Java™ CAPS Basics

Implementing Common EAI Patterns

Michael Czapski ▪ Sebastian Krueger ▪ Brendan Marry
Saurabh Sahai ▪ Peter Vaneris ▪ Andrew Walker

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Sun Microsystems, Inc. has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications.

Sun, Sun Microsystems, the Sun logo, J2ME, J2EE, Java Card, and all Sun- and Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside the United States, please contact: International Sales, international@pearsoned.com.

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to informit.com/onlineedition
- Complete the brief registration form
- Enter the coupon code RSGP-E1MF-1USJ-UKFG-MUS6

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Java CAPS basics : implementing common EAI patterns / Michael Czapski ...
[et al.].
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-713071-9 (hardcover : alk. paper)
ISBN-10: 0-13-713071-6 (hardcover : alk. paper)
1. Java (Computer program language) 2. Enterprise application integration (Computer systems) I. Czapski, Michael.
QA76.73.J38J3633 2008
005.13'3—dc22 2008007526

Copyright © 2008 Sun Microsystems, Inc.
4150 Network Circle, Santa Clara, California 95054 U.S.A.
All rights reserved.

Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, Fax: (617) 671-3447.

ISBN-13: 978-0-13-713071-9
ISBN-10: 0-13-713071-6
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, April 2008

Preface

In their book *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* [EIP], Gregor Hohpe and Bobby Woolf elaborate on the subject of Enterprise Application Integration using messaging. They present, discuss, and illustrate over sixty EAI design patterns. These patterns, they believe, are key patterns most designers of EAI solutions will use when building enterprise integration solutions. Most examples in [EIP] use raw C# and raw Java to illustrate details of EAI patterns under discussion. Most of these patterns can be implemented succinctly, elegantly, and comprehensively using tools and technologies provided in the Sun Java Composite Application Platform Suite [Java CAPS].

This book is about implementing selected enterprise integration patterns, discussed in [EIP], using Java CAPS as the means to building practical enterprise integration solutions. It bridges the gap between the somewhat abstract pattern language and the practical implementation details. It is designed for integration architects, solution architects, and developers who wish to quickly implement enterprise solutions with Java CAPS. It discusses how enterprise integration patterns can be implemented quickly and efficiently by leveraging the Java CAPS tools and the authors' field experience.

While this book discusses Java CAPS implementation of [EIP] patterns, it does not discuss the patterns in depth. It is assumed that you are already familiar with the subject and need to apply the theoretical knowledge using Java CAPS.

This book is also about basics of the essential Java CAPS Suite components, based on the premise that you cannot apply patterns if you cannot effectively use the tools with which to do it. Since the complete Java CAPS offering has so many components, including ones that are not essential to integration, this book elaborates only on the basic integration tools: eGate, eInsight, eWays, and Java Message Service (JMS).

This book also provides information you may need to effectively use Java CAPS. A considerable amount of Java CAPS-related material, provided in the text, is not published anywhere else.

The accompanying CD-ROM provides over 60 detailed examples that illustrate concepts and patterns under discussion. Some examples are high level, illustrating specific points. Other examples follow a step-by-step approach.

Java CAPS projects discussed and developed as examples are available for import and perusal.

How This Book Is Organized

This book is divided into three sections. Section I, "Preliminaries," contains chapters that discuss integration and background Java CAPS topics, including enterprise integration styles, Java CAPS architecture, and project structure and deployment.

Section II, "Patterns Review and Application," covers most [EIP] patterns with discussion of Java CAPS approaches to implementing them. This section includes chapters dealing with message exchange patterns, message correlation, messaging infrastructure, message routing, message construction, message transformation, messaging endpoints, and system management patterns and concepts. While discussing Java CAPS implementation of specific patterns, relevant Java CAPS concepts and methods are also discussed. When discussing implementations of the Message Sequence pattern, for example, Java CAPS concepts of JMS serial mode concurrency, Sun SeeBeyond JMS Message Server FIFO modes, and serializing eInsight Business Processes via JMS and XA are also discussed.

Section III, "Specialized Java CAPS Topics," discusses non-pattern matters of importance like solution partitioning, subprocess and Web Services implementation, management, reusability, scalability and resilience options, and others that are not covered elsewhere. This section also covers security features of Java CAPS.

The accompanying CD-ROM contains over 60 detailed examples implementing most of the patterns and concepts under discussion as well as two complete example solutions using many of the patterns discussed and illustrated in this book. The CD-ROM also contains a detailed practical walkthrough of generation and use of cryptographic objects such as X.509 Certificates, PKCS#12 and JKS Keystores, and related matters.

About the Examples

Conventions

Java CAPS Enterprise Designer (eDesigner) is a NetBeans-based Integrated Development Environment (IDE), which developers use to design and build Java CAPS

integration solutions. The vast majority of tasks can be accomplished in eDesigner by means of manipulating components represented by graphical objects, connecting graphical objects with lines, filling information in dialog boxes and property sheets, and choosing components in drop-down menus. The intention was to make development of integration solutions easy for business analysts and similar persons whose coding skills might not be up to the task in nongraphical environments. Since development of Java CAPS solutions results in production of J2EE Enterprise Applications, this graphical orientation might come as a bit of a surprise to hardcore J2EE developers used to writing raw Java and the fine-grained control they exercise through deployment descriptors and other J2EE artifacts. Be that as it may, you will find most Java Collaboration examples shown using Java source code rather than its graphical equivalent. This is principally to make the samples concise, clearly showing essential parts of each solution, and to minimize wasting space on graphics that add no particular value to the discussion. Every one of the Java Collaborations could have been shown in "Standard mode," but that would have required numerous pictures, pretty much one for each Java statement, to illustrate what just a few lines of Java code can show in just a few lines. Figure P-1 shows an example of a Java Collaboration in Standard mode.

Evident are several lines of pseudocode in the Business Rules pane and just one mapping in the Business Rules Designer pane.

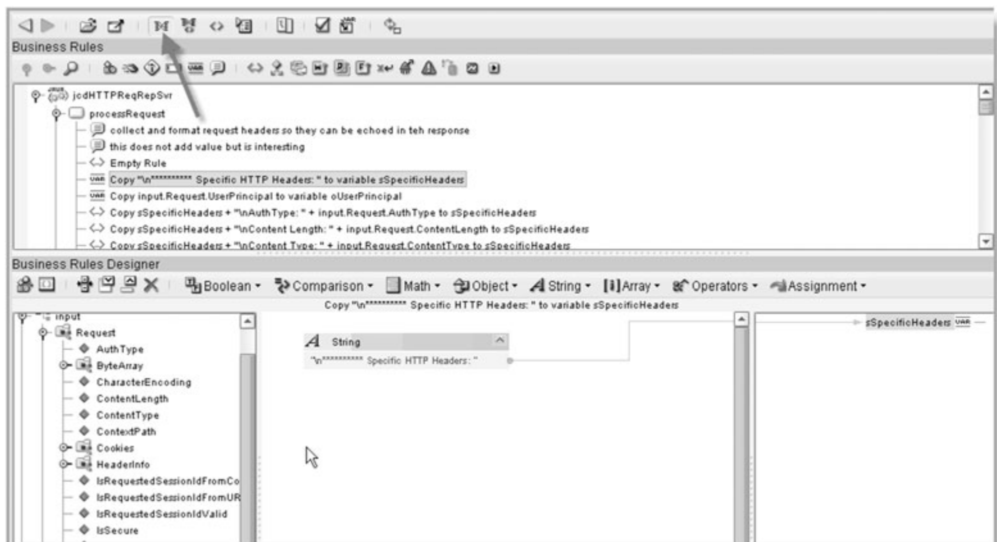

FIGURE P-1: Java Collaboration in Standard mode

In contrast, the same Java Collaboration in Source Code mode is much more illuminating, as seen in Figure P-2, as it shows all there is to know about 30 or so lines of code all at once.

Since switching between Standard mode and Source Code mode is a button-click away, we chose to use Source Code mode for Java Collaboration examples.

eInsight Business Processes are much easier to understand when presented in the graphical view, appearing similar to what is shown in Figure P-3. Object icons, taken from the Business Process Modeling Notation, are quite pleasant to look at, in this author's opinion.

In contrast, working directly with BPEL4WS XML source, an example of which is shown in Figure P-4, which is possible, is in this author's opinion bordering on cruel and unusual punishment, just as working directly with any other XML-based procedural language would be.

So, Java Collaborations are mostly shown in the Source Code mode, and eInsight Business Processes are mostly shown with the Business Rules Designer.

How you work with the tool is still up to you.

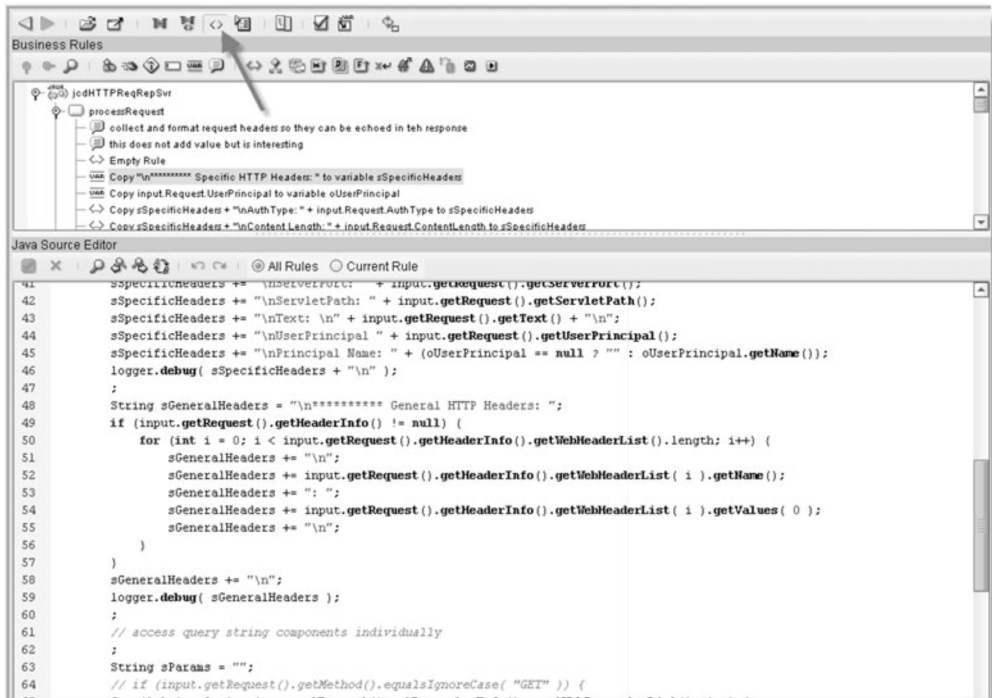

FIGURE P-2: Java Collaboration in Source Code mode

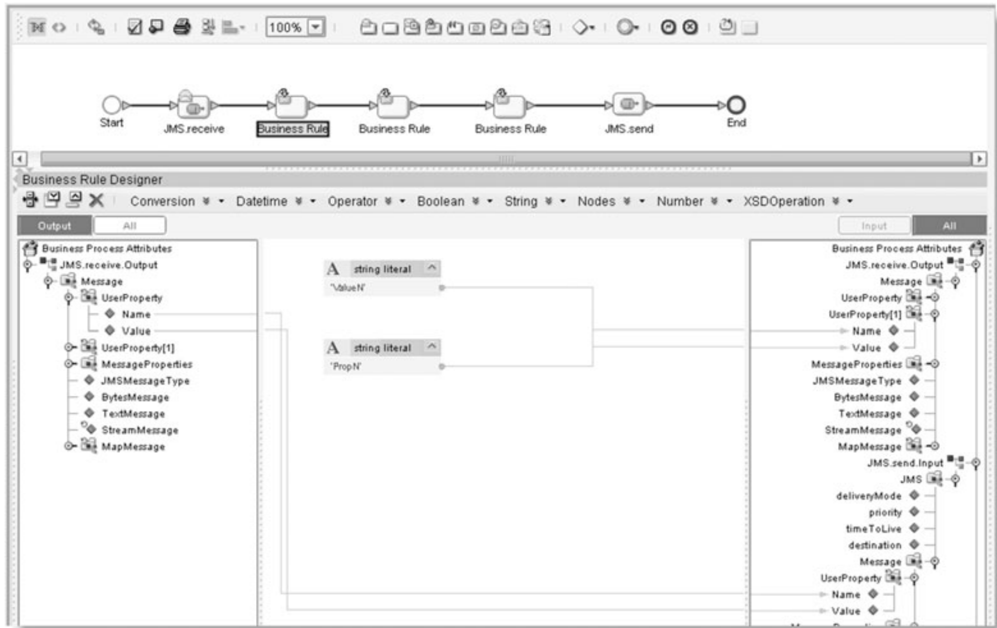

FIGURE P-3: eInsight Business Processes in the graphical view

Start — JMS.receive — Business Rule — Business Rule — Business Rule — JMS.send — End

```
84              </sbynbpelex:forEach>
85          </assign>
86          <assign name="Business Rule"
87                  sbynpxp:XLoc="337.0"
88                  sbynpxp:YLoc="43.0">
89              <copy>
90                  <from expression="&apos;Prop01&apos;"/>
91                  <to container="JMS.send.Input"
92                      part="JMS"
93                      query="/JMS/Message/UserProperty[ ( count(getContainerData(&apos;JMS.receive.Output&apos;, &apos;Message&apos;.
94              </copy>
95              <copy>
96                  <from expression="&apos;Value01&apos;"/>
97                  <to container="JMS.send.Input"
98                      part="JMS"
99                      query="/JMS/Message/UserProperty[ ( count(getContainerData(&apos;JMS.receive.Output&apos;, &apos;Message&apos;.
100             </copy>
101         </assign>
102         <assign name="Business Rule"
103                 sbynpxp:XLoc="450.0"
104                 sbynpxp:YLoc="43.0">
105             <copy>
```

FIGURE P-4: BPEL4WS XML source

LIST OF ILLUSTRATIONS AND EXAMPLES

To include both discussion and all relevant examples in one book would have made it over 1,000 pages in length—too large for printing. As a consequence, this book discusses Java CAPS facilities, focusing on their application to implementation of enterprise integration patterns with high-level illustrations. References to detailed examples are provided in PDF format on the accompanying CD-ROM. The PDF on the CD-ROM provides both detailed illustrations for most of the patterns as well as two completely worked-through Java CAPS–based case study solutions that implement a number of the patterns discussed in this book.

