A Practical Guide to the IBM Autonomic Computing Toolkit

Understand the basics of autonomic computing
Add autonomic capabilities to components
Learn from practical examples

Bart Jacob
Richard Lanyon-Hogg
Devaprasad K Nadgir
Amr F Yassin

ibm.com/redbooks

This edition applies to Version 1, Release 2, of the IBM Autonomic Computing Toolkit, which is available from http://www.ibm.com/developerworks/autonomic.

© Copyright International Business Machines Corporation 2004. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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

**Figures** ................................................................. vii

**Notices** ................................................................ xi
**Trademarks** ................................................................. xii

**Preface** ........................................................................ xiii

- The team that wrote this redbook .................................... xiii
- Become a published author ............................................. xiv
- Comments welcome ...................................................... xv

**Part 1. Introduction** .................................................. 1

**Chapter 1. Introduction to Autonomic Computing** ........ 3

1.1 What is autonomic computing? ................................. 4
  1.1.1 Guiding principles ........................................... 4

1.2 Autonomic computing concepts ................................. 6
  1.2.1 Managed resources ........................................ 7
  1.2.2 Autonomic manager ....................................... 8
  1.2.3 Autonomic manager collaboration ..................... 9
  1.2.4 Autonomic manager knowledge ....................... 11
  1.2.5 Policies for autonomic managers ..................... 11

1.3 Evolving to autonomic computing ............................. 12
  1.3.1 Architectural levels of maturity ....................... 14

1.4 Value of autonomic computing .................................. 15
  1.4.1 Software vendors ....................................... 16
  1.4.2 Enterprises ............................................. 17

1.5 Architectural overview ............................................. 17
  1.5.1 Solution install ......................................... 18
  1.5.2 Common systems administration .................... 18
  1.5.3 Problem determination ................................ 19
  1.5.4 Autonomic monitoring ................................. 19
  1.5.5 Complex analysis ...................................... 20
  1.5.6 Policy-based management ............................ 21
  1.5.7 Heterogeneous workload management ............. 21
  1.5.8 Summary ............................................... 21

**Chapter 2. Introduction to the IBM Autonomic Computing Toolkit** .... 23

2.1 Key component areas .............................................. 24

2.2 Autonomic Computing Toolkit technologies ................. 25
6.2.1 Defining the ITSO_CBELog_Monitor resource model ........................ 93
6.3 Implementing the ITSO_CBELog_Monitor resource model .................... 94
  6.3.1 Managed Object Format (MOF) ........................................... 94
  6.3.2 Instrumentation Library Type (ILT) Java class ............................ 97
  6.3.3 Generating the resource model using Resource Model Builder ............ 111
  6.3.4 Understanding and editing the decision tree JavaScript ..................... 141
6.4 ITSO_CBESocket_Monitor resource model ....................................... 147
  6.4.1 Resource model design .................................................... 148
  6.4.2 Parameters for ITSO_CBESocket_Monitor resource model ................. 150
  6.4.3 Creating the ITSO_CBESocket_Monitor resource model ..................... 152
  6.4.4 Using the Resource Model Builder ....................................... 169
6.5 Contents of Resource Model Package ............................................. 180
6.6 Summary ...................................................................................... 182

Chapter 7. Autonomic Management Engine ............................................. 183
7.1 Introduction to Autonomic Management Engine .................................. 184
7.2 Writing an embedding application for AME ....................................... 186
  7.2.1 Designing the embedding application ....................................... 186
  7.2.2 Building and launching the AME embedding application ................. 197
7.3 Simple Agent Reference Application (SARA) ...................................... 201
7.4 Advanced topics ............................................................................ 202
  7.4.1 Adding actions and events to resource models ............................... 203
  7.4.2 Setting up the configuration files and environment ......................... 213
7.5 Summary ...................................................................................... 214

Part 3. Samples .................................................................................... 215

Chapter 8. Application examples .......................................................... 217
8.1 Sample application ........................................................................... 218
8.2 Maturity level 2 - Using GLA and LTA ............................................. 223
8.3 Maturity level 4 - Using GLA and AME ............................................. 228
8.4 Generating Common Base Events directly ........................................ 236
8.5 Adding sockets to communicate events to the autonomic manager .......... 243
8.6 Using a database to store received events .......................................... 247
8.7 Summary ...................................................................................... 248

Part 4. Appendixes ............................................................................... 249

Appendix A. Regular expressions ......................................................... 251
What is a regular expression? ............................................................... 252
How they work ..................................................................................... 252
  Regular expressions: most commonly used matching characters .............. 252
  Regular expression examples ............................................................ 254
# Figures

1-1 Autonomic computing attributes .................................................. 5
1-2 Autonomic computing control loop ................................................. 6
1-3 Sensors and effectors ................................................................. 7
1-4 IT systems can share resources to increase efficiency .................. 10
1-5 Direct and indirect management of DB 3 by six autonomic managers .... 11
1-6 Autonomic maturity index .......................................................... 14
1-7 Autonomic maturity levels for the IT infrastructure ...................... 15
2-1 Autonomic computing control loop ................................................. 26
4-1 Generating Common Base Events directly from the application ........ 45
4-2 Using the Generic Log Adapter to generate Common Base Events ...... 46
4-3 Autonomic manager and managed resource touchpoints ................. 49
4-4 Manager touchpoint assigns itself to the managed resource touchpoint 50
4-5 Sending a Common Base Event to the manager’s touchpoint .......... 51
5-1 Using the Generic Log Adapter to generate Common Base Events .... 54
5-2 Structural view of the adapter configuration file ......................... 55
5-3 Generic Log Adapter perspective ................................................. 58
5-4 Create a new simple project ....................................................... 59
5-5 Specify the project name and location ......................................... 60
5-6 Create a new Generic Log Adapter file ....................................... 61
5-7 Specify the Generic Log Adapter file name .................................. 62
5-8 Provide sample log file that will be used as input for the GLA ....... 62
5-9 The Generic Log Adapter adapter file opened using the Rule Editor .. 63
5-10 Add entries to the Common Base Event ...................................... 64
5-11 Adding an application to the sourceComponentId ....................... 65
5-12 The substitution rule for application attribute ........................... 65
5-13 Modifying the component information with role outputter ............ 67
5-14 Select the Context Instance in the adapter file and rerun adapter .. 68
5-15 Run, refresh, and step through the template log file .................... 69
5-16 Importing a log file ................................................................. 72
5-17 Identifying the log file, host, application, and project properties .... 73
5-18 The log file opened using the log view ..................................... 74
5-19 Specifying the Common Base Event information to be displayed ... 75
5-20 Choose the sorting criteria to sort the events ............................ 76
5-21 Importing a symptom database ................................................ 77
5-22 Analyzing a log file using the symptom database ....................... 78
5-23 Creating a new symptom database ........................................... 79
5-24 New symptom database .......................................................... 80
5-25 Adding a new symptom .......................................................... 81
5-26 Adding a solution to the symptom ........................................ 81
5-27 Add a new directive .......................................................... 82
6-1 Autonomic resource model components overview .................. 91
6-2 Overview of the Service Object API interaction ...................... 92
6-3 Directory listing for the working directory ............................. 112
6-4 Compiling the ILT class ...................................................... 113
6-5 Creating the jar file ITSO_CBELog_Monitor.jar ...................... 114
6-6 Welcome screen after invoking the Resource Model Builder .......... 115
6-7 Creating a new project ...................................................... 115
6-8 Creating a new Tivoli Management Project ............................ 116
6-9 Choosing the project name and the folder ............................. 116
6-10 Selecting the Basic Resource Model Wizard ......................... 117
6-11 Choosing the platforms for the resource model ...................... 118
6-12 Selecting the data source for the resource model through Wizard . 119
6-13 Selecting the CIM Datasource in cimv2 namespace ................ 120
6-14 Selecting the default name space ....................................... 121
6-15 Compiling the MOF (Managed Object Format) file .................. 122
6-16 Selecting the MOF file for compilation ............................... 123
6-17 Creating a new instance of the CIM class by compiling the MOF file 123
6-18 After compiling the .MOF file ........................................... 124
6-19 Creating ITSO_CBELog_Monitor data source ......................... 125
6-20 Selecting the properties to raise a trigger condition ............... 126
6-21 Selecting the limiting conditions for triggering Events .......... 127
6-22 Selecting the triggering condition - phase 1 ........................ 128
6-23 Selecting the triggering condition - phase 2 ........................ 128
6-24 Selecting the triggering condition - phase 3 ........................ 129
6-25 Selecting the parameters to log ....................................... 130
6-26 The data source wizard completion for the resource model ....... 131
6-27 Selecting the name for the resource model, events, and logging .... 132
6-28 Selecting the cycle time for monitoring the resource ............... 132
6-29 Storing the project workspace in .jrm file ........................... 133
6-30 Completing the New Tivoli Management Project wizard ............ 134
6-31 Workspace for ITSO_CBELog_Monitor resource model project .... 135
6-32 CIM Classes tab of the resource model ............................... 136
6-33 The Events tab in the Resource Model Builder ....................... 137
6-34 Thresholds tab in the resource model ................................ 138
6-35 Parameters sections for the resource model ......................... 139
6-36 Select the file and platforms supported for dependencies .......... 140
6-37 The Dependencies tab after adding all the dependencies .......... 140
6-38 Resource Model Package ................................................. 146
6-39 Generating various components of the resource model ............. 147
6-40 Flow chart for the socket resource model ........................... 149
6-41 Interaction for ITSO_CBESocket_Monitor resource model .......... 152
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-42</td>
<td>Compiling the ILT class ITSO_CBESocket_Monitor.java</td>
<td>169</td>
</tr>
<tr>
<td>6-43</td>
<td>Data source view for ITSO_CBESocket_Monitor resource model</td>
<td>170</td>
</tr>
<tr>
<td>6-44</td>
<td>Specifying the event triggering condition for the resource model</td>
<td>171</td>
</tr>
<tr>
<td>6-45</td>
<td>Furnishing the internal name, event, and logging names</td>
<td>172</td>
</tr>
<tr>
<td>6-46</td>
<td>Project front view of ITSO_CBESocket_Monitor resource model</td>
<td>173</td>
</tr>
<tr>
<td>6-47</td>
<td>Parameters for the ITSO_CBESocket_Monitor resource model</td>
<td>174</td>
</tr>
<tr>
<td>6-48</td>
<td>Dependencies tab in the resource model work space</td>
<td>175</td>
</tr>
<tr>
<td>7-1</td>
<td>Autonomic Management Engine hosting environment</td>
<td>185</td>
</tr>
<tr>
<td>7-2</td>
<td>Embedding application and related components</td>
<td>186</td>
</tr>
<tr>
<td>7-3</td>
<td>Building the AME embedding application</td>
<td>198</td>
</tr>
<tr>
<td>7-4</td>
<td>Application SARA shipped with AME tool</td>
<td>202</td>
</tr>
<tr>
<td>7-5</td>
<td>Events tab in the resource model</td>
<td>203</td>
</tr>
<tr>
<td>7-6</td>
<td>Adding Actions to the resource model</td>
<td>204</td>
</tr>
<tr>
<td>7-7</td>
<td>Adding the extra dependencies</td>
<td>210</td>
</tr>
<tr>
<td>7-8</td>
<td>Adding the CIM method for Actions</td>
<td>211</td>
</tr>
<tr>
<td>7-9</td>
<td>Adding the CIM method as Actions to the resource model</td>
<td>212</td>
</tr>
<tr>
<td>8-1</td>
<td>Scenario overview for maturity level 2</td>
<td>223</td>
</tr>
<tr>
<td>8-2</td>
<td>Using the GLA to converting log entries to Common Base Events</td>
<td>229</td>
</tr>
<tr>
<td>8-3</td>
<td>Generating Common Base Events directly</td>
<td>237</td>
</tr>
<tr>
<td>8-4</td>
<td>Using sockets between the manager touchpoint and the manager</td>
<td>244</td>
</tr>
<tr>
<td>8-5</td>
<td>Using a database for events to be used by the manager</td>
<td>247</td>
</tr>
</tbody>
</table>
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:

@server™
AIX®
Cloudscape™
developerWorks®
DB2®
e-business on demand™
IBM®
ibm.com®
Redbooks™
Redbooks (logo)™
Tivoli Enterprise™
Tivoli Enterprise Console®
Tivoli®
WebSphere®

The following terms are trademarks of other companies:

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

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.

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

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

This IBM® Redbook provides practical information related to using the IBM Autonomic Computing Toolkit. Using this toolkit, software developers can provide facilities to enable applications or other system components to participate in an autonomic environment.

The autonomic computing initiative from IBM defines a variety of capabilities that allow 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, multi-vendor, distributed environment are still evolving. The process of implementing and deploying systems capable of autonomic behavior is an evolutionary one.

IBM provides the Autonomic Computing Toolkit to assist software developers in enabling components, products, and solutions to become autonomic. The toolkit also provides the base for developers of management software to be able to manage entities that have been properly enabled. As with the technologies themselves, this toolkit will evolve as new capabilities and tools become available.

This redbook provides a view of the Autonomic Computing Toolkit as it exists today and describes how it can be used to enable autonomic capabilities. This redbook primarily focuses on the problem determination capabilities provided by the Autonomic Computing Toolkit. Other facilities, such as Solution Installation and Integrated Console support, to name two, will be covered in future books.

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.

Bart Jacob is a Senior Consulting IT Specialist at IBM Corp - International Technical Support Organization, Austin Center. He has 23 years of experience providing technical support across a variety of IBM products and technologies, including communications, object-oriented software development, and systems management. He has over 10 years of experience at the ITSO, where he has been writing IBM Redbooks™ and creating and teaching workshops around the world on a variety of topics. He holds a Masters degree in Numerical Analysis from Syracuse University.
Richard Lanyon-Hogg is a Principal Consultant, Networking & Systems Management, within IBM Global Services, EMEA. He has more than 23 years of experience in the IT and Telecommunications industry, with seven years across a variety of lead technical roles within IBM, including running the Microsoft® and Wireless consulting practices. Over the years, he has written extensively on Systems and Service Management and currently chairs the Middlesex University Wireless IT research centre Industrial Advisory Board. He is Fellow of the British Computer Society and holds a Masters degree in Computer Science.

Devaprasad K Nadgir is a Software Engineer at IBM Software Labs, Bangalore, India. He has been with IBM for four years, after receiving his Engineering degree from SJCE, Mysore, India. He has been involved with the IBM Java™ Development Kit on various platforms since then. His area of expertise includes J2EE, Web Services, and JVM internals.

Amr F Yassin is an IT Specialist with IBM Cairo Technology Development Center, Egypt. He has been with IBM since 2002. Currently, he provides technical support for the Learning Content Management System with Digital Rights Management project. He is a previous member of the XSL working group and XML Protocol working group at the W3C. Before joining IBM, Amr worked as a Research Member for Philips Research Center in New York. He holds an MSc degree in Computer Science from University of Nevada, Reno and BSc. in Communication and Electronics from Ain Shams University in Cairo. He has several patents related to XML technologies filed with the US patent office.

Thanks to the following people from the IBM Autonomic Computing organization for their support and assistance in developing this redbook:

Kesavan Kannan
Jim Cybrynski
Phil Johnson
Shajeer Mohammed

**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 Internet note to:
  redbook@us.ibm.com

► Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. JN9B Building 003 Internal Zip 2834
  11400 Burnet Road
  Austin, Texas 78758-3493
In the first part of this redbook, we introduce the reader to general concepts around autonomic computing and the IBM Autonomic Computing Toolkit. In addition, we provide a high-level view of the steps required to enable a component or application for autonomic behavior. This part lays the groundwork for the more technical content and examples provided throughout the rest of the book.

This part includes the following chapters:

- Chapter 1, “Introduction to Autonomic Computing” on page 3
- Chapter 2, “Introduction to the IBM Autonomic Computing Toolkit” on page 23
- Chapter 3, “Steps to enable a managed resource” on page 31
Introduction to Autonomic Computing

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 redbook is to introduce the reader to the IBM Autonomic Computing Toolkit and to use it to develop and implement an autonomic system.
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.

1.1.1 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, 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)
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 may 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 run-time 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. The following figure represents the control loop that is the core of the autonomic architecture.

![Figure 1-2 Autonomic computing control loop](image-url)
1.2.1 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 touchpoint; see Figure 1-3) that is available to 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.

![Diagram](image)

*Figure 1-3 Sensors and effectors*

Web services can (and will) be used to implement sensor-effector functions. By utilizing 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.
1.2.2 Autonomic manager

The autonomic manager is a component that implements the control loop. The architecture dissects the loop into four parts that share knowledge:

- 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 allow 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 may ask the monitor part to collect more or less information. There could also be situations where the monitor part may 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 implementors to leverage existing approaches, and support multiple binding techniques and multiple marshalling techniques.

Multiple levels of autonomic managers

As implied by the references above 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 may 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 allow 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 may 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 allows it to be implemented in a phased approach with benefits accruing at every level.

### 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 on page 10 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 may 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.
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.

Since 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.
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 like 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

While a detailed discussion of how policies would be implemented is outside the scope of this redbook, for architectural completeness, a brief explanation is provided.
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. 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:

**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 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 allows 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.
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:

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

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

Once 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

Below are a few examples of the results delivered by implementing autonomic computing solutions with self-management characteristics:

- **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 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 Architectural overview

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
1.5.1 Solution install

Today, there are a myriad of 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 install and configuration information in a consistent manner, autonomic managers can share the facilities as well as 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 allows users to manage solutions rather than managing individual systems or products. Administrative console functions range from setup and configuration to solution run-time 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 on both the kinds of data that must be collected as well as 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 legacy data sources (such as logs and traces) as well as data that is supplied using the standard format and categorization. To accommodate this legacy 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 as well as sensors to control data collection.


1.5.4 Autonomic monitoring

Autonomic monitoring is a capability that provides an extensible run-time environment for an autonomic manager to gather and filter data obtained through sensors. Autonomic managers can utilize 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: Allows 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%, then 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, then it may 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 2.2, “Autonomic Computing Toolkit technologies” on page 25 and Chapter 7, “Autonomic Management Engine” on page 183 for details on the Autonomic Management Engine, which is an example implementation using the IBM Autonomic Computing Toolkit.

### 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 run-time 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 autonomic computing components and phases of maturity in implementation.
Introduction to the IBM Autonomic Computing Toolkit

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. 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.

Though there are facilities to allow for development of autonomic managers, our primary focus in this redbook will be 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 autonomic capabilities. The content of the 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 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 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 Autonomic Computing Toolkit. The Autonomic Computing Toolkit includes a problem determination scenario performing self-healing as well as two automated installation scenarios performing self-configuring tasks.

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

### 2.2 Autonomic Computing Toolkit technologies

The technologies and tools contained in the current version of the Autonomic Computing Toolkit are intended to assist product developers develop autonomic capabilities in their products.

An example of an implementation of an autonomic manager is provided by the Autonomic Management Engine (AME). AME includes built-in representations of the four parts of the control loop (monitor, analyze, plan, and execute).

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 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 more information on Common Base Events, see the documentation for the Hyades project from the Eclipse organizations (http://www.eclipse.org/hyades/).

The Log and Trace Analyzer can be used to consume these 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.

Solution installation and deployment technologies are also part of the Toolkit that demonstrate another example of the interaction between an autonomic manager and managed resources to achieve a self-management behavior, in this case, self-configuring. The Solution Install components of the Toolkit will not be covered in this redbook.

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 multi-product customer IT environments. The IT environments can include IBM hardware and software, business partner applications, and/or client applications. Administrative console functions range from setup and configuration to run-time monitoring and control.

Looking at the control loop shown in Figure 2-1 on page 26, let us consider the technologies and capabilities that are 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 interface. In the case of unsolicited events, a current application or resource might generate its own log file. In this case, the Generic Log Adapter 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 4, “Common Base Events and touchpoints” on page 41.

The Generic Log Adapter (GLA) 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. The 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.
A single GLA run time 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.

The Adapter Rule Editor tool is used in conjunction with the GLA. It provides the tooling to create the specific parser rules that are used by the GLA at run time to create Common Base Event objects.

The Adapter Rule Builder is also available in the Autonomic Computing Toolkit GLA bundle.

Moving clockwise around the loop, the next component is monitoring. This component can either consume unsolicited Common Base Events or it can request specific information from the sensor. Resource models are the building blocks for monitoring the IT environment via automated best practices. They 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 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 at a managed resource, 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.

The autonomic manager consists of the monitoring, analysis, plan, and execute components. The Autonomic Computing Toolkit current ships a relatively simple black-box implementation of these components called Autonomic Management
Engine (AME). It 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 Autonomic Computing Toolkit in the AME bundle.

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, matches this incident based on predefined rules against the symptoms of an available symptom database, and returns 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 rather than using the rules-based parser provided by the GLA. Parsers written in Java code are commonly used when the individual log messages are very long and complex.

The Autonomic Computing Toolkit provides numerous parsers and rules for several existing IBM products.

### 2.3 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 redbook, we are not focusing on either of these facilities. Instead, we focus on facilities around problem management, with specific emphasis on enabling resources to be managed. To do this, though, requires some knowledge of the control loop defined by the autonomic computing architecture. We utilize an autonomic
manager embodied in the AME facility, resource models, tools, and APIs to generate Common Base Events.
Steps to enable a managed resource

In the previous chapters, we introduced the concepts of autonomic computing and the IBM Autonomic Computing Toolkit. Before we address the specifics of using the Autonomic Computing Toolkit and the technologies it includes, a brief overview is now given on the key steps involved in enabling a managed resource and developing a rudimentary control loop implementation to manage it.
3.1 Overview of a rudimentary control loop system

In the following, we describe one implementation of a rudimentary control loop system using some of the facilities of the Autonomic Computing Toolkit. The intention is to provide basic concepts and to position some of the tools we describe in much more detail throughout the rest of this redbook.

The following somewhat simplifies and summarizes what must be accomplished and provides a basis for the following discussion:

- Step 1. The managed resource (or an agent on its behalf) must generate information regarding its state or operational status. Events such as START, STOP, ERROR, and so on are logged in an application specific proprietary log file. For this scenario, let us assume the log file is text-based.

- Step 2. The information received from the managed resources must be translated into a standard format so that the receiver of the information (in this instance, the AME) can easily parse and act upon the information received. With this illustration, the Generic Log Adapter is used. The Generic Log Adapter Rule Builder is used to define the rules used by the GLA to adapt log entries to an XML format. The XML record produced is known as a Common Base Event. The parsing rules produced by the Adapter Rule Builder are contained in a file with an extension of ‘.adapter’.

- Step 3. The Managed Resource Touchpoint is the term used to describe the interface to the managed resource. The interface to the AME, which is the bundle of components containing the control loop logic, is referred to as the autonomic manager. In our scenario, the Common Base Event data structure containing the managed resource event passes through these two standard interfaces. The GLA run-time engine, using the ‘.adapter’ file, reads the managed resource proprietary log file. It is responsible for generating the Common Base Events and passing them to the autonomic manager.

- Step 4. The manager (in this case the AME) parses the received information (Common Base Events), performs some analysis, and, based on the results, executes the appropriate action. To enable this step, the developer will use the stand-alone tool (Resource Model Builder) to create and configure a resource model used by the AME. The script contains the information and data necessary to process the Common Base Events and undertake the analysis of the received data, plan and execute corrective actions.

The AME instantiates the CIM classes generated by the Resource Model Builder. These classes form part of the control loop kernel. The action(s) to be performed are formulated within the control loop. The sending of these actions should be accomplished through a standard interface that is independent of the managed resource itself. As noted earlier, the interface
that receives the action requests for the managed resource is called the Managed Resource Touchpoint (effector interface).

- Step 5. The Managed Resource Touchpoint object representing the managed resource receives the action request from the AME. Commands are executed or resource specific management interfaces are utilized to perform the specified action(s).

The IBM Autonomic Computing Toolkit provides facilities to assist with each of these steps. Each of these steps and the associated tools are explained in further detail in the following sections.

### 3.2 Generating information or events (Steps 1 and 2)

In existing environments, an application or other resource could write to its own unique log file, write to a more generic system log file, or generate some other sort of notification or event, such as an SNMP trap.

Though a custom management application designed to work with a specific resource could be written to consume such information (log entries or events), one of the values of the autonomic computing architecture is the ability for a conforming manager to be able to manage any conforming resource. Therefore, a standard way of representing operational status or other informational notifications must be used. This is the concept behind the Common Base Event as already mentioned and described in further detail in Chapter 4, “Common Base Events and touchpoints” on page 41.

The first step in developing an autonomic solution is to enable the generation of Common Base Events from managed resources. This can be performed in a number of ways, but generally the two ways are:

1. For an existing resource or application that cannot be easily modified, a resource specific adapter will need to be developed that will convert the information currently being generated by the resource into a Common Base Event structure.

2. For new resources/applications or those that can be modified, generate Common Base Events conforming to the defined standards. This will eliminate the need for adapters and their on-going maintenance and provide more efficient communication of events/information to any conforming management environment.

In case number one, the IBM Autonomic Computing Toolkit provides a facility to generate log adapters. This facility makes it easy to define and deploy adapters that can monitor resource specific or system log files and generate Common
Base Events. For more information on using the Generic Log Adapter, please see Chapter 5, “Generic Log Adapter and Log and Trace Analyzer” on page 53.

In case number two, again the IBM Autonomic Toolkit provides Java classes and interfaces required to programmatically generate Common Base Events. Application developers can take advantage of these APIs to create events that can be sent to an autonomic manager. See Chapter 4, “Common Base Events and touchpoints” on page 41 for a further information on the available programming interfaces.

### 3.3 Sending events (Step 3)

As previously indicated, there are several ways by which the XML structured Common Base Events can be sent to the AME. In the illustration provided above, the log entries are in textual format and written to a log file. These entries are in turn read by the CIM classes instantiated by the AME.

The interface that exists between the manager (AME) and the managed resource is known collectively as the manageability interface. The interface to the managed resource is referred to as the managed resource touchpoint.

Sensors and effectors use a set of interaction styles that formalize and define how an autonomic manager and its managed resources interact. Typical uses for these interaction styles are:

- **Sensor retrieve-state**: This is used by an autonomic manager to query state information from a managed resource. The autonomic manager asks for information and the managed resource synchronously returns it.

- **Sensor receive-notification**: A managed resource uses this style to asynchronously send event information to an autonomic manager.

- **Effector perform-operation**: Used by an autonomic manager to issue a command to a managed resource. This command is typically used to change states or properties.

- **Effector call-out-request**: Used by a managed resource to consult with an external entity before taking certain actions (for example, to learn what changes are allowed prior to changing values). The managed resource uses this interaction style to gather information from an autonomic manager before making a change.

The current release of the Autonomic Computing Toolkit only supports the receive-notification style of interaction, in which a managed resource touchpoint asynchronously sends Common Base Events to an autonomic manager. The interface that is exposed to provide this interface is the sendEvent() interface. It is
exposed by the AutonomicManagerTouchpointSupport class. Please refer to Chapter 7, “Autonomic Management Engine” on page 183 for an in depth view of the APIs.

3.4 Control loop (Step 4)

A resource model lets the developer describe how to identify, how to notify, and how to correct availability and performance problems of monitored resources in terms of:

- Quality parameters
- Events and recovery actions that must be triggered when the quality of the monitored resources decreases under the thresholds fixed by selected parameters

In addition, a resource model allows the developer to describe the data related to monitored resources that must be logged for any further analysis.


The IBM Resource Model Builder is a programming tool for creating, modifying, testing, and packaging resource models for use within an autonomic computing environment.

The Resource Model Builder enables the developer to specify events, parameters, and thresholds for existing and new resource models. Developers are able to identify problems in real time, notify other systems, and take autonomic, corrective actions.

The basic resource model created by the Resource Model Builder, and executed by the AME, contains the following components:

- Scripts: For example, the VisitTree() function contains the monitoring algorithm that is called after each cycle time has elapsed. Any further monitoring code required by the developer would need to be added within this function.
- MOF file: AME uses an information model standard called CIM-M12 to model resources to be monitored. CIM classes, represented by Model Object Format (MOF) files, define the format of data objects that are to be monitored. M12 Java providers invoke classes that assist in translating monitored objects into CIM classes.
Configuration file: Contains platform specific information, such as cycle time in seconds, platform architecture, and so on.

Message catalog: Contains various key parameters, such as the resource model name, which is used by the AME for identification purposes.

Java message catalog: Contains various descriptions that tie back to the Message catalog.

Resource model settings file: This XML component contains all the tags and corresponding data elements for the AME to execute the resource model.

The Autonomic Management Engine (AME) provides level 4 (adaptive) autonomic manager functionality in the Autonomic Computing Toolkit. AME has been implemented to use the Receive-Notification interaction style to consume Common Base Events. In programming terms, this means that AME has been provided with a sendEvent interface (which can be invoked from a Managed Resource Touchpoint) and that it has been designed to consume the Common Base Events that are passed in through this interface. The component within AME that is responsible for monitoring for incoming Common Base Events is called a resource model and the component responsible for driving changes to a managed resource is AME’s action manager.

The objects contained in the resource model are the elements that allow AME to consume Common Base Events and act upon their contents.

3.5 Management actions (Step 5)

Architecturally, the Managed Resource Touchpoint associated with a managed resource will be responsible for converting the action requests received from the manager to actual commands or calls to the managed resource to perform the appropriate action. At this stage, this simply implies that the developers of managed resources should ensure that they provide sufficient and robust interfaces and/or commands that will allow for their resource to be manageable in an autonomic environment. In the current version of the Autonomic Computing Toolkit, the effector interface of the Managed Resource Touchpoint is not implemented. Therefore, as will be seen in our examples throughout the rest of the redbook, non-standard interfaces and methods must be used as placeholders for invoking management operations.
3.6 Summary

In summary, developers looking to enable managed resources will need to:

1. Either use the interfaces available in the toolkit to generate Common Base Events directly or use the Generic Log Adapter tool to read the current log file and generate Common Base Events.

2. Send the Common Base Events to the Autonomic Manager using the ManagerTouchpointSupport object’s sendEvent() API. This can be done within the application or through the Outputter object within the Generic Log Adapter.

3. Use the provided Resource Model Builder to develop a Resource Model that will consume the Common Base Events, decide upon an appropriate action, and initiate the action.

More advanced functions to be included in the control loop, such as advanced analysis and policy handling, would have to be implemented using proprietary code today. Future versions of the Toolkit will provide facilities and APIs to simplify and standardize their use.
In this part of the redbook, we provide more technical detail on the use of the tools and APIs provided with the IBM Autonomic Computing Toolkit that provide the autonomic computing technologies and capabilities.

Chapter 4, “Common Base Events and touchpoints” on page 41 describes Common Base Events and how they can be generated and handled by touchpoints.

