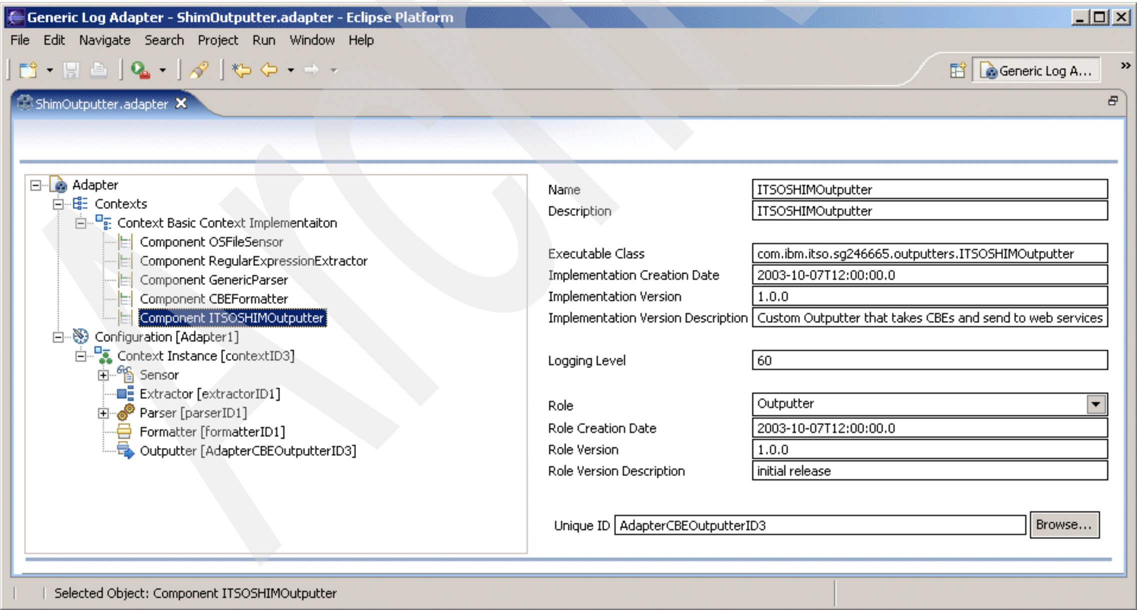


Figure 4-38 Adapter configuration file using ITSOSHIMOutputter

The Outputter element in context instance does not require any properties. The Output Type is set to undeclared, as shown in Figure 4-39.

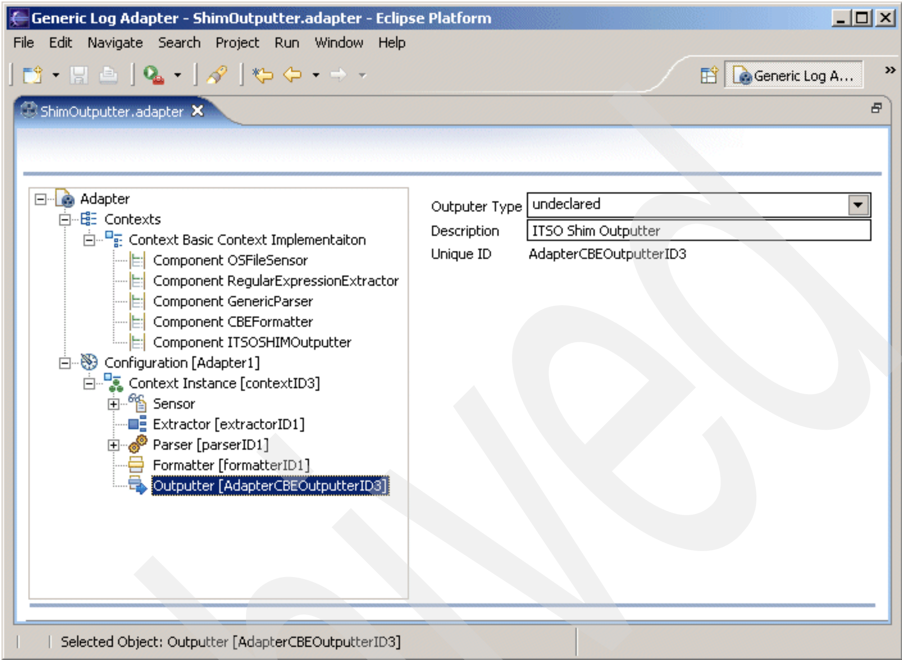


Figure 4-39 Context Instance Outputter element for ITSOSHIMOutputter

After the adapter configuration file is created, it needs to be deployed using one of the methods described in 4.4, “Deploying and running the adapter configuration file” on page 121.

Due to the ITSOSHIMOutputter custom outputter functionality, it also requires the following libraries in addition to the libraries required by the Generic Log Adapter Runtime. The following JAR files must be included in the CLASSPATH to run the Generic Log Adapter Runtime to run the adapter file using the ITSOSHIMOutputter. We include these libraries in the examples provided with this book. For details, refer to Appendix A, “Additional material” on page 379.

- awp-common.jar
- j2ee.jar
- mail.jar
- soap.jar

> **Note:** The libraries listed in this section are required for the ITSOSHIMOutputter. Make sure to include all the libraries required for your custom outputter in the CLASSPATH of the Generic Log Adapter Runtime.

5

Log and Trace Analyzer

The Log and Trace Analyzer (LTA) is used as a maturity level 2 autonomic manager. It facilitates the viewing, analysis, and correlation of log files. These log files can be produced by one component or multiple components in the system. It enables an administrator to analyze the correlation between events created by different products, link them, and look for possible solutions.

We discuss the following topics in this chapter:

- Log and Trace Analyzer overview
- Using Agent Controller for monitoring remote logs
- Creating a custom log parser for the Log and Trace Analyzer
- Creating a custom correlator for the Log and Trace Analyzer

5.1 Log and Trace Analyzer overview

The Log and Trace Analyzer (LTA) enables viewing, analysis, and correlation of log files. An administrator can analyze the correlation between events created by different products, link them, and look for possible solutions. This tool uses data in the Common Base Event format and provides the log visualization and analysis of the data. Refer Chapter 3, “Common Base Events” on page 35 for more information about Common Base Events.

The Log and Trace Analyzer contains a log analysis engine and a correlation engine.

The log analysis engine takes an event that is recorded in a log file, matches this event based on predefined rules against the symptom database, and returns the solutions and directives that match the symptom.

The correlation engine has the capability of correlating the time stamp and record ID of different log files. It is also possible to implement a custom correlation engine with LTA. For more informations about writing a custom correlation engine, refer 5.4, “Creating a custom correlator for the Log and Trace Analyzer” on page 179.

The IBM Autonomic Computing Toolkit provides Log and Trace Analyzer Version 3.0.0 and contains the Log and Trace Analyzer plug-ins. Refer to the *Log and Trace Analyzer for Autonomic Computing V3.0.0 Installation Guide*, which is shipped with the LTA installation media, for details about the LTA installation.

The Log and Trace Analyzer includes the following major features:

- Viewing the log files
- Analyzing the log files using the symptom database
- Correlating the log files

The following sections describe these features of the Log And Trace Analyzer.

5.1.1 Viewing the log files

The Log View shows the log records of an imported log. The properties and values of each log record are shown in this view.

To import and view the log files in Log and Trace Analyzer, complete the following steps:

1. Start Eclipse.

2. We can use an existing project or create a new project by opening a simple project by selecting **File** → **New** → **Project** → **Simple** → **Project**. Give the project a name, and click **Finish**.
3. Import the log file by selecting **File** → **Import** → **Log File**. Click **Next**, as in Figure 5-1.

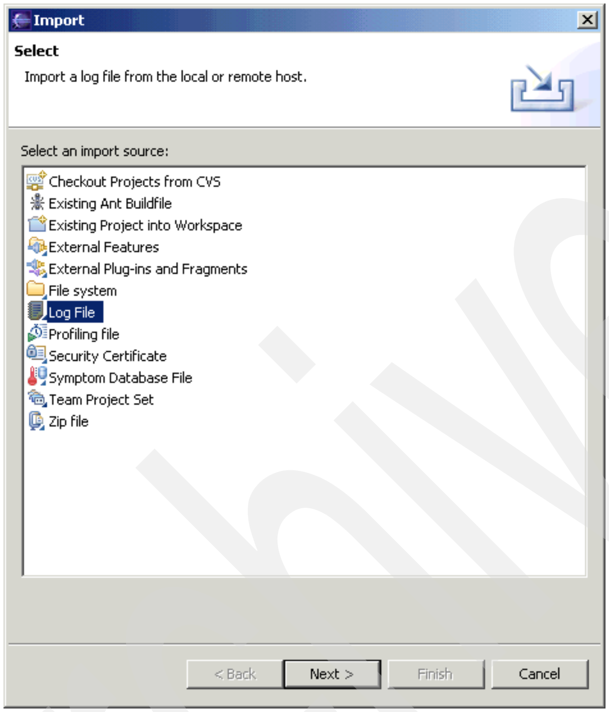


Figure 5-1 Import a log file

4. Click **Add** to add the log file.

5. Choose the log file type, for example, **IBM DB2 Universal Database diagnostic log**, as shown in Figure 5-2. The Host name in the properties of the log file here is `localhost`, because we are viewing the log files in the local system.

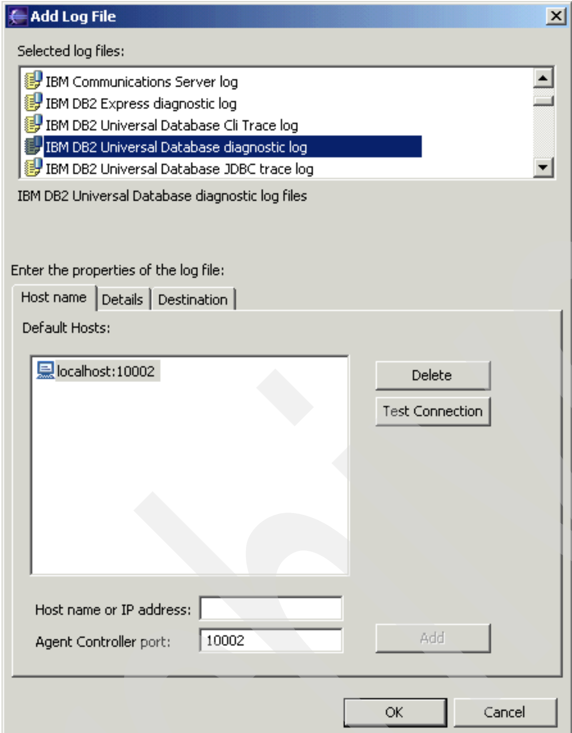


Figure 5-2 Add Log File

6. Now in the Details tab in the properties of the log file, choose the log file name that need to view, as shown in Figure 5-3.

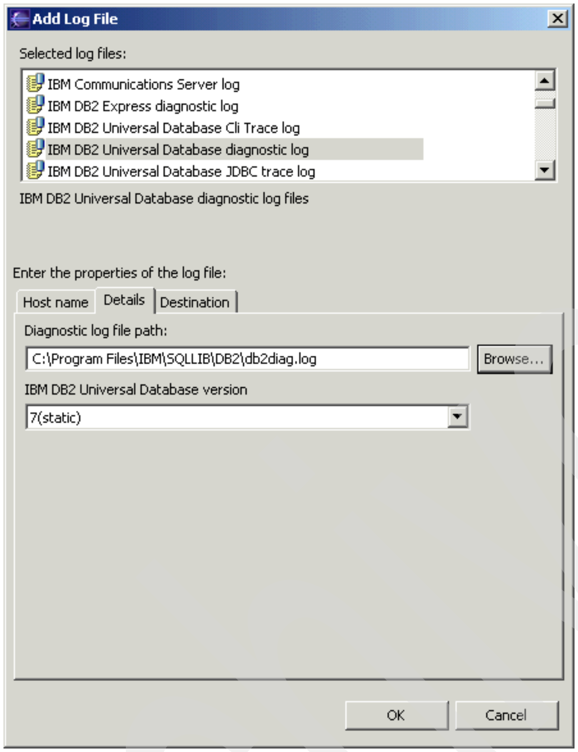


Figure 5-3 Select the file to view

7. From the Destination tab in the properties of the log file, enter the destination project in the **Project** field.
8. Click **OK** and then **Finish** to add the log file to import the file. The imported file will open in the Log View, as shown in Figure 5-4 on page 142.

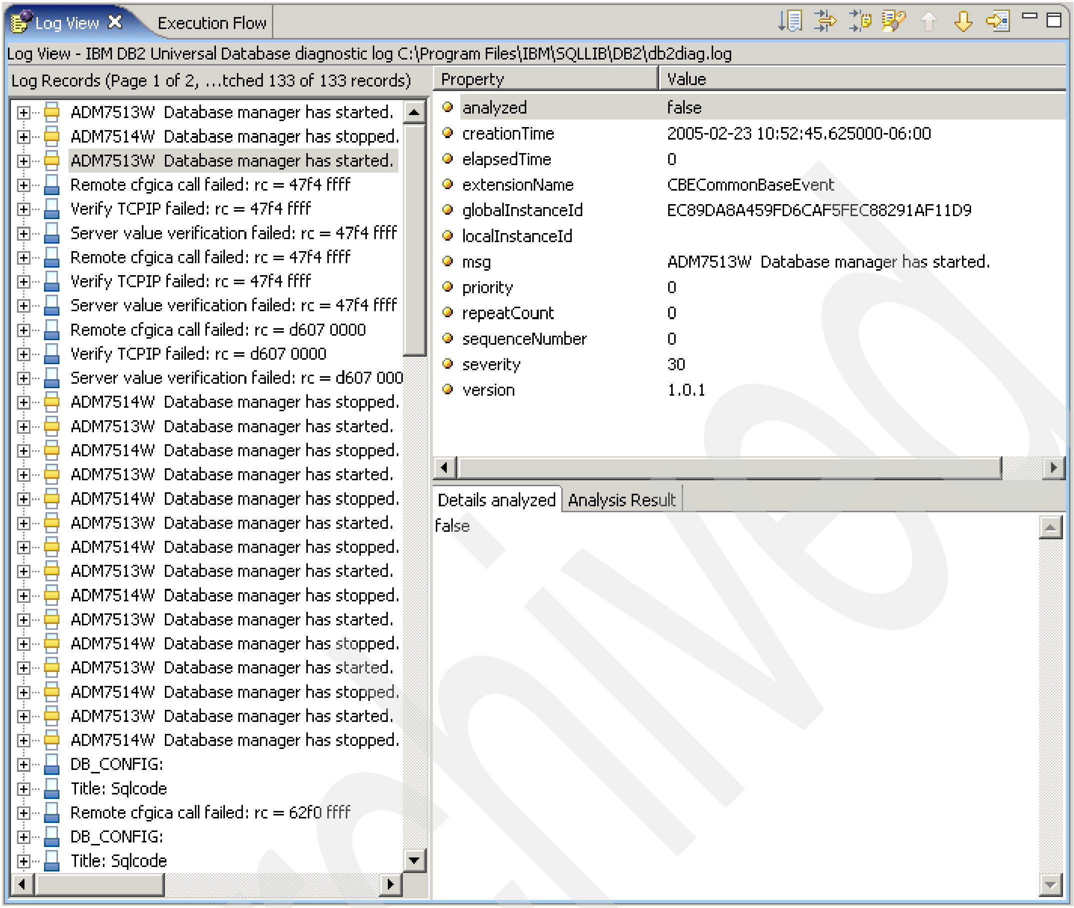


Figure 5-4 Log View for the imported log file

There are four panes in the Log View, as we can see in the Figure 5-4. The following descriptions show the information that these panes contain:

Log Records	This lists the parsed events from the imported log file. There are various identifiers to indicate the severity of each events in the Log Records pane.
Property and Value	This shows the properties and their values. For more information about the properties, refer to Chapter 3, “Common Base Events” on page 35.
Details analyzed	The Details pane shows further details about the property selected in the Property window.

Analysis Result — If the log record has been analyzed, this shows the solution that was identified. For more information about analyzing the log files, refer to 5.1.2, “Analyzing the log files using the symptom database” on page 143.

5.1.2 Analyzing the log files using the symptom database

The analysis of the log files is performed by comparing the events against a symptom database that loaded in memory. The solution reported in the Analysis Result pane of the Log View suggests the action that required for the reported problems. To analyze the log events, we create the symptom database, add symptoms to this database, and finally analyze the logs.

Creating a symptom database

A symptom database is an XML file with possible symptoms, solutions, and directives. A symptom database contains the following elements:

Symptoms — The symptoms are possible problems that an application could encounter.

Solutions — Each symptom is associated with one or more reasons for the symptom.

Directives — The directives are the instructions to solve the problem.

To create a symptom database, perform the following steps. Here, we are creating a symptom database called *SampleSymptomdb*.

1. Start Eclipse.
2. Select **File** → **New** → **Other**. Expand **Profiling and Logging**, select **Symptom Database**, and click **Next**. Figure 5-5 on page 144 shows the Select wizard.

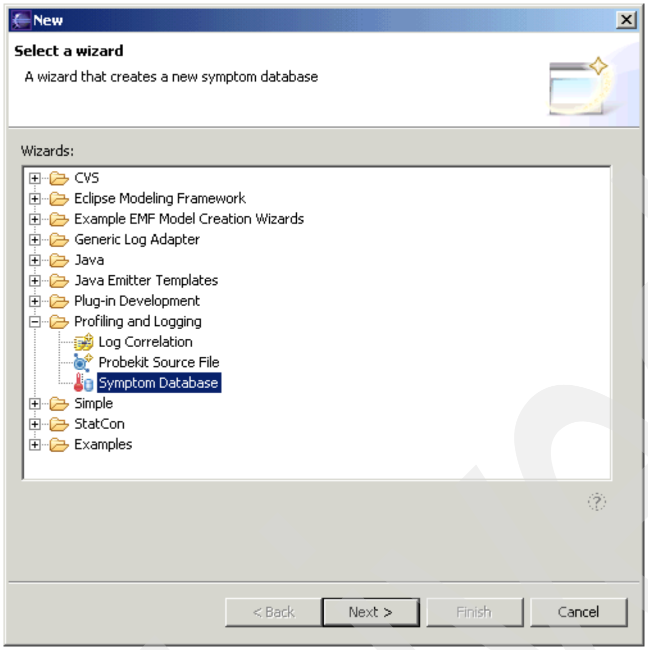


Figure 5-5 Select wizard for the creation of Symptom Database

3. Next, we specify the Location and Name of the symptom database. The symptom database in our case study is `SampleSymptomdb`, and the Location is `\LTA_project`, as shown in Figure 5-6 on page 145. The location can be set by clicking **Browse** to browse for your location.

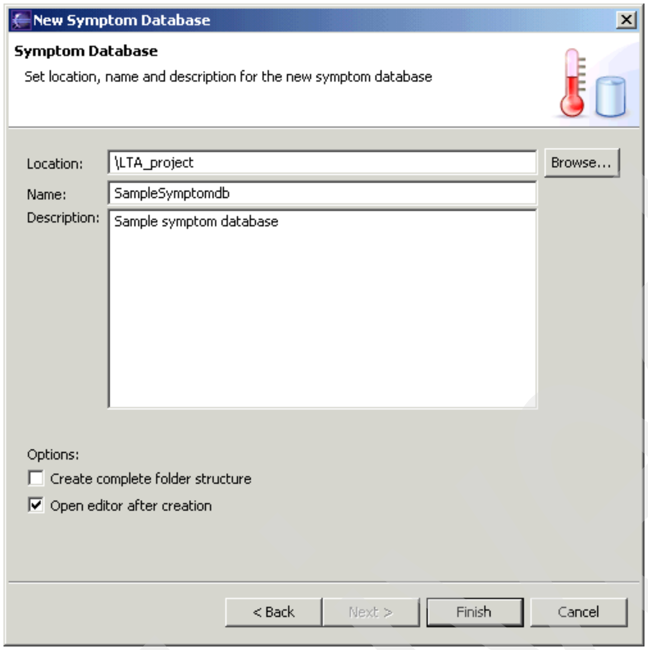


Figure 5-6 Set the Location and Name of the symptom database

4. Click **Finish** after entering the location and name of the symptom database. SampleSymptomdb will open in the symptom database editor. We edit the system database on the Details tab of the symptom database editor, as shown in Figure 5-7.

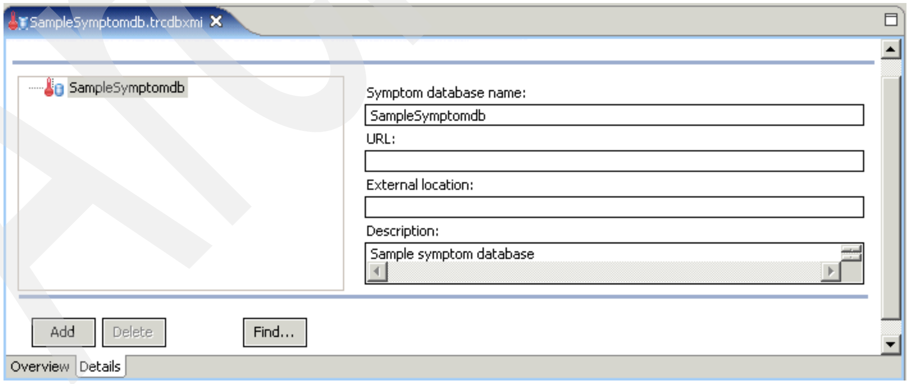


Figure 5-7 SampleSymptomdb opened in the symptom database editor

Now, we have created the symptom database named SampleSymptomdb. The symptoms and solutions that we create in this example are for the following IBM DB2® log files that are available in the db2diag.log file:

- ADM7514W Database manager has stopped.
- ADM6017E Tablespace "SYSCATSPACE (ID "0") is full.

The following steps describe how to edit the SampleSymptomdb database to add the symptoms, solutions, and directives:

1. Select **SampleSymptomdb** in the symptom database editor.
2. Click **Add** to add a symptom. In the Symptom name field on right pane enter `ADM7514W : Database warning`.
3. In the Match patterns field, enter `ADM7514W`, as shown in the Figure 5-8.

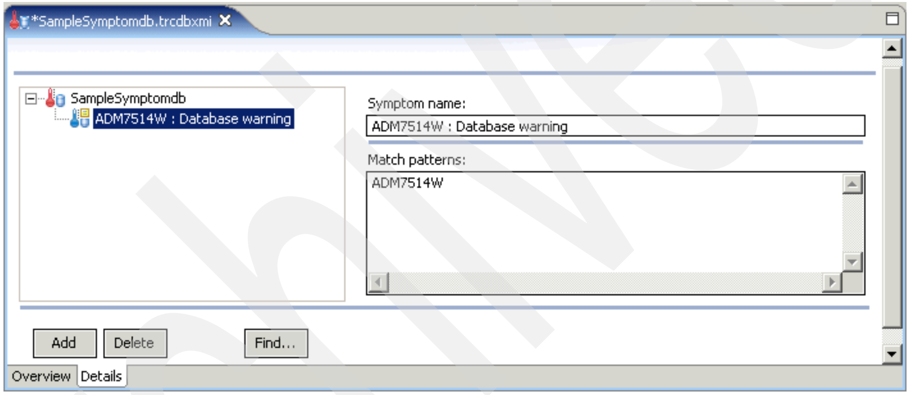


Figure 5-8 Add a symptom to SampleSymptomdb

4. To add a solution to this symptom, click **ADM7514W : Database warning**, and then click **Add**.
5. Enter `Database Manager has stopped` in the Description field on right pane, as shown in Figure 5-9 on page 147. Now, save the file by clicking **File** → **Save**.

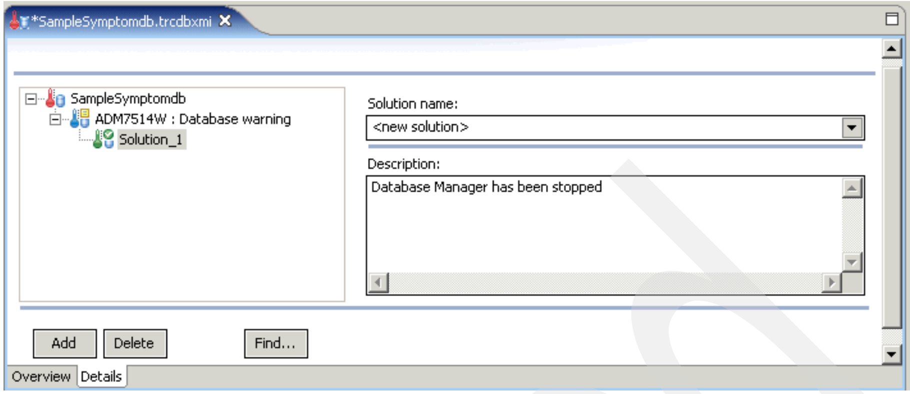


Figure 5-9 Add a solution for the symptom

6. Next, we add a directive to the solution. A directive is a remedy for the symptom. There can be multiple directives for a symptom that could contribute to the ultimate solution. To add directive to the solution, select the solution **Database Manager has been stopped**, and then click **Add**.
7. In the Description field on right pane, enter `Restart the database`, and then save the file by clicking **File** → **Save**. Now the symptom database looks as shown in Figure 5-10. Within the workbench, the database exists in an XML format.

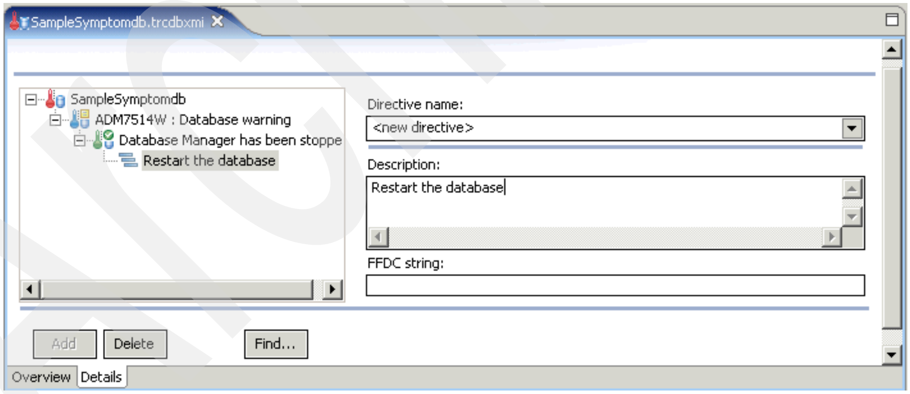


Figure 5-10 The SampleSymptomdb with symptom, solution, and directive

We need to repeat these steps to add all the other symptoms and solutions.

The symptom database can be exported to an XML formatted file. The following steps describe how to export the database to an XML formatted file:

1. Expand the **Symptom Databases** in the Navigator view. Right-click the newly created **SampleSymptomdb.trcdbxmi** and click **Export**. The Export wizard opens, as shown in Figure 5-11.

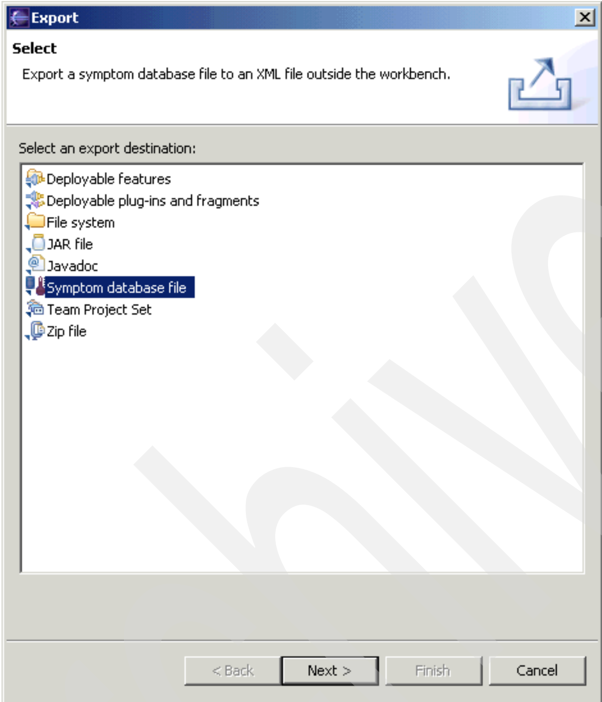


Figure 5-11 Export wizard to export the symptom database

2. Select **Symptom database file**, as shown in Figure 5-11, and click **Next**.
3. Select the export destination by clicking **Browse**, and then click **Finish**, as shown in Figure 5-12 on page 149.

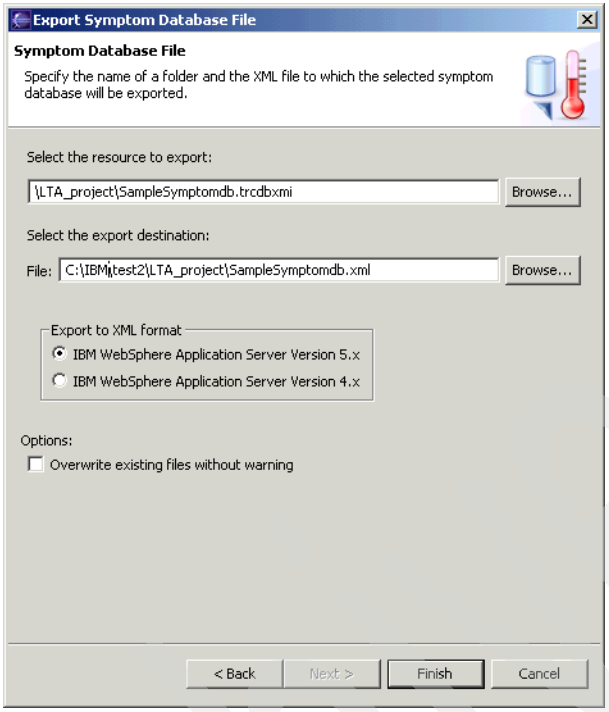


Figure 5-12 Create SampleSymptomdb.xml by exporting the symptom database

The listing in Example 5-1 shows the SampleSymptomdb.xml that we created.

Example 5-1 Listing of SampleSymptomdb.xml file

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE symptomDatabase [

<!ELEMENT symptomDatabase (runtime+)>
<!ATTLIST symptomDatabase>

<!ELEMENT runtime (symptom+ , solution+ , directive*)>
<!ATTLIST runtime
    id ID #REQUIRED
    name CDATA #IMPLIED
    symptomUrl CDATA #IMPLIED
    localExternalFileLocation CDATA #IMPLIED>

<!ELEMENT symptom (matchPattern+)>
<!ATTLIST symptom
    id ID #REQUIRED
    description CDATA #IMPLIED
```

```
    solutions IDREFS #REQUIRED>

<!ELEMENT matchPattern EMPTY>
<!ATTLIST matchPattern
    name CDATA #IMPLIED
    value CDATA #REQUIRED>

<!ELEMENT solution EMPTY>
<!ATTLIST solution
    id ID #REQUIRED
    description CDATA #IMPLIED
    directives IDREFS #IMPLIED>

<!ELEMENT directive EMPTY>
<!ATTLIST directive
    id ID #REQUIRED
    description CDATA #IMPLIED
    directiveString CDATA #REQUIRED>
]>

<!--  Created on Thursday, March 24, 2005 at 11:58:34:0656 AM CST -->

<symptomDatabase>
    <runtime id="Runtime_0" name="SampleSymptomdb" symptomUrl=""
localExternalFileLocation="">

        <!-- Symptoms: -->

        <symptom id="Symptom_0" description="ADM7514W : Database Warning"
solutions="Solution_0">
            <matchPattern name="null" value="ADM7514W"/>
        </symptom>
        <symptom id="Symptom_1" description="ADM0500E : Severe authority issue"
solutions="Solution_1">
            <matchPattern name="null" value="ADM0500E"/>
        </symptom>
        <symptom id="Symptom_2" description="ADM0501C : DB2 Instance has
terminated abnormally" solutions="Solution_2">
            <matchPattern name="null" value="ADM0501C"/>
        </symptom>
        <symptom id="Symptom_3" description="ADM0502C : DB2 Instance
terminated" solutions="Solution_3">
            <matchPattern name="null" value="ADM0502C"/>
        </symptom>

        <!-- Solutions: -->

        <solution id="Solution_0" description="Database Manager has stopped"
directives="Directive_0"/>
```

```
        <solution id="Solution_1" description="No proper access rights to the
user account" directives="Directive_1"/>
        <solution id="Solution_2" description="A stack overflow exception has
occured" directives="Directive_2"/>
        <solution id="Solution_3" description="DB2 instance has terminated
abnormally" directives="Directive_3"/>

        <!-- Directives: -->

        <directive id="Directive_0" description="Restart the database"
directiveString=""/>
        <directive id="Directive_1" description="Ensure the user account has
the correct access rights" directiveString=""/>
       <directive id="Directive_2" description="Increase the default sack size
for db2syscs.exe using the db2hdr utility" directiveString=""/>
        <directive id="Directive_3" description="Increase the AGENT_STACK_SZ
DBM configuration parameter" directiveString=""/>
    </runtime>
</symptomDatabase>
```

Analyzing the log files

Analyzing log records means comparing log records using specified symptom databases that are loaded in memory. The solution is reported in the Analysis Result pane.

The following steps describe how to analyze a particular log record displayed in the Log View:

1. Select the log record that needs to be analyzed in the Log View. In our example, it is **ADM7514W Database manager has stopped**. Right-click it, and then select **Analyze** → **Default Log Analyzer**, as shown in Figure 5-13 on page 152.

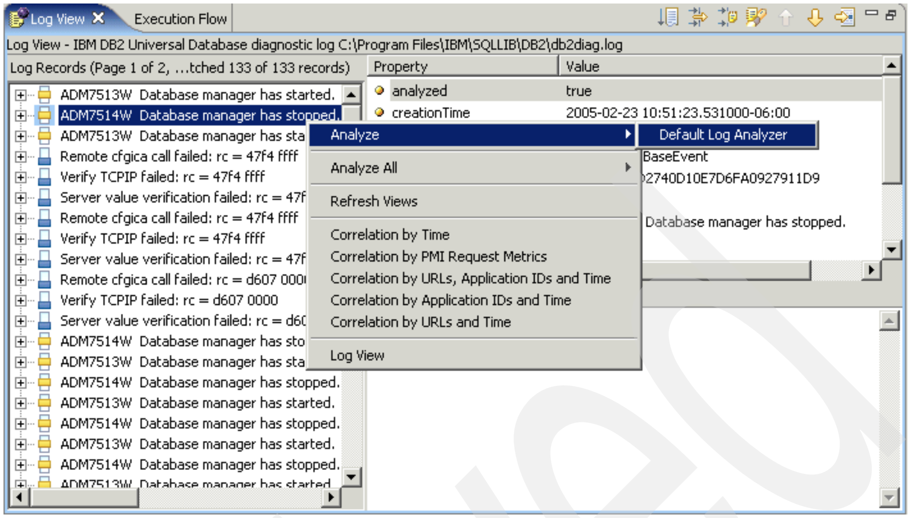


Figure 5-13 Analyze a log record

2. If a match is found in the symptom database, the record will be highlighted in blue. Click the **Analysis Result** tab to view the solution information, as shown in Figure 5-14.

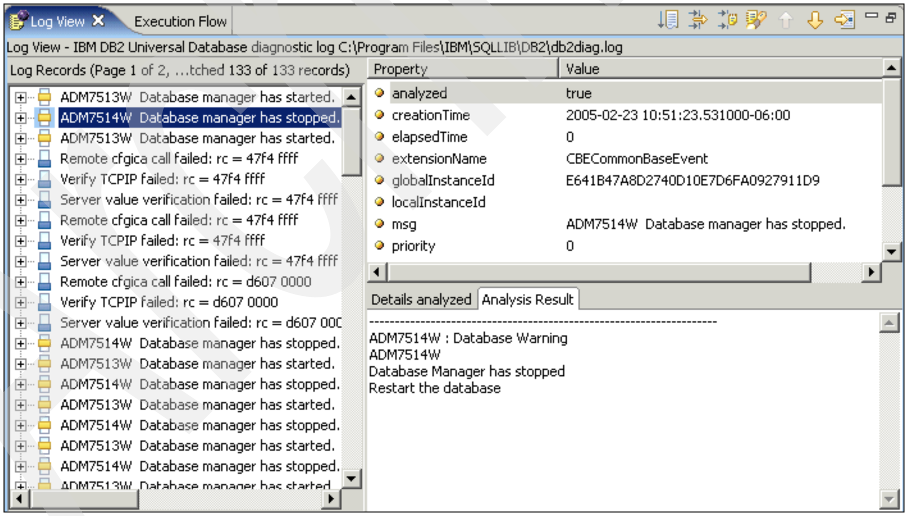


Figure 5-14 The Analysis Result view

To analyze all the log records, right-click any record in the Log View, and then select **Analyze All** → **Default Log Analyzer**.

5.1.3 Importing a symptom database

The symptom database can be imported from a remote host or from a local host. In this scenario, we import the previously created SampleSymptomdb.xml file into the workspace as symptomdb.trcdbxmi. To import a symptom database, perform the following steps:

1. Start Eclipse.
2. Select **File** → **Import**, click **Symptom Database File** in the Import wizard, and click **Next**, as shown in Figure 5-15.

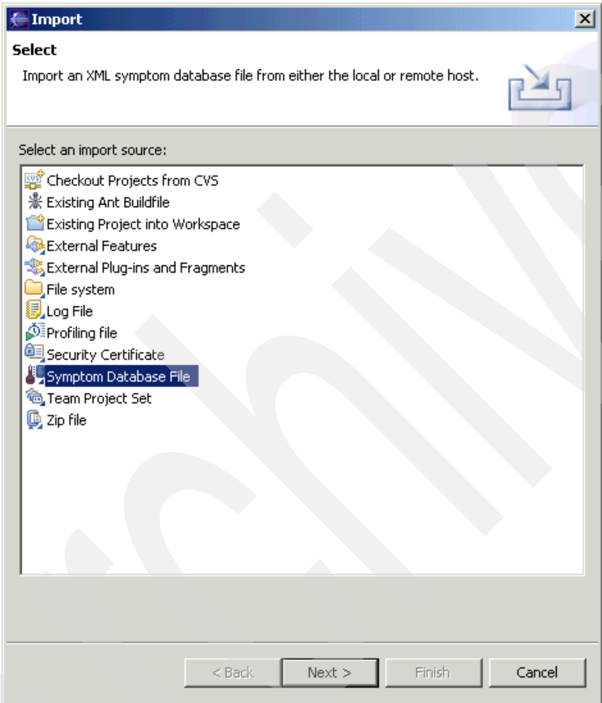


Figure 5-15 Import a symptom database

3. Select the location of the symptom database. In our example, the symptom database is located on the local machine. For this reason, we select **Local host** and select the symptom database XML file that needs to be imported. Now, specify the name and location of the imported resource. This is shown in Figure 5-16 on page 154.

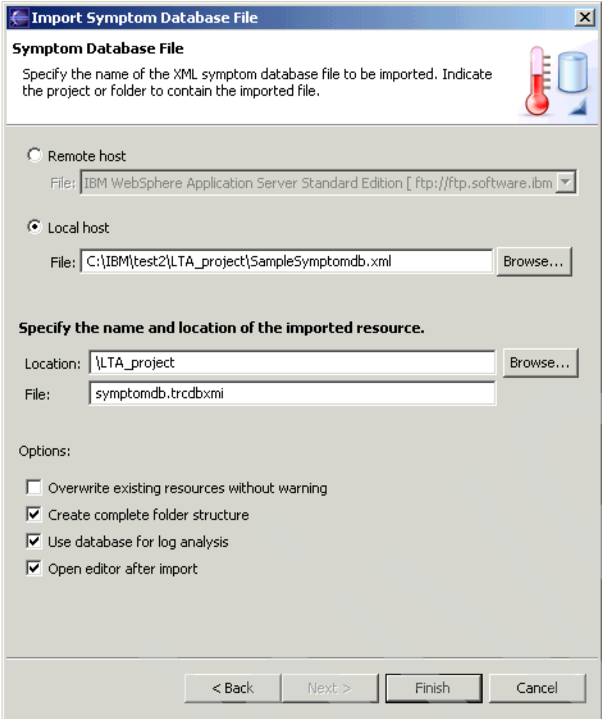


Figure 5-16 Set the details to import the symptom database

4. Click **Finish**, and database will be imported.

5.1.4 Working with multiple symptom databases

If multiple symptom databases are in use, we can specify against which symptom database the log records need to be analyzed.

To do this, click **Windows** → **Preferences**, expand the **Profiling and Logging** and **Log View**, and then click **Symptom Database**. This will list all the symptom databases that are in use. Select the symptom database that needs to be used to analyze the log. In this example, it is **/LTA_project/symptomdb.trcdbxmi**, as shown in Figure 5-17 on page 155.

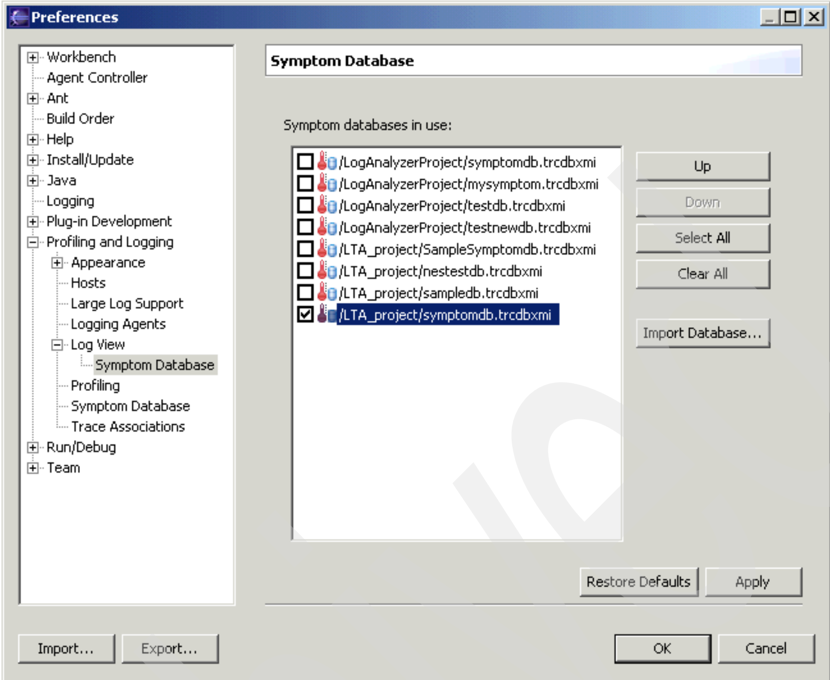


Figure 5-17 Select the symptom database that needs to be used

5.1.5 Correlating the log files

When using multiple applications that interact with each other, we can correlate their log files to be able to analyze the events. Correlation in the Log and Trace Analyzer means finding the relation between the distributed log records and learning the influence of one log record on another. The log records can be from the same log file or from different log files; this relation between the log records can be based on the different properties or a combination of the properties of the Common Base Event.

Consider an application that connects to a database. If an error occurs in the database causing the application to disconnect, it is difficult to track down the problem only by looking into the application error logs. The error logs of the application and the database need to correlate to pinpoint the problem. The Log and Trace Analyzer correlation helps here to correlate both the log files. The correlation of the log files is performed on the log attributes, such as time stamp and severity.

The following steps describe how to correlate two log files. In this example, a small snippet of the application log that shows an insertion failure in a DB2 database is used for the correlation. The DB2 database was down during this time. The correlation was done based on severity. The severity correlation is a custom correlation presented in 5.4, “Creating a custom correlator for the Log and Trace Analyzer” on page 179. Refer to that section for more details about how to create a custom correlator. Perform the following steps to correlate log files:

1. Start Eclipse and import the log files that need to be correlated. Refer to 5.1.1, “Viewing the log files” on page 138 for information about importing the log files. In this example, we import the application log and the db2diag.log files.
2. In the Log Navigator, select the files that need to be correlated, and then right-click and select **New** → **Log Correlation**. The wizard, as shown in Figure 5-44 on page 190, opens and shows the selected log records in the list on right side. Enter a Name for the correlation and click **Next**.

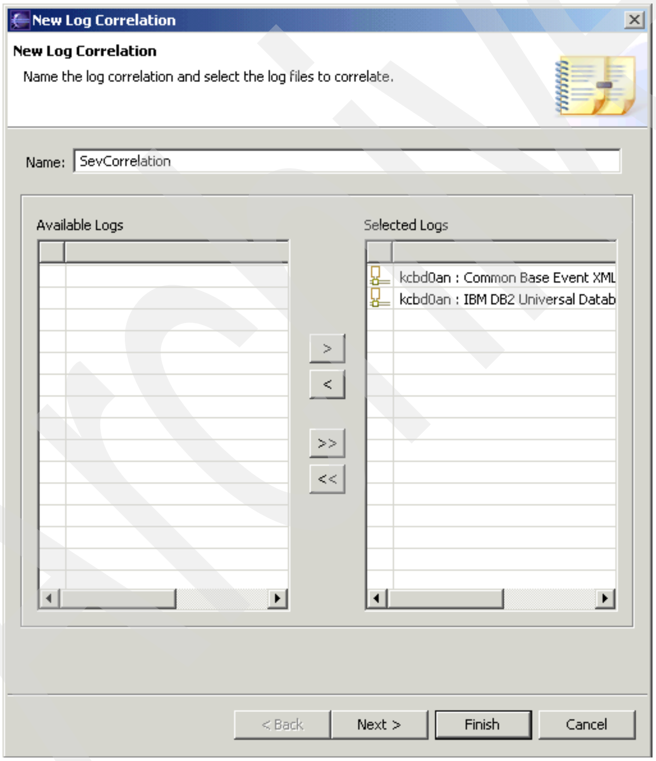


Figure 5-18 Create New Log Correlation

3. Select **Severity correlation** from the list of Correlation schemas, and click **Finish**.

4. In Log Navigator, right-click **SevCorrelation** and select **Open With → Log interactions**. A Unified Modeling Language (UML) diagram similar to the one shown in Figure 5-19 will be displayed on right pane.

   This UML diagram represents the events in the log files and their correlation with the other events in the same log file and across the logs.

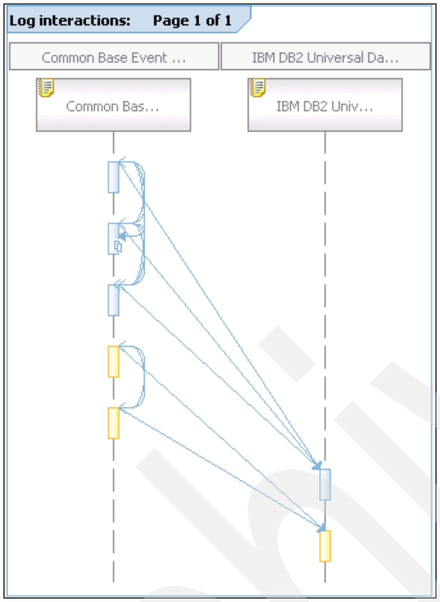
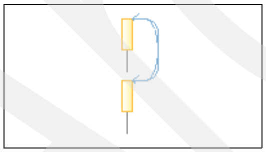
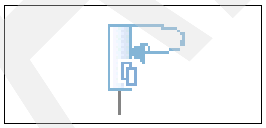


Figure 5-19 UML sequence diagram

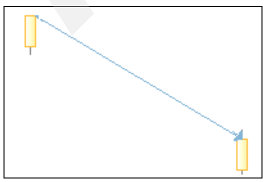
The following descriptions show the representation of the UML sequence diagram shown in Figure 5-19:



Correlation between the events in the same log file



Correlation in the same node



Correlation between the events in two different log files

5.2 Using Agent Controller for monitoring remote logs

The Agent Controller provides the convenience of analyzing a log file from a remote machine without having to transfer a copy of the file to a local machine. By installing the Agent Controller on a remote machine that runs an application, users can use the Log and Trace Analyzer (LTA) running on another machine (monitoring system) to import and analyze a log file that is continuously updated by the application on the remote machine. The Remote Agent Controller (RAC) is available with the Agent Controller package in the IBM Autonomic Computing Toolkit. The RAC is used for processing remote log files using the Log and Trace Analyzer.

In the example shown in this section, by using the Log and Trace Analyzer, we monitor the Common Base Events created by the Generic Log Adapter on a remote machine. The remote machine is installed with the Agent Controller and the Generic Log Adapter. The illustration in Figure 5-20 shows the RAC with remote log monitoring using the Log and Trace Analyzer.

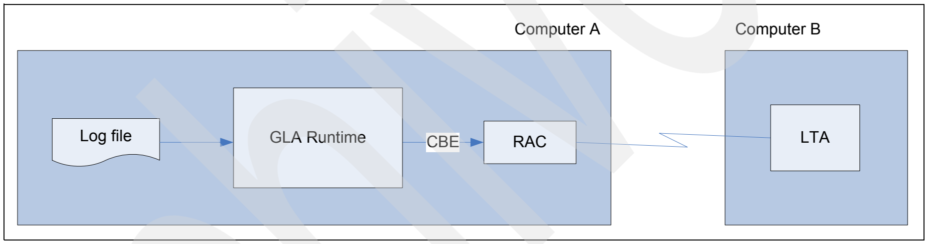


Figure 5-20 Illustration of RAC with Generic Log Adapter and Log and Trace Analyzer

Here, the Generic Log Adapter is configured to write the Common Base Events to the logging agent using the LoggingAgentOutputter outputter type with executable class org.eclipse.hyades.logging.adapter.outputters.CBELogOutputter.

For the example presented in Figure 5-20, we need to configure the following items:

- On computer A:
  - Configure the Agent Controller and start the service.
  - Configure the Generic Log Adapter to write to the logging agent.
- On computer B:
  - Configure the Log and Trace Analyzer to monitor the logging agent.

The following sections describe the step-by-step methods for configuring the Agent Controller, configuring the Generic Log Adapter to use the logging agent, and configuring the Log and Trace Analyzer to manually attach to and start monitoring the agent.

Configuring the Agent Controller

To configure the Remote Agent Controller (RAC), the serviceconfig.xml file needs to be edited. This file is loaded by the Agent Controller when it is started.

The serviceconfig.xml file can be found in the <RAC_install>\config directory, where <RAC_install> is the remote agent installation directory.

Add the following lines in serviceconfig.xml file to create an agent with the name, for example, DB2LogAgent. In our example, DB2LogAgent will be used by the Generic Log Adapter outputter to write the formatted Common Base Event messages.

```
<Agent configuration="default" name="DB2LogAgent" type="Logging">
</Agent>
```

Example 5-2 shows the complete serviceconfig.xml file after modification.

Example 5-2 Modified serviceconfig.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<AgentControllerConfiguration activeConfiguration="default" filePort="10005"
jvm="C:\Program Files\IBM\Java142\jre\bin\classic\jvm.dll" loggingDetail="LOW"
loggingLevel="INFORMATION" port="10002" processPolling="true"
securedPort="10003" version="3.0.1.1">
   <AgentControllerEnvironment configuration="default">
      <Variable name="JAVA_PATH" position="replace" value="C:\Program
Files\IBM\Java142\bin\java.exe"/>
      <Variable name="RASERVER_HOME" position="replace" value="C:\Program
Files\IBM\AgentController"/>
      <Variable name="PATH" position="prepend" value="%RASERVER_HOME%\bin"/>
      <Variable name="SYS_TEMP_DIR" position="replace"
value="C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp"/>
      <Variable name="LOCAL_AGENT_TEMP_DIR" position="replace"
value="%SYS_TEMP_DIR%"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\commons-logging.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hcframe.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hexcore.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hexl.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hexr.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hl14.jar"/>
```

```
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlcommons.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlcore.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlevents.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlcbe101.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\ecore.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\common.jar"/>
   </AgentControllerEnvironment>
   <Application configuration="default" executable="java.exe"
location="%SYS_TEMP_DIR%" path="%JAVA_PATH%"/>
   <Hosts configuration="default">
      <Allow host="ALL"/>
   </Hosts>
   <Agent configuration="default" name="DB2LogAgent" type="Logging">
     </Agent>
   <Plugin path="%RASERVER_HOME%\plugins"/>
</AgentControllerConfiguration>
```

Restart the Agent Controller service to use the modified configuration file. In Windows, the service name is *IBM Rational® Agent Controller*.

After it is restarted, the RAC configuration is ready, and the next step is to configure the Generic Log Adapter to use the agent named DB2LogAgent to write the formatted CBEs.

Configuring the Generic Log Adapter

The following steps describe how to configure the Generic Log adapter to use the DB2LogAgent that we defined in the previous section. The Generic Log Adapter (GLA) here reads the db2diag.log file of DB2 and writes to the RAC. A GLA adapter named *db2parser* is created in this example for this purpose.

The following configuration is required for the sensor and outputter for the GLA. Refer Chapter 4, “Generic Log Adapter” on page 77, for more information about the Generic Log Adapter and creating the adapter configuration file.

Sensor configuration

Configure the sensor to read the db2diag.log and sensor properties directory and fileName as follows:

directory	<DB2_INSTALL>\SQLLIB\DB2\db2diag.log
fileName	db2diag.log

Outputter configuration

Configure the context component to use the CBELogOutputter, as shown in Figure 5-21.