The CD-ROM also contains a chapter dealing with cryptographic objects used to configure security-related aspects of the suite.

Illustrations and examples included on the CD-ROM are listed in Table P-1, with section headers in the order of appearance. Of the over 60 examples, most are developed in a step-by-step manner, deployed, and exercised. In most cases, results of execution are shown and discussed.

TABLE P-1: Illustrations and Examples in Part II

Chapter/Section	*Example Topic*
Hello Java CAPS World	Introductory step-by-step example. Basic file transfer projects with separate implementations using eGate and eInsight.
Event Message Using Scheduler	Event Message pattern implementation. Basic scheduled solution using Event Message triggered by a Scheduler eWay.
External Scheduler Example	Event Message pattern implementation. Scheduled solution using external scheduler and an external TCP Sender client injecting an Event Message through a TCP Server eWay.
JMS Request/Reply Invoker for eInsight	Request/Reply pattern implementation. New Web Service Java Collaboration for invoking JMS Request/Reply functionality from an eInsight Business Process.

Table P-1: Illustrations and Examples in Part II *(continued)*

Chapter/Section	*Example Topic*
JMS Request/ Response Auction Pattern	Request/Reply pattern implementation. Java Collaboration and JMS Request/Reply–based implementation of an Auction pattern where the fastest responder wins.
HTTP Request/ Response	Request/Reply pattern implementation. A series of HTTP requestor and HTTP responder implementations using both HTML and XML payloads, prefaced by a recap of HTTP principles and mechanics.
SOAP Request/ Response	Request/Reply pattern implementation. Specialization of a HTTP request/response implementation using explicitly constructed and parsed SOAP XML messages as requests and responses.
Web Service Request/ Reply	Request/Reply pattern implementation. Web Service request/response implementation as an example of a Request/Response pattern using both eInsight Business Processes and Java Collaborations.
JMS Serial Mode Concurrency	Message Sequence pattern. Using Sun SeeBeyond JMS Message Server facilities for implementation of message sequence preserving solutions.
Sun SeeBeyond JMS FIFO Modes	Using Sun SeeBeyond JMS Message Server facilities for implementation of message sequence preserving solutions. Series of examples demonstrates the impact of different Sun SeeBeyond JMS Message Server FIFO modes on message sequence.
Serializing Business Processes with XA	Message Sequence pattern. Examples illustrating the impact of imposing XA transactionality on eInsight Business Processes and how it affects message sequence preservation (new in v. 5.1)
Message Expiration	Java Collaboration and JMS-based example illustrating Message Expiration.

(continues)

Table P-1: Illustrations and Examples in Part II *(continued)*

Chapter/Section	***Example Topic***
Batch Local File Streaming	Data streaming examples using Batch Local File eWay with discussion of buffering and its impact on message throughput.
eTL Streaming	Very basic eTL example streaming data from a flat file to a database table and a functionally equivalent Java Collaboration example.
Temporary JMS Destinations	Anti-example illustrating the use of an explicitly created JMS temporary destination.
Static Selector	Example illustrating the use of static JMS selectors.
Dynamic Selector	Example illustrating the use of dynamic JMS selectors, constructed at runtime and used in a Java Collaboration to explicitly choose messages to receive.
Resilient JMS with JMS Grid	Example of a simple JMS Grid-based automatic JMS Client failover.
JMS Message Body Formats	Example collaboration that inspects JMS header properties to determine JMS message body format and branch.
Dead Letter Channel in 5.1.2	Example exercising JMS Redelivery Handling functionality with undeliverable message being delivered to a Dead Letter Queue.
eInsight XA Transactionality	Example inducing success and failure outcomes that demonstrate XA transactionality of an eInsight Business Process with side effects in non-XA-capable resources.
eInsight Persistence	Illustration of eInsight persistence, eInsight monitoring, and eInsight restart-recovery of in-flight process instances.
Resequencer: Basic Version	Simple resequencer with memory-based message buffer.
Resequencer: Persisted Version	More sophisticated resequencer with RDBMS-based message buffer and message sequence persistence.
Routing Slip	Routing slip-based message routing solution using JSM and Java Collaborations.
JMS User Properties Envelope Wrappers	Series of examples demonstrating Envelope Wrapper pattern implemented using JMS message header properties in Java Collaborations and eInsight Business Processes.

Table P-1: Illustrations and Examples in Part II *(continued)*

Chapter/Section	***Example Topic***
Content Enricher	Content Enricher implementation using eInsight and Oracle eWay to receive a Purchase Order, enrich it with pricing information for an Oracle database, and produce an Invoice.
Polling File System	Series of examples polling local file system using Scheduler eWay–driven and Batch Inbound–driven Batch Local File eWay to implement polling solutions.
Polling JMS Destination	Try-wait-retry FTP delivery solution implementing JMS Destination polling for retry scheduling.
Durable Subscriber	Example illustrating the concept of a JMS topic durable subscriber and behavioral differences between nondurable and durable subscribers.
Idempotent Receiver	Example illustrating message duplication detection–based Idempotent receiver solution.
Multi-Input Service Activator	Example using the OpenTravel Alliances XML Schema documents to implement a Service Activator solution that allows the business service to be activated through a file submission, JMS message submission, and Web Services invocation.
Monitoring eInsight-based Solutions	Example illustrating eInsight persistence, persistence for reporting, and runtime monitoring of eInsight Business Process instances.
Simple Alert Processor for a JMS Channel	Example implementing a simple solution that receives and processes Alert Agent alerts delivered through the JMS Alert Channel.
Catching "Uncatchable" Exceptions	Example using Alert Agent infrastructure to catch and process exceptions that occur outside the Java Collaborations and Business Processes and therefore cannot be caught and processed in JCDs or BPs. Catching and processing a database connectivity exception is the subject of this example.
Programmatic Management	Example Java Collaboration that uses the JMX instrumentation to programmatically start and stop a specified Java CAPS component, such as another Java Collaboration or a Business Process service.

(continues)

Table P-1: Illustrations and Examples in Part II *(continued)*

Chapter/Section	***Example Topic***
JMS Latency	Java Collaboration and eInsight Business Process examples illustrating calculation of JMS message delivery latency.
eInsight Correlation Processor: First Cut	Example of an incorrect, naïve implementation of eInsight correlation.
eInsight Correlation Processor: Second Cut	Example of correct implementation of a simple eInsight correlation with a simple correlation key.
Derived Correlation Identifiers	Example of a more complex eInsight correlation with a structured, message-derived correlation key based on a subprocess-based correlation key derivation solution.
Derived Correlation Identifiers: Alternative	Alternative example of a more complex eInsight correlation with a structured, message-derived correlation key based on Java Collaboration correlation key derivation preprocessor.
Message Relationship Patterns	Series of examples of using eInsight correlation facilities to implement Message Relationship patterns: Header-Items-Trailer Correlation, Any Order Two Items Correlation, Any Order Two Items Correlation with Timeout, Items-Trailer Correlation, Header Counted Items Correlation, Counted and Timed Items Correlation, Timed Items Correlation, Scatter-Gather Correlation.
Items-Trailer Correlation	Reimplementation of the Items-Trailer Correlation using a Java Collaboration, JMS, and dynamic JMS selectors—no eInsight.
Using New Web Service Collaborations	Example of using New Web Service Java Collaborations to implement reusable modules for use as activities in eInsight Business Processes.
Using eInsight Subprocesses for Reusability	Series of examples of using eInsight subprocesses as reusable components for use as activities in eInsight Business Processes: Request/Response, OneWay Operation, and Notification Subprocess implementations.
Using eInsight Web Services for Reusability	Series of examples of using Web Services as reusable components for use as activities in eInsight Business Processes: Request/Response, OneWay Operation, and Notification Web Service implementations.
JMS-Triggered Java Collaborations	Example of exception and JMS redelivery handling in a JMS message–triggered Java Collaboration.

Table P-1: Illustrations and Examples in Part II *(continued)*

Chapter/Section	***Example Topic***
Other Java Collaborations	Example of exception processing in a non-JMS message–triggered Java Collaboration.
JMS-Triggered Business Processes	Example of exception handling in a JMS message–triggered eInsight Business Process demonstrating behavior differences between XA and non-XA processes.
Fault Handlers	Example of using Fault Handlers in eInsight Business Processes.
Secure Sockets Layer (SSL, TLS)	Series of examples illustrating the use of SSL in Java CAPS solutions, both HTTP eWay- and Web Services-based. Covers server-side and mutual authentication for both server and client endpoints.
Web Service, Stored Procedures, and XA	Complete case study implementing an Employee Database Maintenance process with a multi-database update, Web Services, Oracle Stored Procedures, and an XA Business Process. This example implements and exercises a large number of patterns and suite features.
Example Travel Reservation	Complete Travel Reservation case study using Web Services orchestration and eInsight exception handling and compensation. This example implements and exercises a large number of patterns and suite features.
Handling Repeating Nodes in BPEL	Example of handling repeating nodes from XSD-based OTD in eInsight.
XML Deep Parse vs. Shallow Parse	Example implementing lazy XMLparse.
Using Multi-Operation WSDL	Example of implementing a multi-operation Web Service using WSDL and eInsight.
Cryptographic Objects	Step-by-step discussion of cryptographic objects required to configure PKI-related aspects of Java CAPS, with tools, commands, and scripts necessary to create certificate signing requests, convert between various certificate formats, and create various keystore types.

CHAPTER SIX

Message Routing

6.1 Introduction

This chapter discusses message routing patterns. It includes discussion and application of patterns from [EIP] Messaging Systems and Message Routing. The chapter briefly discusses where a Java CAPS solution developer can make routing decisions and discusses each of the routing patterns in turn, specifically Splitter, Aggregator, Resequencer, Scatter-Gather, Routing Slip, Process Manager, and Message Broker.

6.2 Overview

A messaging-based integration solution, whether or not and however it transforms messages as they pass through, inevitably routes messages from one or more sources to one or more destinations. A Java CAPS solution can make message routing decisions in four areas: the JMS Message Server, the connectivity map, the Java Collaboration definition, and the eInsight Business Process. Typical solutions that use just the eGate infrastructure would perform routing through the JMS Message Server, the connectivity map, and possibly the Java Collaborations. Typical solutions that use eInsight Business Process Management (BPM) would perform routing predominantly within eInsight Business Processes but may also route in the connectivity map. In all but the simplest solutions, routing will likely be performed by multiple components.

Routing in the JMS Message Server is performed as a consequence of configuring nondefault redelivery handling, which can divert messages to Dead Letter Queues. This issue was discussed in Chapter 5, "Messaging Infrastructure," section 5.13.

The connectivity map, the graphical representation of how Java CAPS components are connected, is the means to both collect all integration solution components that will be deployed as part of a single enterprise application and to configure certain aspects of the message endpoints that are logical in nature, such as JMS Destination names and properties, or names and name patterns for file system objects. The simplest functional Java CAPS solution must have a minimum of two components: a message source and a service that operates on messages from that source. Unlikely as it may seem, in special circumstance, such an apparently useless solution might be valid and reasonable. What [EIP] calls the Channel Purger would be an example of a solution that receives messages from an endpoint and routes them to nowhere. Figure 6-1 shows a connectivity map for a basic Channel Purger.

This is the simplest example of message routing: Fixed Routing [EIP].

Note

A Java CAPS implementer would typically look at the connectivity map for routing information—which components publish and subscribe to which JMS Destinations and how many, and which JMS Destinations are subscribed to/published to by an eInsight Business Process. For that reason, a solution that makes explicit routing decisions in Java Collaboration Definitions (JCDs) or Business Processes will be more difficult to analyze by an implementer new to it. It will also make it harder for the original developers to recall where and how routing decisions are made. If no other considerations dictate specific choices, given a choice of explicit routing in a JCD and explicit routing in an eInsight Business Process, choose the latter, as its graphical depiction of processing logic makes it more obvious that explicit routing takes place. Multiple subscriptions and/or publications by a service on a connectivity map are a strong hint that explicit routing is taking place inside a service component.

Message Router [EIP], a specialized Filter [EIP], represents a component in an integration solution that causes messages to be passed from a source to a destination depending on a possibly empty set of criteria. Unlike connectivity map–

FIGURE 6-1: Channel Purger

based fixed routing, Message Router variants that make explicit routing decisions programmatically can all be implemented in a Java CAPS solution using either JCDs or eInsight Business Processes or both.

The following sections discuss implementation of most of the router patterns using Java CAPS as the infrastructure.

6.3 Fixed Router

A fixed router, one where a single channel is a source of messages and a single channel is a destination, is the most trivial form of a Message Router. You would typically configure the connectivity map source and destination to configure a fixed router. If necessary, however, a Java Collaboration or a Business Process can be constructed to explicitly choose a destination if that destination is a JMS Destination.