Chapter 5, “Generic Log Adapter and Log and Trace Analyzer” on page 53 covers the detailed usage of the Generic Log Adapter and the Log and Trace Analyzer tools.

Chapter 6, “Resource models” on page 85 describes the details of resource models and how to create them.

Chapter 7, “Autonomic Management Engine” on page 183 discusses the details of the AME environment and how to write a custom autonomic manager.
Common Base Events and touchpoints

This chapter describes Common Base Events, how they can be generated, and the touchpoint interfaces that can be used to pass them to an autonomic manager.
4.1 What is a Common Base Event?

Common Base Events are used to provide events in a structured way. The Common Base Event is emerging to be the *de facto* standard for reporting events in many enterprise applications. Common Base Events are defined as a part of the Eclipse/Hyades project. For more information, see [http://www.eclipse.org/hyades/](http://www.eclipse.org/hyades/).

Different kinds of events, such as logging, tracking, management, and business events, can be mapped to Common Base Events. Common Base Events are XML structures.

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
2. The identification of the component that is *affected* by the situation (Could be the same as the reporting component)
3. The *situation* itself

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

In the Common Base Event, the *situation* entry includes the event information in detail. It includes a collection of information and is flexible enough to include extra parameters to accommodate component specific entries.

For component specific attributes that are not included in the Common Base Event data model, it is recommended to use the ExtendedDataElement entry to describe it. For more information about the ExtendedDataElement and other Common Base Event elements check the latest version of the *Autonomic Computing Toolkit Developer's Guide*.

The Common Base Event structure contains the following elements:

- CommonBaseEvent
  - reporterComponentId
    - location (optional but recommended)
    - locationType
For every situation we create, there are some required subelements. These required subelements change from one situation to another. For example, StartSituation requires successDisposition and situationQualifier, while ReportSituation requires reportCategory.

For more detailed information about the Common Base Event elements and the recommended or restricted vocabulary for each element, please check the latest version of the *Autonomic Computing Toolkit Developer’s Guide*.

### 4.2 Common Base Event schema

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 appears in the Common Base Event.

There is a best practice vocabulary or restricted vocabulary for some Common Base Event elements. For example, the situationQualifier in the StartSituation has a restricted vocabulary for “START INITIATED”, “RESTART INITIATED”, or “START COMPLETED”.

For those unfamiliar with XML, it is described in detail at http://www.w3.org/TR/2000/REC-xml-20001006. An XML schema introduction can be found at http://www.w3.org/TR/2001/REC-xmlschema-0-20010502. Also, there are a lot of tutorials and other information about XML and XML schema available on the Web.

### 4.2.1 Example of a Common Base Event

Example 4-1 demonstrate what a Common Base Event looks like. It represents an internal successful start situation. By looking at this Common Base Event, we can see that the event was generated by a sub-component called ITSOSubComponent under a component called ITSOSimpleAppl. This application is located on a machine that has IP version 4 and its IP name is server1.itso.ibm.com.

**Example 4-1  Example of a Common Base Event**

```xml
<CommonBaseEvent creationTime="2004-01-16T18:14:27Z" globalInstanceId="N1FB97200C5B11D88000AB0D1D704CDE" msg="[Fri Jan 16 18:14:27 IST 2004] ITSO001I SampleManagedResource starting..." severity="20" version="1.0.1">
  <sourceComponentId application="ITSOSimpleAppl1" component="ITSO Simple App1" componentIdType="Name" location="server1.itso.ibm.com" locationType="IPV4" subComponent="ITSOSubComponent"/>
  <msgDataElement>
    <msgId>ITSO0001I</msgId>
  </msgDataElement>
  <situation categoryName="StartSituation">
    <situationType xsi:type="StartSituation" reasoningScope="INTERNAL" successDisposition="SUCCESSFUL" situationQualifier="START INITIATED"/>
  </situation>
</CommonBaseEvent>
```
4.3 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.events package. This Java package is available from the Eclipse Web site at http://www.eclipse.org/hyades and is shipped with the Autonomic Computing Toolkit as part of the Generic Log Adapter. 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 may also want an application to generate Common Base Events and send them directly to a resource manager. 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 4-1).

![Figure 4-1 Generating Common Base Events directly from the application](image)

The Common Base Event API provides an interface for the Common Base Event called ICommonBaseEvent. It is part of the package org.eclipse.hyades.logging.events. This package is provided by the Eclipse project at http://www.eclipse.org/hyades. CommonBaseEventImpl implements the ICommonBaseEvent interface, which also can be found in the same Java package.

The Autonomic Computing Toolkit provides an interface for the Autonomic Manager called ManagerTouchpointSupport that includes a method called sendEvent(). It can be used to send the generated Common Base Event from the managed resource to the autonomic manager. 4.4, “Generating Common Base Events using APIs” on page 46 describes, in more detail, how to generate Common Base Events natively from the application.
Generating Common Base Events using GLA

Currently, most enterprise applications provide log files for tracing the application behavior. The Generic Log Adapter provides a way to convert legacy log files into the Common Base Event format (see Figure 4-2). 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/editing adapter configuration files that specifically generate the output in a Common Base Event format as well as testing the adapter configuration file before deploying it.

Chapter 5, “Generic Log Adapter and Log and Trace Analyzer” on page 53 describes how to generate Common Base Events using the Generic Log Adapter in more detail.

4.4 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 available at http://www.eclipse.org/hyades for generating Common Base Events using this Java library. In our Java project, we need to include the library files that include the Common Base Event APIs library. These libraries are in the hlevents.jar file.

The Common Base Event API documentation can be found at <PDSzenario>\amtapi\lib\CBEjavadoc.zip, where <PDSzenario> is the installation directory of the Problem Determination Scenario that is provided by the Autonomic Computing Toolkit.
4.4.1 Create the Common Base Event

Using the ICommonBaseEvent interface, we can create a Common Base Event as follows.

1. Create an empty Common Base Event by using:
   
   ```java
   ICommonBaseEvent myCBE = (CommonBaseEventImpl)
   SimpleEventFactoryImpl.getInstance().createCommonBaseEvent();
   ```

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

4.4.2 Filling in the Common Base Event

1. Define the source component that experiences the problem using the IComponentIdentification interface:
   
   ```java
   IComponentIdentification sourceComp =
   EventItemsFactory.createIComponentIdentification(...);
   ```

2. Assign the source component to the Common Base Event:
   
   ```java
   myCBE.setSourceComponentId(sourceComp);
   ```

3. Define the reporter component that senses the problem using the IComponentIdentification interface:
   
   ```java
   IComponentIdentification reporterComp =
   EventItemsFactory.createIComponentIdentification(...);
   ```

4. Assign the reporter component to the Common Base Event:
   
   ```java
   myCBE.setReporterComponentId(reporterComp);
   ```

5. Fill in the message:
   
   ```java
   myCBE.setMsg("This is a testing CBE message");
   ```

6. Fill in the Severity:
   
   ```java
   myCBE.setSeverity((short) 20);
   ```

7. Fill in the creation time:
   
   ```java
   myCBE.setCreationTime("2004-01-26T13:00:00Z");
   ```

8. Fill in the rest of the Common Base Event information as needed.
4.4.3 Using sendEvent() to send the Common Base Event

Once the Common Base Event contains the needed information, it is ready to be sent to the assigned manager.

1. The manager’s touchpoint is typically assigned to the resource during its initialization. The assignManager() method is part of the interface for the managed resource touchpoint and is used, as its name implies, to assign the manager. This should only be done once during the managed resource initialization phase.

   managedResourceTouchPoint.assignManager(managerTouchPoint);

2. When the managed resource needs to send an event to the manager, it will use the sendEvent() method of the manager’s touchpoint:

   managerTouchPoint.sendEvent(myCBE);

3. The manager’s touch point will receive the Common Base Event and process it as needed by the autonomic manager.

   For an example of an application that generates Common Base Events natively, please see 8.4, “Generating Common Base Events directly” on page 236.

4.5 Managed resource and manager touchpoints

The communication between the autonomic manager and the managed resource is done through their APIs. The API of the managed resource is defined by the IManagedResourceTouchpoint interface, while the API for the manager is defined by IAutonomicManagerTouchpointSupport. They are part of the packages com.ibm.autonomic.resource and com.ibm.autonomic.manager respectively. These packages are provided as part of the toolkit in the amtapi.jar. This jar file can be found at <PDSenario>\amtapi\lib\amtapi.jar.

The touchpoints can use RMI to register themselves, discover managers/managed resources, and to invoke remote methods on the discovered touchpoints (see Figure 4-3 on page 49). The current version of the Toolkit only provides touchpoint support that works on a single machine. In future versions of the Autonomic Computing Toolkit, facilities such as RMI may provide the extensibility to use these touchpoints across systems.
The current version of the Autonomic Computing Toolkit provides the APIs needed to send Common Base Events from the managed resource to the autonomic manager. It does not provide the needed methods to send effector actions from the autonomic manager to the managed resource yet. These APIs/methods are expected in a future release of the Toolkit.

### 4.5.1 The interface IManagedResourceTouchpoint

The interface IManagedResourceTouchpoint is part of the package com.ibm.autonomic.resource, which is provided by the Autonomic Computing Toolkit. This interface defines the API used to contact the managed resource. In the current version of the Toolkit, only one method is defined to access the managed resource. This method is called `assignManager()`. It takes a reference to the autonomic manager touchpoint support as an input. It also throws an RMI remote exception:

```java
public void assignManager(IAutonomicManagerTouchpointSupport mgrTP) throws java.rmi.RemoteException
```
The assignManager() method must be implemented by the Managed Resource Touchpoint to assign the manager to the managed resource. This reference to the autonomic manager touchpoint will be used to send Common Base Events to the manager touchpoint later on.

An implementation of the IManagedResourceTouchpoint interface must implement the assignManager() method. A typical implementation would retain the reference to the manager's touchpoint and use it whenever it needed to communicate with the manager's touchpoint, for example, to send an event using the sendEvent() method (see Figure 4-4).

![Figure 4-4 Manager touchpoint assigns itself to the managed resource touchpoint](image)

4.5.2 The interface IAutonomicManagerTouchpointSupport

The interface IAutonomicManagerTouchpointSupport is part of the package com.ibm.autonomic.manager, which is provided by the Autonomic Computing Toolkit. This interface defines the API used to communicate with the autonomic manager.

In the current version of the Toolkit, only one method is defined to communicate with the autonomic manager. This method is called sendEvent(). It takes a common base event object as an input parameter. It also throws an RMI remote exception:

```java
public void sendEvent(ICommonBaseEvent cbe) throws java.rmi.RemoteException;
```

The sendEvent() method must be implemented by the Autonomic Manager Touchpoint Support to handle the incoming Common Base Events. It can pass this event information on to the autonomic manager in whatever way makes
sense for the particular implementation. For example, it could store an incoming Common Base Event into a database, convert it into an XML string and store it into a Common Base Event log, or any other action to provide the incoming Common Base Event to the autonomic manager (see Figure 4-5).

![Figure 4-5 Sending a Common Base Event to the manager's touchpoint](image)

**4.6 Summary**

Common Base Events represent system or application events. They can be represented in XML format or sent directly to the Autonomic Manager as an object. The application can generate the Common Base Events natively by using the provided APIs. For legacy applications that store their events in a log file, Common Base Events can be generated from these log files by the use of the Generic Log Adapter (GLA).

The generated Common Base Events will be sent to the Autonomic manager by the use of touchpoints. These touchpoints are the APIs defined for both the managed resource as well as the autonomic manager.
Chapter 5. Generic Log Adapter and Log and Trace Analyzer

In this chapter, we discuss how to convert legacy log files into Common Base Events that can be delivered to autonomic managers. This chapter also discusses how to analyze the generated Common Base Events using the Log and Trace Analyzer tool and how to filter and order the incoming Common Base Events. It also discusses how to use the symptom database to help in analyzing the generated Common Base Events and how to write a symptom database for a custom application.
5.1 Generic Log Adapter

The Generic Log Adapter is a tool provided by the Autonomic Computing Toolkit to convert log messages from text-based log file format to Common Base Event format. The generated Common Base Event messages can be consumed by different autonomic managers. Figure 5-1 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 Rule Editor, see the Generic Log Adapter User's Guide provided with the Generic Log Adapter.

![Figure 5-1 Using the Generic Log Adapter to generate Common Base Events](image)

5.1.1 Generic Log Adapter configuration file overview

The adapter configuration file consists of a number of contexts. Each context is responsible for describing the rules of how to convert a certain log file to Common Base Events. Accordingly, the adapter file is capable of handling more than one log file at the same time.

Each context in the adapter configuration file consists of a series of components that describe the conversion rules for the associated log file. It also contains other information needed to run the Generic Log Adapter outputter class, sensor class, input log file, and output information, such as the file name. Each context will run as a separate thread independent of the other contexts in the same adapter configuration file. Figure 5-2 on page 55 describe the structure of the adapter configuration file.
Figure 5-2  Structural view of the adapter configuration file

- 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:
  - SingleOSFileSensor
  - StaticParserSensor
  - AdaptorSensor

- 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
- 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:

a. **Global Processing Phase**

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

b. **Attribute Processing Phase**

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

The parser can “tokenize” the message into a series of name value pairs during the global processing phase, and then refer to these tokens by name during the attribute processing phase.

The package that contains these parsers is

org.eclipse.hyades.logging.adapter.parsers. There is only one parser provided by the Generic Log Adapter:

- 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.

The package that contains these formatters is

org.eclipse.hyades.logging.adapter.formatters. There is only one formatter provided by the Generic Log Adapter:

- 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 three outputters provided by the Generic Log Adapter:

- `CBEFileOutputter`
- `CBEStdoutOutputter`
- `CBELogOutputter`

### 5.1.2 Creating the adapter configuration file

The Generic Log Adapter provides a plug-in to Eclipse. This plug-in is called the *Adapter Rule Editor*. The adapter configuration file can be easily edited using the Adapter Rule Editor that runs under Eclipse. Below are some steps to generate a new adapter configuration file from scratch using the Eclipse Generic Log Adapter Rule Editor.

1. Start Eclipse by running `<GLA_Install>\Dev\eclipse\eclipse.exe`, where `<GLA_Install>` is the install directory of the Generic Log Adapter. Typically, it will be `C:\Program Files\IBM\AutonomicComputingToolkit\GLA`.

2. Open a Generic Log Adapter Perspective by selecting `Window → Open Perspective → Other → Generic Log Adapter`. A window similar to Figure 5-3 on page 58 should appear.
3. Create a new project to work with by selecting **File → New → Other → Simple → Project**, and click **Next**. A window similar to Figure 5-4 on page 59 should appear.
4. Give it a project name, such as SimpleGLAProject, and click **Finish**. We can specify a certain location on the drive where we want to create our project or use the default location (see Figure 5-5 on page 60).
Figure 5-5 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 click **Next**. A window similar to Figure 5-6 on page 61 should appear.
6. Choose the project we want to include this adapter configuration file in, give it a name with a “.adapter” extension, and click **Next**, as in Figure 5-7 on page 62.
7. Specify the template log file to use as an input log file for the Generic Log Adapter engine, as in Figure 5-8, and click the Finish button (not shown here).
8. Expand the adapter configuration file hierarchy to be able to see the full tree view, as in Figure 5-9.

![Figure 5-9 The Generic Log Adapter adapter file opened using the Rule Editor](image)

9. Now we are ready to use the Adapter Rule Editor to edit our adapter configuration file.

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, please refer to 5.1.6, “Writing a custom outputter” on page 70.

### 5.1.3 Editing the adapter configuration file

Now we have an empty adapter configuration file. In the following steps, we will describe how to write the substitute rules to convert the log file message to the different Common Base Event entries.

1. Open the project in Eclipse.

2. Open the adapter configuration file in the Rule Editor by double-clicking on the adapter configuration file.

3. Expand the adapter tree.
4. Select **Adapter** → **Context Instance** → **Parser** → **CommonBaseEvent**, which defines the structure of the generated Common Base Event.

5. To add the needed entries in the Common Base Event, select **Adapter** → **Context Instance** → **Parser** → **CommonBaseEvent**, right-click, and select **Add** → **sourceComponentId**, as in Figure 5-10.

![Figure 5-10 Add entries to the Common Base Event](image)

6. To add an attribute to the sourceComponentId, select **sourceComponentId**, right-click, and select **Add** → **application**, as in Figure 5-11 on page 65.
7. To specify the substitution rule of what will be written in the application attribute, select **application**, right-click, and select **Add -> Substitution Rule**.

8. Now define the substitution rule to define what will be written in the generated Common Base Event.

9. 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 in Figure 5-12.

![Figure 5-11 Adding an application to the sourceComponentId](image)

**Figure 5-11 Adding an application to the sourceComponentId**

<table>
<thead>
<tr>
<th>Match</th>
<th><code>^(.*)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Positions</td>
<td></td>
</tr>
<tr>
<td>Substitute</td>
<td><strong>ITSOSimpleApp1</strong></td>
</tr>
</tbody>
</table>

![Figure 5-12 The substitution rule for application attribute](image)

**Figure 5-12 The substitution rule for application attribute**
10. We need to repeat the previous steps to add the following subelements:

- **CommonBaseEvent**
  - `reporterComponentID`
    - location (optional but recommended)
    - `locationType`
    - `component`
    - `subComponent`
    - `componentIdType`
    - `componentType`
  - `sourceComponentID`
    - location (optional but recommended)
    - `locationType`
    - `component`
    - `subComponent`
    - `componentIdType`
    - `componentType`
  - `msgDataElement` (optional but recommended)
  - `creationTime`

- **Situation**
  - `SituationType`
  - `reasoningScope`
  - `CategoryName`

For every situation we create, there are some required subelements. These required subelements change from one situation to another. For example, `StartSituation` requires `successDisposition` and `situationQualifier`, while `ReportSituation` requires `reportCategory`.

11. Repeat the previous steps to generate any other optional Common Base Event elements that are needed. For more information about the different Common Base Event elements, check the *Autonomic Computing Toolkit Developer’s Guide*.

12. Identify the Outputter class we are using to handle the generated Common Base Events. This can be done by selecting **Adapter → Contexts → Context → Component** (that has a component Role = Outputter) and changing the **Executable Class**. The default outputter class is
org.eclipse.hyades.logging.adapter.outputters.CBELogOutputter, as shown in Figure 5-13.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hyades Logging Agent Outputter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Default Logging Agent</td>
</tr>
<tr>
<td>Executable Class</td>
<td>org.eclipse.hyades.logging.adapter.outputters.CBELogOutputter</td>
</tr>
<tr>
<td>Implementation Creation Date</td>
<td>Fri Jan 30 10:14:48 CST 2004</td>
</tr>
<tr>
<td>Implementation Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Implementation Version Description</td>
<td>First Implementation</td>
</tr>
<tr>
<td>Logging Level</td>
<td>99</td>
</tr>
<tr>
<td>Role</td>
<td>Outputter</td>
</tr>
<tr>
<td>Role Creation Date</td>
<td>Fri Jan 30 10:14:48 CST 2004</td>
</tr>
<tr>
<td>Role Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Role Version Description</td>
<td>First Role Version</td>
</tr>
</tbody>
</table>

Figure 5-13   Modifying the component information with role outputter

13. The previous step needs to be repeated per need for the other components with different roles, such as Sensor, Extractor, Parser, Formatter, Message Filter, Common Base Event Filter, and Outputter.

5.1.4 Testing the adapter configuration file

The Generic Log Adapter perspective within Eclipse enables us to test the adapter configuration file we are creating in an easy way. Import the sampleLTA.adapter file into a simple Eclipse project. This sample adapter file is provided with the redbook as an example of an adapter configuration file. For more details, refer to 8.2, “Maturity level 2 - Using GLA and LTA” on page 223. 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, and click on the **Rerun adapter** icon, as in Figure 5-14 on page 68.
4. Now we can run, refresh, and step through the log template file and see the generated Common Base Event in the Outputter Result tab, as in Figure 5-15 on page 69.

Figure 5-14 Select the Context Instance in the adapter file and rerun adapter
5. If the result is incorrect, change the substitution rule, save the file, and re-run the adapter.

**Tip:** In some cases, with the early version of the Toolkit we were using, we found that we had to wait for a short while to let the environment refresh and use the new adapter file. In some cases, we had to restart Eclipse for the changes to take effect.

### 5.1.5 Deploying and running the adapter configuration file

After testing the adapter configuration file, now we are ready to deploy it. We need to make sure 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.
Deploying the adapter configuration file is done by copying it to the machine that will run the Generic Log Adapter engine. We can use a command line argument or we can write a small Java class to start an instance of the Generic Log Adapter engine. We also need to pass the adapter configuration file to the created instance of the Generic Log Adapter engine.

**Starting the Generic Log Adapter engine from a command line**

The org.eclipse.hyades.logging.adapter.Adapter class comes with its own main method to start the Generic Log Adapter engine. The Generic Log Adapter engine can be started through a batch file or shell script file located at `<GLAinstall>in\gla.bat` (or gla.sh), where `<GLAinstall>` is the location of the Generic Log Adapter installation.

**Writing a Java class to start the Generic Log Adapter engine**

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 8-6 on page 226 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:

- xmlParserAPIs.jar
- jakarta-oro-2.0.7.jar
- xerces.jar
- hlevents.jar
- hlcore.jar
- hglao.jar

### 5.1.6 Writing a custom outputter

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 XML format to a log. A custom outputter could, for example, be used to deliver a Common Base Event directly to an autonomic manager through its ManagerTouchpointSupport object.

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(). It receives an array of objects, usually CommonBaseEvent objects, and processes them. The signature of the method is:

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

In the case of using touchpoints to communicate with the autonomic manager, it will call the sendEvent() method and pass CommonBaseEvent object. Example 5-1 is an extract of a sample custom outputter called SampleOutputter; for a full example of a custom outputter, please refer to Example 8-7 on page 230.

**Example 5-1  Sample outputter class**

```java
import org.eclipse.hyades.logging.adapter.*;
import org.eclipse.hyades.logging.adapter.impl.*;

public class SampleOutputter extends ProcessUnit implements IOutputter
{
    public SampleOutputter()
    {
        super();
        ....
    }

    public Object[] processEventItems(Object[] msgs)
    {
        ........
        for (int i = 0; i < msgs.length; i++)
        {
            if (msgs[i] != null)
            {
                managerTPSupport1.sendEvent((ICommonBaseEvent) msgs[i]);
            }
        }
        return (ICommonBaseEvent[])processedmsgs;
    }
}
```
5.2 Log and Trace Analyzer

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 the administrator to analyze the correlation between events created by different products, link them, and look for possible solutions.

5.2.1 Adding the log file to Log and Trace Analyzer

The log file of an application can be analyzed using the Log and Trace Analyzer.

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 it a name, and click Finish.

3. Import the log file by selecting File → Import → Log File, and click Next, as in Figure 5-16.

4. Click Add to add the log file.

5. Choose the log file type (for example, IBM WebSphere® Application Server trace log).
6. In the different tabs, identify the log file properties, for example, host server, application details, and destination project (see Figure 5-17). Make sure to fill in all the requested fields. As an example, while importing an activity log file for WebSphere, it asks for the WebSphere Application Server installation directory. The requested information differs from one type of log file to another.

![Add Log File](image)

**Figure 5-17  Identifying the log file, host, application, and project properties**

7. Click **Finish** to add the log file to the project.

8. Now the log file will be opened in the log view, as in Figure 5-18 on page 74.
Figure 5-18  The log file opened using the log view

9. The events will be color coded to identify the risk level of the event.

5.2.2 Analyze log events created by one product

The Log and Trace Analyzer can be used to analyze the log events of an application. This can be used to identify the different levels of severity and their relationship.

1. Open the project in Eclipse.
2. Open the log file in the log view, as described before.
3. Click on the event to see the event information in the right panel.
4. After clicking on the File Properties icon, modify which Common Base Event elements are to be displayed or not, as in Figure 5-19 on page 75.
5. Different sorting mechanisms can be used to sort the events, for example, severity, time, ID, priority, and so on (see Figure 5-20 on page 76). This can be done by clicking the **Sort Log Records** button.
5.2.3 Correlate log events from different products

In the case of using multiple applications that interact with each other, we can correlate their log files to be able to analyze the events better, for example, WebSphere Application Server and DB2®.

1. Open the project in Eclipse.
2. Open the Profiling and Logging perspective and create a Simple Project.
3. Import both log files into the Log and Trace Analyzer by selecting File → Import → Log File.
4. Right-click on the log file and select Open With → Log Thread Interactions or Log Interactions.
5. Choose the Correlation Schema between the files, for example, Default Time.
6. The files will open in the Sequence Diagram view.
7. Events happening at the same time, either in the same thread or different threads, will have an arrow linking them.
5.2.4 Using the symptom database to analyze log files

The symptom database is an XML file that describes different events and suggested solutions for problems. The symptom database is simply an XML file that contains symptoms with associated explanations of that symptom and directions for how to recover from an error. Symptoms can be errors or event messages expressed in plain text format, as in log files.

Symptom databases are used to analyze the events after converting them to Common Base Events and propose actions to be taken if needed. It is used mainly for assisting the administrators in finding the problem(s) in the system. It can be considered as the analyze part of a maturity level 2 autonomic system.

Importing a symptom database
1. Open the project in Eclipse.
1. Open the log file using the log view.
2. Select File → Import → Symptom Database File. A window similar to Figure 5-21 should appear.
3. Choose the symptom database that matches the log file.
4. If a symptom database has been previously imported, it will ask for confirmation to override the old one.
5. Click **Finish**.

6. It may be better to use the **Remote host** option to get the latest version of the symptom database.

### Using the symptom database to analyze a log file

1. Start Eclipse.

2. Import the log file in a simple project and open it in the Log View.

3. Right-click on any event and select **Analyze** to analyze this event only, or select **Analyze All** to analyze all the events.

4. Select **Default Log Analyzer**.

5. The events with available analysis will have a different background color.

6. By clicking on an event that has an analysis, we get the detailed analysis for that event and suggested action for it (see Figure 5-22).

7. Order the events by time, severity, or other attributes, and analyze the log file using the symptom database to solve the error in this application or among different applications interacting together.

8. Search for a certain event using the event attributes.

---

**Figure 5-22** Analyzing a log file using the symptom database
Creating a new symptom database for a custom application

We may need to create a custom symptom database to analyze an application if there is no symptom database already defined for it, for example, this may be required for a home grown application. After creating the custom symptom database, we can use it to analyze the custom application log file.

As mentioned before, the symptom database is simply an XML file that contains symptoms with associated explanations of that symptom and directives for how to recover from an error. The editor keeps the database information in a file with the extension “trcdxmi”. After editing the database, it can be exported to an XML file and then copied to the serving URL.

To create a new symptom database:

1. Open the project in Eclipse.
2. Select File → New → Other → Profiling and Logging → Symptom Database.
3. Click Next. A window similar to Figure 5-23 should appear.
4. Choose the Project to add the symptom database to and name it.
5. Click Finish.

Figure 5-23 Creating a new symptom database

6. Use the Profiling and Logging perspective.
7. Now we have an empty symptom database.
8. The symptom database will have two tabs: Overview and Details. To edit it, go to the **Details** view. A window similar to Figure 5-24 should appear.

![New symptom database](image)

*Figure 5-24 New symptom database*

9. Now, identify the location (URL) where this database will be published and a simple description about it.

10. To add a symptom to the database, right-click on the root of the element database and choose **Add Symptom**.

11. For the symptom, identify a symptom name and the matching pattern to find this symptom, as in Figure 5-25 on page 81.
12. Right-click on the symptom and choose Add Solution.

13. Type the description of the solution and the solution name will be inferred automatically from it, as in Figure 5-26.

14. Right-click on the solution and choose Add Directive. A window similar to Figure 5-27 on page 82 should appear.
15. Identify the directive description to recover from this error.

16. We can repeat the previous steps to include all the expected events from this application.

17. The generated symptom database can be exported to an XML file (as in Example 5-2) by choosing the database, right-clicking, and selecting Export → Symptom database file.