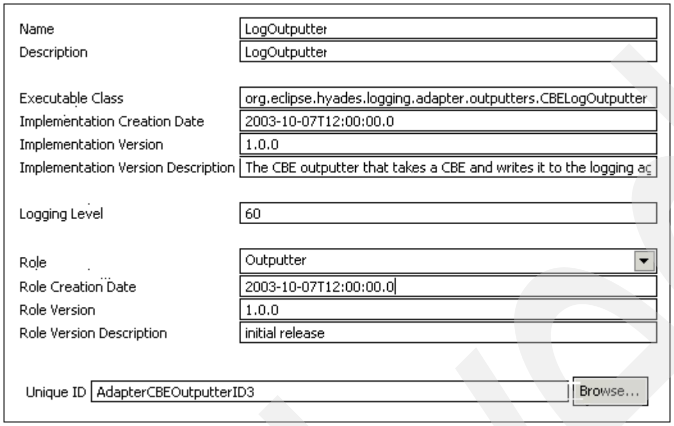


Figure 5-21 Configuring CBELogOutputter in the Generic Log Adapter

Perform the following steps to configure the outputter in the context instance:

1. Set the Outputter Type to **LoggingAgentOutputter**.
2. Add a property to the outputter by right-clicking **Outputter** and selecting **Add** → **Property**. Use the following values to configure it:

   Property Name `agentName`

   Property Value `DB2LogAgent`

3. Add another property to the outputter using the previous steps and configure the attributes as follows:

   Property Name `waitUntilLoggingTime`

   Property Value `30000`

   > **Note:** The waitUntilLoggingTime value specifies the time in milliseconds to wait for the user to attach and start monitoring the agent before it starts writing to the agent. If this property is not specified, the outputter will not wait for the agent to be monitored before writing to it.

Now, we are ready with the Generic Log Adapter configuration to use the logging agent.

The final step is to configure the Log and Trace Analyzer to monitor the configured remote agent.

Configuring the Log and Trace Analyzer

Following steps describe how to attach to and monitor the remote agent:

1. Start Eclipse.
2. Select **Window** → **Preferences** and click **Profiling and Logging**. On the right pane, select **Enable logging**.
3. Select **Window** → **Open Perspective** → **Other**. A wizard, as shown in Figure 5-22, opens.

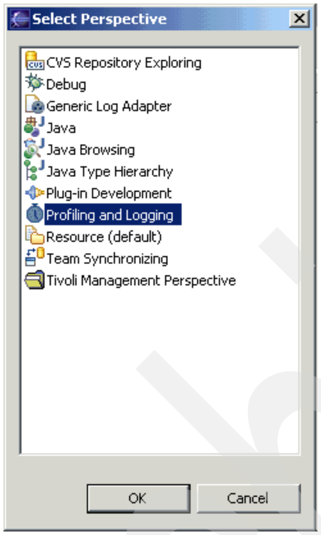


Figure 5-22 Select the Profiling and Logging perspective

4. Select the **Profiling and Logging** perspective, as shown in Figure 5-22, and click **OK**.

5. Select **Run** → **Profile** and click **Attach - Java Process**. The wizard shown in Figure 5-23 opens, where we can create a new profile and attach the DB2LogAgent.

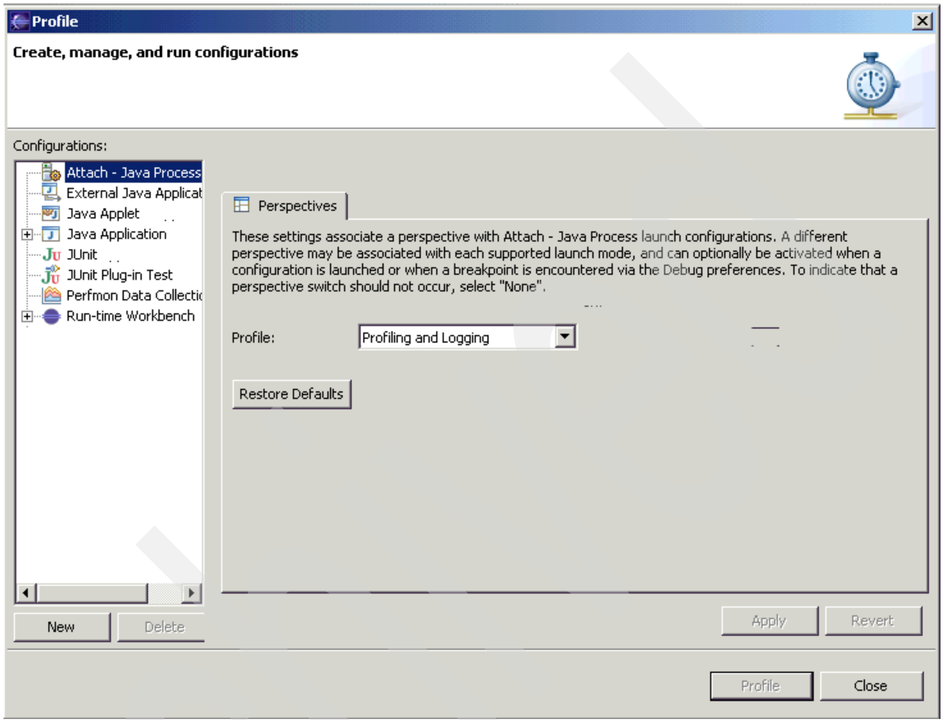


Figure 5-23 Wizard for creating a new profile

6. Right-click **Attach - Java Process** and select **New** to attach a new Java process and name it, for example, `DB2LogProfile`. This is shown in Figure 5-24.

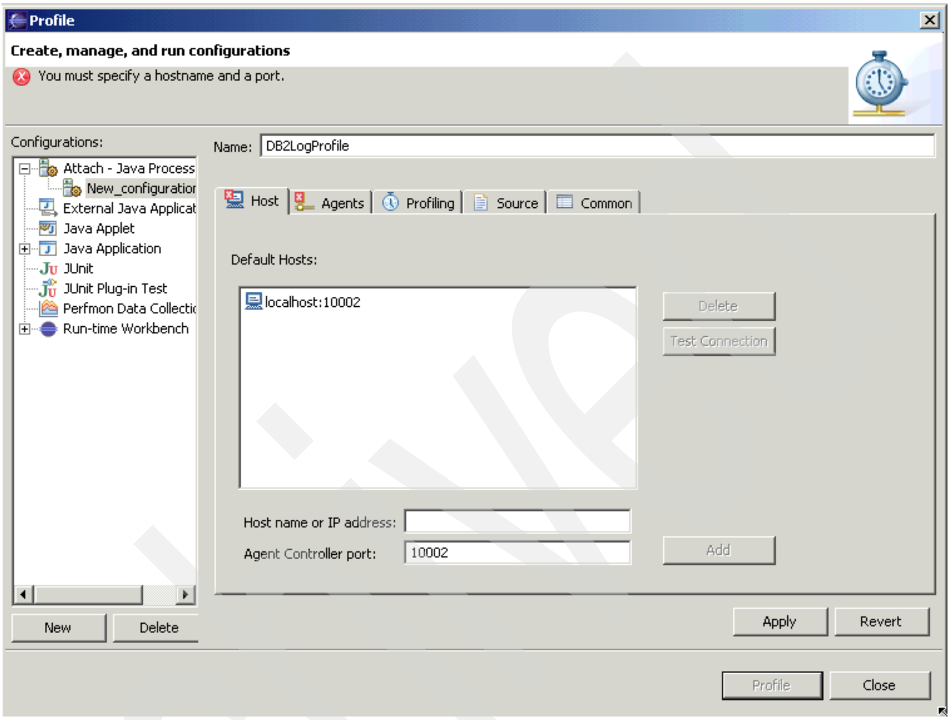


Figure 5-24 Create a new profile

7. Now, we need to add the remote host name where the Agent Controller logs the Common Base Events. In the Host tab, enter the Host name or IP address and the Agent Controller port on the remote system and click **Add**.

   The default port that the Agent Controller uses is 10002. We use the default port in this example. Figure 5-25 on page 165 shows the configuration of the host.

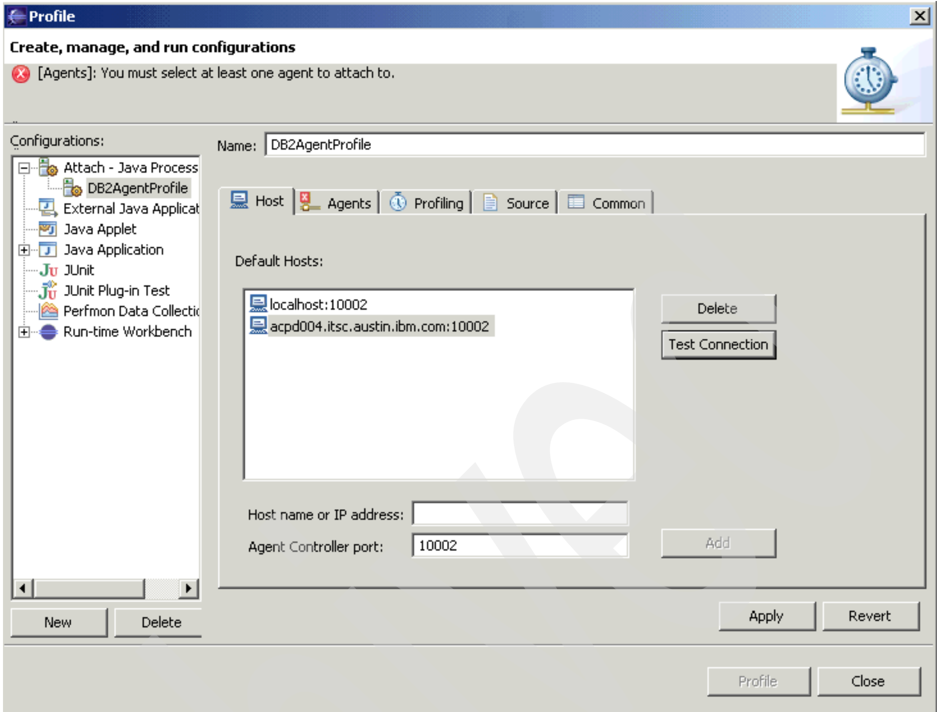


Figure 5-25 Configure the remote host in profile

8. Click **Test Connection** to test the connection to the remote system. If the connection succeeds, click **Apply**.

The Profiling Tool of the Log and Trace Analyzer provides the ability to attach to a running application. Therefore, to attach the agent, we need to launch the application that writes to the logging agent. For this example, we already created a Generic Log Adapter that writes to the logging agent. In our example, the name of the adapter that we created, which converts the db2diag.log into Common Base Events, is db2parser.adapter. Now, run this adapter by using the following command:

```
gla db2parser.adaper
```

The DB2LogAgent can now be attached to the profile for monitoring. The following steps describe how to attach the DB2LogAgent and monitor the remote log:

1. Click **Run** → **Profile**, expand **Attach - Java Process**, and click **DB2LogProfile**.

2. Go to the Agents tab. The agents that are currently active are listed, as shown in Figure 5-26.

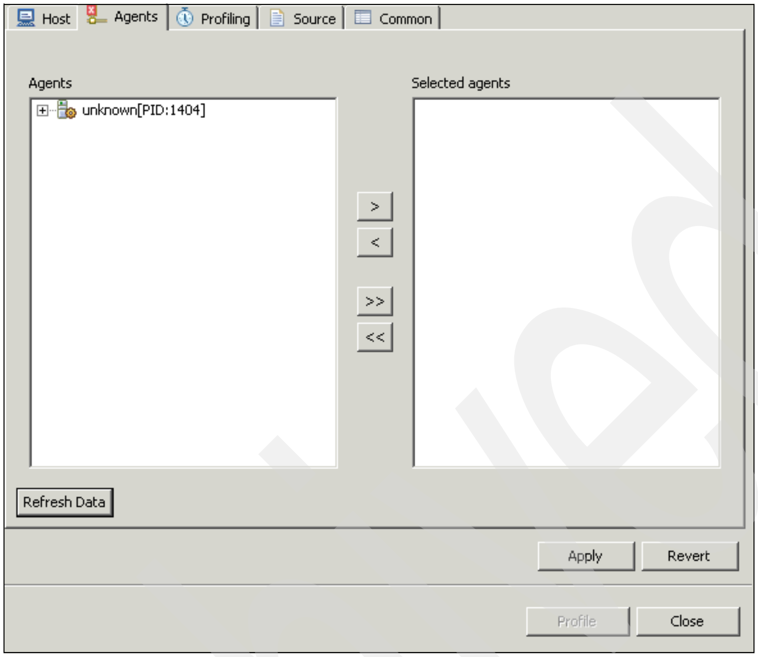


Figure 5-26 List of running agents

3. Select **DB2LogAgent** and click **>** to select the agent to monitor.
4. Click **Apply**, and then click **Profile**. The attached log agent is displayed in the Profiling Monitor, as shown in Figure 5-27 on page 167.

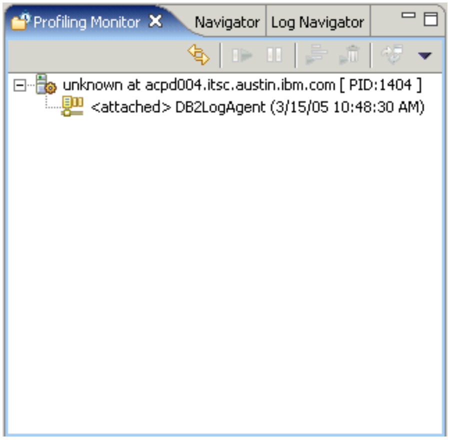


Figure 5-27 Display of DB2LogAgent in Profiling Monitor

5. Right-click the attached **DB2LogAgent** in the Profiling Monitor and select **Start Monitoring**, as shown in Figure 5-28.

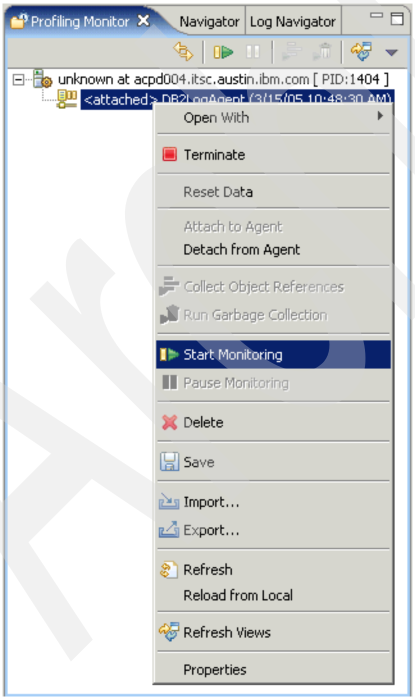


Figure 5-28 Start Monitoring DB2LogAgent

6. Again, right-click the attached **DB2LogAgent** and select **Open With** → **Log View** to display the Common Base Events. The events can be viewed in the Log View, as shown in Figure 5-29.

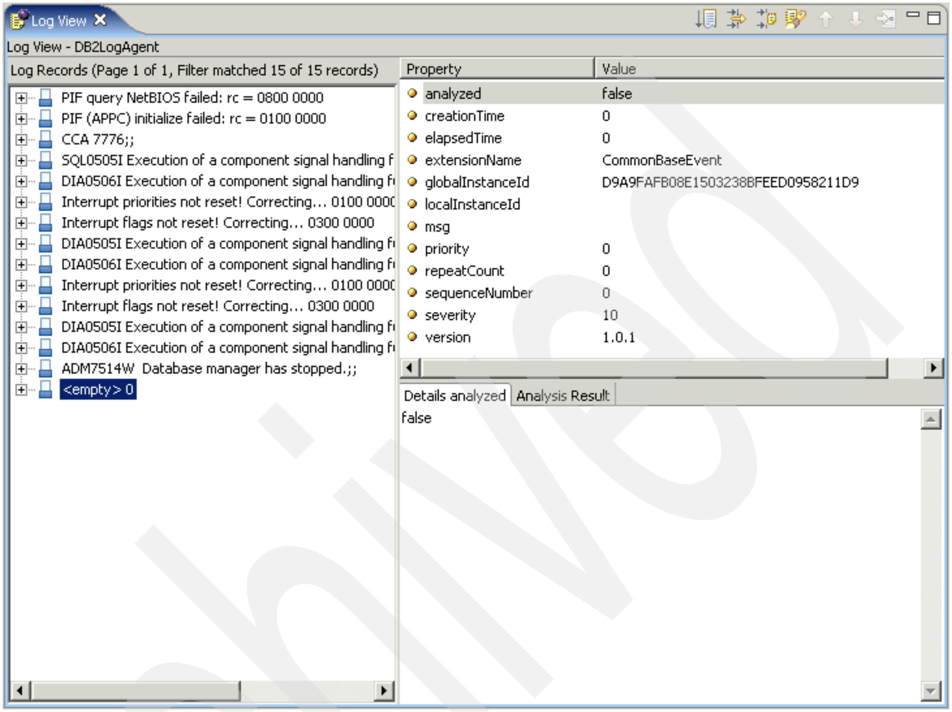


Figure 5-29 The events listed in Log View using the Profile Monitor

5.3 Creating a custom log parser for the Log and Trace Analyzer

Log parsers can be created and made available with the Import wizard of the Log and Trace Analyzer (LTA) to parse the log files created by an application. Custom log parsers are defined to LTA using plug-ins. In the plug-in definition, a Generic Log Adapter (GLA) configuration file must be specified to perform the parsing tasks.

In this example, we create a log parser named ITSOParser to parse a log file called sample.log. This log file is created by a sample application named SampleApplication. The SampleApplication used in this chapter is the same as the one used in Chapter 4, “Generic Log Adapter” on page 77. The ITSOParser also displays the sample.log file content in the Log View of the Log and Trace Analyzer. Figure 5-30 shows a high-level view of what we discuss in this section.

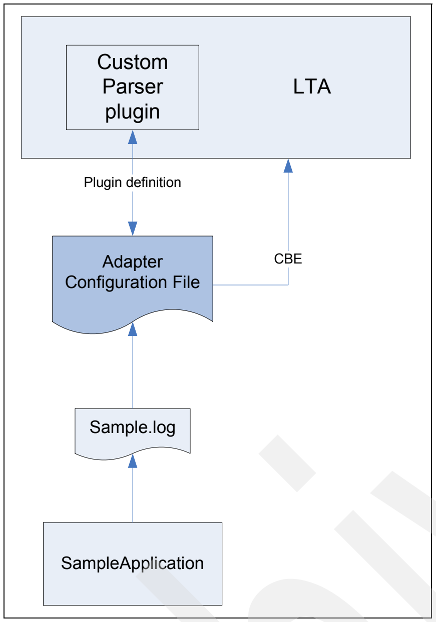


Figure 5-30 Custom parser example scenario

The simpleGLA.adapter, also created in Chapter 4, "Generic Log Adapter" on page 77, is also used here with a modified outputter. The outputter will be configured to send the formatted Common Base Events (CBEs) to a logging agent.

To create the ITSOParser, we perform the following steps:

1. Create the log parser plug-in, named ITSOParser in our example.
2. Create the adapter configuration file for parsing the sample.log file.
3. Test the log parser.
4. Deploy the custom log parser.

5.3.1 Creating the log parser plug-in

The following steps describe the definition of a plug-in to the Log and Trace Analyzer tool that uses a GLA configuration file for parsing a log file:

1. Start Eclipse.

2. Select **File** → **New** → **Other**, expand **Plug-in Development**, and select **Plug-in Project**. Click **Next**.
3. Enter `ITSOParser` for the Project Name, and then click **Next**.
4. Clear the **Generate the Java class that controls the plug-in's life cycle** option, as shown in Figure 5-31. Click **Finish**. The ITSOParser opens in the Plug-in Development perspective.

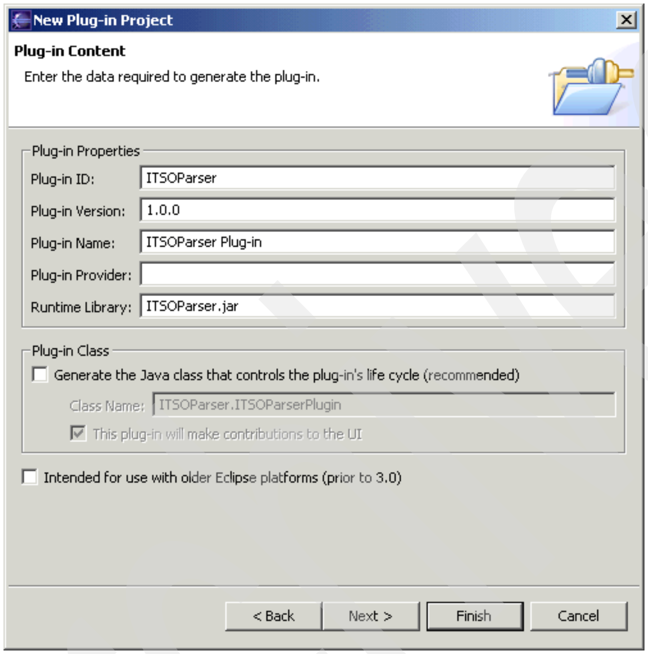


Figure 5-31 Create the ITSOParser Plug-in Project

5. Click the **Dependencies** tab, and then click **Add**. The following plug-ins need to be added:
   – org.eclipse.core.boot
   – org.eclipse.core.resources
   – org.eclipse.core.runtime
   – org.eclipse.hyades.logging.adapter.config
   – org.eclipse.hyades.logging.parsers
   – org.eclipse.core.runtime.compatibility
   – org.eclipse.hyades.logging.adapter
   – org.eclipse.ui
6. Click the **plugin.xml** tab and add an extension, as shown in bold in Example 5-3 on page 171.

Example 5-3 The plugin.xml file for the custom parser ITSOParser

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="ITSOParser"
   name="ITSOParser Plug-in"
   version="1.0.0"
   provider-name="">

   <runtime>
      <library name="ITSOParser.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.core.runtime.compatibility"/>
      <import plugin="org.eclipse.hyades.logging.adapter"/>
      <import plugin="org.eclipse.hyades.logging.adapter.config"/>
      <import plugin="org.eclipse.hyades.logging.parsers"/>
      <import plugin="org.eclipse.core.boot"/>
      <import plugin="org.eclipse.core.resources"/>
   </requires>

   <extension
         point="org.eclipse.hyades.logging.parsers.logParser">
         <parser
            name="Sample Rules Adapter for Sample Application"
            icon=""
            description="%STR_MYAPP_PARSER_DESCRIPTION"
            class="com.ibm.itso.sg246665.LogParser.AppLogParser"
            ui_name="Sample Rules Adapter for Sample Application"
            id="com.ibm.itso.sg246665.LogParser.AppLogParser">
         <field
               useBrowse="true"
               defaultValue="./GLAAdapter/sample.log"
               name="File Name"
               helpContextId=""
               tooltip="%STR_MYAPP_TOOLTIP1"
               id="file_path"
               browseType="*.log">
         </field>
         <field
               useBrowse="false"
               defaultValue="MyApp 1.0(rules)"
               name="Supported versions"
```

```
                helpContextId=""
                tooltip="%STR_MYAPP_TOOLTIP2"
                ui_type="combobox"
                id="version">
          </field>
          <parserParameter
                name="MyApp 1.0(rules)"
                value="./GLAAdapter/simpleGLA.adapter">
          </parserParameter>

       </parser>
    </extension>


</plugin>
```

The following descriptions explain the extension defined in Example 5-3 on page 171:

– The <parser> tag is the high-level information for the parser. The attributes are:

name	The name of this parser. This can be the same as ui_name.
description	A description of the parser that will be shown in the Import Log File wizard when this parser is selected.
icon	The icon that will be shown in the Import Log File wizard for this parser. No icons are used in this example.
class	The full class name of the parser wrapper class.
ui_name	The name of this parser that will be included in the Selected log file list in the Import Log File wizard.
id	A unique identifier for this parser extension point.

– The first <field> tag of the plugin.xml file in Example 5-3 on page 171 describes the location of the log file that needs to be parsed and viewed in the Log and Trace Analyzer. In this example, we use the sample.log file. The sample.log file is in the GLAAdapter folder of the ITSOParser project. This file is provided as a default log file that needs to be parsed.

– The second <field> tag describes the version of the adapter configuration file used for parsing. There is only one version of the rule adaptor file that we create in this example, named `MyApp 1.0(rules)`.

– The <parserParameter> tag is used to specify a GLA adapter configuration file.

Figure 5-32 shows the Import Log file wizard with this plug-in.

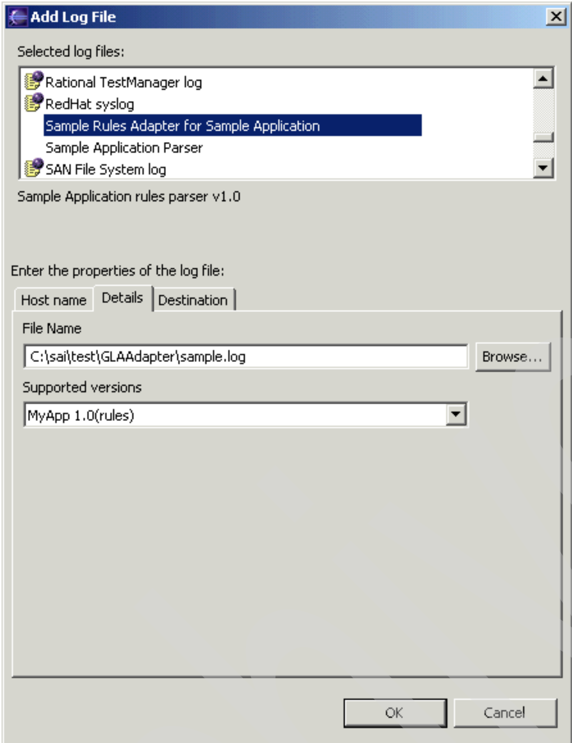


Figure 5-32 Import Log wizard with custom log parser added

7. Create a plugin.properties file in the ITSOParser project. The plugin.properties file can be used to define the substitution variables used in the plugin.xml file. Example 5-4 shows the plugin.properties file.

Example 5-4 Defining the substitution variables for the plugin.xml file

```
# Properties for ITSOParser Plugin

pluginName = ITSOParser
providerName = IBM

# logParser extension point message(s):
STR_MYAPP_PARSER_DESCRIPTION = Sample Application rules parser v1.0
STR_MYAPP_TOOLTIP1           = Location of the log file
STR_MYAPP_TOOLTIP2           = Version of the log file to import
```

8. In Package Explorer, right-click **ITSOParser** and select **New** → **Class**. Enter the Package and Name for the new class. The class should extend org.eclipse.hyades.logging.adapter.config.StaticParserWrapper. Figure 5-33 shows the creation of `AppLogParser` for this example.

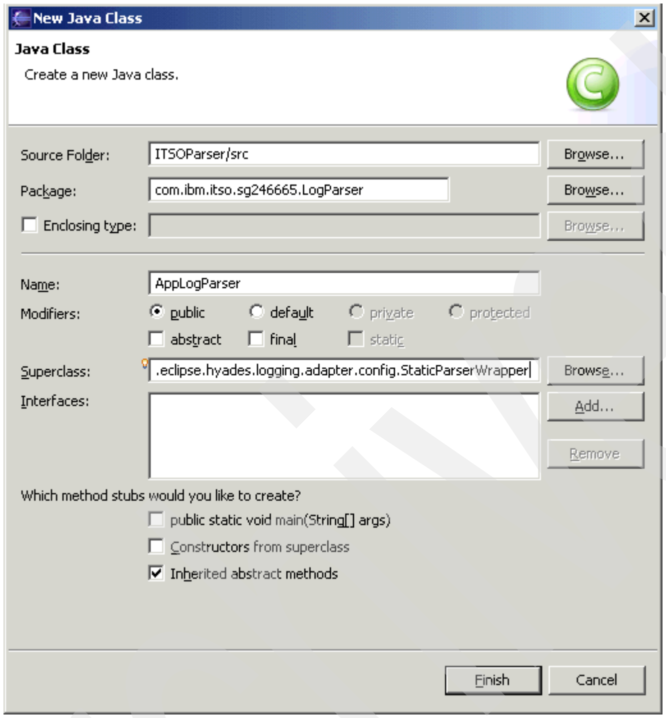


Figure 5-33 Create the AppLogParser class

9. Click **Finish** and add a method named *AppLogParser*. The complete AppLogParser class now is as shown in Example 5-5.

Example 5-5 The AppLogParser class

```
package com.ibm.itso.sg246665.LogParser;

import org.eclipse.hyades.logging.adapter.config.StaticParserWrapper;

/**
 * @author ITSO Redbook Team
*/
public class AppLogParser extends StaticParserWrapper {
   public AppLogParser(){
      super();
      currentPlugin="ITSOParser";
   }
}
```

5.3.2 Creating the adapter configuration file

In the plugin.xml file of the ITSOParser project, we specified the simpleGLA.adapter GLA configuration file for parsing the log file called sample.log. In this section, we modify the GLA configuration file created in 4.2, “Creating a simple adapter” on page 83. The modification will allow the GLA configuration file to send the CBE events to a logging agent outputter.

The following items need to be configured in the GLA configuration file:

- ► The outputter component for the context
- ► The context instance outputter parameters

The outputter component type must be set to StaticParserOutputter, which writes the externalized records to a logging agent known to the Log and Trace Analyzer. The logging agent will be attached to and monitored by the Log and Trace Analyzer automatically during the import logs task. The StaticParserOutputter executable class is org.eclipse.hyades.logging.adapter.config.outputters.StaticParserOutputter.

Figure 5-34 shows the outputter component configuration in the context of the adapter file.

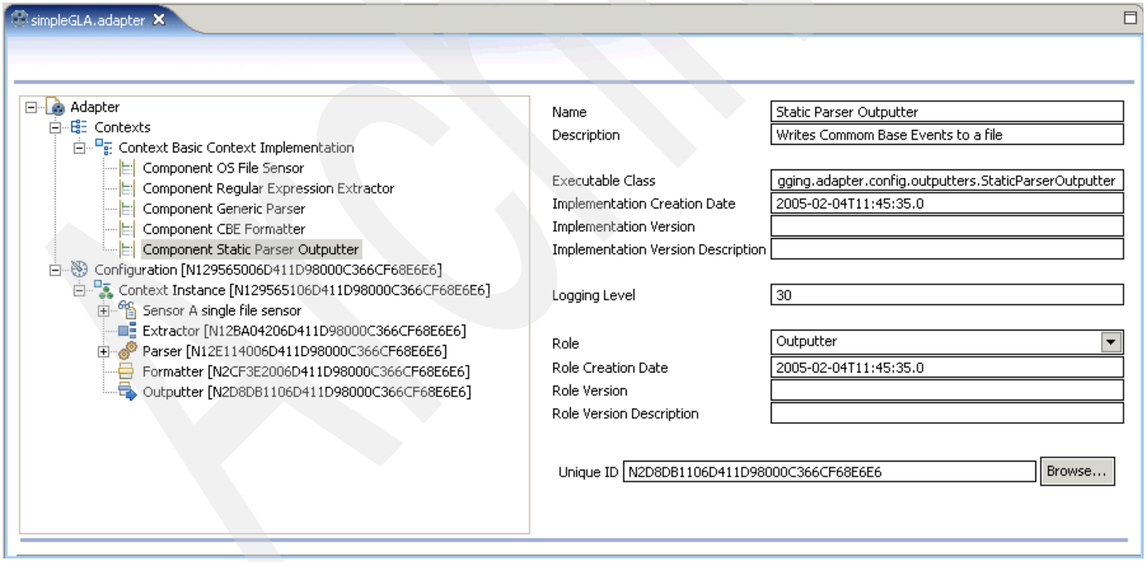


Figure 5-34 Outputter configuration for the log parser

Because StaticParserOutputter writes the Common Base Event into a logging agent, the Outputter Type in the Context Instance can be **LoggingAgentOutputter**, as shown in Figure 5-35 on page 176.

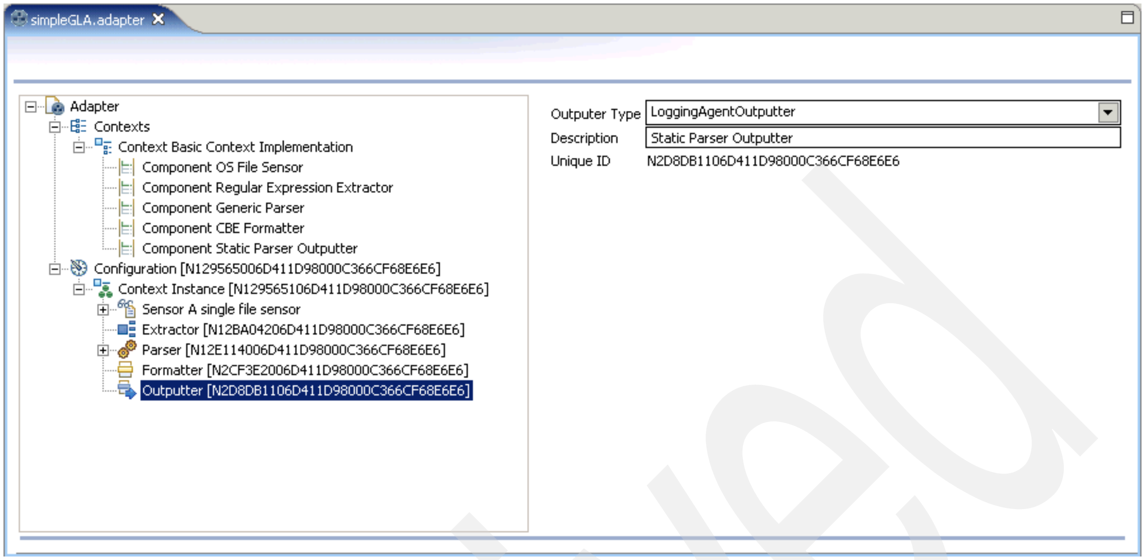


Figure 5-35 The Outputter configuration in Context Instance

5.3.3 Testing the log parser

Now that we have created the log parser named ITSOParser, perform the following steps to test the new log parser with the Log and Trace Analyzer tool:

1. Select **ITSOParser** from the Package Explorer, and then click **Run** → **Run As** → **Run-time Workbench**. A new Eclipse session will start.
2. On this new Eclipse session, select the **Profiling and Logging** perspective, and then click **File** → **Import**.
3. Select **Log File** and click **Next**.
4. Click **Add**, which will show the Add Log File wizard.

5. In the Add Log File wizard, select the parser name. In our example, we select the newly created **Sample Rules Adapter for Sample Application** parser. Select the Host name, and in the Details tab, select the log file to be parsed (sample.log in this example), as shown in Figure 5-36.

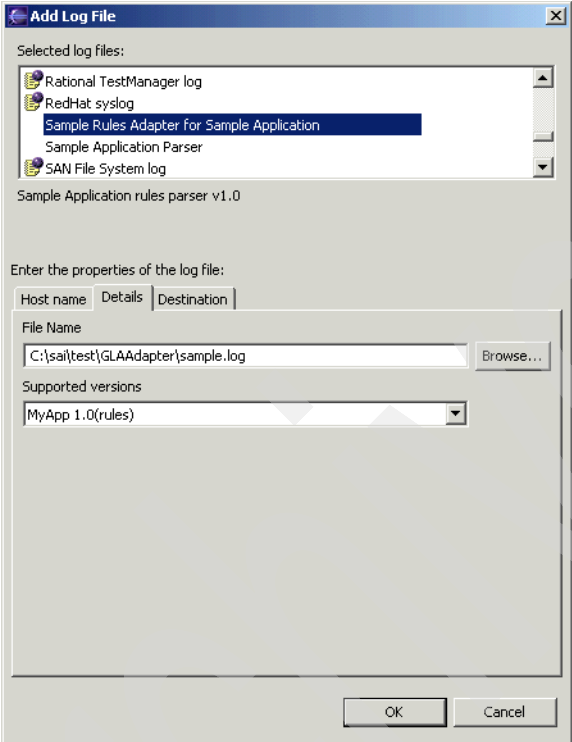


Figure 5-36 Import Log wizard with the custom log parser added

6. Click **OK** and then **Finish**. The log records from the from sample log event are displayed in the Log View as Common Base Events, as shown in Figure 5-37 on page 178.

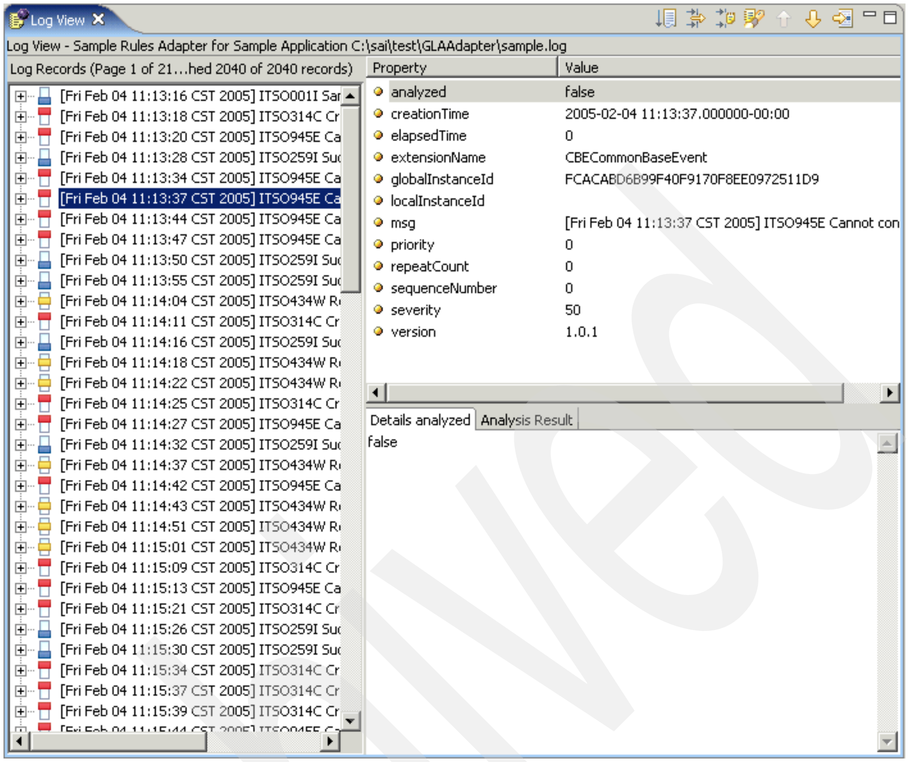


Figure 5-37 Log View using custom parser

5.3.4 Deploying the custom parser

To deploy the newly created custom parser into an Eclipse environment, it needs to be packaged first. Perform the following steps to package the log parser:

1. Open the ITSOParser plug-in project in Eclipse, and open the plugin.xml file.
2. Click **build.properties**, and add `GLAAdapter` and `plugin.properties`, as shown in Example 5-6.

Example 5-6 Modified build.properties

```
source.ITSOParser.jar = src/
output.ITSOParser.jar = bin/
bin.includes = GLAAdapter/,\
               plugin.xml,\
               plugin.properties,\
               ITSOParser.jar
```

3. Export the plug-in files by selecting **File** → **Export**.

4. Select **Deployable plug-ins and fragments** and click **Next**.
5. Select the plug-in from the Available Plug-ins and Fragments list, select Deploy as **a single ZIP file**, and enter the Destination File name, as shown in Figure 5-38.

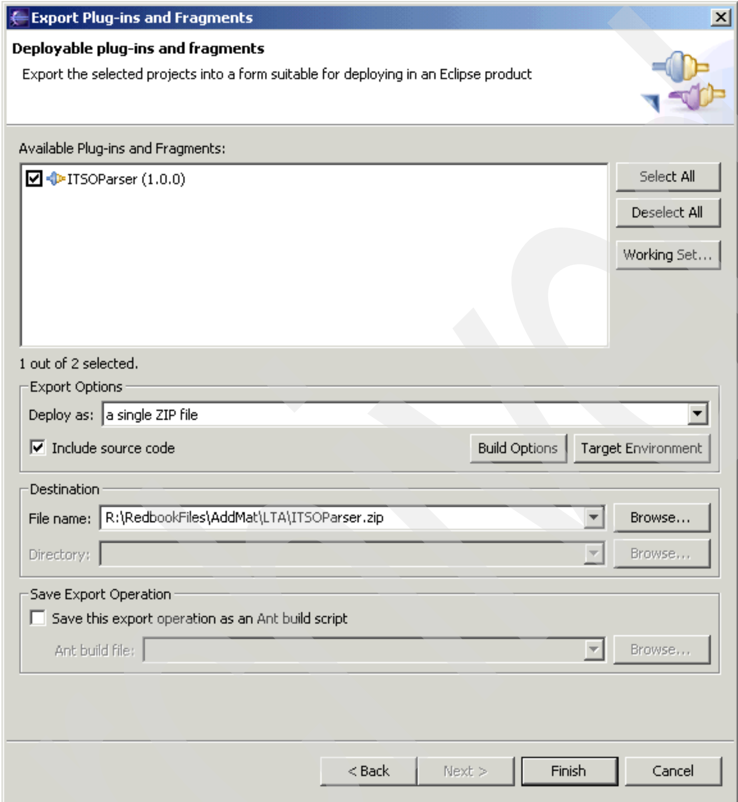


Figure 5-38 Export the Plug-ins and Fragments

6. Unzip the ZIP file that we created (in this example, ITSOParser.zip) in the plugin directory of the Eclipse installation. Now, it is ready to run the newly created log parser plug-in using the Log and Trace Analyzer.

5.4 Creating a custom correlator for the Log and Trace Analyzer

In this section, we present an example of a correlation engine created to correlate the log records based on the severity level.

To create a custom correlation engine for LTA, you must extend the org.eclipse.hyades.logc.logInteractionView extension point. This extension point is defined in the org.eclipse.hyades.logc plug-in.

There are two classes that we create for this custom correlator engine:

- com.ibm.itso.sg246665.correlator.SevCorEngine

  This class defines the correlation logic and implements the org.eclipse.hyades.logc.extensions.ILogRecordCorrelationEngine interface.

- com.ibm.itso.sg246665.correlator.SimpleParserFilter

  This class is used to filter the log records that need to be correlated. Although our example does not need filtering log records, this class is used, because it is mandatory for the correlator. This class implements the org.eclipse.hyades.logc.extensions.ILogRecordFilter interface.

In our example, the correlation engine reads an IBM DB2 Universal Database™ diagnostic log and an application log, which natively creates a log into the Common Base Event format, as shown in Figure 5-39.

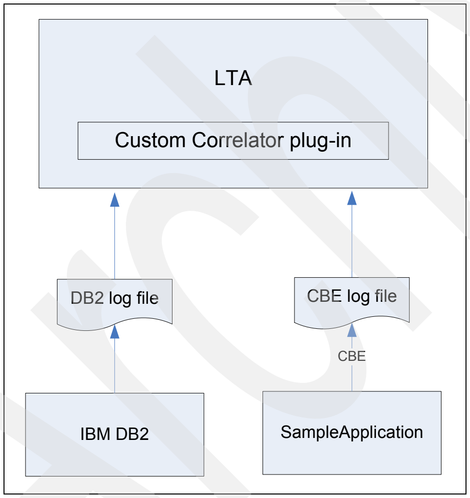


Figure 5-39 Custom correlator example scenario

The following section describes the steps needed to create the custom correlation engine based on the severity level.

5.4.1 Creating the custom correlation plug-in

The following steps describe the creation of a plug-in to the Log and Trace Analyzer tool for defining the custom correlator:

1. Start Eclipse.
2. Create a plug-in project in Eclipse. To create a plug-in project, click **File** → **New** → **Project**, expand **Plug-in Development**, and select **Plug-in Project**. Click **Next**.
3. Enter the Project Name, for example, `LogCorrelator`, and click **Next**.
4. Clear the **Generate the Java class that controls the plug-in's life cycle** option, as shows in Figure 5-40, and then click **Finish**. The project will open in the Plug-in perspective.

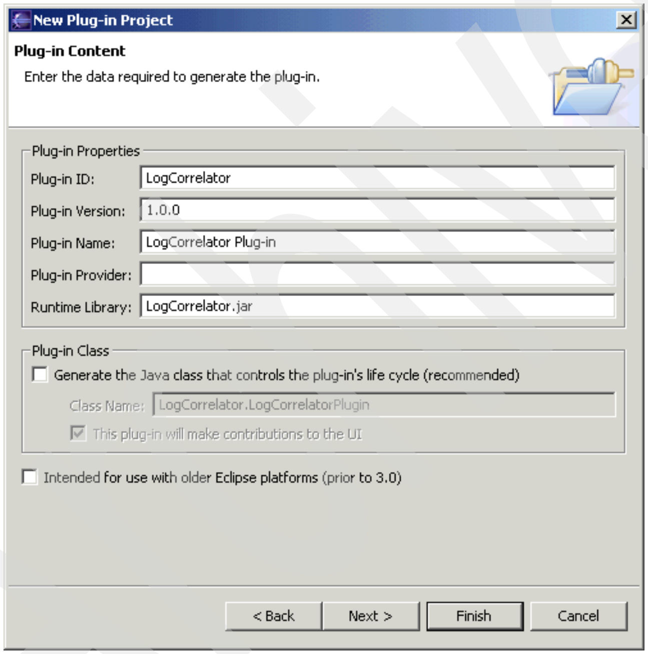


Figure 5-40 Create the Plug-in Project for the correlator

5. Click the **Dependencies** tab and click **Add**. You are required to add the following plug-ins in the dependencies:
   - org.eclipse.emf.common
   - org.eclipse.emf.ecore
   - org.eclipse.hyades.logc
   - org.eclipse.hyades.models.cbe
   - org.eclipse.hyades.models.hierarchy

6. Click the **plugin.xml** tab and add an extension to the plug-in to define the LogRecordFilter and LogRecordCorrelationEngine, as listed in Example 5-7.

Example 5-7 Listing of the plugin.xml file for the LogCorrelator

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="LogCorrelator"
   name="LogCorrelator Plug-in"
   version="1.0.0"
   provider-name="">

   <runtime>
      <library name="LogCorrelator.jar">
         <export name="*"/>
      </library>
   </runtime>
   <requires>
      <import plugin="org.eclipse.emf.common"/>
      <import plugin="org.eclipse.emf.ecore"/>
      <import plugin="org.eclipse.hyades.logc"/>
      <import plugin="org.eclipse.hyades.models.cbe"/>
      <import plugin="org.eclipse.hyades.models.hierarchy"/>
   </requires>
   <extension
            point="org.eclipse.hyades.logc.logInteractionView">
            <view
                    log_types="%LOG_TYPES"
                    name="%CORR_NAME"
                    description="%CORR_DESC">

                    <LogRecordFilter
                            log_type="%LOG_TYPE_SIMPLE"

class="com.ibm.itso.sg246665.correlator.SimpleParserFilter">
                    </LogRecordFilter>

                    <LogRecordCorrelationEngine

class="com.ibm.itso.sg246665.correlator.SevCorEngine">
                    </LogRecordCorrelationEngine>
            </view>
     </extension>

</plugin>
```

7. In the Package Explorer, right-click the plug-in project and select **New** → **File**. Enter the file name as `plugin.properties`. Define the variables used in plugin.xml in this file. Example 5-8 lists the plugin.properties for this sample.

Example 5-8 Listing of plugin.properties

```
# The following is a list of variables used for the 'custom.SevCorEngine'
plugin
CORR_NAME = Severity correlation
CORR_DESC  = Correlates log records using the date.

# This variable holds a listing of the log file that this correlation engine
# support.
# Multiple log files must be separated by a comma.
LOG_TYPES  = Common Base Event XML Log,IBM DB2 Universal Database diagnostic
log
LOG_TYPE_SIMPLE = Common Base Event XML log,IBM DB2 Universal Database
diagnostic log
```

8. Create the class com.ibm.itso.sg246665.correlator.SevCorEngine. Right-click **LogCorrelator** and select **New** → **Class**. Enter the Package, Name, and Interface for the new class, as shown in Figure 5-41 on page 184.

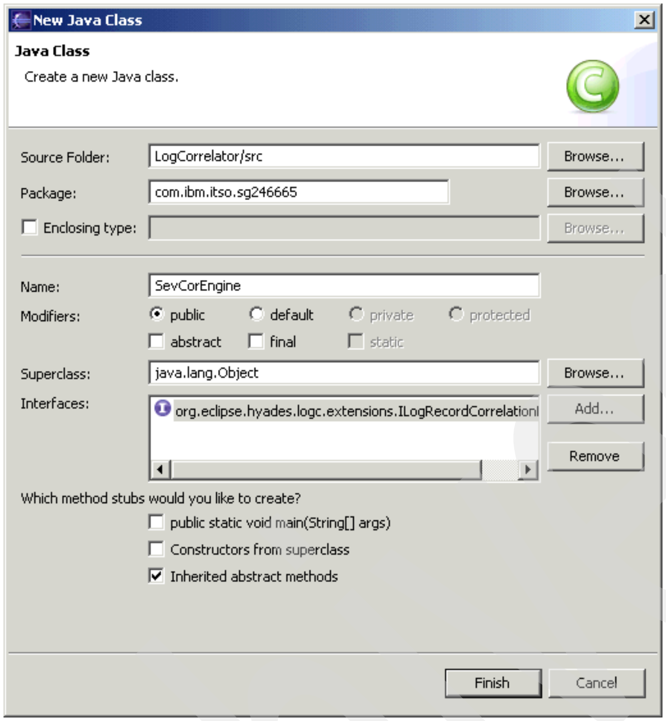


Figure 5-41 Create the SevCorEngine class

9. Click **Finish**. This will create a SevCorEngine Java class that implements the org.eclipse.hyades.logc.extensions.ILogRecordCorrelationEngine interface. Now, add the logic for the correlation into this class. Example 5-9 lists the Java class for the correlation by severity.

Example 5-9 Listing of the SevCorEngine class

```
package com.ibm.itso.sg246665.correlator;

import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.common.util.BasicEList;
import org.eclipse.hyades.logc.extensions.ICorrelationMonitor;
import org.eclipse.hyades.logc.extensions.ILogRecordCorrelationEngine;
import org.eclipse.hyades.models.hierarchy.CorrelationContainerProxy;
import org.eclipse.hyades.logs.correlators.RecordList;
import org.eclipse.hyades.models.cbe.CBECommonBaseEvent;
import org.eclipse.hyades.models.hierarchy.CorrelationContainer;
import org.eclipse.hyades.models.hierarchy.CorrelationEngine;


/**
```

```
 * @author ITSO Redbook Team
 *
 */
public class SevCorEngine implements ILogRecordCorrelationEngine {

    private CorrelationEngine correlationEngine = null;
    private CorrelationContainer correlationContainer = null;

    /* The name and type of the simple correlation engine */
    private final String CORRELATION_NAME = "Severity correlation";
    private final String CORRELATION_TYPE = "Correlated";

    /*
     * This function makes the correlation between the log records.
     */

   public void correlate(CorrelationContainerProxy correlationContainerProxy,
EList logFiles,
      ICorrelationMonitor mon) {

       correlationEngine = correlationContainerProxy.getCorrelationEngine();
        correlationContainer =
correlationContainerProxy.getCorrelationContainer();

        if (correlationEngine != null) {
            correlationEngine.setType(CORRELATION_TYPE);
            correlationEngine.setName(CORRELATION_NAME);
            correlationEngine.setId(CORRELATION_NAME);
        }

        /* Traverse through each of the log files that a correlation
         * needs to be made
*/
        for (int i = 0; i < logFiles.size(); i++) {
            /* For each of the existing log file, traverse through its log
records
             * and make the necessary correlations                     */
            if (logFiles.get(i) != null) {
                /* Store the list corresponding to the log records of the i-th
                 * logFile
*/
                EList recordList = ((RecordList) logFiles.get(i)).getList();

                /* Make the necessary correlations */
                makeCorrelations(recordList, logFiles, i);
            }
        } // End of for-loop
    } // End of correlator (EList)
```

```
    /*
     * A helper method to 'correlate(EList)' that is used to make the
necessary
     * correlations between the log records
     *
     * @param recordList - A list of log records for which correlations are
made
     * logFiles - 'logRecord' belongs to a log file listed under this
parameter
     * logFileIndex - The index of 'logFiles' identifying the log file that
     * logRecord belongs to
     */
    private void makeCorrelations(EList recordList, EList logFiles,
                                                  int logFileIndex) {

        /* Traverse through each of the log records and make the
         * necessary correlations
*/
        for (int j = 0; j < recordList.size(); j++) {
            /* Make the correlation for the j-th log record */
            setPartners(recordList.get(j), logFiles, logFileIndex);
        } // End of for-loop
    } // End of makeCorrelations (EList)

    /*
     * A helper method to 'makeCorrelations (EList, EList, int)' used to set
     * the partners
     *
     * @param logRec        - The log record for which partners are set
     *        logFiles      - 'logRecord' belongs to a log file listed under
this
     *                          parameter
     *        logFileIndex  - The index of 'logFiles' identifying the log file
that
     *        logRecord belongs to
     */
    private void setPartners(Object logRec, EList logFiles, int logFileIndex)
{

        /* The log records are mapped to a Common Base Event */
        CBECommonBaseEvent logRecord = (CBECommonBaseEvent) logRec;

        /* The list of records that will be checked */
        EList recordList = null;

        /* The correlators list and the correlated records of 'logRecord' */
        EList correlators = null;

        /* Get the message severity of the passed log record */
```

```
        short msgSeverity = getMsgSeverity(logRecord);

        /* Traverse through the logFiles (starting from index 'logFileIndex')
and
        *  make the proper correlations*/
        for (int i = logFileIndex; i < logFiles.size(); i++) {

            /* The record list of the i-th logFile */
            recordList = ((RecordList) logFiles.get(i)).getList();

            /* Traverse through the record list of the i-th log file and make
the
             * proper correlations */
            if (recordList != null) {
                for (int j = 0; j < recordList.size(); j++) {
                    if (recordList.get(j) != null) {
                      if (recordList.get(j) != logRec) {
                       if(getMsgSeverity(recordList.get(j))
                           == msgSeverity) {
                         addCorrelation((CBECommonBaseEvent)
                                         logRec, (CBECommonBaseEvent)
                                         recordList.get(j));
                       }
                } // End of the j-th loop
            }
          }
            }
        }// End of the i-th loop
    } // End of setPartners (Object, EList, int)

    private EList addCorrelation(CBECommonBaseEvent artifact,
    CBECommonBaseEvent associatedEvent) {

        EList correlations = (EList)

correlationContainer.getCorrelations().get(artifact);

        if (correlations == null) {
            correlations = new BasicEList();
            correlations.add(associatedEvent);
            correlationContainer.getCorrelations().put(artifact,
correlations);
        }
        else {
            correlations.add(associatedEvent);
        }

        return correlations;
    }
```

```
    /*
     * Returns the severity associated with a log record
     */
    private short getMsgSeverity(Object logRecord) {
       short sev = ((CBECommonBaseEvent) logRecord).getSeverity();
       return sev;
   }

}
```

10. Now, repeat this process to create the class com.ibm.itso.sg246665.correlator.SimpleParserFilter, implementing the org.eclipse.hyades.logc.extensions.ILogRecordFilter interface. Example 5-10 lists the class created for this example.

