List of Patterns

Aggregator (268) How do we combine the results of individual but related messages so that they can be processed as a whole?

Canonical Data Model (355) How can you minimize dependencies when integrating applications that use different data formats?

Channel Adapter (127) How can you connect an application to the messaging system so that it can send and receive messages?

Channel Purger (572) How can you keep leftover messages on a channel from disturbing tests or running systems?

Claim Check (346) How can we reduce the data volume of message sent across the system without sacrificing information content?

Command Message (145) How can messaging be used to invoke a procedure in another application?

Competing Consumers (502) How can a messaging client process multiple messages concurrently?

Composed Message Processor (294) How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?

Content Enricher (336) How do we communicate with another system if the message origination does not have all the required data items available?

Content Filter (342) How do you simplify dealing with a large message when you are interested only in a few data items?

Content-Based Router (230) How do we handle a situation in which the implementation of a single logical function is spread across multiple physical systems?

Control Bus (540) How can we effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area?

Correlation Identifier (163) How does a requestor that has received a reply know which request this is the reply for?

Datatype Channel (111) How can the application send a data item such that the receiver will know how to process it?

Dead Letter Channel (119) What will the messaging system do with a message it cannot deliver?

Detour (545) How can you route a message through intermediate steps to perform validation, testing, or debugging functions?

Document Message (147) How can messaging be used to transfer data between applications?

Durable Subscriber (522) How can a subscriber avoid missing messages while it’s not listening for them?

Dynamic Router (243) How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?

Envelope Wrapper (330) How can existing systems participate in a messaging exchange that places specific requirements, such as message header fields or encryption, on the message format?

Event Message (151) How can messaging be used to transmit events from one application to another?

Event-Driven Consumer (498) How can an application automatically consume messages as they become available?

File Transfer (43) How can I integrate multiple applications so that they work together and can exchange information?

Format Indicator (180) How can a message’s data format be designed to allow for possible future changes?

Guaranteed Delivery (122) How can the sender make sure that a message will be delivered even if the messaging system fails?

Idempotent Receiver (528) How can a message receiver deal with duplicate messages?

Invalid Message Channel (115) How can a messaging receiver gracefully handle receiving a message that makes no sense?

Message Broker (322) How can you decouple the destination of a message from the sender and maintain central control over the flow of messages?

Message Bus (137) What architecture enables separate applications to work together but in a decoupled fashion such that applications can be easily added or removed without affecting the others?

Message Channel (60) How does one application communicate with another using messaging?

Message Dispatcher (508) How can multiple consumers on a single channel coordinate their message processing?

Message Endpoint (95) How does an application connect to a messaging channel to send and receive Messages?

Message Expiration (176) How can a sender indicate when a message should be considered stale and thus shouldn’t be processed?
Message Filter (237)  How can a component avoid receiving uninteresting messages?
Message History (551)  How can we effectively analyze and debug the flow of messages in a loosely coupled system?
Message Router (78)  How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?
Message Sequence (170)  How can messaging transmit an arbitrarily large amount of data?
Message Store (555)  How can we report against message information without disturbing the loosely coupled and transient nature of a messaging system?
Message Translator (85)  How can systems using different data formats communicate with each other using messaging?
Message (66)  How can two applications connected by a message channel exchange a piece of information?
Messaging Bridge (133)  How can multiple messaging systems be connected so that messages available on one are also available on the others?
Messaging Gateway (468)  How do you encapsulate access to the messaging system from the rest of the application?
Messaging Mapper (477)  How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
Messaging (53)  How can I integrate multiple applications so that they work together and can exchange information?
Normalizer (352)  How do you process messages that are semantically equivalent but arrive in a different format?
Pipes and Filters (70)  How can we perform complex processing on a message while maintaining independence and flexibility?
Point-to-Point Channel (103)  How can the caller be sure that exactly one receiver will receive the document or perform the call?
Polling Consumer (494)  How can an application consume a message when the application is ready?
Process Manager (312)  How do we route a message through multiple processing steps when the required steps may not be known at design time and may not be sequential?
Publish-Subscribe Channel (106)  How can the sender broadcast an event to all interested receivers?
Recipient List (249)  How do we route a message to a dynamic list of recipients?
Remote Procedure Invocation (50)  How can I integrate multiple applications so that they work together and can exchange information?
Request-Reply (154)  When an application sends a message, how can it get a response from the receiver?
Resequencer (283)  How can we get a stream of related but out-of-sequence messages back into the correct order?
Return Address (159)  How does a replier know where to send the reply?
Routing Slip (301)  How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design time and may vary for each message?
Scatter-Gather (297)  How can you maintain the overall message flow when a message must be sent to multiple recipients, each of which may send a reply?
Selective Consumer (515)  How can a message consumer select which messages it wishes to receive?
Service Activator (532)  How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?
Shared Database (47)  How can I integrate multiple applications so that they work together and can exchange information?
Smart Proxy (558)  How can you track messages on a service that publishes reply messages to the Return Address specified by the requestor?
Splitter (259)  How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
Test Message (569)  What happens if a component is actively processing messages but garbles outgoing messages due to an internal fault?
Transactional Client (484)  How can a client control its transactions with the messaging system?
Wire Tap (547)  How do you inspect messages that travel on a Point-to-Point Channel?
Enterprise
Integration Patterns
The Addison-Wesley Signature Series provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: Great books come from great authors. Books in the series are personally chosen by expert advisors, world-class authors in their own right. These experts are proud to put their signatures on the covers, and their signatures ensure that these thought leaders have worked closely with authors to define topic coverage, book scope, critical content, and overall uniqueness. The expert signatures also symbolize a promise to our readers: You are reading a future classic.

Visit informit.com/awss for a complete list of available products.
Enterprise Integration Patterns

Designing, Building, and Deploying Messaging Solutions

Gregor Hohpe
Bobby Woolf

With Contributions by
Kyle Brown
Conrad F. D’Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon

Addison-Wesley
Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City
To my family and all my friends who still remember me after I emerged from book “crunch mode”

—Gregor

To Sharon, my new wife

—Bobby
Contents

Foreword by John Crupi ......................................................... xv
Foreword by Martin Fowler ................................................... xvii
Preface .............................................................................. xix
Acknowledgments ................................................................. xxv
Introduction ........................................................................ xxix

Chapter 1: Solving Integration Problems Using Patterns ............... 1
  The Need for Integration ......................................................... 1
  Integration Challenges .......................................................... 2
  How Integration Patterns Can Help .......................................... 4
  The Wide World of Integration ................................................ 5
  Loose Coupling ..................................................................... 9
  One-Minute EAI ................................................................. 11
  A Loosely Coupled Integration Solution ................................... 15
  Widgets & Gadgets ’R Us: An Example ................................... 17
    Internal Systems ................................................................. 18
    Taking Orders ...................................................................... 18
    Processing Orders ............................................................. 20
    Checking Status ............................................................... 26
    Change Address ............................................................... 30
    New Catalog ...................................................................... 32
    Announcements .................................................................. 33
    Testing and Monitoring ..................................................... 34
    Summary ........................................................................... 37

Chapter 2: Integration Styles .................................................. 39
  Introduction ........................................................................... 39
  File Transfer (by Martin Fowler) ............................................. 43
Contents

Shared Database (by Martin Fowler) .............................................. 47
Remote Procedure Invocation (by Martin Fowler) ......................... 50
Messaging ................................................................. 53

Chapter 3: Messaging Systems ..................................................... 57
  Introduction ......................................................... 57
  Message Channel .................................................. 60
  Message .......................................................... 66
  Pipes and Filters ................................................... 70
  Message Router .................................................... 78
  Message Translator ............................................... 85
  Message Endpoint ................................................ 95

Chapter 4: Messaging Channels ................................................... 99
  Introduction ......................................................... 99
  Point-to-Point Channel ............................................ 103
  Publish-Subscribe Channel ...................................... 106
  Datatype Channel ................................................ 111
  Invalid Message Channel ....................................... 115
  Dead Letter Channel ............................................. 119
  Guaranteed Delivery ............................................. 122
  Channel Adapter .................................................. 127
  Messaging Bridge .................................................. 133
  Message Bus ........................................................ 137

Chapter 5: Message Construction ............................................... 143
  Introduction ......................................................... 143
  Command Message ................................................ 145
  Document Message ............................................... 147
  Event Message ..................................................... 151
  Request-Reply ..................................................... 154
  Return Address .................................................... 159
  Correlation Identifier ............................................ 163
  Message Sequence ................................................ 170
  Message Expiration ............................................... 176
  Format Indicator ................................................... 180
Chapter 6: Interlude: Simple Messaging ................................. 183
  Introduction .......................................................183
  Request-Reply Example ........................................183
  Publish-Subscribe Example .................................185
JMS Request-Reply Example .........................................187
  Request-Reply Example ........................................187
  Request-Reply Code ...........................................189
  Invalid Message Example ..................................196
  Conclusions .....................................................197
.NET Request-Reply Example .........................................198
  Request-Reply Example ........................................198
  Request-Reply Code ...........................................200
  Invalid Message Example ..................................205
  Conclusions .....................................................206
JMS Publish-Subscribe Example .......................................207
  The Observer Pattern .....................................207
  Distributed Observer ....................................208
  Publish-Subscribe .........................................209
  Comparisons ...................................................212
  Push and Pull Models ...................................213
  Channel Design ..............................................219
  Conclusions .....................................................222
Chapter 7: Message Routing ..............................................225
  Introduction .....................................................225
  Content-Based Router .....................................230
  Message Filter ................................................237
  Dynamic Router .............................................243
  Recipient List ................................................249
  Splitter .........................................................259
  Aggregator .....................................................268
  Resequencer ...................................................283
  Composed Message Processor ..........................294
  Scatter-Gather ...............................................297
  Routing Slip ...................................................301
CONTENTS