18. Click Next.

19. Choose the XML file name to export to and the current database in the editor (usually with extension *.trcdbcxmi).

Example 5-2  Generated symptom database file in XML format

```xml
<?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 Tuesday, February 03, 2004 at 4:41:35:0562 PM CST -->

<symptomDatabase>
    <runtime id="Runtime_0" name="itsocustom" symptomUrl="http://localhost/xyz"
        localExternalFileLocation=""/>

        <!-- Symptoms: -->
        <symptom id="Symptom_0" description="ITSO945E_Symptom"
            solutions="Solution_0">
            <matchPattern name="null" value="ITSO945E"/>
        </symptom>

        <!-- Solutions: -->
        <solution id="Solution_0" description="Restart Database Connection"
            directives="Directive_0"/>

        <!-- Directives: -->
        <directive id="Directive_0" description="Data connection restart
Resolution: Restart database or the data source |--- ITSO Multi Version ---|"
            directiveString=""/>
    </runtime>
</symptomDatabase>
20. Now we have the symptom database, we can copy the XML file to the location specified by the URL and make it available for different application administrators.

5.3 Summary

We can use the Generic Log Adapter to convert existing application log files to Common Base Events. The Generic Log Adapter engine takes an adapter configuration file as an input. The adapter configuration file specifies the input log file and the outputer used to handle the generated Common Base Events. It also defines the rules of what will be written in the different fields of the generated Common Base Event. It uses regular expressions to define these rules.

The Log and Trace Analyzer can be used to analyze the generated Common Base Events and sort them. It also can be integrated with the symptom database to provide guidance to the system administrator for recommended actions.
Resource models

This chapter discusses the essential concepts required to build resource models using the Resource Model Builder provided with the IBM Autonomic Computing Toolkit. It is recommended that the reader be familiar with the concepts of the Tivoli Monitoring architecture. The redbook *IBM Tivoli Monitoring Version 5.1.1: Creating Resource Models and Providers*, SG24-6900 is an excellent reference.

This chapter provides a brief overview of resource model concepts and shows practical examples of building resource models for use by an autonomic management engine. The examples given here are intended to be educational and should be customized to suit specific IT solution needs. The examples are also not intended to explain the syntax, use of, or nuances of the Java and JavaScript languages, the Java development environment, or object oriented development terminology and methods.
6.1 Introduction to resource models

Resource models provide the infrastructure and definition of how a resource should be monitored. In this perspective, resource models encapsulate the information about the resource, the data to be collected from the resource, and the logic or monitoring algorithm to analyze the collected data.

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 like CIM (Common Information Model)/WMI (Windows® Management Instrumentation). These interfaces are called providers. Resource models act as a critical bridge between sensors and effectors in the autonomic architecture. In fact, resource models can define and provide the logic for the monitoring, analysis, plan, and execute phases of the autonomic control loop. Though, as their name implies, they typically provide these facilities for specific resources. Higher level facilities may be used to provide these services for solutions or complex combinations of individual resources.

A typical autonomic resource model should contain the following.

- Common Information Model (CIM) classes
- Cycles
- Events
- Thresholds
- Parameters
- Logging information
- Decision Tree Script
- Dependencies

The Resource Model Builder provided with the Autonomic Computing Toolkit is used to build such a resource model. The following sections provide a brief overview of each of these components.

Common Information Model (CIM) classes

A set of Common Information Model classes is sometimes referred to as the dynamic model, which forms a very important part of the resource model.

The CIM classes are responsible for:

- The description of the resources to be monitored
- The available properties of each resource
- An implementation to collect the data to be monitored
The CIM class contains the properties of the resource that is monitored. A CIM class is associated with a provider that retrieves the information from the resource. On Windows operating systems, one can use the Windows Management Instrumentation (WMI) providers as a facility to collect data from a monitored resource that is provided by the underlying operating system. The Resource Model Builder uses the WMI on Windows operating systems for monitoring specific resources for which WMI provides an implementation to collect data. Autonomic resource models can use Java-based providers for collecting data from any resource. The advantage here is a common implementation across all platforms.

There can be synchronous or asynchronous ways of collecting data. The implementation for such a mechanism of data collection should be implemented in the CIM classes.

The data collected can be filtered using the WQL (WMI Query Language) in the case of Windows.

More information on WQ, can be found on the Microsoft Developers Network library at http://msdn.microsoft.com/library/default.asp.

Java CIM classes should implement standard interfaces provided by the Instrumentation Library Type interfaces. For more information on these interfaces’ Instrumentation, please refer to the IBM Tivoli Monitoring Resource Model Builder User's Guide v1.1.0, SC32-1391.

These Instrumentation Library Type (ILT) interfaces must be implemented by any CIM class we will use. The ILT interface is understood by the Autonomic Management Engine (AME) that provides a platform to execute CIM classes and scripts provided by the resource model. The AME is discussed in detail in Chapter 7, “Autonomic Management Engine” on page 183.

**Note:** As the CIM classes implement the ILT (Instrumentation Library Type) interfaces, CIM classes are also referred to as ILT classes.

The Autonomic Management Engine contains a CIM Object Manager (CIMOM). CIMOM loads CIM classes and invokes various ILT interface methods, which in turn gather data from the monitored resources.

**Cycles**
This parameter of the resource model represents the regular interval to gather data from the monitored resource. As an example, one can monitor the disk space every 60 seconds.
Events
An event is a change in the state of a resource. Events may trigger actions if they are enabled. An indication is generated when the state of a given resource meets a specific criteria defined.

These indications can be configured to generate an event when a set of indications aggregate. The cycles during which the indication is generated are called occurrences. The cycles during which no indication is generated are called holes. One can define an acceptable number of occurrence and hole combinations. For example, if through five cycles, the CPU utilization is over a particular threshold (occurs) four times with one hole (utilization not at the threshold), then raise an event.

Events are associated with the following terms:

- Attributes: A string or a numeric value that is collected by the Resource Model. An event will have relevant attributes associated with it. A key attribute is supposed to uniquely identify the event to which it refers to and should be used in the aggregation process for the event. As an example, an event indicating that a process is consuming more than 80% of the CPU cycle may have attributes like processID, the total memory consumed, the total processor time consumed, the process name, and so on. The processID can be a key attribute.

- Actions: This represents one or more recovery procedures for the events that are generated. Actions can be configured to automatically trigger when a specific event occurs. Actions can be of the following types.
  - Programs: External programs can be run as actions with parameters.
  - CIM Methods: Methods provided by CIM or the WMI can be used with parameters to invoke relevant actions.

Custom actions can be configured for these events, which are discussed in 7.4.1, “Adding actions and events to resource models” on page 203.

Thresholds
A monitoring algorithm is associated with a threshold, which is a numeric value used to represent an acceptable limit for a resource. As an example, if the CPU percentage of usage crosses 90, then an event is triggered. The 90% becomes the threshold for the CPU consumed parameter.

Parameters
Parameters represent the instances that you want to monitor in the resource. They can be a list of strings or numeric values. They can be accessed through the CIM classes to act as a limiting factor for one or more of the resources that
are monitored. The current version of the Resource Model Builder supports the following types of parameters:

- Boolean list
- Choice list
- Numeric list
- String list

**Logging information**
The resource model uses a local database mechanism to store the data to be logged. The logged data may refer to the parameters, events, thresholds, actions, or any other such trace statements used for debugging purposes. By default, logging is disabled in the Resource Model Builder.

Further, the logging function consists of the following elements:

- **Context**: A general problem to which the resource monitoring activity relates to, for example, the percentage usage of CPU on a heavily loaded server.
- **Resource**: The resource about which data is being logged, in relation to the defined context, such as a specific processor. A single context can contain more than one resource.
- **Properties**: Specific attributes of the defined resource, for example, information about the processor. For each resource, multiple properties can be specified, which can be both numeric and string values. For the specified properties, key properties must be defined that clearly identify the instance of a resource.

**Dependencies**
The CIM classes implement the actual data collection logic and may sometimes depend on other utilities and classes to perform its operations.

**Decision tree script**
The Resource Model Builder automatically generates a script called the decision tree script. The decision tree script contains the template logic and variables specified when a resource model is created and configured.

The decision tree script contains functions for executing the logic required for autonomic solutions. The following are the main functions and critical parts of the decision tree script that builds the infrastructure for taking logical actions.

**Main()**
Created when the resource model decision tree script is written in Visual Basic. As the implementation is in Visual Basic, this becomes a part of the Windows resource model implementation. This method is also created if using JavaScript,
but is not visible at run time. Visual Basic scripts require Main because ActiveX controls need the code to run the script. In the examples used in this redbook, we always use JavaScript to provide cross platform portability of resource models. Therefore, the Main() function is not used.

**SetDefaultConfiguration()**

Initializes the object based on the settings defined through events, thresholds, parameters, and actions. SetDefaultConfiguration() is called only once, when the resource model starts. Additional initialization code can be written at the end of this function as needed.

**Init()**

Called when the settings defined in SetDefaultConfiguration are replaced with values specified in the monitoring profiles. The new values replace the previous settings. This function can be modified with additional initialization code if required for the resource model.

**VisitTree()**

VisitTree() is the method containing the monitoring algorithm and is called at the beginning of each cycle. VisitTree() should check various key parameters and should implement the algorithm for achieving an autonomic solution. It should process the collected data according to thresholds and parameters settings, so as to take necessary actions.

In an autonomic resource model, this method becomes critical as the Common Base Events have to be analyzed here and appropriate actions based on these Common Base Events initiated by this script.
Figure 6-1 describes various parts of a resource model.

The decision tree script accesses various information provided by the CIM classes (also known as ILT classes) for its decision algorithm. The information provided by the CIM classes can be accessed through a standard API called the Service Object API. The Service Object API provides information about various attributes of the monitored resource and can be used by the decision tree script for implementing autonomic decision algorithms.

Figure 6-2 on page 92 shows an overview of interaction between the resource model, CIM classes, decision tree scripts, and the Service Object API.
6.2 Designing an autonomic resource model

Before creating the resource model, one should have a clear description of the resource to be monitored, the mechanism for monitoring, thresholds, parameters, dependencies, cycle time, events, logic for taking actions on these events, and so on.

In an autonomic environment, as discussed in previous chapters, Common Base Events will be generated and these Common Base Events act as input to the resource models, that is, the resources generating or providing the Common Base Events will be the resource that will be monitored. The Common Base
Events will have enough information about the situation to enable the resource model to act.

As an example, let us design a resource model to monitor a log file consisting of Common Base Events called CBEOut.log. In this simplified case, we are assuming that Common Base Events have been generate and written to a log through a mechanism such as the Generic Log Adapter described in Chapter 5, “Generic Log Adapter and Log and Trace Analyzer” on page 53.

6.2.1 Defining the ITSO_CBELog_Monitor resource model

The CIM classes constitute an important role in implementing the standard interfaces required for the Autonomic Management Engine to provide an environment for the resource models to execute the decision tree script.

The logic for the resource model is explained below:

1. The resource that we will monitor is the log file consisting of the Common Base Events.

2. We will gather information about all the Common Base Events that typically lie between <CommonBaseEvent> and </CommonBaseEvent> tags. We can do this by searching the log file and matching lines with a regular expression. The following regular expression can be used to get the relevant information between <CommonBaseEvent> and </CommonBaseEvent> tags for a specific event type (containing the string ITSO314C):

\(<\text{CommonBaseEvent} +.*\text{ITSO314C}.*?\text{</CommonBaseEvent}>\)

For simplicity, we search for the occurrences of a particular message ID string and monitor the count as to how many times this Common Base Event has occurred. ITSO314C is generated by a sample application that refers to a typical critical error message. The number of times that this message occurs in CBEOut.log is monitored.

3. The CIM implementation (or Instrumentation Library Type (ILT) implementation) should keep track of the number of lines that match the regular expression. The ILT implementation is responsible for returning this value to the requestor.

4. The number of occurrences of such events can be requested through the Service Object interface (Svc) in the VisitTree() function by the getProperty() method.

5. If the number of matching lines exceeds five, we know that a critical threshold has been crossed and will generate an event and take appropriate actions.

As explained earlier, the CIM class must implement an ILT interface. This interface is provided in the com.tivoli.dmunix.ep.touchpoint.base package. The
example discussed in the next section implements an ILT interface, as described by the M12Instrumentation, to search for a pattern represented by a regular expression in the log Common Base Event log file (CBEOut.log) and returns the number of matching lines through the Service Object API calls.

6.3 Implementing the ITSO_CBELog_Monitor resource model

The implementation of this resource model includes a variety of steps, including the use of different tools and writing code to build the required resource model. Chapter 5, “Engineering a Java ILT”, of IBM Tivoli Monitoring Version 5.1.1: Creating Resource Models and Providers, SG24-6900 can be referred to for detailed definitions. The following steps are followed to develop the resource model to monitor a resource called CBEOut.log:

1. The Managed Object Format (MOF) file ITSO_CBELog_Monitor.mof was written in a text editor.

2. Resource Model Builder is used to compile this .MOF file and create the resource model with various attributes.

3. The ILT implementation (Provider class embedded in the ILT implementation) ITSO_CBELog_Monitor.java is written using a text editor.

4. The Resource Model Builder (RMB) workbench is used to build the Resource Model Package and to edit the decision tree script.

6.3.1 Managed Object Format (MOF)

The management information is explained in a language based on the Interface Definition Language (IDL) called the Managed Object Format. The MOF syntax is a way to describe object definitions in a text format. It establishes the syntax for writing definitions, classes, associations, properties, references, methods, and instance declarations with their associated qualifier.

Comments in MOF files are permitted and advised. Example 6-1 contains the comments used for the ITSO_CBELog_Monitor.mof file.

Example 6-1  Commenting in MOF files

// Filename : ITSO_CBELog_Monitor.mof
// Description:
// Gives textual definition for the ITSO_CBELog_Monitor CIM class.
Description
Description is one of the CIM qualifiers used to describe the function of the data
source created by this MOF file. The work bench uses this for displaying the
information about the CIM data source created from this qualifier from the MOF
file. The description section of the MOF file used in ITSO_CBELog_Monitor is
shown in Example 6-2.

Example 6-2  Description of the MOF file
[
Description ("ITSO_CBELog_Monitor: This sample resource model is created to"
  " educate the readers about the resource model concepts."
  " This resource model monitors a file name <Key parameter> "
  " FileName </Key parameter>"
  " and return the number of occurrences of the regular expression"
  " given as a parameter to the resource model. This resource "
  " model exhibits only a part "
  " of the functions provided by M12 qualifiers."),
]

Provider
The provider qualifier is used within the ITSO_CBELog_Monitor MOF file to
define the library or the class that will implement an instance interface to the
sample provider class. The M12JavaProvider is embedded in the Resource
Model Builder. Example 6-3 gives information about the declaration for the Tivoli
M12 Java provider.

Example 6-3  Provider qualifier for the MOF file
provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider"),

M12_Instrumentation
This is the qualifier that describes the Instrumentation Library Type class. It
should not only give the class property, but also the operations supported by this
class. In this example, the ITSO_CBELog_Monitor class is the implementation of
the ILTInterface and supports ENUMERATION and GET operations. This tells
the class loader which class it should associate for the enumeration function of
the instrumentation. Additionally, the added GET operation of the class is used to
retrieve the return values from the class. Example 6-4 can be used to describe
the class that is in the com.ibm.tivoli.monitoring package.

Example 6-4  Provider information in MOF file

M12_Instrumentation {
   "Java.com.ibm.tivoli.monitoring.ITSO_CBELog_Monitor | | ENUM",
   "Java.com.ibm.tivoli.monitoring.ITSO_CBELog_Monitor | | GET"}

Key parameter
The key property qualifier (Example 6-5) is used within the
ITSO_CBELog_Monitor class to define the file name. The parameters passed to
the resource model will include the property ITSO_CBELog_Model_FileName,
which will contain the file name. This property will be initialized when the instance
of this resource model is created by the Autonomic Management Engine
environment.

Example 6-5  Key parameters and attributes

//Attributes for the I LT implementation
   [ Description("The Filename property is provided by
ITS_CBELog_Model_FileName as"
      " a parameter to the Resource Model."),
      key ]
   string FileName;

Data source class
The data source class will be looked up by the Resource Model Builder and is
defined as Example 6-6. The class name with its associated qualifiers becomes
the data source in the resource model.

Example 6-6  Data source information in MOF files

class ITSO_CBELog_Monitor
{
    //Attributes for the I LT implementation
       [ Description("The Filename property is provided by
ITS_CBELog_Model_FileName as"
           " a parameter to the Resource Model."),
       key ]
    string FileName;
    sint32 Num; // Attribute, Integer returning the number of occurrences.
};
6.3.2 Instrumentation Library Type (ILT) Java class

The ILT class ITSO_CBELog_Monitor is the implementation of the ILTInterface. This acts as the bridge between the monitoring services layer and the execution layer, which is explained in Chapter 7, “Autonomic Management Engine” on page 183.

The M12 public operations handle the actual implementation defined by the ILTInterface. In the ILT class of this resource model, we implement various methods required by the interface. We also have defined a few other private or service methods that are used to gather data. These methods are added only for the sake of clarity and simplicity. Numerous ways of implementing these classes are possible and can be extended to support multiple instances.

In the example shown, the provider class is merged within the ILT implementation. In a more real world example, the provider class can contain the service routines and can be instantiated by the ILT class. Various parts of the ILT Java interface are explained in the following sections.

Comments

The comments in Example 6-7 are self explanatory and are documented as shown. They form a very useful part of the source code and are a suggested practice.

Example 6-7 Commenting in the ILT class

/////////////////////////////////////////////////////////////////////////////////////////
//
// File name : ITSO_CBELog_Monitor.java
//
// Description : This java program implements the ILTInterface to search for
//               a regular expression in the given <code>FileName</code>. The value of
//               the integer and the offset is passed in <code>Num</code> which returns
//               the total number of occurrences of the search pattern.
//
// File History :
//
// Date                       Author                   Description
// 01/23/04                   ITSO Redbook team        Initial file

/////////////////////////////////////////////////////////////////////////////////////////
Package
The package naming standard is followed as per the guidelines suggested by Tivoli Monitoring Services. The package name is shown as in Example 6-8. This naming convention ensures that all the required classes of the ILT are handled by the same management engine class loader for consistency reasons.

Example 6-8  Package information for the ILT class

```java
// Package details.
package com.ibm.tivoli.monitoring;
```

**Note:** It is not mandatory that all of the ILT classes be defined as part of the same package mentioned above. We recommend that the class loader used to load these ILT classes will have similar access rights as that of the other classes in the Tivoli Monitoring infrastructure. However, we tested this recommendation by using other package names and did not run into any issues in our limited testing.

Imports
The ILT class depends on various packages and classes for gathering and providing information. The dependencies are imported as shown in Example 6-9.

Example 6-9  Imports for the ILT class

```java
import com.tivoli.dmunix.ep.touchpoint.base.*;
import com.tivoli.javautils.Trace;
import java.util.Enumeration;
import java.util.*;
import gnu.regexp.*;
import java.lang.*;
import java.io.*;
```

The first import, `com.tivoli.dmunix.ep.touchpoint.base.*`, refers to the ILT Interface classes and the ILT public operations. For further documentation on these classes, you can refer to *IBM Tivoli Monitoring Workbench User's Guide Version 5.1.1*, SH19-4571. The classes pertaining to this import are found in `dm_m12.jar`.

This jar file can be found in the `.\eclipse\plugins\com.ibm.tivoli.monitoring_1.2.0\Lcfnew\Tmw2k\Unix\bin` directory after installing the Resource Model Builder provided with the Autonomic Toolkit.

The second import, `com.tivoli.javautils.Trace`, is used for writing relevant information to log files. Log entries are written to the log files, as indicated by the
properties file. The location of the log files are defined by the logging utility used by the manager. These properties file are discussed in Chapter 7, “Autonomic Management Engine” on page 183.

The java.regexp.* package is imported to allow us to use methods related to regular expressions for matching the lines read from the file to the regular expression given. The classes are embedded in gnu-regexp-1.1.4.jar.

The java.util Enumeration is imported to support the Enumeration class, which can be used to create a series of elements. See the enumerateInstances() method javadoc for further details.

The java.util.* provides several useful classes, such as Vectors, Hashtables, and so on, and are used in parsing the parameters used by the Autonomic Management Engine.

The java.io.* is imported to support file IO operations.

**Instrumentation Library Type implementation**
The ITSO_CBELog_Monitor class implements the ILTInterface class, as demonstrated in Example 6-10.

*Example 6-10   The ILT class implementing the ILTInterface*

```java
public class ITSO_CBELog_Monitor implements ILTInterface {
}
```

**Constructor**
The constructor used in this example is very simple and initiates the tracing class. This is called whenever the ILT class is loaded into memory and can be extended to perform any other initialization required by the ILT.

*Example 6-11   Constructor for the ILT class*

```java
public ITSO_CBELog_Monitor() {
    trace = Trace.getTraceInstance();
}
```

**ILT public methods**
The Instrumentation Library Type class implements the public methods declared in the ILTInterface class. For our purposes, all of the public methods are described here, though some with trivial implementations. The ILT need not really have an implementation in the body of all of these methods. Some functions may not be supported by the resource model and can return null.
values. Adding logging to these methods is a helpful way of debugging the resource model, as the interaction between various autonomic tools increases the complexity of the environment. The methods in the ILTInterface implementation are explained below.

**getProperty()**

This method, as shown in Example 6-12, returns the value for the M12_Instrumentation qualifier property, as mentioned in the MOF file. The management engine can request any property through the Service Object interface calls (Svc.GetStrProperty, Svc.GetNumProperty, and so on) on every cycle to decide on the actions to be taken. As an example, the Num property declared in the MOF file can be obtained during each cycle through this interface public method.

*Example 6-12  getProperty method in the ILT class*

```java
public String getProperty( M12ObjectIdentity m12objectidentity,
        String sPropertyName,
        String sMapping,
        ParameterSet parameterset )
    throws M12Exception
{
    try
    {
        // Variable definitions
        StringBuffer stringbuffer = new StringBuffer();
        Vector vector = new Vector();

        // Get the parameters of the patterns from the caller.
        // This is mentioned in the PARAMETERS section of the resource model
        // If the parameters are null, then we don't have anything to search.
        // We return a null in that case.
        Enumeration enumeration = parameterset.parametersNames();
        Vector vParam = (Vector)parameterset.getParam("CBE_search_PATTERN");
        if(vParam == null)
        {
            trace.log(1, "ITSO_CBELog_Monitor", "ERROR: getProperty: no parameters found");
            return null;
        }

        // convert the vector into String
```
String asCBE_search_Pattern[] = new String[vParam.size()];

for(int iIndex = 0; iIndex < vParam.size(); iIndex++)
{
    // Get the string array element from the vector element.
    asCBE_search_Pattern[iIndex] = (String)vParam.elementAt(iIndex);
    trace.log( 3, "ITSO_CBELog_Monitor",
    "getProperty: asCBE_search_Pattern[" + iIndex + "] = " + asCBE_search_Pattern[iIndex]);
}

M12IdentityElement m12IdentityElement =
m12ObjectIdentity.getScopingPath()[0];
M12PropertySet m12PropertySet = m12IdentityElement.getIdentity();


String sKey = m12PropertySet.getProperty("FileName");
trace.log(3, "ITSO_CBELog_Monitor",
    "getProperty: called for FileName from m12propertyset " + sKey);

String sValue = getAttribute(sKey, sPropertyName, asCBE_search_Pattern);
trace.log(3, "ITSO_CBELog_Monitor",
    "getAttribute: requested " + sPropertyName + " = " + sValue);
return sValue;
}

catch(Exception exception)
getMultipleProperties()

This method, shown in Example 6-13, supports handling of multiple properties. The implementation is similar to getProperty(), but is valuable when a number of properties are requested through the M12PropertySet.

Example 6-13  getMultipleProperties in the ILT class

```java
public M12PropertySet getMultipleProperties(
    M12ObjectIdentity m12objectidentity,
    Vector vPropertyList,
    String sMapping,
    ParameterSet parameterset)
    throws M12Exception
{
    try {
        // Variable Declaration.
        StringBuffer stringbuffer = new StringBuffer();
        Vector vector = new Vector();

        // As documented in getProperty method

        Enumeration enumeration = parameterset.parametersNames();
        Vector vParam = (Vector)parameterset.getParam("CBE_search_Pattern");
        if(vParam == null) {
            trace.log( 1, "ITSO_CBELog_Monitor", "ERROR: getMultipleProperties: no
            parameters found");
            return null;
        }

        // Process the ITSO_CBELog_Monitor_FileName parameter.
        String asCBE_search_Pattern[] = new String[vParam.size()];
        for(int iIndex = 0; iIndex < vParam.size(); iIndex++) {
            // Get the string array element from the vector element.
            asCBE_search_Pattern[iIndex] = (String)vParam.elementAt(iIndex);
            trace.log( 3, "ITSO_CBELog_Monitor", "getmultipleProperties: asITSO_CBELog_Monitor_Pattern[" + iIndex + "] = " + asCBE_search_Pattern[iIndex]);
        }
    }
```
M12IdentityElement m12identityElement =
    m12objectIdentity.getScopingPath()[0];
M12PropertySet m12propertyset = m12identityElement.getIdentity();

// Determine which instance is being checked by the
// Management Layer.
String sKey = m12propertyset.getProperty("FileName");
M12PropertySet m12propertyset1 = new M12PropertySet();
trace.log(3, "ITSO_CBELog_Monitor",
    "getMultipleProperties: called for FileName from m12propertyset1 "+ sKey);
for(int i = 0; i < vPropertyList.size(); i++)
{
    // Get the property being checked by the Management Layer.
    String sProperty = (String)vPropertyList.elementAt(i);

    // Get the current attribute values.
    String sValue = getAttribute(sKey, sProperty, asCBE_search_Pattern );
    m12propertyset1.setProperty(sProperty, sValue);
}

trace.log(3, "ITSO_CBELog_Monitor",
    "getMultipleProperty: returning "+
    WriteBuffer(m12propertyset1));
return m12propertyset1;
}

} catch(Exception exception)
{
    trace.exception("ITSO_CBELog_Monitor", "getMultipleProperties", exception);
    M12Exception m12exception = new M12Exception(exception);
    throw m12exception;
}


enumerateInstances()

This method (Example 6-14) supports the enumeration of instances of the CIM
class. This function will determine how many instances of the monitored object
exists and handles the appropriate configurations for all instances. The return
value is an Enumeration of M12ObjectIdentity objects that identify all the
instances belonging to the class specified.

Example 6-14  enumerateInstances in the ILT class

public Enumeration enumerateInstances( M12ClassPath m12classpath,
    String sMapping,
    ParameterSet parameterset )
    throws M12Exception
{
try {
    // Variable Declaration.
    StringBuffer stringbuffer = new StringBuffer();
    Vector vector = new Vector();

    // Get the parameters from the caller.
    // We are obtaining the parameters prior to declaring instances so
    // we can configure an instance for each parameter. These parameters
    // are declared in the PARAMETERS section of the resource model.
    //
    Enumeration enumeration = parameterset.parametersNames();
    Vector vParam = (Vector)parameterset.getParam("ITSO_CBELog_Monitor_FileName");

    if(vParam == null) {
        trace.log(1, "ITSO_CBELog_Monitor", "ERROR: enumerateInstances: no parameters found");
        return null;
    }

    // Process the ITSO_CBELog_Monitor_FileName parameter to a string.
    String asITSOArrayGui[] = new String[vParam.size()];
    for(int iIndex = 0; iIndex < vParam.size(); iIndex++) {
        // Get the string array element from the vector element.
        asITSOArrayGui[iIndex] = (String)vParam.elementAt(iIndex);
        trace.log(3, "ITSO_CBELog_Monitor",
            "enumerateInstances: asITSOArrayGui[" + iIndex + "] = " + asITSOArrayGui[iIndex]);
    }

    String aiInst[] = getInstances(asITSOArrayGui);

    if(aiInst == null) {
        trace.log(1, "ITSO_CBELog_Monitor", 

"enumerateInstances: no instances found");
return null;
}

for(int jIndex = 0; jIndex < aiInst.length; jIndex++)
{
trace.log(3, "ITSO_CBELog_Monitor", "enumerateInstances: aiInst[" +
        jIndex + "] = " + aiInst[jIndex]);
M12PropertySet m12propertyset = new M12PropertySet();
m12propertyset.setProperty("FileName",
        aiInst[jIndex]);
M12IdentityElement m12identityelement =
        new M12IdentityElement(m12classpath.getClassName(),
        m12classpath.getNameSpace(),
        m12propertyset);
vector.add(new M12ObjectIdentity(new M12IdentityElement[
        {m12identityelement}]));
}

return vector.elements();
}
catch(Exception exception)
{
trace.exception("ITSO_CBELog_Monitor", "enumerateInstances", exception);
M12Exception m12exception = new M12Exception(exception);
throw m12exception;
}

---

**setProperty()**

This method is part of the ILT interface. However, for our example implementation, it is not used, as we are not setting any properties. In this example, we can simply return a null value. We can, however, add an instruction to create a trace entry if any request for setProperty() is made by the management engine through the trace statement, as shown in Example 6-15.

*Example 6-15  setProperty in the ILT class*

```java
public String setProperty( M12ObjectIdentity m12objectidentity,
        String sName,
        String sValue,
        String sMapping,
        ParameterSet parameterset )
        throws M12Exception
{
        trace.log(3, "ITSO_CBELog_Monitor", "setProperty: " +
                "sName=" + sName +
                ", sValue=" + sValue +
                ");
```
invokeMethod()  
This method provides interfaces that can be used to remotely execute methods. The resource model explained in this example does not support this method and hence returns a null. If these interfaces are implemented, this method can be invoked. It is included in this example for explanatory purposes. See 7.4, “Advanced topics” on page 202 for a discussion of the InvokeMethod() implementation.

create()  
This method is not supported at this time, and only tracing is included for educational and debugging purposes.

destroy()  
This method is not supported at this time, and only tracing is included for educational and debugging purposes.

Service or private methods  
These methods are private methods we added for better understanding and simplification of the code. The ILT public methods use these methods to gather the information required. From a utility perspective, these methods are more maintainable and extendable.

getAttribute()  
This method is used by the get(Property) public method to fetch the value of the property during each processing cycle. The global values are updated for future iterations in this method. The code flow and the logic is documented as shown in Example 6-16.

Example 6-16  getAttribute function of the ILT class

private String getAttribute( String sValue,
       String sProperty,
       String asCBE_search_Pattern[] )
       throws Exception
{

   // Check for each property whose value is requested ...

}
if( (sProperty.equals("Num")))
{
    String offset = "-1";
    int r = 0;

    ///////////////////////////////////////////////////////////////////////////
    // Retrieve the offset value for the filename from the offsetarray
    // hashtable. If the offsetarray doesn't have the appropriate offset,
    // assign "-1" to the offset variable and keep it in the offsetarray.
    // //////////////////////////////////////////////////////////////////////////

    String v = (String)offsetarray.get(sValue);
    if ( v == null ) {
        offset = "-1";
        offsetarray.put(sValue, offset);
    } else {
        offset = v;
    }

    ///////////////////////////////////////////////////////////////////////////
    // Run the getITSO_CBELog_MonitorVar_Num() and assign the result
    // to the intVar[] array.
    // intVar[] = { Num, offset }
    // We will then store these values in the global hash table so that
    // they can be retrieved next time for further processing.
    // The first integer contains the number of occurrences ( Num ) and
    // the second integer ( intVar[1] ) contains the offset of the previous
    // iteration while reading the log file.
    // //////////////////////////////////////////////////////////////////////////

    int intVar[] = getITSO_CBELog_MonitorVar_Num(sValue, asCBE_search_Pattern,
                                                offset);

    // Keep the new offset value in the offsetarray hashtable.
    offsetarray.put(sValue, String.valueOf(intVar[1]));

    // Trace
    trace.log( 3, "ITSO_CBELog_Monitor", "getAttribute result offset" +
                intVar[1] + " EventNum = " + intVar[0] + " FileName = " + sValue);
    trace.log( 3, "ITSO_CBELog_Monitor", "global array: offset" +
              (String)offsetarray.get(sValue));
// Return the EventNum value.
return String.valueOf(intVar[0]);
}

if( (sProperty.equals("FileName")))
{
// Return the FileName value
return sValue;
} else {
    throw new Exception(sProperty + ": unknown property");
}

convertToString()

This is a method used to convert the M12PropertySet to a string buffer and return the string buffer so that it can be used for logging purposes. This function enumerates over the elements of the m12propertyset object, converts them into strings, and appends a string buffer with the property and value equivalents. The code snippet is shown in Example 6-17. Another method called getInstances() is used for a similar reason.

Example 6-17  convertToString of the ILT class

private String convertToString( M12PropertySet m12propertyset )
{
    Enumeration enumeration = m12propertyset.propertyNames();
    StringBuffer stringbuffer = new StringBuffer();
    String sElement;
    String sProperty;
    for(; enumeration.hasMoreElements();
        stringbuffer.append(sElement + " = " + sProperty + " ")
    {
        sElement = (String)enumeration.nextElement();
        sProperty = m12propertyset.getProperty(sElement);
    }
    return stringbuffer.toString();
}

getITSO_CBLog_MonitorVar_Num()

This method returns the value of the total number of lines that match the given search string and the last line number in an integer array. This last line number is used for counting the number of matching lines in each cycle of the getProperty function call.
The code flow can be explained by the following steps:

1. Check for the property whose value is required. If it is the file name, return the file name, as the file name is the key property and will be available with all the instances.

2. If the property requested is Num, then check for the global hash table “offsetarray” for the file name and the last line read.

3. Construct a regular expression object.

4. Open the file given as a parameter to the method and construct the line reader object.

5. Read a line into a string.

6. For all strings required to be searched, check if the string read from the file matches the regular expression.

7. If the regular expression given has a matching expression in the read line, increment the count based on the current line number. If the previously read line number is less than the current line number, then increment the count_above; if the previous read line number is greater than the current line number, then increment the count_below variable.

8. Repeat steps 5 to 7 for all lines.

9. Check if the offset passed to this method is -1. If it is -1, then we are just starting the process. Hence, return the count as 0 and the last line number in the array.

10. If offset is not -1, then return the total number of counts and the last line number as the return array.

11. The total count. If the file is edited in between the cycle time, then the current last line after reading the file will be less than the offset (last line number read in the previous cycle). In this case, pass the actual count (that is, count_above) for accurate results.

Example 6-18 is the code snippet for the above function.