Example 5-10 Listing of the SimpleParserFilter class

```
package com.ibm.itso.sg246665.correlator;

import org.eclipse.emf.common.util.EList;
import org.eclipse.hyades.logc.extensions.ILogRecordFilter;

/**
 * @author ITSO Redbook Team
 */
public class SimpleParserFilter implements ILogRecordFilter {

   /* (non-Javadoc)
    * @see
org.eclipse.hyades.logc.extensions.ILogRecordFilter#filter(org.eclipse.emf.comm
on.util.EList)
    */
   public EList filter(EList arg0) {
      return arg0;
   }

}
```

5.4.2 Testing the custom correlation engine

The following steps describe how to test the newly created correlation engine, LogCorrelator:

1. In the Package Explorer, click **Log Correlator**.
2. Select **Run** → **Run As** → **Run-time Workbench**.

3. In the Profiling and Logging perspective of the Run-time Workbench, click **File** → **Import**, select **Log File**, and click **Next**.
4. Click **Add** and click **IBM DB2 Universal Database diagnostic log**. From the Details tab, and **select the db2diag.log** and click **OK**. Use the same process to open the second log file. In our example, we selected our sample application log file.
5. **Common Base Event XML log** has to be selected from the Selected log files list as the application log used in this example named CBEApplicationLogfile.log created by ScenarioApplication. Figure 5-42 shows the files to import for the correlation.

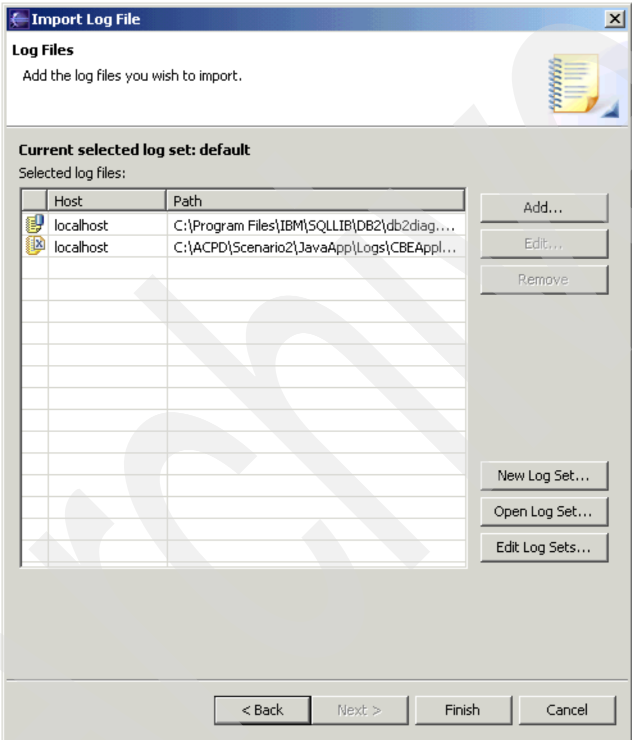


Figure 5-42 Import the log files for correlation

6. Click **Finish**.
7. In the Log Navigator, clear the **Hide Hosts** option, as shown in Figure 5-43 on page 190.

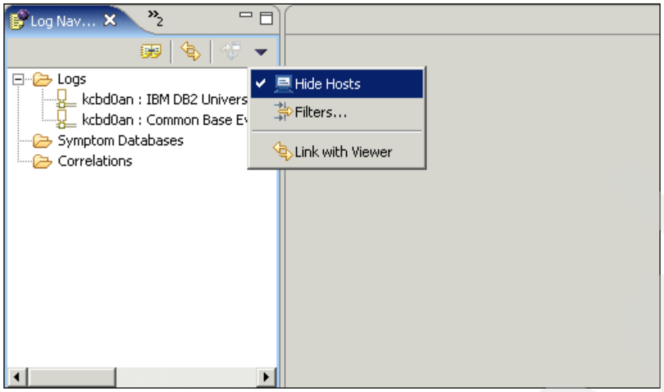


Figure 5-43 Change the hosts in the Log Navigator

8. Right-click the host name in the Log Navigator, and then select **Open With → Log Interactions**.
9. Select the newly created **Severity correlation** from the list of correlations, as shown in Figure 5-44.

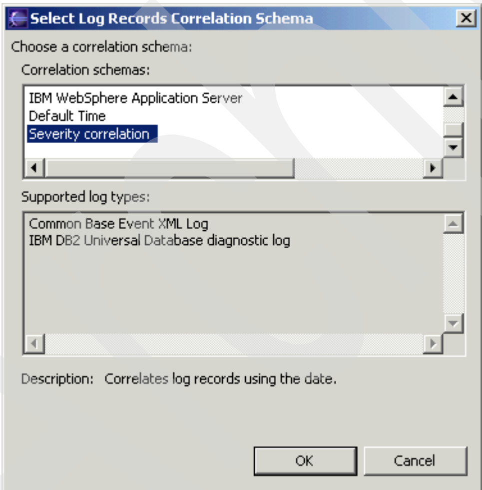


Figure 5-44 Choose the Correlation schema

Now, the correlation will appear in the UML2 Sequence diagram in the right pane.

5.4.3 Deploying the custom correlation

Now that we have created and tested the correlation engine for the severity level, we deploy the custom correlation. Perform the following steps to deploy the custom correlation engine LogCorrelator:

1. Edit the build.properties file to include the plugin.properties in the package, as shown in Example 5-11, and save the file.

Example 5-11 The build.properties file for the correlation engine

```
source.LogCorrelator.jar = src/
output.LogCorrelator.jar = bin/
bin.includes = plugin.properties,\
           plugin.xml,\
             LogCorrelator.jar
```

2. Select the **LogCorrelator** project in Package Explorer.
3. Click **File** → **Export**, select **Deployable Plug-ins and Fragments**, and click **Next**. Select Deploy as **a single ZIP file** and enter the Destination File name. Click **Finish**.
4. Unzip the package under the plugins directory of Eclipse.

6

Autonomic Management Engine and resource models

The Autonomic Management Engine is a fully functional example implementation of an autonomic manager. It uses resource models created by the IBM Tivoli Resource Model Builder tool to provide customized autonomic behavior.

We discuss the following topics in this chapter:

- Autonomic Management Engine: This section focuses on the purpose of the Autonomic Management Engine and how it works.
- Resource models: This section describes the purpose of the resource model and provides basic template details.
- Simple Agent Reference Application (SARA): This section provides a high-level view of the different components and installation of a resource model.
- How to debug a resource model: This section contains a description about how to debug a resource model using the SARA runtime environment.

6.1 Autonomic Management Engine

Without autonomic computing, even known potential problems can cause a long system down time due to the need of human interaction to solve the problem. The time to repair an incident might take long as shown in Figure 6-1.

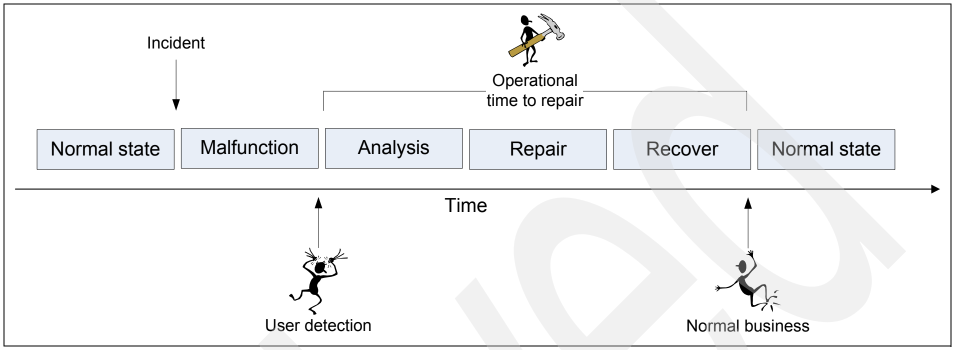


Figure 6-1 Human handling of problems

Using autonomic computing technologies, specifically an autonomic manager, the time to repair an incident is shortened. In addition, the user might not detect interruption of services, as shown in Figure 6-2.

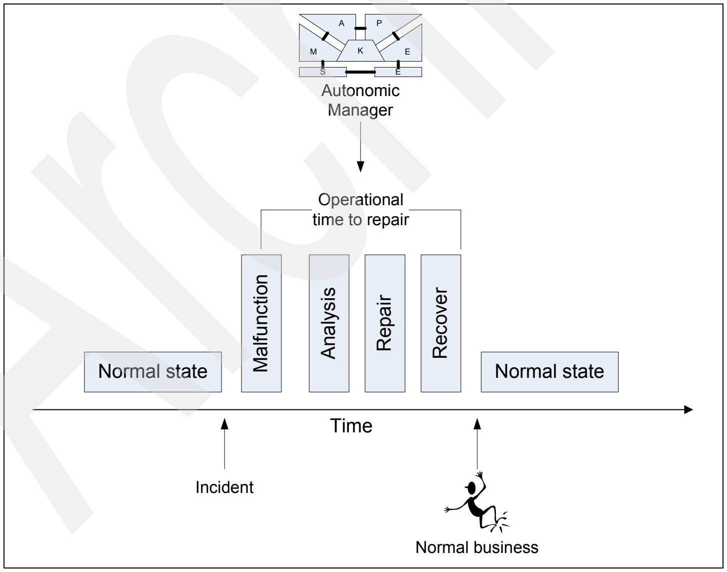


Figure 6-2 Autonomic Management Engine handling of problems

As can be seen by comparing Figure 6-1 on page 194 and Figure 6-2 on page 194, the system might be down for a long time with human interaction, while the autonomic manager is capable of handling the problem as it appears. In the best cases, the user will not even experience the problem.

The IBM Autonomic Computing Toolkit provides a functional example implementation of an autonomic manager, the Autonomic Management Engine.

The following sections provide an introduction to the Autonomic Management Engine and describe its architecture for a better understanding of how it uses resource models created by the Resource Model Builder (RMB) tool to provide customized autonomic behavior.

6.1.1 Introduction to Autonomic Management Engine

The Autonomic Management Engine is an autonomic manager implementation that monitors system resources, sends aggregated events, and performs corrective actions for problems.

The central functionality of the Autonomic Management Engine is the intelligent control loop, also called MAPE-K, as shown in Figure 6-3 on page 196. The intelligent control loop is executed at a regular interval. It consists of the following parts:

Monitor	Provides the mechanisms that collects, aggregates, and filters the events, metrics, and topologies collected from a managed resource using sensors.
Analyze	Provides the mechanisms to correlate and model complex situations to identify situations where corrective actions are needed.
Plan	Provides the mechanisms to structure the needed actions based on business policies.
Execute	Provides the mechanisms for execution of a plan with considerations for on-the-fly updates. The corrective actions are executed by effectors.
Knowledge	Is the knowledge shared by the four main parts.

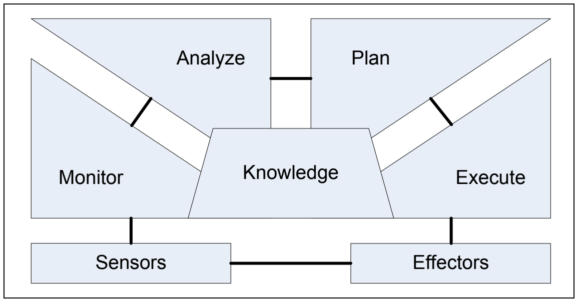


Figure 6-3 Autonomic Management Engine control loop

The Autonomic Management Engine provides a hosting environment for the resource model decision algorithms. The decision algorithms in the resource model are created using JavaScript, for which the Autonomic Management Engine is responsible for providing a JavaScript engine.

One of the important tasks of the Autonomic Management Engine is to provide a service object for the decision algorithm. The service object API is used to access various information such as the state of the monitored resource. A detailed description of the service object API information can be found in *IBM Tivoli Monitoring Resource Model Builder User's Guide*, SC32-1391.

The IBM Autonomic Computing Toolkit is shipped with a sample Autonomic Management Engine named SARA, for Simple Agent Reference Application. SARA has a core implementation that provides all the basic building blocks for running a resource model. SARA consists of a core package and a front-end command-line interface. We discuss SARA later in this chapter in 6.3, "Simple Agent Reference Application (SARA)" on page 222.

The Autonomic Management Engine can be used as a stand-alone management engine or embedded into applications. The stand-alone method uses SARA to provide the environment to run custom resource models. Chapter 7, "Advanced Autonomic Management Engine topics" on page 231 provides detailed information about how to embed the Autonomic Management Engine into an application. The embedding application is responsible for loading the resource model, initializing instances of the resource models, and providing event sinks and action launcher plug-ins, as required.

The decision algorithm of the resource model (the VisitTree() method) is run by the Autonomic Management Engine at a given cycle time defined by the resource model descriptor. The resource model descriptor is an XML file generated by the Resource Model Builder. In 6.2, "Resource models" on page 199, we provide details about how the Autonomic Management Engine and the resource models are related.

6.1.2 Autonomic Management Engine structure

The Autonomic Management Engine is designed to be highly plugable so that it can be adjusted to almost any environment. It is configured using property files and resource models (RMs). Figure 6-4 shows a high-level view of the structure of the Autonomic Management Engine.

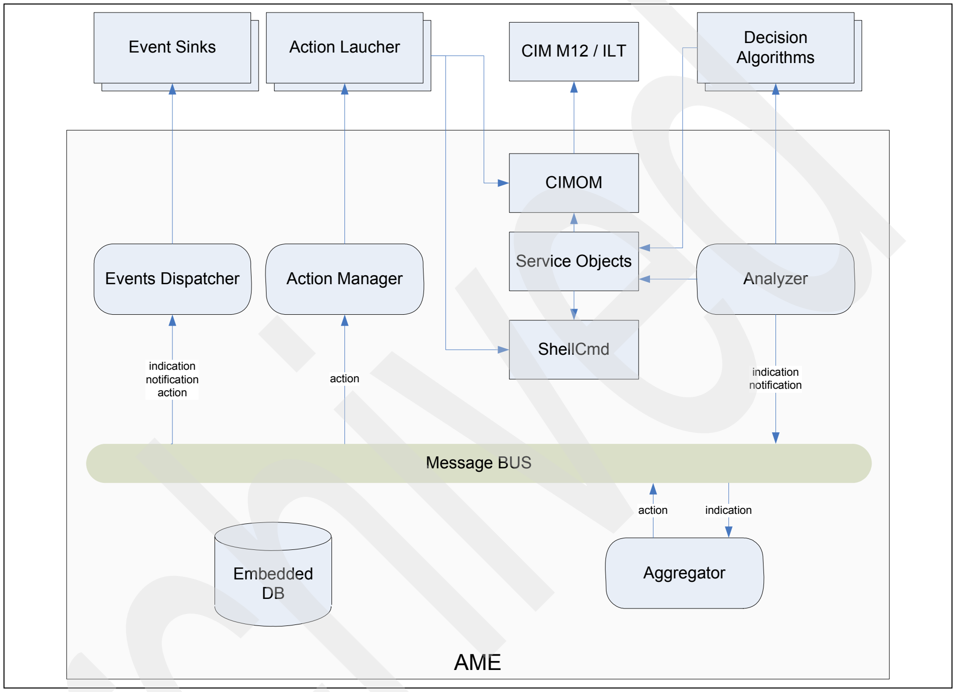


Figure 6-4 Autonomic Management Engine structure

The Autonomic Management Engine is built around a message bus that distributes events to the interested components (event dispatcher, action manager, analyzer, data logger and aggregator). The way each component handles the events is configured in the resource model and is described in 6.2, "Resource models" on page 199.

The events can occur on the message bus in three different types:

- Indication

  An event that could indicate a problem. This occurs when the state of a resource meets a defined criteria.

- Aggregated Indication

  When an indication has occurred a specified number of times, an action event is created. This might trigger a corrective action.

- Clearing

  This event contains the result of an corrective action.

The following sections provide a description of each Autonomic Management Engine component presented in Figure 6-4 on page 197.

Analyzer

The analyzer is responsible for monitoring the resources and issues indication events to the message bus if needed. The analyzer executes the decision algorithms as defined by the resource model. The decision algorithms gather information using the service object; this includes shell commands (ShellCmd) and Common Information Model (CIM) methods. The CIM methods are called through the Common Information Model Object Manager (CIMOM), which is responsible for compiling and registering Managed Object Format (MOF) files and loading the M12 Java providers.

Action manager

The action manager calls action launchers when an aggregated indication event is received from the message bus. Two action launchers are delivered together with the Autonomic Management Engine. The first is for executing shell commands using the ShellCmd. The second is for executing CIM methods using the CIMOM.

Event dispatcher

The event dispatcher sends events to plugable event sinks. The event sinks can be configured to handle specific events. If an event sink is configured to receive a specific event, it will receive all three types (Indication, Aggregated Indication, and Notification). Two event sinks are delivered together with the Autonomic Management Engine, one for IBM Tivoli Enterprise™ Console® and one for IBM Tivoli Business System Manager.

Aggregator

The aggregator aggregates the indication events so that if an indication event has occurred a certain number of times (in consecutive cycles), an aggregated indication event is issued to the message bus. It is possible to configure holes so that, for example, an aggregated indication event is triggered if the indication event occurs two out of three times (one hole).

6.2 Resource models

A resource model is a package defining how an Autonomic Management Engine should handle a specific resource. It contains information about how to collect data about the resource and how to handle the possible events relating to the resource.

Before starting to create resource models, you should consider how much the resource models will cover. Figure 6-5 shows a high-level overview of different components. The advantage of gathering everything in one resource model is that all relevant information can be taken into consideration when analyzing a problem. However, the resource model might become difficult to develop and maintain as it handles several different situations.

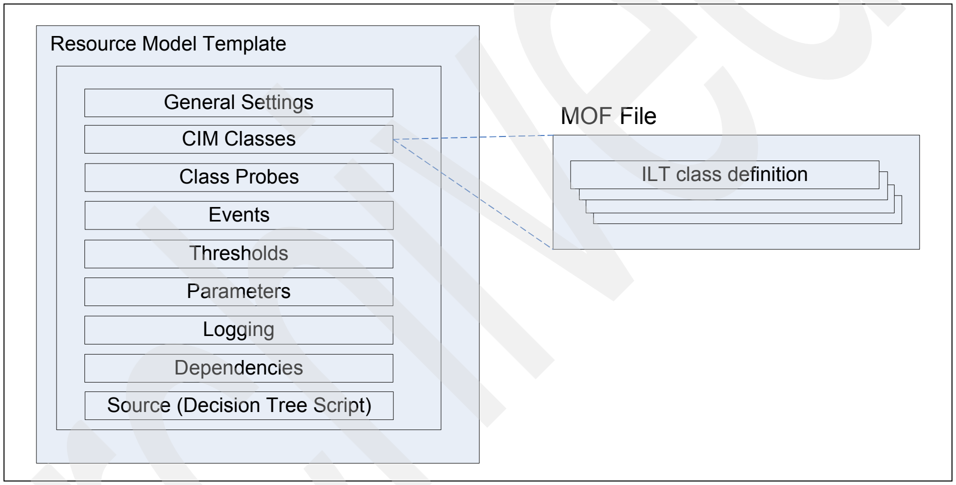


Figure 6-5 Resource model components

The resource model is a combination of various conditions, criteria, and information about the target to be monitored, trigger conditions, events, and actions based on the events. The resource model retrieves information about a monitored resource through various interfaces such as Common Information Model (CIM) and ShellCmd (shell commands). A resource model acts as a bridge between sensors and effectors in the autonomic architecture. In fact, a resource model defines and provides the logic for the monitoring, analysis, plan, and execute phases of the MAPE control loop. Higher-level facilities can be used to provide these services for solutions or complex combinations of individual resources.

Figure 6-6 on page 200 shows how the resource model (RM) components relate to the Autonomic Management Engine (AME) structure.

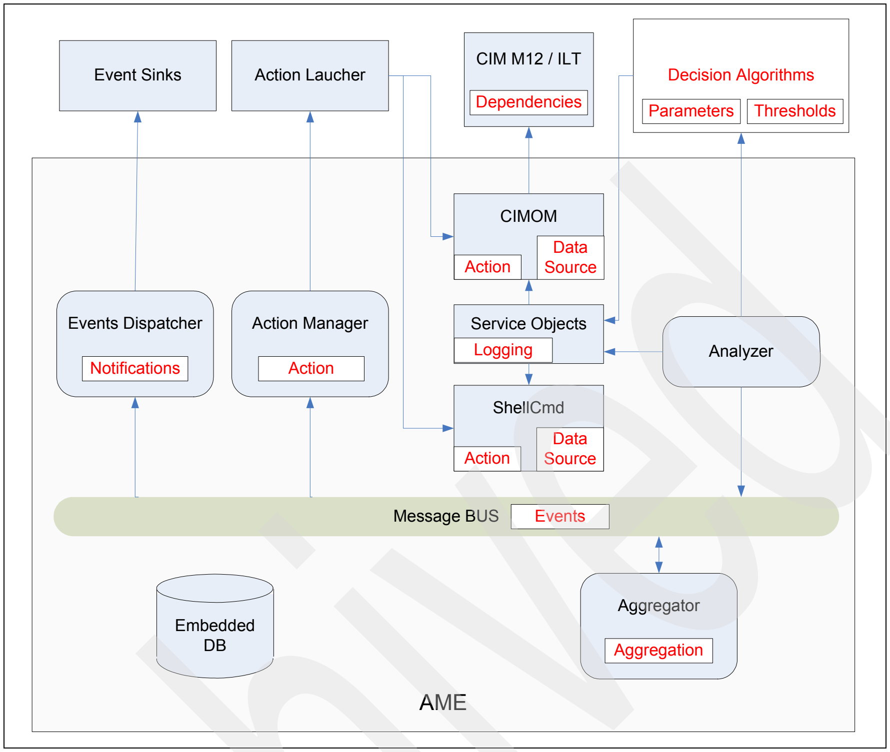


Figure 6-6 Autonomic Management Engine and resource model relationship

The Resource Model Builder has the following building blocks, which are described in the following sections:

- General settings
- Common Information Model (CIM) classes
- Classic probes
- Events
- Thresholds
- Parameters
- Logging
- Dependencies
- Source (decision tree script)

The relationship between the components of the resource model and the building blocks of the Resource Model Builder is shown in Figure 6-7 on page 201.

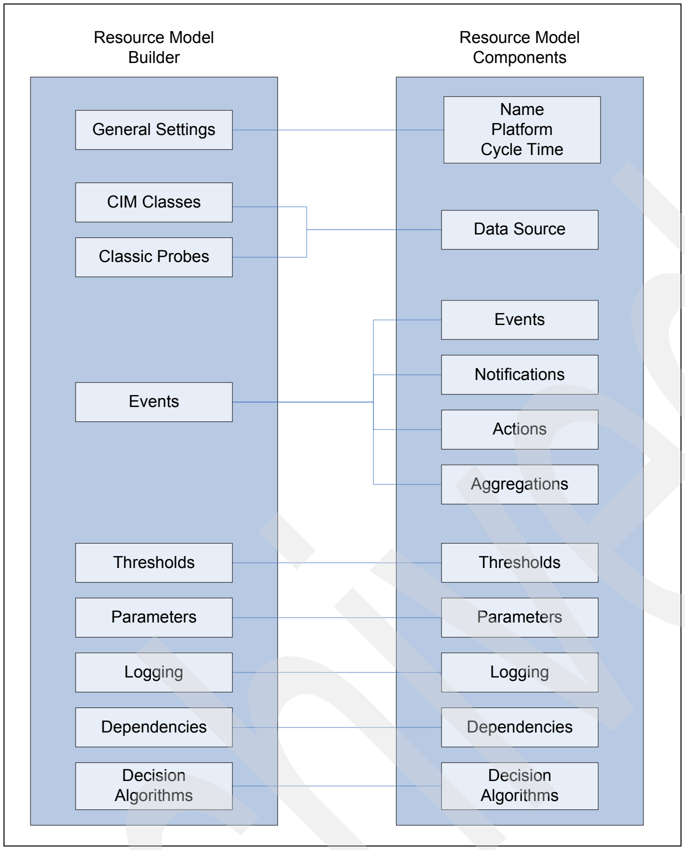


Figure 6-7 Resource model components and Resource Model Builder

To create a resource model, the IBM Autonomic Computing Toolkit provides the Resource Model Builder, as described in 6.2.1, "Resource Model Builder" on page 202. All the components of the resource model are described from the Resource Model Builder point of view.

The following sections provide descriptions of the different components of the resource model and the Resource Model Builder. Refer to 8.2.1, "Creating the Eclipse project" on page 261 for implementation information.

When the resource model is completed, it has to be bundled into a package that can be installed in an Autonomic Management Engine. This is described in Chapter 8, "Problem Determination scenario one" on page 259.

6.2.1 Resource Model Builder

The Resource Model Builder provided with the IBM Autonomic Computing Toolkit is used to build resource models. The IBM Autonomic Computing Toolkit provides an Eclipse-based editor plug-in that can be installed onto an Eclipse framework. The Resource Model Builder (RMB) plug-in tool can be used for creating, modifying, and packaging resource models for use with the Autonomic Management Engine (AME).

Installation

After installing the Eclipse V3.0.1, EMF Plug-in V2.0.1, XSD Plug-in V2.0.1 development kit, install the editor plug-ins provided by the IBM Autonomic Computing Toolkit for the Resource Model Builder.

To install these plug-ins, perform the following steps:

1. Extract the contents of the RMB_v1-1-3_win32.zip file into your Development Kit folder from the prerequisite installation.
2. Click **Yes to All** when prompted to overwrite the files.

The installation of the plug-ins is complete when the copying finishes.

The following sections focus on different components of a resource model in the Resource Model Builder. For a full description of what the Resource Model Builder is capable of and how it works, see *IBM Tivoli Monitoring Resource Model Builder User's Guide*, SC32-1391.

Resource Model Builder general settings

To create an empty resource model using the Resource Model Builder, follow these steps:

1. To start the Eclipse, select **Start** → **Programs** → **IBM Autonomic Computing Toolkit** → **Developmentkit** → **Start Eclipse**.
2. Open the Tivoli Management Perspective by selecting **Window** → **Open Perspective** → **Other** → **Tivoli Management Perspective**.
3. Create a new Tivoli Management Project by selecting **File** → **New** → **Project** and then selecting **Tivoli Management Project**, as shown in Figure 6-8 on page 203.

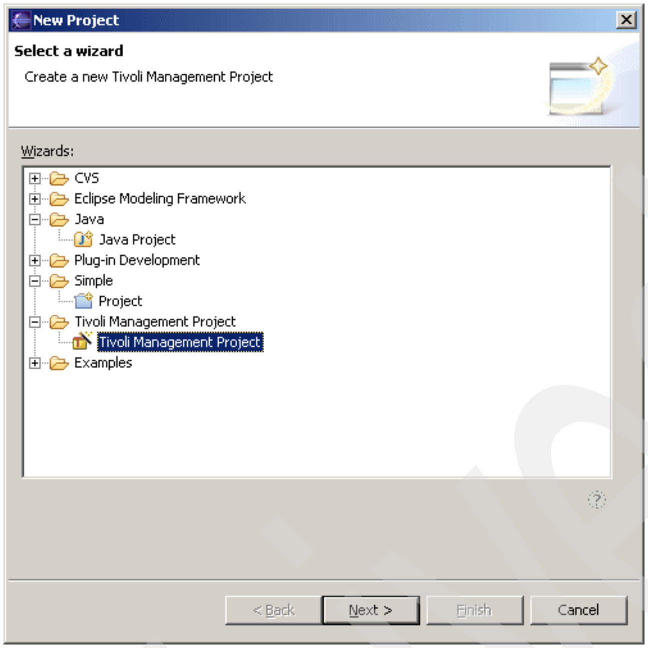


Figure 6-8 New Tivoli Management Project

4. Click **Next**.
5. Enter your Project name, for example, `MyManagementProject`, and click **Next**.
6. Now, it is not possible to start the Basic Resource Model Wizard or the Empty Resource Model Wizard. To create an empty project, just click **Finish**. Your workspace should look similar to the one shown in Figure 6-9 on page 204.

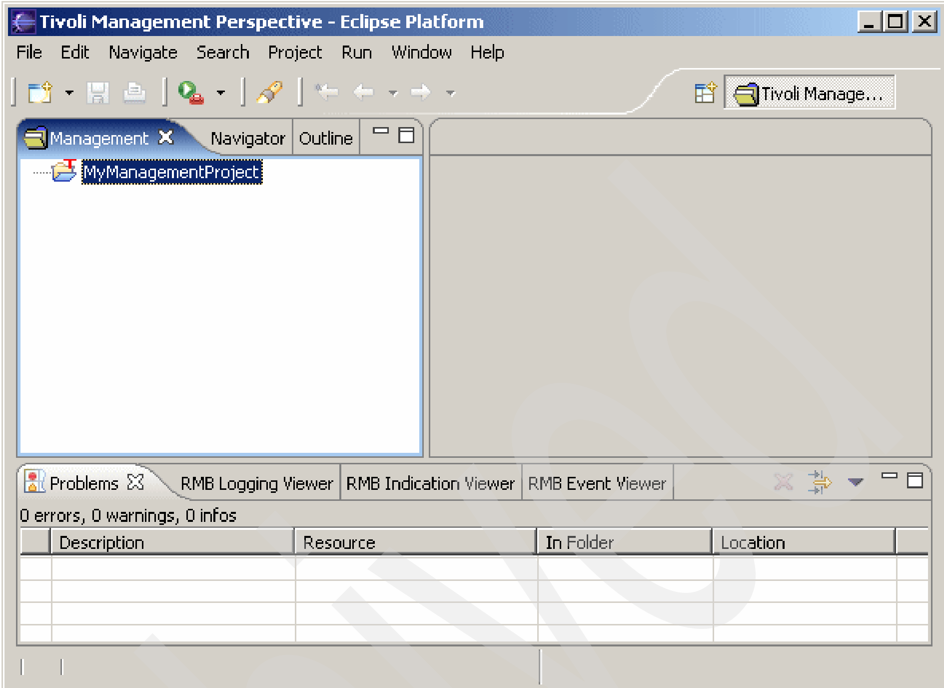


Figure 6-9 Empty project workspace

Now that the project is in place, the resource models can be created. To create an empty resource model, follow these steps:

1. Right-click the project **MyManagementProject** and select **New** → **Empty Resource Model Wizard**. The Empty Resource Model Wizard window opens, as shown in Figure 6-10 on page 205.

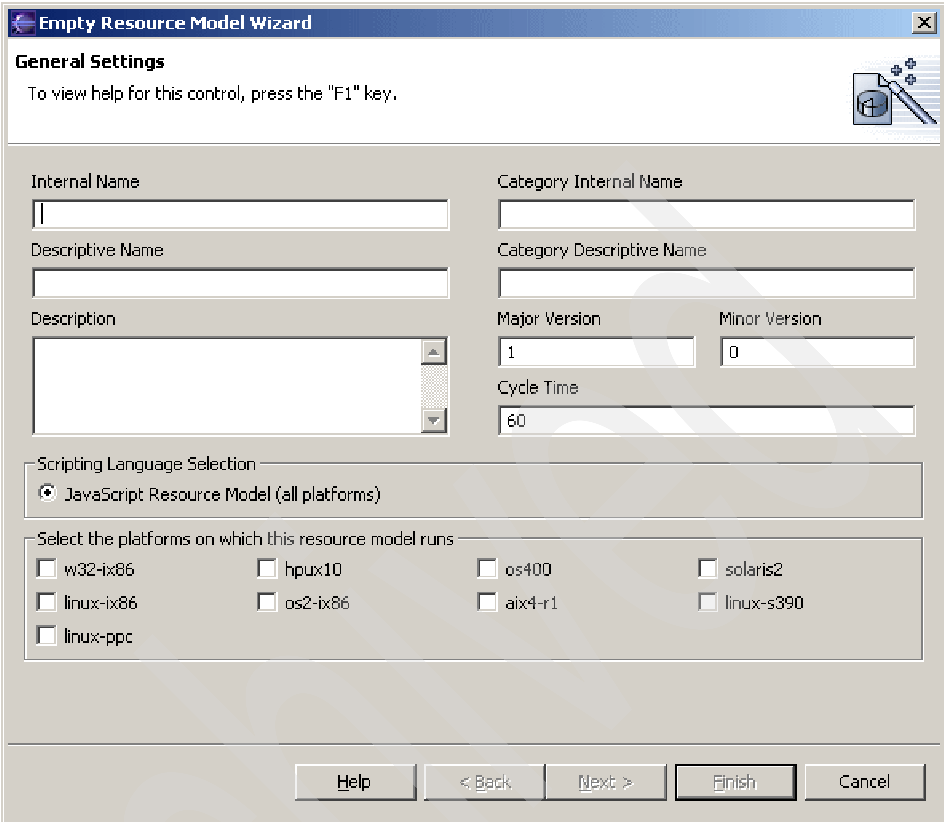


Figure 6-10 Empty Resource Model Wizard

2. Fill in all the fields with the following values and click **Next**:

Internal Name	Internal name of the resource model. This name must be alphanumeric, start with an alphabetic letter, and contain no blanks. For example, `MyStorage`.
Descriptive Name	Name that appears in the IBM Tivoli Monitoring profile list of resource models. For example, `My disk storage`.

Description
: Text that describes the resource model. This is used for documenting the resource model.
For example, `Monitors the free disk space in a specific folder. If the amount of free disk gets too low then all files older than one day is zipped (except zip files). If there is still not enough disk space then all zip files older than one week is deleted.`

Category Internal Name
: Internal name for grouping resource models on the IBM Tivoli Monitoring profile Category display. This name must be alphanumeric, start with an alphabetic letter, and contain no blanks. All resource models that have the same Category Internal Name will be displayed under the same category option.
For example, `storage`.

Category Descriptive Name
: This is the name that is displayed in the IBM Tivoli Monitoring profile Category display. This name will appear in the IBM Tivoli Monitoring windows and in the message catalog.
For example, `Disk storage`.

Major Version
: The major version number assigned to a resource model, which together with the Minor Version prevent installation of an earlier version than the one currently installed. Initial resource models should have a Major Version of 1.
For example, `1`.

Minor Version
: The minor version numbers assigned to a resource model, which together with the Major Version prevent installation of an earlier version than the one currently installed. Initial resource models should have a Minor Version of 0.
For example, `0`.

Cycle Time
: This is the default cycle time for the resource model. The value of the Cycle Time determines how often the decision algorithm gets called.
For example, `60`.

Scripting Language Selection
: Used to set the script type of the decision algorithm to **JavaScript**, which enables the resource model to be used on all platforms.

Select the platforms on which this resource model runs
Used to select the platforms on which the resource model should be able to run.
For example, **w32-ix86**.

3. Next, select the project to contain the resource model, as shown in Figure 6-11. Specify the file name of the resource model and click **Finish**. By default, the wizard suggests the internal name followed by `.jrm`.

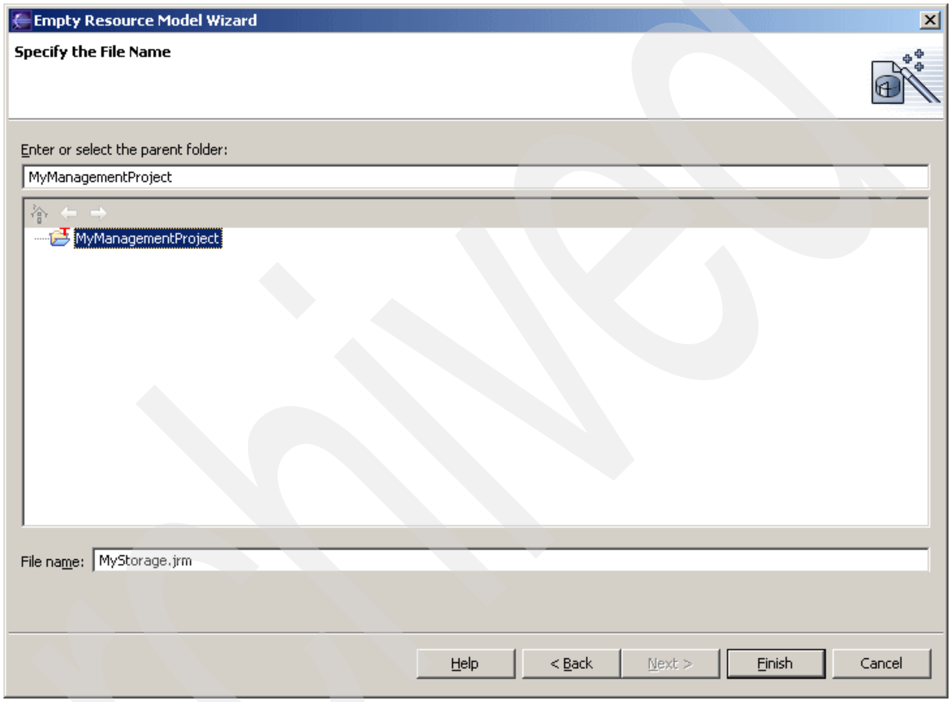


Figure 6-11 Select project for resource model

Now, your workspace should look similar to the one shown in Figure 6-12 on page 208. The steps can be repeated for as many resource models as needed.

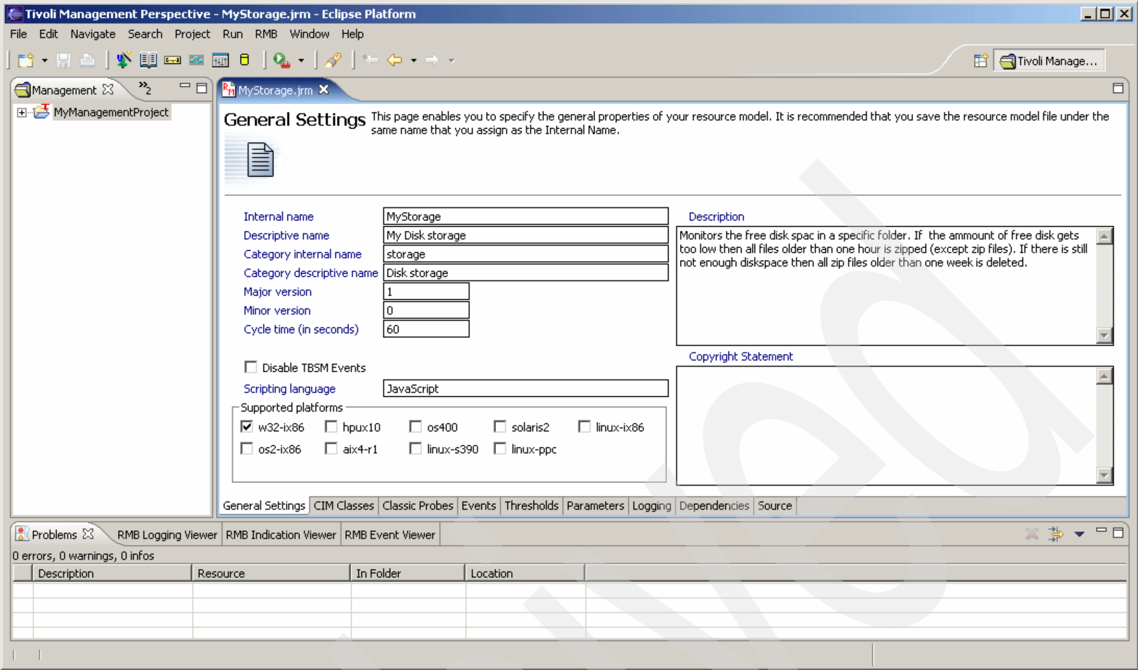


Figure 6-12 General Settings of the Resource Model Builder

The previous steps should be repeated to create a new resource model using Tivoli Resource Model Builder.

6.2.2 Common Information Model classes

The Common Information Model (CIM) classes are essential elements of the resource model. The CIM classes are used for monitoring a resource by collecting information about the resource. The CIM class describes the state of a monitored resource with a list of properties.

The CIM classes are defined by a MOF file and a provider. The provider can either be the Microsoft Windows Management Instrumentation (WMI) or Java classes implementing the Instrumentation Library Type (ILT) interface. The Autonomic Management Engine only supports Java classes implementing the ILT interface.

There are synchronous or asynchronous ways of collecting data. The implementation of such a mechanism of data collection should be implemented in the CIM classes. The synchronous way is used when the resource model can access the information directly from the managed resource, as discussed in Chapter 8, “Problem Determination scenario one” on page 259. The asynchronous way is used when the resource model can access the information either from a log file or directly from a event producer, as discussed in Chapter 9, “Problem Determination scenario two” on page 305.

Note: Because the CIM classes implement the Instrumentation Library Type (ILT) interfaces, CIM classes are also referred to as ILT classes.

The CIM classes are administered by a Common Information Model Object Manager (CIMOM). The CIMOM loads CIM classes and invokes various ILT interface methods, which, in turn, gather data from the monitored resources. The data is gathered before the decision algorithm is executed. See 6.2.9, “Source (decision tree script)” on page 221 for more information.

The following sections describe the MOF file and the ILT classes.

- Managed Object Format
- Instrumentation Library Type interface

Managed Object Format (MOF)

The MOF file is a textual definition of a CIM class. The CIM specification defines a language based on the Interface Definition Language (IDL) called Managed Object Format (MOF), which defines the elements of a CIM class.

The main components of a MOF specification are textual descriptions of:

- Classes

  Define the structure of the information held in the store. A class can be defined as collection of instances that support the same type.

- Properties

  Define the individual characteristics of each class.

- Associations

  Define the relationship between two or more classes and contains two or more references.

- References

  Define the role that each class plays in that association.

- Methods

  Define the behavior of the class.

- Instance declarations

  Define the instruction to create a new instance where the object's key values do not already exist, or an instruction to modify an existing instance where an object with identical key values already exists.

A simplified version of the MOF syntax with the M12 instrumentation is shown in Figure 6-13 on page 211. Note that the Backus-Naur Normal Form (BNF) meta-language is not complete. For the complete documentation of the MOF syntax, see the Distributed Management Task Force (DMTF) Web site at:

http://www.wbemsolutions.com/tutorials/CIM/index.html

```
MOF_file          = *comment class_declaration
comment           = "//" comment_text /n
class_declaration = qualifierList /n
                    "class" className [ ":" superClass ] /n
                    "{" *classFeatures "}" ";"
qualifierList     = "[" qualifier [ "," qualifier ] "]"
qualifier         = "Description("" descriptionText "")"
                  | "provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")"
                  | "M12_Instrumentation("Java." Java_ILT_className "|" MappingString "|"
                    Operation "")"
                  | "key"
                  | "static"
classFeatures     = propDeclaration | methodDeclaration
propDeclaration   = [ qualifierList ] dataType propertyName
                    [ array ] [ "=" defaultValue ] ";"
methodDeclaration = [ qualifierList ] dataType methodName "(" [ parameterList ] ")" ";"
parameterList     = parameter *( "," parameter )
parameter         = [ paramQualifier ] ( dataType | className "REF") parameterName
                    [ array ]
paramQualifier    = "[" ( "in" | "out" ) "]"
array             = "[" [positiveDecimalValue] "]"
datatype          = "string"
                  | "uint8"
                  | "uint16"
                  | "uint32"
                  | "uint64"
                  | "sint8"
                  | "sint16"
                  | "sint32"
                  | "sint64"
                  | "real32"
                  | "real64"
                  | "datetime"
                  | "char16"
                  | "boolean"
```

Figure 6-13 Simple MOF BNF with M12

The following keywords can be used in a MOF file:

- M12_Instrumentation

  A nonstandard CIM qualifier that applies to classes, properties, and methods. In order for a resource model to run in the Autonomic Management Engine, the MOF file should support the M12_Instrumentation qualifier. The M12 specification requires that all classes have an ENUM operation. This tells the class loader which class to associate with the enumeration functionality of the instrumentation. You can also provide a GET operation for the class to specify which class will be associated with all non-specified properties. That is, it is possible to associate specific properties within the class to specific classes on an individual basis, which allows the singular collection of the property data through the class method getProperty or getMultipleProperties.

- class

  Defines a collection of the properties or methods.

- Description

  A qualifier that provides a simple text description of the attributes present in the MOF file. You must enclose the description text in parenthesis and straight quotation marks. The description qualifier is highly recommended as a best practice for both classes and class properties, because the description information can be viewed by CIM browsing utilities after the MOF file is compiled into a localized CIM repository. This is extremely useful when multiple resource model developers will implement the same CIM class.