Given the connectivity map shown in Figure 6-2, we would expect that the Java Collaboration publishes messages to the JMS Destination (queue) qDummy-Destination.

Inspection of the collaboration source, shown in Figure 6-3, reveals that it is the JMS Destination (queue) qNewQueue that is the actual destination of messages. This destination is hardcoded in the fixed router.

The same effect could be achieved by explicit assignment of the destination queue name prior to sending the message, as shown in Figure 6-4.

Given the connectivity map shown in Figure 6-5, we would again expect the queue qDummyDestination to be the destination of messages.

Inspecting the business rules embedded in the eInsight Business Process, shown in Figure 6-6, reveals this to not be the case.

In this example, an explicit assignment of a JMS Destination name to the destination node of the JMS OTD results in messages being explicitly routed to a JMS Destination (queue) qNewJMSDestination.

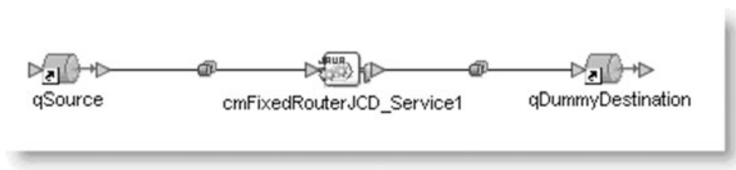

Figure 6-2: Connectivity map of an implicit fixed router

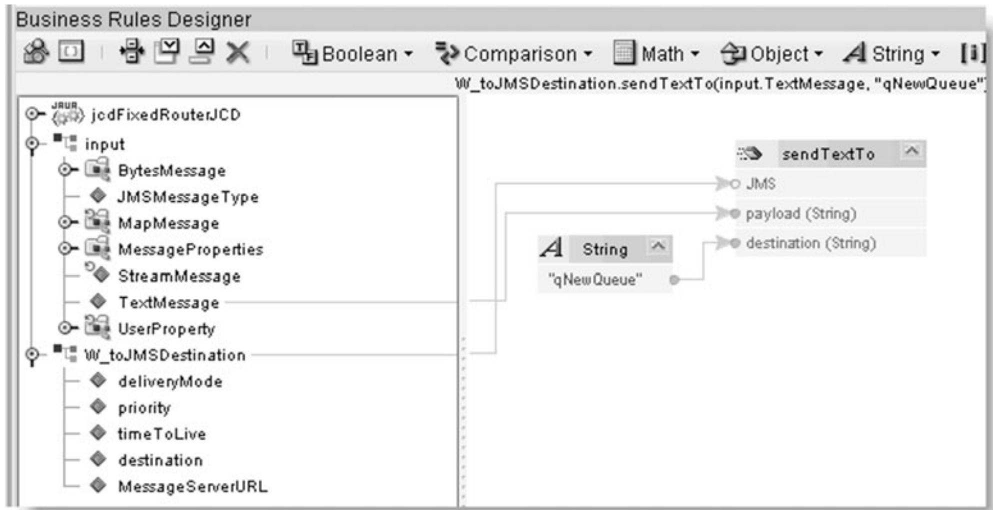

FIGURE 6-3: Hardcoded JMS queue name in a fixed router, which uses a sendTo() OTD method

FIGURE 6-4: Hardcoded JMS queue name in a fixed router using a "destination" OTD node

FIGURE 6-5: Implicit fixed router connectivity map

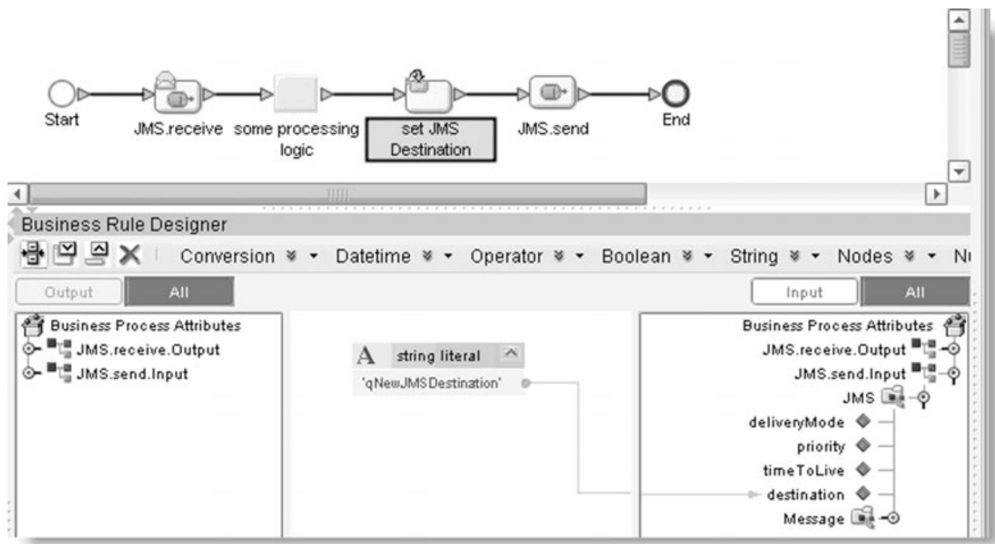

FIGURE 6-6: Explicit JMS queue assignment in a Business Process

Note

In all of the previous examples, the JMS Destination name contained the literal Dummy. This is a hint to the developer who inspects the connectivity map that the actual destination is likely different and is configured within the component that publishes to the "dummy" destination. This is a good practice suggestion, since no part of Java CAPS enforces naming conventions.

6.4 CONTENT-BASED ROUTER

Content of the message may dictate the destination to which the message must be delivered. Content-based Router [EIP] inspects the message it receives and sends it to a destination depending on the content.

In a simple case, a Java Collaboration or a Business Process would have a set of destinations hardcoded within a switch or an if-then-else construct that operates on all or part of the message. An example in Figure 6-7 is a simple Java Collaboration that illustrates dynamic JMS Destination selection.

```
public void receive( com.stc.connectors.jms.Message input, com.stc.connectors.jms.JMS W_toJMSDestination )
    throws Throwable
{
    if (input.getTextMessage().toUpperCase().endsWith( "PRIMARY" )) {
        W_toJMSDestination.setDestination( "qToPrimary" );
    } else if (input.getTextMessage().toUpperCase().endsWith( "SECONDAR" )) {
        W_toJMSDestination.setDestination( "qToSecondary" );
    } else {
        W_toJMSDestination.setDestination( "qToCatchOther" );
    }
    W_toJMSDestination.sendText( input.getTextMessage() );
}
```

FIGURE 6-7: Hardcode dynamic router

A Business Process that implements a dynamic router can be constructed similarly, as the example in Figure 6-8 shows.

Here a decision gate inspects a message to determine which branch to follow. A Business Rules activity assigns a string literal to the JMS Destination's destination attribute, and the JMS.send activity gets the message delivered to the destination so set.

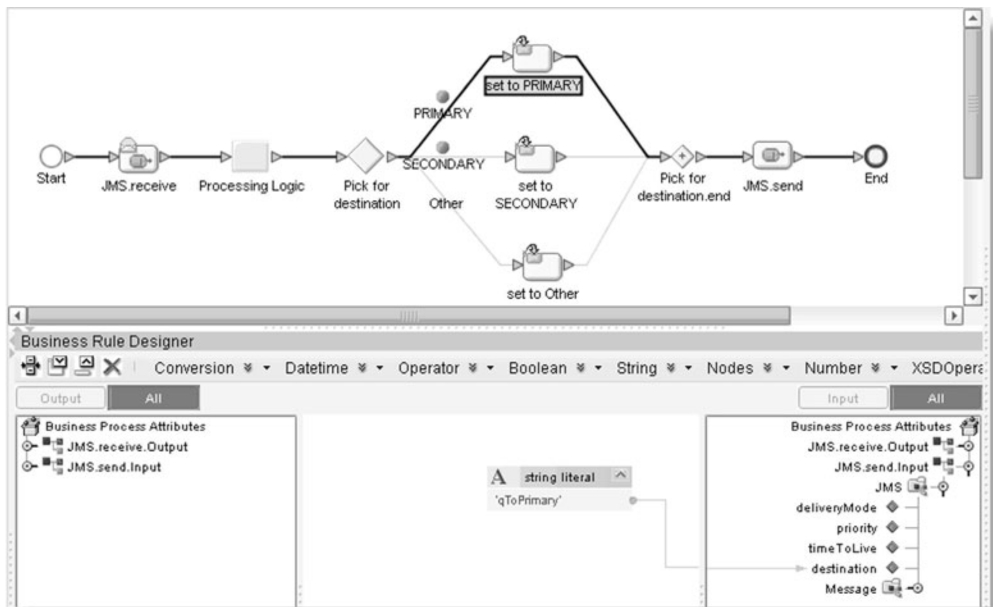

FIGURE 6-8: eInsight Business Process–based dynamic router

Whether a JCD or a Business Process is used, the connectivity map for the example will be identical to that used for the fixed router in the example in the previous section (i.e., the name of the JMS Destination will be unrelated to the actual JMS Destination to which messages will be delivered).

In the two examples shown in Figures 6-7 and 6-8, a conditional was evaluated to determine the destination of the message, which was hardcoded. If additional destinations were required, the collaboration or the process would have to be modified, and the application containing it would have to be redeployed to propagate changes to the runtime environment. This implementation of a Content-based Router is potentially a high-maintenance implementation if destinations change frequently.

In a special case, you could use the message, or the message component, as the complete name or a part of the name of the destination. You would not need to use a conditional or hardcode destination names.

The JCD in Figure 6-9 appends the first 10 characters of the input message to a literal qDest to form the name of the destination. The message is then written to the destination with the resulting name.

If the first 10 characters of messages were PRIMARY??? and SECONDARY?, where ? represents a space character, the resulting destination names would be qDestPRIMARY and qDestSECONDARY respectively. If using the Sun SeeBeyond JMS implementation, which does not require you to preconfigure JMS

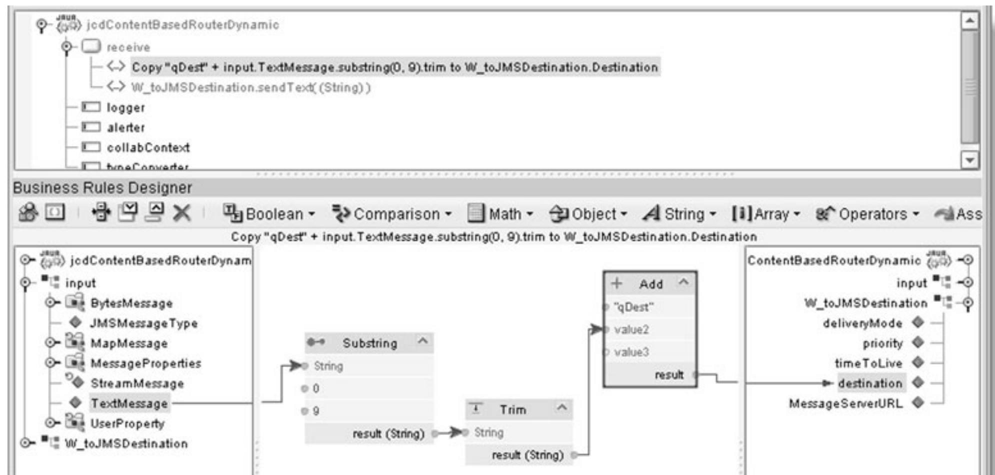

FIGURE 6-9: Dynamically generating JMS Destination names

Destinations ahead of time but rather creates JMS Destinations on first reference if they don't already exist, this could result in a completely dynamic Content-based Router. You could introduce a message whose initial characters were something other than PRIMARY??? and SECONDARY?, and the Sun SeeBeyond JMS implementation would create a new JMS Destination with that appropriate name and deliver the message there. There may not be a receiver for the message delivered to the new destination, but the Content-based Router itself would be dynamic and would not require modification. Addition of a destination would not require redeployment of the application containing such a router if no other changes were needed to take advantage of the new route. This solution does not require maintenance of the router if the number of destinations changes but makes it difficult to determine to how many destinations messages are routed, as it removes the setting of the content, upon which routing decisions are made, from the router to some other component upstream from the router or even outside the integration solution altogether.

In the previous examples, a very simple text message was used and some leading or trailing characters were extracted from the text for use in the conditional or as a part of a destination name. Messaging systems will rarely deal with such unstructured text messages. Much more likely, messages will be structured. The contents of one or more fields in the message will then be used for routing decisions or destination name derivation. For simplicity, we will continue using simple text messages wherever message structure has no bearing on the discussion.

These trivial examples demonstrate how explicit routing can be performed programmatically within a JCD or a Business Process. This method will be used to set destinations for more complex Message Routers.

6.5 Message Filter

Message Filter [EIP] is a component in an integration solution that selectively processes messages. A Java CAPS solution offers two ways in which a Message Filter can determine whether or not to process a message.

If the message source is a JMS Destination, such as a queue or a topic, the Message Filter can be configured, through the connectivity map, to only accept messages whose attributes match an SQL-like selection expression. This method leverages the JMS selector mechanism. Rather than receiving a message and, if not of interest, discarding it, the JMS selector–based Message Filter prevents

delivery of messages that do not match the selector expression to the filtering receiver. This mechanism is static in that the selection expression is configured through the connectivity map and cannot be changed without redeploying the enterprise application.