```
Example 6-18  getITSO_CBELog_MonitorVar_Num method of the ILT class

private  synchronized int[] getITSO_CBELog_MonitorVar_Num( String i,
        String as[],
        String offset )
{
    // initializations ..
    int count_above = 0;
    int count_below = 0;
    int obj = 0;
    int lastline;
    int[] retarray = new int[2];
```
try {

    RE re = null;        // Declare the Regular Expression
    REMatch mat;

    String sl = offset;  // offset = retrieve offset(i);

    trace.log(3, "ITSO_CBELog_Monitor","offset returned is : "+offset);

    // Create a FileReader object
    FileReader fr = new FileReader(i);
    BufferedReader br = new BufferedReader(fr);
    LineNumberReader lr = new LineNumberReader(fr);

    // Read strings in the file
    String line;
    while((line = lr.readLine()) != null) {

        for(int iIndex = 0; iIndex < as.length; iIndex++) {

            // Evaluate the line if the search pattern matches...
            try {
                re = new RE(as[iIndex]);
            } catch (REException e) {
                break;
            }

            Integer tmp = Integer.valueOf(sl);
            obj = tmp.intValue();

            // Pattern match
            mat = re.getMatch(line);
            if (mat != null) {

                if (obj > lr.getLineNumber()) {
                    count_above++;  // Increase the counter older than the offset
                } else {
                    count_below++;  // Increase the counter newer than the offset
                }
            }
        }
    } // End of while loop

    // Store the last line number for this iteration ..
lastline = lr.getLineNumber();

trace.log(3, "ITSO_CBELog_Monitor","Lastline parameter is"+lastline);

// Close the FileReader object
fr.close();

if ( offset == "-1" ) {   // If the offset is ",-1", this is the first time.
    retarray[0] = 0 ;
    retarray[1] = lastline ;
} else {
    if ( lastline < obj ) {   // If the lastline is lower than offset, the file was
        retarray[0] = count_above;       // Returns the c1 value which is the
        retarray[1] = lastline;         // counter older than offset
    } else {                  // If lastline is upper than offset,
        retarray[0] = count_below+count_above;       // returns the c1+c2 value
        retarray[1] = lastline;         // which is the total counter
    }
}
}catch (Exception e) {
    trace.log(3,"ITSO_CBELog_Monitor","ERROR: getITSO_CBELog_MonitorVar_Num:
Exception"+e);
}

trace.log(3, "ITSO_CBELog_Monitor","count_above, count_below are "+count_above+
" "+count_below);
return retarray;

6.3.3 Generating the resource model using Resource Model Builder

We now have the ITSO_CBELog_Monitor.mof file and the ILT class
ITSO_CBELog_Monitor.java file, which implements the ILT Interface.

Now we need to build a resource model, which is nothing but the encapsulation
of various configuration files, logic scripting, parameter settings, and dependency
files. In this section, we discuss how to generate the Resource Model Package,
which can be run in the Autonomic Management Engine environment. The .mof
and the ILT class used can be found in the additional materials provided with this
Compiling the ILT class

The ILT class ITSO_CBELog_Monitor.java that implements the ILTInterface has to be compiled and packaged into a .jar file. Care should be taken to include the dependent packages that this Java program depends on. The following are the extra jar files required and can be found by searching in the installation directory of the Resource Model Builder:

- javutils.jar: Used for logging purposes.
- dm_m12.jar: Used for the M12Instrumentation implementation
- gnu-regexp-1.1.X.jar: Used for matching regular expressions

The gnu-regexp can be obtained externally from the Internet. There are many Web sites from which information about regular expression can be obtained. The following is one such link:

http://www.cacas.org/java/gnu/regexp/

Note: The gnu-regexp-1.1.X.jar is not provided as part of the downloadable materials for this redbook. The reader should download the jar files from the link specified above. The version of this jar file may change with time.

For explanatory purposes, let us copy these jar files to a folder called dependencies in our working directory, as shown in Figure 6-3.
Now we have all the files required for building our package. We will set the path
to the Java home directory, set the CLASSPATH to include the appropriate jar
files, and compile the ILT implementation ITSO_CBELog_Monitor.java to obtain
the package class files.

As seen in Figure 6-4, after the compilation, a directory called ./com
corresponding to the Java package is created. This is because we supplied the
-d parameter to the Java compiler, informing the compiler to create the package
directory structure in the current directory.

![Command Prompt]
```
C:\ITSO_CBELog_Monitor>set PATH=c:\j2sdk1.4.1_06\bin;%PATH%
C:\ITSO_CBELog_Monitor>set CLASSPATH=c:\ITSO_CBELog_Monitor\dependencies\javautils.jar;c:\ITSO_CBELog_Monitor\dependencies\dn_m12.jar;c:\ITSO_CBELog_Monitor\dependencies\gnu-regexp-1.1.1.jar;%CLASSPATH%
C:\ITSO_CBELog_Monitor>javac -d . ITSO_CBELog_Monitor.java
```

![Command Prompt]
```
C:\ITSO_CBELog_Monitor>dir
Volume in drive C is IBM_PRELOAD
Volume Serial Number is 9C37-9CB1
Directory of C:\ITSO_CBELog_Monitor
01/28/2004 11:04 AM <DIR>         .
01/28/2004 11:04 AM <DIR>         ..
01/28/2004 11:04 AM <DIR>         com
01/27/2004 10:39 AM <DIR>         dependencies
01/27/2004 04:27 PM  37,473 ITSO_CBELog_Monitor.java
01/27/2004 10:27 AM  1,721 ITSO_CBELog_Monitor.mof
                      2 File(s)       39,194 bytes
                      4 Dir(s)     1,588,281,344 bytes free
C:\ITSO_CBELog_Monitor>
```

Figure 6-4   Compiling the ILT class

**Creating the ILT package**

We now have the class files in the directories, as given in the package
information of the class. We will use the jar utility to create the jar file
ITSO_CBELog_Monitor.jar, as shown in Figure 6-5 on page 114.

As seen in the figure, the “-cvf” parameter is given for “create”, “verbose”, and
“archive filename”, respectively, for the jar command to create the
ITSO_CBELog_Monitor.jar file in the current directory.
We now have the jar file that the Autonomic Management Engine (AME) will load to access the respective ILT interface, as mentioned in the MOF file.

The next step is to open the Resource Model Builder and create a Resource Model Package. By default, the Resource Model Builder 1.2 (current version) will be installed in the C:\Program Files\IBM WebSphere Studio\eclipse\ directory.

Figure 6-6 on page 115 shows the welcome screen when the Resource Model Builder is started.
Figure 6-6  Welcome screen after invoking the Resource Model Builder

As a standard practice, we create a project to contain the ITSO_CBELog_Monitor resource model, as shown in Figure 6-7, by clicking the New icon.

This will open a window to select the Wizard, and we choose the Tivoli Management Project to create the resource model, as shown in Figure 6-8 on page 116.
The next step is to give the name and destination directory for the Resource Model Builder project, as shown in Figure 6-9.
The next step is to use the *Basic Resource Model Wizard* to create the basic template required for the resource model. Figure 6-10 shows the window to choose this Wizard.

![Create A New Tivoli Management Project](image)

*Figure 6-10  Selecting the Basic Resource Model Wizard*

The Basic Resource Model Wizard window allows us to select various parameters of the resource model. The Autonomic Computing Toolkit is supported on Linux, AIX®, and Windows as of now. We will choose these three platforms so that this resource model can be used on these platforms, as shown in Figure 6-11 on page 118. Note that we can also choose to select Visual Basic or JavaScript for our scripting language. We choose JavaScript, as it is supported on all platforms we might want to deploy this resource model to.
The next window is used to choose the data source. In the context of the autonomic computing architecture, we will be using the CIM data source, which is understood by the Autonomic Management Engine. As shown in Figure 6-12 on page 119, choose the CIM/WMI data source.
The CIM class is coded in the ITSO_CBELog_Monitor.mof file and should be created in the namespace of the workbench. Figure 6-13 on page 120 gives the default “cimv2” namespace that is opened after clicking the **CIM/WMI Datasource** button.
Figure 6-13  Selecting the CIM Datasource in cimv2 namespace

The CIM v2 namespace consists of the CIM implementations provided by the Tivoli Monitoring Services. We can create our CIM class ITSO_CBELog_Monitor in this namespace or in the default namespace. We can change the namespace to default by modifying the root\cimv2 to root\default and by pressing the refresh button, as shown in Figure 6-14 on page 121.
We have to add our CIM data source, which can be done by compiling the ITSO_CBELog_Monitor.mof file, as discussed in 6.3.1, “Managed Object Format (MOF)” on page 94. To compile the MOF file, choose the Mof Compiler button. This will bring up the page to compile the .MOF file, as shown in Figure 6-15 on page 122.

**Note:** The name space displayed in this window of the wizard is the previously used name space by the Resource Model Builder. There are no restrictions on having the CIM data source in the default name space.
Press the **Go!** button to compile our MOF file. We can check the syntax or create a binary .MOF file from this Wizard. In this example, we compile our MOF file and ensure that we do not have any errors, as shown in Figure 6-16 on page 123 through Figure 6-18 on page 124.
Figure 6-16  Selecting the MOF file for compilation

Figure 6-17  Creating a new instance of the CIM class by compiling the MOF file
After pressing the **Finish** button, we know if there are any syntactical errors in the MOF file by checking the message window. Since the .MOF file does not have any errors, the compiler shows **Done!** in Figure 6-18.

![Figure 6-18   After compiling the .MOF file](image)

**Attention:** It should be observed that the Compiler Output window shows **Done!** after compiling the MOF file. If there are errors in the MOF file, they can be seen in this window, and the MOF file should be recompiled after fixing any errors.

We have now successfully compiled the MOF file and a data source is now added to the CIM Datasource repository under the root\default name space. After pressing the **Finish** button, refresh the name space by clicking the yellow connect button shown Figure 6-19 on page 125. Notice that our CIM data source ITSO_CBELog_Monitor is seen in the default name space with its FileName and Num parameters on the right hand side.
After selecting our CIM data source as ITSO_CBELog_Monitor, we provide various parameters to the Datasource, as shown below.

1. Select the property that can be used to trigger an event and in turn some actions. This is shown in Figure 6-20 on page 126, where the parameter Num is chosen. As discussed in the logic for the resource model, we will generate an event if the value of the Num returned is greater than five. This returned value indicates that the number of occurrences of the regular expression has exceeded five and it is time to generate an event and take actions.

Select the Num property from the left panel and click the > button to add it to the Selected properties panel. This tells the Resource Model Builder that this parameter will be responsible for generating an event in future.
2. The next step is to filter the data collected. This would be helpful when the data we are collecting is a large amount of data through an ILT class and we want to restrict the resource model to a specific set of functions. This is supported only on Windows and uses the WQL (WMI Query Language) to filter this data. Refer to the MicroSoft Developer’s Network Web site (http://msdn.microsoft.com/library) for more about WQL.

In the current example, we are not filtering any of the data that we are collecting. Figure 6-21 on page 127 shows this window.
3. In the next page of the Wizard, select the triggering condition to generate an event. In this example, for explanatory purposes, we generate an event if the Num value returned is greater than 5. In subsequent examples, we discuss how to associate actions with these events.

The triggering condition for our sample resource model would be that if the Num value is greater than 5, then we generate an event.

Select the property Num, as shown in Figure 6-22 on page 128, and click the > button to add a condition.
This will prompt for information regarding the condition, as shown in Figure 6-23. The condition for this scenario is that the value of Num is greater than 5.
The generated events can be sent to Tivoli Enterprise™ Console (TEC), Tivoli Business Systems Manager (TBSM), or other such supported management applications. From the Autonomic Computing Toolkit perspective, we use an Autonomic Management Engine that will receive these events and invoke actions. The plug-ins described and configured in the Autonomic Management Engine will use these events. These plug-ins can further be configured so that the information is sent to the respective management application. Refer to 7.4, “Advanced topics” on page 202 for more details.

The severity of the messages can also be indicated here. These events can be associated with actions and can be used for debugging purposes. Actions are also consumed by the Autonomic Management Engine and need Action Launcher plug-ins configured. This topic is discussed in 7.4, “Advanced topics” on page 202. For the current example, we will not associate any actions with this resource model.

The condition and the parameters can be seen in the right hand panel after defining the condition shown below in Figure 6-24.

![Figure 6-24  Selecting the triggering condition - phase 3](image)
4. The next page of the Wizard can be used to reference the parameters being monitored that can be logged. This is an optional selection, and selecting all of the parameters is suggested for debugging purposes. Select the parameters on the left side of the panel and click the > button to add it to the right hand side. In Figure 6-25, only one parameter is shown. This will automatically generate the code in the script that is required to log the values to the log file.

![Figure 6-25 Selecting the parameters to log](image)

5. Finish the Wizard window for selecting the CIM data source and its parameters. Figure 6-26 on page 131 explains that the data source selected is ITSO_CBLog_Monitor and the associated triggering condition is that the parameter Num is greater than 5.
After pressing the **Finish** button, this wizard completes the configuration of the data source for the resource model. This updated configuration of the data source is followed by defining a few final parameters.

As shown in Figure 6-27 on page 132, give the name for the resource model as ITSO_CBELog_Monitor and leave the events generated as is.

The next step is to define the cycle property for the resource model. As discussed in the introduction to resource models at the beginning of this chapter, the cycle time refers to the time at which the properties of the resource being monitored are fetched. Figure 6-28 on page 132 shows that the cycle time chosen is 60 seconds. This means that every 60 seconds, the VisitTree() function in the decision tree script is invoked and the number of occurrences of the regular expression is checked.
Figure 6-27  Selecting the name for the resource model, events, and logging

Figure 6-28  Selecting the cycle time for monitoring the resource
The next page defines the folder name where the Java resource model (.jrm) project file is created. As shown in Figure 6-29, specify the directory as the current project directory.

![Figure 6-29 Storing the project work space in .jrm file](image)

Click the **Finish** button, which completes the Wizard for creating a basic resource model. Figure 6-30 on page 134 and Figure 6-31 on page 135 shows the wizard completion and the project work space for the resource model with various parameters and names.
It can be observed that all the various components of a typical resource model can be seen in the right hand side panel in Figure 6-31 on page 135 in different tabs. That is, there are tabs for General Settings, Events, Thresholds, Logging, Source, and so on. The tabs are located at bottom of the large frame on the right side of the window.
Specifying resource model parameters

We now have a project work space and the basic layout for our resource model. The resource model parameter tabs should be configured so that the CIM class is given the required parameters like the file name and the search string. In addition to this, we provide parameters for generating events based on thresholds and add logging for debugging purposes.

General Settings

This tab allows for the specification of basic information required for the resource model. Internal Name is the name for the resource model used. Scripting language is JavaScript, as the resource model is used across platforms.
**CIM Classes**

So far, we have used the Wizard to create the resource model. We have selected the CIM data source already and have set the filtering condition as none. This tab can be used to define additional parameters or CIM classes if required.

When we create an empty resource model by selecting **File → New → Empty Resource Model Wizard**, we can use this tab to add our CIM class and give the required filtering parameters. Figure 6-32 is a pictorial view of the CIM Classes tab.

![CIM Classes tab of the resource model](image)

**Figure 6-32 CIM Classes tab of the resource model**

**Classic Probes**

Classic Probes can be used to add Tivoli Distributed Monitoring classic probes. For our example, we do not need these classic probes. This tab will be blank.
Events
As seen in Figure 6-33, the Resource Model Builder Wizard has added a default Event called Ev_ITSO_CBELog_Monitor_Num_too_high. This is associated to various parameters, as seen in the figure. We can associate actions for these events by clicking the Actions button. When we associate actions, the Autonomic Management Engine has to be configured with the plug-ins required to execute these actions. The advanced topics discussed in the Autonomic Management Engine Developer’s Guide provided with the Autonomic Computing Toolkit can be referenced for more information.

Thresholds
The threshold property is created through the Wizard and has the default Wizard created properties associated, as shown in Figure 6-34 on page 138.

The description section in this tab explains which event is triggered when this threshold is exceeded.
Parameters

Going back to our implementation in the CIM class, we have designed the CIM class to read the file name to be monitored and to search for a string “ITSO314C” in this file. The associated parameters for these two variables in the CIM classes can be found in the code snippets of the CIM (or ILT) class, as shown below.

In the getProperty() method of ITSO_CBELog_Monitor.java (ILT implementation):

```
Vector vParam = (Vector)parameterset.getParam("CBE_search_Pattern");
```

In the enumerateInstances() method of ITSO_CBELog_Monitor.java:

```
Vector vParam = (Vector)parameterset.getParam("ITSO_CBELog_Monitor_FileName");
```

These two parameters with values can be added in the tab, as shown in Figure 6-35 on page 139.
Logging
While creating the resource model using the Wizard, as shown in Figure 6-25 on page 130, the parameters mentioned for logging are listed in this tab. More parameters for logging can be added.

Dependencies
This is an important section for the resource model. The resource model should contain all the classes (in terms of jar files), the .MOF file, and any other properties file the ILT implementation (CIM class) uses. In our example, we use the javaultils.jar, dm_m12.jar and gnu-regexp-1.1.X.jar as explained in “Compiling the ILT class” on page 112. We need to add all these jar files into the dependencies tab, as shown in Figure 6-36 on page 140 and Figure 6-37 on page 140.
This completes the process of using the Resource Model Builder Wizard to create the resource model.

**Note:** The gnu-regexp-1.1.x.jar is included as a dependency. As the additional material provided with this redbook does not provide this jar file, the CLASSPATH is modified to load the gnu-regexp-1.1.x.jar file. This is a workaround if some dependencies cannot be shipped with the resource model.
6.3.4 Understanding and editing the decision tree JavaScript

The Resource Model Builder creates the JavaScript that is a part of the resource model. The Autonomic Management Engine has routines to invoke these methods and are discussed below.

**SetDefaultConfiguration()**

The purpose of this method is explained in 6.1, “Introduction to resource models” on page 86. The example shown below can be further customized for logging and any other initialization. This can be edited from the Source tab in the Resource Model Workbench.

*Example 6-19  SetDefaultConfiguration function of the decision tree JavaScript*

```javascript
function SetDefaultConfiguration (Svc)
{
    Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "SetDefaultConfiguration entered");

    // General info section
    //<<GENERAL_INFO>>
    Svc.SetModelName("ITSO_CBELog_Monitor");
    Svc.SetProfileName("1075295290568");
    Svc.SetCycleTime(60);
    //<<\GENERAL_INFO>>

    // Thresholds section
    //<<THRESHOLDS_INFO>>
    Svc.DefineThreshold("Thr_Num_gt", 5.0);
    //<<\THRESHOLDS_INFO>>

    // Parameters section
    //<<PARAMETERS_INFO>>
    Svc.DefineStrParameter("ITSO_CBELog_Monitor_FileName", "CBEOut.log");
    Svc.DefineStrParameter("CBE_search_Pattern", "ITSO314C");
    //<<\PARAMETERS_INFO>>

    // Dynamic model section
    //<<DATA_INFO>>
    Svc DEFINE_CLASS ("CIM", "ITSO_CBELog_Monitor",
    "root\default:ITSO_CBELog_Monitor", " ", "Num", "FileName", "None", " ", 0, 1);
    //<<\DATA_INFO>>

    // Event definition section
    //<<EVENTS_INFO>>
    Svc DEFINE EVENT ("Ev_ITSO_CBELog_Monitor_Num_too_high", "UpperBound,Num",
    "FileName");
    //<<\EVENTS_INFO>>
}```
// Logging definition section

//<<LOGGING_INFO>>
Svc.DefineLogInst("ITSO_CBELog_Monitor_Availability",
"ITSO_CBELog_Monitor", "FileName", "Num", "FileName");
//<<\LOGGING_INFO>>

// Place your additional initializing code below

Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "SetDefaultConfiguration exited");

return (0);
}

**Init()**

The Init() method is used for initializing various parameters when the instance of
the resource model is started. The requirements for these initializations are
based on the decision logic used and the type of resources we are monitoring.

The Wizard generated Init() function will not contain any logic in it. The user has
to modify this method in the script to meet their requirements.

In our example, we initialize the Actions.bat or Actions.sh based on the type of
the platform on which the resource model is run. The logic and the
implementation used here is simple for explanatory purposes and should be
modified according to the specific needs of the solution. As shown in
Example 6-20, the Init() method is modified to perform the following:

- Initialize global variables for storing the type of the platform and the script to
be run as an action.
- Initialize the script name based on the type of the platform.
- Associate the parameters to the instance of the class.
- Add logging statements appropriately for debugging purposes.

**Example 6-20  Init function of the decision tree JavaScript**

```javascript
var interpType = ""; // Store the operating system type
var actionScript = ""; // Store the script run when the event is generated

function Init(Svc)
{
    Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "ITSO_CBELog_Monitor Init entered");

    var dimension = 0;
    var t = 0;
    interpType = Svc.GetInterp();
```
Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Init interpType is : "+interpType);

if ( interpType == "w32-ix86")
{
    actionScript = "Actions.bat" ;
} else
{
    actionScript = "Actions.sh" ;
}

dimension = Svc.GetStrParameterCount("ITSO_CBELog_Monitor_FileName");
if ( dimension > 0 )
{
    Svc.Trace(0,"ITSO_CBELog_Monitor RM Associating parameter for filename ");
    Svc.AssociateParameterToClass
("ITSO_CBELog_Monitor_FileName","ITSO_CBELog_Monitor");
}

dimension = Svc.GetStrParameterCount("CBE_search_Pattern");
if ( dimension > 0 )
{
    Svc.AssociateParameterToClass
("CBE_search_Pattern","ITSO_CBELog_Monitor");
    Svc.Trace(0,"ITSO_CBELog_Monitor RM Associating parameter for Pattern ");
}

Svc.Trace(0,"ITSO_CBELog_Monitor: Init - ended ");
return (0);

The Svc used in the script refers to the Service Object and documentation on various functions provided by this interface can be found in Appendix A, “Service Object Method Library”, in IBM Tivoli Monitoring Workbench User’s Guide, SH19-4571, available at the IBM Publications Web site:

http://publib.boulder.ibm.com/

VisitTree()

The VisitTree() method of the JavaScript explained in this example is a simple implementation of invoking a script (Actions.bat or Actions.sh) whenever an event is triggered.

In the example given below, Line numbers 36 to 40 has been added so that we execute the actionScript file whenever the threshold is reached.
Example 6-21  Visit tree function of the ITSO_CBELog_Monitor resource model

```javascript
function VisitTree(Svc)
{
  //vars for data source: ITSO_CBELog_Monitor
  var curITSO_CBELog_MonitorNum;

  var curITSO_CBELog_MonitorFileName;

  var hPropTable;
  var numOfInstances;
  var idx;

  var ParamCount;
  var ParamIdx;
  var Different;

  hPropTable = Svc.CreateMap();

  //triggering logic for data source: ITSO_CBELog_Monitor
  numOfInstances = Svc.GetNumOfInst("ITSO_CBELog_Monitor");
  for(idx = 0; idx < numOfInstances; idx++) {
    Svc.RemoveMapAll(hPropTable);
    //Handle numeric properties
    Svc.SetMapNumElement(hPropTable, "Num", curITSO_CBELog_MonitorNum);
    //Handle string properties
    curITSO_CBELog_MonitorFileName = Svc.GetStrProperty("ITSO_CBELog_Monitor", idx, "FileName");
    Svc.SetMapStrElement(hPropTable, "FileName", curITSO_CBELog_MonitorFileName);

    if (curITSO_CBELog_MonitorNum > Svc.GetThreshold("Thr_Num_gt")) {
      Svc.SetMapNumElement(hPropTable, "UpperBound", Svc.GetThreshold("Thr_Num_gt"));
      Svc.SendEventEx ("Ev_ITSO_CBELog_Monitor_Num_too_high", hPropTable);
      var shellRC = Svc.ShellCmd(actionScript);
      var str_result = Svc.GetShellStdOut();
      var str_err = Svc.GetShellStdErr();
      Svc.Trace(TRACE_FINEST, TRACE_SOURCE, "Returned result from actionScript is: "+str_result);
      Svc.Trace(TRACE_FINEST, TRACE_SOURCE, "Returned error from action Script is: "+str_err);
    }
  }
}
```
A hash table of the Num property returned is maintained within the VisitTree() method, and when the value exceeds a defined threshold of ‘Thr_Num_gt’, an event is generated. In this example, we generate an event and will not do anything with this event. If we associate actions or if we want to use this event with any other management console like Tivoli Enterprise Console® or Integrated Solution Console, these events will be helpful. The StdOut and StdErrs are logged for debugging purposes.

**Building the resource model package**

The resource model is a collection of various configuration files, resource bundle, JavaScripts, CIM classes, providers, dependencies, and the MOF file (see Figure 6-38 on page 146). It is a zip file consisting of all these related files. They are packaged in a way so that is understood by the Autonomic Management Engine.
The Resource Model Builder package can be built by selecting ITM → Generate Package → TAME (zip). The Resource Model Builder creates a ITSO_CBELog_Monitor.zip file that contains the various files. Various platforms are identified by their respective platform types, as given below.

- __aix4_r1__ITSO_CBELog_Monitor.zip: AIX related files
- __linux_ix86__ITSO_CBELog_Monitor.zip: Linux related files
- __w32-ix86__ITSO_CBELog_Monitor.zip: Windows related files

The contents of a resource model are discussed in detail at the end of this chapter.

The Autonomic Management Engine that loads this resource model opens this zip file and, based on the platform, uses different configuration files. As seen in the VisitTree() method, the platform specific information can be obtained by calling the GetInterp() method in the Service Object interface.

If there are errors in the configuration or script file the messages for correction can be seen by opening Window → Show View → Tasks. This will open a
window in the bottom pane of the Resource Model Builder workspace and can be checked for errors before building.

Other parts of the resource model can be generated if required by pressing various options, as seen in Figure 6-39.

![Figure 6-39 Generating various components of the resource model]

### 6.4 ITSO_CBESocket_Monitor resource model

In the previous example, the resource we were monitoring was a file called CBEOut.log (or as configured by the ITSO_CBESocket_Monitor_FileName parameter in the resource model). The CBEOut.log is populated with Common Base Events by the ManagerResourceTouchpointSupport class, which writes the events to a file in its `sendEvent()` method.

In a real life scenario, if we want to extend the functionality of the Autonomic Manager to another machine, and if we want to send the Common Base Events to a different machine, we can make use of sockets to write these Common Base Events to a socket instead of a file. This provides the ability to maintain the management engine and the resource model on different machines. This could allow the management engine to manage resources across more than one system.

As another example, Common Base Events can be populated in a database by the ManagerResourceTouchpointSupport and the resource model can be configured to read these Common Base Events from the database.
6.4.1 Resource model design

The resource model for listening on the socket primarily varies in the ILT implementation when compared to the previously designed resource model. Instead of a file name, this resource model takes the port number as a key parameter. As explained in the flow chart in Figure 6-40 on page 149, the CIM class creates another thread called the CBEServerThread, which opens a socket for listening for Common Base Events.

This port number can be mentioned in the parameter section of the resource model. When the ILT interface is instantiated, we start this thread and start monitoring the port for any incoming data. The data received on this port can be directly maintained in a buffer. For design considerations, we will store it to a temporary file called serverStore.tmp in the current directory.

This example is now easy to understand, as the getProperty() method in the ILT interface now gathers the data from the serverStore.tmp file and the decision tree script is given the number of occurrences of the search string in this temporary file.

Note: At this level of the Autonomic Computing Toolkit, the management engine and the managed resource are supported on a single machine or a single server. The communication APIs between different servers are still under development. This example is intended to be purely educational and needs to be configured based on specific IT solution needs.
Logic flow in ITSO_CBESocket_Monitor resource model

1. The ITSO_CBESocket_Monitor is configured with the port number, threshold, events, and the actions.
2. The resource model is loaded and executed by the Autonomic Management Engine environment.

3. A ServerSocket thread called CBEServerThread reads the port and opens a socket to read the data.

4. When the data arrives on this port, it reads the data and stores it to a temporary file.

5. The Svc.GetNumProperty() call is made from the resource model decision tree on every cycle. This reads the temporary file (or the internal buffer instead of a file) and returns the number of matching lines.

6. If this number exceeds the threshold of 5, specific actions are triggered by the VisitTree() method of the decision tree JavaScript. In our example, we will execute a simple batch file or shell script that writes a file in the current directory giving instructions for the Managed Application to take further actions. In our somewhat simplistic implementation, our resource model initiates an action by creating a file that is monitored by the managed application. In the future, the effector interface of the managed resource touchpoints will be used to initiate actions.

### 6.4.2 Parameters for ITSO_CBESocket_Monitor resource model

As per the above design, we now list the various parameters required for this resource model (Table 6-1 through Table 6-4 on page 151).