- provider

  A Common Object Model (COM) server that communicates with managed objects to access data and event notifications from a variety of sources such as the system registry or a custom DLL. After the information is collected, providers forward them to the CIM Object Manager for integration and interpretation. The provider used by the Autonomic Management Engine resource model is com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider. It has to be specified as:

  ```
  provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ```

- key

  The key property qualifier used to define which property is the primary key or index to the class.

- static

  The method qualifier used to define a method as static. That is, it is called without context.

The following fields are used in combination with the keywords:

- Java_ILT_className

  The Java class that implements the ILT and must be specified with the complete package (with the no.class extension). This class must be able to understand the mappingString and support the operation.

- mappingString

  A string whose meaning is known to the ILT.

- Operation

  The type of operation to be performed. The allowed operations are:

  – ENUM
    Used only for the qualifier associated with the class. It enumerates the resource instances.

  – GET
    Used for the qualifier associated with the class and property. It gets the resource attributes.

  – INVOKE
    Used for the qualifier associated with class and method. It invokes an instance or class method.

The data types correspond to Java data types, as shown in Table 6-1.

Table 6-1 MOF and Java data types

MOF data type	Java data type
string	java.lang.String
char16	java.lang.String
boolean	boolean or java.lang.Boolean
datetime	java.util.Date
uint8	
uint16	
uint32	
uint64	
sint8	byte or java.lang.Byte
sint16	short or java.lang.Short
sint32	int or java.lang.Integer
sint64	long or java.lang.Long
real32	double or java.lang.Double
real64	double or java.lang.Double

Refer to scenario one in 8.3, “Creating the Managed Object Format file” on page 264 for an example of a MOF file.

Instrumentation Library Type (ILT) interface

The Instrumentation Library Type (ILT) class implements the public methods declared in the ILT interface. The ILT is a Java interface called com.tivoli.dmunix.ep.touchpoint.base.ILTInterface, which can be found in ame-m12-mode.jar delivered with the Autonomic Management Engine (AME). For our purposes, all of the public methods are described here; be aware that some of the methods are not supported by AME and can return null values. The following sections describe the methods in the ILT interface implementation.

enumerateInstances()

This method supports the enumeration of instances of the CIM class. This function will determine how many instances should be analyzed. This can based on the number of resources monitored, or if a log file is analyzed, the number log entries. For each instance, an appropriate configuration is created. The return value is an enumeration of M12ObjectIdentity objects that identify all the instances belonging to the specified class. An example is shown in Example 8-6 on page 274, where the number of instances depends on the number of paths monitored.

getProperty()

This method is used by the service object to collect the value of a single property. It is only used if the property is qualified with a M12_Instrumentation within the MOF file. Those individual properties would be obtained through this method.

getMultipleProperties()

This method is used by the service object to collect the value of a several properties. It is only used if the class is qualified with a M12_Instrumentation with a GET operation within the MOF file. The value for all the properties will be collected with this method.

setProperty()

The setProperty() method is not supported by the Autonomic Management Engine; therefore, it should be left empty, just returning null.

invokeMethod()

There are two invoke methods, one for invoking based on the class name (for static methods) and one for invoking based on an identified CIM instance (for non-static methods). The params parameters contain parameters from the resource model: inParams contains the parameters specified with the [in] qualifier in the MOF file, and outParams contains the parameters specified with the [out] qualifier in the MOF file.

create()

This method is not supported by the Autonomic Management Engine; therefore, it should be left empty, just returning null.

destroy()

This method is not supported by the Autonomic Management Engine; therefore, it should be left empty, just returning null.

To create the JAR file for the ILT classes, refer to 8.7.1, “Creating a JAR file with the ILT class and utility class” on page 279.

The Resource Model Builder has a CIM repository with CIM classes. Custom CIM classes have to register into the CIM repository. Refer to 8.7.2, “Registering custom CIM class using Resource Model Builder” on page 282 for the sequence of steps to register CIM classes into the CIM repository.

6.2.3 Classic probes

Classic probes can be used to add Tivoli Distributed Monitoring classic probes as data sources. The classic probes are not used by the Autonomic Management Engine.

6.2.4 Events

The Events tab defines how the Autonomic Management Engine will handle events used by the resource model. An event has several options that need to be configured, and it is possible to configure corrective actions for the event. To add events, click **Add Event**.

The following event options can be customized:

- Internal name

  The internal name of an event must be unique within the entire set of resource models installed on the Autonomic Management Engine. If two or more resource models specify events with identical names, you cannot install those resource models on the same Autonomic Management Engine.

- Descriptive name

  The descriptive name of this event must be informative and recognizable for logging and reporting information.

- Message

  This is the message the event sends to the Tivoli Enterprise Console server or Tivoli Business Systems Manager. When typing the message, specify event attributes that are variables between @ symbols, for example, `This process @ProcessName@, @ProcessID@ is consuming too much CPU`.

- Send to

  Who should receive the event. Valid values are:

  – TBSM
    To send the event notification to the Tivoli Business Systems Manager server as the default.

  – TEC
    To send the event notification to the Tivoli Enterprise Console server as the default.

- Default event hierarchy

  Clear this option to create an event hierarchy in Tivoli Enterprise Console by specifying a parent for this event in the Event Parent text box. By default, this option is selected, and the event parent is set to TMW_Event.

- Event parent

  The parent of this event in the Tivoli Enterprise Console hierarchy. This enables you to organize events into hierarchies that are understood by Tivoli Enterprise Console. When the resource model is saved, the parent is saved to the workspace XML file. By default, the event parent is set to TMW_EVENT.

- Severity

  The degree of severity for notification to the Tivoli Enterprise Console server or Tivoli Business Systems Manager. The following severity levels are available:

  – HARMLESS
    Informational, nobody was impacted by the event.

  – MINOR
    Can be used in situations where one user was impacted once, but when the user retries it works. No impact on the business.

  – WARNING
    Can be used for situations where a potential problem might be developing. But no still no impact on business.

- CRITICAL
  Can be used in situations where part of the application is not working correctly and corrective actions have to be performed immediately. Impact on business.
- FATAL
  Can be used in situations where nothing works in the application and there is a high impact on business.

- Description

  The descriptive text that is displayed on the IBM Tivoli Monitoring windows and that provides information for the operator running the resource model.

- Properties

  Add or remove properties from the event. Properties detail an event by enabling you to obtain as much information as you need from the event. For example, an event might indicate that your disk subsystem has problems. By associating properties such as bytes per second, disk name, or available disk space, you can identify the problem more easily.

  To add an property:

  a. Click **Add**, or right-click inside the table.

  b. Enter the name of the property in the Name column.

  > **Note:** If you are going to enable logging for this resource model, use the same names for properties here and in the logging definitions when specifying the same metric.

  c. Select the type as **Numeric** or **String** column.

  d. Select if it is a key or not. Key properties are the most meaningful and play a fundamental role in the event aggregation process.

  e. Click **OK**.

  f. Select the **Unit** column for the newly created property and select the applicable property unit.

  > **Note:** The same property units that can be assigned to logged numeric properties, which are Other by default.

- Aggregation settings

  The conditions under which this indication becomes an aggregated indication event. These conditions are:

  – Clearing Event

    Select Clearing Event if you want the system to send a clearing event when the circumstances that generated the event are resolved. The Tivoli Enterprise Console server and Tivoli Business Systems Manager use the clearing event to close the corresponding error event. For example, if the resource being monitored is a service, the event is sent when the service is not available. The event might have an action associated with it to restart the service (or the restart might be done manually). When a cycle is run without generating the indication, it is expected that the situation has been solved and a clearing event is send.

  – Number of occurrences

    The number of indications that must occur before an aggregated indication event is generated. The number of occurrences must be greater than zero. One is the default value for this resource model. It can be changed at run time in the IBM Tivoli Monitoring application.

  – Number of holes

    The maximum number of monitoring cycles allowed with no indication events. The number of holes must be greater than or equal to zero. Zero is the default value for this resource model. It can be changed at run time in the IBM Tivoli Monitoring application.

Actions

Corrective actions can be added to each event by clicking the **Actions** button. Here, you can add CIM methods, programs, and shell commands.

Add CIM method

To add a CIM method, click **Add CIM Method**, and then select the CIM class and method. For each key and parameter, select an Event attribute from the drop-down list. In addition, a Descriptive name and a Description must be added. When done, click **OK**.

> **Important:** Although it is possible to select a CIM method without specifying the Descriptive name and Description, it is required. It is *not* possible to generate the Autonomic Management Engine package without the Descriptive name and Description.

6.2.5 Thresholds

Thresholds are used to specify limitations on resources. The thresholds should be used in the decision algorithm when analyzing the collected information. To add thresholds, click the **Add Treshold** button.

Thresholds are specified with the following options:

- Internal name

  Name used in the decision algorithm to determine if a property has exceeded the treshold.

- Descriptive name

  Descriptive name that is used in the IBM Tivoli Monitoring tool.

- Default value

  The description of this threshold as it displays on the IBM Tivoli Monitoring tool.

- Description

  A description of what the threshold is about and how it is used.

6.2.6 Parameters

Parameters are used for values that can be configured when the resource model is deployed. The parameter values can vary between resource model instances. The parameters can have multiple values. To add a parameter, click the **Add Parameter** button.

Parameters are specified with the following options:

- Internal name

  Name used in the decision algorithm to access the parameter values.

- Descriptive name

  Descriptive name that is used in IBM Tivoli Monitoring tool.

- Description

  A description of what the parameter is about and how it is used.

- List type

  Specifies the type for the list. It can be one of the following:

  – Unrestricted
    Unrestricted lists offer the option to restrict the types of values accepted to only strings or only numbers. String lists can be used to specify parameter values such as application names. Numeric lists can be useful to define event IDs or port numbers.

  – Restricted String
    Restricted lists enable you to specify whether an administrator will be able to select one or more values from the parameter list in IBM Tivoli Monitoring desktop and enable or disable a description field for each parameter in the list. Depending on the setting, one or more of the parameters can be selected by default.

6.2.7 Logging

The Logging tab is used to specify which information about a resource model to store as a history. The information is stored in the AME database. Logging is disabled by default and will not be further described. For more information about logging, see *IBM Tivoli Monitoring Resource Model Builder User's Guide*, SC32-1391.

6.2.8 Dependencies

Dependencies are used to specify files needed to execute the resource model. This includes JAR files with ILT classes, native libraries, and DLLs if needed.

To add a file, click the **Add** button. In the window that opens, specify the file and the supported platforms.

Refer to 8.7.2, "Registering custom CIM class using Resource Model Builder" on page 282 for details.

Important: Remember to include the MOF files for the data sources.

Note: When you add some files to the dependency list, they are imported; therefore, there are no links to the original file. When you generate the Autonomic Management Engine package, the files will be included as they were when they were added to the dependency list.

6.2.9 Source (decision tree script)

The Resource Model Builder automatically generates a script called the decision tree script, which implements the decision algorithm. The language for the decision tree script is JavaScript.The decision tree script consists of several variables and the three functions;

- SetDefaultConfiguration(Svc)
- Init(Svc)
- VisitTree(Svc)

The Autonomic Management Engine has routines to invoke these methods, which we discuss here. The decision tree script will be edited on the Source tab, as shown in Figure 6-12 on page 208.

> **Important:** It is important that all three methods in the decision tree script return 0 on success; otherwise, the execution of the resource model will stop. When creating an empty resource model, the methods Init() and VisitTree() do not return anything. Remember to add the statement return(0); at the end of each method.

The calling sequence of the tree methods is shown in Figure 6-14 on page 222, where the three JavaScript functions are shown with a gray box. When a resource model instance is started, the initialization is performed in three steps. First, the SetDefaultConfiguration() method is executed, then the configuration is overridden with the configuration of the instance, and finally the Init() method is executed to initialize variables and Java classes based on the configuration. The cycle part of the decision script contains the following five activities (see Figure 6-14 on page 222):

1. *Collect data* is where information about the monitored resource is collected. This is done by collecting values of the properties on the CIM classes. How often the values are collected depends on the configuration of the CIM class.
2. The *VisitTree()* method is called to analyze the collected data and send indication events if needed.
3. The indication events are aggregated into aggregated events depending on the configuration of the events. Potentially, a clearing event is send if a situation has been solved.
4. If aggregated events are issued, the corrective actions are invoked for the events.
5. The last thing is to wait for the next cycle.

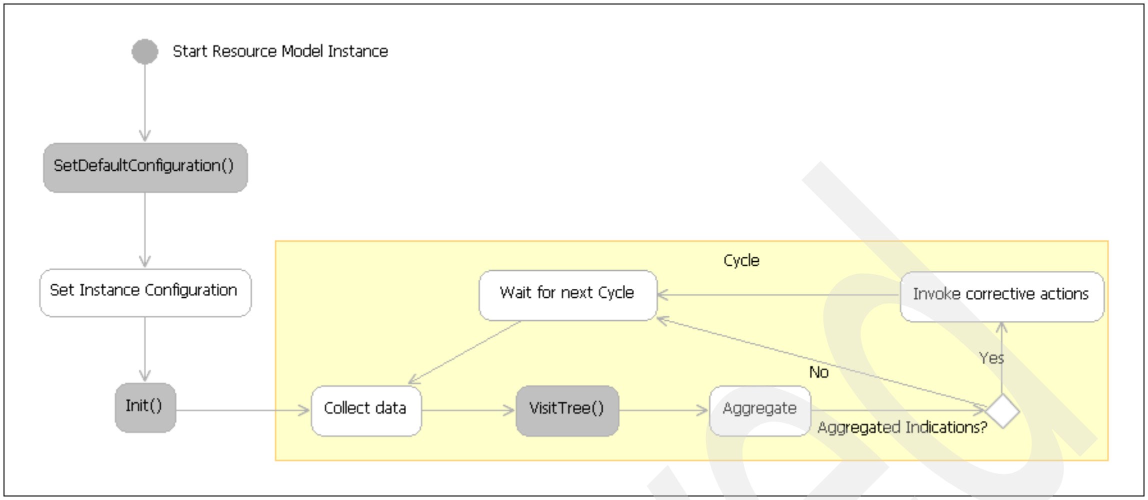


Figure 6-14 Decision tree activity diagram

Refer to 8.7.2, “Registering custom CIM class using Resource Model Builder” on page 282 for details.

6.3 Simple Agent Reference Application (SARA)

The current version of the Autonomic Management Engine tool is shipped with a sample embedding application called SARA. This tool can be used as a replacement for writing your own embedding application, as explained later. This tool is intended only for demonstration purposes.

The tool comes with the following default plug-ins:

- Serviceability: JLog-based implementation plug-ins for logging trace information
- CIM method invocation action launcher: Plug-in for invoking CIM methods
- Shell command action launcher: Plug-in for invoking shell scripts or programs when run through ShellCmd
- EIF event sink: An event sink plug-in that sends EIF formatted events to Tivoli Enterprise Console or other such servers

The SARA configuration files are located in the <%AME_HOME%>\sara\config folder, where <%AME_HOME%> is the installation directory of the Autonomic Management Engine tool.

SARA provides a command-line interface for various actions that can be programmatically executed by any typical embedding application. The command-line interface is started by executing the **sara.bat** or **sara.sh** file located in %AME_HOME%\sara.

Figure 6-15 shows the command-line interface of the SARA application. As seen in the figure, various operations performed by the embedding application can be executed as commands so that various instances of the resource model types can be created and started using the command line.

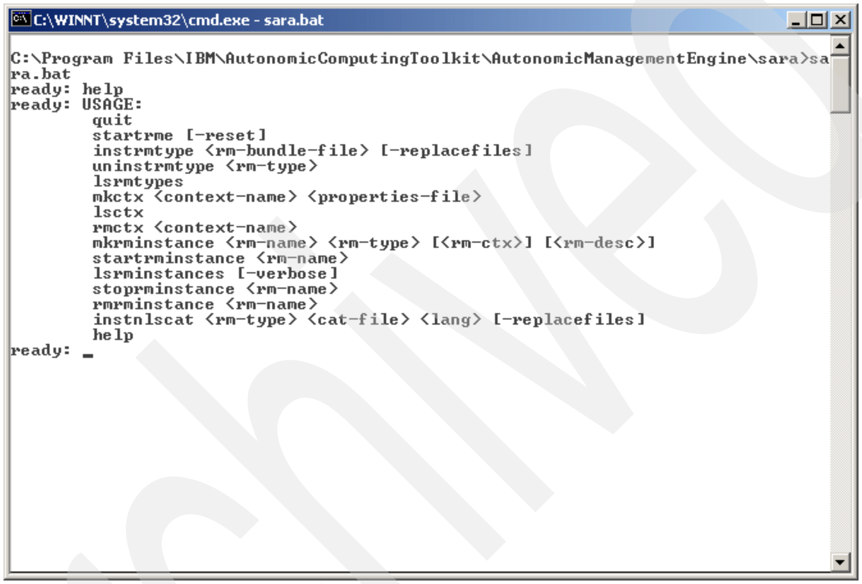


Figure 6-15 The SARA application shipped with Autonomic Management Engine tool

The following list provides a short description of each of the commands:

- **startrme [-reset]**

  This is the first command that must be run. It initializes SARA and the underlying Autonomic Management Engine services. If you do not run the **startrme** command first, you will see an error message. If you specify the **–reset** option, AME resets its previous configuration; all resource model instances are deleted, all contexts are removed, and all resource model types are uninstalled. This option is useful only for test purposes.

- **instrmtype <rm-bundle-file> [-replacefiles]**

  Installs a resource model type based on the resource model package. The **instrmtype** command installs the contents of the resource model package in a well-known directory structure defined by the embedding application (that is, ./work/rmbundles for SARA). The **–replacefiles** option allows the replacement of some files installed with other resource models.

- **uninstrmtype <rm-type>**

  Uninstalls a resource model type. All instances of the type have to be removed first or you will receive an error.

- **lsrmtypes**

  Lists all installed resource model types.

- **mkctx <context-name> <properties-file>**

  Creates a context with a set of properties (key-value pairs) based on the content of a property file.

- **lsctx**

  Lists the available contexts.

- **rmctx <context-name>**

  Removes a context.

- **mkrminstance <rm-name> <rm-type> [<rm-ctx>] [<rm-desc>]**

  Creates an instance of a resource model. An instance of a resource model is created from its resource model type associating (optionally) a context and a descriptor.

- **lsrminstances [-verbose]**

  Lists resource model instances. If you use the **–verbose** option, you also get status information.

- **startrminstance <rm-name>**

  Starts a resource model instance.

- **stoprminstance <rm-name>**

  Stops a resource model instance.

- **rmrminstance <rm-name>**

  Removes a resource model instance. The resource model instance is removed regardless of its current state (running or stopped).

- **instnlscat <rm-type> <cat-file> <lang> [-replacefiles]**

  Installs an additional national language support (NLS) catalog file (Tivoli message catalogs) for a resource model type.

- **`quit`**

  Exits SARA.

6.4 How to debug a resource model

Depending on what needs to be debugged, several approaches can be used. The following sections provide an example of how to debug the Resource Model Builder using the Simple Agent Reference Application (SARA). With the Resource Model Builder, you can check the JavaScript and invoke a JavaScript debugger, but it is not possible to test the ILT classes. With SARA, it is also possible to use the JavaScript debugger with the ILT classes implementation.

6.4.1 Debugging JavaScript

The Resource Model Builder can be used together with Microsoft Script Debugger to debug the JavaScript. Resource models with custom ILT classes cannot be debugged with the Resource Model Builder, because the Resource Model Builder cannot be configured to find the ILT classes; therefore, only the JavaScript not relying on data from ILT classes can be debugged with the Resource Model Builder.

To debug the JavaScript, perform the following steps:

1. The trace is configured from the Preferences window. From the menu, select **Window** → **Preferences**. In the Preferences window, select **Tivoli Management** → **Resource Model Builder**, as shown in Figure 6-16 on page 226.

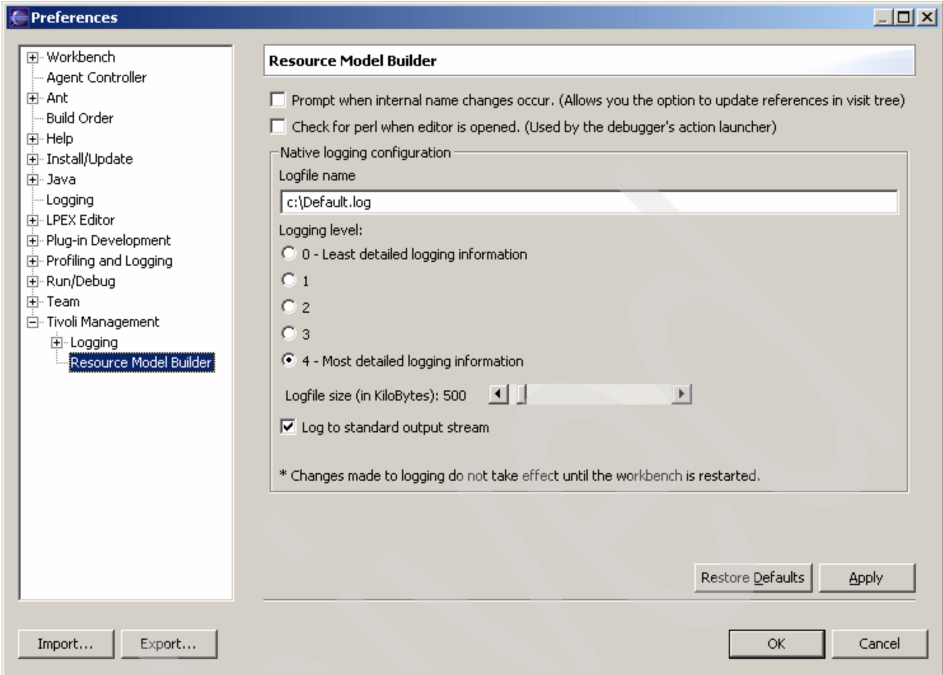


Figure 6-16 Resource Model Builder Preferences

2. Select logging level **4 - Most detailed logging information** and **Log to standard output stream**. This will show the most information in the JavaScript console.
3. Now, restart the Resource Model Builder for the settings to take effect.
4. The JavaScript can be debugged with debugging tools, such as Microsoft Script Debugger. The Microsoft Script Debugger can be downloaded from Microsoft for free from the following Web site:

   http://microsoft.com/downloads/details.aspx?FamilyId=2F465BE0-94FD-4569-B3C4-DFFDF19CCD99&displaylang=en
5. To execute the JavaScript, go to the Source tab of the JRM file and click the **Run Script** icon. If a debugger is installed, it will open; otherwise, the JavaScript will run as it would be run in an Autonomic Management Engine.

Note: During the writing of this book, running the JavaScript with the Run Script icon has only been found useful for a syntax check of the JavaScript.

6.4.2 Debugging CIM classes with SARA

Here, we show how to run SARA inside the Resource Model Builder as a Java application in its own project. The following steps describe how to configure a project for running SARA:

1. Open the Java perspective.
1. Create a Java project for SARA, for example, `SARA runtime`.
2. Copy the config directory from AME_HOME\sara\config to the SARA Runtime project.
3. From the menu, select **Run** → **Debug**. A window opens similar to the one shown in Figure 6-17.

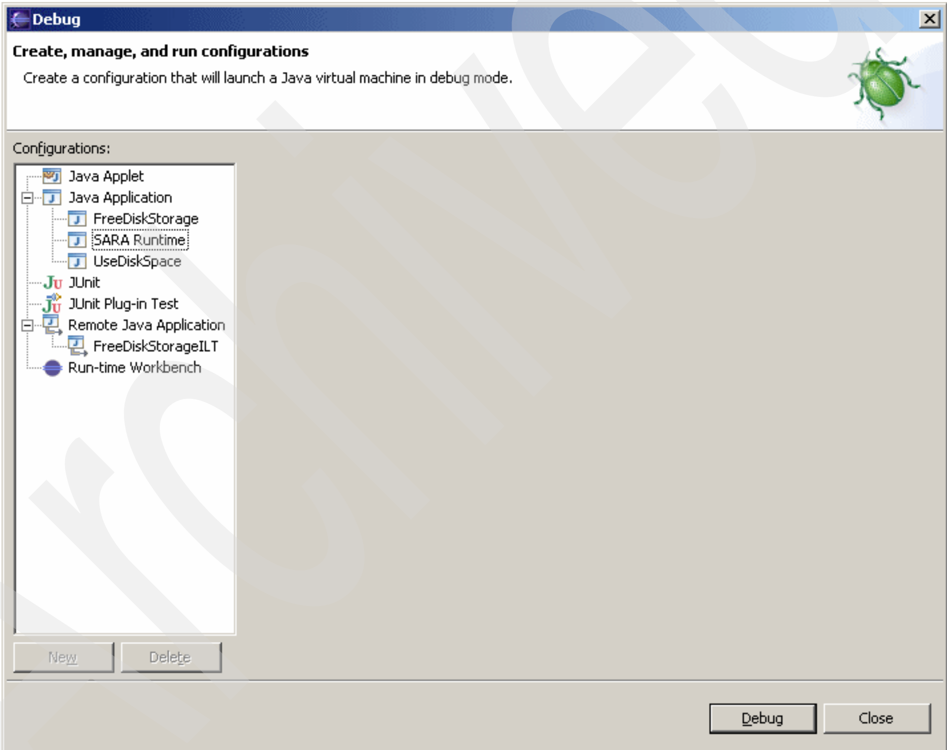


Figure 6-17 Debug window

4. Create a new Java application by selecting **Java Application** and clicking **New**. Configure the Java Application as follows:
   – Main tab

     Name `SARA Runtime`

Main class	`com.ibm.amw.sara.cli.RMEShell`

– Arguments tab

VM aguments	`-Ddb2j.stream.error.file="${workspace_loc:SARA Runtime}\\logs\\db2j.log" -Ddb2j.stream.error.logSeverityLevel=3000 -Ddb2j.infolog.append=true -DARCH=w32-ix86 -Djava.endorsed.dirs="C:\PROGRA~1\IBM\AUTONO~1\AUTONO~1\\lib\\endorsed"`
Working directory	`${workspace_loc:SARA Runtime}`

– Classpath tab

directory	`C:\Program Files\IBM\AutonomicComputingToolkit\AutonomicManagementEngine\lib` `SARA Runtime\config`
JAR files	All JAR files in the directory C:\Program Files\IBM\AutonomicComputingToolkit\AutonomicManagementEngine\lib and C:\Program Files\IBM\AutonomicComputingToolkit\AutonomicManagementEngine\lib\endorsed

– Source tab

The CIM projects source folder `DiskSpaceCIM\src`

– Environment tab

PATH	`${workspace_loc:SARA Runtime}\work\rmbundles\customscripts;${workspace_loc:SARA Runtime}\work\rmbundles\lib;${workspace_loc:SARA Runtime}\config`

5. Click **Debug** to start SARA with debug for the first time.
6. In the console, enter the **`startrme`** command and press Enter. This will create the directory logs and work under the SARA Runtime project.
7. To install the resource model, use the **`instrmtype`** command, for example:

```
instrmtype "<fullPath>\MyStorage.zip" -replacefiles
```

Where `<fullPath>` is the full path to the MyStorage.zip package. This package is presented in Chapter 8, “Problem Determination scenario one” on page 259.

8. To create an instance of the resource model, use the **mkrminstance** command. For example, to create an instance named MS1, use:

```
mkrminstance MS1 MyStorage
```

9. To start the instance, use the **startrminstance** command, for example:

```
startrminstance MS1
```

The file SARA Runtime\logs\trace.log will contain all the trace information, both from the JavaScript and the ILT class.

Now, if a breakpoint is placed in the ILT class, the Resource Model Builder will switch to the Debug perspective where it will be possible to step through the code.

For a description of the commands that can be used in the SARA command-line interface, see 6.3, “Simple Agent Reference Application (SARA)” on page 222.

7

Advanced Autonomic Management Engine topics

This chapter provides information about advanced topics using the Autonomic Management Engine (AME).

We discuss the following topics in this chapter:

- Embedding the Autonomic Management Engine into applications

  This section provides details about how to embed an Autonomic Management Engine into an application.

- Structure of resource model

  This section describes the structure of a resource model package.

7.1 Embedding the Autonomic Management Engine into applications

The Autonomic Management Engine (AME) core exposes various APIs and interfaces for applications. The embedding application is responsible for implementing the required interfaces and plug-ins to enable the AME core to perform the required operations.

Figure 7-1 gives an overview of the embedding application and the associated components.

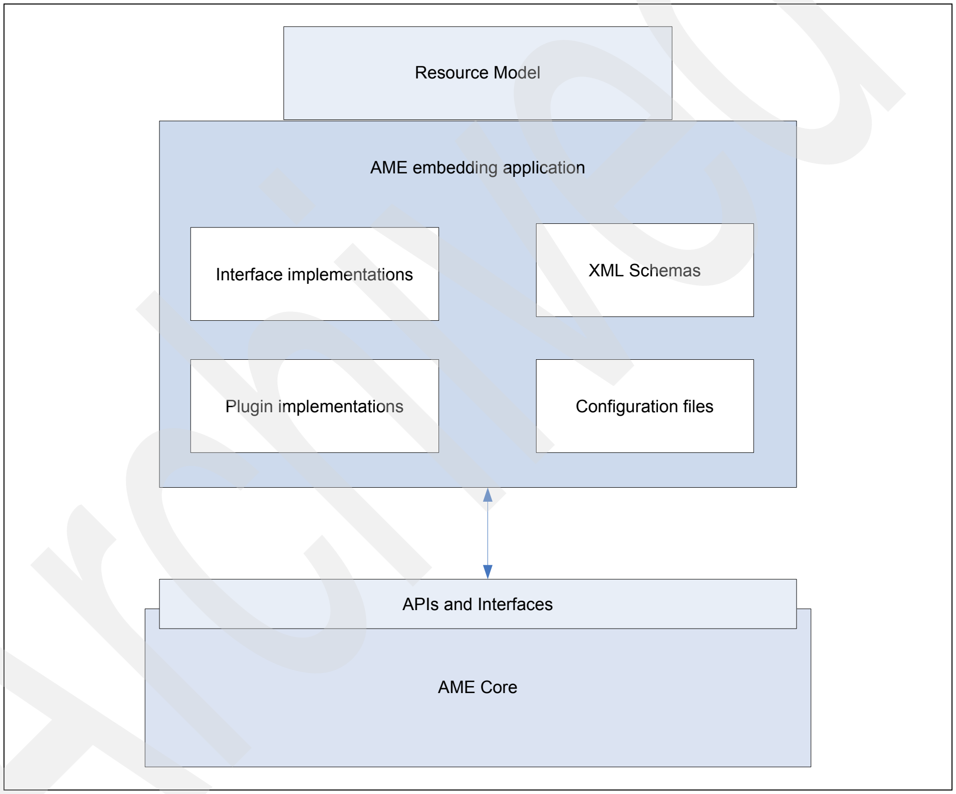


Figure 7-1 Embedding application and related components

The AME documentation can be referred to for more information about the APIs, interfaces, and internal working modules.

7.1.1 Designing the embedding application

The embedding application is responsible for providing the implementation for interfaces and invoking the required API calls to load the resource model instances. The AME core components consume various parameters given by the embedding application. An embedding application has the responsibility to provide the required events and action manager plug-ins along with the configuration files.

The the tasks of an embedding application include:

- Implementing the RMIdentifier and RMIdentifierPattern interfaces.
- Providing the required plug-ins for action launchers and event sinks.
- Configuring the rme.config file for these plug-ins.
- Making API calls to load resource model instances to the AME core.

Interfaces

There are two interfaces, RMIdentifier and RMIdentifierPattern, that an embedding application should implement in order to use the API calls provided by the AME core.

RMIdentifier

This interface must be implemented by the embedding AME application in order to identify the resource model instances with unique names instead of just the label. The RMIdentifier implementation class can be used to store various attributes of a resource model, which can be helpful in identifying various instances of resource models in a unique fashion.

This public interface describes two methods:

- equals(RMIdentifier rmID): This must indicate a mechanism to compare two resource model identifiers. The Autonomic Management Engine uses the equals() method to distinguish different resource model instances.
- getKey(): This method should return a unique string encoding of a resource model identifier.

In the example embedding application, the name of the RMIdentifier interface is returned as a unique identifier. There can be other cases where any other parameter of the resource model can be returned in combination to uniquely identify a resource model instance.

Example 7-1 on page 234 provides the implementation of the RMIdentifier implementation class called RMIdentifierImpl.java.

Example 7-1 RMIdentifierImpl.java implementation

```
package com.ibm.itso.sg246665.embeddedame;

import java.io.Serializable;

import com.ibm.amw.rme.RMIdentifier; // from ame-core.jar

/**
 * Implements the RMIdentifier for the embedded AME.
 */
public class RMIdentifierImpl implements RMIdentifier, Serializable {

  // The name of a resource model instance
  private String name;

  // The type of a resource model
  private String type;

  // The context a resource model
  private String context;

  /**
   *
   * @param name
   * @param type
   * @param context
   */
  public RMIdentifierImpl(String name, String type, String context) {
    this.name = name;
    this.type = type;
    this.context = context;
  }

  /**
   * Two resource models are equal if their names are same. The getKey()
   * function is used to get the unique identifier for the RM instances.
   */
  public boolean equals(RMIdentifier rmId) {
    if (this.getKey().equals(rmId.getKey())) {
      return true;
    }
    return false;
  }

  /**
   * Return a uniqueu identifier.
   */
  public String getKey() {
```

```
    return name;
  }

  /**
   * @return Returns the context.
   */
  private String getContext() {
    return (context == null) ? "" : context;
  }

  /**
   * @return Returns the name.
   */
  private String getName() {
    return name;
  }

  /**
   * @return Returns the type.
   */
  private String getType() {
    return type;
  }

  /**
   * Return a String representation of the RM
   *
   */
  public String toString() {
    return "rm-name = " + name + "- rm-type = " + type + "- rm-context = "
        + context;
  }
}
```

RMIdentifierPattern

Another public interface that the embedding application must implement is RMIdentifierPattern. This interface is used to filter various resource model identifiers inside the AME core. The AME core method, Engine.getRMIdentifiers(RMIdentifierPattern), uses this implemented class to filter different resource model identifiers.

The method isMatched(RMIdentifier) in this public interface should implement logic to filter based on various types of filtering methods.

Example 7-2 on page 236 shows a RMIdentifierPatternImpl class that implements the RMIdentifierPattern public interface, which uses regular expressions for identifying the resource models. For the regular expression, the

Jakarta Regexp Version 1.3 is used. You can obtain it from the following Web site under Apache License:

http://jakarta.apache.org/regexp/index.html

For licensing information, refer to:

http://jakarta.apache.org/commons/license.html

Example 7-2 Example of class implementing the RMIdentifierPattern interface

```
package com.ibm.itso.sg246665.embeddedame;

import org.apache.regexp.RE;

import com.ibm.amw.rme.RMIdentifier;
import com.ibm.amw.rme.RMIdentifierPattern;

/**
 * Implementation for RMIdentifierPattern public interface with regular
 * expressions. This applies to only complex environments and AME with
 * multiple Resource Models.
 *
 * @author ITSO Redbook Team
 */

public class RMIdentifierPatternImpl implements RMIdentifierPattern {
  /**
   * Pattern for name
   */
  private RE rmNamePattern;

  /**
   * Pattern for type
   */
  private RE rmTypePattern;

  /**
   * Pattern for context
   */
  private RE rmContextPattern;

  /**
   * Constructor for different patterns.
   *
   */

  public RMIdentifierPatternImpl(String rmNamePattern,
      String rmTypePattern,
      String rmContextPattern) {
```

```
    this.rmNamePattern = createrePattern(rmNamePattern);
    this.rmTypePattern = createrePattern(rmTypePattern);
    this.rmContextPattern = createrePattern(rmContextPattern);
  }

  /**
   * Creates the pattern as a regular expression. If the pattern is empty or
   * only contains '*' then it is relaced with ".*"
   *
   * @param pattern
   *          String with a regular expression.
   * @return A regular expression matching the pattern.
   */
  private RE createrePattern(String pattern) {
    String adjPattern = pattern;
    if (adjPattern == null || adjPattern.equals("")
        || adjPattern.equals("*")) {
      adjPattern = ".*";
    }
    return new RE(adjPattern);
  }

  /**
   * This interface method must indicate whether the specified RM identifier
   * matches the pattern or not. This is used only for filtering the RM
   * identifiers representing RM instances.
   */
  public boolean isMatched(RMIdentifier rmId_) {
    RMIdentifierImpl rmId = (RMIdentifierImpl) rmId_;

    return (rmNamePattern.match(rmId.getName())
        && rmTypePattern.match(rmId.getType()) && rmContextPattern
        .match(rmId.getContext()));
  }
}
```

Embedding application class

The APIs provided by the Autonomic Management Engine can be obtained in the Autonomic Management Engine documentation provided with the IBM Autonomic Computing Toolkit. The embedding application should use these API calls to set up the hosting environment for the Autonomic Management Engine.

An application embedding the Autonomic Management Engine has the following responsibilities:

- Configure the Autonomic Management Engine and subcomponents. This includes the embedded database, logging (for trace), auction launchers, and event sinks.
- Start the resource model engine, which is the core of the Autonomic Management Engine.
- Install the resource model types from the resource model ZIP files.
- Install context files (resource model configuration) if applicable.
- Create instances of the resource models. Create multiple instances if applicable.
- Start the resource model instance or instances.

The AME APIs are called appropriately to perform the previous sequence of functions for the embedding application. Along with this set of required functions, various other functions can be used to:

- Uninstall resource model types.
- List the resource model types.
- List contexts.
- Remove contexts.
- List the resource model instances running in the Autonomic Management Engine.
- Stop a resource model instance.
- Install an NLS catalog for a resource model type.

A detailed implementation, along with documentation, is provided in the ScenarioApplicationEmbeddedAme.java sample (refer to Appendix A, "Additional material" on page 379). We describe a few important pieces of the embedding application in the following discussion. This AME embedding application is a basic application that loads the resource model name given as a parameter to the class and creates a single instance of the resource model called RMINSTANCE. This class can further be used and configured according to your IT solution needs.

The embedding application class ScenarioApplicationEmbeddedAme.java defines variables for a few underlying AME classes; RMPackageManager, RMManager, ContextManager, HistoricalDataReader, and RMBundleManager are declared.

When the embedding application starts the AME, these values are obtained as shown in Example 7-3. Refer to the MyAme.start(boolean) method for further documentation.

Example 7-3 Initialize and start the resource model management engine

```
// Obtain references to the core AME components using the rme factory
    rmeFactory = EngineFactory.getInstance(appName, mode);
    // Initialize the RMBundleManager anchor dir and the interp is set here ..
    rmBundleManager = RMBundleManager.getInstance(rmeFactory);
    // Obtain a reference to the package manager
    rmPackageManager = rmeFactory.getRMPackageManager();
    // Obtain a reference to the Resource Model Manager
    rmManager = rmeFactory.getRMManager();
    // Obtain a reference to the Context manager
    contextManager = rmeFactory.getContextManager();
```

The next step is to install the resource model type by creating a ZipInputStream for the resource model type package given as a parameter to the application, as shown in Example 7-4.

Example 7-4 Calling the installRMType function

```
  /**
   * Installs a Resource Model package specified by a zip file name.
   *
   * @param zipfilename
   *          Resource Model package
   * @return The name of the Resource Model type from the zip file
   * @throws InvalidRMDescriptorException
   * @throws SetPermissionsOnFileException
   * @throws RMTypeAlreadyPresentException
   * @throws FileNotFoundException
   * @throws IOException
   * @throws InvalidRMBundleException
   */
  public String installRMzip(String zipfilename)
      throws InvalidRMDescriptorException,
      SetPermissionsOnFileException,
      RMTypeAlreadyPresentException,
      FileNotFoundException,
      IOException,
      InvalidRMBundleException {
    AmeLog.tr.entry(Level.DEBUG_MAX, this, "installRMzip", zipfilename);
    String installedRMType = null;
    ZipInputStream zipInputStream = new ZipInputStream(new FileInputStream(
        zipfilename));
    installedRMType = rmBundleManager.installRMType(zipInputStream, true);
    AmeLog.tr.exit(Level.DEBUG_MAX, this, "installRMzip", installedRMType);
```

```
    return installedRMType;
  }
```

The ZipInputStream in the previous example refers to the ZipInputStream of the resource model ZIP file given as a parameter. The Boolean value refers to whether or not the existing files need to be replaced.

The next step is to create an instance of the resource model. This is accomplished by the createRMInstance() method, which creates an instance of the RMIdentifierImpl (implementing RMIdentifier interface) and loads it into the autonomic resource model manager. Example 7-5 provides a description of this function with comments.

Example 7-5 Resource model instance in ScenarioApplicationEmbeddedAme.java

```
/**
   * This creates the Resource Model instance. There can be multiple instances
   * of the same resource model in the Autonomic Management Engine which can
   * monitor different types of resources.
   *
   * @param rmName
   *          Name of the instance. It must be unique in the Engine and cannot
   *          be null.
   * @param rmType
   *          Type of the Resource Model.
   * @param rmCtx
   *          Context that is associated with the instance. This can be
   *          created by the createContext. It can be null if there are no
   *          contexts.
   * @param rmDescriptor
   *          The settings of the Resource model instance. The default
   *          resource model descriptor settings file will be used if this
   *          parameter is null.
   */
  public void createRMInstance(
      String rmName,
      String rmType,
      String rmCtx,
      String rmDescriptor)
      throws RMAlreadyPresentException,
      RMOperationFailureException,
      ContextNotFoundException,
      RMNotFoundException,
      InvalidRMDescriptorException {

    // Instantiate the RMIdentifier class
    RMIdentifierImpl rmid = new RMIdentifierImpl(rmName, rmType, rmCtx);
```

```
 // Create the Resource Model Instance
 rmManager.create(rmid, rmCtx);

 // If this variable is false,after the load function below, then delete
 // the instance.
 boolean done = false;

 try {
  rmManager.load(rmid, rmDescriptor);
  // Catch the various exceptions, delete the rm, log the error, and
  // rethrow the exception
 } catch (DecInitFailureException e) {
  rmManager.delete(rmid);
  AmeLog.tr.exception(Level.FATAL, this, "createRMInstance", e);
  throw e;
 } catch (RMNotFoundException e) {
  rmManager.delete(rmid);
  AmeLog.tr.exception(Level.FATAL, this, "createRMInstance", e);
  throw e;
 } catch (InvalidRMDescriptorException e) {
  rmManager.delete(rmid);
  AmeLog.tr.exception(Level.FATAL, this, "createRMInstance", e);
  throw e;
 } catch (RMOperationFailureException e) {
  rmManager.delete(rmid);
  AmeLog.tr.exception(Level.FATAL, this, "createRMInstance", e);
  throw e;
 } catch (IllegalStateException e) {
  rmManager.delete(rmid);
  AmeLog.tr.exception(Level.FATAL, this, "createRMInstance", e);
  throw e;
 }
}
```

The next step is to start the resource model instance called RMINSTANCE. This is accomplished by the startRMInstance() method, as shown in Example 7-6.

Example 7-6 Starting the resource model instance

```
/**
 * Start the resource model with a specified name.
 * @param rmName
 * @throws RMNotFoundException
 * @throws RMOperationFailureException
 */
public void startRMInstance(String rmName)
  throws RMNotFoundException,
  RMOperationFailureException {
 RMIdentifierPatternImpl thisrm = new RMIdentifierPatternImpl(rmName,
```

```
        "*", "*");
    RMIdentifier[] rmids = rmManager.getIdentifiers(thisrm);

    if (rmids != null && rmids.length > 0) {
      // Start each of the found instances
      for (int i = 0; i < rmids.length; i++) {
        rmManager.start(rmids[0]);
      }
    } else {
      throw new RMNotFoundException(new RMIdentifierImpl(rmName, "-", "-"));
    }
  }
```

The main method is implemented to optionally install and start the resource model. When that is done, it will go into a loop implementing a command-line interface. From this command-line interface, you can list the installed resource model types and instances, show some detailed information about the instances, stop instances, start instances, and exit the AME by entering **quit**.

7.1.2 Building and launching the embedding application

We now have a resource model package, the interfaces required by the AME core, and the AME embedding application.

Building the ITSO_RB_AME.jar file

The interface files and the ITSO_RB_AME.java file are compiled after setting the required CLASSPATH and PATH environment variables. The build.xml file provided with the project files (refer to Appendix A, “Additional material” on page 379) shows the elements involved in creating the ITSO_RB_AME.jar file, consisting of the embedding application and the interfaces required by the AME core. See Example 7-7.

Example 7-7 ITSO build.xml

```
<?xml version="1.0"?>
<!-- ======================================================================
     Feb 16, 2005 9:38:24 AM

     Scenario application
     Pack the class files

     Author: IBM Redbook Team
     ====================================================================== -->
<project name="ITSO_RB_AME" default="default">

   <property file="build.properties"></property>
```

```
    <description>
            Pack the ITSO Redbook AME  class files.
     </description>

    <!-- ================================
           target: default
          ================================ -->
    <target name="default" depends="buildJarFile,buildZipFile"
description="Pack the ITSO Redbook AME class files">

    </target>

    <!-- - - - - - - - - - - - - - - - - -
           target: buildJarFile
          - - - - - - - - - - - - - - - - - -->
    <target name="buildJarFile">
       <jar destfile="${jarFile}" basedir="bin"
manifest="META-INF\MANIFEST.MF">

       </jar>
       <jar destfile="${jarFile}" basedir="src" update="true" />

    </target>
    <!-- - - - - - - - - - - - - - - - - -
           target: buildZipFile
          - - - - - - - - - - - - - - - - - -->
    <target name="buildZipFile">
       <zip destfile="${zipFile}">
          <fileset dir=".">
             <include name="config/*.*" />
             <include name="lib/*.*" />
             <include name="*.txt" />
             <include name="*.bat" />
             <include name="DatabaseConnectionRM.zip" />
          </fileset>
       </zip>
    </target>
</project>
```

Setting up the configuration files

To run the embedding application, the AME core refers to configuration files and properties files for loading the required plug-ins and classes. In this example, we use the default configuration as given in the AME installation. However, it is worth mentioning the importance of the configuration files and how you can use these configuration files for advanced AME setup.

rme.config

This file consists of important parameters for the resource model engine. The following parameters are referenced in this configuration file:

- dir.* parameters: These properties refer to the directory structure for the storage of various components of the resource model package.
- serviceability: This refers to the fully qualified logging and tracing class.
- eventsinks.*: This refers to parameters required for an EventSink plug-in for the AME core. We have, as of now, used the default plug-ins, which are part of the Autonomic Management Engine installation package. Refer to the "Advanced Topics" chapter of the *Autonomic Management Engine Developer's Guide* for implementation details about event sinks.
- actionslaunchers.*: These parameters refer to the ActionLauncher plug-in for the AME core. We use the defaults as of now, and advanced topics can be found in the AME documentation.
- modes.CimM12: This should refer to the Common Information Model (CIM) implementing class. The default value is to use the Tivoli M12 DMTF CIM mode class.
- datalog.*: This refers to the database properties that will be used by AME core for storing internal data for logging.
- shell: This property is used by the default AME action launcher plug-in for executing shell commands.

m12_mode.properties

This properties file should mention parameters for the CIM Object Manager running in the AME core.

jlog.properties

This file is used by the AME core for logging purposes. This might not need editing, but is a required property file for logging data for debugging purposes.

> **Note:** The default configuration files can be found in the <AME_HOME>\sara\config folder. These configuration files are part of the application Simple Agent Reference Application (SARA) shipped with the AME tool. In 6.3, "Simple Agent Reference Application (SARA)" on page 222, we briefly discuss this application.

Running the AME application

The embedding AME application should be invoked with the resource model type details and should set all the required classes in the CLASSPATH.

The script provided in Example 7-8 describes the CLASSPATH settings required by the embedding application.

Example 7-8 The startITSO_RB_AME.bat file

```
@echo off
REM # Start the scenario application

REM ############### Environment configuration
REM # Adjust the following env variables to match your system
set SCENARIO_HOME=.
set AME_HOME=C:\Program
Files\IBM\AutonomicComputingToolkit\AutonomicManagementEngine
set JRE_HOME=C:\Program Files\IBM\Java142\jre

REM ############### Resource Models to be installed
REM SET RESOURCE_MODELS=MyStorage.zip

REM ############### Configures PATH
set PATH=%JRE_ROOT%

REM # Expand the path in order to contain customscripts shipped with
REM # resource models
SET
PATH="%AME_HOME%\sara\work\rmbundles\customscripts;%AME_HOME%\sara\work\rmbundl
es\lib;%AME_HOME%\sara\config;%PATH%"

REM # Add the PATH environment variable in the env.properties configuration
file.
REM # It could be used by the Shell utility provided by the Ame Core Services.
echo PATH=%SCENARIO_HOME%\work\rmbundles\customscripts>
"%AME_HOME%\sara\config\env.properties"

REM ############### Configure Classpath
REM # RME JARS
SET CLASSPATH=%AME_HOME%\lib\ame-core.jar
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\ame-utils.jar;

REM # CLOUDSCAPE JAR
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\db2j.jar;

REM # PLUGIN JARS
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\eseif.jar
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\eif.jar
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\alcm.jar;
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\logutil.jar;
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\sal.jar

REM # NLS JARS
```

```
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\AmeCoreMessages.jar;
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\AmeMessages.jar

REM # LOGGING JARS
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\jlog.jar

REM # CONFIG DIR
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\config

REM # XERCES DIR (Required for J2SE 1.3.1, J2SE 1.4.1 picks it up from endorsed
dirs )
REM # SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\endorsed\xercesImpl.jar;
REM # SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\endorsed\xmlParserAPIs.jar

REM # RHINO SAVASCRIPT JAR
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\js.jar

REM # Add support for CimM12 mode
SET CLASSPATH=%CLASSPATH%;%AME_HOME%\lib\ame-m12-mode.jar

REM # Add application jakata regex utility jar files
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\jakarta-regexp-1.3.jar

REM # Add application jakata commons utility jar files
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\commons-net-1.3.0.jar

REM # Add database driver
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\db2jcc.jar

REM # Add CBE library files
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\common.jar
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\ecore.jar
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\hlcbe101.jar
SET CLASSPATH=%CLASSPATH%;%SCENARIO_HOME%\lib\hlcore.jar

cd "%SCENARIO_HOME%"
"%JRE_HOME%\bin\java" -cp "%CLASSPATH%" -jar lib/ScenarioApplication.jar
%RESOURCE_MODELS%
```

The embedding application ITSO_RB_AME takes the resource model type as the argument for loading the resource model instances. The **startITSO_RB_AME** command will run with the associated resource model type. For example, if MyStorage.zip is the resource model package, the batch file (or script files) can be run as:

```
startITSO_RB_AME.bat
```

Note: Operating systems might have a limit on the length of the command or the length of the environment variables. This creates difficulties when the "SET CLASSPATH" (or equivalent) command in the command prompt exceeds the limit defined by the OS. A workaround is to clear the CLASSPATH environment variable before starting the AME.

7.2 Structure of resource model

The resource model ZIP file is one single ZIP file with <RMType>.zip as the name format. The <RMType>.zip file is composed of a set of other ZIP files that have __<interp>__<RMType>.zip as the name format. These files are used by the AME and are called interp dependent files. The resource model package structure is:

interp
: Indicates the platform for which the resource model is designed. For example, w32-ix86 indicates the supported Microsoft Windows platforms. Allowed values are w32-ix86, hpux10, os400, solaris2, linux-ix86, os2-ix86, aix4-r1, linux-s390, and zos.

RMType
: A single resource model can be designed to run on several interp types. The resource model bundle contains one interp-dependent file for each interp. Example 7-9 shows a resource model bundle for a resource model designed for the solaris2 and w32-ix86 platforms.

Example 7-9 Generic structure of a resource model

```
<RMType>.zip
{
__w32-ix86__<RMType>.zip
__solaris2__<RMType>.zip
}
```

For a resource model installation on a particular platform, only the associated interp-dependent file is considered; all other entries in the ZIP file are skipped.

In the example shown in Example 7-9, on a Windows platform, only __w32-ix86__RMType.zip is considered. If it is not found, the installation fails.

Example 7-10 on page 248 shows the structure of the __interp__<RMType>.zip file.

Example 7-10 Generic structure of a resource model type

```
__interp__<RMType>.zip
{
RMType.js
RMType.xml
msgcat/<locale>/<RMType>.cat
Class/<RMType>_<locale>.class
[One or more .mof files]
[One or more jar files / native shared libraries implementing
ILTs]
}
```

The structure includes:

RMType.js — This file has the JavaScript code, which is executed after the resource model is installed and started.

RMType.xml — This is an XML file that contains various parameters that control the resource model.

RMType.cat — This is a catalog file that has the strings used in the resource model.

As shown in Example 7-10, the resource model bundle actually includes two message bundles:

```
msgcat/<locale>/<RMType>.cat
Class/<RMType>_<locale>.class
```

Both of these contain the same messages. The difference is that one is a message catalog (.cat), and the other is a Java resource bundle (.class). Although a message catalog is included in the resource model bundle, the AME does not use it. It is there for the convenience of an embedding application. The plug-ins that are included with the AME only use the Java resource bundle by default.

7.2.1 Directory tree structure

After the resource model is installed with the AME, the following directory structure is created. Note that the location of the Customscripts subdirectory should be read as work\rmbundles\Customscripts in the following information.

work — The working directory contains runtime information.

- eif — This directory contains the EIF event sink cache and is present only if the event sink is used.
- m12 — This directory contains the CIM-related runtime information (MOF files installed, and so on).

rmbundles — This directory contains installed resource model code for unpacking resource model bundles.

- .installed — This directory contains a signature file that has information about the resource models installed.
- Classes\<ClassName> — This directory contains all the Java Archive (JAR) files that are added as a dependency to the resource model. The <ClassName> is the value of the class specified while building the resource model in the Resource Model Builder.
- Config — The resource model descriptor file is stored in this folder.
- Customscripts — Any file other than a JAR file added as a dependency is extracted to this folder.
- Dec — The JavaScript code that would be executed after the resource model is installed and started is extracted to this folder.
- Mof — The MOF file used by the resource model is extracted to this folder.
- Msgcat — The catalog file that has the strings and messages used in the resource model is extracted to this folder.

RME — This directory contains AME runtime information (resource model instances, contexts, and so on).

7.2.2 The rme.config file

The parameters required by the AME to install the resource model are specified in the rme.config properties file. This file should be present in the JAVA classpath referenced by the AME. A default copy of this file is provided with the AME.

This file should contain at least the following keys:

- dir.anchor

  The root directory where the engine creates its resource models' deployment tree.

- dir.schemas

  Schema directory containing both engine-generated basic schemas and custom action launchers and sinks.

- dir.rmbundles

  The directory where the content of a resource model package is unpacked by the resource model deployer.

- serviceability

  The fully qualified logging and tracing class. If this key is missing, a default tracing and logging utility is used. We strongly recommend using the default only for development purposes.

- modes

  CimM12, by default its value is the Tivoli M12 DMTF CIM mode class. This mode uses the Touchpoint Service Layer (TSL) to collect data from resource models' Interface Library Types (ILTs) instances.

- datalog.db.jdbc.driver

  The class implementing the JDBC driver for the database used to store the resource models' logged data. By default, an IBM Cloudscape instance is used.

- datalog.db.url

  This contains the URL used by the JDBC driver to connect to the database.

Example 7-11 provides an example of the rme.config file.

Example 7-11 The rme.config file example

```
dir.anchor=work
dir.schemas=config
dir.rmbundles=work/rmbundles
serviceability=com.ibm.amw.plugin.log.ServiceabilityImpl
modes.CimM12=com.ibm.amw.rmemode.m12.CimM12ModeImpl
datalog.db.jdbc.driver=com.ibm.db2j.jdbc.DB2jDriver
datalog.db.url=jdbc:db2j:data
```

7.2.3 The m12_mode.properties file

The m12_mode.properties file is required when the default AME mode, which is based on the Tivoli M12 DMTF CIM implementation, is used as the data collection mode.

In this case, the following keys should be customized to have it work properly:

- com.ibm.amw.m12.dir.cimom

  The directory where parsed MOF files are stored through Java serialization.

- com.ibm.amw.m12.dir.jars.dynamic

  The directory location for resource models dynamic dependencies, for example, JAR files.

- com.ibm.amw.m12.dir.jars.static

  The directory location where static dependencies, such as JAR files common to all the resource models belonging to a specific Proactive Analysis Component (PAC), are stored.

- com.ibm.amw.m12.dir.mof

  The directory from which the M12 mode loads the MOF files associated with a specific resource model. The files are parsed and stored in the location specified by the com.ibm.amw.m12.dir.cimom key.

For example, the SARA application specifies the m12 property file, as shown in Example 7-12.

Example 7-12 The m12_mode.properties file example

```
com.ibm.amw.m12.dir.cimom=./work/m12/cimom/persistence/
com.ibm.amw.m12.dir.jars.dynamic=./work/rmbundles/Classes/
com.ibm.amw.m12.dir.jars.static=./work/m12/shared/
com.ibm.amw.m12.dir.mof=./work/rmbundles/Mof/
```

7.2.4 Structure of the JavaScript file

The JavaScript code contains the algorithm that governs the entire monitoring process. This file is also called a decision tree script. The JavaScript file used with a resource model has a .js extension. The Resource Model Builder automatically generates the JavaScript decision tree script, which contains the information you specified when you created and configured the resource model. The script has a default structure that is common to all resource models. By default, the decision tree script contains three basic functions, but you can add more. The default functions are:

- SetDefaultConfiguration

  The SetDefaultConfiguration function initializes the object on the basis of the settings defined in the events, thresholds, and parameters dialog boxes. If you change one of the settings contained in these dialog boxes, the corresponding data is updated in this function. On the contrary, you cannot change those settings by modifying them directly in this function. The SetDefaultConfiguration function is called just once, when the resource model is started. Therefore, if necessary, you can write additional initialization code at the end of this function.

- Init

  The Init function is called after the settings defined in the SetDefaultConfiguration function have been replaced with the values coming from the resource model descriptor file. At this point, the settings are replaced with the new ones.

- VisitTree

  The VisitTree function contains the monitoring algorithm and is called at the beginning of each cycle time. Write the monitoring algorithm in JavaScript, defining how to use all the values and variables previously set. This function checks the algorithm and implements it. It processes the collected data according to threshold and parameter settings, and, if necessary, sends an event or executes an action.

7.2.5 Structure of the resource model descriptor file

The resource model descriptor is an XML document that conforms to a set of W3C schemas that can be modified to change various parameters that control the resource model.

The Resource Model Builder automatically generates the descriptor when an AME resource model package is created. The hosting AME environment parses this XML files for initializing various parameters associated with the resource model.

The resource model descriptor is available in the __interp__<RMType>.zip file.

A typical resource model descriptor file can have information embedded in tags, as shown in Example 7-13. A detailed description of the resource model descriptor configuration XML file is provided in the *Autonomic Management Engine Developer's Guide*. We suggest that you become familiar with the various tags used in these configuration files.

Example 7-13 Resource model descriptor template

```
<Rm category="category", rmSpecification=" major.minor",
typeName="type",version="major.minor">
<Description></Description>
<Identify>
<Analyzer>
<Modes></Modes>
<DecFile></DecFile>
<CycleTime></CycleTime>
<Thresholds></Thresholds>
<StrParameters>
<Parameters></Parameters>
```

```
</StrParameters>
</Analyzer>
<Aggregator>
<Indication></Indication>
<ClearingPolicy></ClearingPolicy>
<NotificationPolicy></NotificationPolicy>
<MsgInfo></MsgInfo>
<KeyFields></KeyFields>
<Rule></Rule>
<Consolidation></Consolidation>
<Severity></Severity>
</Aggregator>
</Identify>
<Notify></Notify>
<Cure></Cure>
<DataLog><DataLog>
</RM>
```

7.2.6 Structure of the MOF file

The Common Information Model (CIM) provides a common definition of management information for systems, networks, applications, and services and allows for vendor extensions. The CIM is composed of a specification and a schema. The schema provides the actual model descriptions, and the specification defines the details for integration with other management models. The CIM schema enables applications from different developers on different platforms to describe management data in a standard format so that it can be shared among a variety of management applications. There are potentially many ways in which CIM management information could be represented to exchange information. The CIM specification defines a language based on the Interface Definition Language (IDL) called Managed Object Format (MOF), which defines the elements of the CIM. The MOF language enables a developer to use a standard text editor or development environment to describe a set of CIM classes. See Example 7-14 on page 254.

The main components of a MOF specification are textual descriptions of:

- Classes

  Define the structure of the information held in the store and are collection of instances that support the same type.

- Properties

  Define individual characteristics of each class.

- Associations

  Define the relationship between two or more classes and contain two or more references.

- References

  Define the role that each class plays in that association.

- Methods

  Define the behavior of the class.

- Instance declarations

  Define the instruction to create a new instance where the object's key values do not already exist, or an instruction to modify an existing instance where an object with identical key values already exists.

Example 7-14 MOF file