Java CAPS provides the means to implement a dynamic selection solution using a Java Collaboration. This technique, discussed at length in Chapter 5, section 5.6.7, and Chapter 11, "Message Correlation," section 11.11, allows selection expression to change at runtime, thus providing the means to implement dynamic routing solutions.

6.6 Recipient List

By Saurabh Sahai

Often, it is required that a message be selectively sent to more than one recipient. The recipients that are to receive each message are determined either dynamically, based on the message content, or statically, based on external business rules. For example, an expense approval request message, pertaining to expenses below a certain amount, may get sent to the immediate manager for approval, whereas a message above the defined limit must also be sent to the business unit head for special approval.

A recipient list processor is similar to a Content-based Router; however, unlike a Content-based Router that routes the message to a specific destination based on message content, a recipient list processor sends the message to one or more designated recipients. The list of recipients can be static, hardcoded within the implementation, or dynamic, provided to the implementation from an externally maintained source. The latter approach provides greater flexibility as it allows the recipient list to be dynamically configured.

In Java CAPS, a recipient list can be implemented using either a JCD or a Business Process. In either case, once the message is received, the list of intended recipients is computed from the available recipients, and the message is forwarded as required.

The Java Collaboration shown in Figure 6-10 receives an expense report message for an amount greater than $300. Based on business rules, the collaboration looks up additional approvers that are required to approve this expense report and sends a copy of the message to these approvers in addition to the default

approver. This is an example of a dynamic recipient list, where the collaboration uses information stored in an external store such as the organization's Lightweight Directory Access Protocol (LDAP) server to create the required recipient list. Exception processing has been omitted in Figure 6-10 to focus on the essentials of the example.

In the example in Figure 6-10, the additional approval threshold has been hardcoded within the Java Collaboration. In a more realistic example, externally configurable delegation of authority rules would be loaded into the collaboration using one of the techniques for dealing with dynamic runtime reconfiguration of components, discussed elsewhere in the book.

The example uses a hypothetical sendMail() method to send the expense report to a recipient for approval. The collaboration could have equally validly used multiple JMS Destinations, a single JMS Destination with target recipient indicated using JMS user properties, a Batch eWay, WebSphere MQ eWay, or any number of other endpoints, as dictated by the environment or business requirements.

```
private static final float DEF_APPROVER_EXPENSE_LIMIT = 300.0f;

public void receive( com.stc.connectors.jms.Message input, xsd.ExpenseReport516785849.ExpenseReport_ otdExpenseReport,
                     com.stc.connectors.jms.JMS otdJMS ) throws Throwable
{
    // unmarshal the incoming expense report into the ExpenseReport OTD
    String expenseReport = input.getTextMessage();
    otdExpenseReport.unmarshalFromString( expenseReport );
    // compute the expense total
    float expenseTotal = 0f;
    int numExpenseItems = otdExpenseReport.getExpenseDetails().countExpenseItem();
    for (int i = 0; i < numExpenseItems; i++) {
        expenseTotal += otdExpenseReport.getExpenseDetails().getExpenseItem( i ).getAmount();
    }
    // Determine the recipients for this expense report
    String employeeID = otdExpenseReport.getEmployeeID();
    if (expenseTotal <= DEF_APPROVER_EXPENSE_LIMIT) {
        // send the expense report to the default approver.
        // lookup the email id for the employee's default approver
        String defaultApproverEmailId = lookupApprovingManagerEmailId( employeeID );
        sendMail( defaultApproverEmailId, expenseReport );
    } else {
        // lookup email IDs for senior approvers required to approve expenses above the predefined limit
        String[] snrApproverEmailIDs = lookupSeniorApproveEmailIds( employeeID );
        for (int i = 0; i < snrApproverEmailIDs.length; i++) {
            // send expense report to each designated approver
            sendMail( snrApproverEmailIDs[i], expenseReport );
        }
    }
}
```

Figure 6-10: Recipient list example

6.7 Splitter

By Saurabh Sahai

An incoming message may encapsulate one or more submessages. It is often desired to process submessages independently as separate messages. For example, an order message may consist of multiple order line items, each of which corresponds to a unique item type and may be fulfilled by a separate inventory store.

A splitter solves the problem of processing a composite message comprising multiple submessages, each of which may be processed differently by breaking up the message into individual messages and sending each separate message for further processing by a downstream component.

A splitter can be implemented in Java CAPS in multiple ways. Java Collaborations can be used to receive the composite message, iterate over the individual submessages, and, on the basis of the message content, send each of them to a unique destination that is responsible for processing a specific type of message.

The collaboration shown in Figure 6-11 is an example of processing an incoming message consisting of multiple order items. The collaboration iterates over each order item and creates a new message, enriched with the original order item information, and sends it for processing by a specific system. The item number contained in each order item is used to determine the destination address where the enriched order item message is to be sent. Exception processing has been omitted in the example to focus on the essentials of the example.

Splitter is a component that, as the name suggests, breaks messages into component parts. How easy or difficult it is to split original messages and create component messages largely depends on the size and complexity of the message structures involved. As a general rule, it is easier to handle a composite message with more than one level of components using a Java Collaboration than to do so using an eInsight Business Process. Implementing nested loops in a Java Collaboration is easier and more compact than doing the same in Business Process Execution Language (BPEL) using the graphical environment. While loops, whether single-level or nested, are clearly visible in an eInsight Business Process graphic, making it obvious that splitting is taking place, it is necessary to reset the target OTD structure prior to its being populated in each iteration, which is neither obvious nor easily discovered by the casual observer.

```
/**
 * Receives an Order consisting of multiple order items each of which is to be processed by an
 * individual inventory system. The collaboration iterates over each of these individual
 * order items and sends an enriched message containing the order item and the original order id
 * to the appropriate inventory item for the fulfillment of the order item. For this example,
 * the queue for each inventory that will be used to fulfill the order item is determined using
 * the item number embedded in the order item message.
 */
public void receive( com.stc.connectors.jms.Message input, xsd.Order595920194.CompositeOrder otdCompositeOrder,
                     xsd.Order595920194.SubOrder otdSubOrder, com.stc.connectors.jms.JMS otdJMS )
    throws Throwable
{
    String inputOrder = input.getTextMessage();
    /*
      unmarshal the incoming composite message into the CompositeMessage OTD
     */
    otdCompositeOrder.unmarshalFromString( inputOrder );
    int numOrderItems = otdCompositeOrder.countOrderItem();
    String orderId = otdCompositeOrder.getOrderId();
    otdSubOrder.setOrderId( orderId );
    for (int i = 0; i < numOrderItems; i++) {
        String itemNumber = otdCompositeOrder.getOrderItem( i ).getItemNumber();
        int itemQty = otdCompositeOrder.getOrderItem( i ).getQuantity();
        /*
          Update the outgoing message with the itemNumber and quantity.
         */
        otdSubOrder.getOrderItem().setItemNumber( itemNumber );
        otdSubOrder.getOrderItem().setQuantity( itemQty );
        /*
          Send the individual order item to the inventory system.
          The inventory queue is identified by concatenating "q" with the item number.
         */
        String qInventorySystem = "q" + itemNumber;
        otdJMS.sendTextTo( otdSubOrder.marshalToString(), qInventorySystem );
    }
}
```

FIGURE 6-11: Dynamic content-based routing

6.8 AGGREGATOR

Aggregator [EIP] is a special Filter that collects related messages until some completeness condition has been reached, at which point it processes the related messages to obtain a single aggregated message that is then passed on to the next component.

An Aggregator must be able to correlate related messages, store related messages until ready to process, determine when the completeness condition is met, and implement the aggregation logic.

Java CAPS eInsight engine is a convenient tool to use for building Aggregators, as it inherently supports correlations and transparently stores related messages. The specific eInsight Business Process must only implement the completeness condition and the aggregation logic in order to become a specific Aggregator.

At the heart of every Aggregator is correlation logic, logic that determines which messages are related and therefore are subject to aggregation. Chapter 11 discusses at length the topic of message correlation with references to a number of specific examples presented in Part II (located on the accompanying CD-ROM). Chapter 11, section 11.10, discusses in detail a number of correlation implementations that incorporate an Aggregator.

Implementing an Aggregator without the benefit of eInsight is much harder. In addition to having to implement a completeness condition and aggregation logic, the solution designer must also do all the work related to storing and correlating messages. Chapter 11, section 11.11, discusses this topic and presents an example of how to accomplish the task of storing and correlating messages using just eGate and the Sun SeeBeyond JMS Message Server.

[EIP] discusses a number of completeness conditions an Aggregator might implement: Wait for All, Timeout, First Best, Timeout with Override, and External Event. Variants of these completeness conditions are discussed in Chapter 11, section 11.10. Implementing these conditions using eInsight with correlations is rather trivial. Implementing most of them using just eGate is much more difficult.

[EIP] also discusses a number of aggregation algorithms, including Select Best, Condense, and Collect for Later. Variants of these are also discussed in Chapter 11, section 11.10. Since aggregation will not start until all related messages are collected—that is, until the completeness condition has been satisfied—implementing aggregation logic is equally simple whether eInsight or eGate is used.

6.9 Resequencer

By Sebastian Krueger

Messages can arrive out of order for many reasons. If these messages are required to be delivered in sequence to a downstream component, the easiest solution would be to make sure that they never get out of order in the first place. Such approaches were discussed in Chapter 4, "Message Exchange Patterns," section 4.8. However, there may be times when we don't have a choice of how the upstream components are implemented; for example, we may control only the receiving side. Thus, the need to implement a component that will reorder messages may arise.

A number of implementations of a resequencer are possible. Chapter 4, section 4.2, "Resequencer," in Part II, discusses and illustrates two implementations

using examples: a simple buffered resequencer and a persisted resequencer, both of which are discussed in the remainder of this section.

The simple buffered resequencer operates as follows. When the resequencer receives a message, it adds that message to an internal buffer. It then sends all consecutive messages from the buffer.

In order to send all consecutive messages, the resequencer component needs to know the current sequence number index and whether a message with this index is in the buffer. If the index is not found in the buffer, then the message has not arrived yet. The resequencer will not send out buffered messages until at least the next message arrives.

A resequencer implementation requires all messages to have a unique sequence number. Not only do these sequence numbers have to be unique, they also have to be consecutive. That is, no gaps are allowed to exist in the sequence.

If a message gets lost and never arrives, messages would be queued up and would never be sent out because the resequencer component would be waiting for a message that will never arrive. To get around this issue, the designer could implement a solution whereby the resequencer only waits a set time for a message to arrive and then moves on to the next messages, effectively ignoring the message that never arrived. However, what if the message arrives late? What if a duplicate message arrives? Strategies for dealing with these conditions would have to be considered in designing a robust resequencer. The designer could, for example, discard the message, or send it to a Dead Letter Channel for alerting and auditing purposes.

When a simple resequencer starts, it expects the first message to have a sequence number of 0. However, what if this is not the case? For example, the resequencer might restart. Unless the sequence is persisted, the resequencer would expect the message sequence to start at 0 again. There are two ways to get around this problem. An initialization message could be sent that informs the resequencer which number is the start of the sequence. Alternatively, the sequence could be persisted so that it can be recovered in appropriate circumstances.

Another point that a simple resequencer does not handle is buffer overrun. If too many messages get queued up, the HashMap, used to store messages while assembling message sequences, may get too large to fit into the JVM allocated memory.

Implementation of a robust and scalable resequencer would require a significant amount of code and would likely be domain specific. An improved resequencer is discussed next. Implementation of a perfect resequencer is beyond the scope of this book.

An improvement to the previous resequencer would be to implement the buffer as a database table. By moving the message buffer to a persistent store, we solve two problems exhibited by the previous implementation. First, we effectively have an unlimited buffer, so no buffer overflow will occur. Second, in case of a server failure, buffered messages are not lost.

There are still unresolved issues with this persisted resequencer. The initialization of sequence numbers expects the first message to always start at 0. Also, we have not accounted for messages that never arrive. We briefly touched on the some of these issues in the previous section. They are out of the scope of this chapter.

While the two resequencers discussed in this chapter are by no means perfect, they do give examples of simple resequencers and give an indication of what is required to implement a robust resequencer.

6.10 Composed Message Processor

Composed Message Processor [EIP] is a higher-order component of a messaging system that accepts a message, breaks it up into submessages that are dispatched and processed by multiple lower-order components, then reassembles submessages into a final message.

In Java CAPS, as in any messaging system, implementation of a Composed Message Processor requires the use of correlations. Superficially, Composed Message Processor pattern is no different from the Scatter-Gather pattern [EIP]. Both involve breaking a message and reassembling the pieces once they are processed by independent intermediate components.

Chapter 11 discusses correlation implementation options provided by Java CAPS and presents a number of correlation examples. Section 11.10.5 provides a Java CAPS example of a Scatter-Gather pattern and Composed Message Processor pattern implementation.

6.11 Scatter-Gather

The Scatter-Gather [EIP] pattern involves breaking up a message, or replicating a message, delivering multiple messages to multiple components, then collecting related messages back together. Implementation of a Scatter-Gather pattern requires the use of correlations. It is discussed in various sections of Chapter 11.

6.12 Routing Slip

Routing Slip [EIP] is a mechanism that can be used to dynamically route a message through a series of components such that individual components do not embed routing logic. The route the message is to take can be computed by a router that embeds the necessary logic. This route is then attached to the message as a Routing Slip, and each component through which the message passes, once it performs its processing, forwards the message onto the next component specified in the Routing Slip. [EIP] discusses at length the rationale behind a desire to implement the Routing Slip pattern. Java CAPS, and its underlying JMS Message Server implementation, lends itself to building Routing Slip–based solutions; however, eInsight Business Processes may be better, in many circumstances, as an approach to conditional component invocation and dynamic route determination.