**Parameters**

*Table 6-1 Parameters for ITSO_CBESocket_Monitor resource model*

<table>
<thead>
<tr>
<th>Parameter and type</th>
<th>Sample value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITSO_CBESocket_Monitor_Port</td>
<td>4445 (Randomly chosen)</td>
</tr>
<tr>
<td>CBE_search_Pattern</td>
<td>ITSO314C</td>
</tr>
</tbody>
</table>

**Events**

*Table 6-2 Events generated by ITSO_CBESocket_Monitor RM*

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITSO_CBESocket_Monitor_too_high</td>
<td>&gt; 5</td>
</tr>
</tbody>
</table>

**Threshold**

*Table 6-3 Threshold for generating event in ITSO_CBESocket_Monitor RM*

<table>
<thead>
<tr>
<th>Threshold name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thr_EventNum_gt</td>
<td>5</td>
</tr>
</tbody>
</table>
### General Settings and values

**Table 6-4  General settings and other relevant values for the RM**

<table>
<thead>
<tr>
<th>Elements</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Name</td>
<td>ITSO_CBESocket_Monitor</td>
</tr>
<tr>
<td>Descriptive Name</td>
<td>ITSO_CBESocket_Monitor</td>
</tr>
<tr>
<td>Cycle time</td>
<td>60 seconds</td>
</tr>
<tr>
<td>Platforms</td>
<td>windows, aix4-r1, or linux-ix86</td>
</tr>
<tr>
<td>Scripting</td>
<td>JavaScript</td>
</tr>
<tr>
<td>Logging</td>
<td>Num and Port</td>
</tr>
<tr>
<td>Dependencies</td>
<td>ALL: ITSO_CBESocket_Monitor.mof javautils.jar dm_m12.jar</td>
</tr>
<tr>
<td></td>
<td>gnu-regexp-1.1.1.jar</td>
</tr>
<tr>
<td></td>
<td>ITSO_CBESocket_Monitor.jar</td>
</tr>
</tbody>
</table>

The summary of the flow of various methods and interaction between different parts of the resource model can be pictorially represented, as shown in Figure 6-41 on page 152.
6.4.3 Creating the ITSO_CBESocket_Monitor resource model

The following sections provide details on creating this new resource model. In many cases, the implementation is just a minor variation of the previous resource model.

The MOF File
The MOF file for this resource model defines the CIM data source, the provider, the M12Instrumentation, the class name, and the attributes of the class name as shown in Example 6-22 on page 153.
Example 6-22  MOF file for the socket resource model

////////////////////////////////////////////////////////////////////////
//
// Filename : ITSO_CBESocket_Monitor.mof
//
// Description:
//   Gives textual definition for the ITSO_CBESocket_Monitor CIM class.
//
// File History:
//
// Date                        Author                   Reason/Description
//----------------------------------------------------------------------------
// 01/26/2004                  ITSO Redbook team        Initial file
//
////////////////////////////////////////////////////////////////////////

[Description ("ITSO_CBESocket_Monitor: This sample resource model is created to" "educate the readers about the resource model concepts."
  "This resource model monitors a PORT<Key parameter > "
  "Port</Key parameter>" "and return the number of occurrences of the regular expression"
  "given as a parameter to the resource model. This resource "
  "model exhibits only a part "
  "of the functionalities provided by M12 qualifiers."),
provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider"),
M12_Instrumentation {
  "Java.com.ibm.tivoli.monitoring.ITSO_CBESocket_Monitor | | ENUM",
  "Java.com.ibm.tivoli.monitoring.ITSO_CBESocket_Monitor | | GET"}]
class ITSO_CBESocket_Monitor
{
    //Attributes for the ILT implementation
    [Description("The PORT property is provided by
ITS_CBESocket_Monitor_FileName as"
      "a parameter to the Resource Model.")],
    key ]
    string Port;
    sint32 Num; // Attribute, Integer returning the number of occurences.
};
The ILT implementation
The CIM data source, as mentioned in the MOF file, refers to the following class, which implements the ILT interface:
Java.com.ibm.tivoli.monitoring.ITSO_CBESocket_Monitor

This means that the ILT interface is ITSO_CBESocket_Monitor in the package com.ibm.tivoli.monitoring. Various parts of the ILT implementation are explained below. We will only show those aspects of the ILT that vary from the previous example.

Imports
The imports for this resource model will be almost the same, except that a socket implementation is used to capture the Common Base Events arriving at a port number given by the resource model parameters, as given in Table 6-1 on page 150. Example 6-23 shows the imports for this ILT interface.

Example 6-23 Imports for ITSO_CBESocket_Monitor resource model

```
// Package details.
package com.ibm.tivoli.monitoring;

import com.tivoli.dmunix.ep.touchpoint.base.*; // Tivoli Monitoring Class : "dm_m12.jar"
import com.tivoli.javaultils.Trace; // Tivoli Monitoring : "javaultils.jar"
import java.util.Enumeration; // Java Class
import java.util.*; // Java Class
import gnu.regexp.*; // GNU REGEXP package : "gnu-reg-exp1.1.1.jar"
import java.lang.*; // Java Class
import java.io.*; // Java Class
import java.net.*; // Java Class for Socket implementation
```

ILT class implementation
This ILT class should implement the ILTInterface, as shown in Example 6-24. The ILT interface is declared in M12Specifications, which is contained in dm_m12.jar (provided with the Resource Model Builder).

Example 6-24 Resource model class implementing the ILTInterface

```
public class ITSO_CBESocket_Monitor implements ILTInterface
```
**Constructor**

The constructor for this ILT implementation should also start a thread that starts listening for incoming Common Base Events on the specified port. Example 6-25 shows the starting of the thread in the constructor.

**Example 6-25  Constructor for the ILT implementation**

```java
public ITSO_CBESocket_Monitor()
{
    trace = Trace.getTraceInstance();
    //Start the thread to listen on the Socket

    try {
        trace.log(1, "ITSO_CBESocket_Monitor", "The port number is:"+port);
        CBEServerThread cbest = new CBEServerThread();
        cbest.setPort(port);
        cbest.start();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

As seen in the example, the thread name is CBEServer and is started by calling the start() method of the instance. When the resource model instance is instantiated, the global variable Port is set and the setPort() method is used to set the port number for this server thread to operate.

**ILT public methods**

The public methods used in this resource model differs from the previous example in terms of the variables used.

**getProperty()**

This public method (Example 6-26 on page 156) in the ILT interface is used when the resource model is requested to give the details of the properties declared for the data source in the MOF file. This method is responsible for giving the value of the Num or the Port number whenever requested by the Autonomic Management Engine.

This method stores the parameters as declared in the resource model, validates them and calls the getAttribute() service method to return the value requested. We return a null in that case.
Example 6-26  getProperty method for ITSO_CBESocket_Monitor resource model

public String getProperty( M12ObjectIdentity m12objectidentity,
    String sPropertyName,
    String sMapping,
    ParameterSet parameterset )
    throws M12Exception
{
    try
    {
        // Variable definitions
        StringBuffer stringbuffer = new StringBuffer();
        Vector vector = new Vector();

        // Get the parameters of the patterns from the caller.
        // This is mentioned in the PARAMETERS section of the resource model
        // If the parameters are null, then we dont have anything to search.
        // We return a null in that case.
        //
        Enumeration enumeration = parameterset.parametersNames();
        Vector vParam = (Vector)parameterset.getParam("CBE_search_Pattern");
        if(vParam == null)
        {
            trace.log(1, "ITSO_CBESocket_Monitor", "ERROR: getProperty: no parameters
            found");
            return null;
        }
        // convert the vector into String
        String asCBE_search_Pattern[] = new String[vParam.size()];
        for(int iIndex = 0; iIndex < vParam.size(); iIndex++)
        {
            // Get the string array element from the vector element.
            asCBE_search_Pattern[iIndex] = (String)vParam.elementAt(iIndex);
            trace.log(3, "ITSO_CBESocket_Monitor",
            "getProperty: asCBE_search_Pattern[" + iIndex + "] = " + asCBE_search_Pattern[iIndex]);
        }
        M12IdentityElement m12identityelement =
m12objectidentity.getScopingPath()[0];
M12PropertySet m12propertyset = m12identityelement.getIdentity();

///////////////////////////////////////////////////////////////////////////
// Determine which instance is being checked by the
// Management Layer.
// As the Port is declared as the [Key] parameter in the MOF
// file, the file name for each instance can be obtained by m12propertyset
// This would be instantiated when the resource model starts an instance.
///////////////////////////////////////////////////////////////////////////

String sKey = m12propertyset.getProperty("Port");

trace.log(3, "ITSO_CBESocket_Monitor",
    "getProperty: called for Port from m12propertyset " + sKey);

///////////////////////////////////////////////////////////////////////////
// Get the attributes is being checked by the Management Layer.
// Get the current attribute values.
// For the sake of simplicity we have used a service method called the
// getAttribute to retrieve the data for us.
// When the getProperty call is made from the resource model, the getAttribute
// return the required attribute.
// See the documentations for getAttribute method.
///////////////////////////////////////////////////////////////////////////

String sValue = getAttribute(sKey, sPropertyName, asCBE_search_Pattern);
trace.log(3, "ITSO_CBESocket_Monitor",
    "getAttribute: requested " + sPropertyName + " = " + sValue);
return sValue;
}
catch(Exception exception)
{
    trace.exception("ITSO_CBESocket_Monitor", "getProperty", exception);
    M12Exception m12exception = new M12Exception(exception);
    throw m12exception;
}
}

**getMultipleProperties()**

This method (Example 6-27 on page 158) will be the same as
ITSO_CBELog_Monitor.java ILT file. The only difference would be to fetch the
correct parameters declared in the parameters section. This function populates
the M12PropertySet using the getAttribute() method and returns all the properties as an Enumeration object.

Example 6-27  getMultipleProperties method for ITSO_CBESocket_Monitor resource model

```java
public M12PropertySet getMultipleProperties(
    M12ObjectIdentity m12objectidentity,
    Vector vPropertyList,
    String sMapping,
    ParameterSet parameterset )
throws M12Exception
{
    try {
        // Variable Declaration.
        StringBuffer stringbuffer = new StringBuffer();
        Vector vector = new Vector();

        // As documented in getProperty method
        Enumeration enumeration = parameterset.parametersNames();
        Vector vParam = (Vector)parameterset.getParam("CBE_search_Pattern");
        if(vParam == null) {
            trace.log(1, "ITSO_CBESocket_Monitor", "ERROR: getMultipleProperties: no parameters found");
            return null;
        }

        // Process the ITSO_CBESocket_Monitor_Port parameter.
        String asCBE_search_Pattern[] = new String[vParam.size()];
        for(int iIndex = 0; iIndex < vParam.size(); iIndex++) {
            // Get the string array element from the vector element.
            asCBE_search_Pattern[iIndex] = (String)vParam.elementAt(iIndex);
            trace.log(3, "ITSO_CBESocket_Monitor", "getmultipleProperties: asITSO_CBESocket_Monitor_Pattern[" + iIndex + "] = " + asCBE_search_Pattern[iIndex]);
        }

        M12IdentityElement m12identityelement =
            m12objectidentity.getScopingPath()[0];
        M12PropertySet m12propertyset = m12identityelement.getIdentity();

        // Determine which instance is being checked by the Management Layer.
        String sKey = m12propertyset.getProperty("Port");
        M12PropertySet m12propertyset1 = new M12PropertySet();
        trace.log(3, "ITSO_CBESocket_Monitor", "getmultipleProperties: asITSO_CBESocket_Monitor_Pattern[" + iIndex + "] = " + asCBE_search_Pattern[iIndex]);
    }
```
"getMultipleProperties: called for Port from m12propertyset1 "
+ sKey);
for(int i = 0; i < vPropertyList.size(); i++)
{
    // Get the property being checked by the Management Layer.
    String sProperty = (String)vPropertyList.elementAt(i);

    // Get the current attribute values.
    String sValue = getAttribute(sKey, sProperty, asCBE_search_Pattern );
    m12propertyset1.setProperty(sProperty, sValue);
}
trace.log(3, "ITSO_CBESocket_Monitor",
"getMultipleProperty: returning " +
convertToString(m12propertyset1));
return m12propertyset1;
} 
catch(Exception exception)
{
    trace.exception("ITSO_CBESocket_Monitor", "getMultipleProperties", exception);
    M12Exception m12exception = new M12Exception(exception);
    throw m12exception;
}

enumerateInstances()
Enumerate instances, in this example, should instantiate the resource model
instances and should set the class property called port so that the server thread
can be started to listen for incoming Common Base Events on this port.

The port value is mentioned as a string in the parameters section of the resource
model. The parameter name for this value is:

ITSO_CBESocket_Monitor_Port

This property is read using the getParam() function of the parameter set and is
assigned to the port variable, as shown in Example 6-28.

Example 6-28  enumerateInstances method for ITSO_CBESocket_Monitor resource model

public Enumeration enumerateInstances(M12ClassPath m12classpath,
        String sMapping,
        ParameterSet parameterset )
    throws M12Exception
    {
        try
        {
            // Variable Declaration.
            StringBuffer stringbuffer = new StringBuffer();

            //
Vector vector = new Vector();

// Get the parameters from the caller.
// We are obtaining the parameters prior to declaring instances so
// we can configure an instance for each parameter. These parameters
// are declared in the PARAMETERS section of the resource model.

Enumeration enumeration = parameterset.parametersNames();
Vector vParam = (Vector)parameterset.getParam("ITSO_CBESocket_Monitor_Port");

if(vParam == null)
{
    trace.log(1, "ITSO_CBESocket_Monitor", "ERROR: enumerateInstances: no
parameters found");
    return null;
}

// Process the ITSO_CBESocket_Monitor_Port parameter to a string.
String asITSOArrayGui[] = new String[vParam.size()];
for(int iIndex = 0; iIndex < vParam.size(); iIndex++)
{
    // Get the string array element from the vector element.
    asITSOArrayGui[iIndex] = (String)vParam.elementAt(iIndex);
    trace.log(3, "ITSO_CBESocket_Monitor",
              "enumerateInstances: asITSOArrayGui[" + iIndex + "] = " + asITSOArrayGui[iIndex]);
}

// Acquire a vector (pointer) from the calling Management Engine.
// This section handles the registration of each instance of this
// Instrumentation.

String aiInst[] = getInstances( asITSOArrayGui );

if(aiInst == null)
{
    trace.log(1, "ITSO_CBESocket_Monitor",
              "enumerateInstances: no instances found");
    return null;
}
for(int jIndex = 0; jIndex < aiInst.length; jIndex++)
{
    trace.log( 3, "ITSO_CBESocket_Monitor", "enumerateInstances: aiInst[" +
        jIndex + "] = " + aiInst[jIndex]);
    M12PropertySet m12propertyset = new M12PropertySet();
    m12propertyset.setProperty("Port",
        aiInst[jIndex]);

    // Set the port number so that the server thread can be started..

    this.port = aiInst[jIndex];

    M12IdentityElement m12identityelement =
        new M12IdentityElement( m12classpath.getClassName(),
            m12classpath.getNameSpace(),
            m12propertyset);
    vector.add( new M12ObjectIdentity( new M12IdentityElement[]
        { m12identityelement } ) );
}

return vector.elements();
}
}
catch(Exception exception)
{
    trace.exception("ITSO_CBESocket_Monitor", "enumerateInstances", exception);
    M12Exception m12exception = new M12Exception(exception);
    throw m12exception;
}
}

**Unused methods**

The following listed methods are not used in this example and hence will return null, as explained in the earlier section:

- setProperty()
- invokeMethod() with M12Classpath
- invokeMethod() with M12ObjectIdentity
- create()
- destroy()

Some of the methods have trace statements added for debugging purposes.

**getAttribute()**

getAttribute() is used by the getProperty() and the getMultipleProperties() methods to fetch the data.
This method (Example 6-29) maintains the class variable called “offset”, which stores the value of the last line read from the temporary file created by the server thread. The function getITSO_CBESocket_MonitorVar_Num() is called to get the number of occurrences of the search pattern, as given in the parameters section of the resource model.

**Example 6-29 getAttribute method for ITSO_CBESocket_Monitor resource model**

```java
private String getAttribute( String sValue,
                      String sProperty,
                      String asCBE_search_Pattern[] )
    throws Exception
{

    // Check for each property whose value is requested ...
    //
    if( (sProperty.equals("Num")))
    {
        String offset = "-1";
        int r = 0;

        // Retrieve the offset value for the Port from the offsetarray
        // hashtable. If the offsetarray doesn't have the appropriate offset,
        // assign "-1" to the offset variable and keep it in the offsetarray.
        //
        String v = (String)offsetarray.get(sValue);
        if ( v == null ) {
            offset = "-1";
            offsetarray.put(sValue, offset);
        } else {
            offset = v;
        }

        // Run the getITSO_CBESocket_MonitorVar_Num() and assign the result
        // to the intVar[] array.
        // intVar[] = { Num, offset }
        // We will then store these values in the global hash table so that
```
// they can be retrieved next time for further processing.  
// The first integer contains the number of occurrences ( Num ) and  
// the second integer ( intVar[1] ) contains the offset of the previous  
// iteration while reading the log file.  
///////////////////////////////////////////////////////

int intVar[] = getITSO_CBESocket_MonitorVar_Num( asCBE_search_Pattern, offset);

// Keep the new offset value in the offsetarray hashtable.  
offsetarray.put(sValue, String.valueOf(intVar[1]));

// Trace  
trace.log( 3, "ITSO_CBESocket_Monitor", "getAttribute result offset" +  
          intVar[1] + " Num = " + intVar[0] + " Port = " + sValue);
trace.log( 3, "ITSO_CBESocket_Monitor", "global array: offset" +  
          (String)offsetarray.get(sValue));

// Return the Num value.  
return String.valueOf(intVar[0]);

if( (sProperty.equals("Port")))
{
  // Return the Port value  
  return sValue;
} else {
  throw new Exception(sProperty + ": unknown property");
}

getITSO_CBESocket_MonitorVar_Num()  
This method (Example 6-30 on page 164) monitors a file called serverStore.tmp  
that is created in the current folder by the server thread listening on the datagram  
socket at the specified port.

For the purposes of our example, we have chosen to use a temporary file. Other  
methods like pipes or buffers can also be used to store these incoming Common  
Base Events. The search for regular expressions has to be performed in a string  
rather than the file. In this case, the index of the string read in the previous cycle  
can be used to differentiate between various cycles in which the resource model  
fetches the data.
Example 6-30  getITSO_CBESocket_MonitorVar_Num method for ITSO_CBESocket_Monitor resource model

```java
private synchronized int[] getITSO_CBESocket_MonitorVar_Num(String as[], String offset)
{
    // initializations ..
    int count_above = 0;
    int count_below = 0;
    int obj = 0;
    int lastline;
    int[] retarray = new int[2];

    try {
        RE re = null;        // Declare the Regular Expression
        REMatch mat;

        String sl = offset;  // offSet = retrieve offSet(i);

        trace.log(3, "ITSO_CBESocket_Monitor","offset returned is : "+offset);

        // Create a FileReader object
        FileReader fr = new FileReader("serverStore.tmp");
        BufferedReader br = new BufferedReader(fr);
        LineNumberReader lr = new LineNumberReader(fr);

        // Read strings in the file
        String line;
        while((line = lr.readLine()) != null) {
            for(int iIndex = 0; iIndex < as.length; iIndex++) {

                // Evaluate the line if the search pattern matches...

                try {
                    re = new RE(as[iIndex]);
                } catch (REException e) {
                    break;
                }

                Integer tmp = Integer.valueOf(sl);
                obj = tmp.intValue();

                // Pattern match
                mat = re.getMatch(line);
                if (mat != null) {
```
if (obj > lr.getLineNumber()) {
    count_above++;  // Increase the counter older than the offset
} else {
    count_below++;  // Increase the counter newer than the offset
}
}
} // End of while loop

// Store the last line number for this iteration ..
lastline = lr.getLineNumber();

trace.log(3, "ITSO_CBESocket_Monitor","Lastline parameter is"+lastline);

// Close the FileReader object
fr.close();

if (offset == "-1") {  // If the offset is "-1", this is the first time.
    retarray[0] = 0;
    retarray[1] = lastline;
} else {
    if (lastline < obj) {  // If the lastline is lower than offset, the file was
        retarray[0] = count_above;  // Returns the c1 value which is the
        retarray[1] = lastline;
    } else {  // If lastline is upper than offset,
        retarray[0] = count_below+count_above;  // returns the c1+c2 value
        retarray[1] = lastline;
    }
}

} catch (Exception e) {
    trace.log(3,"ITSO_CBESocket_Monitor","getITSO_CBESocket_MonitorVar_Num: Exception"+e);
}

trace.log(3, "ITSO_CBESocket_Monitor","count_above, count_below are "+count_above+" "+count_below);
return retarray;
CBEServer thread class
This server thread gets started when the resource model instances are enumerated and instantiated.

This class opens a port for receiving the data through sockets and listens through a receive() call.

Constructors
The constructors, as shown in Example 6-31, initialize the trace facilities and creates a new DatagramSocket object.

setPort()
This method initializes the port member variable of the class. A conversion of String to Integer is done in this method. This method is called by the ILT constructor to set the port for this class.

storeData()
This method creates a BufferedWriter and writes the incoming string to the file, as specified in the logFileName variable.

run()
This method is executed when the thread is started using the start() method. This method performs the following.
1. Creates a File object for temporary storage of the incoming stream.
2. Checks if there is a file called stop_ame; if so, exit the loop.
3. Creates a buffer of length 20,000 (considering that the Common Base Events can be large) and receive the packets transmitted to the port.
4. Trims the received stream and stores it to the temporary file.
5. Clears the buffer to receive the next incoming stream.
6. If the received stream is “end”, then the method terminates the thread; otherwise, go to step 2.

Example 6-31  ServerThread for listening on a specific port for the ITSO_CBESocket_Monitor resource model

class CBEServerThread extends Thread {

    protected DatagramSocket socket = null;
    protected String logFileName = new String("serverStore.tmp");

    private int port = 0;
    private File stop_thread;
}
private Trace trace;

public CBEServerThread() throws IOException {
   this("CBEServerThread");
}

public CBEServerThread(String name) throws IOException {
   super(name);
   trace = Trace.getTraceInstance();
   trace.log(1, "ITSO_CBESocket_Monitor", "Started the CBEServerThread.. the port being ":"+port);
   socket = new DatagramSocket(4445);
}

public void setPort(String port) {
   Integer tempInt = Integer.valueOf(port);
   this.port = tempInt.intValue();
}

private void storeData(String s) {
   BufferedWriter out;
   try {
      out = new BufferedWriter(new FileWriter(logFileName,true));
      out.write(s);
      out.close();
   }catch(Exception ee){
      ee.printStackTrace();
   }
}

public void run() {
   try {
      File temp = new File(logFileName);
      if(temp.exists()) {
         temp.delete();
      }
   }catch(Exception ee) {
      // do nothing
      System.out.println("The temporary file may be in use.. ");
   }

   while (true) {
      try {
         stop_thread = new File("stop_ame");
         if( stop_thread.exists()) {
            stop_thread.delete();
            socket.close();
            System.exit(0);
         }
      }
   }
}
byte[] buf = new byte[20000];

    // receive request
    DatagramPacket packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);
    String received = new String(packet.getData());
    System.out.println("Server Received a CBE of length: "+received.trim().length());
    if(received.trim().equals("end")) {
        socket.close();
        stop_thread = new File(logFileName);
        if( stop_thread.exists()) {
            stop_thread.delete();
        }
        System.exit(0);
        }else {
            storeData(received.trim());
            storeData("\n");
            storeData("\n");
            buf = null;
        }
}

) catch (IOException e) {
    try {
        File temp = new File(logFileName);
        if(temp.exists()) {
            temp.delete();
        }
    }catch(Exception ee) {
        // do nothing
        System.out.println("The temporary file may be in use.. ");
    }
    socket.close();
e.printStackTrace();
}

Compiling the ILT
The ILT Interface implementation is compiled after setting the proper class path for the dependencies. The steps shown in Figure 6-42 on page 169 can be used to compile and build the package.
6.4.4 Using the Resource Model Builder

The Resource Model Builder is used to create the ITSO_CBESocket_Monitor resource model. The steps followed here would be similar to the steps followed in the earlier example given in 6.3.3, “Generating the resource model using Resource Model Builder” on page 111.

Use the following steps to create this resource model:

2. Use the Basic Resource Model Wizard to add the CIM Data source.
3. Choose the platforms supported for this resource model.
4. Compile the MOF file ITSO_CBESocket_Monitor.mof file using the MOF Compiler in the resource model. After compiling without errors and refreshing the view in the default name space, the ITSO_CBESocket_Monitor data source can be seen in the default name space, as shown in Figure 6-43 on page 170.
5. Select the Num property as the property used for the triggering condition in the “Select Properties” page in the wizard. Go to the next page in the wizard.

6. The Set Filtering Condition page of the wizard is left as it is, as the data collection is not filtered for this resource model. Press the Next button.

7. Set the Event triggering Condition for Num property to be greater than 5 in the “Specify Event Triggering Conditions” page of the Wizard, as shown in Figure 6-44 on page 171.
8. Select both Num and Port properties to log in the next page of the Wizard.
10. Enter the Internal Name, events, and logging parameter names, as shown in Figure 6-45 on page 172.

*Figure 6-44  Specifying the event triggering condition for the resource model.*
11. Specify the cycle time of the resource model as 60 seconds.
12. Give the project name for this resource model as ITSO_CBESocket_Monitor and finish the Wizard.

After completing the Wizard, the project work space with the furnished details appears as shown in Figure 6-46 on page 173.
13. In the Events tab of the resource model workspace, uncheck the **Send events to TBSM and TEC**. If we choose to send these events to TBSM or TEC, separate Event handlers have to be implemented in the Autonomic Management Engine layer.

In the Parameters section of the resource model workspace, specify the two parameters read by the ILT. Please refer to Table 6-1 on page 150 for details on the parameters for this resource model.

Figure 6-47 on page 174 shows the parameters specified for this resource model.
14. The dependencies for the resource model are given in the Dependencies tab of this workspace. This must include the .MOF file, along with jar files and other properties files that any Java class depends on. These will be loaded by the Autonomic Management Engine class loader. Figure 6-48 on page 175 shows the dependencies added for this resource model.
This completes the steps to be followed using the Resource Model Builder to build the resource model.

**Understanding and editing the JavaScript**

The JavaScript generated automatically by the Resource Model Builder has to be edited so that we have the logic and implementation of the autonomic control loop.

The script should initiate various parameters in the `Init()` method and should do the required action of invoking the `Actions.bat` or `Actions.sh` script when the threshold is reached.
**setDefaultConfiguration()**

This function sets the default configuration settings and parameter initialization of the resource model. The code snippet Example 6-32 is directly taken from the automatically generated code, and we do not need to modify anything here.

**Example 6-32  SetDefaultConfiguration method for ITSO CBESocket Monitor resource model decision tree script**

```java
function SetDefaultConfiguration (Svc) {
    Svc.Trace(TRACEFINEST, TRACESOURCE + "SetDefaultConfiguration entered");

    // General info section
    //<<GENERAL_INFO>>
    Svc.SetModelName ("ITSO_CBESocket_Monitor");
    Svc.SetProfileName ("1075714535368");
    Svc.SetCycleTime (60);
    //<<\GENERAL_INFO>>

    // Thresholds section
    //<<THRESHOLDS_INFO>>
    Svc.DefineThreshold ("Thr_Num_gt", 5.0);
    //<<\THRESHOLDS_INFO>>

    // Parameters section
    //<<PARAMETERS_INFO>>
    Svc.DefineStrParameter ("ITSO_CBESocket_Monitor_Port", "4445");
    Svc.DefineStrParameter ("CBE_search_Pattern", "ITSO314C");
    //<<\PARAMETERS_INFO>>

    // Dynamic model section
    //<<DATA_INFO>>
    Svc.DefineClass ("CIM", "ITSO_CBESocket_Monitor",
    "root\default:ITSO_CBESocket_Monitor", ",", "Num", "Port", "None", ",", 0, 1);
    //<<\DATA_INFO>>

    // Event definition section
    //<<EVENTS_INFO>>
    Svc.DefineEvent ("Ev_ITSO_CBESocket_Monitor_Num_too_high",
    "UpperBound,Num", "Port");
    //<<\EVENTS_INFO>>

    // Logging definition section
    //<<LOGGING_INFO>>
    Svc.DefineLogInst ("ITSO_CBESocket_Monitor_Availability",
    "ITSO_CBESocket_Monitor", "Port", "Num", "Port");
    //<<\LOGGING_INFO>>

    // Place your additional initializing code below
}
```
Init()

The Init() function (Example 6-33) by default does not contain any code. We have to modify this function so that we initialize various parameters. This function identifies whether the running platform is Windows or UNIX®. If it is Windows, the action script to be run is Actions.bat; otherwise, it is Actions.sh. This function validates that the parameters required for the resource model are given and associates the same to the instance of the resource model.

Example 6-33  Init function for ITSO_CBESocket_Monitor resource model decision tree script

```javascript
var interpType = "" ; // Store the operating system type
var actionScript = "" ; // Store the script run when the event is generated

function Init(Svc)
{
    Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Init entered");
    var dimension = 0;
    var t = 0;

    interpType = Svc.GetInterp();
    Svc.Trace(TRACE_FINEST, TRACE_SOURCE + "Init interpType is : "+interpType);

    if ( interpType == "w32-ix86")
    {
        actionScript = "Actions.bat"
    } else
    {
        actionScript = "./Actions.sh"
    }

    dimension = Svc.GetStrParameterCount("ITSO_CBESocket_Monitor_Port");
    if ( dimension > 0 )
    {
        Svc.Trace(0,"ITSO_CBESocket_Monitor RM Associating parameter for port ");
        Svc.AssociateParameterToClass("ITSO_CBESocket_Monitor_Port","ITSO_CBESocket_Monitor");
    }

    dimension = Svc.GetStrParameterCount("CBE_search_Pattern");
    if ( dimension > 0 )
```
VisitTree()

This method is called after every cycle time (60 seconds in this case) and is the function that can include the logic for the resource model.

For this resource model, when the number of occurrences of the search pattern goes above five, the script has to be run. This is done by the ShellCmd method of the Service Object, as shown in Example 6-34.

Example 6-34  VisitTree function for ITSO_CBESocket_Monitor resource model decision tree script

```javascript
function VisitTree(Svc)
{
    //vars for data source: ITSO_CBESocket_Monitor
    var curITSO_CBESocket_MonitorNum;
    var curITSO_CBESocket_MonitorPort;

    var hPropTable;
    var numOfInstances;
    var idx;

    var ParamCount;
    var ParamIdx;
    var Different;

    hPropTable = Svc CreateMap();