Process Manager ............................................. 312
Message Broker ............................................. 322

Chapter 8: Message Transformation .......................... 327
Introduction ................................................... 327
Envelope Wrapper ........................................... 330
Content Enricher ............................................. 336
Content Filter .................................................. 342
Claim Check .................................................... 346
Normalizer ...................................................... 352
Canonical Data Model ......................................... 355

Chapter 9: Interlude: Composed Messaging .................... 361
Loan Broker Example .......................................... 361
Obtaining a Loan Quote ...................................... 361
Designing the Message Flow .................................. 362
Sequencing: Synchronous versus Asynchronous .............. 364
Addressing: Distribution versus Auction ...................... 366
Aggregating Strategies: Multiple Channels versus
Single Channel .................................................. 368
Managing Concurrency ....................................... 368
Three Implementations ........................................ 369
Synchronous Implementation Using Web Services
(by Conrad F. D’Cruz) ............................................ 371
Solution Architecture .......................................... 371
Web Services Design Considerations ......................... 372
Apache Axis ..................................................... 376
Service Discovery .............................................. 379
The Loan Broker Application .................................. 379
Components of the Loan Broker Application .................. 381
Client Application .............................................. 396
Running the Solution .......................................... 397
Performance Limitations ...................................... 399
Limitations of This Example .................................. 400
Summary ......................................................... 400
Asynchronous Implementation with MSMQ .................... 401
Loan Broker Ecosystem ........................................ 401
Laying the Groundwork: A Messaging Gateway ............... 402
Contents

Base Classes for Common Functionality .........................405
Designing the Bank .................................................410
Designing the Credit Bureau ......................................412
Designing the Loan Broker .......................................413
Refactoring the Loan Broker ....................................431
Putting it All Together ...........................................435
Improving Performance ...........................................435
A Few Words on Testing .........................................440
Limitations of This Example .....................................443
Summary ............................................................444

Asynchronous Implementation with TIBCO ActiveEnterprise
(by Michael J. Rettig) .............................................445

Solution Architecture ............................................445
The Implementation Toolset .......................................448
The Interfaces .......................................................451
Implementing the Synchronous Services .......................452
The Loan Broker Process .........................................455
Managing Concurrent Auctions ................................459
Execution ..........................................................460
Conclusions .........................................................462

Chapter 10: Messaging Endpoints .................................463
Introduction .........................................................463
Messaging Gateway ...............................................468
Messaging Mapper ................................................477
Transactional Client ..............................................484
Polling Consumer ................................................494
Event-Driven Consumer .........................................498
Competing Consumers ...........................................502
Message Dispatcher ..............................................508
Selective Consumer ..............................................515
Durable Subscriber ...............................................522
Idempotent Receiver .............................................528
Service Activator ................................................532

Chapter 11: System Management .................................537
Introduction .........................................................537
Control Bus ........................................................540
## CONTENTS

Detour .................................................. 545
Wire Tap ............................................... 547
Message History ............................... 551
Message Store ................................. 555
Smart Proxy ..................................... 558
Test Message .................................. 569
Channel Purger ................................ 572

Chapter 12: Interlude: System Management Example ............................. 577

- Loan Broker System Management .................. 577
- Instrumenting the Loan Broker ..................... 578
- Management Console ............................ 579
- Loan Broker Quality of Service .................... 579
- Verify the Credit Bureau Operation ................. 587
- Credit Bureau Failover ............................ 592
- Enhancing the Management Console ............... 595
- Limitations of This Example ....................... 602

Chapter 13: Integration Patterns in Practice .................................. 603

- Case Study: Bond Pricing System *(by Jonathan Simon)* .............. 603
  - Building a System ................................ 603
  - Architecture with Patterns ....................... 604
  - Structuring Channels ............................ 610
  - Selecting a Message Channel ...................... 614
  - Problem Solving with Patterns .................... 618
  - Flashing Market Data Updates ..................... 618
  - Major Production Crash .......................... 620
  - Summary ........................................ 623

Chapter 14: Concluding Remarks .................................................. 625

- Emerging Standards and Futures
  in Enterprise Integration *(by Sean Neville)* ................. 625
- The Relationship between Standards and Design Patterns .......... 626
- Survey of Standards Processes and Organizations .................. 627
- Business Process Components and Intra-Web Service Messaging .................. 629
- ebXML and the Electronic Business Messaging Service (ebMS) ........... 631
Contents

Business Process Execution Language for Web Services (BEPL4WS) ...................... 634
Web Service Choreography Interface (WSCI) ......................... 636
Java Business Process Component Standards ....................... 637
WS-* ...................................................... 639
Conclusions .............................................. 647

Bibliography ........................................... 649

Index .................................................. 659
Foreword

by John Crupi

What do you do when a new technology arrives? You learn the technology. This is exactly what I did. I studied J2EE (being from Sun Microsystems, it seemed to be the logical choice). Specifically, I focused on the EJB technology by reading the specifications (since there were no books yet). Learning the technology, however, is just the first step—the real goal is to learn how to effectively apply the technology. The nice thing about platform technologies is that they constrain you to performing certain tasks. But, as far as the technology is concerned, you can do whatever you want and quite often get into trouble if you don’t do things appropriately.

One thing I’ve seen in the past 15 years is that there seem to be two areas that software developers obsess over: programming and designing—or more specifically, programming and designing effectively. There are great books out there that tell you the most efficient way to program certain things in Java and C#, but far fewer tell you how to design effectively. That’s where this book comes in. When Deepak Alur, Dan Malks, and I wrote Core J2EE Patterns, we wanted to help J2EE developers “design” better code. The best decision we made was to use patterns as the artifact of choice. As James Baty, a Sun Distinguished Engineer, puts it, “Patterns seem to be the sweet spot of design.” I couldn’t agree more, and luckily for us, Gregor and Bobby feel the same way.

This book focuses on a hot and growing topic: integration using messaging. Not only is messaging key to integration, but it will most likely be the predominant focus in Web services for years to come. There is so much noise today in the Web services world, it’s a delicate and complex endeavor just to identify the specifications and technologies to focus on. The goal remains the same, however—software helps you solve a problem. Just as in the early days of J2EE and .NET, there is not a lot of design help out there yet for Web services. Many people say
Foreword

Web services is just a new and open way to solve our existing integration problems—and I agree. But, that doesn’t mean we know how to design Web services. And that brings us to the gem of this book. I believe this book has many of the patterns we need to design Web services and other integration systems. Because the Web service specifications are still battling it out, it wouldn’t have made sense for Bobby and Gregor to provide examples of many of the Web service specifications. But, that’s okay. The real payoff will result when the specifications become standards and we use the patterns in this book to design for those solutions that are realized by these standards. Then maybe we can realize our next integration goal of designing for service-oriented architectures.

Read this book and keep it by your side. It will enhance your software career to no end.

John Crupi
Bethesda, MD
August 2003
Foreword

by Martin Fowler

While I was working on my book Patterns of Enterprise Application Architecture, I was lucky to get some in-depth review from Kyle Brown and Rachel Reinitz at some informal workshops at Kyle’s office in Raleigh-Durham. During these sessions, we realized that a big gap in my work was asynchronous messaging systems.

There are many gaps in my book, and I never intended it to be a complete collection of patterns for enterprise development. But the gap on asynchronous messaging is particularly important because we believe that asynchronous messaging will play an increasingly important role in enterprise software development, particularly in integration. Integration is important because applications cannot live isolated from each other. We need techniques that allow us to take applications that were never designed to interoperate and break down the stovepipes so we can gain a greater benefit than the individual applications can offer us.

Various technologies have been around that promise to solve the integration puzzle. We all concluded that messaging is the technology that carries the greatest promise. The challenge we faced was to convey how to do messaging effectively. The biggest challenge in this is that messages are by their nature asynchronous, and there are significant differences in the design approaches that you use in an asynchronous world.

I didn’t have space, energy, or frankly the knowledge to cover this topic properly in Patterns of Enterprise Application Architecture. But we came up with a better solution to this gap: find someone else who could. We hunted down Gregor and Bobby, and they took up the challenge. The result is the book you’re about to read.
I’m delighted with the job that they have done. If you’ve already worked with messaging systems, this book will systematize much of the knowledge that you and others have already learned the hard way. If you are about to work with messaging systems, this book will provide a foundation that will be invaluable no matter which messaging technology you have to work with.

Martin Fowler
Melrose, MA
August 2003
Preface

This is a book about enterprise integration using messaging. It does not document any particular technology or product. Rather, it is designed for developers and integrators using a variety of messaging products and technologies, such as

- Message-oriented middleware (MOM) and EAI suites offered by vendors such as IBM (WebSphere MQ Family), Microsoft (BizTalk), TIBCO, WebMethods, SeeBeyond, Vitria, and others.
- Java Message Service (JMS) implementations incorporated into commercial and open source J2EE application servers as well as standalone products.
- Microsoft’s Message Queuing (MSMQ), accessible through several APIs, including the System.Messaging libraries in Microsoft .NET.
- Emerging Web services standards that support asynchronous Web services (for example, WS-ReliableMessaging) and the associated APIs such as Sun Microsystems’ Java API for XML Messaging (JAXM) or Microsoft’s Web Services Extensions (WSE).

Enterprise integration goes beyond creating a single application with a distributed n-tier architecture, which enables a single application to be distributed across several computers. Whereas one tier in a distributed application cannot run by itself, integrated applications are independent programs that can each run by themselves, yet that function by coordinating with each other in a loosely coupled way. Messaging enables multiple applications to exchange data or commands across the network using a “send and forget” approach. This allows the caller to send the information and immediately go on to other work while the information is transmitted by the messaging system. Optionally, the caller can later be notified of the result through a callback. Asynchronous calls and callbacks can make a design more complex than a synchronous approach, but an asynchronous call can be retried until it succeeds, which makes the communica-
tion much more reliable. Asynchronous messaging also enables several other advantages, such as throttling of requests and load balancing.