In a fixed Routing Slip solution, the message, once it leaves the router, is passed from component to component. There is no opportunity to change the message route once it is computed. An alternative to this approach is to have the router compute the next component to which to send the message, send the message to it, and have the component return the message back to the router once it is done. Routing decisions are still centralized, making routing logic simple to maintain, and the route the message takes can be changed by the router at any time based on the outcome of message processing.

In Java CAPS, a Routing Slip, or return destination, can be attached to the message in one of two ways. It can be passed via JMS user-defined properties if the message is passed from component to component over JMS, or the message can be packaged into an Envelope Wrapper and the Routing Slip can be incorporated into the Envelope metadata.

Envelope Wrapper is discussed at length in Chapter 8, “Message Transformation,” section 8.2. The route, computed by the router, could be represented in the Envelope node as a series of labels delimited by some delimiter, or an ordered, repeating collection of labels.

Chapter 4, section 4.3, “Routing Slip,” in Part II, illustrates this discussion with an example implementation of a Routing Slip pattern using Java Collaborations and JMS.

We could have used an eInsight Business Process to implement the kind of functionality that the Routing Slip facilitates. Each processing component could have been implemented as a New Web Service Java Collaboration or an eInsight subprocess. The Business Process would orchestrate execution of these components according to routing logic rules it implements.

6.13 Process Manager

One of the Routing Slip solutions involves a single routing component that determines the next component to which a message must be sent and receives the message back once the component is finished with it. This central routing component directs flow of messages using routing logic it embeds. Since it always receives messages that are processed by the processing components, it can modify the route a message is to take based on the outcome of processing by a particular component. Thus, the route the message finally takes may be different from the route a fixed router, which does not use intermediate processing results for routing decisions, would have determined.

Process Manager is a component that implements conditional routing logic and orchestrates execution of other processing components. Java CAPS supports implementation of the Process Manager, with functionality as described in the opening paragraph, as a Java Collaboration using JMS, possibly in Request/Reply mode, to dispatch and receive messages to and from processing components. The disadvantage of this approach is that the routing logic is hidden away in the Java code and, depending on the size and complexity of logic involved, may be difficult to understand.

Java CAPS eInsight Business Process Manager provides a graphical Business Process modeling environment. It overcomes the understandability limitations of a Java-only implementation and offers a number of features for Business Process modeling, component orchestration, and runtime monitoring that are not available with Java-only implementations.

Using eInsight Business Process Manager, you can implement any desired routing and component orchestration solutions. eInsight examples appear in most sections of this book and illustrate all manner of solutions of varying complexity. When a dynamic routing or component orchestration is required, eInsight Business Process can be developed to satisfy the requirement.

6.14 Message Broker

Message Broker [EIP] is an EAI architectural style wherein a component of a messaging system implements centralized routing for all messages flowing through the system. [EIP] also uses the term hub-and-spoke when referring to this architectural style. SeeBeyond's DataGate 3.6 product, predecessor to eGate 4.x, ICAN 5.0, and Java CAPS 5.1, is a Message Broker–based EAI package.

Message Broker architecture allows decoupling of senders from receivers. The senders need not know where the messages are going, and receivers need not know from where the messages are coming. The Message Broker embeds all routing logic necessary to get messages from senders to receivers. This centralizes routing logic maintenance.

In Java CAPS, and ICAN before it, each connectivity map could be considered to represent a Message Broker–based solution. In effect, all collaborations and Business Processes present in a connectivity map route messages from sources to destinations. Each eInsight Business Process could also be considered a Message Broker implementation, as it, too, makes routing decisions when orchestrating a series of activities. Collections of connectivity maps sharing common channels could be considered hierarchies of Message Brokers [EIP].

While Java CAPS can certainly be used to implement centralized routing solutions in the spirit of Message Broker, doing so does not appear particularly necessary or particularly advantageous.

6.15 Chapter Summary

This chapter discussed [EIP] message routing and message routing–related patterns. It included discussion and application of patterns from [EIP] Messaging Systems and Message Routing.

The chapter briefly discussed where a Java CAPS solution developer can make routing decisions and discussed each of the routing patterns, specifically Splitter, Aggregator, Resequencer, Scatter-Gather, Routing Slip, Process Manager, and Message Broker.

Index

A

Aggregator pattern, 78–79, 172–173
  See also Envelope Wrapper
  See also Message Sequence
  See also Splitter
Alert Agent. *See also* Alert Services; Alerts.
  Alert Codes, 251–257
  alerts
    delivering, 248–249
    eInsight Business Processes, starting/stopping, 251
    endpoint support for, 251
    event notification schema, 257
    filtering, 250, 257, 259–260
    recipients, defining, 249–250, 257
    Web Services providers, starting/stopping, 251
  channel types, 246
  channels
    electronic mail, 246–249
    JMS, 246–249
    minimum number, 246
    network management. *See* SNMP.
    SMTP, 246–249
    SNMP, 246–249
  configuring, 246
  reasons for using, 259
Alert Codes, 251–257
Alert Services. *See also* Alert Agent; Alerts.
  closeSession request, 274–275, 296
  filtering alerts, 287–296
  filters, examples, 295–296
  getAlertQueryFields request, 287–292
  getAlerts, with filter, 292–295
  getAllAlerts request, 281–284
  invalid session ID, 296
  listing alerts, 281–284
  Message Codes, 290–291
  observeAllAlerts request, 285
  observing alerts, 284–286
  Query Fields, listing, 287–292
  resetAllAlerts request, 286
  resetting alerts, 286
  session not found, 296
  SOAP Fault, 296
  table types, 291–292
  URl for, 271
Alerts. *See also* Alert Agent; Alert Services.
  delivering, 248–249
  eInsight Business Processes, starting/stopping, 251
  endpoint support for, 251
  event notification schema, 257
  filtering
    codes for, 250–257
    examples, 295–296
    prior to SNMP, 259–260
    by severity, 257
    xxxAllAlerts operations, 287–288
  listing, 281–284
  managing, 271
  observing, 284–286
  recipients, defining, 249–250, 257
  resetting, 286
  Web Services providers, starting/stopping, 251
Any Order Two Items Correlation pattern, 358–360
Any Order Two Items Correlation with Timeout pattern, 360
Application connectivity, 401–403
Application Server Domain. *See* Logical host.
Architecture, Java CAPS
  configuration information, storing, 18–19
  connectivity maps
    associating with physical resources, 21. *See also* Deployment, profiles.
    creating, 20–21
  context, 14–16

Architecture, Java CAPS, *continued*
definition, 13
deployment, 50–53
deployment profiles, 21–22
design time environment, 16–19
developer authentication information, storing, 18
eDesigner IDE, 16–17
high availability
application connectivity, 401–403
Enterprise Manager, 397
failover, intersite, 403–406
failover, intrasite, 402–403
Integration Server, 397–398
introduction, 396
IQ Manager, 398
JMS Grid, 398–401
JMS Grid cluster, 399
JMS Grid network, 399
queue failover, 404–405
replication, Grid-based, 405–406
replication, queue manager disk-based, 406
Repository, 396
UDDI Registry, 397
history of, 13–14
LDAP (Lightweight Directory Access Protocol), 18–19
logical host, 18, 21
management facilities, 19
messaging infrastructure, 18–19
monitoring facilities, 19
Repository Server, 16–19
run time environment, 16–19
solution development stages, 20–23
Asynchronous notification, 94
Auction pattern, 72
Automation. *See* Enterprise Manager, Command-Line Client.

B

Backing up data, 36–39
Batch eWay Streaming, 88–89
Batch FTP eWay, 4
Batch Inbound eWay, 4, 212–213
Batch Local File eWay, 4, 212
Batch Record eWay, 4
BOB (Business Object Broker), 14
BPEL (Business Process Execution Language)
Competing Consumers, 129
message correlation, 337–338
BPEL4WS (Business Process Execution Language for Web Services). *See* BPEL (Business Process Execution Language).
BPM (Business Process Management), 344–349
Branching, 42
Breaking up messages
See Data Streaming
See Message Sequence
See Scatter-Gather
See Splitter
Build process, 54–55
BytesMessage format, 132

C

Canonical Data Model, 206
Centralized routing, 177–178
Channel Adapter pattern, 150–151. *See also* EWay Adapters; Messaging Endpoints.
Channel Purger pattern, 329–331
Channel types, 246
Channels
distributing messages to. *See* Message Dispatcher.
electronic mail, 246–249
handling messages of different types. *See* Message Dispatcher.
JMS, 246–249
minimum number, 246
one message in, multiple out. *See* Publish-Subscribe Channel.
one message in, one out. *See* Point-to-Point Channel.
single message type. *See* Datatype Channel.
SMTP, 246–249
SNMP, 246–249
undeliverable messages. *See* Dead Letter Channel; Invalid Message Channel.
Checked-in state, 40, 46
Checked-out state, 40, 46
Ciphers, 428–429
Claim Check pattern, 205–206. *See also* Content Enricher; Envelope Wrapper.
Clear-text channel, 420
Client authentication, 99

closeSession request, 274–275, 296
Collaboration, 100–104. *See also* Java Collaborations.
Combining messages
See Aggregator.
See Envelope Wrapper.
See Scatter-Gather.
COM/DCOM eWay, deployment limitation, 21
Command Message pattern, 60
Command-line tools, 54–55. *See also* Enterprise Manager, Command-Line Client.
Commit, 100
Compensation, 394–395
Competing Consumers pattern
BPEL (Business Process Execution Language), 129
configuring, 114
Connection Consumer mode, 128–129
description, 217–218
FIFO mode, 80–81, 114, 129
implementing, 127–129
Java Collaborations, 127–129
message order, preserving, 114
receivers
eGate, 127–129
eInsight Business Processes, 129–131
Java Collaborations, 127–129
preserving message ordering, 131
Component paths, listing, 277
Components
disabled, 279–280
invalid, 279–280
listing, 268–269, 275–277
state, monitoring, 318
status, obtaining, 278–279
stopping/starting, 280
tagging, 43
testing, 327–328
Composed Message Processor pattern, 175
Composite Applications, 8, 158
Concurrency. *See also* Competing Consumers.
IQ Manager
Connection Consumer Mode, 105
default mode, 105
enabling, 105
Server Session Pool property, 105
overview, 105
Conditional routing, 177
Configuration information, storing, 18–19
Connection Consumer Mode, 105, 128–129
Connection Factories, 96
Connectivity maps
associating with physical resources, 21
Connection Consumer mode, displaying, 128
creating, 20–21, 28–30
definition, 26
directory structure, 32
dragging and dropping into, 29
hierarchical organization, 28–32
listing variables, 35
mapping variables, 35
misspelling destination names, 29–30
packaging into EAR files, 21
queue consumer concurrency, 80
Connector OTDs, 181
Constants, 33–36
Consumers
competing. *See* Competing Consumers.
event-driven, 216
selective, 219
Content Enricher pattern, 203–204. *See also* Claim Check; Envelope Wrapper.
Content Filter pattern, 204
Content-based router, 165–168
Contention for resources, 131
Context, 14–16
Control Bus pattern, 318
Correlation, definition, 336. *See also* Message correlation.
Correlation Identifiers
derived, 349–354
derived, alternatives to, 354–357
description, 343–344
Correlation Keys, 337–338, 345–349
Correlation patterns. *See* Message Relationship.
Correlation processors, 336
Correlation Sets, 345–349
Counted and Timed Items Correlation pattern, 363
Counted-and-timed-items correlation, 363
Cron, 62
Cryptographic Handshake, 91

D

Data
  extraction and load, 88, 90
  structures. *See* OTDs (Object Type Definitions).
  transfer, 88, 90. *See also* FTP (File Transfer Protocol).
Data Streaming pattern, 88–90
Database sharing, 4–5
Datatype Channel pattern
  definition, 132
  endpoint-dependent datatypes, 133
  Java Collaborations, 134
  JMS message body formats, 132–133
  multiple datatypes, 134–135
Dead Letter Channel pattern. *See also* Invalid Message Channel.
  IQ Manager, 137–138
  Java CAPS, 5.1.2 and greater, 139–140
  Java CAPS, prior to 5.1.2, 137–138
  JMS Redelivery feature, 137–138
  overview, 136
  redelivery, 137–140
Dead Letter Queues. *See* Dead Letter Channel.
Deadlocks, 131
De-duping messages, 222
Deleting messages. *See* Discarding messages.
Delimited Envelope Wrapper, 190–192
Delivery mode, configuring, 106–107
Delivery time, calculating, 84
Dependent projects, 37, 39
Deployment
  architecture, 50–53
  EAR files, 51–52
  logical break points, 9
  management
    branching, 42
    checked-in state, 40, 46
    checked-out state, 40, 46
    circumventing, 42
    command-line tools, 54–55
    deployment profile snapshots, 43–46
    project tagging, 43
    retrieved state, 40
    third-party systems, 46–49
    unique user ID requirement, 42
    version history, 41
    version isolation, 42
  to multiple physical environments, 22
  profiles
    configuration settings, 33–36
    constants, 33–36
    creating a directory for, 30–31
    creating from a single connectivity map, 36
    definition, 9, 21–22
    directory structure, 32
    exporting, 39
    mapping components, 9–11
    naming, 30–31
    snapshots, 43–46
    variables, 33–36
  scripting, 54–56. *See also* Enterprise Manager, Command-Line Client.
Deployment Profile Editor, 35, 43–46
Derived Correlation Identifiers, 349–357. *See also* Correlation Identifiers.
Design time environment, 16–19
Destinations. *See* JMS Destinations.
Detour pattern, 318–319
Developer authentication information, storing, 18
Digital signatures, 91
Discarding messages, 83–85, 243–244, 327–328
Distributing components
  eGate, 384–386
  eInsight, 387
  overview, 383–384
Document Message pattern, 60
Duplicate messages, 174, 220–223
Durable Subscriber pattern, 219–220
Durable subscriptions, 98, 219–220
Dynamic router, 165–168. *See also* Message Dispatcher.
Dynamic routing, 176
Dynamic selectors, 109–114, 366–369