    //triggering logic for data source: ITSO_CBESocket_Monitor
    numOfInstances = Svc.GetNumOfInst("ITSO_CBESocket_Monitor");
    for(idx = 0; idx < numOfInstances; idx++){
        Svc.RemoveMapAll(hPropTable);
    }
}
```
//Handle numeric properties
curITSO_CBESocket_MonitorNum = 
Svc.GetNumProperty("ITSO_CBESocket_Monitor", idx, "Num");
Svc.SetMapNumElement(hPropTable, "Num", curITSO_CBESocket_MonitorNum);

//Handle string properties
curITSO_CBESocket_MonitorPort = 
Svc.GetStrProperty("ITSO_CBESocket_Monitor", idx, "Port");
Svc.SetMapStrElement(hPropTable, "Port", curITSO_CBESocket_MonitorPort);

if (curITSO_CBESocket_MonitorNum > Svc.GetThreshold("Thr_Num_gt")) {
   Svc.SetMapNumElement(hPropTable, "UpperBound", Svc.GetThreshold("Thr_Num_gt"));
   Svc.SendEventEx ("Ev_ITSO_CBESocket_Monitor_Num_too_high", hPropTable);
   var shellRC = Svc.ShellCmd(actionScript);
   var str_result = Svc.GetShellStdOut();
   var str_err = Svc.GetShellStdErr();
   Svc.Trace(TRACE_FINEST,TRACE_SOURCE+"Returned result from actionScript is: "+str_result);
   Svc.Trace(TRACE_FINEST,TRACE_SOURCE+"Returned error from action Script is: "+str_err);
}
Svc.LogInstEx ("ITSO_CBESocket_Monitor_Availability","ITSO_CBESocket_Monitor", hPropTable);
Svc.DestroyMap(hPropTable);
return (0);

Various other trace statements have been added for debugging purposes.

**Building the ITSO_CBESocket_Monitor Resource Model Package**

We have now furnished all the required details to the Resource Model Builder. The package consisting of various components like configuration files, dependencies, JavaScript, MOF file, Resource Model Bundles, Resource Model Context XML files, Java Messaging catalogs, and the CIM data source is built using the Resource Model Builder. To build the package, select **ITM → Generate Package → Tame (zip)** and specify the file name as ITSO_CBESocket_Monitor.zip.

The generated zip file contains all the required files that are used by the Autonomic Management Engine for the platforms supported.
6.5 Contents of Resource Model Package

This section briefly describes the contents of the Resource Model Package generated from the Resource Model Builder. 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
- CIM classes, which implement M12 providers
- Native libraries, if any
- Scripts for action, if applicable
- Dependencies
- Platform specific packages

The Autonomic Management Engine embedding environment unpacks this package for deploying. The platform specific package in turn contains the specific files required for the deployment of the package on specific platforms.

A general convention used for representing the platform specific package inside the Resource Model Package is:

__<interp>__<RMTyp>.zip

Where:

**interp**
Stands for the platforms supported by the resource model. This can currently be w32-ix86, linux-ix86, or aix4-r1.

**RMTyp**
Refers to the name of the resource model. A resource model can have multiple instances in the AME environment and can have different names for the instances.

The contents of a specific Resource Model Package is shown below:

```xml
<RMTyp.zip> {
  Security Catalog ( RMTyp.cat)
  Configuration file ( RMTyp.conf)
  RMTypDescription.zip
  __win32-ix86__RMTyp.zip
  __linux-ix86__RMTyp.zip
  __aix4-r1__RMTyp.zip
}
```

The description package for the resource model will contain specific XML configuration files containing various parameters, a message catalog, a security
catalog, a baroc file consisting of parameter information, and the
<RMType>.class file, which contains the information about the resource bundle.

The contents of the platform specific Resource Model Package is shown below:

__interp__<RMType>.zip {
Decision tree script ( RType.js )
XML description of the RM ( RType.xml )
Message catalog ( RType.cat )
Resource Bundle information ( ResourceBundle_<locale>.class )
MOF files ( RType.MOF )
ILT implementation class or jar file
Dependencies if any
}

A detailed description of the resource model descriptor configuration XML file is
given in the Autonomic Management Engine Developer's Guide. It is suggested
that the user becomes familiar with the various tags used in these configuration
files.

The hosting AME environment parses this XML files for initializing various
parameters associated with the resource model. A typical resource model
descriptor file may have information embedded in tags, as shown below:

<Rm typeName="type", category="category", version="major.minor",
rmSpecification="major.minor"> <Rm>
<Description></Description>
<Identify></Identify>
<Analyzer></Analyzer>
<CycleTime></CycleTime>
<DecFile></DecFile>
<Modes></Modes>
<Thresholds></Thresholds>
<StrParameters></StrParameters>
<Parameters></Parameters>
<Aggregator></Aggregator>
<Indication></Indication>
<ClearingPolicy></ClearingPolicy>
<NotificationPolicy></NotificationPolicy>
(MsgInfo></MsgInfo>
<KeyFields></KeyFields>
<Rule></Rule>
<Consolidation></Consolidation>
<Severity></Severity>
<Notify></Notify>
<Cure></Cure>
<DataLog><DataLog>
The parser embedded in the Autonomic Management Engine hosting environment reads this XML data to parse and take appropriate actions as necessary.

6.6 Summary

Resource models constitute an important part of autonomic solutions. Within the context of a resource, the resource model can provide the monitoring, analysis, plan, and execute functions of the autonomic control loop.

Resource models are designed and deployed using the Autonomic Management Engine to suit the specific needs of IT solutions. The Resource Model Builder provides various built in tools and infrastructure to build such Resource Model Packages.

The next chapter describes the Autonomic Management Engine environment and how these resource model packages are deployed.
Chapter 7. Autonomic Management Engine

This chapter discusses the Autonomic Management Engine hosting environment, writing embedding applications for AME, setting up the environment, and launching the embedding AME application with resource models. 7.4, “Advanced topics” on page 202 provides a brief overview of adding custom action launchers.

It is recommended that the user also refers to the Autonomic Management Engine Developer’s Guide (provided with the Autonomic Computing Toolkit) while reading this chapter.
7.1 Introduction to Autonomic Management Engine

The Autonomic Management Engine (AME) provides a hosting environment for the resource model decision algorithms. The autonomic resource models are created using JavaScript and, in a broader perspective, the Autonomic Management Engine (AME) is responsible for providing a JavaScript engine for running these decision algorithms.

One of the important tasks for AME is to provide the required Service Object for resource model JavaScripts. JavaScripts used in the resource models use the Service Object API to access various information. A detailed description of the Service Object API information can be found in Appendix B, “Service object method library”, of the *IBM Tivoli Monitoring Resource Model Builder User’s Guide v1.1.0*, SC32-1391.

The Autonomic Computing Toolkit is shipped with a sample Autonomic Management Engine (sometimes referred to as TAME). It has a core implementation that provides all the basic building blocks for running a resource model. The decision algorithm of the resource model (the VisitTree() method) is run by the AME 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. The AME core performs further analysis on the given resource model descriptor to make decisions as to how the resource model should be run.

AME can be further configured with event sinks and action launchers discussed later in this chapter, to forward events and take actions. These events and actions will be read from the resource model descriptor and AME performs actions and forwards events based on the event sinks and action launcher plug-ins associated with it.

To enable the Autonomic Management Engine to recognize and run the resource model, an application embedding the AME is written. This embedding application is responsible for loading the resource model, initializing instances of the resource models, and providing event syncs and action launcher plug-ins, if required. Figure 7-1 on page 185 gives an overall picture of the hosting environment.
As shown in Figure 7-1, the AME hosting environment uses a local database to store the various data required for its operation. In the current version of AME supplied with the Autonomic Computing Toolkit, an IBM Cloudscape™ database instance is used.

As discussed in the previous chapter, resource models are associated with Common Information Model classes (CIM classes), which are responsible for fetching data from the resource that is being monitored. The AME core consists of a Common Information Model Object Manager (CIMOM).

CIMOM is responsible for:
- Loading the required classes
- Compiling MOF files and registration of the CIM data source, when required
- Loading of M12 Java providers or ILT classes

CIMOM exposes different variables requested through the Service Object APIs and is a critical part of the AME hosting environment.
7.2 Writing an embedding application for AME

The AME core exposes various APIs and interfaces for applications. Embedding applications are responsible for implementing the required interfaces and plug-ins to enable the AME core to perform the required operations.

Figure 7-2 gives an overview of the embedding application and the associated components.

![Diagram of AME core components](image)

Figure 7-2 Embedding application and related components

The AME documentation can be referred to for more information on the APIs, interfaces, and internal working modules.

7.2.1 Designing the embedding application

The embedding application is responsible for providing the implementation for interfaces and calling 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 following are the tasks of an embedding application:

- Implement RMIdentifier and RMIdentifierPattern interfaces.
- Provide the required plug-ins for action launchers and event sinks.
- Configure rme.config for these plug-ins.
- Make API calls to load resource model instances to the AME core.

**Interfaces**
There are two interfaces 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. 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 between 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 our example embedding application, we return the name of the RMIdentifier as a unique identifier. There can be various 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 gives the implementation of the RMIdentifier implementation class called RMIdentifierImpl.java.

```java
Example 7-1  RMIdentifierImpl.java implementation

package com.itso.autonomic;

import com.ibm.amw.util.*;  //Java class
import com.ibm.amw.rme.RMIdentifier;  //RME.jar public interface
import java.io.Serializable;  //Java class

public class RMIdentifierImpl implements RMIdentifier, Serializable {

    // The name of a resource model instance
    private String _name;

}```
private String _type;
private String _context;

public String getName() {
    return _name;
}
public String getType() {
    return _type;
}
public String getContext() {
    return _context;
}

public RMIdentifierImpl (String name, String type, String context) {
    Assert.that(name!=null);
    Assert.that(type!=null);
    _name = name;
    _type = type;
    _context = context; if(_context==null)_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 unique identifier.
public String getKey () {
    return _name;
}

// 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 can 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.
For clarification purposes, we show a RMIdentifierPatternImpl class that implements the RMIdentifierPattern public interface, which is used either to return all the resource model identifiers by specifying a wild card ("*"), or return a specific resource model identifier whose name matches the pattern.

For this purpose, a simple pattern class is created, as shown in Example 7-2, with self-explanatory comments.

**Example 7-2  Example of class implementing RMIdentifierPattern interface**

```
package com.itso.autonomic;

import com.ibm.amw.rme.RMIdentifier;
import com.ibm.amw.rme.RMIdentifierPattern;

// Basic matching utility class.
// If the requested pattern is * or *.* then we are looking for all the available
// instances of the Resource Model. Otherwise a specific resource model, whose
// name is mentioned.

final class Simple_Pattern {

    private char[] _pattern;

    public Simple_Pattern(String pattern) {

        StringBuffer b = new StringBuffer(pattern);
        // manipulate the pattern so that, if it ends with *.*, make it ends with *

        if ( pattern.endsWith(".*") ) {
            b.delete(b.length()-2, b.length());

```

```
pattern = b.toString().toCharArray();

// This method returns true if the pattern to be matched is *
// or the given Resource Model Name.
// When the AME searches for a list of available RM instances
// all the instances are returned when the below call is made
// RMIdentifierPatternImpl allrms = new RMIdentifierPatternImpl("*", "*", "*");
// If the embedding AME application wants a RMIdentifierPatternImpl object
// of a specific Resource Model, embedding application will make a call as shown
// below
// RMIdentifierPatternImpl thisrm = new RMIdentifierPatternImpl(rmName, "*", "*");
// Refer to the ITSO_AME.java for more details.

public boolean matches(String s) {
    String _patterns = new String(_pattern);

    if ( _pattern.length==1 && _pattern[0]=='*' ) {
        return true;
    } else if( s.equals(_patterns.trim())){
        return true;
    }
    return false;
}

// Implementation for RMIdentifierPattern public interface
// This can be extended to use regular expressions to find the
// instances of the resource model.
// This applies to only complex environments and AME with multiple

public class RMIdentifierPatternImpl implements RMIdentifierPattern {
    private Simple_Pattern _rmNamePattern;
    private Simple_Pattern _rmTypePattern;
    private Simple_Pattern _rmContextPattern;

    // Constructor for different patterns
    // In ITSO example, we do not use Type or Context pattern matching, However the same
    // algorithm
    // can be extended to do so..

    public RMIdentifierPatternImpl(String rmNamePattern, String rmTypePattern, String rmContextPattern) {
        _rmNamePattern = new Simple_Pattern(rmNamePattern);
        _rmTypePattern = new Simple_Pattern(rmTypePattern);
        _rmContextPattern = new Simple_Pattern(rmContextPattern);
    }
Embedding application class - ITSO_AME.java

The APIs provided by the AME core can be obtained in the AME documentation provided along with the AME Toolkit. The embedding application should use these API calls to set up the hosting environment for the AME core.

The following are the responsibilities of any application embedding itself into the AME core:

▶ Set the required CLASSPATH and environment variables required for the execution of the AME core. This requires setting the CLASSPATH and giving the required parameters for logging and RMI policy file location for running the AME application.

▶ Start the Autonomic Management Engine, also referred to as the resource model engine (RME).

▶ Install the resource model type from the resource model zip file.

▶ Install any context files, if applicable.

▶ Create an instance of the resource model. Create multiple instances, if applicable.

▶ Start the resource model instance(s).

The AME APIs are called appropriately to perform the above sequence of functions for the embedding application. Along with these set of required functions, various other functions can be used to:

▶ Uninstall resource model types
A detailed implementation, along with documentation, is provided in the ITSO_AME.java sample (see Appendix B, “Additional material” on page 257). A few important pieces of the embedding application are discussed below. 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 IT solution needs.

The embedding application class ITSO_AME.java declares a few underlying AME classes.

As seen in Example 7-3 the Autonomic Management Engine, RMPackageManager, RMManager, ContextManager, HistoricalDataReader, and RMBundleManager are declared.

**Example 7-3  ITSO_AME embedding application declarations**

```java
public class ITSO_AME
{
    // Create an instance of the Autonomic Management Engine embedding application
    // This can be accessed through getInstance static method
    private static ITSO_AME _self = null;

    // Define underlying Resource Model Engine classes

    private EngineFactory _rme;
    private RMPackageManager _rmPackageManager;
    private RMManager _rmManager;
    private ContextManager _contextManager;
    private HistoricalDataReader _historicalDataReader;
    private RMBundleManager _rmBundleManager;
}
```

When the embedding application starts the AME, these values are obtained by static references, as shown in Example 7-4 on page 193. Please refer to the ITSO_AME.java start() method for further documentation.
Example 7-4  Initializing the resource model management engine’s declarations

```java
_rme = EngineFactory.getInstance(appName, mode);
_rmpackageManager = _rme.getRMPackageManager();
_rmManager = _rme.getRMManager();
_contextManager = _rme.getContextManager();

_historicalDataReader = null;
try
{
    _historicalDataReader = _rme.getHistoricalDataReader();
}
catch (RMEException e)
{
    // Data collection feature not enabled.
    TameLog.tr.exception(Level.DEBUG_MIN, "ITSO_AME", "Data collection
    feature is not enabled ? ..", e);
}

// Initialize the RMBundleManager
// anchor dir and the interp is set here ..

_rmbundleManager = RMBundleManager.getInstance(_rme);
```

A facility to use the previously installed resource model bundle is provided in the embedding application. If this facility is not required, then the previously installed resource model bundle is removed before installing the resource model from the zipped resource model package, as shown in Example 7-5.

Example 7-5  Excerpt from ITSO_AME.java

```java
_rmbundleManager = RMBundleManager.getInstance(_rme);

if (!mode)
{
    // Remove all RM bundles
    String[] rms = _rmbundleManager.listRMTypes();
    for (int n = rms.length - 1; n >= 0; --n)
    {
        try
        {
            _rmbundleManager.removeRMBundle(rms[n]);
        }
        catch (Exception e)
        {
            RMEException re = new RMEException(e);
            throw re;
        }
    }
}
The next step is to install the resource model type by creating a ZipInputStream for the “<RMType>.zip” given as a parameter to the application, as shown in Example 7-6.

```
Example 7-6   Calling the installRMType function in ITSO_AME.java

installedRMTyp = _rmBundleManager.installRMType(z, true);
```

The “z” in the above 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 needs to be replaced.

The next step is to create an instance of the resource model. The instance name used in Example 7-7 is “RMINSTANCE”, and a utility function called createRMInstance is called to create the resource model instance.

```
Example 7-7   Creating an RM instance in ITSO_AME.java

createRMInstance("RMINSTANCE", ResourceModel, null, null);
```

The createRMInstance method creates an instance of the RMIdentifierImpl (implementing RMIdentifier interface), creates an instance using RMIdentifier and the context (if any), and loads into the autonomic resource model manager. Example 7-8 provides a description of this function with comments.

```
Example 7-8   Creating the resource model instance in ITSO_AME.java

public void createRMInstance(String rmName, String rmType, String rmCtx, String rmDescriptor)

// This creates the Resource Model instance.
// After creating the RM Type, and the context ( which is null here ), an instance is created.
// There can be multiple instances of the same resource model in the Autonomic Management Engine
// which can monitor different types of resources.
// Parameters
// rmName - name of the instance. It must be unique in the Engine and cannot be null
// rmType - Type of the Resource Model ( Ex: ITSO_CBELog_Monitor )
// rmCtx - Context that is associated with the instance. This is created by the createContext.
// If there are no contexts, it can be null
// rmDescriptor - The settings of the Resource model instance. The default resource model descriptor settings file will be used if this parameter is null.
```
The next step is to start the resource model instance called the "RMINSTANCE". This is accomplished by the startRMInstance() method, as shown in Example 7-9 on page 196.
Example 7-9   Starting the resource model Instance in ITSO_AME.java

// Start the Resource Model instance

public void startRMInstance(String rmName) throws RMNotFoundException,
RMOperationFailureException
{
    checkStarted();
    RMIdentifierPatternImpl thisrm = new RMIdentifierPatternImpl(rmName,
    "*", "*");
    RMIdentifier[] rmids = _rmManager.getIdentifiers(thisrm);

    if (rmids != null && rmids.length == 1)
    {
        _rmManager.start(rmids[0]);
    }
    else
    {
        throw new RMNotFoundException(new RMIdentifierImpl(rmName, "-",
        ")
    }
}

Various other utility functions are coded and documented in the ITSO_AME.java sample. A few of the functions are not used, but are provided for educational purposes. Please refer to the ITSO_AME.java source code for more details.

Stopping the embedding application

The embedding application in a real environment may run indefinitely and need only be stopped when the entire Autonomic Solution is shut down. However, a method to perform the shutdown of such an embedding application still should be developed. There is no defined interface in the current level of the Autonomic Computing Toolkit to perform this function.

As a workaround, we can create a thread when the embedding application starts the instance of the resource model type. This thread monitors for a file named "STOP" in the current folder and terminates the program when it is created.

Another script can be used to create the STOP file. This script might be called stop_ame, and this command would result in stopping the embedding application. The code snippet shown in Example 7-10 on page 197 is the thread used to monitor for such a file.
Example 7-10 Stopping the embedding AME application

class StopThread extends Thread
{
   // This method is called when the thread runs
   public void run()
   {
      while (true)
      {
         boolean exists = (new File("stop_ame")).exists();
         try
         {
            sleep(1000);
         }
         catch (InterruptedException e)
         {
            System.out.println(e.toString());
         }
         if (exists)
         {
            boolean success = (new File("stop_ame")).delete();
            System.exit(0);
         }
      }
   }
}

7.2.2 Building and launching the AME embedding application

We now have a resource model package, the interfaces required by the AME core, and the AME embedding application.

Building the ITSO_AME.jar

The interface files and the ITSO_AME.java is compiled after setting the required CLASSPATH and PATH environment variables, as shown below. Figure 7-3 on page 198 shows the steps involved in creating the ITSO_AME.jar, consisting of the embedding application and the interfaces required by the AME core.
Figure 7-3  Building the AME embedding application

Setting up the configuration files
For running the embedding application, the AME core refers to a few 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 one can use these configuration files for advanced AME setup.

**rme.config**
This file consists of important parameters for the resource model engine. The following are the parameters referenced in this configuration file:

- **dir.*** parameters: These properties refer to the directory structure for storage of various components of the resource model package.
- **serviceability**: This refers to the fully qualified logging and tracing class.
- **eventsinks.***: Refers to various parameters required for an EventSink plug-in for the AME core. We have, as of now, used the default plug-ins, which are a part of the Autonomic Management Engine installation package. Please refer to the Advanced Topics chapter of the *Autonomic Management Engine Developer's Guide* for implementation details for EventSyncs.
- **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 CIM implementing class. The default value is to use the Tivoli M12 DMTF CIM mode class.
datalog.*: 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 various 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 may not need editing, but is a required property file for logging data for debugging purposes.

**Note:** The default configuration files can be found under `<AME_HOME>\sara\config` folder. These configuration files are part of the application Simple Agent Reference Application (SARA) shipped along with the AME tool. 7.3, “Simple Agent Reference Application (SARA)” on page 201 briefly discusses this application.

### Running the AME application
The embedding AME application should be invoked with the resource model type details and should take care of setting all the required classes in the CLASSPATH.

The script given in Example 7-11 describes various CLASSPATH settings required by the embedding application.

**Example 7-11  startAME.bat file**

```bash
@echo off
setlocal

REM SCRIPTS TO START THE AME

SET AME_TOP=C:\Program Files\IBM\AutonomicComputingToolkit\AME

SET ARCH=w32-ix86

REM SET THE PLUGIN JARS AND CLEAR THE OLDER CLASSPATH TO WORK AROUND THE LIMITATION FOR CLASSPATH LENGTH IN WINDOWS.
SET CLASSPATH=%AME_TOP%\lib\plugin.jar;%AME_TOP%\lib\tame-utils.jar;

REM ADD AME RELATED CLASSES
```
SET CLASSPATH=%AME_TOP%\lib\tame-core.jar;%AME_TOP%\lib\js.jar;%AME_TOP%\lib\db2j.jar;%CLASSPATH%

REM ADD THE DEFAULT PLUGINS PROVIDED WITH AME
SET CLASSPATH=.;%AME_TOP%\lib\eseif.jar;%AME_TOP%\lib\sara.jar;%AME_TOP%\lib\alcm.jar;%AME_TOP%\lib\logutil.jar;%AME_TOP%\lib\sal.jar;%CLASSPATH%

REM ADD NLS JARS
SET CLASSPATH=%AME_TOP%\lib\TameCoreMessages.jar;%AME_TOP%\lib\TameMessages.jar;%CLASSPATH%

REM ADD EXTERNAL JARS USED BY AME CORE
SET CLASSPATH=%AME_TOP%\lib\eif.jar;%AME_TOP%\lib\jlog.jar;%CLASSPATH%

REM ADD CLASSES FOR CIM SUPPORT
SET CLASSPATH=%AME_TOP%\lib\tame-m12-mode.jar;%CLASSPATH%

REM SET THE CLASSPATH FOR CONFIG DIRECTORY
REM THIS WILL BE SAME AS FOUND IN %AME_TOP%\sara\config DIRECTORY
SET CLASSPATH=%CLASSPATH%;.\config;

REM SET THE PATH FOR THE WORKING DIRECTORY OF AME
SET PATH=%PATH%;.\work\rmbundles\customscripts;.\work\rmbundles\lib;.\config;

REM * Add the PATH environment variable in the env.properties configuration file.
REM * It could be used by the Shell utility provided by the Tame Core Services.

echo PATH=.\work\rmbundles\customscripts;%PATH% > ".\config\env.properties"

REM SET THE CLASSPATH FOR THE EMBEDDING AME APPLICATION
set CLASSPATH=.;\lib\ITSO_AME.jar;%CLASSPATH%

REM SET THE CLASSPATH FOR THE DEPENDENCIES
REM mailapi.jar, smtp.jar and activation.jar is used only for sending emails.
REM See the Advanced section of the book.
SET CLASSPATH=.;\lib\dependencies\mailapi.jar;.;\lib\dependencies\gnu-regexp-1.1.1.jar;.;\lib\dependencies\smtp.jar;.;\lib\dependencies\activation.jar;%CLASSPATH%

rem Start the AME in this window ..
The embedding application ITSO_AME takes the resource model type as the argument for loading the resource model instances. The startAME command should be run with the associated resource model type. For example, if ITSO_CBELog_Monitor.zip is the resource model package, the batch file (or script files) can be run as:

```
startAME.bat ITSO_CBELog_Monitor
```

**Note:** Operating systems may have a limit on the length of the command or the length of the environment variables. This creates difficulty 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.

### 7.3 Simple Agent Reference Application (SARA)

The current version of Autonomic Management Engine tool is shipped with a sample embedding application called SARA. This tool can be used as a replacement to writing our own embedding application, as explained below. This tool is intended only for demonstration purposes.

The tool comes with the default plug-ins listed below.

- **Serviceability:** JLog based implementation plug-ins for logging the data.
- **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 TEC or such other servers.

SARA's configuration files are located in `%AME_HOME%\sara\config` folder, where AME_HOME is the installation directory of the AME tool.

SARA provides a command line interface for various actions that can be programatically executed by any typical embedding application.

Figure 7-4 on page 202 shows the command line options of the SARA application. As seen in the figure, various operations done by the embedding
application can be executed by command line parameters so that various instances of the resource model types are created and started using the command line.

![Command Prompt - sara.bat](image)

**Figure 7-4 Application SARA shipped with AME tool**

The AME documentation provided with the tool provides detailed information and is a suggested read for understanding SARA and other API methods in the AME core.

### 7.4 Advanced topics

The examples provided in this and the previous chapter can be further customized to meet specific IT solution needs. This can be done by extending the capabilities in some areas of the resource models and autonomic management embedding application. This section discusses some of the options that are available for such extensions.
7.4.1 Adding actions and events to resource models

In 6.3.3, “Generating the resource model using Resource Model Builder” on page 111, we discussed how to add events based on a triggering condition.

We generated an event called Ev_ITSO_CBLog_Monitor_Num_too_high when the number of occurrences of a specific resource model parameter exceeds a threshold of “Thr_Num_gt”. We also ensured that we unchecked the Send to TEC or TBSM options so that we did not send event actions.

Refer to Figure 7-5 as a reminder of the configuration we created.

![Figure 7-5 Events tab in the resource model](image)

One can choose to send this event to the Tivoli Enterprise Console or Tivoli Business Systems Manager. In this case, the embedding AME application has to implement the plug-in and set up the required environment.
In a similar fashion, we can also add actions instead of executing the shell script using the Svc.ShellCmd API, as explained in previous chapters.

In Figure 7-5 on page 203, we can click on the **Actions** and associate actions through the Resource Model Builder, as shown in Figure 7-6.

![Figure 7-6  Adding Actions to the resource model](image)

The resource model can be added to:

- Run a program by giving the full path
- Run a shell script with its parameters
- Run a CIM method

**Adding a shell command**

When an action, as shown in Figure 7-6, is added to the resource model, the Resource Model Builder converts these actions into a set of XML instructions. After re-building the resource model package, we can see that the Resource Model Builder adds the following XML statements (Example 7-12) to the resource model descriptor XML.

**Example 7-12  Resource model descriptor showing actions**

```xml
<Cure>
  <!--This shell action will be launched for the given indication-->
  <shell:ActionLauncher name="shell">
    <shell:Indication name="Ev_ITSO_CBELog_Monitor_Num_too_high">
      <shell:Actions xsi:type="shell:ShellCmds">
        <shell:ShellCmd ExecutableFile="Actions.bat">
          <shell:DescriptiveName Catalog="ITSO_CBELog_Monitor" Default="Actions.bat" Id="36"/>
        </shell:ShellCmd>
      </shell:Actions>
    </shell:Indication>
  </shell:ActionLauncher>
</Cure>
```

**Note:** Tivoli Enterprise Console and Event Plug-ins are not discussed in the context of this redbook. For more details, please refer to the documentation provided with the AME toolkit. However, the action launchers specified in the following sections can be referred to for similarities with the event sinks.
The next step is to configure the AME environment so that the AME core understands this set of XML instructions to execute the required Action.

The shell command executor plug-in is provided by default with the AME. The next section explains how to configure the AME embedding application to use these plug-ins.

**Adding a CIM method**

Any CIM methods can be added to a resource model. The descriptor changes accordingly for any event with the extra `<Cure>` tag, indicating that a method has to be run on this occurrence of the event. This identification by the embedding AME application is achieved by implementing a corresponding plug-in to run these methods.

To enable the AME embedding application to make use of such actions, the Resource Model Builder can be used to associate CIM methods as actions to events.

What follows are the steps to add CIM methods as actions to events. This explanation assumes that the user is familiar with the concepts discussed in previous chapters. We extend the ITSO_CBELog_Monitor resource model and add a method called sendEmail, which should send an e-mail to an administrator, for example, that an event has occurred that needs attention.

1. Modify the ITSO_CBELog_Monitor.MOF file so that the data source is capable of performing an INVOKE function. There is a need for a CIM method to be declared in the .MOF file so that the CIM method is identified by the embedding AME application. Example 7-13 gives the modified .MOF file where the INVOKE M12Instrumentation and the sendEmail method has been added.

   **Example 7-13   MOF file for associating CIM methods with Events**

   ```plaintext
   //////////////////////////////////////////////////////////////////////////
   //
   // Filename : ITSO_CBELog_Monitor.mof
   //
   ```
// Description:
// Gives textual definition for the ITSO_CBELog_Monitor CIM class.
//
// File History:
//
// Date                        Author       Reason/Description
//------------------------------------------------------------------------------
// 01/26/2004                  ITSO Redbook team   Initial file
//------------------------------------------------------------------------------

[Description ("ITSO_CBELog_Monitor: This sample resource model is created to"
  "educate the readers about the resource model concepts."
  "This resource model monitors a file name <Key parameter> "
  "FileName </Key parameter>"
  "and return the number of occurrences of the regular expression"
  "given as a parameter to the resource model. This resource"
  "model exhibits only a part"
  "of the functionalities provided by M12 qualifiers.")],
provider("com.tivoli.dmunix.ep.touchpoint.cimom.ifc.M12JavaProvider"),
M12_Instrumentation {
  "Java.com.ibm.tivoli.monitoring.ITSO_CBELog_Monitor | | ENUM",
  "Java.com.ibm.tivoli.monitoring.ITSO_CBELog_Monitor | | GET",
  "Java.com.ibm.tivoli.monitoring.ITSO_CBELog_Monitor | | INVOKE"
}
]
class ITSO_CBELog_Monitor
{
  //Attributes for the ILT implementation
  [Description("The Filename property is provided by
  ITS_CBELog_Model_FileName as"
    "a parameter to the Resource Model.")],
    key ]
    string FileName;

  sint32 Num; // Attribute, Integer returning the number of occurrences.

  [Description("A CIM method which is used to send an Email after an
  event occurs ..")]
  string sendEmail();
};

2. The ILT class has to be modified so that the AME embedding application
invokes this implemented method of the ILTInterface. As explained in 6.3.2,
“Instrumentation Library Type (ILT) Java class” on page 97, the ILT implementation class is modified so that the invokeMethod() function of this implementation calls the required method, as shown in Example 7-14.