**Who Should Read This Book**

This book is designed to help application developers and system integrators connect applications using message-oriented integration tools:

- **Application architects and developers** who design and build complex enterprise applications that need to integrate with other applications. We assume that you’re developing your applications using a modern enterprise application platform such as the Java 2 Platform, Enterprise Edition (J2EE), or the Microsoft .NET Framework. This book will help you connect the application to a messaging layer and exchange information with other applications. This book focuses on the integration of applications, not on building applications; for that, we refer you to *Patterns of Enterprise Application Architecture* by Martin Fowler.

- **Integration architects and developers** who design and build integration solutions connecting packaged or custom applications. Most readers in this group will have experience with one of the many commercial integration tools like IBM WebSphere MQ, TIBCO, WebMethods, SeeBeyond, or Vitria, which incorporate many of the patterns presented in this book. This book helps you understand the underlying concepts and make confident design decisions using a vendor-independent vocabulary.

- **Enterprise architects** who have to maintain the “big picture” view of the software and hardware assets in an enterprise. This book presents a consistent vocabulary and graphical notation to describe large-scale integration solutions that may span many technologies or point solutions. This language is also a key enabler for efficient communication between the enterprise architect and the integration and application architects and developers.

**What You Will Learn**

This book does not attempt to make a business case for enterprise application integration; the focus is on how to make it work. You will learn how to integrate enterprise applications by understanding the following:
• The advantages and limitations of asynchronous messaging as compared to other integration techniques.

• How to determine the message channels your applications will need, how to control whether multiple consumers can receive the same message, and how to handle invalid messages.

• When to send a message, what it should contain, and how to use special message properties.

• How to route a message to its ultimate destination even when the sender does not know where that is.

• How to convert messages when the sender and receiver do not agree on a common format.

• How to design the code that connects an application to the messaging system.

• How to manage and monitor a messaging system once it’s in use as part of the enterprise.

What This Book Does Not Cover

We believe that any book sporting the word “enterprise” in the title is likely to fall into one of three categories. First, the book might attempt to cover the whole breadth of the subject matter but is forced to stop short of detailed guidance on how to implement actual solutions. Second, the book might provide specific hands-on guidance on the development of actual solutions but is forced to constrain the scope of the subject area it addresses. Third, the book might attempt to do both but is likely never to be finished or else to be published so late as to be irrelevant. We opted for the second choice and hopefully created a book that helps people create better integration solutions even though we had to limit the scope of the book. Topics that we would have loved to discuss but had to exclude in order not to fall into the category-three trap include security, complex data mapping, workflow, rule engines, scalability and robustness, and distributed transaction processing (XA, Tuxedo, and the like). We chose asynchronous messaging as the emphasis for this book because it is full of interesting design issues and trade-offs, and provides a clean abstraction from the many implementations provided by various integration vendors.

This book is also not a tutorial on a specific messaging or middleware technology. To highlight the wide applicability of the concepts presented in this
book, we included examples based on a number of different technologies, such as JMS, MSMQ, TIBCO, BizTalk, and XSL. However, we focus on the design decisions and trade-offs as opposed to the specifics of the tool. If you are interested in learning more about any of these specific technologies, please refer to one of the books referenced in the bibliography or to one of the many online resources.

How This Book Is Organized

As the title suggests, the majority of this book consists of a collection of patterns. Patterns are a proven way to capture experts’ knowledge in fields where there are no simple “one size fits all” answers, such as application architecture, object-oriented design, or integration solutions based on asynchronous messaging architectures.

Each pattern poses a specific design problem, discusses the considerations surrounding the problem, and presents an elegant solution that balances the various forces or drivers. In most cases, the solution is not the first approach that comes to mind, but one that has evolved through actual use over time. As a result, each pattern incorporates the experience base that senior integration developers and architects have gained by repeatedly building solutions and learning from their mistakes. This implies that we did not “invent” the patterns in this book; patterns are not invented, but rather discovered and observed from actual practice in the field.

Because patterns are harvested from practitioners’ actual use, chances are that if you have been working with enterprise integration tools and asynchronous messaging architectures for some time, many of the patterns in this book will seem familiar to you. Yet, even if you already recognize most of these patterns, there is still value in reviewing this book. This book should validate your hard-earned understanding of how to use messaging while documenting details of the solutions and relationships between them of which you might not have been aware. It also gives you a consolidated reference to help you pass your knowledge effectively to less-experienced colleagues. Finally, the pattern names give you a common vocabulary to efficiently discuss integration design alternatives with your peers.

The patterns in this book apply to a variety of programming languages and platforms. This means that a pattern is not a cut-and-paste snippet of code, but you have to realize a pattern to your specific environment. To make this translation easier, we added a variety of examples that show different ways of imple-
menting patterns using popular technologies such as JMS, MSMQ, TIBCO, BizTalk, XSL, and others. We also included a few larger examples to demonstrate how multiple patterns play together to form a cohesive solution.

Integrating multiple applications using an asynchronous messaging architecture is a challenging and interesting field. We hope you enjoy reading this book as much as we did writing it.

About the Cover Picture

The common theme for books in the Martin Fowler Signature Series is a picture of a bridge. In some sense we lucked out, because what theme would make a better match for a book on integration? For thousands of years, bridges have helped connect people from different shores, mountains, and sides of the road.

We selected a picture of the Taiko-bashi Bridge at the Sumiyoshi-taisha Shrine in Osaka, Japan, for its simple elegance and beauty. As a Shinto shrine dedicated to the guardian deity for sailors, it was originally erected next to the water. Interestingly, land reclamation has pushed the water away so that the shrine today stands almost three miles inland. Some three million people visit this shrine at the beginning of a new year.

Gregor Hohpe
San Francisco, California

Bobby Woolf
Raleigh, North Carolina

September 2003
www.enterpriseintegrationpatterns.com
The Pioneer Plaque by Dr. Carl Sagan
A message to extraterrestrial life forms.
Acknowledgments

Like most books, *Enterprise Integration Patterns* has been a long time in the making. The idea of writing about message-based integration patterns dates back to the summer of 2001 when Martin Fowler was working on *Patterns of Enterprise Application Architecture* (*P of EAA*). At that time, it struck Kyle Brown that while *P of EAA* talked a lot about how to create applications, it touches only briefly on how to integrate them. This idea was the starting point for a series of meetings between Martin and Kyle that also included Rachel Reinitz, John Crupi, and Mark Weitzel. Bobby joined these discussions in the fall of 2001, followed by Gregor in early 2002. The following summer the group submitted two papers for review at the Pattern Languages of Programs (PLoP) conference, one authored jointly by Bobby and Kyle and the other by Gregor. After the conference, Kyle and Martin refocused on their own book projects while Gregor and Bobby merged their papers to form the basis for the book. At the same time, the *www.enterpriseintegrationpatterns.com* site went live to allow integration architects and developers around the world to participate in the rapid evolution of the content. As they worked on the book, Gregor and Bobby invited contributors to participate in the creation of the book. About two years after Kyle’s original idea, the final manuscript arrived at the publisher.

This book is the result of a community effort involving a great number of people. Many colleagues and friends (many of whom we met through the book effort) provided ideas for examples, ensured the correctness of the technical content, and gave us much needed feedback and criticism. Their input has greatly influenced the final form and content of the book. It is a pleasure for us to acknowledge their contributions and express our appreciation for their efforts.

Kyle Brown and Martin Fowler deserve special mention for laying the foundation for this book. This book might have never been written were it not for Martin’s writing *P of EAA* and Kyle’s forming a group to discuss messaging patterns to complement Martin’s book.
ACKNOWLEDGMENTS

We were fortunate to have several contributors who authored significant portions of the book: Conrad F. D'Cruz, Sean Neville, Michael J. Rettig, and Jonathan Simon. Their chapters round out the book with additional perspectives on how the patterns work in practice.

Our writers’ workshop participants at the PLoP 2002 conference were the first people to provide substantial feedback on the material, helping to get us going in the right direction: Ali Arsanjani, Kyle Brown, John Crupi, Eric Evans, Martin Fowler, Brian Marick, Toby Sarver, Jonathan Simon, Bill Trudell, and Marek Vokac.

We would like to thank our team of reviewers who took the time to read through the draft material and provided us with invaluable feedback and suggestions:

Richard Helm
Luke Hohmann
Dragos Manolescu
David Rice
Russ Rufer and the Silicon Valley Patterns Group
Matthew Short

Special thanks go to Russ for workshopping the book draft in the Silicon Valley Patterns Group. We would like to thank the following members for their efforts: Robert Benson, Tracy Bialik, Jeffrey Blake, Azad Bolour, John Brewer, Bob Evans, Andy Farlie, Jeff Glaza, Phil Goodwin, Alan Harriman, Ken Hejmanowski, Deborah Kaddah, Rituraj Kirti, Jan Looney, Chris Lopez, Jerry Louis, Tao-hung Ma, Jeff Miller, Stilian Pandev, John Parello, Hema Pillay, Russ Rufer, Rich Smith, Carol Thistlethwaite, Debbie Utley, Walter Vannini, David Vydra, and Ted Young.

Our public e-mail discussion list allowed people who discovered the material on www.enterpriseintegrationpatterns.com to chime in and share their thoughts and ideas. Special honors go to Bill Trudell as the most active contributor to the mailing list. Other active posters included Venkateshwar Bommineni, Duncan Cragg, John Crupi, Fokko Degenaar, Shailesh Gosavi, Christian Hall, Ralph Johnson, Paul Julius, Orjan Lundberg, Dragos Manolescu, Rob Mee, Srikanth Narasimhan, Sean Neville, Rob Patton, Kirk Pepperdine, Matthew Pryor, Somik Raha, Michael Rettig, Frank Sauer, Jonathan Simon, Federico Spinazzi, Randy Stafford, Marek Vokac, Joe Walnes, and Mark Weitzel.
We thank Martin Fowler for hosting us in his signature series. Martin’s endorsement gave us confidence and the energy required to complete this work.