E

EAR files
  deploying to the runtime environment, 51–52
  description, 8–9
  generating, 9
  multiple deployments, 22
  names
    default, 30–32
    maximum length, 32
    renaming, 27
    setting with deployment profile properties, 32
  packaging connectivity maps, 21

eDesigner, 16–17, 25
eGate
  Competing Consumers, 127–129
  component distribution, 384–386
  correlation, with dynamic selectors, 366–369
  resilience, 384–386
  scalabilty, 384–386
eInsight, Business Processes
  Competing Consumers, 129–131
  components, distributing, 387
  Datatype Channel pattern, 135
  default processing model, 217
  Guaranteed Delivery pattern
    guaranteed nondelivery on failure, 146
    overview, 145–146
    persistence, 147–148
    rolling back transactions, 135, 146–148
    XA transactionality, 146–147
  persistence, enabling, 233–235
  Request/Reply pattern, 65, 70–72, 74
  resilience, 387
  scalabilty, 387
  serializing processing, 131
  starting/stopping, alerts for, 251
  XA transactionality, 131
eInsight, Correlation Processor
  automatic correlation, 344–349
  BPM (Business Process Management), 344–349
  Correlation Keys, 337–338, 345–349
  Correlation Sets, 345–349
eInsight, reusability
  subprocesses
    Notification relationships, 375, 376–378, 382
    OneWayOperation relationships, 375, 376, 382
    overview, 373–375
    Request/Reply subprocess, 375–376, 382
    WSDL definitions, 373–375
  Web Services
    Notification Web Service, 381–382
    OneWayOperation Web Service, 381–382
    Request/Reply Web Service, 378–382
    servlet context, 379–380
eInsight engine, 172
Electronic mail channels, 246–249
Encryption
  ciphers, 428–429
  messages in transit, 91
  public keys, 417–419
  SSL (Secure Sockets Layer), 417
Endpoint-dependent datatypes, 133
Endpoints
  See Channel Adapter.
  See eWay Adapters.
  See Messaging Endpoints.
Enriching messages, 78–79
  See also Claim Check.
  See also Content Enricher.
  See also Envelope Wrapper.
Enterprise Applications, 23. *See also* Projects; Solutions.
Enterprise Designer, 16–17, 25
Enterprise Manager
  Command-Line Client
    alert management, 271
    credentials, 267
    currently deployed components, listing, 268–269
    host name, 267
    Java Collaboration, 269–271
    port number, 267
    runtime methods, listing, 267–268
    service state, obtaining, 269–271
    usage notes, 267
  high-availability architecture, 397
Enterprise Manager, Web Service API
  Alert Services
    closeSession request, 274–275, 296
    filtering alerts, 287–296
    filters, examples, 295–296
    getAlertQueryFields request, 287–292
    getAlerts, with filter, 292–295
    getAllAlerts request, 281–284
    invalid session ID, 296
    listing alerts, 281–284
    Message Codes, 290–291
    observeAllAlerts request, 285
    observing alerts, 284–286
    Query Fields, listing, 287–292
    resetAllAlerts request, 286
    resetting alerts, 286
    SOAP Fault, 296

Enterprise Manager, Web Service API, *continued*
  Alert Services, *continued*
    table types, 291–292
    URl for, 271
  Login Services
    description, 273–275
    openSession request, 273–275
    Session IDs, obtaining, 273–275, 275
    URL for, 271
  Runtime Services
    closeSession request, 274–275
    disabled components, 279–280
    getComponentList request, 275–277
    invalid components, 279–280
    listing component paths, 277
    listing components, 275–277
    obtaining component status, 278–279
    SOAP Fault, 279–280
    stopping/starting components, 280
    URl for, 271
  Service Manager Services
    available services, listing, 272–273
    description, 272–273
    URL for, 271
  Session IDs
    after restart, 274–275
    invalid, 274–275
    obtaining, 273–275, 275
  WSDL interface definitions, 271–272, 280
Envelope Wrapper pattern
  See also Aggregator.
  See also Claim Check.
  See also Content Enricher.
  See also Message Sequence.
  See also Splitter.
  message sequencing, 78–79
  OTD
    delimited Envelope Wrapper, 190–192
    JMS properties, 201–202
    overview, 188–189
    XML within XML, 192–201
  sequence metadata, 79
ETL (Extract, Transfer, and Load), 88. *See also* Data Streaming.
eTL Streaming, 90
Event Message pattern, 60
Event notification schema, 257
Event-Driven Consumer pattern, 216
Event-driven consumers, 216
Events
  notification. *See* Alert Agent; Alert Services; Alerts.
  SNMP. *See* Traps.
  triggering, 62
eWay Adapters. *See also* Channel Adapter; HTTP eWay; Messaging Endpoints.
  Batch FTP eWay, 4
  Batch Inbound eWay, 4
  Batch Local File eWay, 4
  Batch Record eWay, 4
  COM/DCOM, 6
  description, 5
Exception handling
  faults handlers, 392–393
  faults in Business Processes, 390–393
  higher-level, 393
  Java collaborations, 388–390
Expiration, messages, 82–86, 144
Exporting
  deployment profiles, 39
  projects, 36–39
Extensible Markup Language (XML)
  appropriate use of, 180–181
  enveloping within XML, 192–201
Extract, Transfer, and Load (ETL), 88. *See also* Data Streaming.

F

Factoring solutions. *See* Distributing components.
Failover, 402–406
Fault handlers, 392–393
Faults, in Business Processes, 390–393
Fault-tolerance, 94. *See also* Resilience.
FIFO modes
  Competing Consumers, 114, 129
  effect on Connection Consumer mode, 129
  IQ Manager, 114
  JMS Message Server, 80–81
File systems, polling, 211–214
File Transfer Protocol (FTP), 433–435
File transfer security, 433–435
Files
  reading, 211–214
  renaming, 212–213

Filtering
  alerts
    codes for, 250–257
    examples, 295–296
    prior to SNMP, 259–260
    by severity, 257
    xxxAllAlerts operations, 287–288
  Content Filter pattern, 204
  getAlerts requests, 292–295
  JMS selectors. *See also* Message Filter.
    dynamic, 109–114
    overview, 107
    static, 107–109
  Message Filter pattern, 168–169. *See also* JMS selectors.
  messages, 204
Fixed router, 163–165
Format indicator, 86–88
Format Indicator pattern, 86–88. *See also* Messaging Bridge.
Fragmenting messages
  See Data Streaming.
  See Message Sequence.
  See Scatter-Gather.
  See Splitter.
FTP (File Transfer Protocol), 433–435
Fully Concurrent Processing, 80
Fully Serialized Processing, 80

G

GET method, 73–74
getAlertQueryFields request, 287–292
getAlerts, with filter, 292–295
getAllAlerts request, 281–284
getComponentList request, 275–277
GMT (Greenwich mean time), 83–84
Grid. *See* JMS Grid.
Grid clusters. *See* JMS Grid, clusters.
Guaranteed Delivery pattern
  delivery mode, configuring, 106–107
  eInsight Business Processes
    guaranteed nondelivery on failure, 146
    overview, 145–146
    persistence, 147–148
    rolling back transactions, 135, 146–148
    XA transactionality, 146–147
  Java CAPS facilities for, 141–142
  JMS-based
    message expiration, 144
    Persistent Delivery mode, 145
    required services, 143–144
    transacted sessions, 144–145
  Nonpersistent Delivery mode, 106–107
  overview, 140
  persistence, 142–143
  Persistent Delivery mode, 106–107
  requirement for, 140–141
  solution-specific, 148–149
  transparent replication, 143
Guaranteed nondelivery on failure, 146

H

Header-Counted-Items Correlation pattern, 362–363
Header-Items-Trailer Correlation pattern, 357–358
Heap graph, 304
Heartbeat-based schedulers, 63
High-availability architecture
  introduction, 396
  Java CAPS components
    application connectivity, 401–403
    Enterprise Manager, 397
    failover, intersite, 403–406
    failover, intrasite, 402–403
    Integration Server, 397–398
    IQ Manager, 398
    JMS Grid, 398–401
    JMS Grid cluster, 399
    JMS Grid network, 399
    queue failover, 404–405
    replication, Grid-based, 405–406
    replication, queue manager disk-based, 406
    Repository, 396
    UDDI Registry, 397
Hostname verification, 432
HTTP (Hypertext Transfer Protocol)
  Basic Authentication, 410–415
  connectors, 29
  GET method, 73–74
  listener port assignments, 419
  POST method, 73–74
  Proxy Server configuration, 409–410

HTTP (Hypertext Transfer Protocol), *continued*
Request/Reply, 73–74
Server, 73–74
Server/Responder, 419
HTTP eWay
clear-text channel, 420
client/server projects, 419
mutual authentication
client configuration, 425–426
exercising the channel, 427–428
server configuration, 424–425
purpose of, 73–74
server-side authentication, 420–424
SSL (Secure Sockets Layer), 427–428

I

Idempotence, 220
Idempotent Receiver pattern, 220–223
Importing projects, 36–39
Inspecting messages, 100–104
Integration Server
description, 8
high-availability architecture, 397–398
keystores, referencing, 99
truststores, referencing, 99
Integration styles
centralized *vs.* distributed, 8–11
database sharing, 4–5
file transfer, 3–4
messaging, 6–7
remote procedure invocation, 5–6
resiliency, 8–11
scalabilty, 8–11
service orchestration, 7–8
Invalid Message Channel pattern, 136. *See also* Dead Letter Channel.
Invalid Message Queues. *See* Dead Letter Channel.
Invalid session ID, 296
IQ Manager. *See also* JMS Message Server.
automatic destination creation, 29–30
concurrency
Connection Consumer Mode, 105
default mode, 105
enabling, 105
Server Session Pool property, 105
Connection Consumer mode, displaying, 128
Dead Letter Channel pattern, 137–138
FIFO modes, 114
high-availability architecture, 398
JMS Destinations
checking existence, 98
creating, 97–98
destroying, 97–98
temporary, 98–99
JMS selectors
dynamic, 109–114
overview, 107
static, 107–109
JMS transactions
commit, 100
definition, 99
inspecting messages, 100–104
purpose of, 100
scope, 100
Transacted mode behavior, 104
Transacted setting, 103–104
Transactionality property, 103
uncommitted messages, 100
message journaling, 117–119
misspelling destination names, 29–30
persistence
delivery mode, configuring, 106–107
description, 105–107
guaranteed delivery, 106
JMS delivery mode *vs.* subscription durability, 106
redelivery handling, 116–117
security, 99
throttling, 115–116
transactionality
commit, 100
definition, 99
inspecting messages, 100–104
purpose of, 100
scope, 100
Transacted mode behavior, 104
Transacted setting, 103–104
Transactionality property, 103
uncommitted messages, 100
Items-trailer correlation, 360–362, 367–369
Items-Trailer Correlation pattern, 360–362, 367–369

J

Java CAPS
  communication with third-party applications. *See* Channel Adapter.
  JMS properties, setting, 107–109
  JMX Console, 297–303
  Repository. *See* Repository, Java CAPS.
Java Collaborations
  Competing Consumers, 127–129
  Datatype Channel pattern, 134
  default processing model, 217
  exception handling, 388–390
  receive method, examples, 65–66, 67–68
  Request/Reply pattern, 65–72
  service state, obtaining, 269–270
  starting/stopping, 270–271
  state, obtaining, 269–271
Java Management Extensions (JMX). *See* JMX (Java Management Extensions).
Java Message Service (JMS). *See* JMS (Java Message Service).
Java Naming and Directory Interface (JNDI), 109–112
JCA-compliant adapters. *See* eWay Adapters.
JCD (Java Collaboration Definition), 101–103
JCE (Java Cryptography Extension), 428–429
JConsole, enabling, 303–307
JMS (Java Message Service)
  administration, 94
  asynchronous notification, 94
  channels, 246–249
  clusters. *See* JMS Grid, clusters.
  Connection Factories, 96
  delivery mode *vs.* subscription durability, 106
  fault-tolerance, 94. *See also* JMS Grid, clusters.
  FIFO modes, 79, 80–81
  Grid. *See* JMS Grid.
  integrating non-Java environments, 95–96. *See also* Messaging Bridge.
  interoperability, 95. *See also* Messaging Bridge.
  latency, 313–317
  load-balancing, 94
  message body formats, 132–133
  message repository, 94
  message type definitions, 94
  OTDs, 201–202
  overview, 94–95
  physical implementation issues, 95
  point-to-point message, 96. *See also* JMS topics.
  properties, setting with Java CAPS, 107–109
  publish-subscribe messaging, 96. *See also* JMS queues.
  Redelivery feature, 137–138
  Request/Reply pattern, 64–72, 371–372
  resilience, 119–126. *See also* JMS Grid.
  security, 94–95
  Serial mode concurrency, 79–80
  serializing business processes, 81–82
JMS Destinations. *See also* JMS queues.wire protocols, 94–95
  creating, 29
  description, 96
  IQ Manager
    checking existence, 98
    creating, 97–98
    destroying, 97–98
    temporary, 98–99
  polling, 214–216
JMS Grid
  Admin Console, 124–126
  administering, 124–126
  clusters
    configuration, 120–122
    definition, 119
    high-availability architecture, 399
    illustration, 120
    size, 120
    transparent replication, 119
    uses for, 120, 124
  features, 119
  high-availability architecture, 398–401
  network, 399
  network configuration, 120–121
JMS Message Server. *See also* IQ Manager.
  discarding messages, 96–97
  FIFO modes, 79–80, 114
  guaranteed delivery, 143–145
  interoperability, 95–96
  message expiration, 82–84, 144
  Message Journaling, 85
  Messaging Bridge, 151–156
  overview, 18–22
  persistence, 105–107
  Persistent Delivery mode, 145