Example 7-14  invokeMethod of the ILT implementation - ITSO_CBELog_Monitor.java

_paragraph_1 //Method Name : invokeMethod
_paragraph_1 //
_paragraph_1 //Syntax:
_paragraph_1 //  public java.lang.String invokeMethod( M12ObjectIdentity targetInstance,
_paragraph_1 //                      java.lang.String methodName,
_paragraph_1 //                      java.lang.String mappingString,
_paragraph_1 //                      ParameterSet parms,
_paragraph_1 //                      ParameterSet inParms,
_paragraph_1 //                      ParameterSet outParms)
_paragraph_1 //  throws M12Exception
_paragraph_1 //
_paragraph_1 //Parameters:
_paragraph_1 //  targetInstance - M12ObjectIdentity that identifies the instance
_paragraph_1 //    whose method has to be called.
_paragraph_1 //  methodName - The name of the method to be called.
_paragraph_1 //  mappingString - Any string that has been specified in the
_paragraph_1 //    M12_Instrumentation qualifier for the INVOKE
_paragraph_1 //    operation type for this method.
_paragraph_1 //  parms - A ParameterSet object filled by the client with
_paragraph_1 //    parameters for this method.
_paragraph_1 //  inParms - A ParameterSet object filled by the client with
_paragraph_1 //    parameters to be passed to the method.
_paragraph_1 //  outParms - A ParameterSet object created by the client and
_paragraph_1 //    filled by the method with output results.
_paragraph_1 //
_paragraph_1 //Description:
_paragraph_1 //  Invokes the specified method on the identified CIM instance. This is not
_paragraph_1 //  supported in this Resource Model, hence we will return null.
_paragraph_1 //
_paragraph_1 //Returns:
_paragraph_1 //  String - The result of the method. Result values have to be CIM
_paragraph_1 //  standard types and ILT converts them to string format
_paragraph_1 //  according to the CIM standards.
_paragraph_1 //
_paragraph_1 //Exceptions Thrown:
_paragraph_1 //  M12Exception
_paragraph_1 //
public String invokeMethod( M12ObjectIdentity m12objectidentity,
        String sMethodName,
        String sMapping,
        ParameterSet parameterset,
        ParameterSet parameterset1,
        ParameterSet parameterset2 )
        throws M12Exception
{
    trace.log(3, "ITSO_CBELog_Monitor", "invokeMethod(2): " +
            "sMethodName=" + sMethodName +
            ", sMapping=" + sMapping);

    try
    {
        if ( sMethodName.trim().equals("sendEmail") ) {
            sendEmail();
            return "0";
        }else {
            trace.log(3,"ITSO_CBELog_Monitor","Err: invokeMethod returning
null...
"n);
            return null;
        }
    }
    catch (Exception e) {
        trace.log(3,"ITSO_CBELog_Monitor","Exception in invokeMethod"+e);
        M12Exception exc = new M12Exception(e);
        throw exc;
    }
}

The sendEmail method uses the JavaTM Mail API to configure and send e-mail, as shown in Example 7-15.

Example 7-15  sendEmail method

///
///
/// Method Name : sendEmail
///
/// Description : This method is called by the CIM plugin and sends an email
to the
/// person in the to list.
void sendEmail() {
    try {
        Properties props = new Properties();
        props.put("mail.smtp.host","d23m0174.in.ibm.com");
        Session ss = Session.getInstance(props,null);
        MimeMessage message = new MimeMessage(ss);
        InternetAddress from = new InternetAddress("CBELog_Monitor");
        message.setFrom(from);
        InternetAddress to = new InternetAddress("devaprasad@itso.ibm.com");
        message.addRecipient(Message.RecipientType.TO,to);
        message.setSubject("Event Notification .. ");
        message.setContent("The CBE Monitor Resource Model has generated an Event. Please look at the application logs for taking further actions.","text/html");
        Transport.send(message);
        trace.log(3,"ITSO_CBELog_Monitor","Message sent ...");
    }catch(Exception e) {
        trace.log(3,"ITSO_CBELog_Monitor",e.toString());
    }
}

The sendEmail function uses the Java Mail APIs, so the imports (Example 7-16) for this ILT class need to contain the corresponding classes.

Example 7-16   ILT class imports

```
import javax.mail.*;       // Java email classes
import javax.mail.internet.*; // Java email classes
```

3. Compile the ILT class and create the jar file, as explained in “Compiling the ILT class” on page 112.

4. Follow the steps explained in Chapter 6, “Resource models” on page 85 to create the resource model ITSO_CBELog_Monitor with the new .MOF file and the ILT implementation package that you have created.

Note: If the data source ITSO_CBELog_Monitor has previously been created under the root\default name space, the older data source needs to be deleted, and then you need to refresh the name space and re-compile the new .MOF file.
5. We now should have the ITSO_CBELog_Monitor resource model package created using the steps mentioned in 6.3, “Implementing the ITSO_CBELog_Monitor resource model” on page 94, but with a modified ILT implementation package and a modified .MOF file.

6. Add the extra dependency of mailapi.jar and activation.jar in the dependencies section, as shown in Figure 7-7.

Figure 7-7 Adding the extra dependencies
Note: We can add these jar file dependencies in the dependencies section of the resource model. This becomes a part of the resource model package.

However, another way is to add these jar files in the class path before invoking the embedding AME application. In this example, we have not included these jar files in the additional material that can be downloaded from our Web site.

They should be downloaded externally from the following links:

- Java Mail APIs:

- Java Activation Framework:

7. Add the actions in the Events tab, so that the CIM method is added as one of the actions. The CIM methods in the data source should be selected with the proper description, as shown in Figure 7-8.

![CIM Action Browser](image)
After entering the CIM method, you can see that the CIM method is added to the actions list, as shown in Figure 7-9.

![Figure 7-9 Adding the CIM method as Actions to the resource model](image)

### Note
The description for the CIM method is a mandatory field. Depending on your screen resolution, you may need to resize the window so that you can enter the details.

8. Save the resource model project and rebuild the resource model package.

You can see, by inspecting the resource model descriptor file, that an extra `<Cure>` tag has been added, as shown in Example 7-17.

**Example 7-17 Excerpt from the RM Descriptor XML file**

```xml
<Cure>
  <!-- This shell action will be launched for the given indication -->
  <shell:ActionLauncher name="shell">
    <shell:Indication name="Ev_ITSO_CBELog_Monitor_Num_too_high">
      <shell:Actions xsi:type="shell:ShellCmds">
        <shell:ShellCmd ExecutableFile="Actions.bat">
          <shell:DescriptiveName Catalog="ITSO_CBELog_Monitor" Default="Actions.bat" Id="36" />
        </shell:ShellCmd>
      </shell:Actions>
    </shell:Indication>
  </shell:ActionLauncher>
  <!-- This CIM action will be launched for the given indication -->
  <cim:ActionLauncher name="CIMMethod">
    <cim:Indication name="Ev_ITSO_CBELog_Monitor_Num_too_high">
      <!-- Description for the CIM method is a mandatory field. Depending on your screen resolution, you may need to resize the window so that you can enter the details. -->
    </cim:Indication>
  </cim:ActionLauncher>
</Cure>
```
The resource model package can now be used with the embedding application or SARA to load an instance in the AME core for monitoring the resource.

Other custom <Cure> implementations can be manually added for any events by editing the descriptor XML file. This should comply with the ActionLauncher.xsd schema and should have an extended XML schema declared in the AME core. A detailed explanation of writing cure action launchers with corresponding plug-ins is given in the AME documentation.

### 7.4.2 Setting up the configuration files and environment

The Action Launchers used for the above Cure Action Launchers in the resource model are provided by the AME toolkit. Please refer to the additional material available for this redbook for the complete set of files.

The configuration files and the environment needs to be set up so that the AME embedding application can refer to these configuration files in the class path and create the resource model instance. What follows are the steps and relevant portions of the configuration and batch files:

1. Create a directory called “config” in the working directory, which will host all the related components of an autonomic control loop system.

2. Copy the rme.config, jlog.properties, and m12_mode.properties from the <AME_HOME>/sara/config directory to this directory.

   The configuration files provided with this version of AME are configured to use the default plug-ins. Any extra plug-in added needs to be configured in rme.config and is discussed in detailed in the AME documentation.

3. Copy the CIMMethod.xsd and ShellCmd.xsd schema files from <AME_HOME>/sara/config directory to the “config” directory that we created.

4. Modify the startAME.bat (see 7.2.2, “Building and launching the AME embedding application” on page 197) so that this working configuration directory is in the CLASSPATH as well as the PATH. Example 7-18 on page 214 is an excerpt from the startAME.bat.
Example 7-18  Excerpt from the startAME.bat for running the embedding AME app

```bash
rem SET THE CLASSPATH FOR CONFIG DIRECTORY
SET CLASSPATH=%CLASSPATH%;.\config;
....
SET PATH=%PATH%;.\work\rmbundles\customscripts;.\work\rmbundles\lib;.\config;
....
```

5. The example uses SMTP protocol to send e-mails. As explained above, we set the CLASSPATH to smpt.jar, mailapi.jar, and activation.jar in the startAME.bat, which launches the embedding AME application (see Example 7-19).

Example 7-19  Setting CLASSPATH

```bash
set
CLASSPATH=.;\lib\ITSO_AME.jar;.;\lib\smtp.jar;.;\lib\mailapi.jar;.;\lib\activation.jar;%CLASSPATH%
```

When AME is invoked along with other components of the autonomic control loop, which includes our sample application, the GLA mechanism to generate Common Base Events, and the AME embedding application with resource model, an e-mail will be sent when the event Ev_ITSO_CBELog_Monitor_Num_too_high is triggered.

The sample given above is very simple and for illustrative purposes. It can be further be extended to include complex algorithms and implemented based on business needs.

**Tip:** The tracing mechanism used in the AME core is very help for debugging the resource model and embedding AME application. The log file name and location mentioned while launching the AME embedding application can be used to track down any problems while running the embedding AME application. The example used here puts the log file called trace.log under the ./logs subdirectory.

### 7.5 Summary

The Autonomic Management Engine provides APIs to embed applications to deploy resource models. Various plug-ins can be designed for debugging, action launchers, or event sinks, to enhance the capabilities of the autonomic solution.

This chapter has provided some basic examples to get the developer started. Please refer to the AME documentation for additional details.
The third part of this book consists of a chapter that provides sample scenarios and implementations. These samples are based on examples that have already been shown in Part 2, but are organized to show more complete implementations at various autonomic maturity levels.
Application examples

In this chapter, we discuss and evolve an autonomic computing scenario based on a sample application that will serve as our managed resource. The sample application currently uses log files to record its events. We demonstrate how these events can be converted to Common Base Events and handled in different ways by an autonomic management engine.

We also show how to modify the application to generate Common Base Events directly without depending on log files as a source for Common Base Events.

The examples in this chapter are based on the various examples and topics covered in this redbook. The source code for our examples and other supporting files are available for download. Please see Appendix B, “Additional material” on page 257 for information about obtaining these sample files.
8.1 Sample application

The sample application that the examples in this chapter are based on is called SampleManagedResource. The application is simply a sample and performs no real function. However, it simulates a real application by randomly choosing various messages that it then writes to a log file called sample.log. It uses a helper class called LogWriter to write to the log. We are going to use this application to demonstrate how facilities provided by the IBM Autonomic Computing Toolkit can be used to manage it and demonstrate different autonomic maturity levels. This application can be found in the SampleManagedResource.jar that is part of the download. The source code is shown in Example 8-1.

Example 8-1  Sample managed resource source code

```java
package com.ibm.itso.autonomic.application;

import java.io.*;

class SampleManagedResource {
    LogWriter logWriter;
    public static String[] msgs = { "ITSO434W Resource utilization above 80%", 
                                 "ITSO945E Cannot connect to data source", 
                                 "ITSO259I Successful connection", 
                                 "ITSO314C Critical internal error-process exiting" };
    public static String startMsg = "ITSO001I SampleManagedResource starting...";
    public static String stoppingMsg = "ITSO900I SampleManagedResource stopping";
    public static String stoppedMsg = "ITSO901I SampleManagedResource stopped!!!";
    public static String restartMsg = "ITSO999I SampleManagedResource restarting";
    File startIndicator;
    File stopIndicator;
    File restartIndicator;
    boolean restart;
    public SampleManagedResource() {
        logWriter = new LogWriter("sample.log");
        startIndicator = new File("start");
        stopIndicator = new File("stop_app");
        restartIndicator = new File("restart");
        restart=true;
    }
    public static void main(String args[]) {
        System.out.println("Starting SampleManagedResource...");
    }
```
SampleManagedResource mainFrame = new SampleManagedResource();
mainFrame.start();
}

void start()
{
    while (restart)
    {
        restart=false;
        logWriter.writeToLog(startMsg);
        pause(2000);
        while (true)
        {
            logWriter.writeToLog(msgs[getRandomInt(0,4)]);
            pause(getRandomInt(1000,10000));

            if (restartIndicator.exists())
            {
                logWriter.writeToLog(restartMsg);
                restartIndicator.delete();
                try
                {
                    stopIndicator.createNewFile();
                } catch (Exception e)
                {
                    System.out.println("Error creating file: "+e.toString());
                }
                pause(2000);
                restart=true;
            }

            if (stopIndicator.exists())
            {
                logWriter.writeToLog(stoppingMsg);
                stopIndicator.delete();
                pause(2000);
                logWriter.writeToLog(stoppedMsg);
                break;
            }
        }
    }
}

// pause method - sleeps for milliseconds
private static void pause(int i)
{
    try
    {
        Thread.sleep(i);
    }
}
The LogWriter helper class is shown in Example 8-2. It is used by the sample application to write to the application log file.

Example 8-2   LogWriter class

```java
package com.ibm.itso.autonomic.application;

import java.io.*;
import java.util.Date;

class LogWriter
{
    String logFileName;
    File   logFile;
    FileWriter logFW;

    public LogWriter(String fn)
    {
        // Create or open logFile;
        try
        {
            logFileName = fn;
            logFile=new File(logFileName);

            if (logFile.exists())
                logFile.delete();
        } catch (Exception e)
        {
            System.out.println("Error: " + e.toString());
        }
    }
}
```
void writeToLog(String s) {
    BufferedWriter out;
    String logEntry;
    Date date = new Date(System.currentTimeMillis());
    logEntry = "[" + date.toString() + "] " + s;
    try {
        out = new BufferedWriter(new FileWriter(logFileName, true));
        out.write(logEntry);
        out.newLine();
        out.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
    }
}

This application will generate a log file with the generated messages.
Example 8-3 shows a sample of the generated log file.

Example 8-3   Sample log file generated by the SampleManagedResource application

Example 8-3

[Fri Jan 30 09:21:44 CST 2004] ITSO001I SampleManagedResource starting...
[Fri Jan 30 09:21:54 CST 2004] ITSO314C Critical internal error-process exiting
[Fri Jan 30 09:21:56 CST 2004] ITSO945E Cannot connect to data source
[Fri Jan 30 09:22:16 CST 2004] ITSO434W Resource utilization above 80%
[Fri Jan 30 09:22:24 CST 2004] ITSO945E Cannot connect to data source
[Fri Jan 30 09:22:26 CST 2004] ITSO259I Successful connection
[Fri Jan 30 09:22:56 CST 2004] ITSO0259I Successful connection
[Fri Jan 30 09:23:02 CST 2004] ITSO314C Critical internal error-process exiting
[Fri Jan 30 09:23:11 CST 2004] ITSO945E Cannot connect to data source
[Fri Jan 30 09:23:17 CST 2004] ITSO314C Critical internal error-process exiting
[Fri Jan 30 09:23:24 CST 2004] ITSO945E Cannot connect to data source

Mapping the sample application messages
Table 8-1 on page 222 represent the mapping of the sample managed resource messages to the Common Base Event situation information.
In the following sections, we discuss how this application may be managed at different maturity levels of autonomic computing. First, we use the Generic Log Adapter to generate Common Base Events and the Log and Trace Analyzer as a maturity level 2 autonomic manager. This primarily demonstrates the use and value of converting existing log file entries to Common Base Events.

Next, we exchange the LTA with AME as a maturity level 4 autonomic manager. In this case, the AME and associated resource model can take automated actions based on specific events.

The next example modifies the sample application to generate Common Base Events directly, eliminating the need for the Generic Log Adapter.

In the last two sections, we discuss how to use different kinds of communications mechanisms between the autonomic manager touchpoint and the AME, using sockets or a database.

Before we can demonstrate the autonomic capabilities, we first must get our sample application running.

To run the sample application:

1. Copy the SampleManagedResource directory to an appropriate directory.
2. Launch the application by:
   - Opening a command line
   - Changing the current directory to the SampleManagedResource directory

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Situation Type</th>
<th>Reasoning Scope</th>
<th>Success Disposition</th>
<th>Situation Qualifier</th>
<th>Situation Disposition</th>
<th>Report Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITSO001I</td>
<td>StartSituation</td>
<td>INTERNAL</td>
<td>SUCCESSFUL</td>
<td>START INITIATED</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO999I</td>
<td>StartSituation</td>
<td>INTERNAL</td>
<td>SUCCESSFUL</td>
<td>RESTART INITIATED</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO900I</td>
<td>StopSituation</td>
<td>INTERNAL</td>
<td>SUCCESSFUL</td>
<td>STOP INITIATED</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO901I</td>
<td>StopSituation</td>
<td>INTERNAL</td>
<td>SUCCESSFUL</td>
<td>STOPPED</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO259I</td>
<td>ConnectSituation</td>
<td>INTERNAL</td>
<td>SUCCESSFUL</td>
<td>n/a</td>
<td>CLOSED</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO945E</td>
<td>ConnectSituation</td>
<td>INTERNAL</td>
<td>UNSUCCESSFUL</td>
<td>n/a</td>
<td>CLOSED</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO314C</td>
<td>ConnectSituation</td>
<td>INTERNAL</td>
<td>UNSUCCESSFUL</td>
<td>n/a</td>
<td>CLOSED</td>
<td>n/a</td>
</tr>
<tr>
<td>ITSO434W</td>
<td>ReportSituation</td>
<td>INTERNAL</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>STATUS</td>
</tr>
</tbody>
</table>
8.2 Maturity level 2 - Using GLA and LTA

We can achieve maturity level 2 of autonomic computing by using the Generic Log Adapter and the Log and Trace Analyzer as tools to help the system administrator identify problems and possible solutions. First, the Generic Log Adapter is used to convert the legacy log file to Common Base Events by using the rules defined in the adapter configuration file. Secondly, the generated Common Base Events will be analyzed by the Log and Trace Analyzer to determine the event characteristics. Thirdly, a Symptom Database may be used to suggest actions for the analyzed events. Figure 8-1 gives an overview of the scenario.

First, we need to write the adapter configuration file that will be used by the Generic Log Adapter to convert the log file to Common Base Events. Below is the sample adapter file used with our application. The adapter configuration file used with our sample application can be found with the downloadable source. It is called sampleLTA.adapter. For more information about writing the adapter
configuration file, refer to 5.1, “Generic Log Adapter” on page 54 and the Introduction to the *Generic Log Adapter for Autonomic Computing* document provided as gla_getting_started.pdf with the Generic Log Adapter component of the IBM Autonomic Computing Toolkit.

An extract of the sampleLTA.adapter file is shown in Example 8-4. In this example, we used SingleOSFileSensor as an input class to read from the log file and CBEFileOutputter to store the generated Common Base Events in an output file called CBEOOut.log. If you are running this scenario more than once, clean or delete the old sample.log and CBEOOut.log files to store only the new events. The file clean.bat provided with the sample code will delete these files.

*Example 8-4  Example of sample adapter configuration file used by the GLA*

```xml
<?xml version="1.0" encoding="ASCII"?>
<adapter:Adapter ..........>
  <hga:Contexts>
    <hga:Context description="...." executableClass="org.eclipse.hyades.logging.adapter.impl.BasicContext" role="context" ....>......
      <hga:Component description="Operating System file sensor" executableClass="org.eclipse.hyades.logging.adapter.sensors.SingleOSFileSensor" role="sensor" .../>
      <hga:Component .........../>
    </hga:Context>
  </hga:Contexts>

  <cc:Configuration description="The component level configurations for this Adapter" uniqueID="....">
    <cc:ContextInstance ....>
      <cc:Sensor description="A single file sensor" maximumBlocking="5" type="SingleFileSensor" uniqueID="....">
        <sensor:SingleFileSensor directory="." fileName="sample.log"/>
      </cc:Sensor>
      <ex:Extractor ..../>
      <cc:Parser uniqueID="....">
        <parser:RuleElement index="....">
          <parser:RuleElement index="...." name="situation">
            <parser:RuleElement index="...." name="situationType">
              ........
            </parser:RuleElement>
          </parser:RuleElement>
        </parser:RuleElement>
        <parser:RuleElement index="....">
          <parser:RuleElement index="...." name="ConnectSituation">
            <parser:RuleAttribute index="...." name="reasoningScope" usePreviousMatchSubstitutionAsDefault="false">
              <SubstitutionRule match=".*ITSO259I.*|.*ITSO945E.*|.*ITSO314C.*" substitute="INTERNAL" useBuiltInFunction="false"/>
            </parser:RuleAttribute>
          </parser:RuleElement>
        </parser:RuleElement>
        ........
      </cc:Parser>
    </cc:ContextInstance>
  </cc:Configuration>
</adapter:Adapter>
```
If required, edit the adapter file to adjust the directory and file names for both the input log file defined in the sensor section and the output file defined in the ouputter section. Example 8-5 indicates what needs to be edited.

Example 8-5  Modify the input and output file attributes in the adapter file

....
<sensor:SingleFileSensor directory="c:\temp" fileName="sample.log"/>
....
<op:SingleFileOutputterType directory="c:\temp" fileName="CBEOut.log"/>
........

We can launch the Generic Log Adapter engine either by using a command line or by using a simple Java application. Example 8-6 on page 226 is a sample Java class to launch it. The following jar files will need to be located in the CLASSPATH:

> jakarta-oror.jar
> hglal.jar
> hexr.jar
> hlcmenus.jar
> hlcore.jar
> hlvents.jar
> xmlParserAPIs.jar
> xercesImo.jar
These files are provided as plug-ins to eclipse. Search for these files under 
<GLA_install>\lib, where <GLA_install> is the directory where the Generic Log 
Adapter was installed, typically in C:\Program 
Files\IBM\AutonomicComputingToolkit\GLA. The Java class to launch the 
generic Log Adapter is included in the glarunner.jar file included with our 
samples. There is a short cut to launch this program by running startGLA.bat, 
after copying the GLA_LTA directory to your system. For more documentation 
about using the Adapter class, see http://www.eclipse.org/hyades.

Example 8-6   GLA launcher program

```java
package com.ibm.itso.autonomic.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 should
be:
            ");
            System.out.println("java  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 Started ...");
                glaadapter.start( separateThread, daemon);
                // To stop the thread
                new StopGLAThread().start();
            } catch (AdapterException ex) {
                ex.printStackTrace();
            }
        }
    }
}```
Based on the file location and the outputter class specified in the adapter file used by the Generic Log Adapter, the outputter will direct the generated Common Base Events to the desired destination, either a file, socket, database, or other destination. We used the CBEFileOutputter to output the generated Common Base Events to a file. This file is called CBEout.log. CBEFileOutputter is part of org.eclipse.hyades.logging.adapter.outputters package. 8.3, “Maturity level 4 - Using GLA and AME” on page 228 gives an example of how to write a custom outputter.

Now we are ready to test our setup. We need to do the following in order:

1. Copy the GLA_LTA directory to the local machine.
2. Launch the application by:
   - Opening a command line
   - Changing the current directory to GLA_LTA directory
   - Typing `startApp.bat`
3. Launch the GLA engine by:
   – Opening a command line
   – Changing the current directory to GLA_LTA directory
   – Typing `startGLA.bat`

Now we will see a file called CBEOut.log generated. It contains the generated Common Base Event from the log file. Now we are ready to use these Common Base Events and analyze them using the Log and Trace Analyzer. For information on using it, please see 5.2, “Log and Trace Analyzer” on page 72.

### 8.3 Maturity level 4 - Using GLA and AME

We can use Generic Log Adapter with an autonomic management engine to achieve a maturity level 4 autonomic system. This can be done by using the generic log adapter to convert the log files into Common Base Events, and send those Common Base Events to the AME hosting environment. The AME will be responsible for analyzing the incoming Common Base Events and perform needed actions according to the defined logic in the resource model. For more information about autonomic levels, please refer to 1.3, “Evolving to autonomic computing” on page 12.

The resource model is discussed in detail in Chapter 6, “Resource models” on page 85. After building the resource model package ITSO_CBELog_Monitor.zip, as discussed in 6.3, “Implementing the ITSO_CBELog_Monitor resource model” on page 94, we can run the embedding AME application called ITSO_AME in the `com.ibm.autonomic` package with this resource model. This constitutes the autonomic management hosting environment. Building the ITSO_AME embedding AME application is discussed in Chapter 7, “Autonomic Management Engine” on page 183.

Figure 8-2 on page 229 explains the architecture of this example. We use the Generic Log Adapter with the sample.adapter file to convert the log file to Common Base Events. The generated Common Base Events will be forwarded using the CustomOutputter to the Autonomic Management Engine through the touchpoint APIs. The manager touchpoint will receive the incoming Common Base Events and store them to a file called CBEOut.log. The AME with the help of the resource model will read the stored Common Base Events from the CBEOut.log and perform actions accordingly.
To implement this example, we write our own outputter class, CustomOutputter (see Example 8-7 on page 230), to send the Common Base Events generated by
the Generic Log Adapter to the autonomic manager using the sendEvent() API provided by the AutonomicManagerTouchpointSupport object. Accordingly, we modify the adapter file to point to the new outputter class and rename it to sample.adapter. For more details about writing a custom outputter class, please check 5.1.6, “Writing a custom outputter” on page 70.

Example 8-7  CustomOutputter class

```java
package com.ibm.itso.autonomic.comm;

import java.rmi.RemoteException;
import org.eclipse.hyades.logging.adapter.*;
import org.eclipse.hyades.logging.adapter.impl.*;
import org.eclipse.hyades.logging.events.*;

public class CustomOutputter extends ProcessUnit implements IOutputter {
    private SampleManagedResourceTouchpoint smrtp = null;

    public CustomOutputter() {
        super();
        try {
            // create an instance and start it up
            smrtp = new SampleManagedResourceTouchpoint();
            smrtp.start();
        } catch (Exception exp) {
            exp.printStackTrace();
        }
    }

    public Object[] processEventItems(Object[] cbes) {
        int i;
        for (i = 0; i < cbes.length; i++) {
            if (cbes[i] != null) {
                try {
                    if (smrtp.getAutonomicManager() != null) {
                        smrtp.sendEventToManager((ICommonBaseEvent)cbes[i]);
                    }
                } catch (RemoteException rex) {
                    rex.printStackTrace();
                }
            }
        }
    }
}
```
catch (Exception e)
    { e.printStackTrace();
  }
}
return cbes;
}

// Used to simulate a Run time environment by testing the CBE's.
public Object[] testProcessEventItems(Object[] cbes) throws AdapterInvalidConfig
{
    if (!(cbes instanceof ICommonBaseEvent[]))
    {
        throw new AdapterInvalidConfig("This outputter will only accept arrays of
ICommonBaseEvent");
    }
    return cbes;
}

// Interface methods. Do nothing in this example.
public void update() throws AdapterInvalidConfig
{
    super.update();
}

The CustomOutputter will be responsible for launching the
SampleManagedResourceTouchpoint. Example 8-8 describes the sample
managed resource touchpoint.

Example 8-8 SampleManagedResourceTouchpoint class

```java
package com.ibm.itso.autonomic.comm;

import java.rmi.RemoteException;
import com.ibm.autonomic.resource.*;

public class SampleManagedResourceTouchpoint extends ManagedResourceTouchpoint
{
    public SampleManagedResourceTouchpoint() throws RemoteException
    {
        // In this example, we don't do anything. You can add additional
        // Initialization code here ..
        super();
    }

    public void start() throws RemoteException
```
Example 8-9 shows the code for the SampleAutonomicManagerTouchpointSupport. It will be responsible for receiving the incoming Common Base Events through the sendEvent() method and storing them to a file called CBEOut.log.

Example 8-9 The code for SampleAutonomicManagerTouchpointSupport