We thank John Crupi for writing the foreword for our book. He has observed the book’s formation from the beginning and has been a patient guide all along without ever losing his sense of humor.

Finally, we owe a great deal to the editing and production team at Addison-Wesley, led by our chief editor, Mike Hendrickson, and including our production coordinator, Amy Fleischer; our project manager, Kim Arney Mulcahy; our copy-editor, Carol J. Lallier; our proofreader, Rebecca Rider; our indexer, Sharon Hilgenberg; as well as Jacquelyn Doucette, John Fuller, and Bernard Gaffney.

We’ve likely missed some names and not given everyone the credit they deserve, and we apologize. But to everyone listed and not listed who helped make this book better, thank you for all your help. We hope you can be as proud of this book as we are.
This page intentionally left blank
Introduction

Interesting applications rarely live in isolation. Whether your sales application must interface with your inventory application, your procurement application must connect to an auction site, or your PDA's calendar must synchronize with the corporate calendar server, it seems that any application can be made better by integrating it with other applications.

All integration solutions have to deal with a few fundamental challenges:

- **Networks are unreliable.** Integration solutions have to transport data from one computer to another across networks. Compared to a process running on a single computer, distributed computing has to be prepared to deal with a much larger set of possible problems. Often, two systems to be integrated are separated by continents, and data between them has to travel through phone lines, LAN segments, routers, switches, public networks, and satellite links. Each step can cause delays or interruptions.

- **Networks are slow.** Sending data across a network is multiple orders of magnitude slower than making a local method call. Designing a widely distributed solution the same way you would approach a single application could have disastrous performance implications.

- **Any two applications are different.** Integration solutions need to transmit information between systems that use different programming languages, operating platforms, and data formats. An integration solution must be able to interface with all these different technologies.

- **Change is inevitable.** Applications change over time. An integration solution has to keep pace with changes in the applications it connects. Integration solutions can easily get caught in an avalanche effect of changes—if one system changes, all other systems may be affected. An integration solution needs to minimize the dependencies from one system to another by using *loose coupling* between applications.
INTRODUCTION

Over time, developers have overcome these challenges with four main approaches:

1. *File Transfer* (43)—One application writes a file that another later reads. The applications need to agree on the filename and location, the format of the file, the timing of when it will be written and read, and who will delete the file.

2. *Shared Database* (47)—Multiple applications share the same database schema, located in a single physical database. Because there is no duplicate data storage, no data has to be transferred from one application to the other.

3. *Remote Procedure Invocation* (50)—One application exposes some of its functionality so that it can be accessed remotely by other applications as a remote procedure. The communication occurs in real time and synchronously.

4. *Messaging* (53)—One application publishes a message to a common message channel. Other applications can read the message from the channel at a later time. The applications must agree on a channel as well as on the format of the message. The communication is asynchronous.

While all four approaches solve essentially the same problem, each style has its distinct advantages and disadvantages. In fact, applications may integrate using multiple styles such that each point of integration takes advantage of the style that suits it best.

**What Is Messaging?**

This book is about how to use messaging to integrate applications. A simple way to understand what messaging does is to consider the telephone system. A telephone call is a synchronous form of communication. I can communicate with the other party only if the other party is available at the time I place the call. Voice mail, on the other hand, allows asynchronous communication. With voice mail, when the receiver does not answer, the caller can leave a message; later, the receiver (at his convenience) can listen to the messages queued in his mailbox. Voice mail enables the caller to leave a message now so that the receiver can listen to it later, which is much easier than trying to get the caller and the receiver on the phone at the same time. Voice mail bundles (at least part
INTRODUCTION

of) a phone call into a message and queues it for later consumption; this is essentially how messaging works.

Messaging is a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery. Programs communicate by sending packets of data called messages to each other. Channels, also known as queues, are logical pathways that connect the programs and convey messages. A channel behaves like a collection or array of messages, but one that is magically shared across multiple computers and can be used concurrently by multiple applications. A sender or producer is a program that sends a message by writing the message to a channel. A receiver or consumer is a program that receives a message by reading (and deleting) it from a channel.

The message itself is simply some sort of data structure—such as a string, a byte array, a record, or an object. It can be interpreted simply as data, as the description of a command to be invoked on the receiver, or as the description of an event that occurred in the sender. A message actually contains two parts, a header and a body. The header contains meta-information about the message—who sent it, where it’s going, and so on; this information is used by the messaging system and is mostly ignored by the applications using the messages. The body contains the application data being transmitted and is usually ignored by the messaging system. In conversation, when an application developer who is using messaging talks about a message, she’s usually referring to the data in the body of the message.

Asynchronous messaging architectures are powerful but require us to rethink our development approach. As compared to the other three integration approaches, relatively few developers have had exposure to messaging and message systems. As a result, application developers in general are not as familiar with the idioms and peculiarities of this communications platform.

What Is a Messaging System?

Messaging capabilities are typically provided by a separate software system called a messaging system or message-oriented middleware (MOM). A messaging system manages messaging the way a database system manages data persistence. Just as an administrator must populate the database with the schema for an application’s data, an administrator must configure the messaging system with the channels that define the paths of communication between the applications. The messaging system then coordinates and manages the sending and receiving of messages. The primary purpose of a database system is to make
sure each data record is safely persisted, and likewise the main task of a messaging system is to move messages from the sender’s computer to the receiver’s computer in a reliable fashion.

A messaging system is needed to move messages from one computer to another because computers and the networks that connect them are inherently unreliable. Just because one application is ready to send data does not mean that the other application is ready to receive it. Even if both applications are ready, the network may not be working or may fail to transmit the data properly. A messaging system overcomes these limitations by repeatedly trying to transmit the message until it succeeds. Under ideal circumstances, the message is transmitted successfully on the first try, but circumstances are often not ideal.

In essence, a message is transmitted in five steps:

1. **Create**—The sender creates the message and populates it with data.
2. **Send**—The sender adds the message to a channel.
3. **Deliver**—The messaging system moves the message from the sender’s computer to the receiver’s computer, making it available to the receiver.
4. **Receive**—The receiver reads the message from the channel.
5. **Process**—The receiver extracts the data from the message.

The following figure illustrates these five transmission steps, which computer performs each, and which steps involve the messaging system:
This figure also illustrates two important messaging concepts:

1. **Send and forget**—In step 2, the sending application sends the message to the message channel. Once that send is complete, the sender can go on to other work while the messaging system transmits the message in the background. The sender can be confident that the receiver will eventually receive the message and does not have to wait until that happens.

2. **Store and forward**—In step 2, when the sending application sends the message to the message channel, the messaging system stores the message on the sender’s computer, either in memory or on disk. In step 3, the messaging system delivers the message by forwarding it from the sender’s computer to the receiver’s computer, and then stores the message once again on the receiver’s computer. This store-and-forward process may be repeated many times as the message is moved from one computer to another until it reaches the receiver’s computer.

The create, send, receive, and process steps may seem like unnecessary overhead. Why not simply deliver the data to the receiver? By wrapping the data as a message and storing it in the messaging system, the applications delegate to the messaging system the responsibility of delivering the data. Because the data is wrapped as an atomic message, delivery can be retried until it succeeds, and the receiver can be assured of reliably receiving exactly one copy of the data.

**Why Use Messaging?**

Now that we know what messaging is, we should ask, Why use messaging? As with any sophisticated solution, there is no one simple answer. The quick answer is that messaging is more immediate than *File Transfer* (43), better encapsulated than *Shared Database* (47), and more reliable than *Remote Procedure Invocation* (50). However, that’s just the beginning of the advantages that can be gained using messaging.

Specific benefits of messaging include:

- **Remote Communication.** Messaging enables separate applications to communicate and transfer data. Two objects that reside in the same process can simply share the same data in memory. Sending data to another computer is a lot more complicated and requires data to be copied from one computer to another. This means that objects have to be “serializable”—that is, they
can be converted into a simple byte stream that can be sent across the network. Messaging takes care of this conversion so that the applications do not have to worry about it.

• **Platform/Language Integration.** When connecting multiple computer systems via remote communication, these systems likely use different languages, technologies, and platforms, perhaps because they were developed over time by independent teams. Integrating such divergent applications can require a neutral zone of middleware to negotiate between the applications, often using the lowest common denominator—such as flat data files with obscure formats. In these circumstances, a messaging system can be a universal translator between the applications that works with each one’s language and platform on its own terms yet allows them to all to communicate through a common messaging paradigm. This universal connectivity is the heart of the *Message Bus* (137) pattern.

• **Asynchronous Communication.** Messaging enables a send-and-forget approach to communication. The sender does not have to wait for the receiver to receive and process the message; it does not even have to wait for the messaging system to deliver the message. The sender only needs to wait for the message to be sent, that is, for the message to be successfully stored in the channel by the messaging system. Once the message is stored, the sender is free to perform other work while the message is transmitted in the background.

• **Variable Timing.** With synchronous communication, the caller must wait for the receiver to finish processing the call before the caller can receive the result and continue. In this way, the caller can make calls only as fast as the receiver can perform them. Asynchronous communication allows the sender to submit requests to the receiver at its own pace and the receiver to consume the requests at its own different pace. This allows both applications to run at maximum throughput and not waste time waiting on each other (at least until the receiver runs out of messages to process).