JMS Message Server, *continued*
redelivery, 116–117
resilience, 119–126
security, 99
selectors, 107–114
Sync to Disk, 145
throttling, 115–116
transacted sessions, 144–145
XA sessions, 144–145
JMS queues. *See also* JMS Destinations; Point-to-Point Channel.
creating, 29
definition, 96
failover, 404–405
journaled, listing, 244–245
listing, 241
message content, displaying, 244–245
messages, listing, 242–243
no receivers, 98
status, displaying, 242
vs. topics, 96–97
JMS selectors. *See also* Message Filter.
dynamic, 109–114
overview, 107
static, 107–109
JMS topics. *See also* Publish-Subscribe Channel.
creating, 29
definition, 96
durable subscriptions, removing, 98
no subscribers, 98
vs. queues, 96–97
JMS transactions
commit, 100
definition, 99
inspecting messages, 100–104
purpose of, 100
scope, 100
Transacted mode behavior, 104
Transacted setting, 103–104
Transactionality property, 103
uncommitted messages, 100
JMSCorrelationID, 337
JMX (Java Management Extensions)
agent, enabling, 303
Java CAPS JMX Console, 297–303
JConsole, enabling, 303–307
JMX agent, enabling, 303
JMX Console, 307–308
log dumps, 298–303
MBeans, 297–303
programmatic management, 308–313
uses for, 297
JMX Console, 307–308
JNDI (Java Naming and Directory Interface), 109–112
Job scheduling. *See* Scheduling.
Journaling messages, 84–86, 117–119, 244–245
Journaling status, checking, 244

K

Keystores, referencing, 99

L

Latency, 313–317
LDAP (Lightweight Directory Access Protocol), 18–19
Load-balancing, 94
Log dumps, 298–303
Logical host, 18, 21
Login Services
description, 273–275
openSession request, 273–275
Session IDs, obtaining, 273–275, 275
URL for, 271

M

Management facilities. *See* Monitoring and management.
MapMessage format, 132–133
Mapping
components, deployment profiles, 9–11
connectivity. *See* Connectivity maps.
variables, 35
Marshaling messages, 180–181. *See also* Serializing messages.
Maximum Lifetime, 83
MBeans, 297–303
MDN (Message Disposition Notification), 77
Message body formats, 132–133
Message Broker pattern, 177–178
Message Bus pattern, 157–158
Message Codes, 290–291
Message concept, 179–180
Message correlation
any order two items, 358–360
any order two items with timeout, 360

BPEL correlations, 337–338
correlation, definition, 336
Correlation Identifiers
derived, 349–354
derived, alternatives to, 354–357
description, 343–344
Correlation Keys, 337–338
correlation processors, 336
counted and timed items, 363
eGate, with dynamic selectors, 366–369
eInsight Correlation Processor
automatic correlation, 344–349
BPM (Business Process Management), 344–349
Correlation Keys, 345–349
Correlation Sets, 345–349
queuing messages manually, 342–343
receive activities, 338–341
eInsight correlations, 337–338
header-counted-items, 362–363
header-items-trailer, 357–358
items-trailer, 360–362, 367–369
JMSCorrelationID, 337
overview, 336
predefined JMS Header fields, 337
scatter-gather, 364–365
timed items, 363–364
Message Dispatcher pattern, 218. *See also* Dynamic router.
Message Disposition Notification (MDN), 77
Message exchange patterns
Aggregator, 78–79
Auction, 72
Command Message, 60
Data Streaming, 88–90
Document Message, 60
Envelope Wrapper, 78–79
Event Message, 60
Format Indicator, 86–88
Message Expiration, 82–86
Message Security, 90–91
Message Sequence
combining messages, 78–79
Competing Consumers, 80–81
enriching messages, 78–79
FIFO modes, 79, 80–81
Fully Concurrent Processing, 80
Fully Serialized Processing, 80
JMS Serial mode concurrency, 79–80
Protected Concurrent Processing, 80
sequence metadata, 79
Serial Concurrency mode, 80
serializing business processes, 81–82
splitting messages, 78–79
Request/Reply
Auction pattern, 72
eInsight Business Process, 65, 70–72
eInsight subprocesses, 74
HTTP eWay, 73–74
HTTP GET method, 73–74
HTTP POST method, 73–74
HTTP Request/Reply, 73–74
HTTP Server, 73–74
Java Collaboration, 65–72
JMS Request/Reply, 64–72
overview, 63–64
requestReply( ) method, 72
SOAP Request/Reply, 74–75
temporary queues, 70
Web Services implementation, 75–76
Return Address, 76–77
Splitter, 78–79
Message Expiration pattern, 82–86
Message Filter pattern, 168–169. *See also* JMS selectors.
Message History pattern, 321–325
Message ID, 222–223
Message Journaling service, 84–85
Message OTDs, 181
Message Relationship patterns
Any Order Two Items Correlation, 358–360
Any Order Two Items Correlation with Time-out, 360
Counted and Timed Items Correlation, 363
Header-Counted-Items Correlation, 362–363
Header-Items-Trailer Correlation, 357–358
Items-Trailer Correlation, 360–362, 367–369
Scatter-Gather Correlation, 364–365
Timed Items Correlation, 363–364
Message repository, 94
Message Security, 90–91
Message selectors. *See* JMS selectors.

Message Sequence pattern
See also Aggregator.
See also Envelope Wrapper.
See also Splitter.
combining messages, 78–79
Competing Consumers, 80–81
enriching messages, 78–79
FIFO modes, 79, 80–81
Fully Concurrent Processing, 80
Fully Serialized Processing, 80
JMS Serial mode concurrency, 79–80
Protected Concurrent Processing, 80
sequence metadata, 79
Serial Concurrency mode, 80
serializing business processes, 81–82
splitting messages, 78–79
Message servers
default. *See* IQ Manager.
Java CAPS JMS Message Server
journaling, 85
selectors, 107–114
JMS Message Server
APIs, 96
bridging independent solutions, 152–156
bridging JMS implementations, 156
delivery mode, configuring, 106–107
FIFO modes, 114
guaranteed delivery, 143–145
message expiration, 144
message sequencing, 114
persistence, 105–107
Persistent Delivery mode, 145
redelivery, 116–117
resilience, 119–126
security, 99–100
Sync to Disk, 145
throttling, 115–116
Transacted Sessions, 144–145
XA Sessions, 144–145
Message Store pattern, 325–327
Message type definitions, 94
Message-format agnostic, 86
Messages
breaking up
See Data Streaming.
See Message Sequence.
See Scatter-Gather.
See Splitter.
buffer overrun, 174
combining
See Aggregator.
See Envelope Wrapper.
See Scatter-Gather.
concurrent processing. *See* Competing Consumers.
de-duping, 222
delivery time, calculating, 84
different types over same channel. *See* Message Dispatcher.
discarding, 83–85, 243–244, 327–328
displaying content, 243
distributing to different channels. *See* Message Dispatcher.
diverting to a different queue, 318–321
duplicates, 174, 220–223
enriching, 78–79. *See also* Envelope Wrapper.
expiration, 82–86, 144
FIFO modes, 79, 80–81
filtering, 204
format indicator, 86–88
formats, transforming. *See* Messages, transformation.
GMT (Greenwich mean time), 83–84
idempotence, 220
inspecting, 100–104
journaled queues, listing, 244–245
journaling, 84–86, 117–119
journaling status, checking, 244
large, 78–79, 88–90
marshaling, 180–181. *See also* Serializing.
Maximum Lifetime, 83
one receiver per, 131
order, preserving, 114, 131
queuing manually, 342–343
receiving
multiple times. *See* Idempotent Receiver.
selectively. *See* Selective Consumer.
redelivery, 137–140
relationship patterns. *See* Message Relationship patterns.
removing information from, 204, 205–206
retaining, 219–220. *See also* Durable subscriptions.
routes, tracking, 321–325

routing
  Aggregator pattern, 172–173
  breaking up messages, 171–172, 175
  centralized, 177–178
  combining messages, 172–173, 175
  Composed Message Processor pattern, 175
  conditional, 177
  content-based router, 165–168
  dynamic router, 165–168
  dynamic routing, 176
  fixed router, 163–165
  Message Broker pattern, 177–178
  Message Filter, 168–169
  overview, 161–163
  Process Manager pattern, 177
  recipient list, 169–170
  Resequencer pattern, 173–175
  resequencing messages, 173–175
  Routing Slip pattern, 176
  Scatter-Gather pattern, 175
  selective processing, 168–169
  Splitter pattern, 171–172
routing patterns
  Aggregator, 172–173
  Composed Message Processor, 175
  Message Broker, 177–178
  Message Filter, 168–169
  Process Manager, 177
  Resequencer, 173–175
  Routing Slip, 176
  Scatter-Gather, 175
  Splitter, 171–172
safety, 220
saving copies of. *See* Journaling.
security. *See* Security.
selective processing, 168–169
sequence numbers, 174
serializing, 173–175. *See also* Marshaling; Message Sequence.
slowing down, 115–116
splitting, 78–79
storing, 325–327
timeTolive property, 83–85
traffic, monitoring, 318
transformation patterns. *See also* Envelope Wrapper.
  Canonical Data Model, 206
  Claim Check, 205–206
  Content Enricher, 203–204
  Content Filter, 204
  Normalizer, 206
uncommitted, 100
UTC (Universal Time Coordinated), 84
Messages, infrastructure patterns
  Channel Adapter, 150–151
  Datatype Channel, 132–135
  Dead Letter Channel, 136–140
  Guaranteed Delivery
    eInsight Business Processes, 145–148
    guaranteed nondelivery on failure, 146
    Java CAPS facilities for, 141–142
    JMS-based, 144–145
    overview, 140, 145–146
    persistence, 142–143, 147–148
    requirement for, 140–141
    rolling back transactions, 146–148
    solution-specific, 148–149
    transparent replication, 143
    XA transactionality, 146–147
  Invalid Message Channel, 136
  Message Bus, 157–158
  Messaging Bridge, 151–157
Messaging Bridge pattern. *See also* Format Indicator.
  independent Java CAPS solutions, 152–156
  other bridging solutions, 156–157
  other JMS implementations, 156
  overview, 151
Messaging Endpoints patterns. *See also* Channel Adapter; EWay Adapters.
  Competing Consumers, 217–218
  Durable Subscriber, 219–220
  Event-Driven Consumer, 216
  Idempotent Receiver, 220–223
  Message Dispatcher, 218
  Messaging Gateway, 209–210
  Polling Consumer
    definition, 211
    file systems, 211–214
    JMS Destinations, 214–216
    other batch pollers, 214
  Selective Consumer, 219
  Service Activator, 223–225
  Transactional Client, 210–211
Messaging Gateway pattern, 209–210
Migration, third-party version system, 46

Monitoring and management. *See also* JMX (Java Management Extensions).
architecture, 19
automating. *See* Enterprise Manager, Command-Line Client.
component state, 318
deployment
branching, 42
checked-in state, 40, 46
checked-out state, 40, 46
circumventing, 42
command-line tools, 54–55
deployment profile snapshots, 43–46
project tagging, 43
retrieved state, 40
third-party systems, 46–49
unique user ID requirement, 42
version history, 41
version isolation, 42
deployments. *See* Deployment, management.
discarding messages, 327–328
diverting messages to a different queue, 318–321
eGate-based solutions, 228–233
eInsight-based solutions, 233–235
event notification. *See* Alert Agent; Alert Services; Alerts.
IQ Manager
description, 235
discarding messages, 243–244
displaying message content, 243
journaled queues, listing, 244–245
journaling status, checking, 244
monitoring features, 235–240
queue messages, listing, 242–243
queue status, displaying, 242
queues, listing, 241
receivers, listing, 241–242
status, displaying, 241
stcmsctrutil utility, 240
JMS latency, 313–317
message traffic, 318
network management. *See* SNMP Agent.
networks. *See* SNMP Agent.
overview, 227–228
patterns
Channel Purger, 329–331
Control Bus, 318
Detour, 318–319
Message History, 321–325
Message Store, 325–327
overview, 317–318
Test message, 327–328
Wire Tap, 319–321
performance data collection, 313–317
persistence, enabling, 233–235
releases. *See* Deployment, management.
runtime performance data, collecting, 315–317
scripting. *See* Enterprise Manager, Command-Line Client.
solution-specific
Channel Purger pattern, 329–331
Control Bus pattern, 318
Detour pattern, 318–319. *See also* Wire Tap.
discarding messages, 327–328
diverting messages to a different queue, 318–321
Message History pattern, 321–325
Message Store pattern, 325–327
monitor component state, 318
monitoring message traffic, 318
overview, 317–318
storing messages, 325–327
Test message pattern, 327–328
testing components, 327–328
tracking message routes, 321–325
Wire Tap pattern, 319–321. *See also* Detour.
storing messages, 325–327
Sun SeeBeyond JMS Message Server, 245
testing components, 327–328
tracking message routes, 321–325
Web Services-based. *See* Enterprise Manager, Web Service API.
Mutual authentication
HTTP eWay, 424–428
SSL (Secure Sockets Layer), 418
Web Services SSL, 431–432

N

Naming
deployment profiles, 30–31
EAR files
default, 30–32
maximum length, 32
renaming, 27
setting with deployment profile properties, 32
files, 212–213
projects, 27, 30–31
Network management. *See* SNMP channels.