```java
package com.ibm.itso.autonomic.comm;

import com.ibm.autonomic.manager.*;
import com.ibm.autonomic.resource.*;
import org.eclipse.hyades.logging.events.ICommonBaseEvent;
import java.rmi.RemoteException;
import java.util.Iterator;
import java.io.*;

public class SampleAutonomicManagerTouchpointSupport extends AutonomicManagerTouchpointSupport
{
    private String cbeFileName = "CBEOut.log";
    private FileWriter cbeFile = null;
    IManagedResourceTouchpoint resolvedTP = null;

    public SampleAutonomicManagerTouchpointSupport() throws RemoteException
    {
        super();
    }

    public void start()
    {
        try
        {
            publish(this, "//localhost/SampleAutonomicManagerTouchpointSupport");
            while(resolvedTP == null)
            {
                try
                {
                    resolvedTP = resolveTouchpoint("//localhost/SampleManagedResourceTouchpoint");
                } catch (RemoteException rexp)
                {
                    try
                    {
                        Thread.sleep(3000);
                    }
                }
            }
        }
    }
```
catch(Exception exp)
{
    exp.printStackTrace();
}
System.out.println("Retry.");
}

// We have found a Resource Touch point. Assign itself as the manager.
// In this example, there is only one resource that the autonomic manager
// is managing. There can be multiple resources with their respective
// Resource touch points.

resourceTouchpoints.add(resolvedTP);
Iterator itr = resourceTouchpoints.iterator();
System.out.println(itr.toString());
while (itr.hasNext())
{
    IMangedResourceTouchpoint tempMRTP = (IMangedResourceTouchpoint) itr.next();
    tempMRTP.assignManager(this);
    System.out.println("Assigned Manager for " + tempMRTP("n");
}System.out.println("Manager started ...");
}

}catch(RemoteException rexp2)
{
    System.out.println("Could not publish the Autonomic Manager to the RMI registry");
    rexp2.printStackTrace();
}

} // This is manager's sendEvent method. Receives events from the outputer.
public void sendEvent(ICommonBaseEvent incbe) throws RemoteException
{
    try
    {
        try
        {
            if (cbeFile == null)
            {
                cbeFile = new FileWriter(cbeFileName, false);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new RemoteException("Cannot create CBEOut.log file : " + e.toString());
        }
    }
if (cbeFile != null)
{
    cbeFile.write/incbe.toString());
    cbeFile.write("\n");
    cbeFile.flush();
}
}
catch (Exception e)
{
    e.printStackTrace();
    throw new RemoteException("Cannot write the incoming CBE to file : " + cbeFileName + "\n" + e.toString());
}
}

public static void main (String[] args) throws RemoteException
{
    StopMGRTPTThread st = new StopMGRTPTThread();
    st.start();

    // create an instance and start it up
    SampleAutonomicManagerTouchpointSupport amts = new SampleAutonomicManagerTouchpointSupport();
    amts.start();
}

// This thread is incorporated to gracefully shutdown the Manager Touch point.
class StopMGRTPTThread extends Thread
{
    String stopfile = "stop_mgrtp";

    // 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);

The previous classes are included in the comm.jar provided in the sample code provided with the redbook. In order to implement this scenario, there are some required jar files. To have access to these jar files, we need to make sure we have done the following:

1. Install Generic Log Adapter.
2. Install AME.
3. Install ISC.
5. Install the PD Scenario.

**Important:** In the current version of the Autonomic Computing Toolkit, the amtapi.jar and CBEdoc.zip files are only installed when the PDScenario package is installed.

Jar files used by the Generic Log Adapter include:

- jakarta-oro.jar
- hgl.jar
- hexr.jar
- hlcommons.jar
- hlcore.jar
- hlevents.jar
- xmlParserAPIs.jar
- xercesImpl.jar

Jar files used by the touchpoints include:

- amtapi.jar (it comes as part of the PD Scenario found at <PDScenario>\amtapi\amtapi.jar)

For more details about the Autonomic Manager and the AME refer to Chapter 6, “Resource models” on page 85 and Chapter 7, “Autonomic Management Engine” on page 183.
In order to run the examples as described in 7.4, “Advanced topics” on page 202, the following jar files are required:

- gnu-regexp-1.1.X.jar (http://www.cacas.org/java/gnu/regexp)

For more details about the required jar files for the advanced topic scenarios, please refer to Example 7-11 on page 199.

**Tip:** Because we are using Java RMI, we need to include the jar files in the CLASSPATH for the Java RMI as well.

A batch file, runLoop.bat, is provided to launch the applications in sequence. The applications will be started in the following sequence:

1. RMIregistry (part of the Java installation)
2. Sample Application (provided in the SampleManagedResource.jar)
3. Generic Log Adapter (provided in glarunner.jar)
   a. Managed Resource Touchpoint (provided in comm.jar)
4. Manager Touchpoint Support (provided in comm.jar)
5. Embedding AME application with the Resource Model (provided by ITSO_AME.jar and ITSO_CBELog_Monitor.zip)

**Attention:** This scenario can be implemented without using the touchpoints by simply writing the generated Common Base Events directly to a CBEOut.log using the default CBEFileOutputter. However, this is not consistent with the autonomic computing architecture that specifies the use of touchpoints. Though our scenario is only running on a single system, the use of touchpoints will enable this scenario to run across multiple systems, as the underlying implementation of touchpoints is enhanced to support it.

## 8.4 Generating Common Base Events directly

In this section, we modify the sample application to generate Common Base Events directly instead of generating a log file and using the Generic Log Adapter to convert the log file to Common Base Events. The application will use the Common Base Event APIs provided by the Hyades project from Eclipse http://www.eclipse.org/hyades. The documentation for the Common Base Event APIs is provided with the Toolkit at <PDS>amtapi\lib\CBEjavadoc.zip. For more information about Common
Base Events and their structures, please refer to the *Autonomic Computing Toolkit Developer’s Guide*. Figure 8-3 describes the layout of the components used in this scenario.

Example 8-10 on page 238 shows how the sample application is modified to generate Common Base Events directly. We add a few methods to generate the Common Base Events. toCBE() converts the message string to an ICommonBaseEvent object. It calls getSituation() to convert the message string to an ISituation object. The generated Common Base Events will be sent to the manager touchpoint using sendEvent() or sendEventToManager(). This example uses the same classes defined in the comm.jar discussed previously.
Example 8-10  Sample application modified to generate Common Base Events directly

```java
package com.ibm.itso.autonomic.application;

import java.io.*;
import java.rmi.RemoteException;
import java.util.*;
import org.eclipse.hyades.logging.events.*;
import org.eclipse.hyades.logging.events.exceptions.*;
import java.text.SimpleDateFormat;
import com.ibm.itso.autonomic.comm.*;

class SampleManagedResourceWithCBE {
    .......
    SampleManagedResourceTouchpoint mgdResourceTP;
    IComponentIdentification sourceComp;
    IComponentIdentification reporterComp;

    public SampleManagedResourceWithCBE(){
        .......
        try{
            mgdResourceTP = new SampleManagedResourceTouchpoint();
            mgdResourceTP.start();
        }catch (RemoteException rexp){
            rexp.printStackTrace();
        }
        // Set the Source Component;
        sourceComp = setSourceComponent();
        reporterComp = setSourceComponent();
    }

    public static void main(String args[])
        throws MissingValueException, ValueTooLongException, RemoteException {
        System.out.println("Starting SampleManagedResource...");
        SampleManagedResourceWithCBE mainFrame = new SampleManagedResourceWithCBE();
        mainFrame.start();
    }

    void start() throws MissingValueException, ValueTooLongException, RemoteException {
        String msg;
        while (restart)
        {
            restart=false;
        }
    }
```
logWriter.writeToLog(startMsg);
mgdResourceTP.sendEventToManager(toCBE(startMsg));
pause(2000);
while (true)
{
  msg = msgs[getRandomInt(0,4)];
  logWriter.writeToLog(msg);
  ICommonBaseEvent cbe = toCBE(msg);
  mgdResourceTP.sendEventToManager(cbe);
  pause(getRandomInt(1000,10000));
  if (restartIndicator.exists())
  {
    logWriter.writeToLog(restartMsg);
    mgdResourceTP.getAutonomicManager().sendEventToManager(toCBE(restartMsg));
    restartIndicator.delete();
    try
    {
      stopIndicator.createNewFile();
    } catch (Exception e)
    {
      System.out.println("Error creating file: "+e.toString());
    }
    pause(2000);
    restart=true;
  }
  if (stopIndicator.exists())
  {
    logWriter.writeToLog(stoppingMsg);
    stopIndicator.delete();
    pause(2000);
    logWriter.writeToLog(stoppedMsg);
    break;
  }
}

private ICommonBaseEvent toCBE (String msg)
  throws MissingValueException,ValueTooLongException
{
  ICommonBaseEvent cbe;
  cbe = (CommonBaseEventImpl)
  SimpleEventFactoryImpl.getInstance().createCommonBaseEvent();
  short i=10;

  if (msg != null)
  {
    cbe.setSourceComponentId(sourceComp);
  }
cbe.setReporterComponentId(reporterComp);
cbe.setMsg(msg);
cbe.setCreationTime(getTime());

char c = msg.charAt(8);
switch (c)
{
    case 'I': // Information
        i = 20;
        break;
    case 'W': // Warning
        i = 30;
        break;
    case 'E': // Error
        i = 40;
        break;
    case 'C': // Critical
        i = 50;
        break;
    default:
        i = 20;
}
cbe.setSeverity(i);
cbe.setSituation(getSituation(msg));

return cbe;
}  
else
{
    return null;
}

private IComponentIdentification setSourceComponent()
{
    IComponentIdentification tempComp;
    try
    {
        tempComp = EventItemsFactory.createIComponentIdentification(
            /* componentIdType, */"Application",
            /* component, */"SampleManagedResource#1.0",
            /* subComponent, */"SampleManagedResource.start()",
            /* locationType, */"Hostname",
            /* location, */"server1.itso.ibm.com",
        )
    } catch (Exception e) {
        // Handle exception
    }

    // Set other properties if needed
    return tempComp;
}
private ISituation getSituation(String msg) throws MissingValueException, ValueTooLongException {

    ISituation situation;
    String msgID = msg.substring(0,8);

    System.out.println("The MsgID = " + msgID);

    if (msgID.equalsIgnoreCase("ITSO001I") || msgID.equalsIgnoreCase("ITSO999I"))
    {
        IStartSituation startSituationType =
            SimpleEventFactoryImpl.getInstance().createStartSituation();
        startSituationType.setReasoningScope("INTERNAL");
        startSituationType.setSuccessDisposition("SUCCESSFUL");
        if (msgID.equalsIgnoreCase("ITSO001I"))
        {
            startSituationType.setSituationQualifier("START INITIATED");
        } else// ITSO999I
        {
            startSituationType.setSituationQualifier("RESTART INITIATED");
        }
        situation = EventItemsFactory.createISituation("StartSituation", startSituationType);
    } else if (msgID.equalsIgnoreCase("ITSO900I") || msgID.equalsIgnoreCase("ITSO901I"))
    {
        IStopSituation stopSituationType =
            SimpleEventFactoryImpl.getInstance().createStopSituation();
        stopSituationType.setReasoningScope("INTERNAL");
        stopSituationType.setSuccessDisposition("SUCCESSFUL");
    } else if (msgID.equalsIgnoreCase("ITSO900I") || msgID.equalsIgnoreCase("ITSO901I"))
    {
        IStopSituation stopSituationType =
            SimpleEventFactoryImpl.getInstance().createStopSituation();
        stopSituationType.setReasoningScope("INTERNAL");
        stopSituationType.setSuccessDisposition("SUCCESSFUL");
    }
if (msgID.equalsIgnoreCase("ITSO900I"))
{
    stopSituationType.setSituationQualifier("STOP INITIATED");
} else if (msgID.equalsIgnoreCase("ITSO9901I"))
{
    stopSituationType.setSituationQualifier("STOP COMPLETED");
}
situation = EventItemsFactory.createISituation("StopSituation", stopSituationType);

if (msgID.equalsIgnoreCase("ITSO259I") || msgID.equalsIgnoreCase("ITSO945E") || msgID.equalsIgnoreCase("ITSO314C"))
{
    IConnectSituation connectSituationType =
        SimpleEventFactoryImpl.getInstance().createConnectSituation();
    connectSituationType.setReasoningScope("INTERNAL");
    connectSituationType.setSituationDisposition("CLOSED");
    if (msgID.equalsIgnoreCase("ITSO259I"))
    {
        connectSituationType.setSuccessDisposition("SUCCESSFUL");
    } else
    {
        connectSituationType.setSuccessDisposition("UNSUCCESSFUL");
    }
    situation =
        EventItemsFactory.createISituation("ConnectSituation", connectSituationType);

} else if (msgID.equalsIgnoreCase("ITSO434W"))
{
    // Report Situation
    IReportSituation reportSituationType =
        SimpleEventFactoryImpl.getInstance().createReportSituation();
    reportSituationType.setReasoningScope("INTERNAL");
    reportSituationType.setReportCategory("STATUS");

    situation =
        EventItemsFactory.createISituation("ReportSituation", reportSituationType);
} else
{
    situation = null;
    System.out.println("A major error happened please check your system log");
}
return situation;
}
8.5 Adding sockets to communicate events to the autonomic manager

In the previous scenarios, the Common Base Events received by the manager touchpoint are stored in a file called CBEOut.log. The resource model then reads the events from this file. We can use other ways to communicate the received Common Base Events to the AME.

In this scenario, we modify the SampleAutonomicManagerTouchpointSupport to forward the incoming Common Base Events received through the sendEvent() API to a socket instead of writing to CBEOut.log (Figure 8-4 on page 244). We also need to modify the resource model to read from that socket instead of reading from CBEOut.log. Example 8-11 on page 244 describes the modification needed in the manager touchpoint support to write to a socket instead of CBEOut.log.
Figure 8-4 Using sockets between the manager touchpoint and the manager

Example 8-11 SampleAutonomicManagerTouchpointSupport_Support

package com.ibm.itso.autonomic.comm;

import com.ibm.autonomic.manager.*;
import com.ibm.autonomic.resource.*;
import org.eclipse.hyades.logging.events.ICommonBaseEvent;
import java.rmi.RemoteException;
import java.util.Iterator;
import java.io.*;
import java.net.*;
public class SampleAutonomicManagerTouchpointSupport_Socket extends AutonomicManagerTouchpointSupport {
    private String cbeFileName = "CBEOut.log";
    private FileWriter cbeFile = null;
    IManagedResourceTouchpoint resolvedTP = null;
    DatagramSocket socket;
    InetAddress address;
    byte[] buf;
    String cbeString;

    public SampleAutonomicManagerTouchpointSupport_Socket() throws RemoteException {
        super();
    }

    public void start() {
        // Similar to the SampleAutonomiManagerTouchpointSupport.start() No change between them
        ....
    }

    // This is manager's sendEvent method. Receives events from the outputer.
    public void sendEvent(ICommonBaseEvent incbe) throws RemoteException {
        try {
            socket = new DatagramSocket();
            address = InetAddress.getByName("localhost");
            cbeString = incbe.toString().trim();
            buf = new byte[cbeString.length()];
            buf = cbeString.getBytes();
            DatagramPacket packet = new DatagramPacket(buf, cbeString.length(), address, 4445);
            socket.send(packet);
            System.out.println("Sent a CBE of length: "+buf.length+" bytes.\n");
            // Clear the buffer for sending next event
            buf = null;
            try {
                Thread.sleep(100);
            } catch (Exception e) {
                socket.close(e);
                e.printStackTrace();
            }
            socket.close();
        } catch (IOException ioe) {
            socket.close();
            // The manager is responsible for receiving this information
        }
    }
}
public static void main (String[] args) throws RemoteException
{
    StopMGRTP_SocketThread st = new StopMGRTP_SocketThread();
    st.start();

    // create an instance and start it up
    SampleAutonomicManagerTouchpointSupport_Socket amts = new SampleAutonomicManagerTouchpointSupport_Socket();
    amts.start();
}

// This thread is incorporated to gracefully shutdown the Manager Touch point.
class StopMGRTP_SocketThread extends Thread
{
    String stopfile = "stop_mgrtp";

    // This method is called when the thread runs
    public void run()
    {
        while (true)
        {
            boolean exists = (new File(stopfile)).exists();
            try
            {
                sleep(2000);
            } catch (InterruptedException e) {
                System.out.println(e.toString());
            }
            if (exists)
            {
                (new File(stopfile)).delete();
                System.exit(0);
            }
        }
    }
}

For the modifications needed in the resource model, please refer to 6.4.3, "Creating the ITSO_CBESocket_Monitor resource model" on page 152.
8.6 Using a database to store received events

Though we do not show a complete example here, this section is intended to show that many methods could be used to pass events from the ManagerTouchpointSupport object to the AME. In this example, we describe how one might use a database to store the incoming events. These stored events can be called and analyzed by the autonomic manager. Figure 8-5 illustrates how this scenario can be viewed. Similarly, other ways of communication can be used to communicate the received events by the autonomic manager touchpoint support to the autonomic manager.

![Diagram](image.png)

*Figure 8-5 Using a database for events to be used by the manager*
We will not discuss this example in detail here. We leave this as an exercise for the reader. The scenario can be achieved by modifying the Manager Touchpoint Support to write the received Common Base Events by the sendEvent() to the database and modifying the resource model and AME to read the events from the database.

8.7 Summary

This chapter has described scenarios based on capabilities described throughout the rest of the redbook. It shows an example of what might be a legacy application that writes events into a log file. The examples describe how to use the Generic Log Adapter and Log and Trace Analyzer to achieve a maturity level 2 autonomic system and how to achieve a maturity level 4 autonomic system by replacing the Log and Trace Analyzer with AME. It also describes how to modify an application to generate Common Base Events directly instead of writing to a log file. In addition, we have described different examples for how to communicate the incoming events received by the manager touchpoint support to the AME, either by using an intermediary file, using sockets, or using a database.
Appendixes
Regular expressions

Regular expressions are often used in both the Generic Log Adapter and in resource models. This appendix provides a brief overview of regular expressions for those readers unfamiliar with them and their syntax.
What is a regular expression?

Regular expressions, commonly called regex, are a set of rules to match text patterns. They are used mainly for searching text or search/replace mechanisms.

How they work

Regular expressions use matching characters to identify the rules for the text search. For example, ‘^’ matches the beginning of the line and ‘.’ matches any character. So the search pattern ^T.n will match the lines that starts with ‘T’ followed by any character and followed by ‘n’. The result of this search could return the following lines:

Tune your car.
Tone it down.
Ten computers can be used in this setup.

Regular expressions: most commonly used matching characters

Table A-1 lists the most commonly used matching characters of regular expressions.

<table>
<thead>
<tr>
<th>Matching Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the beginning of the string.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of the string.</td>
</tr>
<tr>
<td>.</td>
<td>Matches any character.</td>
</tr>
<tr>
<td>*</td>
<td>Matches 0 or more times.</td>
</tr>
<tr>
<td>?</td>
<td>Matches 1 or 0 times.</td>
</tr>
<tr>
<td>+</td>
<td>Matches 1 or more times.</td>
</tr>
<tr>
<td>[ ....]</td>
<td>Matches anything inside this brackets.</td>
</tr>
<tr>
<td>[0-9]</td>
<td>Matches any digit character.</td>
</tr>
<tr>
<td>[a-z]</td>
<td>Matches any small letter character.</td>
</tr>
<tr>
<td>{A-Z}</td>
<td>Matches any capital letter character.</td>
</tr>
<tr>
<td>[0-9a-zA-Z]</td>
<td>Matches any alpha numeric character.</td>
</tr>
</tbody>
</table>
### Matching Character Description

<table>
<thead>
<tr>
<th>Matching Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[^....]</code></td>
<td>Matches any character except what is inside brackets.</td>
</tr>
<tr>
<td><code>( .... )</code></td>
<td>Match the expression in the parentheses as sub-patterns; they can be called later on by using $n$, where n is the location of this subpattern in the matching string.</td>
</tr>
<tr>
<td><code>{n}</code></td>
<td>Matches exactly n times.</td>
</tr>
<tr>
<td><code>{n,}</code></td>
<td>Matches at least n times.</td>
</tr>
<tr>
<td><code>{n,m}</code></td>
<td>Matches at least n times but not more than m times.</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Logical OR.</td>
</tr>
<tr>
<td><code>\d</code></td>
<td>Matches a digit character.</td>
</tr>
<tr>
<td><code>\D</code></td>
<td>Matches a non-digit characters.</td>
</tr>
<tr>
<td><code>\s</code></td>
<td>Matches a whitespace character (space, tab, linefeed, formfeed, and so on).</td>
</tr>
<tr>
<td><code>\S</code></td>
<td>Matches a non-whitespace characters.</td>
</tr>
<tr>
<td><code>\w</code></td>
<td>Matches a word character (alpha numeric plus <code>_</code>).</td>
</tr>
<tr>
<td><code>\W</code></td>
<td>Matches a non-word characters like <code>'</code>, ``, space, and so on.</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>Matches only at a word boundary.</td>
</tr>
<tr>
<td><code>\B</code></td>
<td>Matches only at a non-word boundary.</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Matches an ASCII tab character.</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>Matches an ASCII new line character.</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Matches an ASCII return character.</td>
</tr>
<tr>
<td><code>\f</code></td>
<td>Matches an ASCII form feed character.</td>
</tr>
<tr>
<td><code>\0nnn</code></td>
<td>Matches a given octal character.</td>
</tr>
<tr>
<td><code>\xhh</code></td>
<td>Matches a given 8-bit hexadecimal character.</td>
</tr>
<tr>
<td><code>\xhhhh</code></td>
<td>Matches a given 16-bit hexadecimal character.</td>
</tr>
</tbody>
</table>
For more information about regular expressions, there are many resources available on the Web. Search for these resources using key words like “regular expressions”, “regexp”, “awk”, “perl”, “apache”, and “jakarta”. The Toolkit uses the jakarta regular expression tools provided by Apache. For more documentation about this package, go to:

http://jakarta.apache.org/regexp/apidocs

Regular expression examples

The following are a few examples that might be used in configuring a Generic Log Adapter.

Example 1: Substitute everything
Input message:

[Fri Jan 16 18:15:16 IST 2004] ITS0259I Successful connection

Matching string:

^(.*)

Substitute string:

ITSOSample

Result of the substitution:

ITSOSample

Example 2: Using boolean OR
Input messages:

[Fri Jan 16 18:15:16 IST 2004] ITS0259I Successful connection
[Fri Jan 16 18:16:22 IST 2004] ITS0945E Cannot connect to data source

Matching string:

.*ITS0259I.*|.*ITS0945E.*|.*ITS0314C.*

Substitute string:

INTERNAL

Result of the substitution:

INTERNAL
Example 3: Rearrange the time
Input message:
[Fri Jan 16 18:15:16 IST 2004] ITSO259I Successful connection

Matching string:
^(.*) \w{3} \d{2} \d{2}:\d{2}:\d{2})(.*)(\d{4})(.*$

Substitute string:
$6, $2 $3, $4

Result of the substitution:
2004, Jan 16, 18:15:16

Example 4: Reformulate the time into a standard form
We can rearrange the date provided and convert the month from three letters to numbers to put the date in a standard format that is acceptable by XML Schema. One way to do this is to provide 12 pairs of matching strings and substitute strings, one for every month.

Input message:
[Fri Jan 16 18:15:16 IST 2004] ITSO259I Successful connection

Matching string:
(.*) Jan \d{2} \d{2}:\d{2}:\d{2})(.*)(\d{4})(.*$

Substitute string:
$6-01-$3T$4Z

Result of the substitution:
2004-01-16T18:15:16Z
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/SG246635

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, SG246635.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6635Examples.zip</td>
<td>Zipped source code and other files from the redbook.</td>
</tr>
</tbody>
</table>
6635README.txt  A 'read me' file containing a short description of the contents and installation instructions.

System requirements for downloading the Web material

The files in this download will consume less than 5 MB of space on the target system. However, to use these the IBM Autonomic Computing Toolkit should also be installed. Please see the system requirements for the Autonomic Computing Toolkit.

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 directly structure will be self-explanatory based on the description of the examples in this redbook.
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 on ordering these publications, see “How to get IBM Redbooks” on page 260. Note that some of the documents referenced here may be available in softcopy only.

- *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* (available with the IBM Autonomic Computing Toolkit)
- *Autonomic Management Engine Developer’s Guide* (available with the IBM Autonomic Computing Toolkit)
- *Generic Log Adapter User’s Guide* (provided with the Generic Log Adapter)
- *IBM Tivoli Monitoring Workbench User’s Guide Version 5.1.1*, SH19-4571

Online resources

These Web sites and URLs are also relevant as further information sources:

- GNU Operating System - Free Software Foundation
  
  [http://www.gnu.org](http://www.gnu.org)

- Hyades project from Eclipse
  
IBM Autonomic Computing blueprint

IBM developerWorks autonomic computing page
http://www.ibm.com/developerWorks/autonomic

IBM Publications Web site
http://publib.boulder.ibm.com/

Jakarta regular expression package information
http://jakarta.apache.org/regexp/apidocs

Java Activation Framework information

Java Mail API information
http://java.sun.com/products/javamail/

Microsoft Developer's Network resource for 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
Index

A
actions 36, 86, 88, 129, 203
Actions.bat (or .sh) 175
ActiveX 90
adapter configuration file 57
deploying 69
running 69
testing 67
Adapter Rule Editor 24, 27, 54, 57
AdaptorSensor 55
administrative console 18
advanced topics 202
affected resource 42
AME (see Autonomic Management Engine)
amtapi.jar 48
analyze 6, 8, 25
analyze log events 74
APIs 34, 46
architectural levels of maturity 14
architectural overview 17
assignManager() 48–49
asynchronous messages 7
attributes 5, 88
automation 16
autonomic manager 6
autonomic computing 3
attributes 5
blueprint 12, 20
concepts 6
control loop 6
principles 4
value 15
vision 4
Autonomic Computing Toolkit 23
technologies 25
tools 25
Autonomic Management Engine 24–25, 27, 183
introduction 184
autonomic manager 32
collaboration 9
definition 8
knowledge 11
levels of 8

B
baroc 181
binding techniques 8
blueprint for autonomic computing 12, 20
business policy 13
business workload management 21

C
call-out-request 34
CBEFileOutputter 57, 63, 227
CBEFormatter 56
CBEjavadoc.zip 46
CBELogOutputter 57
CBESStdoutOutputter 57
CIM (see common information model)
CIMOM 87, 185
responsibilities 185
classic probes 136
CLASSPATH 70, 113, 191
collaboration 9
comments 97
Common Base Event 25, 42, 71
APIs 46
definition 42
example 44
schema 43
structure 42
Common Base Events 32–33, 41, 90, 217
API 46
generating 45, 236
Common Information Model 8, 86, 118
classes 32, 86, 136
data source 130
implementation 93
methods 88, 205
common systems administration 17–18
CommonBaseEvent
   API 46
CommonBaseEventImpl 45
complex analysis 17, 20
composite resource domain 9
configuration file 36
configuration information 18
console 25
constructor 99, 155, 166
context
   Generic Log Adapter 54
   resource model 89
control loop 6, 8, 25–26, 32, 35, 49
control loop kernel 32
correlate 8
correlate log events 76
create() 106, 161
creating an ILT package 113
creationTime 43, 66
cure action launchers 213
custom actions 88
customer order application 9
CustomOutputter 229
cycles 27, 86–87

data source 118, 131
data source class 96
decision tree script 86, 89, 141
dependencies 86, 89, 139
deploying the adapter 69
description 95
destroy() 106, 161
developerWorks 23
Distributed Management Task Force 8

eclipse 27, 42, 45, 57
effectors 7, 10, 33–34
EIF event sink 201
embedding application 186
   building and launching 197
   stopping 196
enable a managed resource 31
enumerateInstances() 103, 159
equals(RMIIdentifier rmiID) 187
ERROR status 32
events 86, 88, 137, 203
example system 9
execute 6
ExtendedDataElement 42
extractor
   Generic Log Adapter 55

general settings 135
generating events 33
Generic Log Adapter 24, 26, 32, 34, 45–46, 54–56, 223
   configuration file 54
   context 54
   engine 70
   extractor 55
   formatter 56
   outputter 56
   outputter class 54
   parser 56
   perspective 58
   starting 70
get operations 7
getAttribute() 161
getKey() 187
getMultipleProperties() 102, 157
getProperty() 100, 148, 155
GLA
   rules and parsers 28
GLA (see Generic log Adapter)
gla.bat 70
guiding principles 4

heart monitor 20
heterogeneous workload management 17, 21
hgla.jar 70
hierarchical arrangements 9
hlcore.jar 70
hlcore.jar 70
holes 88
hosting environment 185
human body 4
Hyades 55
Hyades project 42, 45

I
IAutonomicManagerTouchpointSupport 48, 50
ICommonBaseEvent 45, 47
IComponentIdentification 47
ILT
  compiling 168
IManagedResourceTouchpoint 48–50
imports 98, 154, 209
information and documentation 24
Init() 90, 142, 177
Instrumentation Library Type 87, 93, 95
class 206
  compiling 112, 168
  implementation 99
  package 113
Integrated Solutions Console 24–25
integration 16
interaction styles 34
Interface Definition Language 94
introduction 1
invokeMethod() 106, 161
ITSO_AME.java 191
ITSO_CBELog_Monitor resource model 93
ITSO SIMPLE Appl 44
ITSO SubComponent 44

J
jakarta-oro-2.0.7.jar 70
Java
  parsers 28
Java Activation Framework 211
Java CIM classes 87
Java Mail APIs 208–209, 211
Java Management Extensions (JMX) 8
Java message catalog 36
Java resource model 133
JavaScript 89, 117, 184
jlog.properties 199

K
key parameter 96
knowledge 6, 11

L
local data store and reporting 20
local intelligent correlation 20
local persistence checking 19
Log and Trace Analyzer 24–25, 28, 72, 223
  analyze 74
  correlate 76
log events
  correlate 76
log file 26
logging 139
logging information 86, 89
LogWriter 218
LTA (see Log and trace Analyzer)

M
m12_mode.properties 199
M12Instrumentation 87, 95, 112
Main() function 89
managability interface 34
Managed Object Format 35, 94, 100
managed resource 6–7
touchpoint 7, 26
Managed Resource Touchpoint 32, 47
ManagedResourceTouchpoint class 48
Manager Touchpoint 47
ManagerResourceTouchpointSupport 147
marshalling techniques 8
Match field 65
matching characters 252
matrix management protocol 10
maturity index 14
maturity level 14, 72, 77
  adaptive 13
  autonomic 13
  basic 12
  managed 13
  predictive 13
maturity level 2
  sample application 223
maturity level 4
  sample application 228
message catalog 36
message severity 129
messaging bus 8
metrics 8
monitor 6, 8, 25
monitoring 19
self-configuring 3, 5, 18
self-healing 3, 5, 18
self-management 17
self-optimizing 3, 5, 18
self-protecting 3, 6, 18
sendEvent() 34, 45, 48, 50, 71, 147, 230, 237
sensor 55
   Generic Log Adapter 55
sensors 7, 10, 34
Service Object API 91, 184
serviceability 198
set commands 7
SetDefaultConfiguration() 90, 141
setDefaultConfiguration() 176
setPort() 166
setProperty() 105, 161
settings
   resource model 36
shell command 204
Shell Command Action Launcher 201
Simple Agent Reference Application 199, 201
SimpleExtractor 56
SimpleGLAPreject 59
SimpleOSFileSensor 55
situation 42
situationQualifier 43–44
SNMP 8, 33
sockets 147, 243
solution installation 17–18, 24
sourceComponentId 42
standards 16
START status 32
StaticParserSensor 55
STOP status 32
storeData() 166
successDisposition 43
supporting business needs 17
symptom database 77
   creating 79

toCBE() 237
tools 24–25
topologies 8
total cost of ownership 4
touchpoints 6–7, 41, 48
trigger conditions 86, 125

V
value of autonomic computing 15
vectors 99
vendor relationship application 9
virtualization 16
VisitTree() 90, 93, 131, 143, 178
Visual Basic 89, 117

W
Web services 8
wire formats 12
WMI 86, 118
WMI Query Language 87, 126
workforce productivity 17
WQL 126

X
xerces.jar 70
XML 25, 32, 42, 71
   definition 44
XML schema 44
xmlParserAPIs.jar 70

Z
ZipInputStream 194

T
   technologies 24–25
   thresholds 27, 35, 86, 88, 137
   time to value 4
Tivoli Business Systems Manager 129, 203
Tivoli Enterprise Console 129, 203
Tivoli Monitoring architecture 85
Tivoli Monitoring Services 98
A Practical Guide to the IBM Autonomic Computing Toolkit
This IBM Redbook provides practical information related to using the IBM Autonomic Computing Toolkit. Using this toolkit, software developers can provide facilities to enable applications or other system components to participate in an autonomic environment.

IBM provides the Autonomic Computing Toolkit to assist software developers in enabling components, products, and solutions to become autonomic. The toolkit also provides the base for developers of management software to be able to manage entities that have been properly enabled. As with the technologies themselves, this toolkit will evolve as new capabilities and tools become available.

This redbook provides a view of the Autonomic Computing Toolkit as it exists today and describes how it can be used to enable autonomic capabilities. This redbook primarily focuses on the problem determination capabilities provided by the Autonomic Computing Toolkit. Other facilities, such as Solution Installation and Integrated Console support, to name two, will be covered in future books.

For more information: ibm.com/redbooks