• **Throttling.** A problem with remote procedure calls (RPCs) is that too many of them on a single receiver at the same time can overload the receiver. This can cause performance degradation and even cause the receiver to crash. Because the messaging system queues up requests until the receiver is ready to process them, the receiver can control the rate at which it consumes requests so as not to become overloaded by too many simultaneous requests. The callers are unaffected by this throttling because the communication is asynchronous, so the callers are not blocked waiting on the receiver.
• **Reliable Communication.** Messaging provides reliable delivery that an RPC cannot. The reason messaging is more reliable than RPC is that messaging uses a *store-and-forward* approach to transmitting messages. The data is packaged as messages, which are atomic, independent units. When the sender sends a message, the messaging system stores the message. It then delivers the message by forwarding it to the receiver’s computer, where it is stored again. Storing the message on the sender’s computer and the receiver’s computer is assumed to be reliable. (To make it even more reliable, the messages can be stored to disk instead of memory; see *Guaranteed Delivery* [122].) What is unreliable is forwarding (moving) the message from the sender’s computer to the receiver’s computer, because the receiver or the network may not be running properly. The messaging system overcomes this by resending the message until it succeeds. This automatic retry enables the messaging system to overcome problems with the network so that the sender and receiver don’t have to worry about these details.

• **Disconnected Operation.** Some applications are specifically designed to run disconnected from the network, yet to synchronize with servers when a network connection is available. Such applications are deployed on platforms like laptop computers and PDAs. Messaging is ideal for enabling these applications to synchronize—data to be synchronized can be queued as it is created, waiting until the application reconnects to the network.

• **Mediation.** The messaging system acts as a mediator—as in the Mediator pattern [GoF]—between all of the programs that can send and receive messages. An application can use it as a directory of other applications or services available to integrate with. If an application becomes disconnected from the others, it need only reconnect to the messaging system, not to all of the other messaging applications. The messaging system can employ redundant resources to provide high availability, balance load, reroute around failed network connections, and tune performance and quality of service.

• **Thread Management.** Asynchronous communication means that one application does not have to block while waiting for another application to perform a task, unless it wants to. Rather than blocking to wait for a reply, the caller can use a callback that will alert the caller when the reply arrives. (See the Request-Reply [154] pattern.) A large number of blocked threads or threads blocked for a long time can leave the application with too few available threads to perform real work. Also, if an application with a dynamic number of blocked threads crashes, reestablishing those threads will be difficult when the application restarts and recovers its former state. With callbacks, the only threads that block are a small,
known number of listeners waiting for replies. This leaves most threads available for other work and defines a known number of listener threads that can easily be reestablished after a crash.

So, there are a number of different reasons an application or enterprise may benefit from messaging. Some of these are technical details that application developers relate most readily to, whereas others are strategic decisions that resonate best with enterprise architects. Which of these reasons is most important depends on the current requirements of your particular applications. They’re all good reasons to use messaging, so take advantage of whichever reasons provide the most benefit to you.

Challenges of Asynchronous Messaging

Asynchronous messaging is not the panacea of integration. It resolves many of the challenges of integrating disparate systems in an elegant way, but it also introduces new challenges. Some of these challenges are inherent in the asynchronous model, while other challenges vary with the specific implementation of a messaging system.

• Complex programming model. Asynchronous messaging requires developers to work with an event-driven programming model. Application logic can no longer be coded in a single method that invokes other methods, but instead the logic is now split up into a number of event handlers that respond to incoming messages. Such a system is more complex and harder to develop and debug. For example, the equivalent of a simple method call can require a request message and a request channel, a reply message and a reply channel, a correlation identifier and an invalid message queue (as described in Request-Reply [154]).

• Sequence issues. Message channels guarantee message delivery, but they do not guarantee when the message will be delivered. This can cause messages that are sent in sequence to get out of sequence. In situations where messages depend on each other, special care has to be taken to reestablish the message sequence (see Resequencer [283]).

• Synchronous scenarios. Not all applications can operate in a send-and-forget mode. If a user is looking for airline tickets, he or she is going to want to see the ticket price right away, not after some undetermined time.
Therefore, many messaging systems need to bridge the gap between synchronous and asynchronous solutions.

- **Performance.** Messaging systems do add some overhead to communication. It takes effort to package application data into a message and send it, and to receive a message and process it. If you have to transport a huge chunk of data, dividing it into a gazillion small pieces may not be a smart idea. For example, if an integration solution needs to synchronize information between two existing systems, the first step is usually to replicate all relevant information from one system to the other. For such a bulk data replication step, ETL (extract, transform, and load) tools are much more efficient than messaging. Messaging is best suited to keeping the systems in sync after the initial data replication.

- **Limited platform support.** Many proprietary messaging systems are not available on all platforms. Often, transferring a file via FTP is the only integration option because the target platform may not support a messaging system.

- **Vendor lock-in.** Many messaging system implementations rely on proprietary protocols. Even common messaging specifications such as JMS do not control the physical implementation of the solution. As a result, different messaging systems usually do not connect to one another. This can leave you with a whole new integration challenge: integrating multiple integration solutions! (See the *Messaging Bridge* [133] pattern.)

In summary, asynchronous messaging does not solve all problems, and it can even create new ones. Keep these consequences in mind when deciding which problems to solve using messaging.

---

**Thinking Asynchronously**

Messaging is an asynchronous technology, which enables delivery to be retried until it succeeds. In contrast, most applications use synchronous function calls—for example, a procedure calling a subprocedure, one method calling another method, or one procedure invoking another remotely through an RPC (such as CORBA and DCOM). Synchronous calls imply that the calling process is halted while the subprocess is executing a function. Even in an RPC scenario, where the called subprocedure executes in a different process, the caller blocks until the subprocedure returns control (and the results) to the caller. In contrast, when
using asynchronous messaging, the caller uses a send-and-forget approach that allows it to continue to execute after it sends the message. As a result, the calling procedure continues to run while the subprocedure is being invoked (see figure).

Asynchronous communication has a number of implications. First, we no longer have a single thread of execution. Multiple threads enable subprocedures to run concurrently, which can greatly improve performance and help ensure that some subprocesses are making progress even while other subprocesses may be waiting for external results. However, concurrent threads also make debugging much more difficult. Second, results (if any) arrive via a callback mechanism. This enables the caller to perform other tasks and be notified when the result is available, which can improve performance. However, this means that the caller has to be able to process the result even while it is in the middle of other tasks, and it has to be able to remember the context in which the call was made. Third, asynchronous subprocesses can execute in any order. Again, this enables one subprocedure to make progress even while another cannot. But it also means that the sub-processes must be able to run independently in any order, and the caller must be able to determine which result came from which subprocess and combine the results together. As a result, asynchronous communication has several advantages but requires rethinking how a procedure uses its subprocedures.

Distributed Applications versus Integration

This book is about enterprise integration—how to integrate independent applications so that they can work together. An enterprise application often incorporates an n-tier architecture (a more sophisticated version of a client/server
architecture), enabling it to be distributed across several computers. Even though this results in processes on different machines communicating with each other, this is application distribution, not application integration.

Why is an \( n \)-tier architecture considered application distribution and not application integration? First, the communicating parts are tightly coupled—they dependent directly on each other, so one tier cannot function without the others. Second, communication between tiers tends to be synchronous. Third, an application (\( n \)-tier or atomic) tends to have human users who will only accept rapid system response times.

In contrast, integrated applications are independent applications that can each run by themselves but that coordinate with each other in a loosely coupled way. This enables each application to focus on one comprehensive set of functionality and yet delegate to other applications for related functionality. Integrated applications communicating asynchronously don’t have to wait for a response; they can proceed without a response or perform other tasks concurrently until the response is available. Integrated applications tend to have a broad time constraint, such that they can work on other tasks until a result becomes available, and therefore are more patient than most human users waiting real-time for a result.

**Commercial Messaging Systems**

The apparent benefits of integrating systems using an asynchronous messaging solution have opened up a significant market for software vendors creating messaging middleware and associated tools. We can roughly group the messaging vendors’ products into the following four categories:

1. **Operating systems.** Messaging has become such a common need that vendors have started to integrate the necessary software infrastructure into the operating system or database platform. For example, the Microsoft Windows 2000 and Windows XP operating systems include the Microsoft Message Queuing (MSMQ) service software. This service is accessible through a number of APIs, including COM components and the System.Messaging namespace, part of the Microsoft .NET platform. Similarly, Oracle offers Oracle AQ as part of its database platform.

2. **Application servers.** Sun Microsystems first incorporated the Java Messaging Service (JMS) into version 1.2 of the J2EE specification. Since then, virtually all J2EE application servers (such as IBM WebSphere and BEA WebLogic)
provide an implementation for this specification. Also, Sun delivers a JMS reference implementation with the J2EE JDK.

3. *EAI suites.* Products from these vendors offer proprietary—but functionally rich—suites that encompass messaging, business process automation, workflow, portals, and other functions. Key players in this marketplace are IBM WebSphere MQ, Microsoft BizTalk, TIBCO, WebMethods, SeeBeyond, Vitria, CrossWorlds, and others. Many of these products include JMS as one of the many client APIs they support, while other vendors—such as SonicSoftware and Fiorano—focus primarily on implementing JMS-compliant messaging infrastructures.

4. *Web services toolkits.* Web services have garnered a lot of interest in the enterprise integration communities. Standards bodies and consortia are actively working on standardizing reliable message delivery over Web services (i.e., WS-Reliability, WS-ReliableMessaging, and ebMS). A growing number of vendors offer tools that implement routing, transformation, and management of Web services-based solutions.

The patterns in this book are vendor-independent and apply to most messaging solutions. Unfortunately, each vendor tends to define its own terminology when describing messaging solutions. In this book, we strove to choose pattern names that are technology- and product-neutral yet descriptive and easy to use conversationally.

Many messaging vendors have incorporated some of this book’s patterns as features of their products, which simplifies applying the patterns and accelerates solution development. Readers who are familiar with a particular vendor’s terminology will most likely recognize many of the concepts in this book. To help these readers map the pattern language to the vendor-specific terminology, the following tables map the most common pattern names to their corresponding product feature names in some of the most widely used messaging products.