```
// Comments about the MOF file
//
[ Description ("Description of the class"),
M12_Instrumentation("Java.package name of ILT|mappingstring|operation"),
provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
]
class ClassName
{
VariableType VariableName; [Key, Description("Descriptive Text for the Variable Name")]
}
```

The keywords used in the MOF file are:

- M12_Instrumentation

  This is a nonstandard CIM qualifier that applies to classes, properties, and methods. In order for a resource model to run in the AME, the MOF file should support the M12_Instrumentation qualifier. The M12 specification requires that all classes have an ENUM operation. This tells the class loader which class to associate with the enumeration functionality of the instrumentation. You can also provide a GET operation for the class to specify which class will be associated with all non-specified properties. That is, it is possible to associate specific properties within the class to specific classes on an individual basis, which allows the singular collection of the property data through the class method getProperty or getmultipleproperties.

- package name of ILT

  The Java class that implements the ILT, and it must be specified with the complete package (with no .class extension files).

- mappingstring

  A string whose meaning is known to the ILT.

- operation

  The type of operation to be performed. The allowed operations are:

  – ENUM (only for the qualifier associated with the class) enumerates the resource instances.

  – GET (for the qualifier associated with the class and property) gets the resource attributes.

  – INVOKE (for the qualifier associated with class and method) invokes an instance or class method.

- Provider

  A Common Object Model (COM) server that communicates with managed objects to access data and event notifications from a variety of sources such as the system registry or a custom DLL. After the information is collected, providers forward them to the CIM Object Manager for integration and interpretation. The provider used by the AME resource model is com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider.

- class

  A collection of the properties or methods.

- Key

  The key property qualifier used to define which property is the primary key or index to the class. After you have the MOF file created, you can use your operating system's **mofcomp** utility to install the MOF file.

- Description

  A qualifier that provides a simple text description of the attributes present in the MOF file. You must enclose the description text in curved brackets and straight (non-directional) quotation marks. The description qualifier is highly recommended as a best practice for both classes and class properties, because the description information can be viewed by CIM browsing utilities after the MOF file is compiled into a localized CIM repository. This is extremely useful when multiple resource model developers will implement the same CIM class.

Part 3

Case study scenarios

In this part, we provide three Problem Determination scenarios.

8

Problem Determination scenario one

This chapter describes how to solve a particular file system problem using various components of the IBM Autonomic Computing Toolkit. This scenario shows how autonomic computing is used to monitor the IT environment and perform corrective actions when IT resources utilization goes beyond the definition of the threshold limit.

This chapter describes the definition, implementation, package, and execution of this scenario and covers the following topics:

- Scenario description
- Resource model definition
- Creating the Managed Object Format file
- Creating the Common Information Model classes
- Implementing the utility class
- Implementing the Instrumentation Library Type class
- Creating the installation package
- Scenario execution

8.1 Scenario description

This scenario provides a solution when an application reaches a threshold limit of disk utilization. A custom resource model will be monitoring free disk space on the file system. If the free space is below a certain limit, the first corrective action is to ZIP all old files into a backup archive. If the free space continues to drop below a second limit, the second corrective action is to delete old backup archives.

The scenario uses custom Common Information Model (CIM) Instrumentation Library Type (ILT) classes as data sources and for performing corrective actions. The configuration of the resource model is performed with both parameters and thresholds.

Figure 8-1 shows an outline of all components of the scenario. The following sections provide details of this example scenario implementation.

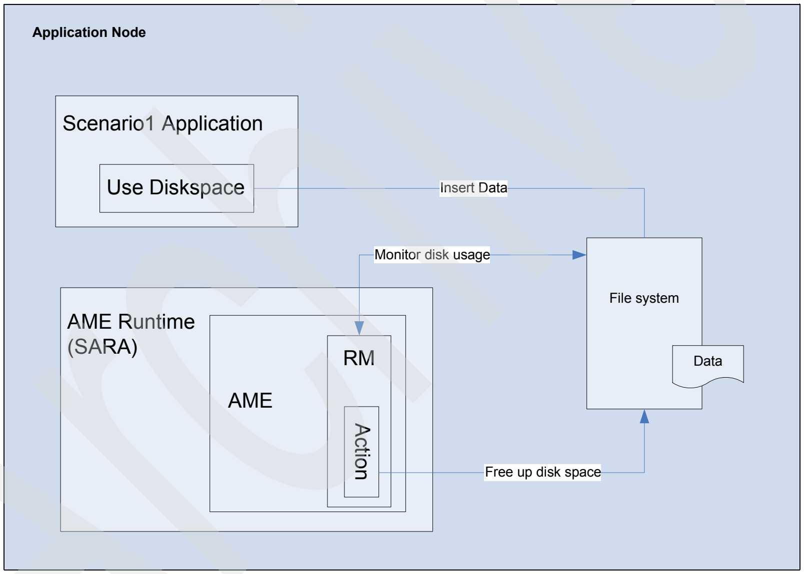


Figure 8-1 Components of scenario one

8.2 Resource model definition

This section defines a resource model for this scenario using the Tivoli Resource Model Builder. Refer to 6.2, "Resource models" on page 199, which outlines the different components of resource models.

8.2.1 Creating the Eclipse project

To create a Tivoli resource model project in the Resource Model Builder, perform the following steps:

1. To start the Eclipse, select **Start** → **Programs** → **IBM Autonomic Computing Toolkit** → **Development kit** → **Start Eclipse**.
2. Open the Tivoli Management Perspective by selecting **Window** → **Open Perspective** → **Other** → **Tivoli Management Perspective**.
3. Create a new Tivoli Management Project by selecting **File** → **New** → **Project** and then selecting **Tivoli Management Project**, as shown in Figure 8-2.

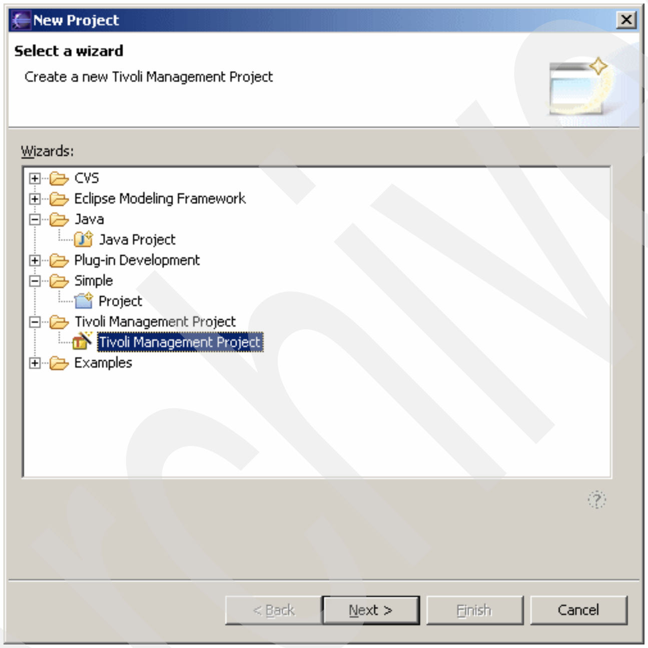


Figure 8-2 New Tivoli Management Project

4. Click **Next**.
5. Enter your Project name, for example `ACPD_Scenario_One`, and click **Next**.
6. Now, it is not possible to start the Basic Resource Model Wizard or the Empty Resource Model Wizard. To create an empty project, just click **Finish**.

8.2.2 General resource model settings

Now that the project is in place, resource models can be created. To create an empty resource model, perform the following steps:

1. Right-click the project **MyManagementProject** and select **New** → **Empty Resource Model Wizard**. The Empty Resource Model Wizard window opens, as shown in Figure 8-3.

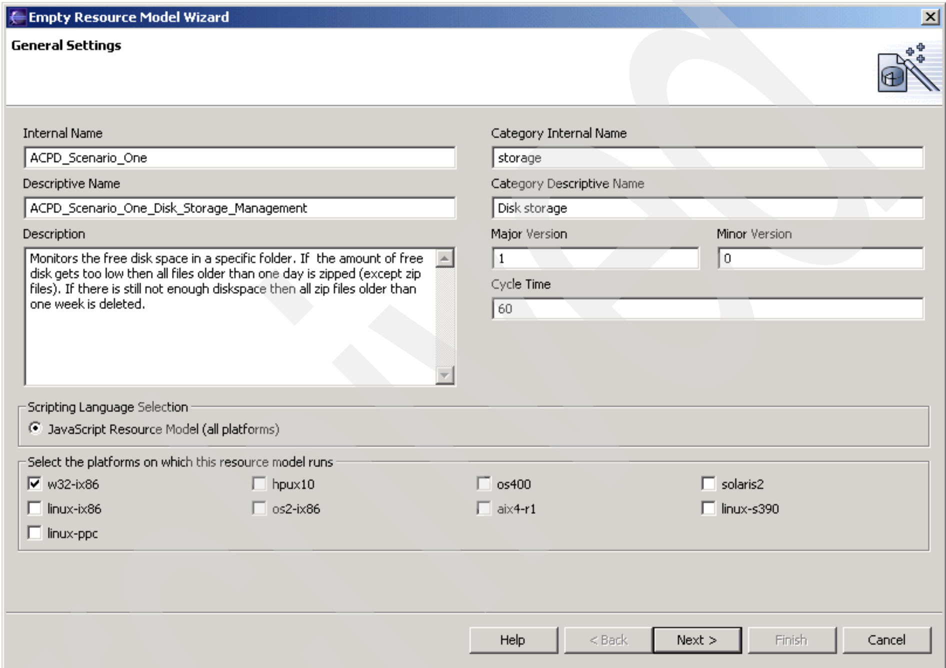


Figure 8-3 Tivoli Empty Resource Model Wizard

2. Click **Next** after filling all the fields with the following information:

Internal Name	`ACPD_Scenario_One`
Descriptive Name	`ACPD_Scenario_One_Disk_Storage`
Description	`Monitors the free disk space in a specific folder. If the amount of free disk gets too low, then all files older than one day are zipped (except zip files). If there is still not enough diskspace, then all zip files older than one week are deleted.`
Category Internal Name	`storage`

Category Descriptive Name	`Disk storage`
Major Version	`1`
Minor Version	`0`
Cycle Time	`60`
Script Language Selection	Set the script type of the decision algorithm to **JavaScript**, which enables the resource model to be used on all platforms.
Select the platforms	**w32-ix86**

3. Now, select the project name to contain the resource model, as shown in Figure 8-4. Specify the File name of the resource model and click **Finish**. By default, the wizard suggests the internal name followed by `.jrm`.

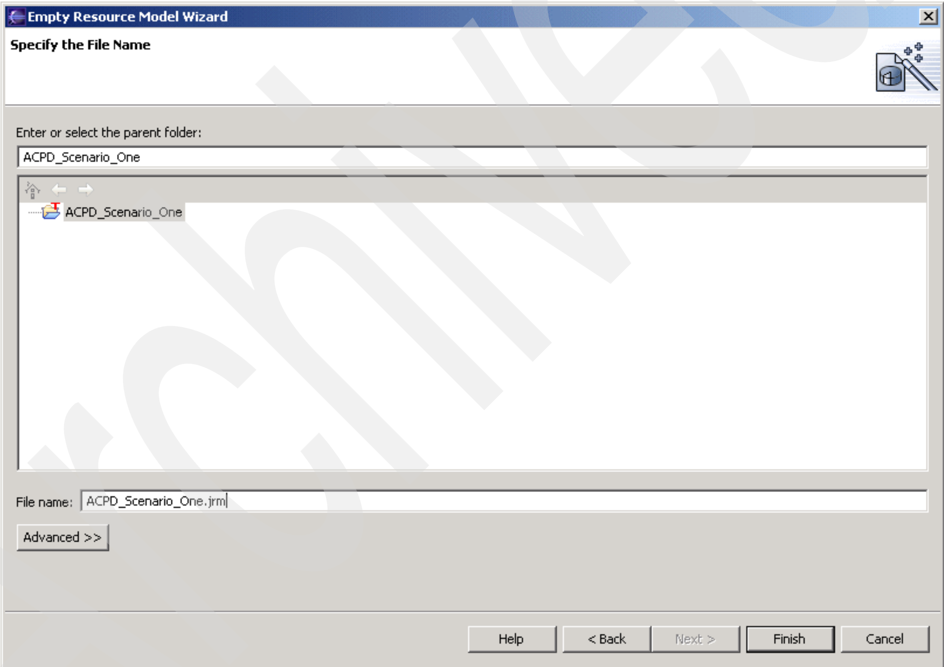


Figure 8-4 Select project for resource model and specify its name.

Now, the workspace should look like the one shown in Figure 8-5 on page 264.

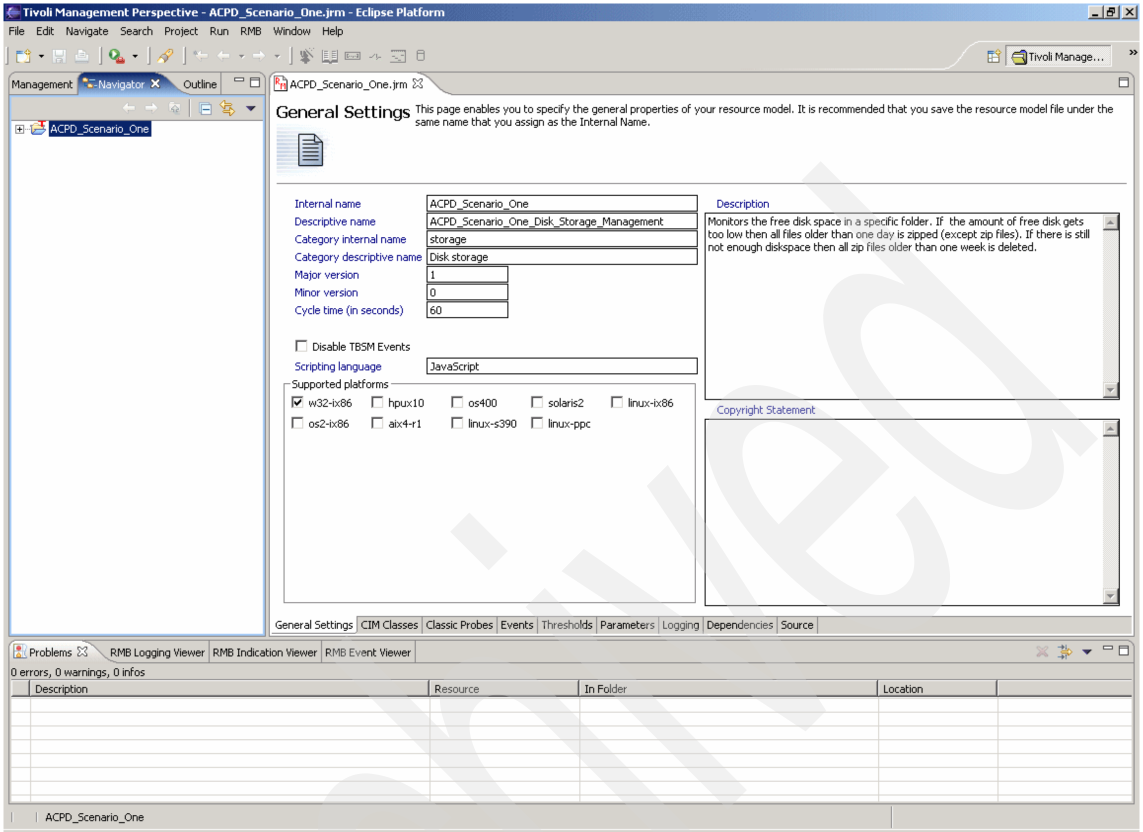


Figure 8-5 General Settings of scenario one

8.3 Creating the Managed Object Format file

In this section, we create a Managed Object Format (MOF) file for scenario one. Refer to “Managed Object Format (MOF)” on page 209 and 7.2.6, “Structure of the MOF file” on page 253 for details.

Example 8-1 on page 265 shows the MOF file created for our example scenario.

Example 8-1 ACPD_Scenario_One.mof for scenario one

```
// The class ITSO_SG246665 is used to group all the custom CIM clases in
// the CIM repository
[
  description("Used to group all CIM classes used in the RedBook SG246665."),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
]
class ITSO_SG246665_SC1 {
};

[
  description("CIM class for monitoring disk space, the CIM is implemented with a custom ILT.
The class contains a property with the ammount of free disk space in bytes and two methods for
performing corrective actions."),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider"),
  M12_Instrumentation{
    "Java.com.ibm.itso.sg246665.rm.example.diskspace.ilt.FreeDiskStorageILT || ENUM",
    "Java.com.ibm.itso.sg246665.rm.example.diskspace.ilt.FreeDiskStorageILT || GET"}
]
class DiskSpace_Monitor:ITSO_SG246665_SC1 {
  [
    description("Key specifying the path on which the free space is monitored."),
    key
  ]
  string path;

  [
    description("Property with the ammount of free space.")
  ]
  sint64 freeSpace;

  [
    description("Zips files older than the specified number of days."),
    M12_Instrumentation
{"Java.com.ibm.itso.sg246665.rm.example.diskspace.ilt.FreeDiskStorageILT || INVOKE"}
  ]
  string zipOldFiles([in] sint32 days);

  [
    description("Delete zip backup files older than the specified number of days."),
    M12_Instrumentation
{"Java.com.ibm.itso.sg246665.rm.example.diskspace.ilt.FreeDiskStorageILT || INVOKE"}
  ]
  string deleteOldZipFiles([in] sint32 days);
};
```

8.4 Creating the Common Information Model classes

In this scenario, a Common Information Model (CIM) class is created to retrieve the free disk space from a particular directory. DiskSpace_Monitor is the CIM class definition for the example scenario. The CIM class is implemented with an ILT class and a utility class.

The CIM class implementation is created in a separate project called *ACPD_Scenario_One_CIM*.

Perform the following steps to define the DiskSpace_Monitor CIM class for the example scenario:

1. Create a new Java project by selecting **File** → **New** → **Project** and then selecting **Java Project**, as shown in Figure 8-6. Click **Next**.

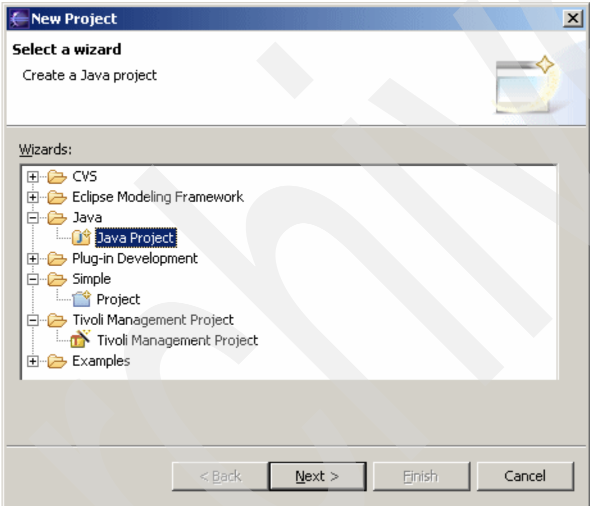


Figure 8-6 New Java Project

2. Specify the Project name, for example, `ACPD_Scenario_One_CIM`, and select **Create separate source and output folders**, as shown in Figure 8-7. Click **Next**.

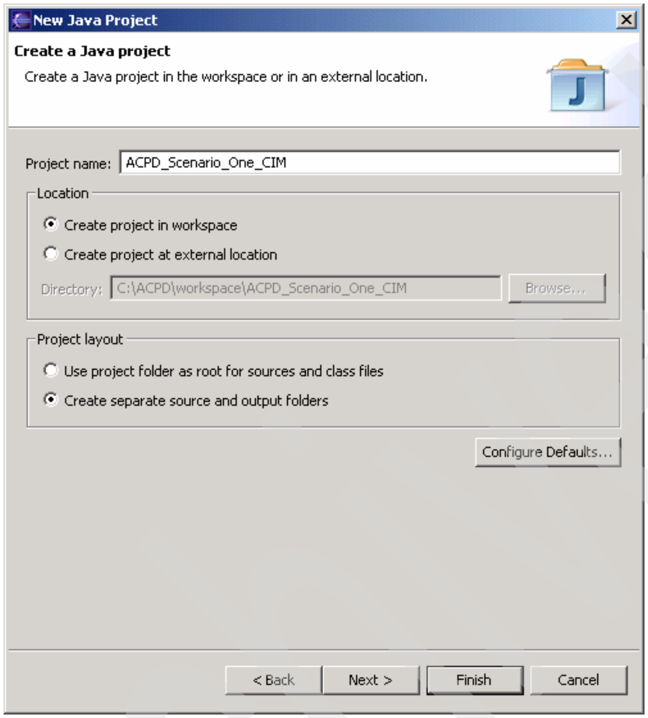


Figure 8-7 ACPD_Scenario_One_CIM Java Project

3. On the Libraries tab, click **Add External JARs** and find the **ame-m12-mode.jar**. This JAR file contains the ILTInterface interface. If the default installation directory was used for the Autonomic Management Engine, it will be found under C:\Program Files\IBM\AutonomicComputingToolkit\AutonomicManagementEngine\lib.

   The build path should look as in Figure 8-8. Click **Finish**.

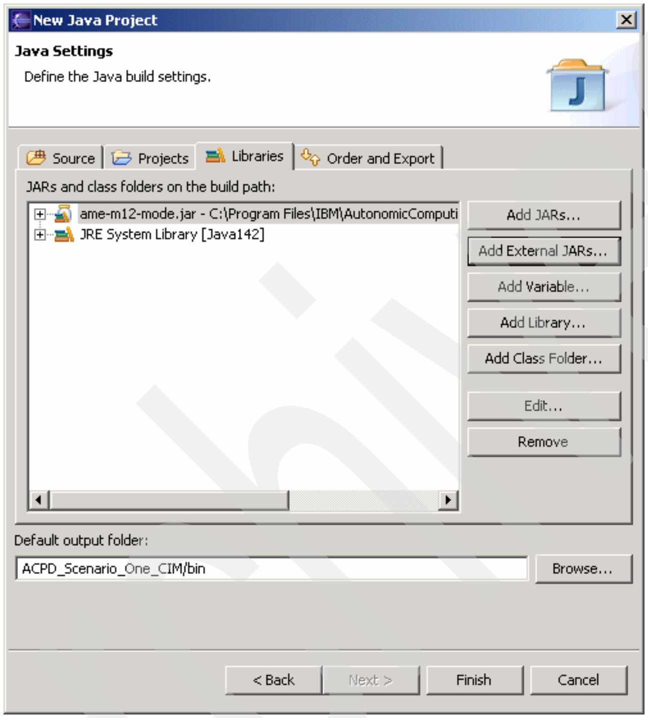


Figure 8-8 ame-m12-mode.jar added to build path

4. Create a new utility class to collect the required data by right-clicking the newly created project and selecting **New** → **Class**. Specify a Package name, for example, `com.ibm.itso.sg246665.sc1.rm.diskspace.ilt`, and the Name, for example, `FreeDiskStorage`, as shown in Figure 8-9. Click **Finish**.

   The implementation of this Java class implementation must contain all the required utility methods and is described in 8.5, "Implementing the utility class" on page 270.

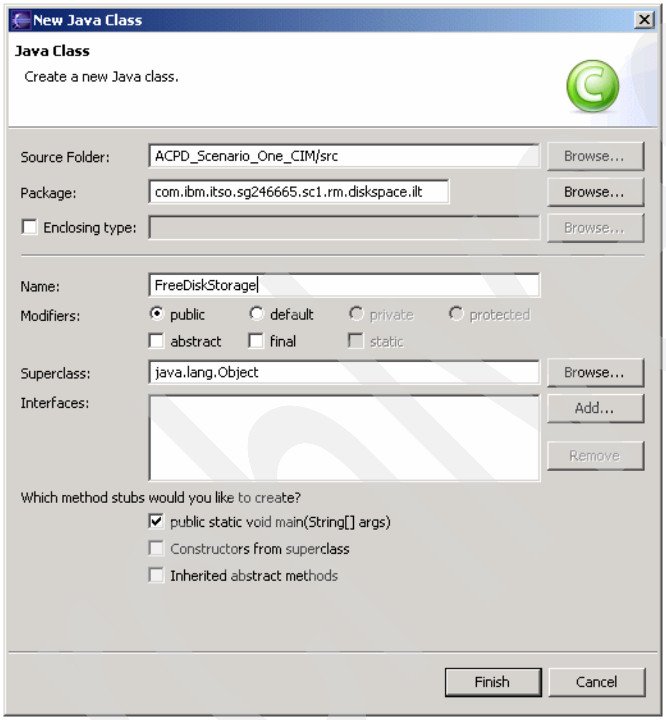


Figure 8-9 FreeDiskStorage class

5. Now, create a new ILT class by right-clicking the project and selecting **New** → **Class**. Specify a Package, for example, `com.ibm.itso.sg246665.example.rm.diskspace.ilt`, and the Name, for example, `FreeDiskStorageIlt`, and add the ILTInterface, as shown in Figure 8-10 on page 270. Click **Finish**.

   The implementation of this ILT Java class implementation must contain all the required methods specific to the ITL interface and is described in 8.6, "Implementing the Instrumentation Library Type class" on page 273.

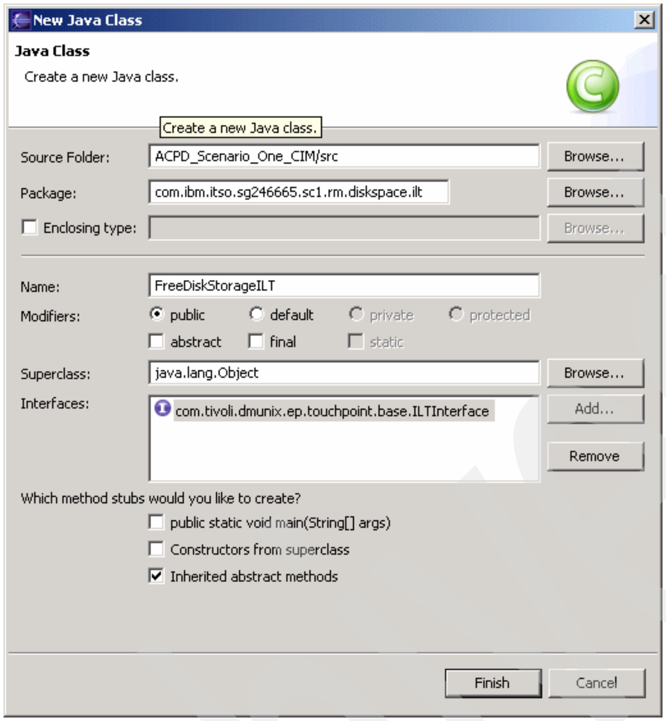


Figure 8-10 FreeDiskStorageILT class

8.5 Implementing the utility class

The implementation of the utility class requires the development of a Java class including the following three utility methods.

- getFreeDiskSpace()
- zipOldFiles()
- deleteOldZipFiles()

The utility class described in this section is available for download. Refer to Appendix A, “Additional material” on page 379 for details.

getFreeDiskSpace()

This method finds available space left under a specified path. Example 8-2 on page 271 shows the implementation.

Example 8-2 getFreeDiskSpace implementation

```
public static long getFreeDiskSpace(String path) throws Exception {
    long bytesFree = -1;

    try {
      Process p = Runtime.getRuntime().exec(
          new String[] { "cmd", "/C", "dir", path }); // for Windows
      BufferedReader reader = new BufferedReader(new InputStreamReader(
          new BufferedInputStream(p.getInputStream())));
      String line;
      while ((line = reader.readLine()) != null) {
        if (line.endsWith("bytes free")) { // for dir on Windows
          StringTokenizer tokenizer = new StringTokenizer(line, " ");
          tokenizer.nextToken();
          tokenizer.nextToken();
          bytesFree = Long.parseLong(tokenizer.nextToken().replaceAll(",",
              ""));
        }
      }
    } catch (NumberFormatException e) {
      e.printStackTrace();
      throw e;
    } catch (IOException e) {
      e.printStackTrace();
      throw e;
    }
    return bytesFree;
  }
```

zipOldFiles()

This method zips files older than a specified number of days into a backup file. Example 8-3 shows the implementation.

Example 8-3 zipOldFiles implementation

```
public static void zipOldFiles(String path, int daysOld) throws Exception {
    // First find all files under the path that is too old.
    File filePath = new File(path);

    // Create the file filter
    FileFilter filter = new OldFilesFilter(daysOld);

    // find files to zip
    File[] filesToZip = filePath.listFiles(filter);

    if (filesToZip != null && filesToZip.length > 0) {
      // Some files need to be zipped
```

```
    File zipFile = new File(filePath, "backup_"
        + System.currentTimeMillis() + ".zip");
    System.out.println("Creating zip backup " + zipFile.getName());

    ZipOutputStream zipOut = new ZipOutputStream(new FileOutputStream(
        zipFile));
    zipOut.setMethod(ZipOutputStream.DEFLATED);
    for (int i = 0; i < filesToZip.length; i++) {
      System.out.println("Adding file " + filesToZip[i].getName()
          + " to " + zipFile.getName());

      ZipEntry zipEntry = new ZipEntry(filesToZip[i].getName());
      zipEntry.setSize(filesToZip[i].length());
      zipEntry.setTime(filesToZip[i].lastModified());

      zipOut.putNextEntry(zipEntry);
      byte[] fileContent = null;
      FileInputStream sourceFile = null;
      try {
        fileContent = new byte[1024];
        sourceFile = new FileInputStream(filesToZip[i]);
        int size = 0;
        while ((size = sourceFile.read(fileContent)) != -1) {
          zipOut.write(fileContent, 0, size);
        }
      } finally {
        // make sure the file is always closed of open.
        if (sourceFile != null) {
          sourceFile.close();
        }
      }
    }
    zipOut.close();
    // Now delete the old files
    for (int i = 0; i < filesToZip.length; i++) {
      System.out.println("Deleting file " + filesToZip[i].getName());
      filesToZip[i].delete();
    }
    System.out.println("Backup done (" + zipFile.getName() + ")");
  }
}
```

deleteOldZipFiles()

This method deletes old backup files from a specified path. Example 8-4 on page 273 shows the implementation.

Example 8-4 deleteOldZipFiles implementation

```
public static void deleteOldZipFiles(String path, int daysOld)
      throws Exception {
    // First find all files under the path that is too old.
    File filePath = new File(path);

    // Create the file filter
    FileFilter filter = new OldZipFilesFilter(daysOld);

    // find files to zip
    File[] filesToDelete = filePath.listFiles(filter);

    if (filesToDelete != null && filesToDelete.length > 0) {
      for (int i = 0; i < filesToDelete.length; i++) {
        System.out.println("Deleting " + filesToDelete[i].getName());
        filesToDelete[i].delete();
      }
    }
  }
```

8.6 Implementing the Instrumentation Library Type class

Implementation of the Instrumentation Library Type (ILT) Java class requires the development of the following ITL interface methods:

- Default constructor
- enumerateInstances()
- getProperty()
- getMultipleProperties()
- invokeMethod()

The ILT java class described in this section is available for download. Refer to Appendix A, “Additional material” on page 379 for details.

Default constructor

The M12 public methods handle the actual implementation defined by the ILTInterface. Which of the methods defined by the ILTInterface to implement depends on the purpose of the CIM class. A few of the methods are implemented for the disk storage example.

First, we recommend that you create a field of type com.ibm.amw.touchpoint.cimom.TSLTrace, which will be able to log information into the AME trace file. Initialize the field in the constructor using com.ibm.amw.touchpoint.cimom.TSLTraceImpl, as shown in Example 8-5

> **Note:** Some examples of ILT classes use a trace class called com.tivoli.javautils.Trace. This is an old class in which some of the non-deprecated methods do not work.

Example 8-5 FreeDiskStorageILT constructor

```
public FreeDiskStorageILT() {
    trace = TSLTraceImpl.getInstance();
    trace.logMsg(TSLTrace.DEBUG_MAX, FreeDiskStorageILT.class.getName(),
        "FreeDiskStorageILT", " Created. ");
  }
```

enumerateInstances()

For each instance, an appropriate configuration is created. The return value is an enumeration of M12ObjectIdentity objects that identify all the instances belonging to the class specified. The enumerate instance method developed for this example scenario is shown in Example 8-6, where the number of instances depends on the number of paths monitored.

Example 8-6 enumerateInstances method

```
public Enumeration enumerateInstances(
      M12ClassPath m12classpath,
      String mapping,
      ParameterSet parSet) throws M12Exception {
    try {
      trace.entry(TSLTrace.DEBUG_MAX, this, "enumerateInstances",
          new Object[] { m12classpath, mapping, parSet });
      // Vector which will contain the Object identities
      Vector returnVector = new Vector();

      /***********************************************************************
       * Get the parameters from the caller (the service object). We are
       * obtaining the parameters prior to declaring instances so we can
       * configure an instance for each parameter value. These parameters are
       * declared in the PARAMETERS section of the resource model.
       **********************************************************************/
      Vector vParam = (Vector) parSet
          .getParam(FreeDiskStorageILT.PAR_FREEDISKSTORAGE_PATH_NAME);

      if (vParam == null) {
```

```
        trace.traceMsg(TSLTrace.ERROR, this.getClass().getName(),
            "enumerateInstances", "ERROR: no parameters found for "
                + FreeDiskStorageILT.PAR_FREEDISKSTORAGE_PATH_NAME);
        trace.exit(TSLTrace.DEBUG_MAX, this, "enumerateInstances", null);
        return null;
      }

      /************************************************************************
       * For each value of FreeDiskStorage_Path create an instance with an
       * identity and configure the key properties (path) for it.
       ***********************************************************************/
      Enumeration pathEnum = vParam.elements();
      while (pathEnum.hasMoreElements()) {
        String path = (String) pathEnum.nextElement();
        trace.traceMsg(TSLTrace.INFO, this, "enumerateInstances",
            " configuring path " + path);
        M12PropertySet m12propertyset = new M12PropertySet();
        // must set all key properties. in this case only path is a key
        // property
        m12propertyset.setProperty("path", path);
        M12IdentityElement m12identityelement = new M12IdentityElement(
            m12classpath.getClassName(), m12classpath.getNameSpace(),
            m12propertyset);
        returnVector.add(new M12ObjectIdentity(
            new M12IdentityElement[] { m12identityelement }));
      }
      trace
          .exit(TSLTrace.DEBUG_MAX, this, "enumerateInstances", returnVector);
      return returnVector.elements();

    } catch (Exception exception) {
      trace.traceException(this, "enumerateInstances", "Exception occured: "
          + exception.getMessage(), exception);
      throw new M12Exception(exception);
    }
  }
```

getProperty()

The getProperty member function implementation of this example scenario using the FreeDiskStorage utility class is shown in Example 8-7 on page 276. This method returns the value for the M12_Instrumentation qualifier property, as mentioned in the MOF file.

Example 8-7 getProperty implementation

```
public String getProperty(
      M12ObjectIdentity objectIdentity,
      String propertyName,
      String mapping,
      ParameterSet parSet) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "getProperty;", new Object[] {
        objectIdentity, propertyName, mapping, parSet });

    // First verify that it is a supported property
    // Only freeSpace is supported
    if (propertyName.equals("freeSpace")) {
      // We only created one identityElement in the object identity
      M12IdentityElement identityElement = objectIdentity.getScopingPath()[0];
      M12PropertySet propertySet = identityElement.getIdentity();
      // Get the path which is a key property.
      String path = propertySet.getProperty("path");
      // obtain the free space from the utility class
      try {
        String result = String
            .valueOf(FreeDiskStorage.getFreeDiskSpace(path));
        trace.exit(TSLTrace.DEBUG_MAX, this, "getProperty", result);
        return result;
      } catch (Exception e) {
        trace.traceException(this, "getProperty",
            "Error occured caling FreeDiskStorage: " + e.getMessage(), e);
        throw new M12Exception(e);
      }
    } else {
      trace.traceMsg(TSLTrace.ERROR, this, "getProperty;",
          "Properyt not supported: " + propertyName);
      throw new M12Exception(new Exception("Properyt not supported: "
          + propertyName));
    }
  }
```

getMultipleProperties()

The property freeSpace is declared, as shown in Example 8-8

Example 8-8 getMultipleProperties implementation

```
public M12PropertySet getMultipleProperties(
      M12ObjectIdentity objectIdentity,
      Vector propertyNames,
      String mapping,
      ParameterSet parSet) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "getMultipleProperties", new Object[]
{
```

```
            objectIdentity, propertyNames, mapping, parSet });

    // Will contain the return values
    M12PropertySet resultPropSet = new M12PropertySet();

    // We only created one identityElement in the object identity
    M12IdentityElement identityElement = objectIdentity.getScopingPath()[0];
    M12PropertySet propertySet = identityElement.getIdentity();
    // Get the path which is a key property.
    String path = propertySet.getProperty("path");

    // Loop through each of the propertyNames
    Enumeration namesEnum = propertyNames.elements();
    while (namesEnum.hasMoreElements()) {
      String propName = (String) namesEnum.nextElement();

      // Verift that it is a supported property
      // Only freeSpace is supported
      if (propName.equals("freeSpace")) {
        // obtain the free space from the utility class
        try {
          String result = String.valueOf(FreeDiskStorage
              .getFreeDiskSpace(path));
          resultPropSet.setProperty(propName, result);
        } catch (Exception e) {
          trace.traceException(this, "getMultipleProperties",
              "Error occured caling FreeDiskStorage: " + e.getMessage(), e);
          throw new M12Exception(e);
        }
      } else {
        trace.traceMsg(TSLTrace.ERROR, this, "getMultipleProperties;",
            "Properyt not supported: " + propName);
        throw new M12Exception(new Exception("Properyt not supported: "
            + propName));
      }
    }
    trace.exit(TSLTrace.DEBUG_MAX, this, "getMultipleProperties",
resultPropSet);
    return resultPropSet;
  }
```

invokeMethod()

The invoke method in Example 8-9 on page 278 is an instance method invocation. It supports two methods: zipOldFiles and deleteOldZipFiles.

Example 8-9 Instance method invocation

```
public String invokeMethod(
      M12ObjectIdentity objectIdentity,
      String method,
      String mapping,
      ParameterSet params,
      ParameterSet inParams,
      ParameterSet outParams) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "invokeMethod", new Object[] {
        objectIdentity, method, mapping, params, inParams, outParams });

    // First get the instance information
    // We only created one identityElement in the object identity
    M12IdentityElement identityElement = objectIdentity.getScopingPath()[0];
    M12PropertySet propertySet = identityElement.getIdentity();
    // Get the path which is a key property.
    String path = propertySet.getProperty("path");
    // obtain the free space from the utility class

    // Verift that it is a supported method
    // Only zipOldFiles and deleteOldZipFiles is supported
    if (method.equals("zipOldFiles")) {
      try {
        String parDaysOld = (String) inParams.getParam("days");
        // The number may be given as a double from the Resource Model.
        int intDays = Double.valueOf(parDaysOld).intValue();
        FreeDiskStorage.zipOldFiles(path, intDays);
        trace.exit(TSLTrace.DEBUG_MAX, this, "invokeMethod");
      } catch (Exception e) {
        trace.traceException(this, "invokeMethod",
            "Error occured caling zipOldFiles: " + e.getMessage(), e);
        throw new M12Exception(e);
      }
    } else if (method.equals("deleteOldZipFiles")) {
      try {
        String parDaysOld = (String) inParams.getParam("days");
        int intDays = Double.valueOf(parDaysOld).intValue();
        FreeDiskStorage.deleteOldZipFiles(path, intDays);
        trace.exit(TSLTrace.DEBUG_MAX, this, "invokeMethod");
      } catch (Exception e) {
        trace.traceException(this, "invokeMethod",
            "Error occured caling deleteOldZipFiles: " + e.getMessage(), e);
        throw new M12Exception(e);
      }
    } else {
      trace.traceMsg(TSLTrace.ERROR, this, "invokeMethod;",
          "Method not supported: " + method);
      throw new M12Exception(new Exception("Method not supported: " + method));
```