New Web Service collaboration, 372–373
Normalizer pattern, 206
Notification, events. *See* Alert Agent; Alert Services; Alerts.
Notification relationships
eInsight subprocesses, 375, 376–378, 382
eInsight Web Services, 381–382

O

ObjectMessage buffer, 132–133
ObjectMessage format, 132
observeAllAlerts request, 285
OneWayOperation relationships
eInsight subprocesses, 375, 376, 382
eInsight Web Services, 381–382
openSession request, 273–275
OTD Wizards, 187
OTDs (Object Type Definitions)
Connector OTDs, 181
definition, 180
Envelope Wrapper pattern
delimited Envelope Wrapper, 190–192
JMS properties, 201–202
overview, 188–189
XML within XML, 192–201
marshaling messages, 180–181
Message OTDs, 181
Oracle table, example, 181–187
types of, 181
Overwriting projects, 39

P

Performance data collection, 313–317
Persistence
enabling, 233–235
Guaranteed Delivery pattern, 142–143, 147–148
IQ Manager
delivery mode, configuring, 106–107
description, 105–107
guaranteed delivery, 106
JMS delivery mode *vs.* subscription durability, 106
Persistent Delivery mode, 145
Point-to-Point Channel, 131. *See also* JMS topics.
Point-to-point message, 96
Polling
Batch Inbound eWay, 212–213
Batch Local File eWay, 212
file systems, 211–214
JMS Destinations, 214–216
other batch pollers, 214
Polling Consumer pattern
definition, 211
file systems, 211–214
JMS Destinations, 214–216
other batch pollers, 214
Port number, 267
POST method, 73–74
Predefined JMS Header fields, 337
Process Manager pattern, 177
Profiles, deployment
configuration settings, 33–36
constants, 33–36
creating a directory for, 30–31
creating from a single connectivity map, 36
definition, 9, 21–22
directory structure, 32
exporting, 39
mapping components, 9–11
naming, 30–31
snapshots, 43–46
variables, 33–36
Project structure. *See also* Connectivity maps; Deployment, profiles.
constants, 33–36
factors influencing, 25
hierarchical organization
connectivity maps, 28–32
deployment profiles, 28–32
description, 26–28
logical solutions, 26
physical deployment, 26
project folders, creating, 27
shared objects, creating, 38
variables, 33–36
Projects. *See also* Enterprise Applications; Solutions.
archives. *See* EAR files.
backing up, 36–39
definition, 25
dependent, 37, 39
development-time artifacts, 25. *See also specific artifacts.*
exporting/importing, 36–39
naming, 27
overwriting on import, 39
release management. *See* Deployment, management.
renaming, 27

Projects, *continued*
  rolling back, 39, 46–49
  tagging, 43
  version management. *See* Deployment, management.
Properties, setting with Java CAPS, 107–109
Protected Concurrent Processing, 80
Public keys, 417–419
Publish-Subscribe Channel, 132. *See also* JMS queues.
Publish-subscribe messaging, 96

Q

Query Fields, listing, 287–292
Queue consumer concurrency, 80
Queues. *See* JMS queues.
Queuing messages manually, 342–343

R

Reading files, 211–214
Reassembling messages. *See* Aggregator; Scatter-Gather.
Receivers
  Competing Consumers
    eGate, 127–129
    eInsight Business Processes, 129–131
    Java Collaborations, 127–129
    preserving message ordering, 131
  listing, 241–242
Recipient list, 169–170
Recipients, checking for. *See* Request/Reply.
Redelivery feature, 137–140
Redelivery handling, 116–117. *See also* Messages, routing.
Relationship patterns. *See* Message Relationship patterns.
Release management. *See* Deployment, management.
Remote Method Invocation (RMI), 6
Remote procedure call (RPC), 5–6
Renaming. *See* Naming.
Replication
  Grid-based, 405–406
  queue manager disk-based, 406
  transparent, 119, 143
Repository, Java CAPS
  backing up, 36–39
  exporting projects, 36–39
  high-availability architecture, 396
  importing projects, 36–39
  overwriting projects, 39
  rolling back transactions, 39
Repository Server, 16–19
Request/Reply pattern
  Auction pattern, 72
  eInsight Business Process, 65, 70–72
  eInsight subprocesses, 74
  HTTP eWay, 73–74
  HTTP GET method, 73–74
  HTTP POST method, 73–74
  HTTP Request/Reply, 73–74. *See also* SOAP Request/Reply.
  HTTP Server, 73–74
  Java Collaboration, 65–72
  JMS Request/Reply, 64–72
  overview, 63–64
  requestReply( ) method, 72
  SOAP, 74–75. *See also* HTTP (Hypertext Transfer Protocol), Request/Reply.
  temporary queues, 70
  using, 371–372
  Web Services implementation, 75–76
Request/Reply relationships
  eInsight subprocesses, 375–376, 382
  eInsight Web Services, 378–382
requestReply( ) method, 72
Resequencer pattern, 173–175
Resequencing messages. *See* Serializing messages.
resetAllAlerts request, 286
Resilience. *See also* High-availability architecture.
  distributing components
    eGate components, 384–386
    eInsight components, 387
    overview, 383–384
  exception handling
    faults handlers, 392–393
    faults in Business Processes, 390–393
    higher-level, 393
    Java collaborations, 388–390
  JMS (Java Message Service), 119–126
Retaining messages, 219–220
Retrieved state, 40
Return Address pattern, 76–77
Reusability
  eInsight subprocesses
    Notification relationships, 375, 376–378, 382
    OneWayOperation relationships, 375, 376, 382

overview, 373–375
Request/Reply subprocess, 375–376, 382
WSDL definitions, 373–375
eInsight Web Services
Notification Web Service, 381–382
OneWayOperation Web Service, 381–382
Request/Reply Web Service, 378–382
servlet context, 379–380
JMS Request/Reply pattern, 371–372
New Web Service collaboration, 372–373
Service Activator, 223–225
RMI (Remote Method Invocation), 6
Rolling back
dependent projects, 39
projects, 39, 46–49
transactions, 135, 146–148
Routing. *See* Messages, routing.
Routing Slip pattern, 176
RPC (remote procedure call), 5–6
RPC/Encoded style, 6
Runtime environment, 16–19
Runtime methods, listing, 267–268
Runtime performance data, collecting, 315–317
Runtime Services
closeSession request, 274–275
disabled components, 279–280
getComponentList request, 275–277
invalid components, 279–280
listing component paths, 277
listing components, 275–277
obtaining component status, 278–279
SOAP Fault, 279–280
stopping/starting components, 280
URl for, 271

S

SAN (storage area network), 401–407
Scalabilty
centralized *vs.* distributed solutions, 8–11
distributing components
eGate, 384–386
eInsight, 387
overview, 383–384
Scatter-Gather Correlation, 364–365
Scatter-Gather pattern, 175
Scheduler Connector, 61
Scheduling
Cron, 62
event triggering, 62
external schedulers, 62–63
heartbeat-based solutions, 63
job identifiers, 62
job parameters, 62
Scheduler Connector, 61
Unix Cron, 62
Windows Task Scheduler, 62
Schema Runtime Environment (SRE), 151
SCP (Secure Copy Protocol), 434
Scripting, 54–56. *See also* Enterprise Manager, Command-Line Client.
Secure File Transfer Protocol (SFTP), 434
Secure Shell (SSH), 434
Secure Sockets Layer (SSL). *See* SSL (Secure Sockets Layer).
Security. *See also* SSL (Secure Sockets Layer).
authentication
client-side, 99
HTTP Basic Authentication, 410–415
mutual, 418, 424–428, 431–432
server-side, 420–424
Cryptographic Handshake, 91
developer authentication information, storing, 18
digital signatures, 91
encryption
ciphers, 428–429
messages in transit, 91
public keys, 417–419
SSL (Secure Sockets Layer), 417
file transfer, 433–435
FTP (File Transfer Protocol), 433–435
HTTP listener port assignments, 419
HTTP Proxy Server configuration, 409–410
HTTP Server/Responder, 419
JMS (Java Message Service), 94–95
messages in transit, 90–91, 99
server-side authentication, 418, 430–431
Sun SeeBeyond IQ Manager, 99
TLS (Transport Layer Security), 415
Web proxy, 409. *See also* HTTP (Hypertext Transfer Protocol), Proxy Server.
Selective Consumer pattern, 219
Selective consumers, 219
Selectors. *See* JMS selectors.
Sequence metadata, 79
Serial Concurrency mode, 80

Serializing messages. *See also* Marshaling; Message Sequence.
business processes, 81–82
eInsight Business Processes, 131
Fully Serialized Processing, 80
JMS business processes, 81–82
Resequencer pattern, 173–175
Server Session Pool property, 105
Server-side authentication
HTTP eWay, 420–424
SSL (Secure Sockets Layer), 418
Web Services SSL, 430–431
Service Activator pattern, 223–225
Service Manager Services
available services, listing, 272–273
description, 272–273
URL for, 271
Servlet context, 379–380
Session IDs, 273–275
Session not found, 296
SFTP (Secure File Transfer Protocol), 434
SMTP channels, 246–249
Snapshots, deployment profiles, 43–46
SNMP Agent
Auto-Routing, 262, 263
configuration files, 266
configuring
Auto-Routing, 262
debug logging, 266
Listener, 264
listening port, 262
overview, 261
passwords, 262
properties, 262
security, 264
Traps, 261–264
usernames, 262
version number, 266
filtering events, 259–260, 263
overview, 259–260
passive monitoring, 260
Traps, 261–264
SNMP channels, 246–249
SOAP Fault, 279–280, 296
SOAP Request/Reply, 74–75
Solutions. *See also* Enterprise Applications; Projects.
deploying. *See* Deployment.
development stages, 20–23
factoring. *See* Distributing components.
logical break points, 9
Source control. *See* Deployment, management.
Splitter pattern, 78–79, 171–172
See also Aggregator.
See also Envelope Wrapper.
See also Message Sequence.
Splitting messages, 78–79
SRE (Schema Runtime Environment), 151
SSH (Secure Shell), 434
SSL (Secure Sockets Layer). *See also* HTTP eWay.
Cryptographic Handshake, 91
encryption, 417
history of, 415–416
JCE (Java Cryptography Extension), 428–429
mutual authentication, 418
protocol description, 416
public keys, 417–419
server-side authentication, 418
strong cipher suites, 428–429
Web Services
hostname verification, 432
mutual authentication channel, 431–432
overview, 430
server-side authentication channel, 430–431
X.509 Certificate, 416–419
State, obtaining, 269–271
Static selectors, 107–109
stcmsctrutil utility, 240
Storage area network (SAN), 401–407
StreamMessage format, 132–133
Strong cipher suites, 428–429
Sun products. *See specific products.*
Sun SeeBeyond. *See specific products.*
Sync to Disk property, 145
System management. *See* Monitoring and management.

T

Tables Runtime Environment (TRE), 151
Tagging, 43
Task Scheduler, 62
Temporary JMS Destinations, 98–99
Temporary queues, 70
Test Message pattern, 327–328
TextMessage format, 132
Third-party version control systems, 46–49
Threads graph, 304–305
Throttling, 115–116

Timed items correlation, 363–364
Timed Items Correlation pattern, 363–364
timeToLive property, 83–85
TLS (Transport Layer Security), 415
Topics. *See* JMS topics.
Transacted mode, 104
Transacted setting, 103–104
Transactional Client, 210–211
Transactionality. *See also* JMS transactions; XA transactionality.
  IQ Manager
    commit, 100
    definition, 99
    inspecting messages, 100–104
    purpose of, 100
    scope, 100
    Transacted mode behavior, 104
    Transacted setting, 103–104
    Transactionality property, 103
    uncommitted messages, 100
  Transactional Client, 210–211
Transactionality property, 103
Transparent replication, 119, 143
Traps, 261–264
TRE (Tables Runtime Environment), 151
Truststores, referencing, 99

U
UDDI Registry, 18, 397
Undeliverable messages. *See* Dead Letter Channel.
UTC (Universal Time Coordinated), 84

V
Variables
  description, 33–36
  listing, 35
  mapping, 35
  runtime values, 35
  type, 33
Version control. *See* Deployment, management.
Version history, 41
Version isolation, 42

W
Web proxy, 409
Web Services
  creating, 29
  providers, alerts for starting/stopping, 251
  Request/Reply pattern, 75–76
  service orchestration, 7–8
  SSL (Secure Sockets Layer)
    hostname verification, 432
    mutual authentication channel, 431–432
    overview, 430
    server-side authentication channel, 430–431
Wire protocols, 94–95
Wire Tap pattern, 319–321
WSDL definitions, 271–272, 280, 373–375

X
X.509 Certificate, 416–419
XA transactionality
  contention for resources, 131
  deadlocks, 131
  effects on eInsight business processes, 82
  eInsight Business Processes, 131
  on eInsight Business Processes, 131
  Guaranteed Delivery pattern, 146–147
  on long-running processes, 131
  serializing business processes, 81–82
XML (Extensible Markup Language)
  appropriate use of, 180–181
  enveloping within XML, 192–201