<table>
<thead>
<tr>
<th>Enterprise Integration Patterns</th>
<th>Java Message Service (JMS)</th>
<th>Microsoft MSMQ</th>
<th>WebSphere MQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Channel</td>
<td>Destination</td>
<td>MessageQueue</td>
<td>Queue</td>
</tr>
<tr>
<td>Point-to-Point Channel</td>
<td>Queue</td>
<td>MessageQueue</td>
<td>Queue</td>
</tr>
<tr>
<td>Publish-Subscribe Channel</td>
<td>Topic</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Message</td>
<td>Message</td>
<td>Message</td>
<td>Message</td>
</tr>
<tr>
<td>Message Endpoint</td>
<td>MessageProducer, MessageConsumer</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Pattern Form

This book contains a set of patterns organized into a pattern language. Books such as Design Patterns, Pattern Oriented Software Architecture, Core J2EE Patterns, and Patterns of Enterprise Application Architecture have popularized the concept of using patterns to document computer-programming techniques. Christopher Alexander pioneered the concept of patterns and pattern languages in his books A Pattern Language and A Timeless Way of Building. Each pattern represents a decision that must be made and the considerations that go into that decision. A pattern language is a web of related patterns where each pattern leads to others, guiding you through the decision-making process. This approach is a powerful technique for documenting an expert’s knowledge so that it can be readily understood and applied by others.

A pattern language teaches you how to solve a limitless variety of problems within a bounded problem space. Because the overall problem that is being solved is different every time, the path through the patterns and how they’re applied is also unique. This book is written for anyone using any messaging tools for any application, and it can be applied specifically for you and the unique application of messaging that you face.

Using the pattern form by itself does not guarantee that a book contains a wealth of knowledge. It is not enough to simply say, “When you face this problem, apply this solution.” For you to truly learn from a pattern, the pattern has to document why the problem is difficult to solve, consider possible solutions that in fact don’t work well, and explain why the solution offered is the best available. Likewise, the patterns need to connect to each other so as to walk you
from one problem to the next. In this way, the pattern form can be used to teach not just what solutions to apply but also how to solve problems the authors could not have predicted. These are goals we strive to accomplish in this book.

Patterns should be prescriptive, meaning that they should tell you what to do. They don’t just describe a problem, and they don’t just describe how to solve it—they tell you what to do to solve it. Each pattern represents a decision you must make: “Should I use Messaging?” “Would a Command Message help me here?” The point of the patterns and the pattern language is to help you make decisions that lead to a good solution for your specific problem, even if the authors didn’t have that specific problem in mind and even if you don’t have the knowledge and experience to develop that solution on your own.

There is no one universal pattern form; different books use various structures. We used a style that is fairly close to the Alexandrian form, which was first popularized for computer programming in Smalltalk Best Practice Patterns by Kent Beck. We like the Alexandrian form because it results in patterns that are more prose-like. As a result, even though each pattern follows an identical, well-defined structure, the format avoids headings for individual subsections, which would disrupt the flow of the discussion. To improve navigability, the format uses style elements such as underscoring, indentation, and illustrations to help you identify important information at a quick glance.

Each pattern follows this structure:

- **Name**—This is an identifier for the pattern that indicates what the pattern does. We chose names that can easily be used in a sentence so that it is easy to reference the pattern’s concept in a conversation between designers.

- **Icon**—Most patterns are associated with an icon in addition to the pattern name. Because many architects are used to communicating visually through diagrams, we provide a visual language in addition to the verbal language. This visual language underlines the composability of the patterns, as multiple pattern icons can be combined to describe the solution of a larger, more complex pattern.

- **Context**—This section explains what type of work might make you run into the problem that this pattern solves. The context sets the stage for the problem and often refers to other patterns you may have already applied.

- **Problem**—This explains the difficulty you are facing, expressed as a question. You should be able to read the problem statement and quickly determine if this pattern is relevant to your work. We’ve formatted the problem to be one sentence delimited by horizontal rules.
• **Forces**—The forces explore the constraints that make the problem difficult to solve. They often consider alternative solutions that seem promising but don’t pan out, which helps show the value of the real solution.

• **Solution**—This part explains what you should do to solve the problem. It is not limited to your particular situation, but describes what to do in the variety of circumstances represented by the problem. If you understand a pattern’s problem and solution, you understand the pattern. We’ve formatted the solution in the same style as the problem so that you can easily spot problem and solution statements when perusing the book.

• **Sketch**—One of the most appealing properties of the Alexandrian form is that each pattern contains a sketch that illustrates the solution. In many cases, just by looking at the pattern name and the sketch, you can understand the essence of the pattern. We tried to maintain this style by illustrating the solution with a figure immediately following the solution statement of each pattern.

• **Results**—This part expands upon the solution to explain the details of how to apply the solution and how it resolves the forces. It also addresses new challenges that may arise as a result of applying this pattern.

• **Next**—This section lists other patterns to be considered after applying the current one. Patterns don’t live in isolation; the application of one pattern usually leads you to new problems that are solved by other patterns. The relationships between patterns are what constitutes a pattern language as opposed to just a pattern catalog.

• **Sidebars**—These sections discuss more detailed technical issues or variations of the pattern. We set these sections visually apart from the remainder of the text so you can easily skip them if they are not relevant to your particular application of the pattern.

• **Examples**—A pattern usually includes one or more examples of the pattern being applied or having been applied. An example may be as simple as naming a known use or as detailed as a large segment of sample code. Given the large number of available messaging technologies, we do not expect you to be familiar with each technology used to implement an example. Therefore, we designed the patterns so that you can safely skip the example without losing any critical content of the pattern.

The beauty in describing solutions as patterns is that it teaches you not only how to solve the specific problems discussed, but also how to create designs
that solve problems the authors were not even aware of. As a result, these patterns for messaging not only describe messaging systems that exist today, but may also apply to new ones created well after this book is published.

Diagram Notation

Integration solutions consist of many different pieces—applications, databases, endpoints, channels, messages, routers, and so on. If we want to describe an integration solution, we need to define a notation that accommodates all these different components. To our knowledge, there is no widely used, comprehensive notation that is geared toward the description of all aspects of an integration solution. The Unified Modeling Language (UML) does a fine job of describing object-oriented systems with class and interaction diagrams, but it does not contain semantics to describe messaging solutions. The UML Profile for EAI [UMLEAI] enriches the semantics of collaboration diagrams to describe message flows between components. This notation is very useful as a precise visual specification that can serve as the basis for code generation as part of a model-driven architecture (MDA). We decided not to adopt this notation for two reasons. First, the UML Profile does not capture all the patterns described in our pattern language. Second, we were not looking to create a precise visual specification, but images that have a certain “sketch” quality to them. We wanted pictures that are able to convey the essence of a pattern at a quick glance—very much like Alexander's sketch. That's why we decided to create our own “notation.” Luckily, unlike the more formal notation, ours does not require you to read a large manual. A simple picture should suffice:

This simple picture shows a message being sent to a component over a channel. We use the word component very loosely here—it can indicate an application that is being integrated, an intermediary that transforms or routes the
message between applications, or a specific part of an application. Sometimes, we also depict a channel as a three-dimensional pipe if we want to highlight the channel itself. Often, we are more interested in the components and draw the channels as simple lines with arrow heads. The two notations are equivalent. We depict the message as a small tree with a round root and nested, square elements because many messaging systems allow messages to contain tree-like data structures—for example, XML documents. The tree elements can be shaded or colored to highlight their usage in a particular pattern. Depicting messages in this way allows us to provide a quick visual description of transformation patterns—it is easy to show a pattern that adds, rearranges, or removes fields from the message.

When we describe application designs—for example, messaging endpoints or examples written in C# or Java—we do use standard UML class and sequence diagrams to depict the class hierarchy and the interaction between objects because the UML notation is widely accepted as the standard way of describing these types of solutions (if you need a refresher on UML, have a look at [UML]).

Examples and Interludes

We have tried to underline the broad applicability of the patterns by including implementation examples using a variety of integration technologies. The potential downside of this approach is that you may not be familiar with each technology that is being used in an example. That’s why we made sure that reading the examples is strictly optional—all relevant points are discussed in the pattern description. Therefore, you can safely skip the examples without risk of losing out on important detail. Also, where possible, we provided more than one implementation example using different technologies.

When presenting example code, we focused on readability over runnability. A code segment can help remove any potential ambiguity left by the solution description, and many application developers and architects prefer looking at 30 lines of code to reading many paragraphs of text. To support this intent, we often show only the most relevant methods or classes of a potentially larger solution. We also omitted most forms of error checking to highlight the core function implemented by the code. Most code snippets do not contain in-line comments, as the code is explained in the paragraphs before and after the code segment.

Providing a meaningful example for a single integration pattern is challenging. Enterprise integration solutions typically consist of a number of heterogeneous
components spread across multiple systems. Likewise, most integration patterns
do not operate in isolation but rely on other patterns to form a meaningful solu-
tion. To highlight the collaboration between multiple patterns, we included more
comprehensive examples as interludes (see Chapters 6, 9, and 12). These solu-
tions illustrate many of the trade-offs involved in designing a more comprehen-
sive messaging solution.

All code samples should be treated as illustrative tools only and not as a
starting point for development of a production-quality integration solution. For
example, almost all examples lack any form of error checking or concern for
robustness, security, or scalability.

We tried as much as possible to base the examples on software platforms
that are available free of charge or as a trial version. In some cases, we used
commercial platforms (such as TIBCO ActiveEnterprise and Microsoft BizTalk)
to illustrate the difference between developing a solution from scratch and
using a commercial tool. We presented those examples in such a way that they
are educational even if you do not have access to the required runtime platform.
For many examples, we use relatively barebones messaging frameworks such as
JMS or MSMQ. This allows us to be more explicit in the example and focus on
the problem at hand instead of distracting from it with all the features a more
complex middleware toolset may provide.