```
    }
    return "Done";
  }
```

The complete FreeDiskStorageILT.java file is provided with this book. Refer to Appendix A, “Additional material” on page 379 for details.

8.7 Creating the installation package

At this point, we have created all the elements required for the resource model. Now, we need to generate the resource model package, which can be run in the Autonomic Management Engine environment. This section includes the steps to create a JAR file for ILT classes and custom CIM configuration using Tivoli Resource Model Builder.

In order to create the installation package for the case study scenario presented in this chapter, we perform the following high-level steps:

1. Creating a JAR file with the ILT class and utility class.
2. Registering custom CIM class using Resource Model Builder.
3. Creating the resource model package.

The following sections provide details about each step.

8.7.1 Creating a JAR file with the ILT class and utility class

To use the ILT Java class and the utility class, create a JAR file by performing the following steps:

1. Right-click the project (**ACPD_Scenario_One_CIM**) and select **Export**. In the Export window, select **JAR file** and click **Next** as shown in Figure 8-11 on page 280.

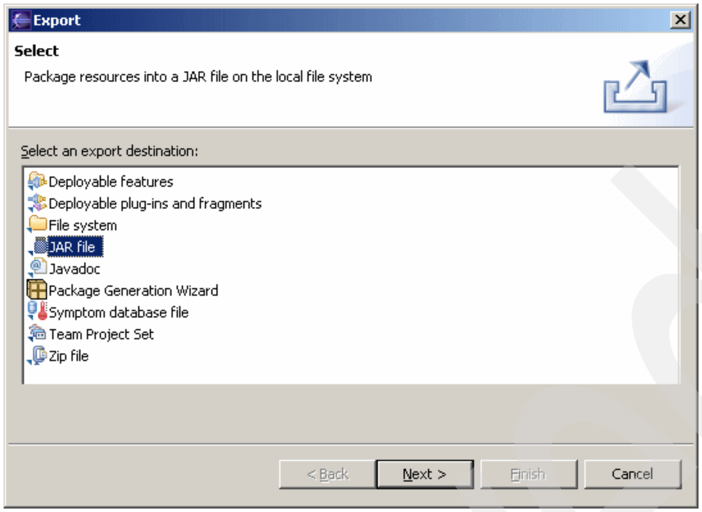


Figure 8-11 Export window

2. Select the **src** folder, make sure the option **Export generated class files and resources** is selected, specify the JAR file name (for example, `ACPD_Scenario_One_CIM.jar`), and click **Next**, as shown in Figure 8-12.

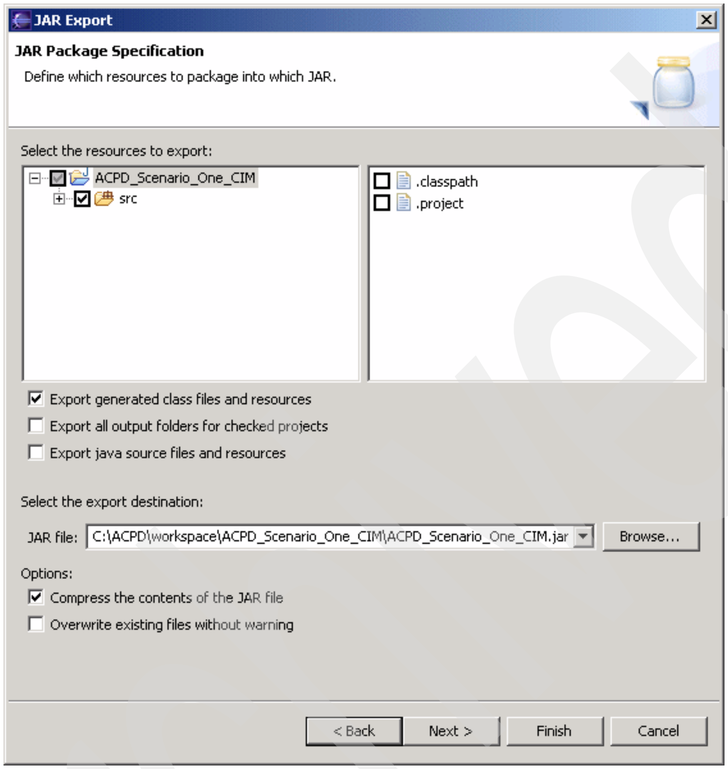


Figure 8-12 JAR Export

3. Select **Save the description of this JAR in the workspace** and click **Finish**, as shown in Figure 8-13 on page 282.

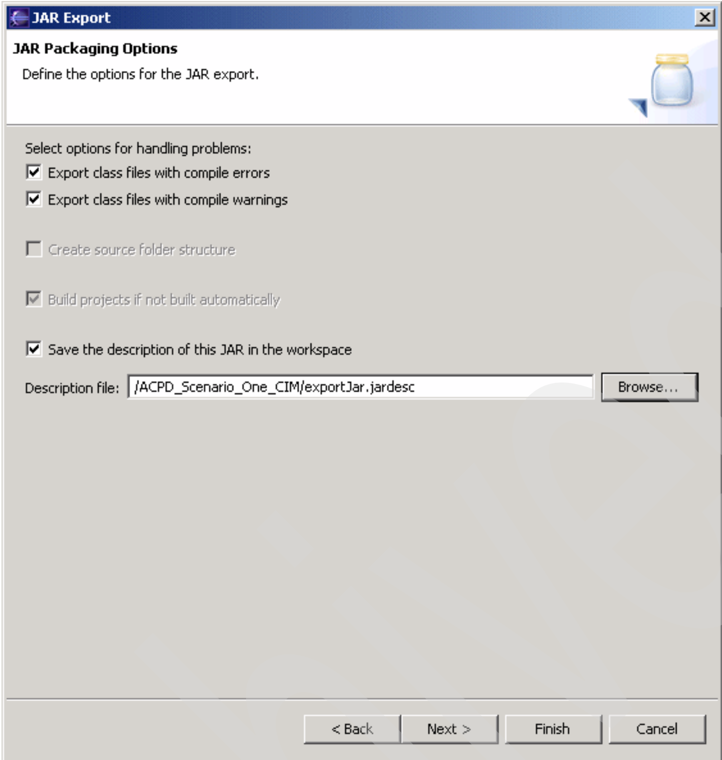


Figure 8-13 JAR Packaging Options

Now, the JAR file ACPD_Scenario_One_CIM.jar is visible under the ACPD_Scenario_One_CIM project.

The JAR file created in this section will be used when defining the dependencies for the resource model.

8.7.2 Registering custom CIM class using Resource Model Builder

This section describe the steps required to register the CIM classes used in the resource model. Refer to 6.2, “Resource models” on page 199, which outlines different subcomponents of the resource model. For our example scenario, we performed the following activities:

1. Including the MOF file into the CIM project
2. Compiling the MOF file
3. Including the CIM class attributes into the resource model
4. Defining events
5. Defining thresholds

6. Defining parameters
7. Defining dependencies
8. Creating the decision tree script

Including the MOF file into the CIM project

The MOF file provides the CIM class definitions used in the resource model and is required to be included in the resource model project folder. In order to do so, perform the following tasks:

1. Copy the ACPD_Scenario_One.mof file, shown in Example 8-1 on page 265, into the ACPD_Scenario_One_CIM project directory, for example, c:\acpd\workspace\acpd_scenario_one.
2. Open the project using Eclipse and refresh the Eclipse GUI.

This MOF file is specific to the scenario presented in this chapter.

Compiling the MOF file

The Resource Model Builder has a CIM repository with CIM classes that can be used. For information about adding custom CIM classes, refer to 6.2.2, “Common Information Model classes” on page 208. To use a CIM class in the resource model and register it in the CIM repository, the MOF file has to be compiled. To compile the MOF file, perform the following steps:

1. From the CIM Classes tab, click the **MOF Wizard** button, as shown in Figure 8-14 on page 284.

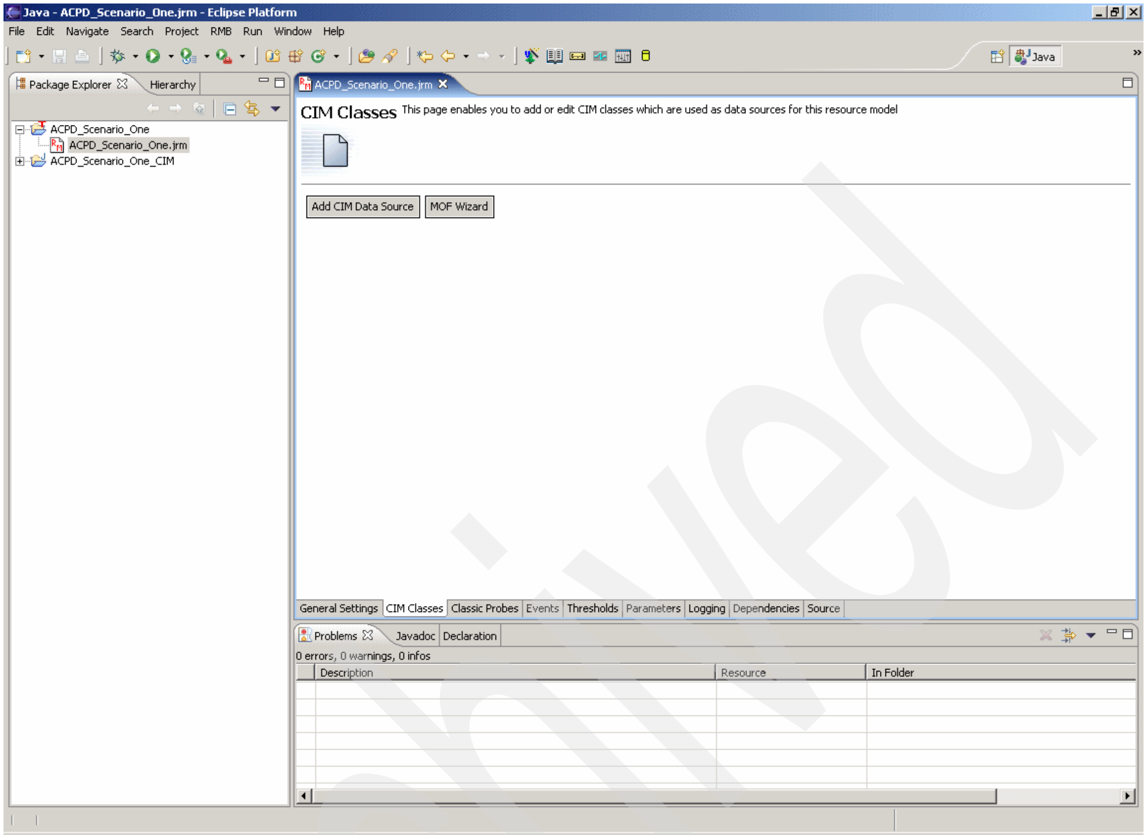


Figure 8-14 CIM Classes tab

2. From the MOF Wizard (shown in Figure 8-15 on page 285), you can compile a MOF file, check the syntax of a MOF file, and create a binary MOF file. Select **Compile a .MOF file** and click **Go!**.

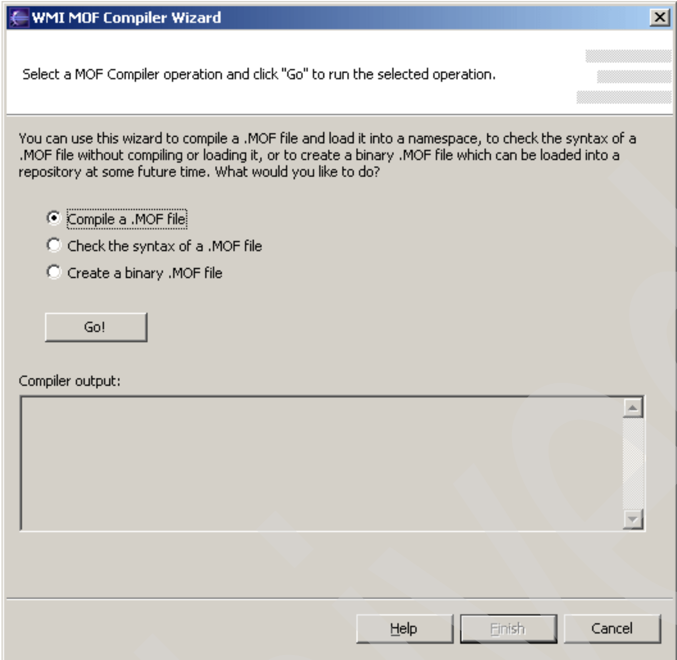


Figure 8-15 MOF Compiler Wizard

3. Now, specify the MOF file from the project. Under Namespace specify `root\cimv2`, and click **Finish**, as shown in Figure 8-16 on page 286.

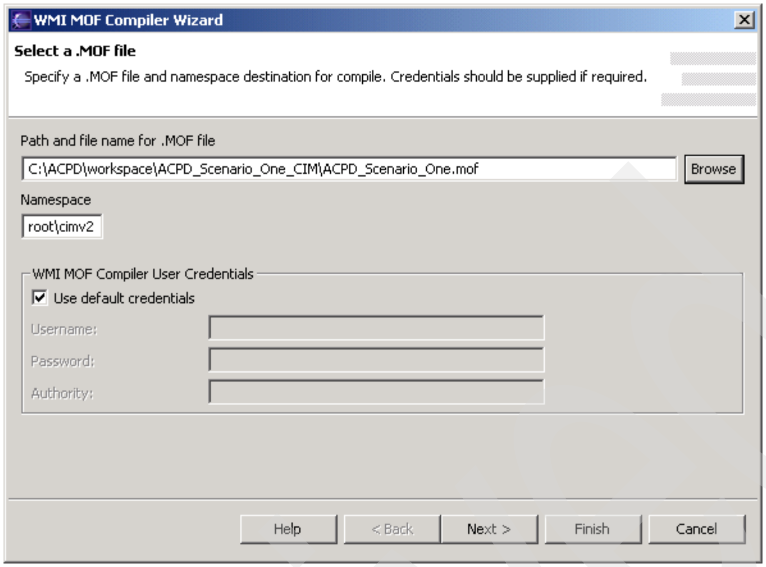


Figure 8-16 Compile a MOF file

The Compiler output field shows the result of the compilation, as shown in Figure 8-17 on page 287. If an error occurs, it is shown with an indication about the problem. Modify the MOF file and recompile it.

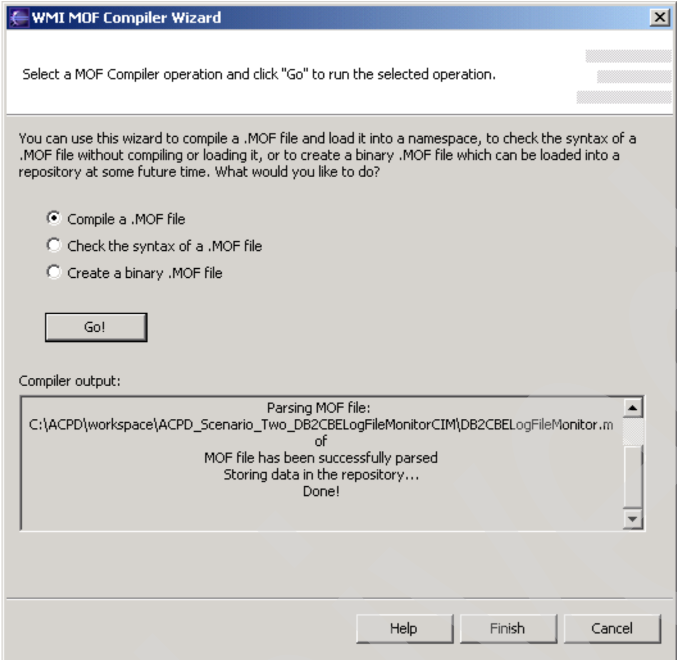


Figure 8-17 MOF compilation results

Including the CIM class attributes into the resource model

To use the CIM class as a data resource, it needs to be configured in the resource model. Perform the following steps:

1. On the CIM Classes tab (see Figure 8-14 on page 284), click **Add CIM Data Source**, which shows the CIM repository.
2. Select the proper class, in our case, **DiskSpace_Monitor**, as shown in Figure 8-18 on page 288, and click **OK**.

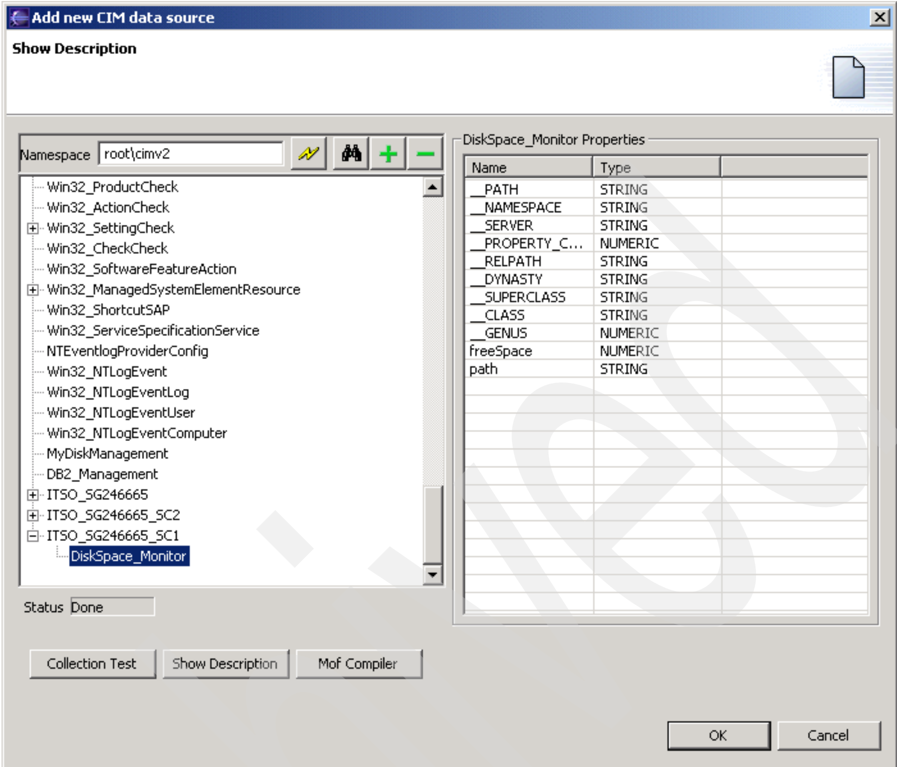


Figure 8-18 CIM repository

This adds `DiskSpace_Monitor` to the CIM Classes tab, as shown in Figure 8-19 on page 289.

Although it is possible to change the alias information using the field **Use Alias**, we do not recommend this. The alias will be used in the decision algorithm and needs to match the CIM class name. If a method in the CIM class will be used in an action on an event, the alias has to be specified to the same name as the name of the CIM class.

> **Restriction**: In the *Filtering* section, you can specify a filter in Windows Management Instrumentation Query Language (WQL). This is only supported for resource models running on Windows platforms.

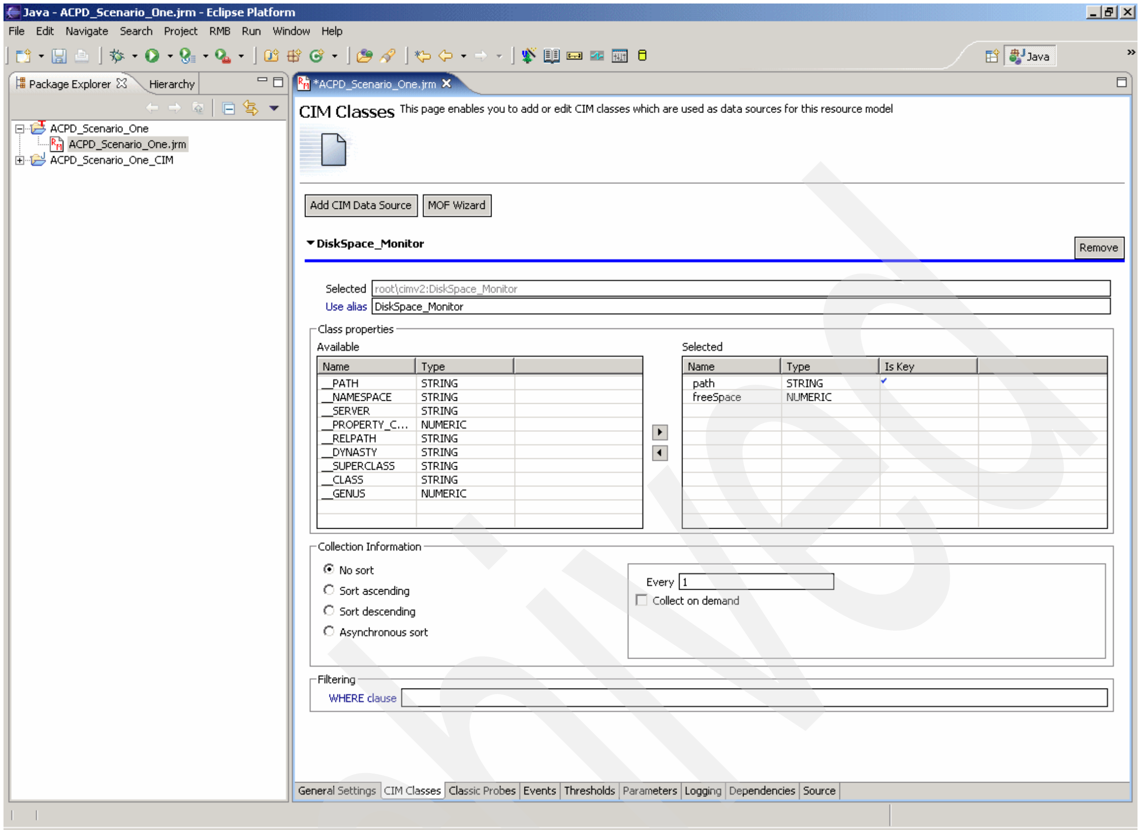


Figure 8-19 CIM data source

Defining events

The Events tab defines how the Autonomic Management Engine will handle events used by the resource model. An event has several options that need to be configured, and it is possible to configure corrective actions for the event. Figure 8-20 on page 290 shows the Events tab with one event. You can add events by clicking **Add Event**.

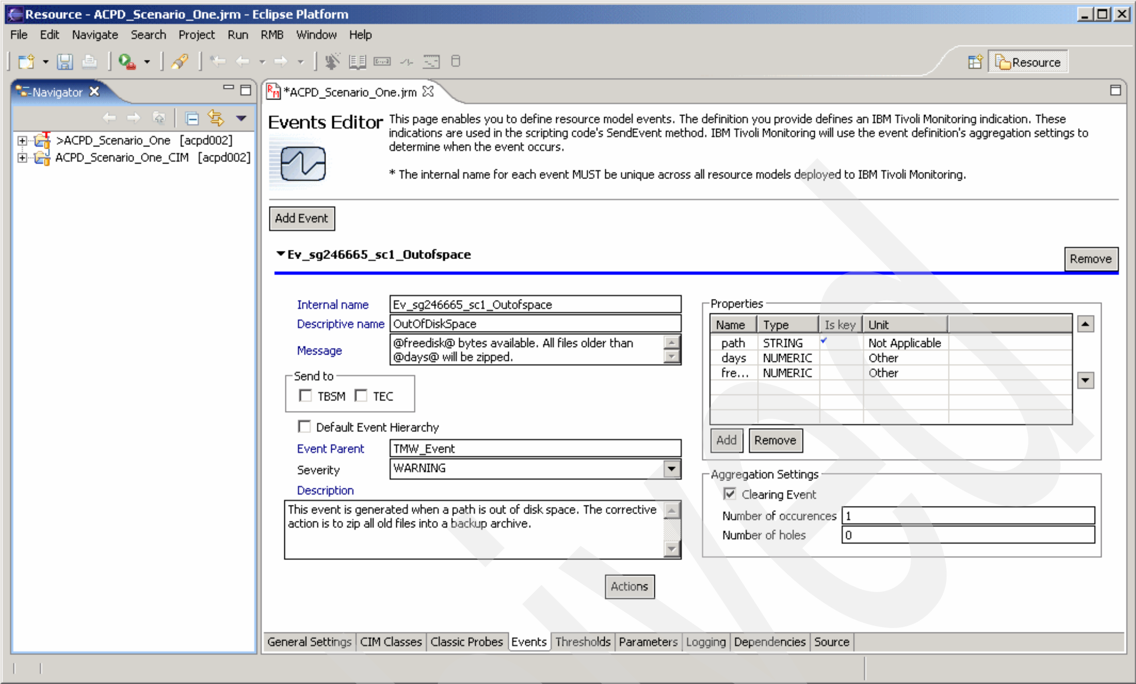


Figure 8-20 Events tab

Corrective actions can be added to each event by clicking the **Actions** button, which opens the Event Actions window, as shown in Figure 8-21. Here, you can add CIM methods, programs, and shell commands.

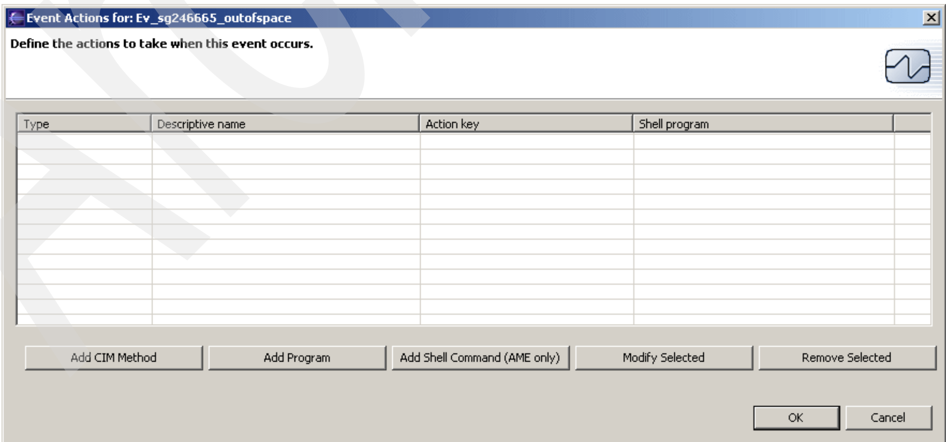


Figure 8-21 Event Actions window

To add a CIM method, click **Add CIM Method** and select the CIM class and method. For each key and parameter, select an Event attribute from the drop-down list, as shown in Figure 8-22. In addition, a Descriptive name and a Description must be added. When done, click **OK**.

> **Important:** Although it is possible to select a CIM method without specifying the Descriptive name and Description, it is required. It is *not* possible to generate a Autonomic Management Engine package without the Descriptive name and Description.

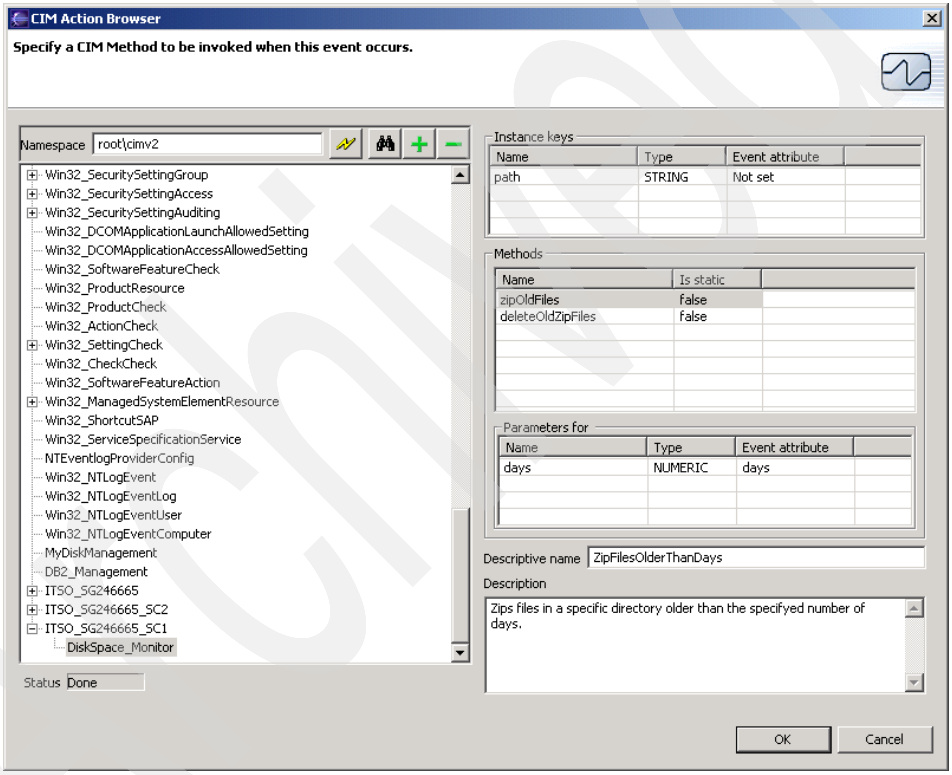


Figure 8-22 Add CIM Method

Table 8-1 on page 292 and Table 8-2 on page 293 provide the events defined for this example scenario. They are:

- Ev_sg246665_sc1_Outofspace

  This event will be triggered to monitor the hard disk space is going out of run with the threshold parameter defined and execute the utility class member when disk space goes under threshold parameter limit. Refer to Table 8-1 for more details.

Table 8-1 First event

Field	**Value**
Internal name	`Ev_sg246665_sc1_Outofspace`
Descriptive name	`OutOfDiskSpace`
Message	`Running low on diskspace in @path@. Only @freedisk@ bytes available. All files older than @days@ will be zipped.`
Default Event Hierarchy	Selected
Severity	`WARNING`
Description	`This event is generated when a path is out of disk space. The corrective action is to zip all old files into a backup archive.`
Properties	Name: `path`, Type: `STRING`, Key: yes Name: `days`, Type: `NUMERIC`, Key: no Name: `freedisk`, Type: `NUMERIC`, Key: no
Clearing Event	Selected
Number of occurrences	`5`
Number of holes	`0`
Action (CMI method)	
Method	`ITSO_SG246665` → `DiskSpace_Monitor.zipOldFiles`
Key (path)	`path`
Parameter (days)	`days`
Descriptive name	`ZipFilesOlderThanDays`
Description	`Zips files in a specific directory older than the specified number of days.`

- Ev_sg246665_sc1_Outofspace_server

  This event will be triggered to delete the archive files when the hard disk space goes below the threshold parameter. Refer to Table 8-2 for more details.

Table 8-2 Second event

Field	**Value**
Internal name	`Ev_sg246665_sc1_Outofspace_server`
Descriptive name	`OutOfDiskSpaceSevere`
Message	`Seriously running low on diskspace in @path@. Only @freedisk@ bytes available. All zip archives older than @days@ will be deleted.`
Default Event Hierarchy	Selected
Severity	`CRITICAL`
Description	`This event is generated when a path is out of disk space and zipping old files does not leave enough space. The corrective action is to delete old zip archives.`
Properties	Name: `path`, Type: `STRING`, Key: yes Name: `days`, Type: `NUMERIC`, Key: no Name: `freedisk`, Type: `NUMERIC`, Key: no
Clearing Event	Selected
Number of occurrences	`3`
Number of holes	`1`
Action (CMI method)	
Method	`ITSO_SG246665` → `DiskSpace_Monitor.deleteOldZipFiles`
Key (path)	`path`
Parameter (days)	`days`
Descriptive name	`DeleteZipFilesOlderThanDays`
Description	`Delete zips files in a specific directory older than the specified number of days.`

Defining thresholds

The Thresholds tab defines how to define thresholds for the Autonomic Management Engine using the resource model. Figure 8-23 shows the Thresholds tab with the thresholds used in this scenario. You can add thresholds by clicking the **Add Threshold** button**.**

Two thresholds are defined for this scenario:

- Thr_free_space_zip

  The lower limit for when space should be freed by zipping files into archives

- Thr_free_space_delete

  The lower limit for when space should be freed by deleting old archives

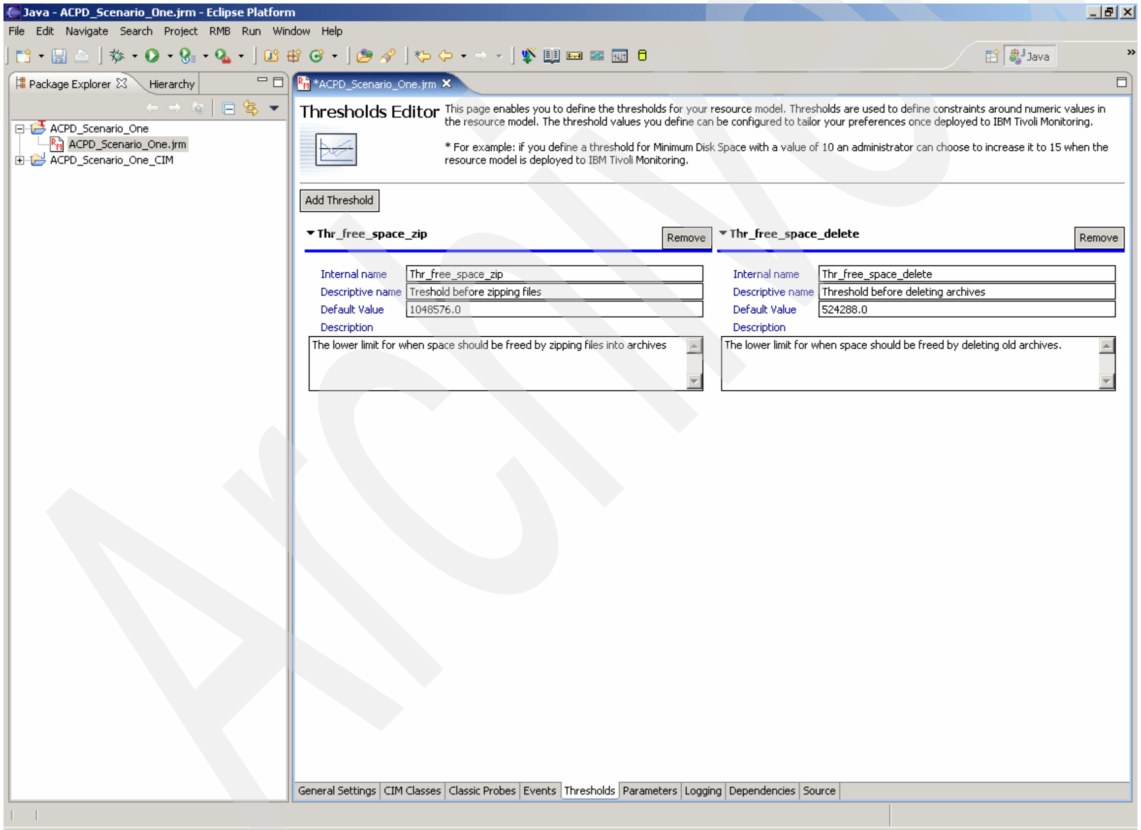


Figure 8-23 Threshold declaration

Defining parameters

Parameters are used for values that can be configured when the resource model is deployed. The parameter values can vary between resource model instances. The parameters can have multiple values. Figure 8-24 shows a parameter. To add a parameter, click the **Add Parameter** button.

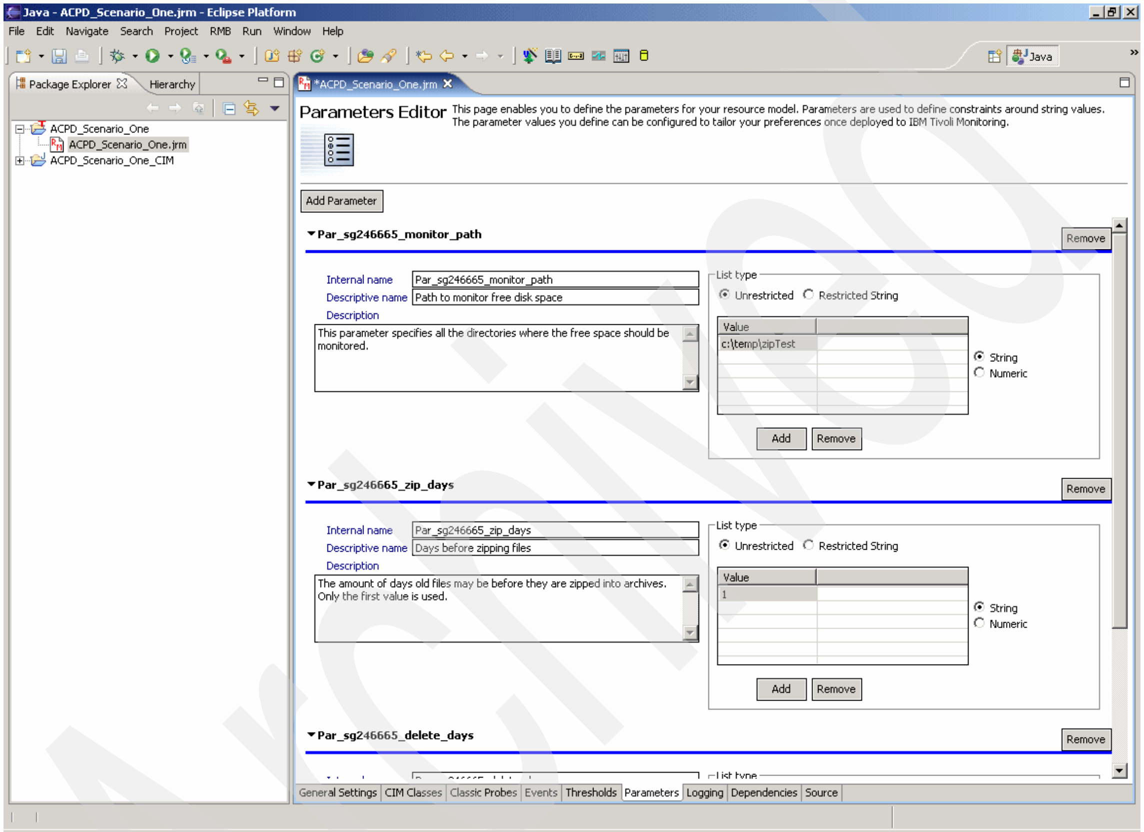


Figure 8-24 Parameters tab

We define the following three parameters for this scenario:

- Par_sg246665_monitor_path

  This parameter specifies all the directories where the free space should be monitored. Refer to Table 8-3 on page 296.

- Par_sg246665_zip_days

  This parameter specifies the number of days old files can exist before they are zipped into archives. Only the first value is used. Refer to Table 8-4 on page 296.

- Par_sg246665_delete_days

  This parameter specifies the number of days old archives can exist before they are deleted. Only the first value is used. Refer to Table 8-5.

The following tables provide the parameters defined for the scenario presented in this chapter.

Table 8-3 First parameter

Field	**Value**
Internal name	`Par_sg246665_monitor_path`
Descriptive name	`Path to monitor free disk space`
Description	`This parameter specifies all the directories where the free space should be monitored.`
List type (**Unrestricted** - **String**)	
Value	`C:\temp\zipTest`
Value	`S:\test`

Table 8-4 Second parameter

Field	**Value**
Internal name	`Par_sg246665_zip_days`
Descriptive name	`Days before zipping files`
Description	`The amount of days old files may be before they are zipped into archives. Only the first value is used.`
List type (**Unrestricted** - **Numeric**)	
Value	`1`

Table 8-5 Third parameter

Field	**Value**
Internal name	`Par_sg246665_delete_days`
Descriptive name	`Days before deleting archives`
Description	`The amount of days old archives may be before they are deleted. Only the first value is used.`

Field	Value
List type (**Unrestricted - Numeric**)	
Value	2

Defining dependencies

Dependencies are used to specify the files needed to execute the resource model. This includes JAR files with ILT classes, native libraries, and DLLs if needed.

There are two dependencies for this scenario:

- ACPD_Scenario_One.mof
- ACPD_Scenario_One_CIM.jar

Figure 8-25 shows the Dependencies tab.

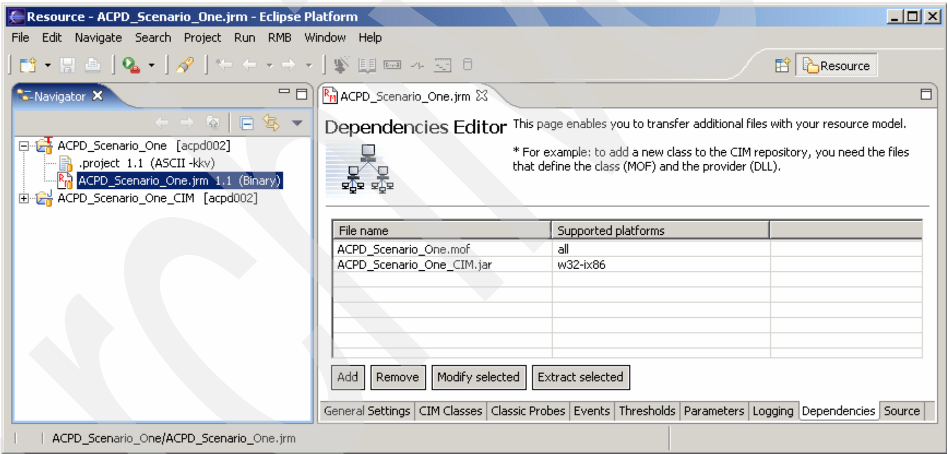


Figure 8-25 Dependencies tab

To add a file, click the **Add** button. In the window that opens, specify the File name and the Supported platforms. For example, specify the files `ACPD_Scenario_One_CIM.jar` and `ACPD_Scenario_One.mof` from the ACPD_Scenario_One_CIM project.

Important: Remember to include the MOF files for the data sources.

Important: When you add some files to the dependency list, they are imported; therefore, there are no links to the original file. When you generate the Autonomic Management Engine package, it will include the files as they were when they were added to the dependency list.

Creating the decision tree script

This section describes the decision tree script developed for the scenario presented in this chapter.

The following methods are defined in the decision tree script:

- SetDefaultConfiguration()
- Init()
- VisitTree()

SetDefaultConfiguration()

This method initializes the object based on the settings specified under the other tabs. The method SetDefaultConfiguration() is called only once, when the resource model starts. Additional initialization code can be written at the end of this function as needed. Example 8-10 shows the SetDefaultConfiguration function as it is generated by the Resource Model Builder based on the example implemented throughout this chapter.

Example 8-10 SetDefaultConfiguration function of the decision tree JavaScript

```
function SetDefaultConfiguration(Svc)
{
   Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start SetDefaultConfiguration");

   // General info section
   //<<GENERAL_INFO>>
   Svc.SetModelName ("ACPD_Scenario_One");
   Svc.SetProfileName ("1110403751344");
   Svc.SetCycleTime (60);
   //<<\GENERAL_INFO>>

   // Thresholds section
   //<<THRESHOLDS_INFO>>
   Svc.DefineThreshold ("Thr_free_space_zip", 1048576.0);
   Svc.DefineThreshold ("Thr_free_space_delete", 524288.0);
   //<<\THRESHOLDS_INFO>>

   // Parameters section
   //<<PARAMETERS_INFO>>
   Svc.DefineStrParameter ("Par_sg246665_monitor_path", "c:\temp\zipTest");
   Svc.DefineStrParameter ("Par_sg246665_zip_days", "1");
```

```
   Svc.DefineStrParameter ("Par_sg246665_delete_days", "2");
   //<<\PARAMETERS_INFO>>

   // Dynamic model section
   //<<DATA_INFO>>
   Svc.DefineClass ("CIM", "DiskSpace_Monitor",
"root\\cimv2:DiskSpace_Monitor", "", "freeSpace", "path", "None", "", 0, 1);
   //<<\DATA_INFO>>

   // Event definition section
   //<<EVENTS_INFO>>
   Svc.DefineEvent ("Ev_sg246665_sc1_Outofspace" , "days,freedisk", "path");
   Svc.DefineEvent ("Ev_sg246665_sc1_Outofspace_server" , "days",
"path,freedisk");
   //<<\EVENTS_INFO>>

   // Logging definition section
   //<<LOGGING_INFO>>
   //<<\LOGGING_INFO>>

   // Place your additional intializing code below.

   Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "End SetDefaultConfiguration");
   return (0);
}
```

Init()

The Init() method is used for initializing various parameters after the instance of the resource model is started. The requirements for these initialization activities are based on the decision logic used and the type of resources that are monitored. The Init() method is called once after a resource model is configured.

The Wizard generated Init() function will not contain any logic in it. The method has to be modified to initialize the classes and variables as needed, for example, initializing some variables depending on the operating system or associating parameters to CIM classes.

As shown in Example 8-11 on page 300, the Init() method is modified to perform the following task:

- Associate the parameter with the CIM class.
- Add logging statements appropriately for debugging purposes.

Example 8-11 Init function of the decision tree JavaScript

```
function Init(Svc)
{
  Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start Init");

  var numberOfParamsPath =
    Svc.GetStrParameterCount("Par_sg246665_monitor_path");
  if (numberOfParamsPath > 0) {
    for(i = 0; i< numberOfParamsPath; i++) {
      Svc.Trace(0,TRACE_SOURCE + "Parameter, path("+i+"): "+
        Svc.GetStrParameter("Par_sg246665_monitor_path",i));
    }
    // Associate the paths to monitor with the CIM class DiskSpcaeMonitor
    Svc.AssociateParameterToClass ("Par_sg246665_monitor_path",
      "DiskSpace_Monitor");
  }
  Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Done Init");
  return (0);
}
```

VisitTree()

The VisitTree() method implements the decision algorithm. It is executed one time in each cycle, after all the data has been collected from the CIM class properties. It is the responsibility of the VisitTree() method to analyze all the collected data and send indication events if a situation is found.

In Example 8-12 on page 301, the VisitTree() method is implemented to analyze free space and create events depending on the amount of free space. The implementation contains the following important sections:

1. Read the configured thresholds into variables.
2. Create a placeholder for event properties.
3. Loop over the number of instances created by the CIM class DiskSpace_Monitor.
4. Because all potential events contain the same event properties (`path` and `freedisk`), the event properties are set before comparing them with the thresholds.
5. Check if free space is below the threshold for zipping files. If it is, send the Ev_sg246665_outofspace event.
6. Check if free space is below the threshold for deleting zipped files. If it is, send the Ev_sg246665_outofspace_severe event.
7. Clean up by destroying the event property placeholder.

Example 8-12 VisitTree function of the ITSO_CBELog_Monitor resource model

```
function VisitTree(Svc)
{
  Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start VisitTree");

  // 1. Read thresholds
  var zipLimit = Svc.GetThreshold("Thr_free_space_zip");
  var deleteLimit = Svc.GetThreshold("Thr_free_space_delete");

  // 2. Placeholder for event properties
  var eventPropMap = Svc.CreateMap();

  // 3. loop over the number of instances of the DiskSpaceMonitor
  var numDSMInst = Svc.GetNumOfInst("DiskSpace_Monitor");
  for (idx = 0 ; idx < numDSMInst ; idx++) {
    var curMonPath = Svc.GetStrProperty("DiskSpace_Monitor",idx,"path");
    var availableSpace = Svc.GetNumProperty("DiskSpace_Monitor",
      idx,
      "freeSpace");

    // 4. Prepare properties for an event
    Svc.RemoveMapAll(eventPropMap);
    Svc.SetMapStrElement(eventPropMap,"path",curMonPath);
    Svc.SetMapNumElement(eventPropMap,"freedisk",availableSpace);

    // 5. Check for zip treshold
    if (availableSpace < zipLimit*1048576) {
      // Create an Ev_sg246665_outofspace event
      Svc.SetMapNumElement(eventPropMap,
        "days",
        Svc.GetNumParameter("Par_sg246665_zip_days",0));
      Svc.SendEventEx("Ev_sg246665_sc1_Outofspace",eventPropMap);
    }

    // 6. Check for delete treshold
    if (availableSpace < deleteLimit*1048576) {
      // Create an Ev_sg246665_outofspace_severe event
      Svc.SetMapNumElement(eventPropMap,
        "days",
        Svc.GetNumParameter("Par_sg246665_delete_days",0));
      Svc.SendEventEx("Ev_sg246665_sc1_Outofspace_server",eventPropMap);
    }
    Svc.Trace(TRACE_FINEST,
      TRACE_SOURCE +
        "Free Disk Space in "+
        curMonPath+
        " "+
        availableSpace+
```

```
        " bytes");
  }
  // 7. Cleanup
  Svc.DestroyMap(eventPropMap);
  Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Done VisitTree");
  return (0);
}
```

8.7.3 Creating the resource model package

This section describes how to generate a resource model package and the contents of the resource model package.

The package is a ZIP file containing the script, dependencies, and settings necessary to install the resource model.

To create the package, follow these simple steps:

1. Open the resource model (`MyStorage.jrm`).
2. From the menu, select **RMB** → **Generate Package** → **AME (zip)**.
3. In the File dialog box, specify the location and name of the package. For example, specify `MyStorage.zip` in the directory of `MyManagementProject`.

The different parts of the resource model can be generated manually from the RMB menu if needed.

The resource model package file consists of the following items, depending on the type of the resource model:

- MOF file
- Configuration files
- Resource model catalog files
- Resource model class files
- Dependencies, including:
  - CIM classes, which implement M12 providers
  - Native libraries, if any
  - Scripts for action, if applicable
- Platform-specific packages

The Autonomic Management Engine embedding environment unpacks this package for deployment. The platform-specific package, in turn, contains the specific files required for the deployment of the package on specific platforms.

8.8 Scenario execution

The files developed for this example scenario are available to download. Refer to Appendix A, “Additional material” on page 379 for instructions about how to obtain the files. This section explains how to execute the example scenario presented in this chapter.

8.8.1 Starting the scenario

To execute this scenario, perform the following steps:

1. After you have downloaded the scenario’s executable files, open a command prompt and move to the directory where you unzipped the scenario files.
2. Create a test directory to be monitored by the scenario's resource models, for example, c:\temp\zipTest:

   ```
   mkdir c:\temp\zipTest
   ```

3. Generate dummy files in this directory using the **execUseDiskSpace.bat** command, or issue the following command:

   ```
   java UseDiskSpace <TEST_DIRECTORY> 64 10
   ```

   Where <TEST_DIRECTORY> is the directory created in the previous step.

   Observe the UseDiskSpace program creating dummy files in the test directory.

4. Change to the directory in which AME has been installed, for example:

   ```
   cd c:\IBM\AutonomicComputingToolkit\AutonomicManagementEngine
   ```

5. Change to the SARA directory and start the SARA runtime environment:

   ```
   cd sara
   sara.bat
   ```

6. When the ready prompt appears, start the RME using the following command:

   ```
   startme
   ```

   RME started will be displayed.

7. Execute the following command to install the example scenario resource models:

   ```
   instrmtype “<SCENARIO_DIRECTORY>\ACPD_Scenario_One.zip” -replacefiles
   ```

   Where <SCENARIO_DIRECTORY> is the directory containing the scenario's downloadable files.

8. Make an instance of the scenario resource models using the following command:

   ```
   mkrminstance <INSTANCE_NAME> ACPD_Scenario_One
   ```

9. Start the resource model instance using the following command:

```
startrminstance <INSTANCE_NAME>
```

10. Observe the AME taking actions, cleaning up the test directory as a predefined disk utilization threshold is reached.

8.8.2 Stopping the scenario execution

To stop the execution of this scenario, perform the following steps:

1. Stop the resource model instance:

```
stoprminstance <INSTANCE_NAME>
```

2. Remove the scenario resource models using the following command:

```
rmrminstance <INSTANCE_NAME>
```

3. Uninstall the resource models from SARA:

```
uninstrmtype ACPD_Scenario_One
```

9

Problem Determination scenario two

This chapter describes how a multi-tiered application can take advantage of the self-healing capabilities of autonomic computing. The scenario described in this chapter shows a typical application that uses a database for data storage. Both the database and application run in independent servers, and the application accesses the database though a Java Database Connectivity (JDBC) connection. This scenario shows how self-managed autonomic technology and tools will be used to monitor the IT environment and perform corrective actions when autonomic managers detect a failure in one of the application's components.

This chapter describes the definition, implementation, package, and execution of this scenario and covers the following topics:

- Scenario description
- Resource model development and implementation
- Generic Log Adapter configuration files using multiple contexts, including custom outputter development and implementation
- Web services development and implementation

9.1 Scenario description

In a distributed environment, enterprise software applications are multi-tiered and their components run on multiple nodes. Very often, these applications establish database connections to insert, modify, and update database records.

In this chapter, we show an example scenario using a two-tier application. The sample Java application runs on the application node and accesses a database hosted by an IBM DB2 server running on the database node. The sample Java application generates Common Base Events (CBEs) natively and logs them into a log file. Figure 9-1 depicts the sample Java application used in this case study scenario chapter.

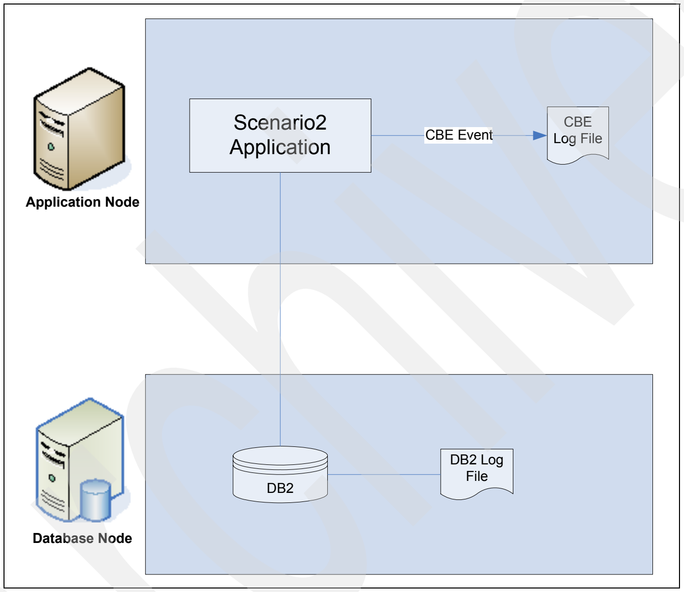


Figure 9-1 Application scenario environment

In order to make this application scenario autonomic, the following actions need to be performed:

- On the database node:
  - Develop a Generic Log Adapter (GLA) context to convert the log information created by DB2 into a CBE format. Common Base Events will be stored in a log file by the GLA outputter.
  - Develop resource models to monitor and take corrective actions based on events created by DB2 and translated by the GLA.

- Develop a second GLA context to convert log information created by DB2 into a CBE format. This GLA context's outputter will use Web services to send the Common Base Events to the application node. A custom outputter for this GLA context, named ITSO-SHIM outputter, will be also developed.

- On the application node:
  - Create resource models to monitor and take corrective actions based on events created by the sample Java application.
  - Develop a simple Web services application to receive Common Base Events from the second GLA context running on the database node. This Web services application, named ITSO-SHIM Web service, will be running on the application node on top of IBM WebSphere Application Server. The ITSO-SHIM Web service will receive CBEs from the second GLA context and store them into a log file in the application node.
  - Create resource models to monitor and take corrective actions based on created events stored in the ITSO-SHIM CBE log file.

Figure 9-2 on page 308 shows the application scenario environment with the autonomic computing elements in place.

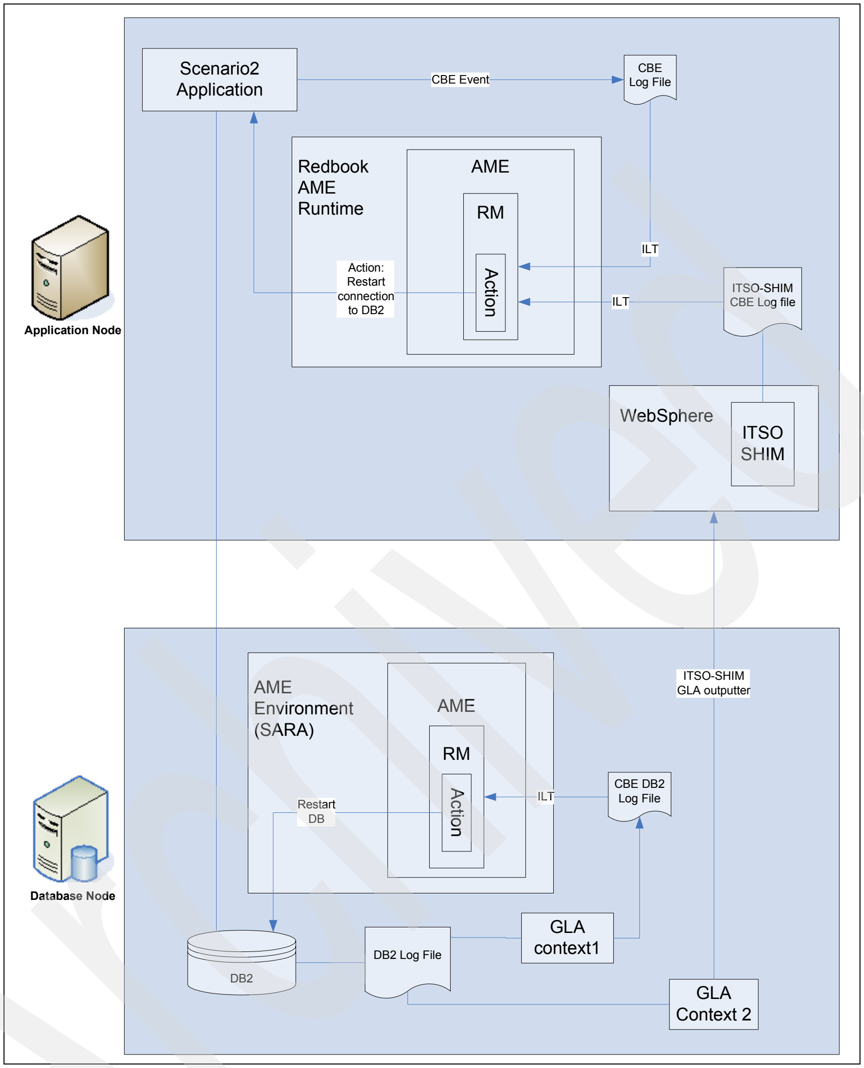


Figure 9-2 Autonomic application scenario environment

Using the IBM Autonomic Computing Toolkit and other development tools, we will create, develop, and configure self-managed autonomic technology. After all the components are developed and deployed on both the application and database nodes shown in Figure 9-2, the following sequence of events will be performed to demonstrate the automation and self-healing capabilities of the autonomic environment. The following steps refer to Figure 9-2:

1. All components of the application are operational, up, and running on their respective nodes.

2. An error is introduced in the environment. In this example scenario, we terminate the database instance execution so that the application will be unable to perform data insertion.
3. The application will log a Common Base Event into its log file reporting a connection failure to the database.
4. A resource model running on the autonomic manager deployed in the application node detects the incident reported by the application using ITL. As an initial step to correct the problem, the resource model signals the application to put data insertion on hold. The application puts data insertion tasks on hold and waits until the incident is resolved.
5. IBM DB2 logs an event in its log file stating that the database instance is down.
6. The GLA Context1, which monitors the IBM DB2 log file, converts the log information into Common Base Event format and stores them into a CBE DB2 log file.
7. A resource model running on the autonomic manager deployed in the database node monitors the CBE DB2 log file using ITL. The resource model analyzes the event reporting the database instance down and takes corrective action. The corrective action is to restart the database instance.
8. IBM DB2 logs an event in its log file stating that the database instance is operational.
9. The GLA Context2, which also monitors the IBM DB2 log file, converts the log information into Common Base Event format and uses a custom outputter to send the event to the application node using Web services. The custom outputter name is ITSO-SHIM outputter.
10. On the application node, a small application named ITSO-SHIM running on WebSphere is accessed as a Web service by the GLA Context2. The ITSO-SHIM Web service receives events from GLA Context2 and saves them into a log file.
11. A resource model running on the autonomic manager deployed in the application node monitors the ITSO-SHIM log file using ITL. The resource model analyzes the event reporting that the database instance is operational and takes corrective action. The corrective action is to restart the database connection to the application's database and communicate to the application that the database is back to an operational state.
12. The application resumes normal activity.

9.2 Scenario development and implementation

The development, implementation, and execution of this example scenario is divided into modules, as shown in Table 9-1.

Table 9-1 Example scenario modules

Modules	Content
Database node module one	Resource model that monitors the CBE DB2 log file using ITL. The resource model analyzes the events and takes action in case it finds a database instance down event. The action taken restarts the database instance.
Database node module two	Generic Log Adapter (GLA) context to convert log information created by DB2 into CBE format. Common Base Events will be stored into a CBE DB2 log file by the GLA outputter.
Database node module three	A second context for the GLA created in database node module two. This GLA context, which also monitors the IBM DB2 log file, converts the log information into Common Base Event format and uses a custom outputter to send the event to the application node using Web services.
Application node module one	Resource model that monitors the application’s CBE log file using ITL. This resource model analyzes the events and takes action in case it finds a connection problem to the application’s database. The action taken signals the application to put data insertion on hold.
Application node module two	A small application named ITSO-SHIM running on WebSphere. ITSO_SHIM is Web service that can receive events from the GLA and save them into a log file.

All the modules created in this chapter are available for download. Refer to Appendix A, “Additional material” on page 379 for details about how to obtain these modules.

In order to develop the self-managing autonomic capabilities of this scenario, the following IBM Autonomic Computing Toolkit Version 2.0 tools and technologies must be installed on both the application and database nodes:

- Autonomic Management Engine V1.1
- Resource Model Builder V1.3

- Generic Log Adapter Runtime and Rule Sets V3.0.0
- Log and Trace Analyzer V3.0.0
- Eclipse Tooling Package V3.0

Refer to Chapter 2, “IBM Autonomic Computing Toolkit overview” on page 25 for information about how to obtain these tools and technologies.

The following sections provide detailed information about each of the modules described in Table 9-1 on page 310.

9.2.1 Database node module one

This section provides details about the development and implementation of the components of database node module one. Figure 9-3 outlines the components that we discuss in this section.

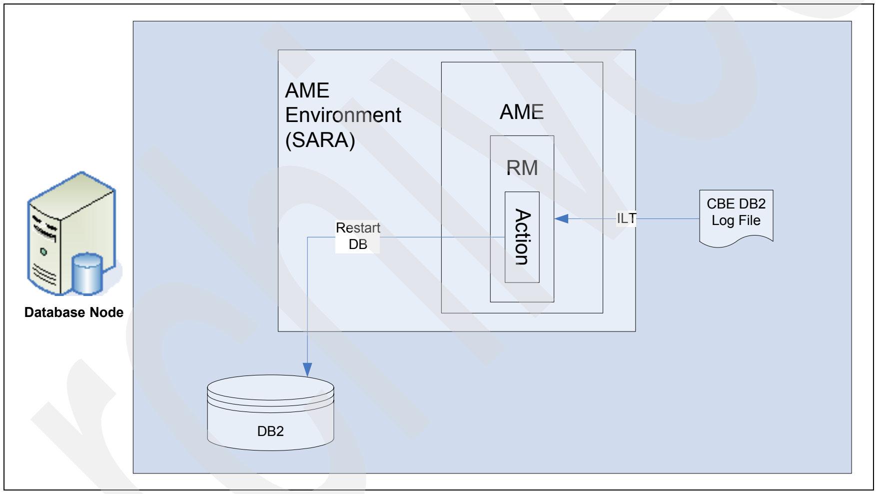


Figure 9-3 Database node module one

We describe the development of this module in the following topics:

- Defining the resource model
- Creating the MOF file
- Creating the CIM classes
- Implementing the DB2CBELogFileAdapterILT ILT class
- Implementing the DB2ResourceMonitorILT ILT class
- Implementing the utility class
- Creating the installation package

Defining the resource model

In this section, we create the resource model for this example scenario module. Refer to the 6.2, “Resource models” on page 199 for information about the resource model definition.

In 8.2, “Resource model definition” on page 260, we provide the sequence of steps required to define a resource model. We followed those same steps for creating the resource models using the following information:

Project name	`ACPD_SC2_DM1_RM`
Internal Name	`DB2ResourceManagement`
Descriptive Name	`DB2 Resource Management`
Description	`Monitors the DB2 CBE log messages. Analyzes the CBE log messages and generates an event to rectify the problem that was identified at the CBE log messages.`
Category Internal Name	`DB2ResourceModel`
Category Descriptive Name	`DB2 Resource Management`
Major Version	`1`
Minor Version	`0`
Cycle Time	`60`
Script Language Selection	**JavaScript**
Select the platforms	**w32-ix86**
DB2 Resource Model Name	`DB2ResourceManagement.jrm`

Creating the MOF file

In this section, we create the MOF file for this example scenario module. Refer to Refer to “Managed Object Format (MOF)” on page 209 and 7.2.6, “Structure of the MOF file” on page 253 for details.

The MOF file created is named DB2ResourceManagement.mof and is presented in Example 9-1 on page 313. The MOF file defines two main interface definitions:

- DB2CBELogFileMonitor

  This interface definition will be used to monitor the DB2 CBE log file.

- DB2ResourceMonitor

  This interface definition will be used to activate an event when there is match in the DB2 CBE log message.

Example 9-1   DB2ResourceManagement.mof

```
// The class ITSO_SG246665 is used to group all the custom CIM clases in
// the CIM repository
[
  description("Used to group all CIM classes used in the RedBook SG246665."),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
]
class ITSO_SG246665_SC2 {
};

[
  description("Read a CBE logfile. For each event create an 'instance' of
com.tivoli.wmftools.ilt.logfileadapter.LogfileAdapter"),
  M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT | %EVAL
'FileName=\"'@USERDATA_VALUE(parLogName)'\";' \"gnuRegExp='(<CommonBaseEvent
+.*?</CommonBaseEvent>)';\" 'cacheCount = 100;' | ENUM"),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
]
class DB2CBELogFileMonitor :ITSO_SG246665_SC2 {
  [key]
  string fileName;

  [key]
  uint32 offset;

  [key]
  string matchedContent;

  [
    description("The situation of the log event."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='<situation categoryName=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string situationCatName;

  [
    description("The severity of the log event."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='severity=\"([0-9]+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string severity;

  [
    description("The reasoningScope of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='reasoningScope=\"(.+?)\"';  SubExp=1; | GET"),
```

```
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string reasoningScope;

  [
    description("The successDisposition of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='successDisposition=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string successDisposition;

  [
    description("The situationQualifier of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='situationQualifier=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string situationQualifier;

  [
    description("The situationDisposition of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='situationDisposition=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string situationDisposition;

  [
    description("The reporter component application."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='<reporterComponentId.+?application=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string reporterComponentApplication;

  [
    description("The reporter component name."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2CBELogFileAdapterILT |
gnuRegExp='<sourceComponentId.+?component=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string sourceComponentName;
};

[
  description("CIM class for Database resource management. The CIM is implemented with a custom
ILT. The class contains a static method for restarting the database."),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider"),
```

```
  M12_Instrumentation("Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2ResourceManagementILT ||
ENUM")
]
class DB2ResourceMonitor :ITSO_SG246665_SC2 {
  [key]
  string dummy;

  [
    description("Restart the database"),
    M12_Instrumentation {"Java.com.ibm.itso.sg246665.sc2.db2.ilt.DB2ResourceManagementILT ||
INVOKE"},
    static
  ]
  string restartDB2Database();
};
```

Creating the CIM classes

In this section, we create the CIM class definitions for this example scenario module. Refer to the 6.2, "Resource models" on page 199 for information about CIM class definitions.

In 8.4, "Creating the Common Information Model classes" on page 266, we provide the sequence of steps required to define a CIM class. We followed those same steps for creating the following CIM classes for DB2 resource management.

Create two projects. The first project monitors the CBE log file messages. The second project takes action after there is a CBE log message.

DB2CBELogFileMonitorCIM project

This project defines the DB2CBELogFileMonitorILT class as follows:

Project Name	`DB2CBELogFileMonitorCIM`
Package Name	`com.ibm.itso.sg246665.sc2.db2.ilt`
ILT Class Name	`DB2CBELogFileAdapterILT`

DB2ResourceMonitorCIM project

This project defines the DB2ResourceMonitorILT class as follows:

Project Name	`DB2ResourceMonitorCIM`
Package Name	`com.ibm.itso.sg246665.sc2.db2.ilt`
ILT Class Name	`DB2ResourceMonitorILT`
Utility Class Name	`DB2ResourceMonitor`

Implementing the DB2CBELogFileAdapterILT ILT class

This ILT class will monitor the DB2 CBE log file, parse the DB2 CBE event, and store the event information in particular class attributes. The following member functions of ILT interface are implemented:

- DB2CBELogFileAdapterILT()
- enumerateInstances()
- parseAssignments()
- getParameterPart()
- getValuePart()
- getProperty()

DB2CBELogFileAdapterILT()

This constructor activates the trace entries for this ILT class.

Example 9-2 Implementation of DB2CBELogFileAdapterILT()

```
DB2CBELogFileAdapterILT() {
trace = TSLTraceImpl.getInstance();
trace.logMsg(TSLTrace.DEBUG_MAX, DB2CBELogFileAdapterILT.class.getName(),
"DB2CBELogFileAdapterILT", " Created. ");
}
```

enumerateInstances()

This member function implementation identifies the file name of the DB2 CBE log file, parses the DB2 CBE log file, and sets the attributes of the DB2CBELogFileAdapter class.

Example 9-3 Implementation of enumerateInstances()

```
public Enumeration enumerateInstances(
      M12ClassPath m12classpath,
      String mapstring,
      ParameterSet parameterset) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "enumerateInstances",
       new Object[] { m12classpath, mapstring, parameterset });

    Vector returnVector = new Vector();
    Properties properties = parseAssignments(mapstring);
    String fileNameList = properties.getProperty("FileName");

    String regExpStr = properties.getProperty("gnuRegExp");
    String cacheCountStr = properties.getProperty("cacheCount");
    int cacheCount = 100;
    if (cacheCountStr != null) {
```

```
  cacheCount = Integer.parseInt(cacheCountStr);
}
String classpath = m12classpath.getClassName();
String nameSpace = m12classpath.getNameSpace();

StringTokenizer fileNameTokenizer = new StringTokenizer(fileNameList,
    ",");

while (fileNameTokenizer.hasMoreTokens()) {
  String fileName = fileNameTokenizer.nextToken();
  File file = null;
  long oldFileLength = 0;
  long newFileLength = 0;
  file = new File(fileName);
  if (!file.isFile()) {
    trace.logMsg(TSLTrace.FATAL, this, "enumerateInstances",
        "{0} is not a file.", fileName);
    continue;
  }

  if (file == null || file.length() == 0) {
    logfiles_.remove(fileName);
    continue;
  } else {
    File logfile1 = (File) logfiles_.get(fileName);
    if (logfile1 == null) {
      logfile1 = file;
      logfiles_.put(fileName, logfile1);
      newFileLength = logfile1.length();
      lastLengthCache_.put(logfile1.getAbsolutePath(), new Long(
          newFileLength));
    } else {
      File file1 = new File(logfile1.getAbsolutePath());
      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
          "creating a temp file to look at file "
              + file1.getAbsolutePath() + ", with current length of "
              + file1.length());
      newFileLength = file1.length();

      Long oldLength = (Long) lastLengthCache_.put(file1
          .getAbsolutePath(), new Long(newFileLength));
      oldFileLength = oldLength.longValue();

      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
          "current length of cached logfile "
              + logfile1.getAbsolutePath() + " was "
              + logfile1.length());
      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
          "previous length of cached logfile "
```

```
            + logfile1.getAbsolutePath() + " was " + oldLength);
    if (!logfile1.equals(file)
        && file.lastModified() > logfile1.lastModified()
        || file1.length() < logfile1.length() || oldLength != null
        && file1.length() < oldLength.longValue()) {
      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
          file1.getAbsolutePath()
              + " has rolled - recreating logfile object.");
      logfiles_.remove(fileName);
      logfile1 = file1;
      logfiles_.put(fileName, logfile1);
      oldFileLength = 0;
      newFileLength = logfile1.length();
      lastLengthCache_.put(logfile1.getAbsolutePath(), new Long(
          newFileLength));
    }
  }
  if (oldFileLength == newFileLength) {
    continue;
  }
  RE classRE = new RE(regExpStr);

  char[] buffer = new char[1024];
  try {
    FileReader fileReader = new FileReader(logfile1);
    fileReader.skip(oldFileLength);
    buffer = new char[(int) (newFileLength - oldFileLength)];
    int charsRead = fileReader.read(buffer);

    String newEvents = new String(buffer, 0, charsRead);
    int eventpos = 0;
    int matchCount = 0;
    int count = 0;
    do {
      classRE.match(newEvents, eventpos);
      matchCount = classRE.getParenCount();
      if (matchCount > 0) {
        String matchedContent = classRE.getParen(0);
        M12PropertySet m12propertyset = new M12PropertySet();
        // must set all key properties. in this case only path is a key
        // property
        m12propertyset.setProperty("fileName", logfile1
            .getAbsolutePath());
        m12propertyset.setProperty("offset", Long
            .toString(oldFileLength + eventpos));
        m12propertyset.setProperty("matchedContent", matchedContent);
        M12IdentityElement m12identityelement = new M12IdentityElement(
            m12classpath.getClassName(), m12classpath.getNameSpace(),
            m12propertyset);
```

```
            returnVector.add(new M12ObjectIdentity(
                new M12IdentityElement[] { m12identityelement }));
            eventpos = classRE.getParenEnd(0);
            count++;
          }
        } while (matchCount > 0 && count <= cacheCount);
        if (oldFileLength + eventpos < newFileLength) {
          lastLengthCache_.put(logfile1.getAbsolutePath(), new Long(
              oldFileLength + eventpos));
        }
      } catch (FileNotFoundException e) {
        trace.logMsg(TSLTrace.FATAL, this, "enumerateInstances",
            "File not found:" + logfile1.getAbsolutePath() + " "
                + e.getMessage());
        continue;
      } catch (IOException e) {
        trace.logMsg(TSLTrace.FATAL, this, "enumerateInstances",
            "Error reding file: " + logfile1.getAbsolutePath() + " "
                + e.getMessage());
        continue;
      }
    }
  } // file name loop (while)

  Enumeration returnEnumeration = returnVector.elements();
  trace.exit(TSLTrace.DEBUG_MAX, this, "enumerateInstances",
      returnEnumeration);
  return returnEnumeration;
}
```

parseAssignments()

This member function parses the properties with the current position and returns the properties.

Example 9-4 Implementation of parseAssignments()

```
int ivCurrPos = 0;
    Properties assignments = new Properties();
    while (ivCurrPos < mappingString.length()) {
      String paramter = getParamterPart(mappingString, ivCurrPos);
      if (paramter == null || paramter.length() == 0) {
        return null;
      }

      ivCurrPos += paramter.length();
      if (mappingString.charAt(ivCurrPos) == '=') {
        ivCurrPos++;
      } else {
```

```
        return null;
      }
      String value = getValuePart(mappingString, ivCurrPos);
      ivCurrPos += value.length();
      if (value.startsWith("\"")) {
        value = value.substring(1, value.length() - 1);
      }
      if (value.startsWith("'")) {
        value = value.substring(1, value.length() - 1);
      }

      if (value == null || value.length() == 0) {
        return null;
      }
      assignments.setProperty(paramter.trim(), value.trim());

      if (ivCurrPos >= mappingString.length()) {
        break;
      }
      if (mappingString.charAt(ivCurrPos) == ';')
        ivCurrPos++;
      else {
        return null;
      }
    }
    return assignments;
```

getParameterPart()

This member function identifies the index of "=" at the i-th element of the mapping string.

Example 9-5 Implementation of getParameterPart()

```
private String getParamterPart(String mappingString, int i) {
    int nextEqual = mappingString.indexOf("=", i);

    if (nextEqual < 0) {
      return null;
    } else {
      return mappingString.substring(i, nextEqual);
    }
  }
```

getValuePart()

This member function provides the value part of the mapping string at the index identified by "i".

Example 9-6 Implementation of getValuePart()

```
private String getValuePart(String mappingString, int i) {
    boolean singleQvoteFlag = false;
    boolean doubleQvoteFlag = false;
    for (int j = 0; j < mappingString.length(); j++) {
      if (mappingString.charAt(i + j) == '\'')
        singleQvoteFlag = !singleQvoteFlag;
      if (mappingString.charAt(i + j) == '"')
        doubleQvoteFlag = !doubleQvoteFlag;
      if (mappingString.charAt(i + j) == ';'
          && !(singleQvoteFlag || doubleQvoteFlag)) {
        return mappingString.substring(i, i + j);
      }
    }

    if (!singleQvoteFlag && !doubleQvoteFlag)
      return mappingString.substring(i);
    else
      return null;
  }

  /
```

getProperty()

This member function provides the gnuRegExp from the matchedContent.

Example 9-7 Implementation of getProperty()

```
public String getProperty(
      M12ObjectIdentity objectIdentity,
      String propertyName,
      String mapstring,
      ParameterSet parSet) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "getProperty;", new Object[] {
        objectIdentity, propertyName, mapstring, parSet });

    Properties properties = parseAssignments(mapstring);
    String subExp = properties.getProperty("SubExp");
    int subExpNumber = Integer.parseInt(subExp);
    String regExpStr = properties.getProperty("gnuRegExp");

    // First verify that it is a supported property
    M12IdentityElement identityElement = objectIdentity.getScopingPath()[0];
    M12PropertySet propertySet = identityElement.getIdentity();
    // Get the path which is a key property.
    String fileName = propertySet.getProperty("fileName");
    long offSet = Long.parseLong(propertySet.getProperty("offset"));
    String matchedContent = propertySet.getProperty("matchedContent");
```

```
  RE propexp = new RE(regExpStr);
  String result = "";
  if (propexp.match(matchedContent)) {
    result = propexp.getParen(subExpNumber);
  }
  trace.exit(TSLTrace.DEBUG_MAX, this, "getProperty", result);
  return result;

}
```

Implementing the DB2ResourceMonitorILT ILT class

In this ILT class, the member function will be executed after there is a match found at the CBE and monitored by the DB2CBE LogFileAdapterILT. The following member functions are implemented in this Java class:

- DB2ResourceMonitorILT()
- enumerateInstances()
- invokeMethod()

DB2ResourceMonitorILT()

This constructor activates the trace entries for this ILT class.

Example 9-8 Implementation of DB2ResourceMonitorILT()

```
public DB2ResourceMonitorILT() {
    trace = TSLTraceImpl.getInstance();
    trace.logMsg(TSLTrace.DEBUG_MAX, DB2ResourceMonitorILT.class.getName(),
        "DB2ResourceMonitorILT", " Created. ");
  }
```

enumerateInstances()

This member function initializes the vector.

Example 9-9 Implementation of enumerateInstances()

```
public Enumeration enumerateInstances(
      M12ClassPath m12classpath,
      String mapping,
      ParameterSet parSet) {
    return new Vector(0).elements();
  }
```

invokeMethod()

This member function identifies the supported function and activates the utility class member function.

Example 9-10 Implementation of invokeMethod()

```
public String invokeMethod(
      M12ClassPath classPath,
      String method,
      String mapping,
      ParameterSet params,
      ParameterSet inParams,
      ParameterSet outParams) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "invokeMethod", new Object[] {
        classPath, method, mapping, params, inParams, outParams });

    // Verify that it is a supported method
    // Only restartDB is supported
    if (method.equals("restartDB")) {
      System.out.println("Resetting connection to database server ");
      boolean result = DB2ResourceMonitor.restartDB();
      if(result)
         System.out.println("Restarting database server is success ");
      else
         System.out.println("Restarting database server failure");
   }
    trace.exit(TSLTrace.DEBUG_MAX, this, "invokeMethod");
    return "restart Database was successful";
  }
```

Implementing the utility class

This section describes the DB2ResourceMonitor utility class. This utility class will be reusable to execute it independently or from any other application. It executes the specific command and provides result to the ILT class. Implementation of this utility class requires the development of a Java class including the restartDB() method.

restartDB()

This method restarts the database instance, opening a command prompt.

Example 9-11 Implementation of restartDB()

```
public static void restartDB() {
 String db2State = null;
 try {
   Process p = Runtime.getRuntime().exec(new String[]
{"cmd","/C","",db2dir+"db2Start"});
```

```
   BufferedReader reader = new BufferedReader(new InputStreamReader(new
BufferedInputStream(p.getInputStream())));
   String line;
   while ((line = reader.readLine()) != null ) {
     if (line.startsWith("SQL")){
        db2State = line.trim();
        System.out.println("\nOutput : "+db2State);}
     }
   }catch(IOException e) {
 e.printStackTrace();
 }
}
```

Creating the installation package

At this point, we have created all the elements required for the resource model. Now, we need to generate the resource model package, which can be run in the Autonomic Management Engine environment. This section includes the information required to create a JAR file for the ILT classes and custom CIM configuration using Tivoli Resource Model Builder.

The following activities need to be performed:

1. Include the MOF file into the project.
2. Compile the MOF file.
3. Include the CIM class attributes into the resource model.
4. Define the events.
5. Define the thresholds.
6. Define the parameters.
7. Define the dependencies.
8. Create the decision tree script.
9. Create the resource model package.

Refer to 8.7, “Creating the installation package” on page 279 for details about each step.

The following sections provide the resource model configuration details for this module.

MOF file

First, specify the DB2ResourceManagement.mof file.

CIM classes

Next, specify both the DB2CBELogFileMonitorCIM and DB2ResourceMonitorCIM classes.

Events

Table 9-2 provides details for the event definitions for this example scenario module.

Table 9-2 DB2 resource model event definition

Field	Value
Internal name	`Ev_DatabaseRestart`
Descriptive name	`Database has to restart`
Message	`Database has to restart.`
Default Event Hierarchy	Selected
Severity	**CRITICAL**
Description	`Database has to restart.`
Properties	Name: `databaseServer`, Type: `STRING`, Key: yes
Clearing Event	Selected
Number of occurrences	`1`
Number of holes	`0`
Action (CMI method)	
Method	`ITSO_SG246665` → `DB2ResourceMonitor` → `restartDB`
Key (path)	`dummy`
Descriptive name	`Restart Database.`
Description	`Restart Database.`

Parameters

Table 9-3 provides details for the parameter definitions for this example scenario module.

Table 9-3 DB2 resource model parameter definition

Field	Value
Internal name	`parLogName`
Descriptive name	`CBE LogfileName`
Description	`CBE LogFileName.`

Field	Value
List type (**Unrestricted** - **String**)	
Value	Full path to the log file, for example, `C:\ACPD\Scenario2\DBModule1\logs\CBEDatabaseLogfile.log`

Defining dependencies

The following dependencies for the resource model are defined for module:

- DB2ResourceManagement.mof

  The MOF file
- DB2CBELogFileMonitorCIM.jar

  The DB2CBELogFileMonitorCIM CIM class package
- DB2ResourceMonitorCIM.jar

  The DB2ResourceMonitorCIM CIM class package

Decision tree script

The section provides the implementation part of the decision script of the resource model created for this module. It includes the following methods:

- Init()

  This function initializes the log file name for the database log file monitor. See Example 9-12.

Example 9-12 Init() function implementation

```
function Init(Svc)
{
   Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start Init");
   Svc.AssociateParameterToClass("parLogName","DB2CBELogFileMonitor");
       return(0);
}
```

- visitTree()

  This member function monitors the DB2LogFileMonitor attributes and makes a decision to generate an event if there is an event match. Example 9-13 on page 327 shows the implementation.

Example 9-13 visitTree() function implementation

```
function VisitTree(Svc)
{
   Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start VisitTree");

   //vars for data source: LogFileMonitor
   var numOfInstances;
   var idx;
   var situationCount = 0; // Count of the number of situations written this cycle

   var str_CBERecord;
   var situationCatName;

   var ParamCount;
   var ParamIdx;
   var Different;

   hPropTable = Svc.CreateMap();

   //triggering logic for data source: LogFileMonitor
   numOfInstances = Svc.GetNumOfInst("LogFileMonitor");
   for(idx = 0; idx < numOfInstances; idx++){

      Svc.RemoveMapAll(hPropTable);

      // read values from cbe
      severity = Svc.GetStrProperty("DB2CBELogFileMonitor", idx, "severity");
      situationCatName = Svc.GetStrProperty("DB2CBELogFileMonitor", idx, "situationCatName");
      reasoningScope = Svc.GetStrProperty("DB2CBELogFileMonitor", idx, "reasoningScope");
      situationQualifier = Svc.GetStrProperty("DB2CBELogFileMonitor", idx,
"situationQualifier");
      situationDisposition = Svc.GetStrProperty("DB2CBELogFileMonitor", idx,
"situationDisposition");
      successDisposition = Svc.GetStrProperty("DB2CBELogFileMonitor", idx,
"successDisposition");
      reporterComponentApplication = Svc.GetStrProperty("DB2CBELogFileMonitor", idx,
"reporterComponentApplication");
      sourceComponentName = Svc.GetStrProperty("DB2CBELogFileMonitor", idx,
"sourceComponentName");

      Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Values: severity='"+severity+"'
situationCatName='"+situationCatName+"' reasoningScope='"+reasoningScope+"'
situationDisposition='"+situationDisposition+"' situationQualifier='"+situationQualifier+"'
successDisposition='"+successDisposition+"'");
      if (severity == 10
         && sourceComponentName == "DB2_UDB"
         && situationCatName == "StopSituation"
         && reasoningScope == "INTERNAL"
```

```
        && successDisposition == "SUCCESSFUL"
        && situationQualifier == "STOP COMPLETED"
    ) {
        Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Database has been stopped.");
        Svc.SetMapStrElement(hPropTable,
            "databaseServer",
            "sourceComponentName");
        Svc.SendEventEx("Ev_DatabaseRestart",hPropTable);
        }

}
Svc.DestroyMap(hPropTable);

return (0);
```

Creating the resource model package

For the instructions about how to package the resource models, see 8.7.3, “Creating the resource model package” on page 302. We followed those steps to create a resource model package for this module. The package name is DB2ResourceManagement.zip.

This package file is provided with this book. For details about how to obtain this file, refer to Appendix A, “Additional material” on page 379.

9.2.2 Database node module two

This section provides details about the development and implementation of the components of database node module two. Figure 9-4 outlines the components that we discuss in this section.

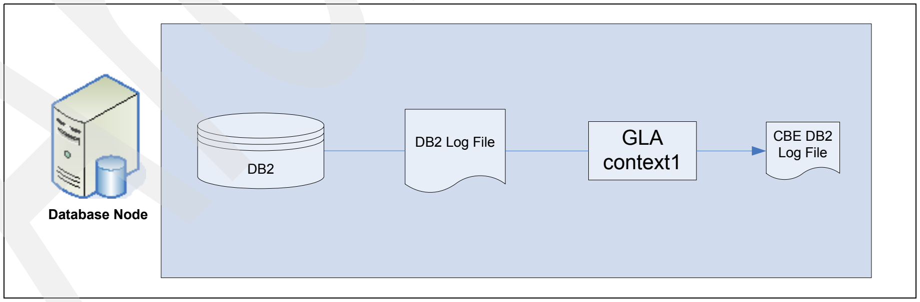


Figure 9-4 Database node module two

This module handles the conversion of events logged by IBM DB2 in the db2diag.log file into the Common Base Events format and stores the events in a CBE DB2 log file. Figure 9-4 illustrates the functionality of this module. The GLA

Context1 adapter in this module is configured with the sensor to read the db2diag.log file and the outputter to write into a file.

GLA adapter file development

For detailed instructions about how to create an Generic Log Adapter (GLA) adapter file, see 4.2, “Creating a simple adapter” on page 83.

A Generic Log Adapter named db2parser.adapter is created using a GLA adapter provided with the Generic Log Adapter tool. We used this file to take advantage of the parsing rules being already defined properly for the specific file we are interested in parsing (db2diag.log). All sample adapters provided by the Generic Log Adapter tool are in the <AC_TOOLKIT_INST>\GenericLogAdapter\config\ directory, where <AC_TOOLKIT_INST> is the installation directory of the IBM Autonomic Computing Toolkit V2.0.

The adapter fie used as the base for the adapter created in this section is <AC_TOOLKIT_INST>\GenericLogAdapter\config\DB2\diag\v8.2\regex.adatper.

The db2parser.adapter converts IBM DB2 events written into the db2diag.log file. This Generic Log Adapter parses the events in db2diag.log, converts them into Common Base Events, and writes to a file named CBEDatabaseLogfile. The adapter is developed using the following context information:

Sensor	SingleFileSensor type with executable class org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor
Outputter	SingleFileOutputter type with executable class org.eclipse.hyades.logging.adapter.outputters.CBEConvergentFileOutputter

For the continuous monitoring of the db2diag.log file, make sure that the **Continuous Operation** option in the Context Instance is selected. Increase the Maximum Idle Time in the Context Instance to a higher value to avoid disconnecting the GLA Runtime if there are no new events logged into db2diag.log for a certain period of time.

Figure 9-5 on page 330 shows the Context Instance settings that we used for this context. The Pause Interval option in the Context Instance is the time the context waits after reaching the end of the log file before trying to read it again.

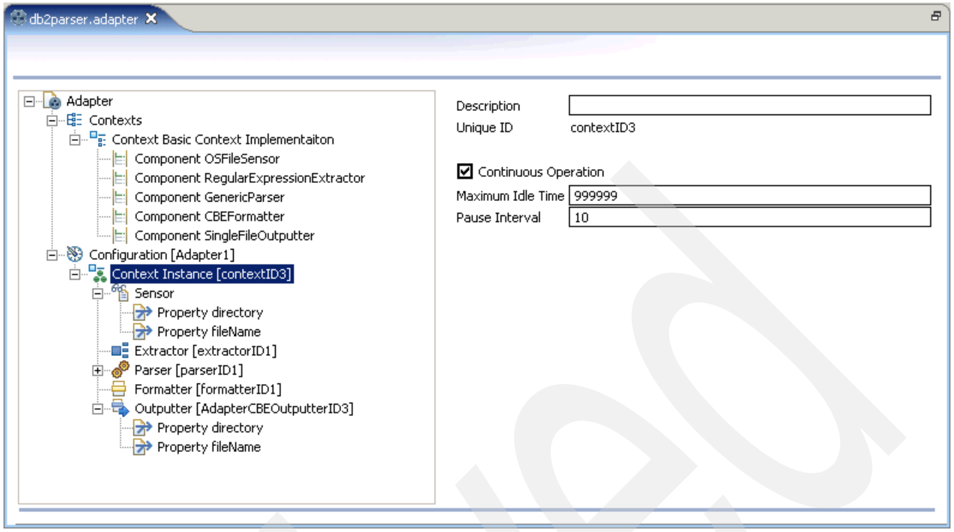


Figure 9-5 Context Instance setting for the scenario

This GLA adapter file is provided with this book. For details about how to obtain the file, refer to Appendix A, “Additional material” on page 379.

9.2.3 Database node module three

This section provides details about the development and implementation of the components of database node module three. Figure 9-6 on page 331 outlines the components that we discuss in this section.

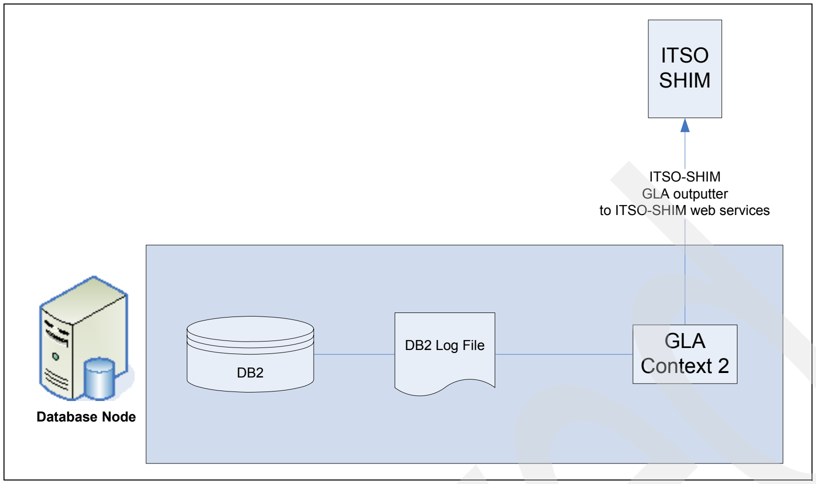


Figure 9-6 Database node module three

This module handles the conversion of events logged in the db2diag.log file into Common Base Events and sends the events to the application node using a Web services component named ITSOSHIM. Figure 9-6 illustrates the functionality of this module. The GLA Context2 adapter in this module is configured with the sensor to read the db2diag.log file and a custom outputter to write the Common Base Events to the ITSOSHIM Web services.

GLA adapter with custom outputter development

For detailed instructions about how to create an GLA adapter file, see 4.2, "Creating a simple adapter" on page 83.

The ShimOutputter.adapter is created for parsing the db2diag.log file into Common Base Events and writing to the Web service component named ITSOSHIM, using a custom outputter named ITSOSHIMOutputter.

The adapter is developed using the following context information:

Sensor	SingleFileSensor type with executable class org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor
Outputter	Undeclared type with executable class com.ibm.itso.sg246665.glarunner.SHIMOutputter

Section 4.5, "Writing a custom outputter" on page 124 contains the implementation of the com.ibm.itso.sg246665.glarunner.SHIMOutputter executable class. It also provides the detailed steps to create the ShimOutputter.adapter file.

A complete listing of the Java class used for the custom outputter is provided in Example 4-4 on page 131 and Example 4-5 on page 132.

For the continuous monitoring of the db2diag.log file, make sure that the **Continuous Operation** option in the Context Instance is selected. Increase the Maximum Idle Time in the Context Instance to a higher value to avoid disconnecting the GLA Runtime if there is no new events logged into db2diag.log for a certain period of time.

This GLA adapter file is provided with this book. For details about how to obtain the file, refer to Appendix A, “Additional material” on page 379.

9.2.4 Application node module one

This section provides details about the development and implementation of the components of application node module one. Figure 9-7 outlines the components that we discuss in this section.

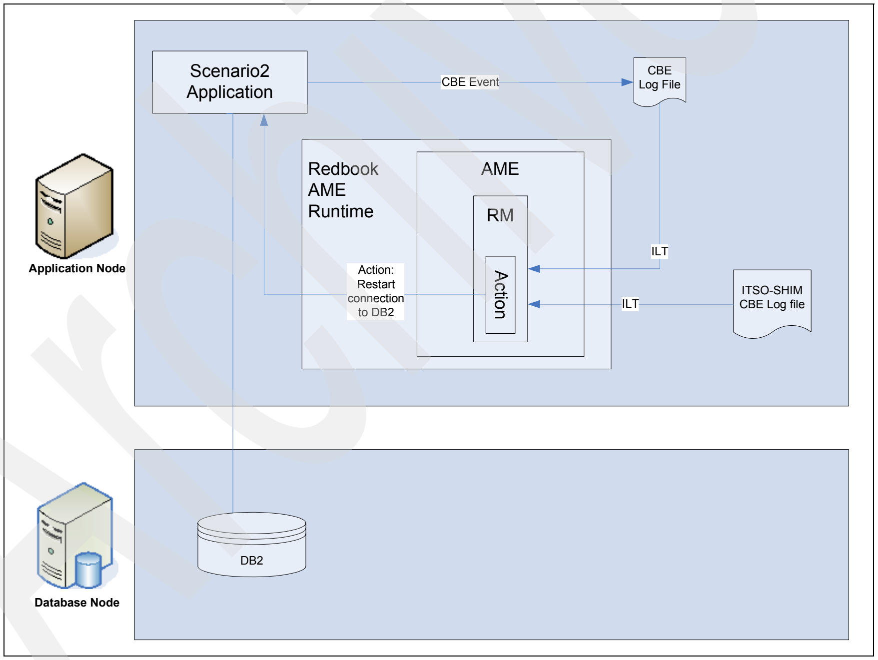


Figure 9-7 Application node module one

We discuss the development of this module in the following topics:

- Defining the resource model
- Creating the MOF File

- Creating the CIM classes
- Implementing the LogFileAdapterILT ILT class
- Implementing the ConnectResetILT ILT class
- Creating the installation package

Defining the resource model

In this section, we create the resource model for this example scenario module. Refer to 6.2, “Resource models” on page 199 for more information about the resource model definition.

In 8.2, “Resource model definition” on page 260, we provide the sequence of steps required to define a resource model. We followed those same steps for creating the resource models using the following information:

Project name	`ACPD_SC2_AM1_DatabaseConnectionRM`
Internal Name	`DatabaseConnectionRM`
Descriptive Name	`LogFileMonitor`
Description	`This Resource Model monitors the LogFileMonitor availability. It resets the connection to the database when necessary.`
Category Internal Name	`Scenario`
Category Descriptive Name	`Scenario`
Major Version	`1`
Minor Version	`0`
Cycle Time	`60`
Scripting Language Selection	**JavaScript**
Select the platforms	**w32-ix86**
DB2 Resource Model Name	`DatabaseConnection.jrm`

Creating the MOF File

In this section, we create the MOF file for this example scenario module. Refer to “Managed Object Format (MOF)” on page 209 and 7.2.6, “Structure of the MOF file” on page 253 for details.

The MOF file we create is named DatabaseConnectionRM.mof and is presented in Example 9-14. The MOF file defines two main interface definitions:

- LogFileMonitor

  This interface definition will be used to monitor the log file.

- ConnReset

  This interface definition will be used to reset the connection to the application's database.

Example 9-14 DatabaseConnectionRM.mof

```
// The class ITSO_SG246665 is used to group all the custom CIM classes in
// the CIM repository
[
  description("Used to group all CIM classes used in the RedBook SG246665."),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
]
class ITSO_SG246665_SC2 {
};

[
  description("Read a CBE logfile. For each event create an 'instance' of
com.tivoli.wmftools.ilt.logfileadapter.LogfileAdapter"),
  M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT | %EVAL
'FileName=\"'@USERDATA_VALUE(parLogName)'\";' \"gnuRegExp='(<CommonBaseEvent
+.*?</CommonBaseEvent>)';\" 'cacheCount = 100;' | ENUM"),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
]
class LogFileMonitor :ITSO_SG246665_SC2 {
  [key]
  string fileName;

  [key]
  uint32 offset;

  [key]
  string matchedContent;

  [
    description("The situation of the log event."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='<situation categoryName=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string situationCatName;

  [
    description("The severity of the log event."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='severity=\"([0-9]+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string severity;
```

```
  [
    description("The reasoningScope of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='reasoningScope=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string reasoningScope;

  [
    description("The successDisposition of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='successDisposition=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string successDisposition;

  [
    description("The situationQualifier of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='situationQualifier=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string situationQualifier;

  [
    description("The situationDisposition of the situation."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='situationDisposition=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string situationDisposition;

  [
    description("The reporter component application."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='<reporterComponentId.+?application=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string reporterComponentApplication;

  [
    description("The reporter component name."),
    M12_Instrumentation("Java.com.ibm.itso.sg246665.sceanrio.ilt.LogFileAdapterILT |
gnuRegExp='<sourceComponentId.+?component=\"(.+?)\"';  SubExp=1; | GET"),
    Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider")
  ]
  string sourceComponentName;
};
```

```
[
  description("CIM class for resetting a Database Connection for the application. The CIM is
implemented with a custom ILT. The class contains a static method for resetting the
connection."),
  Provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider"),
  M12_Instrumentation("Java.com.ibm.itso.sg246665.scenario.sc2.ilt.ConnectResetILT || ENUM")
]
class ConnReset :ITSO_SG246665_SC2 {
  [key]
  string dummy;

  [
    description("Reset the connection to a database"),
    M12_Instrumentation {"Java.com.ibm.itso.sg246665.scenario.sc2.ilt.ConnectResetILT ||
INVOKE"},
   static
  ]
  string resetConnection();
};
```

Creating the CIM classes

In this section, we create the CIM class definitions for this example scenario module. Refer to the 6.2, "Resource models" on page 199 for information about CIM class definitions.

In 8.4, "Creating the Common Information Model classes" on page 266, we provide the sequence of steps required to define a CIM class. We followed those same steps for creating the following CIM classes for DB2 resource management.

Create two projects. The first project monitors the CBE log file messages. The second project takes action after there is a CBE log message.

LogFileMonitor CIM project

This project defines the LogFileAdapterILT class as follows:

Project Name	`ACPD_SC2_AM1_LogFileMonitorCIM`
Package Name	`com.ibm.itso.sg246665.scenario.ilt`
ILT Class Name	`LogFileAdapterILT`

ConnReset CIM project

This project defines the ConnectResetILT class as follows:

Project Name	`ACPD_SC2_AM1_ConnResetCIM`
Package Name	`com.ibm.itso.sg246665.scenario.sc2.ilt`

ILT Class Name `ConnectResetILT`

Implementing the LogFileAdapterILT ILT class

This ILT class will monitor the ITSO-SHIM CBE log file, parse the CBE event, and store the event information in particular class attributes. In this scenario, the ITSO-SHIM log file will be created by the ITSO SHIM Web services. The content of the ITSH-SHIM log file will be identical to the content of the DB2 CBE log file produced by the GLA context running on the database node.

The following member functions of the ILT interface are implemented:

- LogFileAdapterILT()
- enumerateInstances()
- parseAssignments()
- getParameterPart()
- getValuePart()
- getProperty()

LogFileAdapterILT()

This constructor activates the trace entries for this ILT class.

Example 9-15 Implementation of LogFileAdapterILT()

```
public LogFileAdapterILT() {
    trace = TSLTraceImpl.getInstance();
    trace.logMsg(TSLTrace.DEBUG_MAX, LogFileAdapterILT.class.getName(),
        "ConnectResetILT", " Created. ");
  }
```

enumerateInstances()

This member function implementation identifies the file name of the CBE log file and sets the attributes of the class.

Example 9-16 Implementation of enumerateInstances()

```
public Enumeration enumerateInstances(
      M12ClassPath m12classpath,
      String mapstring,
      ParameterSet parameterset) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "enumerateInstances",
        new Object[] { m12classpath, mapstring, parameterset });

    Vector returnVector = new Vector();
    Properties properties = parseAssignments(mapstring);
    String fileNameList = properties.getProperty("FileName");
```

```
String regExpStr = properties.getProperty("gnuRegExp");
String cacheCountStr = properties.getProperty("cacheCount");
int cacheCount = 100;
if (cacheCountStr != null) {
  cacheCount = Integer.parseInt(cacheCountStr);
}
String classpath = m12classpath.getClassName();
String nameSpace = m12classpath.getNameSpace();

StringTokenizer fileNameTokenizer = new StringTokenizer(fileNameList,
    ",");

while (fileNameTokenizer.hasMoreTokens()) {
  String fileName = fileNameTokenizer.nextToken();
  File file = null;
  long oldFileLength = 0;
  long newFileLength = 0;
  file = new File(fileName);
  if (!file.isFile()) {
    trace.logMsg(TSLTrace.FATAL, this, "enumerateInstances",
        "{0} is not a file.", fileName);
    continue;
  }

  if (file == null || file.length() == 0) {
    logfiles_.remove(fileName);
    continue;
  } else {
    File logfile1 = (File) logfiles_.get(fileName);
    if (logfile1 == null) {
      logfile1 = file;
      logfiles_.put(fileName, logfile1);
      newFileLength = logfile1.length();
      lastLengthCache_.put(logfile1.getAbsolutePath(), new Long(
          newFileLength));
    } else {
      File file1 = new File(logfile1.getAbsolutePath());
      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
          "creating a temp file to look at file "
              + file1.getAbsolutePath() + ", with current length of "
              + file1.length());
      newFileLength = file1.length();

      Long oldLength = (Long) lastLengthCache_.put(file1
          .getAbsolutePath(), new Long(newFileLength));
      oldFileLength = oldLength.longValue();

      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
```

```
        "current length of cached logfile "
            + logfile1.getAbsolutePath() + " was "
            + logfile1.length());
    trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
        "previous length of cached logfile "
            + logfile1.getAbsolutePath() + " was " + oldLength);
    if (!logfile1.equals(file)
        && file.lastModified() > logfile1.lastModified()
        || file1.length() < logfile1.length() || oldLength != null
        && file1.length() < oldLength.longValue()) {
      trace.logMsg(TSLTrace.DEBUG_MID, this, "enumerateInstances",
          file1.getAbsolutePath()
              + " has rolled - recreating logfile object.");
      logfiles_.remove(fileName);
      logfile1 = file1;
      logfiles_.put(fileName, logfile1);
      oldFileLength = 0;
      newFileLength = logfile1.length();
      lastLengthCache_.put(logfile1.getAbsolutePath(), new Long(
          newFileLength));
    }
  }
  if (oldFileLength == newFileLength) {
    continue;
  }
  RE classRE = new RE(regExpStr);

  char[] buffer = new char[1024];
  try {
    FileReader fileReader = new FileReader(logfile1);
    fileReader.skip(oldFileLength);
    buffer = new char[(int) (newFileLength - oldFileLength)];
    int charsRead = fileReader.read(buffer);

    String newEvents = new String(buffer, 0, charsRead);
    int eventpos = 0;
    int matchCount = 0;
    int count = 0;
    do {
      classRE.match(newEvents, eventpos);
      matchCount = classRE.getParenCount();
      if (matchCount > 0) {
        String matchedContent = classRE.getParen(0);
        M12PropertySet m12propertyset = new M12PropertySet();
        // must set all key properties. in this case only path is a key
        // property
        m12propertyset.setProperty("fileName", logfile1
            .getAbsolutePath());
        m12propertyset.setProperty("offset", Long
```

```
                .toString(oldFileLength + eventpos));
            m12propertyset.setProperty("matchedContent", matchedContent);
            M12IdentityElement m12identityelement = new M12IdentityElement(
                m12classpath.getClassName(), m12classpath.getNameSpace(),
                m12propertyset);
            returnVector.add(new M12ObjectIdentity(
                new M12IdentityElement[] { m12identityelement }));
            eventpos = classRE.getParenEnd(0);
            count++;
          }
        } while (matchCount > 0 && count <= cacheCount);
        if (oldFileLength + eventpos < newFileLength) {
          lastLengthCache_.put(logfile1.getAbsolutePath(), new Long(
              oldFileLength + eventpos));
        }
      } catch (FileNotFoundException e) {
        trace.logMsg(TSLTrace.FATAL, this, "enumerateInstances",
            "File not found:" + logfile1.getAbsolutePath() + " "
                + e.getMessage());
        continue;
      } catch (IOException e) {
        trace.logMsg(TSLTrace.FATAL, this, "enumerateInstances",
            "Error reding file: " + logfile1.getAbsolutePath() + " "
                + e.getMessage());
        continue;
      }
    }
  } // file name loop (while)

  Enumeration returnEnumeration = returnVector.elements();
  trace.exit(TSLTrace.DEBUG_MAX, this, "enumerateInstances",
      returnEnumeration);
  return returnEnumeration;
}
```

parseAssignments()

This member function parses the properties with the current position and returns the properties.

Example 9-17 Implementation of parseAssignments()

```
int ivCurrPos = 0;
    Properties assignments = new Properties();
    while (ivCurrPos < mappingString.length()) {
      String paramter = getParamterPart(mappingString, ivCurrPos);
      if (paramter == null || paramter.length() == 0) {
        return null;
      }
```

```
      ivCurrPos += paramter.length();
      if (mappingString.charAt(ivCurrPos) == '=') {
        ivCurrPos++;
      } else {
        return null;
      }
      String value = getValuePart(mappingString, ivCurrPos);
      ivCurrPos += value.length();
      if (value.startsWith("\"")) {
        value = value.substring(1, value.length() - 1);
      }
      if (value.startsWith("'")) {
        value = value.substring(1, value.length() - 1);
      }

      if (value == null || value.length() == 0) {
        return null;
      }
      assignments.setProperty(paramter.trim(), value.trim());

      if (ivCurrPos >= mappingString.length()) {
        break;
      }
      if (mappingString.charAt(ivCurrPos) == ';')
        ivCurrPos++;
      else {
        return null;
      }
    }
    return assignments;
```

getParameterPart()

This member function identifies the index of "=" at the i-th element of the mapping string.

Example 9-18 Implementation of getParameterPart()

```
private String getParamterPart(String mappingString, int i) {
    int nextEqual = mappingString.indexOf("=", i);

    if (nextEqual < 0) {
      return null;
    } else {
      return mappingString.substring(i, nextEqual);
    }
  }
```

getValuePart()

This member function provides the value part of the mapping string at the index identified by “i”.

Example 9-19 Implementation of getValuePart()

```
private String getValuePart(String mappingString, int i) {
    boolean singleQvoteFlag = false;
    boolean doubleQvoteFlag = false;
    for (int j = 0; j < mappingString.length(); j++) {
      if (mappingString.charAt(i + j) == '\'')
        singleQvoteFlag = !singleQvoteFlag;
      if (mappingString.charAt(i + j) == '"')
        doubleQvoteFlag = !doubleQvoteFlag;
      if (mappingString.charAt(i + j) == ';'
          && !(singleQvoteFlag || doubleQvoteFlag)) {
        return mappingString.substring(i, i + j);
      }
    }

    if (!singleQvoteFlag && !doubleQvoteFlag)
      return mappingString.substring(i);
    else
      return null;
  }

  /
```

getProperty()

This member function provides the gnuRegExp from the matchedContent.

Example 9-20 Implementation of getProperty()

```
public String getProperty(
      M12ObjectIdentity objectIdentity,
      String propertyName,
      String mapstring,
      ParameterSet parSet) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "getProperty;", new Object[] {
        objectIdentity, propertyName, mapstring, parSet });

    Properties properties = parseAssignments(mapstring);
    String subExp = properties.getProperty("SubExp");
    int subExpNumber = Integer.parseInt(subExp);
    String regExpStr = properties.getProperty("gnuRegExp");

    // First verify that it is a supported property
    M12IdentityElement identityElement = objectIdentity.getScopingPath()[0];
    M12PropertySet propertySet = identityElement.getIdentity();
```

```
    // Get the path which is a key property.
    String fileName = propertySet.getProperty("fileName");
    long offSet = Long.parseLong(propertySet.getProperty("offset"));
    String matchedContent = propertySet.getProperty("matchedContent");

    RE propexp = new RE(regExpStr);
    String result = "";
    if (propexp.match(matchedContent)) {
      result = propexp.getParen(subExpNumber);
    }
    trace.exit(TSLTrace.DEBUG_MAX, this, "getProperty", result);
    return result;

  }
```

Implementing the ConnectResetILT ILT class

In this ILT class, the member function will be executed when there is a match found at the CBE. The following member functions are implemented in this Java class:

- ConnectResetILT()
- enumerateInstances()
- invokeMethod()

ConnectResetILT()

This constructor activates the trace entries for this ILT class.

Example 9-21 Implementation of ConnectResetILT()

```
public ConnectResetILT() {
    trace = TSLTraceImpl.getInstance();
    trace.logMsg(TSLTrace.DEBUG_MAX, ConnectResetILT.class.getName(),
        "ConnectResetILT", " Created. ");
  }
```

enumerateInstances()

This member function initializes the vector.

Example 9-22 Implementation of enumerateInstances()

```
public Enumeration enumerateInstances(
      M12ClassPath m12classpath,
      String mapping,
      ParameterSet parSet) {
    return new Vector(0).elements();
  }
```

invokeMethod()

This member function identifies the supported function and activates the utility class member function.

Example 9-23 Implementation of invokeMethod()

```
public String invokeMethod(
      M12ClassPath classPath,
      String method,
      String mapping,
      ParameterSet params,
      ParameterSet inParams,
      ParameterSet outParams) throws M12Exception {
    trace.entry(TSLTrace.DEBUG_MAX, this, "invokeMethod", new Object[] {
        classPath, method, mapping, params, inParams, outParams });

    // Verift that it is a supported method
    // Only resetDBConnection is supported
    if (method.equals("resetConnection")) {
      // String databaseServer = (String) inParams.getParam("databaseServer");
      System.out.println("Resetting connection to database server ");

      // At this point the fact that the RM and the Application is running in
      // the same JVM is misused. In a real scenario the application may be
      // running in an other JVM, thus RMI or someething similar should be
      // used to interact with the ConnectionPool.
      ConnectionPool.getInstance().setState(
          ConnectionPool.DATABASE_CONNECTION_OK);
    }

    trace.exit(TSLTrace.DEBUG_MAX, this, "invokeMethod");
    return "Connection reset successful";
  }
```

Creating the installation package

At this point, we have created all the elements required for the resource model. Now, we need to generate the resource model package, which can be run in the Autonomic Management Engine environment. This section includes the information required to create a JAR file for the ILT classes and custom CIM configuration using Tivoli Resource Model Builder.

The following activities need to be performed:

1. Include the MOF file into the project.
2. Compile the MOF file.
3. Include the CIM class attributes into the resource model.
4. Define the events.

5. Define the thresholds.
6. Define the parameters.
7. Define the dependencies.
8. Create the decision tree script.
9. Create the resource model package.

Refer to 8.7, "Creating the installation package" on page 279 for details about each step.

The following sections provide the resource model configuration details for this module.

MOF file

First, specify the LogFileMonitor.mof file.

CIM classes

Next, specify both LogFileMonitorCIM and ConnectResetCIM classes.

Events

Table 9-4 provides details for the event definitions for this example scenario module.

Table 9-4 DB2 resource model event definition

Field	Value
Internal name	`Ev_DatabaseStarted`
Descriptive name	`Database has been restarted.`
Message	`Database has been restarted.`
Default Event Hierarchy	Selected.
Severity	**CRITICAL**
Description	`Database has been restarted.`
Properties	Name: `databaseServer`, Type: `STRING`, Key: yes
Clearing Event	Selected
Number of occurrences	`1`
Number of holes	`0`
Action (CMI method)	
Method	`ITSO_SG246665` → `ConnReset` → `resetConnection`

Field	Value
Key	`dummy`
Descriptive name	`Reset Connection.`
Description	`Reset Connection.`

Parameters

Table 9-5 provides details for the parameter definitions for this example scenario module.

Table 9-5 DB2 resource model parameter definition

Field	Value
Internal name	`parLogName`
Descriptive name	`CBE LogFileName`
Description	`CBE LogFileName.`
List type (**Unrestricted** - **String**)	
Value	`.\logs\CBEApplicationLogfile.log` `.\logs\CBEDatabaseLogfile.log`

Defining dependencies

The following dependencies for the resource model are defined for module:

- DatabaseConnectionRM.mof

  The MOF file

- LogFileMonitor.jar

  The LogFileMonitor CIM class package

- ConnectResetCIM.jar

  The ConnectResetCIM CIM class package

Decision tree script

The section provides the implementation part of the decision script of the resource model created for this module. It includes the following methods:

- Init()

  This function initializes the log file name by associating the parLogName parameter to the LogFileMonitor class. See Example 9-24.

Example 9-24 Init() function implementation

```
function Init(Svc)
{
   Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start Init");
   Svc.AssociateParameterToClass("parLogName","LogFileMonitor");
       return(0);
}
```

- visitTree()

  This member function monitors the DB2LogFileMonitor attributes and makes a decision to generate an event if there is an event match. Example 9-25 shows the implementation.

Example 9-25 visitTree() function implementation

```
function VisitTree(Svc)
{
   Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Start VisitTree");

   //vars for data source: LogFileMonitor
   var numOfInstances;
   var idx;
   var situationCount = 0; // Count of the number of situations written this cycle

   var str_CBERecord;
        var situationCatName;

   var ParamCount;
   var ParamIdx;
   var Different;

   hPropTable = Svc.CreateMap();

   //triggering logic for data source: LogFileMonitor
   numOfInstances = Svc.GetNumOfInst("LogFileMonitor");
   for(idx = 0; idx < numOfInstances; idx++){

      Svc.RemoveMapAll(hPropTable);

      // read values from cbe
                severity = Svc.GetStrProperty("LogFileMonitor", idx, "severity");
                situationCatName = Svc.GetStrProperty("LogFileMonitor", idx,
"situationCatName");
                reasoningScope = Svc.GetStrProperty("LogFileMonitor", idx, "reasoningScope");
                situationQualifier = Svc.GetStrProperty("LogFileMonitor", idx,
"situationQualifier");
                situationDisposition = Svc.GetStrProperty("LogFileMonitor", idx,
"situationDisposition");
                successDisposition = Svc.GetStrProperty("LogFileMonitor", idx,
"successDisposition");
```

```
                reporterComponentApplication = Svc.GetStrProperty("LogFileMonitor", idx,
"reporterComponentApplication");
                sourceComponentName = Svc.GetStrProperty("LogFileMonitor", idx,
"sourceComponentName");

                Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Values: severity='"+severity+"'
situationCatName='"+situationCatName+"' reasoningScope='"+reasoningScope+"'
situationDisposition='"+situationDisposition+"' situationQualifier='"+situationQualifier+"'
successDisposition='"+successDisposition+"'");
                if (severity == 60
                  && sourceComponentName == "IBM DB2#8.1"
                  && reporterComponentApplication == "Scenario2"
                  && reasoningScope == "EXTERNAL"
                  && successDisposition == "UNSUCCESSFUL"
                  && ((
                    situationCatName == "ConnectSituation"
                    && situationDisposition == "AVAILABLE"
                  ) || (
                    situationCatName == "RequestSituation"
                    && situationQualifier == "REQUEST COMPLETED"
                  ))
                ) {
                  Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Database connection down");
                  connectionAvailable = false;
                } else if (severity == 10
                  && sourceComponentName == "DB2_UDB"
                  && situationCatName == "StartSituation"
                  && reasoningScope == "INTERNAL"
                  && successDisposition == "SUCCESSFUL"
                  && situationQualifier == "START COMPLETED"
                ) {
                  Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Database has been started/restarted,
resetting db connection.");
                  if (connectionAvailable) {
                    Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Application was not aware that the
database was down!");
                  }
                  connectionAvailable = true;
                  Svc.SetMapStrElement(hPropTable,
                        "databaseServer",
                        "sourceComponentName");
                  Svc.SendEventEx("Ev_DatabaseStarted",hPropTable);
                }

   }

   Svc.DestroyMap(hPropTable);

   return (0);
```

Creating the resource model package

For instructions about how to package the resource models, see 8.7.3, “Creating the resource model package” on page 302. We followed those steps to create a resource model package for this module. The package name is DatabaseConnectionRM.zip.

This package file is provided with this book. For details about how to obtain the file, refer to Appendix A, “Additional material” on page 379.

9.2.5 Application node module two

This section provides details about the development and implementation of the components of application node module two. Figure 9-8 outlines the components that we discuss in this section.

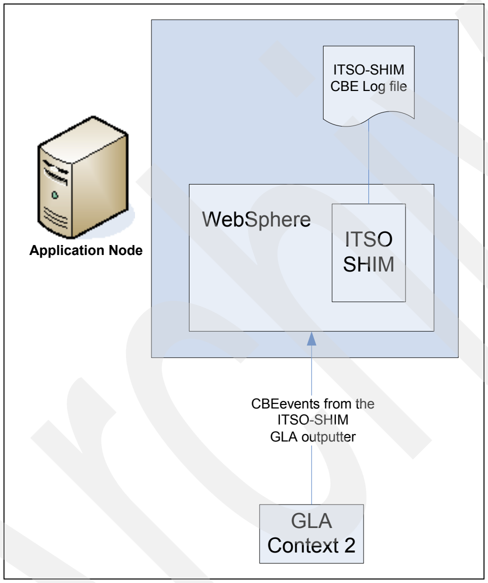


Figure 9-8 Application node module two

In order to be able to receive the events from the GLA custom outputter developed in 9.2.3, “Database node module three” on page 330, we developed a small Java 2 Platform Enterprise Edition (J2EE) application named ITSOShimEAR that will perform the following tasks:

1. Enable Web services for the GLA context running on the database node.
2. Receive CBE events from the GLA context outputter.
3. Write the received CBE events into a log file for processing by the resource model.

The implementation of the ITSOShimEAR application required the development of a class to store the events into a file. We describe this class, CbeWriterThread, in the following section.

CbeWriterThread class

This Java class is a utility class of the Web service. The basic functionality of the class is to write the received CBE into the CBEDatabaseLogfile.log file.

This class requires the attribute name of the log file to be used to store the received log files:

```
private static String logFileName = "./logs/CBEDatabaseLogfile.log";
```

The class contains a method to write the received CBE. The event is stored in the specified file in XML format.

Example 9-26 provides a complete listing of the CbeWriterThread class.

Example 9-26 CbeWriterThread class