The Java examples in this book are based on the JMS 1.1 specification,
which is part of the J2EE 1.4 specification. By the time this book is published,
most messaging and application server vendors will support JMS 1.1. You can
download Sun Microsystems’ reference implementation of the JMS specifica-

The Microsoft .NET examples are based on Version 1.1 of the .NET Frame-
work and are written in C#. You can download the .NET Framework SDK

Organization of This Book

The pattern language in this book, as with any pattern language, is a web of
patterns referring to each other. At the same time, some patterns are more fun-
damental than others, forming a hierarchy of big-concept patterns that lead to
more finely detailed patterns. The big-concept patterns form the load-bearing
members of the pattern language. They are the main ones, the root patterns that
provide the foundation of the language and support the other patterns.
This book groups patterns into chapters by level of abstraction and by topic area. The following diagram shows the root patterns and their relationship to the chapters of the book.

The most fundamental pattern is *Messaging* (53); that’s what this book is about. It leads to the six root patterns described in Chapter 3, “Messaging Systems,” namely, *Message Channel* (60), *Message* (66), *Pipes and Filters* (70), *Message Router* (78), *Message Translator* (85), and *Message Endpoint* (95). In turn, each root pattern leads to its own chapter in the book (except *Pipes and Filters* [70], which is not specific to messaging but is a widely used architectural style that forms the basis of the routing and transformation patterns).

The pattern language is divided into eight chapters, which follow the hierarchy just described:

Chapter 2, “Integration Styles”—This chapter reviews the different approaches available for integrating applications, including *Messaging* (53).

Chapter 3, “Messaging Systems”—This chapter reviews the six root messaging patterns, giving an overview of the entire pattern language.

Chapter 4, “Messaging Channels”—Applications communicate via channels. Channels define the logical pathways a message can follow. This chapter shows how to determine what channels your applications need.

Chapter 5, “Message Construction”—Once you have message channels, you need messages to send on them. This chapter explains the different ways messages can be used and how to take advantage of their special properties.
Chapter 7, “Message Routing”—Messaging solutions aim to decouple the sender and the receiver of information. Message routers provide location independence between sender and receiver so that senders don’t have to know about who processes their messages. Rather, they send the messages to intermediate message routing components that forward the message to the correct destination. This chapter presents a variety of different routing techniques.

Chapter 8, “Message Transformation”—Independently developed applications often don’t agree on messages’ formats, on the form and meaning of supposedly unique identifiers, or even on the character encoding to be used. Therefore, intermediate components are needed to convert messages from the format one application produces to that of the receiving applications. This chapter shows how to design transformer components.

Chapter 10, “Messaging Endpoints”—Many applications were not designed to participate in a messaging solution. As a result, they must be explicitly connected to the messaging system. This chapter describes a layer in the application that is responsible for sending and receiving the messages, making your application an endpoint for messages.

Chapter 11, “System Management”—Once a messaging system is in place to integrate applications, how do we make sure that it’s running correctly and doing what we want? This chapter explores how to test and monitor a running messaging system.

These eight chapters together teach you what you need to know about connecting applications using messaging.

Getting Started

With any book that has a lot to teach, it’s hard to know where to start, both for the authors and the readers. Reading all of the pages straight through assures covering the entire subject area but isn’t the quickest way to get to the issues that are of the most help. Starting with a pattern in the middle of the language can be like starting to watch a movie that’s half over—you see what’s happening but don’t understand what it means.

Luckily, the pattern language is formed around the root patterns described earlier. These root patterns collectively provide an overview of the pattern language, and individually provide starting points for delving deep into the details.
of messaging. To get an overall survey of the language without reviewing all of the patterns, start with reviewing the root patterns in Chapter 3.

Chapter 2, “Integration Styles,” provides an overview of the four main application integration techniques and settles on Messaging [53] as being the best overall approach for many integration opportunities. Read this chapter if you are unfamiliar with issues involved in application integration and the pros and cons of the various approaches that are available. If you’re already convinced that messaging is the way to go and want to get started with how to use messaging, you can skip this chapter completely.

Chapter 3, “Messaging Systems,” contains all of this pattern language’s root patterns (except Messaging [53], which is in Chapter 2). For an overview of the pattern language, read (or at least skim) all of the patterns in this chapter. To dive deeply on a particular topic, read its root pattern, then go to the patterns mentioned at the end of the pattern section; those next patterns will all be in a chapter named after the root pattern.

After Chapters 2 and 3, different types of messaging developers may be most interested in different chapters based on the specifics of how each group uses messaging to perform integration:

- **System administrators** may be most interested in Chapter 4, “Messaging Channels,” the guidelines for what channels to create, and Chapter 11, “System Management,” guidance on how to maintain a running messaging system.

- **Application developers** should look at Chapter 10, “Messaging Endpoints,” to learn how to integrate an application with a messaging system and at Chapter 5, “Message Construction,” to learn what messages to send when.

- **System integrators** will gain the most from Chapter 7, “Message Routing”—how to direct messages to the proper receivers—and Chapter 8, “Message Transformation”—how to convert messages from the sender’s format to the receiver’s.

Keep in mind that when reading a pattern, if you’re in a hurry, start by just reading the problem and solution. This will give you enough information to determine if the pattern is of interest to you right now and if you already know the pattern. If you do not know the pattern and it sounds interesting, go ahead and read the other parts.

Also remember that this is a pattern language, so the patterns are not necessarily meant to be read in the order they’re presented in the book. The book’s
order teaches you about messaging by considering all of the relevant topics in turn and discussing related issues together. To use the patterns to solve a particular problem, start with an appropriate root pattern. Its context explains what patterns need to be applied before this one, even if they’re not the ones immediately preceding this one in the book. Likewise, the next section (the last paragraph of the pattern) describes what patterns to consider applying after this one, even if they’re not the ones immediately following this one in the book. Use the web of interconnected patterns, not the linear list of book pages, to guide you through the material.

Supporting Web Site

Please look for companion information to this book plus related information on enterprise integration at our Web site: www.enterpriseintegrationpatterns.com. You can also e-mail your comments, suggestions, and feedback to us at authors@enterpriseintegrationpatterns.com.

Summary

You should now have a good understanding of the following concepts, which are fundamental to the material in this book:

- What messaging is.
- What a messaging system is.
- Why to use messaging.
- How asynchronous programming is different from synchronous programming.
- How application integration is different from application distribution.
- What types of commercial products contain messaging systems.
You should also have a feel for how this book is going to teach you to use messaging:

• The role patterns have in structuring the material.
• The meaning of the custom notation used in the diagrams.
• The purpose and scope of the examples.
• The organization of the material.
• How to get started learning the material.

Now that you understand the basic concepts and how the material will be presented, we invite you to start learning about enterprise integration using messaging.
This page intentionally left blank
Chapter 2

Integration Styles

Introduction

Enterprise integration is the task of making disparate applications work together to produce a unified set of functionality. These applications can be custom developed in house or purchased from third-party vendors. They likely run on multiple computers, which may represent multiple platforms, and may be geographically dispersed. Some of the applications may be run outside of the enterprise by business partners or customers. Other applications might not have been designed with integration in mind and are difficult to change. These issues and others like them make application integration complicated. This chapter explores multiple integration approaches that can help overcome these challenges.

Application Integration Criteria

What makes good application integration? If integration needs were always the same, there would be only one integration style. Yet, like any complex technological effort, application integration involves a range of considerations and consequences that should be taken into account for any integration opportunity.

The fundamental criterion is whether to use application integration at all. If you can develop a single, standalone application that doesn’t need to collaborate with any other applications, you can avoid the whole integration issue entirely. Realistically, though, even a simple enterprise has multiple applications that need to work together to provide a unified experience for the enterprise’s employees, partners, and customers.

The following are some other main decision criteria.

Application coupling—Integrated applications should minimize their dependencies on each other so that each can evolve without causing problems to the others. As explained in Chapter 1, “Solving Integration Problems Using Patterns,” tightly coupled applications make numerous assumptions about
how the other applications work; when the applications change and break those assumptions, the integration between them breaks. Therefore, the interfaces for integrating applications should be specific enough to implement useful functionality but general enough to allow the implementation to change as needed.

**Intrusiveness**—When integrating an application into an enterprise, developers should strive to minimize both changes to the application and the amount of integration code needed. Yet, changes and new code are often necessary to provide good integration functionality, and the approaches with the least impact on the application may not provide the best integration into the enterprise.

**Technology selection**—Different integration techniques require varying amounts of specialized software and hardware. Such tools can be expensive, can lead to vendor lock-in, and can increase the learning curve for developers. On the other hand, creating an integration solution from scratch usually results in more effort than originally intended and can mean reinventing the wheel.

**Data format**—Integrated applications must agree on the format of the data they exchange. Changing existing applications to use a unified data format may be difficult or impossible. Alternatively, an intermediate translator can unify applications that insist on different data formats. A related issue is **data format evolution and extensibility**—how the format can change over time and how that change will affect the applications.

**Data timeliness**—Integration should minimize the length of time between when one application decides to share some data and other applications have that data. This can be accomplished by exchanging data frequently and in small chunks. However, chunking a large set of data into small pieces may introduce inefficiencies. Latency in data sharing must be factored into the integration design. Ideally, receiver applications should be informed as soon as shared data is ready for consumption. The longer sharing takes, the greater the opportunity for applications to get out of sync and the more complex integration can become.