```
package com.ibm.rcp.autonomic.cei.ws;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.StringWriter;
import java.io.Writer;
import java.util.Vector;

import org.apache.xml.serialize.XMLSerializer;
import org.eclipse.hyades.logging.events.cbe.CommonBaseEvent;
import org.eclipse.hyades.logging.events.cbe.FormattingException;
import org.eclipse.hyades.logging.events.cbe.util.EventFormatter;
import org.w3c.dom.Element;

import com.ibm.rcp.autonomic.cei.CeiSource;
import com.ibm.rcp.autonomic.events.EventUtils;

public class CbeWriterThread extends Thread {
   private static String logFileName = "./logs/CBEDatabaseLogfile.log";

   public static CommonBaseEvent element2Cbe(Element el) {
      XMLSerializer serializer = new XMLSerializer();
      StringWriter sw = new StringWriter();
      serializer.reset();
      serializer.setOutputCharStream(sw);
      try {
         serializer.serialize(el);
```

```
   } catch (Exception e) {
      e.printStackTrace();
   }
   String cbeStr = sw.toString();
   cbeStr = cbeStr.substring(cbeStr.indexOf("<CommonBaseEvent "));
   return xmlStr2Cbe(cbeStr);
}

/**
 * @return Returns the logFileName.
 */
public static String getLogFileName() {
   return logFileName;
}

/**
 * Get the Writer where the events will be written to. If the logOutput is
 * null then a new FileWriter will be made with the logFileName.
 *
 * @return Returns the logOutput.
 * @throws IOException
 *            Error opening logfile
 */
public static Writer getLogOutput() throws IOException {
   FileWriter logOutput = null;
   try {
      logOutput = new FileWriter(logFileName, true);
   } catch (FileNotFoundException e) {
      // propably because the directory does not exist
      new File("./logs").mkdirs();
      logOutput = new FileWriter(logFileName, true);
   }

   // Role the files if needed
   File f = new File(logFileName);
   if (f.length() > 1048576) {
      logOutput.close();
      for (int i = 9; i > 0; i--) {
         File frole = new File(logFileName + "." + i);
         if (frole.exists()) {
            if (i == 9) {
               frole.delete();
            } else {
               frole.renameTo(new File(logFileName + "." + (i + 1)));
            }
         }
      }
      f.renameTo(new File(logFileName + ".1"));
      logOutput = new FileWriter(logFileName, true);
```

```
        }

        return logOutput;
    }

    /**
     * Writes a CBE event to the CBE logfile.
     *
     * @param cbe
     *            The CBE event to be logged.
     */
    public static void logCBEEvent(CommonBaseEvent cbe) {
        // Create a xml string with the CBE
        String cbeXmlFormat = EventFormatter.toCanonicalXMLString(cbe, false);
        Writer logOutput = null;
        try {
            // Write the cbe to the file
            logOutput = getLogOutput();
            logOutput.write(cbeXmlFormat);
            logOutput.write(Character.LINE_SEPARATOR);
            logOutput.flush();

        } catch (IOException e) {
            System.out.println("Error writing CBE logfile " + logFileName);
            e.printStackTrace();
        } finally {
            if (logOutput != null) {
                try {
                    logOutput.close();
                } catch (IOException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }
            }
        }
    }

    /**
     * Specifies the filename of the CBE logfile
     *
     * @param logFileName
     *            The logFileName to set.
     */
    public static void setLogFileName(String logfileName) {
        CbeWriterThread.logFileName = logfileName;
    }

    public static CommonBaseEvent xmlStr2Cbe(String xmlStr) {
        try {
```

```
        return EventFormatter.eventFromCanonicalXML(xmlStr);
    } catch (FormattingException fe) {
        fe.printStackTrace();
        return null;
    }
}

private Vector _buffer;

private boolean _isRunning = false;

private String _providerUrl;

public CbeWriterThread(Vector buffer) {
    _buffer = buffer;
    _isRunning = true;
    _providerUrl = "corbaloc:iiop:localhost:2809";
}

public void run() {
    System.out.println(">>>>>>>>CBE Writer Thread running");
    CeiSource ceiProxy = new CeiSource(_providerUrl);
    while (_isRunning) {
        synchronized (_buffer) {
            while (_buffer.size() > 0) {
                Element cbeXmlElement = (Element) _buffer.remove(0);
                String rootNodeName = cbeXmlElement.getNodeName();
                if (rootNodeName != null
                        && rootNodeName.endsWith("CommonBaseEvents")) {
                    try {
                        CommonBaseEvent[] cbes = EventUtils
                                .xml2Cbes(cbeXmlElement.getOwnerDocument());
                        if (cbes != null) {
                            for (int i = 0; i < cbes.length; i++) {
                                try {
                                    logCBEEvent(cbes[i]);
                                } catch (Exception ee) {
                                    // log this
                                }
                            }
                        }
                    } catch (Exception e) {
                        // from EventUtils; log this
                    }
                } else if (rootNodeName != null
                        && rootNodeName.endsWith("CommonBaseEvent")) {
                    try {
                        CommonBaseEvent cbe = element2Cbe(cbeXmlElement);
                        logCBEEvent(cbe);
```

```
                  } catch (Exception ee) {
                     // log this
                     ee.printStackTrace();
                  }
               }
               try {
                  // now sleep the specified period of time
                  Thread.sleep(100);
               } catch (InterruptedException e) {
               }
            }
         }
      }
   }

   public void setProviderUrl(String providerUrl) {
      _providerUrl = providerUrl;
   }

   public void stopRunning() {
      _isRunning = false;
   }
}
```

9.3 Scenario execution

The files developed for this example scenario are available to download. Refer to Appendix A, “Additional material” on page 379 for instructions about how to obtain the files. This section explains how to execute the example scenario presented in this chapter.

As shown in Figure 9-1 on page 306, this scenario requires two computers running Microsoft Windows 2000 Server or Advanced Server.

In order to implement the self-managing autonomic capabilities of this scenario, the following IBM Autonomic Computing Toolkit Version 2.0 tools and technologies must be installed on both the application and database nodes:

- Autonomic Management Engine V1.1
- Resource Model Builder V1.3
- Generic Log Adapter Runtime and Rule Sets V3.0.0
- Log and Trace Analyzer V3.0.0
- Eclipse Tooling Package V3.0

Refer to Chapter 2, “IBM Autonomic Computing Toolkit overview” on page 25 for information about how to obtain these tools and technologies.

The sample Java application runs on the application node and accesses a database hosted by an IBM DB2 server running on the database node. Therefore, the installation of an IBM DB2 UDB server is a prerequisite in the database node.

The sample application requires a database for data insertion. On the IBM DB2 server, create a database named SAMPLE. The instance owner of the SAMPLE database is ADMINISTRATOR. Two tables must be created in the SAMPLE database, as shown in Table 9-6.

Table 9-6 SAMPLE database tables

Table name	Data type
UERID	INTEGER
USERNAME	VARCHAR

Create an ODBC connection from the application node to the SAMPLE database using the ADMINISTRATOR credentials.

Also, on the database node, start the Telnet service. The Telnet service will be required for the sample application to perform stop and start operations on the DB2 server. The sample application initiates the Telnet session to the database node using an implementation of Jakarta Commons Net. The components required for this operation are available at the Jakarta Web site under the Apache License:

`http://jakarta.apache.org/commons/net/`

For licensing information, refer to:

`http://jakarta.apache.org/commons/license.html`

The sample application runs on the application node. We developed the sample application with an embedded Autonomic Management Engine (AME). For this reason, there is no need to have SARA up and running in the application node. Refer to Chapter 7, “Advanced Autonomic Management Engine topics” on page 231 for details about the development of the sample application embedding AME.

On the application node, we developed a small J2EE application named ITSOShimEAR that will be a Web service for the scenario. The ITSOShimEAR application runs on IBM WebSphere Application Server. During the development of this book, we used IBM WebSphere Application Server Version 5.1. Refer to the following site for information about how to obtain IBM WebSphere Application Server software:

`http://www.ibm.com/software/webservers/appserv/was/`

9.3.1 Starting the database node module one

This section describes how to start the components of database node module one. Perform the following steps:

1. After you download the scenario's executable files, open a command prompt and move to the directory where you unzipped the database node module one scenario files.
2. Change to the directory in which AME has been installed, for example:

   ```
   cd c:\IBM\AutonomicComputingToolkit\AutonomicManagementEngine
   ```

3. Change to the SARA directory and start the SARA runtime environment:

   ```
   cd sara
   sara.bat
   ```

4. When the ready prompt appears, start the RME using the following command:

   ```
   startme
   ```

   `RME started` will be displayed.

5. Execute the following command to install the example scenario resource models:

   ```
   instrmtype “<SCENARIO_DIRECTORY>\DB2ResourceManagement.zip” -replacefiles
   ```

   Where `<SCENARIO_DIRECTORY>` is the directory containing the scenario's downloadable files.

6. Make an instance of the scenario resource models using the following command:

   ```
   mkrminstance <INSTANCE_NAME> DB2ResourceManagement
   ```

7. Start the resource model instance using the following command:

   ```
   startrminstance <INSTANCE_NAME>
   ```

Stopping the database node module one execution

To stop the execution of this scenario, perform the following steps:

1. Stop the resource model instance:

   ```
   stoprminstance <INSTANCE_NAME>
   ```

2. Remove the scenario resource models using the following command:

   ```
   rmrminstance <INSTANCE_NAME>
   ```

3. Uninstall the resource models from SARA:

   ```
   uninstrmtype DB2ResourceManagement
   ```

9.3.2 Starting the database node module two

As described in 4.4, “Deploying and running the adapter configuration file” on page 121, the Generic Log Adapter Runtime can be started through a command-line interface (CLI) using a batch file or shell script file located at <GLAinstall>\bin\gla.bat, where <GLAinstall> is the location of the Generic Log Adapter installation.

Execute the following command on the database node to start the GLA adapter for this scenario:

```
<GLAinstall>\bin\gla.bat <adapter_dir>\db2parser.adapter
```

Where `<adapter_dir>` is the absolute path to the configuration file, and `<GLA_Install>` is the installation directory of the Generic Log Adapter tool.

9.3.3 Starting the database node module three

Similarly to module two, execute the following command on the database node to start the GLA adapter for this scenario:

```
<GLAinstall>\bin\gla.bat <adapter_dir>\ShimOutputter.adapter
```

Where `<adapter_dir>` is the absolute path to the configuration file, and `<GLA_Install>` is the installation directory of the Generic Log Adapter tool.

9.3.4 Starting the application node module one

To start the components of application node module one, perform the following steps:

1. After you downloaded the scenario’s executable files, open a command prompt and move to the directory where you unzipped the application node module one scenario files.
2. The scenario application needs to be started. This is done by executing the **startACPD_SC2_AM1_JavaApplication.bat** script.

   The **startACPD_SC2_AM1_JavaApplication.bat** script needs to be customized to match your environment. Edit the script and change the following line with your settings:

   ```
   REM ############### Environment configuration
   REM # Adjust the following env variables to match your system
   set SCENARIO_HOME=
   set AME_HOME=
   set JRE_HOME=
   ```

3. In addition, the sample application needs to know the DB2 server information. As it is starting, the sample application reads a configuration file in which DB2 server information is stored. This information is stored in the InitCommands.txt file. Edit this file and change the information to match your environment. The following parameters need to be set:

`setAdminUser`	The user ID of the administrator account on the database node. This user ID is required to perform remote access to the database node during execution of the sample application.
`setAdminPassword`	The password of the administrator account on the database node.
`setDbUserName`	The user ID of the instance owner of the SAMPLE database.
`setDbPassword`	The password of the instance owner of the SAMPLE database.
`setDbConnectURL`	The URL for connection to the IBM DB2 server running on the database node, for example, `jdbc:db2://<HOSTNAME>:<PORT>/sample`, where `<HOSTNAME>` is the host name of the database node, and `<PORT>` is the TCP/IP port for IBM DB2.

4. After these variables are properly set, execute the **`startACPD_SC2_AM1_JavaApplication.bat`** script.

   The sample application starts the AME environment, installs, creates an instance, and starts the instance of the DatabaseConnectionRM resource model.

Stopping the application node module one execution

This section provides instructions about how to stop the execution of this scenario.

The sample application provides a prompt CLI. Enter **`quit`** at the prompt to stop the execution of the sample application, AME, and resource models.

9.3.5 Starting the application node module two

This section provides instructions to start the components of application node module two.

Perform the following steps on the WebSphere administrative console:

1. Install the ITSOShimEAR.ear file in a working IBM WebSphere environment. The version used for the development of this book is IBM WebSphere Application Server V5.1. Use the default configuration options during the installation. When the installation process finishes, save the WebSphere configuration.
2. Set the directory in which the ITSOShinEAR application will create the CBE log file:
   a. Select **Servers** → **Application Servers** → **server1**.
   b. Select **Process Definitions**.
   c. Change the working directory to the directory in which you stored the scenario module one files. The ITSOShinEAR application will create the following log file:

      <SCENARIO_ONE>\log\CBEApplicationLogfile.log

      We require the module one directory because the resource model part of scenario one module will read this log file.
3. Restart WebSphere Application Server.
4. Start the ITSOShimEAR application.

9.3.6 The self-healing environment

Now that all the module components are up and running on both the application and database nodes, we introduce an error into the environment and check the autonomic managers taking corrective actions.

After the environment is properly set, the expected behavior of the environment is as follows:

1. All components of the application are operational, up, and running, on their respective nodes. The sample application inserts a record into the database every 10 seconds. A message is displayed in the sample application's prompt stating a successful data insertion:

   ```
   Inserting User NNN into database.
   ```

2. An error is introduced in the environment. In this example scenario, we terminate the database instance execution so that the application will be unable to perform data insertion. This can be achieved by issuing the following command at the sample application prompt:

   ```
   → sc2 stopDatabase
   ```

   This command starts a Telnet session to the database node and issues the **`db2stop force`** command.

Alternatively, you can manually stop DB2 on the database node issuing the **db2stop force** command.

The application will log a Common Base Event into its log file, reporting a connection failure to the database. A message is displayed in the sample application's prompt stating that a database connection is down:

```
Error on database connection, setting the state to down.
```

3. A resource model running on the autonomic manager deployed in the application node detects the incident reported by the application using ITL. As an initial step to correct the problem, the resource model signals the application to put data insertion on hold. The application puts data insertion tasks on hold and waits until the incident is resolved.
4. IBM DB2 logs an event in its log file stating that the database instance is down.
5. The GLA Context1, which monitors the DB2 log file, converts the log information into Common Base Event format and stores them into a CBE DB2 log file.
6. A resource model running on the autonomic manager deployed in the database node monitors the CBE DB2 log file using ITL. The resource model analyzes the event reporting that the database instance down and takes corrective action. The corrective action is to restart the database instance.
7. DB2 logs an event in its log file stating that the database instance is operational.
8. The GLA Context2, which also monitors the DB2 log file, converts the log information into Common Base Event format and uses a custom outputter to send the event to the application node using Web services. The custom outputter name is ITSO-SHIM outputter.
9. On the application node, a small application named ITSO-SHIM running on WebSphere is accessed as a Web service by the GLA Context2. The ITSO-SHIM Web service receives events from GLA Context2 and saves them into a log file.
10. A resource model running on the autonomic manager deployed in the application node monitors the ITSO-SHIM log file using ITL. The resource model analyzes the event reporting that the database instance is operational and takes corrective action. The corrective action is to restart the database connection to the application's database and communicate to the application that the database is back to an operational state.
11. After the database connection is restored, you can verify that the application resumes normal activity when a message is displayed in the sample application's prompt stating a successful data insertion:

```
Inserting User NNN into database.
```

10

Problem Determination scenario three

This chapter describes a variation of the scenario presented in the previous chapter. It shows a case in which an administrator is in charge of collecting, monitoring, and correlating events generated by self-managed autonomic technology. This scenario shows how this can be achieved using self-managed autonomic technology and tools provided by the IBM Autonomic Computing Toolkit Version 2.0.

This chapter describes the definition, implementation, package, and execution of this scenario and covers the following topics:

- Scenario description
- Scenario development and implementation

10.1 Scenario description

In this chapter, we show a variation of the example scenario described in the previous chapter. In this chapter, the sample Java application runs on the application node and accesses a database hosted by an IBM DB2 server running on the database node. The sample Java application generates Common Base Events (CBEs) natively and logs them into a log file. Figure 10-1 depicts the sample Java application used in this case study scenario chapter.

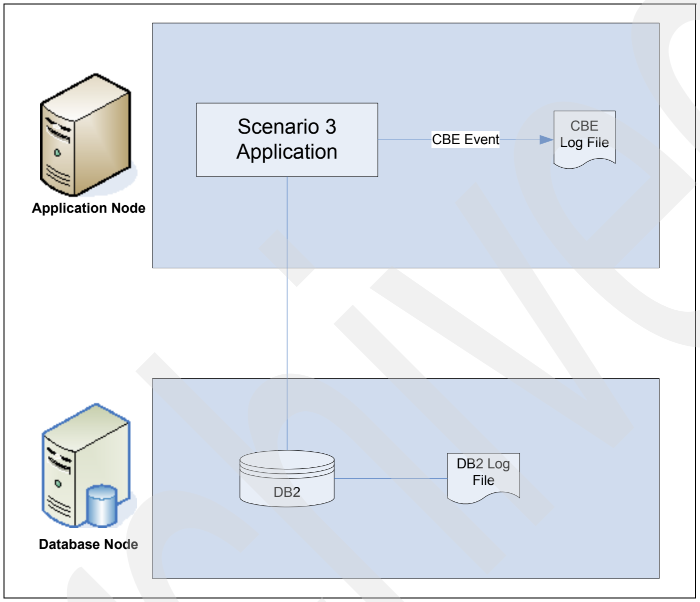


Figure 10-1 Application scenario three environment

In order to provide the required autonomic solution for the application scenario described here, the following actions need to be performed *in addition to* those performed in Chapter 9, “Problem Determination scenario two” on page 305. We only discuss the following steps in this chapter:

- On the database node:
  - Install the Agent Controller component.
  - Develop a Generic Log Adapter (GLA) context to convert the log information created by DB2 into a CBE format. This GLA context will have a custom outputter to write Common Base Events to the Remote Agent Controller (RAC).
- On the application node:
  - Install the Agent Controller component.

– Create a Java class to allow the sample Java application to write its Common Base Events to the Remote Agent Controller.

- On the administration node:
  – Install and configure the log and Log and Trace Analyzer V3.0.0 tool.

Figure 10-2 shows the application scenario environment with the autonomic computing elements in place.

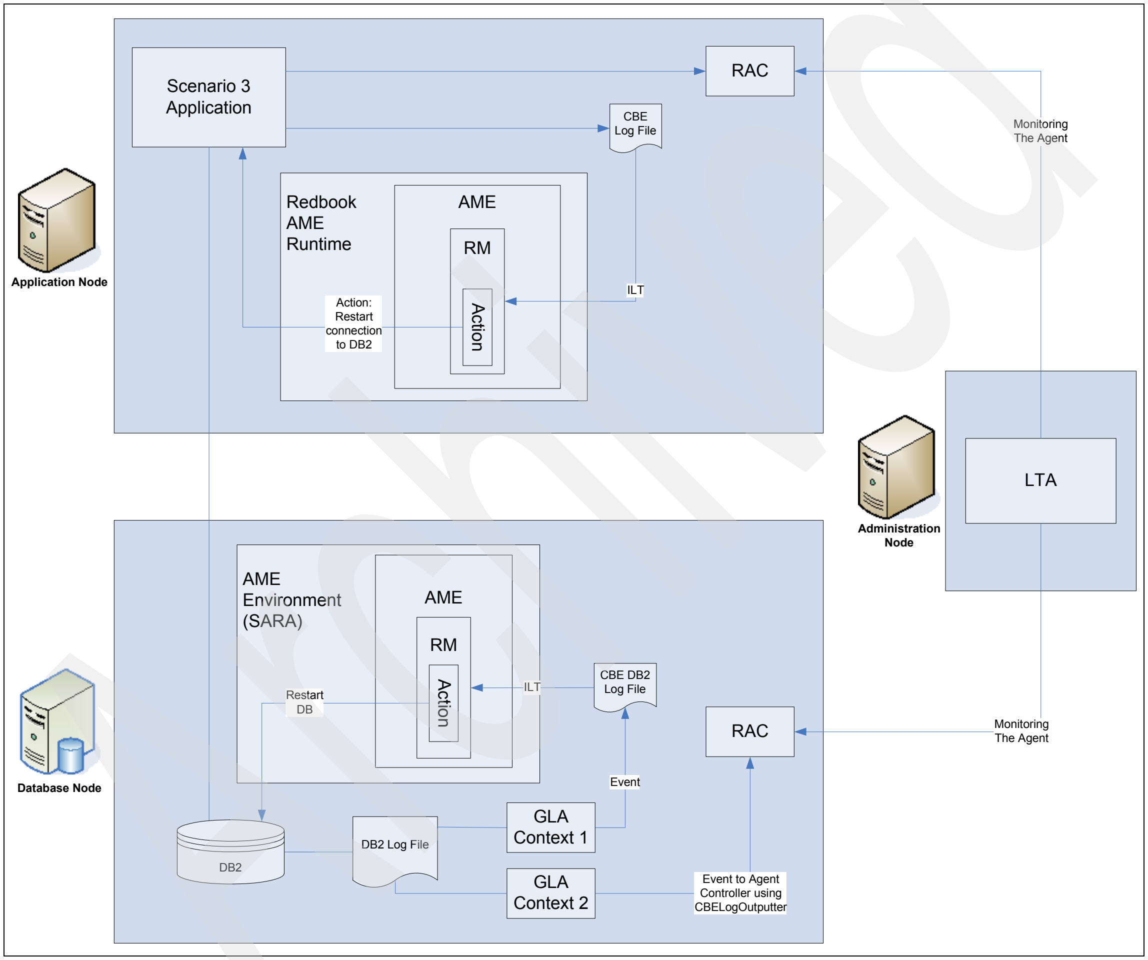


Figure 10-2 Autonomic application scenario three environment

Using the IBM Autonomic Computing Toolkit and other development tools, we create, develop, and configure self-managed autonomic technology, as described in the following sections.

10.2 Scenario development and implementation

The development, implementation, and execution of this example scenario is divided into modules, as shown in the following sections.

All the modules created in this chapter are available to download. Refer to Appendix A, “Additional material” on page 379 for details about how to obtain these modules.

In order to develop the self-managing autonomic capabilities of this scenario, the following IBM Autonomic Computing Toolkit Version 2.0 tools and technologies must be installed on both the application and database nodes:

- Autonomic Management Engine V1.1
- Resource Model Builder V1.3
- Generic Log Adapter Runtime and Rule Sets V3.0.0
- Log and Trace Analyzer V3.0.0
- Eclipse Tooling Package V3.0

Refer to Chapter 2, “IBM Autonomic Computing Toolkit overview” on page 25 for information about how to obtain these tools and technologies.

The following sections provide detailed information about each of the activities required for the development of this scenario, described in 10.1, “Scenario description” on page 362.

10.2.1 Database node module

This section provides details about the development and implementation of the components of the database node module. Figure 10-3 outlines the components that we discuss in this section.

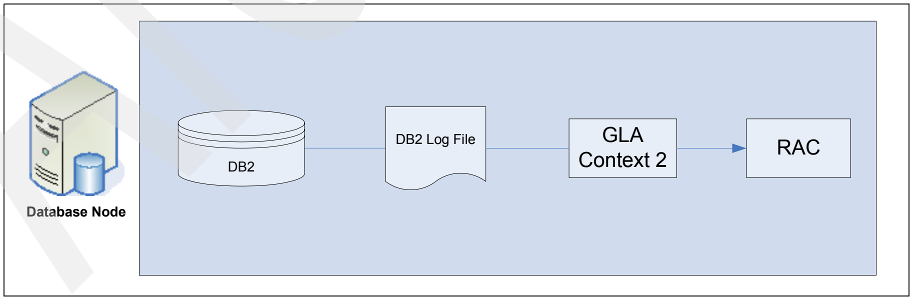


Figure 10-3 Database node module

This module handles the conversion of events logged by IBM DB2 in the db2diag.log file into the Common Base Events format and writes the events to a

Remote Agent Controller (RAC) installed on the same server. Figure 10-3 on page 364 illustrates the functionality of this module. The GLA Context2 adapter in this module is configured with the sensor to read the db2diag.log file and the outputter to write into the RAC.

GLA adapter development

For detailed instructions about how to create an GLA adapter file, see 4.2, “Creating a simple adapter” on page 83.

A Generic Log Adapter named db2parser.adapter is created using a GLA adapter provided with the Generic Log Adapter tool. We used this file to take advantage of the parsing rules being already defined properly for the specific file that we are interested in parsing (db2diag.log). All the sample adapters provided by the Generic Log Adapter tool are in the C:\<AC_TOOLKIT_INST>\GenericLogAdapter\config\ directory, where <AC_TOOLKIT_INST> is the installation directory of the IBM Autonomic Computing Toolkit V2.0.

The adapter fie used as the base for the adapter created in this section is C:\<AC_TOOLKIT_INST>\GenericLogAdapter\config\DB2\diag\v8.2\regex.adapter.

The db2parser.adapter converts IBM DB2 events written into the db2diag.log file. This Generic Log Adapter parses the events in db2diag.log, converts it into Common Base Events, and writes to an agent named DB2LogAgent. The adapter is developed using the following context information:

Sensor: SingleFileSensor type with executable class org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor.

Outputter: LoggingAgentOutputter type with executable class org.eclipse.hyades.logging.adapter.outputters.CBELogOutputter. The agentName used is *DB2LogAgent*.

Configure the context component to use the CBELogOutputter, as shown in Figure 10-4 on page 366. Refer to 5.2, “Using Agent Controller for monitoring remote logs” on page 158 for details.

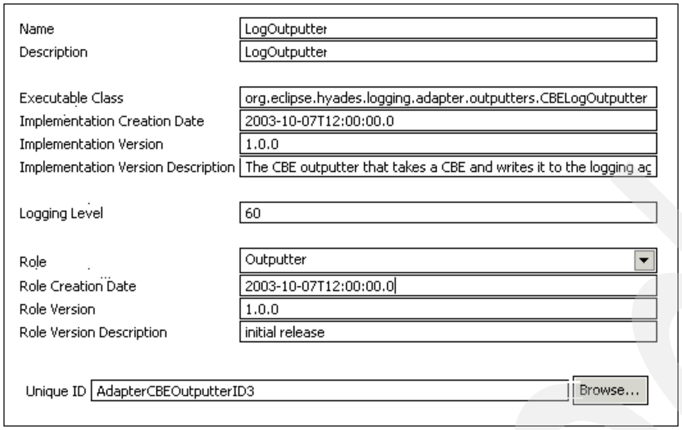


Figure 10-4 The CBELogOutputter in the Generic Log Adapter

For the continuous monitoring of the db2diag.log file, make sure that the **Continuous Operation** option in the Context Instance is selected. Increase the Maximum Idle Time in the Context Instance to a higher value to avoid disconnecting the GLA Runtime if there are no new events logged into db2diag.log for a certain period of time.

Configuring the RAC

After installing the Remote Agent Controller (RAC), it must be configured. The serviceconfig.xml file needs to be edited and modified with the agent name, because this file is loaded by the Agent Controller when it starts.

The serviceconfig.xml file is in the <RAC_install>\config directory, where <RAC_install> is the remote agent installation directory.

Add the following lines in the serviceconfig.xml file to create an agent with the name, for example, DB2LogAgent. In our example, the DB2LogAgent will be used by the Generic Log Adapter outputter to write the formatted Common Base Event messages.

```
<Agent configuration="default" name="DB2LogAgent" type="Logging">
</Agent>
```

Example 10-1 on page 367 shows the complete serviceconfig.xml file after modification. Note that the Agent Controller in this scenario is configured to listen to the default TCP/IP port 10002.

Example 10-1 Modified serviceconfig.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<AgentControllerConfiguration activeConfiguration="default" filePort="10005"
jvm="C:\Program Files\IBM\Java142\jre\bin\classic\jvm.dll" loggingDetail="LOW"
loggingLevel="INFORMATION" port="10002" processPolling="true"
securedPort="10003" version="3.0.1.1">
   <AgentControllerEnvironment configuration="default">
      <Variable name="JAVA_PATH" position="replace" value="C:\Program
Files\IBM\Java142\bin\java.exe"/>
      <Variable name="RASERVER_HOME" position="replace" value="C:\Program
Files\IBM\AgentController"/>
      <Variable name="PATH" position="prepend" value="%RASERVER_HOME%\bin"/>
      <Variable name="SYS_TEMP_DIR" position="replace"
value="C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp"/>
      <Variable name="LOCAL_AGENT_TEMP_DIR" position="replace"
value="%SYS_TEMP_DIR%"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\commons-logging.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hcframe.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hexcore.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hexl.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hexr.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hl14.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlcommons.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlcore.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlevents.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\hlcbe101.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\ecore.jar"/>
      <Variable name="CLASSPATH" position="append"
value="%RASERVER_HOME%\lib\common.jar"/>
   </AgentControllerEnvironment>
   <Application configuration="default" executable="java.exe"
location="%SYS_TEMP_DIR%" path="%JAVA_PATH%"/>
   <Hosts configuration="default">
      <Allow host="ALL"/>
   </Hosts>
   <Agent configuration="default" name="DB2LogAgent" type="Logging">
     </Agent>
   <Plugin path="%RASERVER_HOME%\plugins"/>
</AgentControllerConfiguration>
```

Restart the Agent Controller service to use the modified configuration file. In Windows, the service name is *IBM Rational Agent Controller*.

10.2.2 Application node module

This section describes the development and implementation of the components of the application node module. Figure 10-5 outlines the components that we discuss in this section.

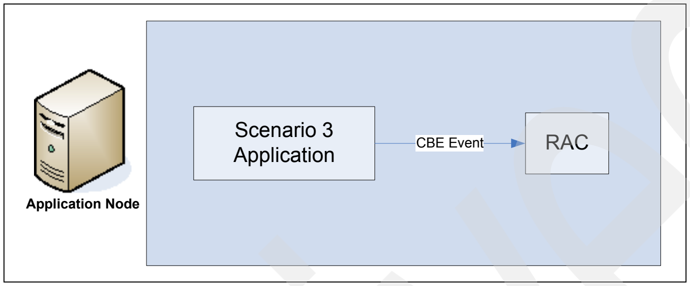


Figure 10-5 Application node module

The application uses the Java APIs provided with the Common Base Event (CBE) Java library to generate the CBEs natively. The application writes the CBEs to both a log file and to a logging agent named *AppLogAgent*. Later in this chapter, a Remote Agent Controller (RAC) is installed and configured to represent the AppLogAgent agent. Refer to 3.7, “Generating Common Base Events using APIs” on page 70 for more information about how to write an application that uses Java APIs for generating the CBEs.

In this scenario, the CBELogger class is developed and used in the application to log the CBEs into the AppLogAgent agent. This class uses the org.eclipse.hyades.logging.core.LoggingAgent class to write the CBEs to the logging agent AppLogAgent. Example 10-2 shows the listing of the CBELogger class.

Example 10-2 Listing of the class that writes to the RAC

```
package com.ibm.itso.sg246665.scenario.logger;

import java.io.*;
import org.eclipse.hyades.logging.events.cbe.CommonBaseEvent;
import org.eclipse.hyades.logging.core.LoggingAgent;
import org.eclipse.hyades.logging.events.cbe.util.EventFormatter;

public class CBELogger
{
  public CBELogger logger;
```

```
static String logAgentName = "AppLogAgent";
static LoggingAgent cbelogger = new LoggingAgent(logAgentName);

public CBELogger() {}

public static void logCBEEvent(CommonBaseEvent cbe)
{

   try {
   cbelogger.write(EventFormatter.toCanonicalXMLString(cbe, false));
   }
   catch(Exception e)
   {
       System.out.println("Error writing CBE logfile ");
       e.printStackTrace();
   }
}
```

Configuring the RAC

Similarly to the process performed on the database node, the Remote Agent Controller (RAC) must be installed on the application node and configured. The serviceconfig.xml file needs to be edited and modified with the agent name, because this file is loaded by the Agent Controller when it starts.

The serviceconfig.xml file is in the <RAC_install>\config directory, where <RAC_install> is the remote agent installation directory.

Add the following lines in the serviceconfig.xml file to create an agent with the name, for example, AppLogAgent. In our example, the AppLogAgent agent will be used by the Generic Log Adapter outputter to write the formatted Common Base Event messages.

```
<Agent configuration="default" name="AppLogAgent" type="Logging">
</Agent>
```

10.2.3 Administration node module

The Agent Controller provides the convenience of analyzing a log file from a remote machine without having to transfer a copy of the file to a local machine. Administrators can then use the Log and Trace Analyzer (LTA) tool running on another machine, the administration node in this example scenario, to import, analyze, and correlate log files that are continuously updated by their respective applications running on remote machines.

In this example scenario, the LTA tool is configured to continuously access information provided by the AppLogAgent and DB2LogAgent RACs running on the application and database nodes, respectively, as shown in Figure 10-6.

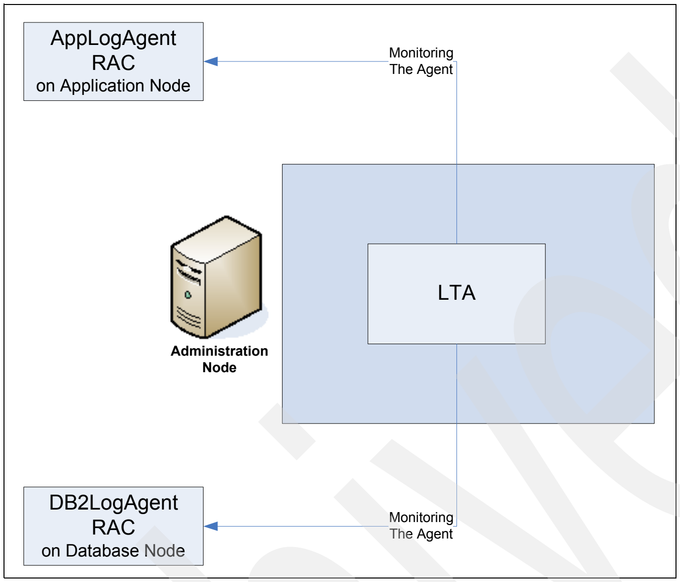


Figure 10-6 LTA set up on the administration node

As described in 5.2, "Using Agent Controller for monitoring remote logs" on page 158, the following sections describe the main steps for attaching and monitoring the RACs.

This following section describes how to attach to and monitor the remote agent.

Creating monitoring profiles

Perform the following steps to create monitoring profiles in the LTA for the RACs running on the application and database nodes:

1. Start Eclipse.
2. Select **Window** → **Preferences** and click **Profiling and Logging**. On right pane, select **Enable logging**.

3. Select **Window** → **Open Perspective** → **Other**. A wizard, as shown in Figure 10-7, opens.

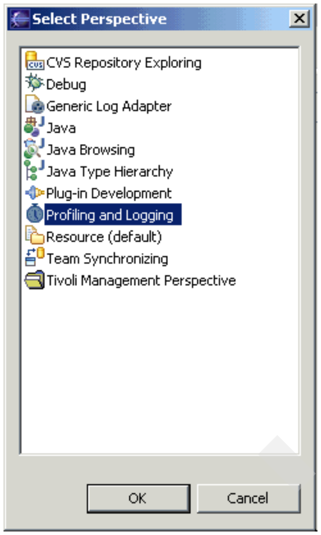


Figure 10-7 Select the Profiling and Logging perspective

4. Select the **Profiling and Logging** perspective, as shown in Figure 10-7, and click **OK**.

5. Select **Run** → **Profile** and click **Attach - Java Process**. Now, the wizard shown in Figure 10-8 opens, where we can create a new profile and attach it to the RAC.

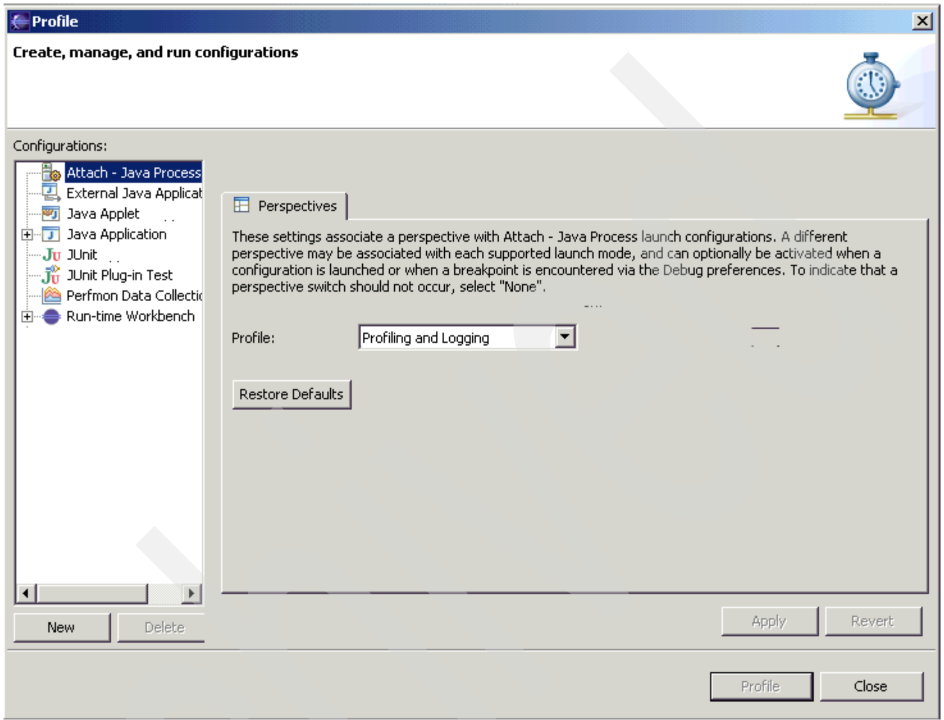


Figure 10-8 Wizard for creating a new profile

6. Right-click **Attach - Java Process** and select **New** to attach a new Java process and name it, for example, `DB2LogProfile`, to monitor the DB2logAgent. This is shown in Figure 10-9.

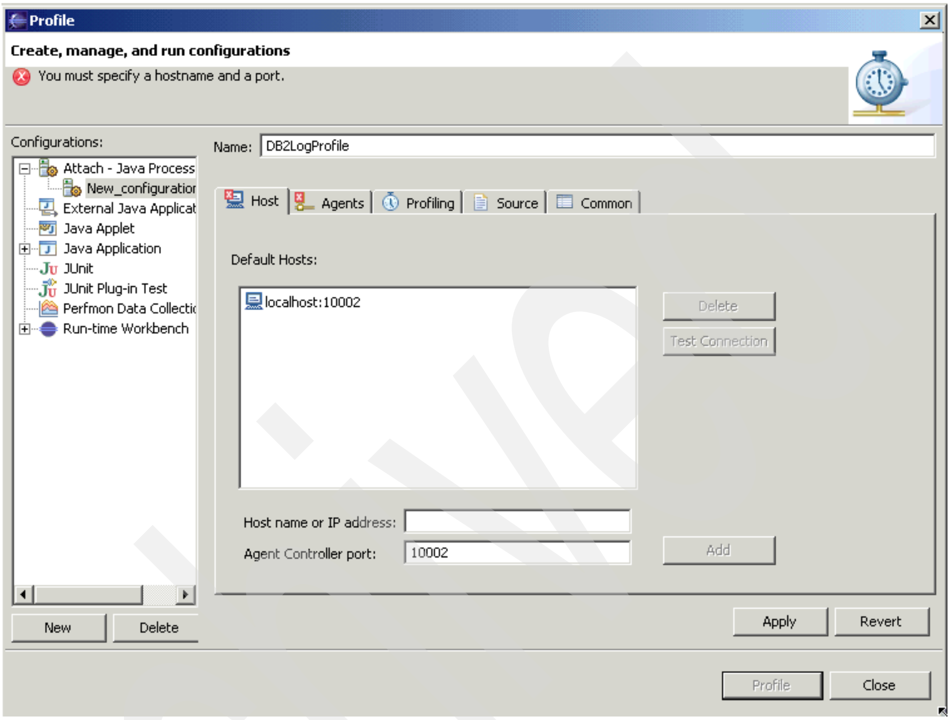


Figure 10-9 Create a new profile

7. Now, we need to add the remote host name where the Agent Controller is logging the Common Base Events. For this, in the Host tab, enter the Host name or IP address and the Agent Controller port on the remote system and click **Add**. Here, we entered the host name of the database node, for example.

   The default port that the Agent Controller uses is 10002. We use the default port in this example. Figure 10-10 on page 374 shows the configuration of the host.

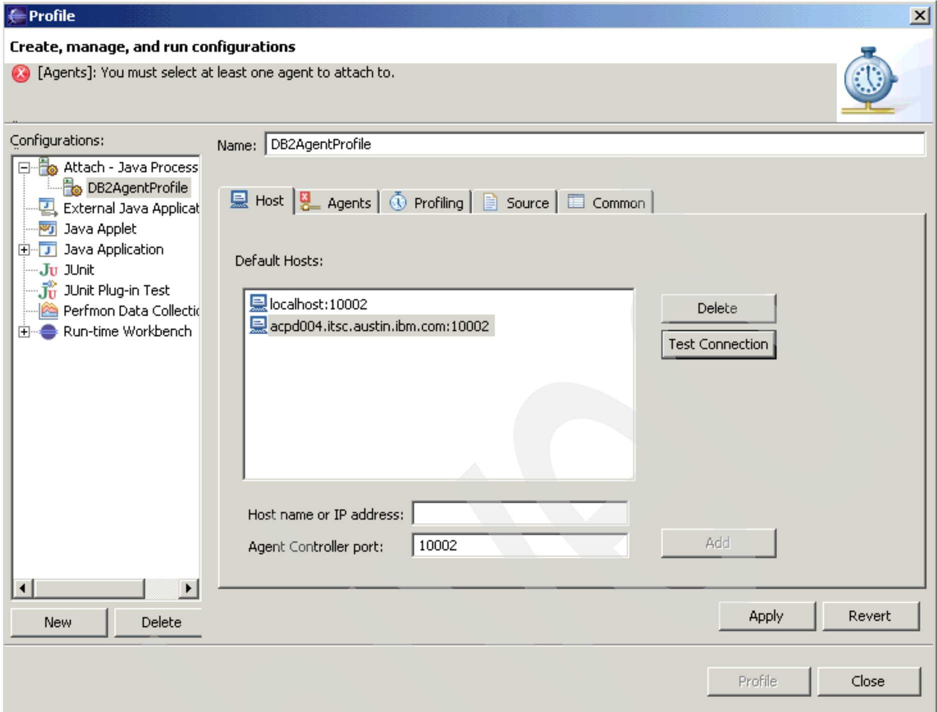


Figure 10-10 Configure the remote host in the profile

8. Click **Test Connection** to test the connection to the remote system. If the connection succeeds, click **Apply**.
9. These steps must be repeated to create a profile for the RAC running on the application node. We named it AppLogProfile, using the host name and port defined to the RAC running on the application node.

Attaching the LTA to the agents

The Profiling Tool of the Log and Trace Analyzer provides the ability to attach to the running agent. Therefore, to attach the agent, we need to launch the applications that write to the logging agent. For this example, the sample application running on the application node and the Generic Log Adapter writing to the AppLogAgent and DB2LogAgent RACs, respectively, must be up and running.

To attach DB2LogAgent and monitor the remote log, complete the following steps:

1. On the LTA console, click **Run** → **Profile**, expand **Attach - Java Process**, and click an available profile, for example, **DB2LogProfile**.

2. Go to the Agents tab. The agents that are currently active for the profile are listed, as shown in Figure 10-11.

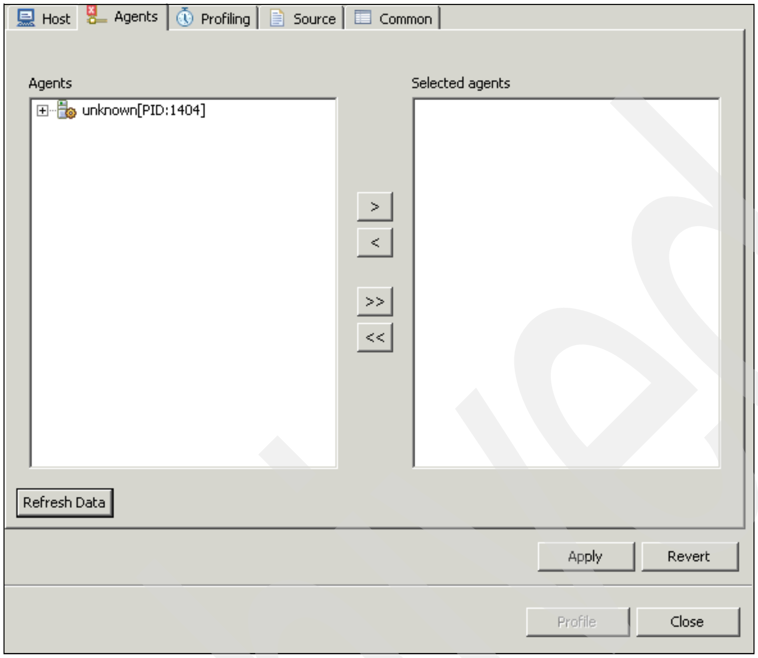


Figure 10-11 List of running agents

3. Now, select **DB2LogAgent** and click **>** to select the agent to monitor.
4. Click **Apply**, and then click **Profile**. The attached log agent is displayed in Profiling Monitor, as shown in Figure 10-12 on page 376.

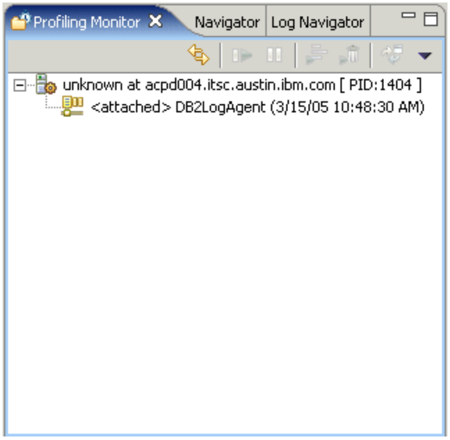


Figure 10-12 Display of DB2LogAgent in Profiling Monitor

5. Right-click the attached **DB2LogAgent** in the Profiling Monitor and click **Start Monitoring**, as shown in Figure 10-13.

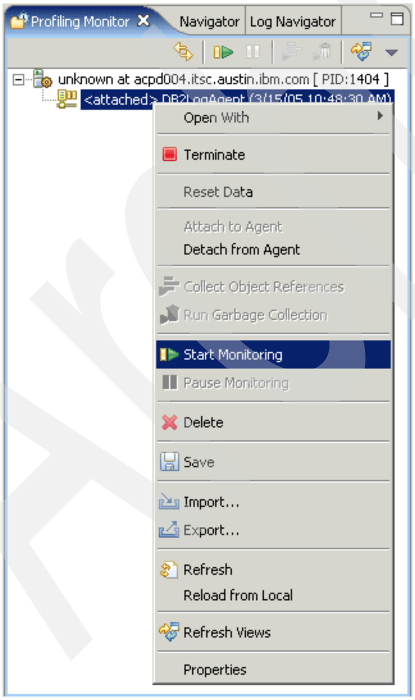


Figure 10-13 Start monitoring DB2LogAgent

6. Again, right-click the attached **DB2LogAgent** and select **Open With** → **Log View** to display the Common Base Events. The events can be viewed in the Log View, as shown in Figure 10-14.

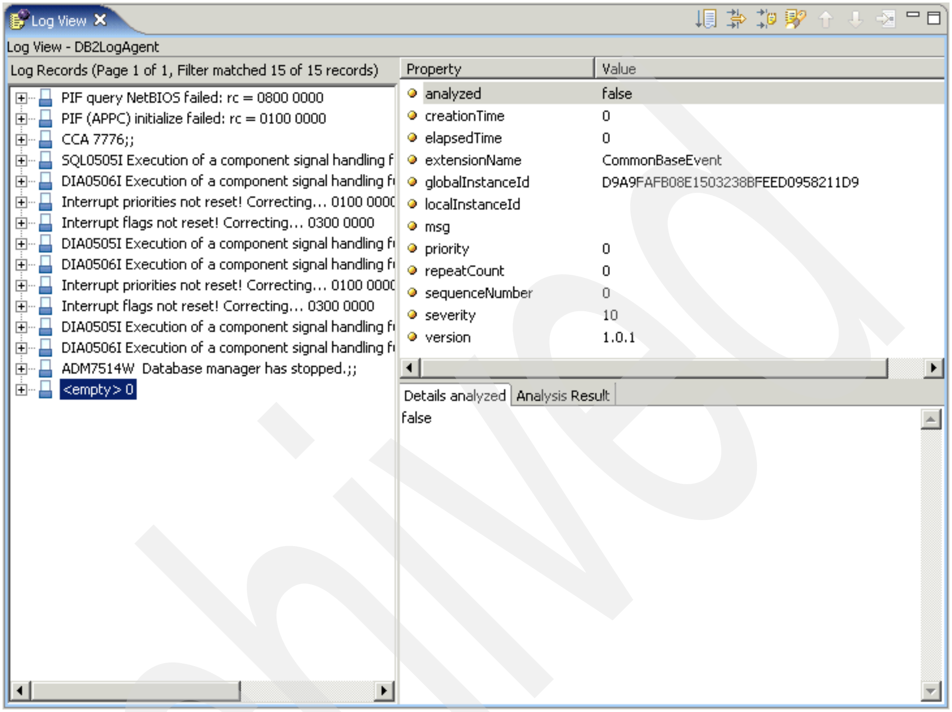


Figure 10-14 The events listed in Log View using the Profile Monitor

7. These steps must be repeated to attach the LTA to the RAC running on the application node and to start monitoring the information provided by the agent.

The log information provided by the monitored agents can be correlated and analyzed to identify the cause of a problem and to find a resolution. The correlation can be performed by clicking in the **Log Navigator** in the left pane of the LTA and creating a new correlation. For more information about correlation, refer 5.1.5, “Correlating the log files” on page 155.

A

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG246665`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246665.

Using the Web material

The additional Web material that accompanies this redbook includes the following file:

File name	*Description*
SG246665.zip	Zipped source code and executable files developed for the examples described in this redbook.

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	300 MB minimum
Operating system:	Microsoft Windows 2000 Server or Advanced Server
Processor:	2.0 GHz or higher
Memory:	1.0 GB minimum

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.

The resulting directory structure will be self-explanatory based on the description of the examples in this redbook. Refer to the *readme* files and follow the instructions provided in the corresponding chapters.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 382. Note that some of the documents referenced here may be available in softcopy only.

- *A Practical Guide to the IBM Autonomic Computing Toolkit,* SG24-6635
- *IBM Tivoli Monitoring Version 5.1.1: Creating Resource Models and Providers,* SG24-6900

Other publications

These publications are also relevant as further information sources:

- *Autonomic Computing Toolkit Developer’s Guide*, SC30-4083 (available with the IBM Autonomic Computing Toolkit)
- *Autonomic Management Engine Developer’s Guide* (available with the IBM Autonomic Computing Toolkit)
- *Generic Log Adapter Getting Started* (available with the IBM Autonomic Computing Toolkit)
- *Generic Log Adapter User’s Guide*, SC30-4079 (available with the Generic Log Adapter)
- *IBM Tivoli Monitoring Resource Model Builder User’s Guide*, SC32-1391
- *Log and Trace Analyzer for Autonomic Computing V3.0.0 Installation Guide* (available with the IBM Autonomic Computing Toolkit)

Online resources

These Web sites and URLs are also relevant as further information sources:

- GNU Operating System, Free Software Foundation

  http://www.gnu.org
- IBM Eclipse Test and Performance Tools Platform (TPTP) open-source project (formally Hyades)

  http://www.eclipse.org/tptp/index.html
- IBM Autonomic Computing Blueprint

  http://www.ibm.com/autonomic/pdfs/ACwpFinal.pdf
- IBM developerWorks autonomic computing page

  http://www.ibm.com/developerWorks/autonomic
- Jakarta regular expression package information

  http://jakarta.apache.org/regexp/apidocs
- JavaBeans Activation Framework information

  http://java.sun.com/products/javabeans/glasgow/jaf.html
- Java Mail API information

  http://java.sun.com/products/javamail/
- Microsoft Developer's Network resource for Windows Management Instrumentation Query Language (WQL)

  http://msdn.microsoft.com/library/default.asp
- Regular Expressions for Java

  http://www.cacas.org/java/gnu/regexp
- XML schema technical report

  http://www.w3.org/TR/2001/REC-xmlschema-0-20010502
- XML technical report

  http://www.w3.org/TR/2000/REC-xml-20001006

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks 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

IBM

Redbooks

Problem Determination Using Self-Managing Autonomic Technology

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Problem Determination Using Self-Managing Autonomic Technology

Add self-managing autonomic capabilities to components

Common Base Events best practices

Practical problem determination scenarios

This IBM Redbook provides practical information related to problem determination using IBM Self-Managing Autonomic Technology, with an emphasis on the components provided by the IBM Autonomic Computing Toolkit.

The primary objective of this book is to empower software developers to provide facilities to enable applications or other system components to participate in an autonomic environment and focuses on the problem determination capabilities provided by the IBM Autonomic Computing Toolkit and other Self-Managing Autonomic Technology.

The tools, components, and techniques used and demonstrated in this book provide the base for automating problem determination and self-healing of applications and components using Self-Managing Autonomic Technology. As with the technologies themselves, components of the IBM Autonomic Computing Toolkit will evolve as new capabilities and tools become available. Software developers are encouraged to check the latest releases of both IBM Autonomic Computing Toolkit and other tools provided by IBM at http://www.ibm.com/developerworks/autonomic.

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

SG24-6665-00 ISBN 073849111X