**Data or functionality**—Many integration solutions allow applications to share not only data but functionality as well, because sharing of functionality can provide better abstraction between the applications. Even though invoking functionality in a remote application may seem the same as invoking local functionality, it works quite differently, with significant consequences for how well the integration works.
Remote Communication—Computer processing is typically synchronous—that is, a procedure waits while its subprocedure executes. However, calling a remote subprocedure is much slower than a local one so that a procedure may not want to wait for the subprocedure to complete; instead, it may want to invoke the subprocedure asynchronously, that is, starting the subprocedure but continuing with its own processing simultaneously. Asynchronicity can make for a much more efficient solution, but such a solution is also more complex to design, develop, and debug.

Reliability—Remote connections are not only slow, but they are much less reliable than a local function call. When a procedure calls a subprocedure inside a single application, it's a given that the subprocedure is available. This is not necessarily true when communicating remotely; the remote application may not even be running or the network may be temporarily unavailable. Reliable, asynchronous communication enables the source application to go on to other work, confident that the remote application will act sometime later.

So, as you can see, there are several different criteria that must be considered when choosing and designing an integration approach. The question then becomes, Which integration approach best addresses which of these criteria?

Application Integration Options

There is no one integration approach that addresses all criteria equally well. Therefore, multiple approaches for integrating applications have evolved over time. The various approaches can be summed up in four main integration styles.

File Transfer (43)—Have each application produce files of shared data for others to consume and consume files that others have produced.

Shared Database (47)—Have the applications store the data they wish to share in a common database.

Remote Procedure Invocation (50)—Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to initiate behavior and exchange data.

Messaging (53)—Have each application connect to a common messaging system, and exchange data and invoke behavior using messages.

This chapter presents each style as a pattern. The four patterns share the same problem statement—the need to integrate applications—and very similar contexts. What differentiates them are the forces searching for a more elegant
solution. Each pattern builds on the last, looking for a more sophisticated approach to address the shortcomings of its predecessors. Thus, the pattern order reflects an increasing order of sophistication, but also increasing complexity.

The trick is not to choose one style to use every time but to choose the best style for a particular integration opportunity. Each style has its advantages and disadvantages. Applications may integrate using multiple styles so that each point of integration takes advantage of the style that suits it best. Likewise, an application may use different styles to integrate with different applications, choosing the style that works best for the other application. As a result, many integration approaches can best be viewed as a hybrid of multiple integration styles. To support this type of integration, many integration and EAI middleware products employ a combination of styles, all of which are effectively hidden in the product’s implementation.

The patterns in the remainder of this book expand on the Messaging (53) integration style. We focus on messaging because we believe that it provides a good balance between the integration criteria but is also the most difficult style to work with. As a result, messaging is still the least well understood of the integration styles and a technology ripe with patterns that quickly explain how to use it best. Finally, messaging is the basis for many commercial EAI products, so explaining how to use messaging well also goes a long way in teaching you how to use those products. The focus of this section is to highlight the issues involved with application integration and how messaging fits into the mix.
An enterprise has multiple applications that are being built independently, with different languages and platforms.

How can I integrate multiple applications so that they work together and can exchange information?

In an ideal world, you might imagine an organization operating from a single, cohesive piece of software, designed from the beginning to work in a unified and coherent way. Of course, even the smallest operations don’t work like that. Multiple pieces of software handle different aspects of the enterprise. This is due to a host of reasons.

- People buy packages that are developed by outside organizations.
- Different systems are built at different times, leading to different technology choices.
- Different systems are built by different people whose experience and preferences lead them to different approaches to building applications.
- Getting an application out and delivering value is more important than ensuring that integration is addressed, especially when that integration doesn’t add any value to the application under development.

As a result, any organization has to worry about sharing information between very divergent applications. These can be written in different languages, based on different platforms, and have different assumptions about how the business operates.

Tying together such applications requires a thorough understanding of how to link together applications on both the business and technical levels. This is a lot easier if you minimize what you need to know about how each application works.
What is needed is a common data transfer mechanism that can be used by a variety of languages and platforms but that feels natural to each. It should require a minimal amount of specialized hardware and software, making use of what the enterprise already has available.

Files are a universal storage mechanism, built into any enterprise operating system and available from any enterprise language. The simplest approach would be to somehow integrate the applications using files.

Have each application produce files that contain the information the other applications must consume. Integrators take the responsibility of transforming files into different formats. Produce the files at regular intervals according to the nature of the business.

An important decision with files is what format to use. Very rarely will the output of one application be exactly what’s needed for another, so you’ll have to do a fair bit of processing of files along the way. This means not only that all the applications that use a file have to read it, but that you also have to be able to use processing tools on it. As a result, standard file formats have grown up over time. Mainframe systems commonly use data feeds based on the file system formats of COBOL. UNIX systems use text-based files. The current method is to use XML. An industry of readers, writers, and transformation tools has built up around each of these formats.

Another issue with files is when to produce them and consume them. Since there’s a certain amount of effort required to produce and process a file, you usually don’t want to work with them too frequently. Typically, you have some regular business cycle that drives the decision: nightly, weekly, quarterly, and so on. Applications get used to when a new file is available and processes it at its time.

The great advantage of files is that integrators need no knowledge of the internals of an application. The application team itself usually provides the file. The file’s contents and format are negotiated with integrators, although if a
package is used, the choices are often limited. The integrators then deal with the transformations required for other applications, or they leave it up to the consuming applications to decide how they want to manipulate and read the file. As a result, the different applications are quite nicely decoupled from each other. Each application can make internal changes freely without affecting other applications, providing they still produce the same data in the files in the same format. The files effectively become the public interface of each application.

Part of what makes File Transfer simple is that no extra tools or integration packages are needed, but that also means that developers have to do a lot of the work themselves. The applications must agree on file-naming conventions and the directories in which they appear. The writer of a file must implement a strategy to keep the file names unique. The applications must agree on which one will delete old files, and the application with that responsibility will have to know when a file is old and no longer needed. The applications will need to implement a locking mechanism or follow a timing convention to ensure that one application is not trying to read the file while another is still writing it. If all of the applications do not have access to the same disk, then some application must take responsibility for transferring the file from one disk to another.

One of the most obvious issues with File Transfer is that updates tend to occur infrequently, and as a result systems can get out of synchronization. A customer management system can process a change of address and produce an extract file each night, but the billing system may send the bill to an old address on the same day. Sometimes lack of synchronization isn’t a big deal. People often expect a certain lag in getting information around, even with computers. At other times the result of using stale information is a disaster. When deciding on when to produce files, you have to take the freshness needs of consumers into account.

In fact, the biggest problem with staleness is often on the software development staff themselves, who frequently must deal with data that isn’t quite right. This can lead to inconsistencies that are difficult to resolve. If a customer changes his address on the same day with two different systems, but one of them makes an error and gets the wrong street name, you’ll have two different addresses for a customer. You’ll need some way to figure out how to resolve this. The longer the period between file transfers, the more likely and more painful this problem can become.

Of course, there’s no reason that you can’t produce files more frequently. Indeed, you can think of Messaging (53) as File Transfer where you produce a file with every change in an application. The problem then is managing all the files that get produced, ensuring that they are all read and that none get lost. This goes beyond what file system–based approaches can do, particularly since
there are expensive resource costs associated with processing a file, which can get prohibitive if you want to produce lots of files quickly. As a result, once you get to very fine-grained files, it’s easier to think of them as Messaging (53).

To make data available more quickly and enforce an agreed-upon set of data formats, use a Shared Database (47). To integrate applications’ functionality rather than their data, use Remote Procedure Invocation (50). To enable frequent exchanges of small amounts of data, perhaps used to invoke remote functionality, use Messaging (53).
An enterprise has multiple applications that are being built independently, with different languages and platforms. The enterprise needs information to be shared rapidly and consistently.

How can I integrate multiple applications so that they work together and can exchange information?

File Transfer (43) enables applications to share data, but it can lack timeliness—yet timeliness of integration is often critical. If changes do not quickly work their way through a family of applications, you are likely to make mistakes due to the staleness of the data. For modern businesses, it is imperative that everyone have the latest data. This not only reduces errors, but also increases people's trust in the data itself.

Rapid updates also allow inconsistencies to be handled better. The more frequently you synchronize, the less likely you are to get inconsistencies and the less effort they are to deal with. But however rapid the changes, there are still going to be problems. If an address is updated inconsistently in rapid succession, how do you decide which one is the true address? You could take each piece of data and say that one application is the master source for that data, but then you'd have to remember which application is the master for which data.

File Transfer (43) also may not enforce data format sufficiently. Many of the problems in integration come from incompatible ways of looking at the data. Often these represent subtle business issues that can have a huge effect. A geological database may define an oil well as a single drilled hole that may or may not produce oil. A production database may define a well as multiple holes covered by a single piece of equipment. These cases of semantic dissonance are much harder to deal with than inconsistent data formats. (For a much deeper discussion of these issues, it's really worth reading Data and Reality [Kent].)

What is needed is a central, agreed-upon datastore that all of the applications share so each has access to any of the shared data whenever it needs it.
Integrate applications by having them store their data in a single *Shared Database*, and define the schema of the database to handle all the needs of the different applications.

If a family of integrated applications all rely on the same database, then you can be pretty sure that they are always consistent all of the time. If you do get simultaneous updates to a single piece of data from different sources, then you have transaction management systems that handle that about as gracefully as it ever can be managed. Since the time between updates is so small, any errors are much easier to find and fix.

*Shared Database* is made much easier by the widespread use of SQL-based relational databases. Pretty much all application development platforms can work with SQL, often with quite sophisticated tools. So you don’t have to worry about multiple file formats. Since any application pretty much has to use SQL anyway, this avoids adding yet another technology for everyone to master.

Since every application is using the same database, this forces out problems in semantic dissonance. Rather than leaving these problems to fester until they are difficult to solve with transforms, you are forced to confront them and deal with them before the software goes live and you collect large amounts of incompatible data.

One of the biggest difficulties with *Shared Database* is coming up with a suitable design for the shared database. Coming up with a unified schema that can meet the needs of multiple applications is a very difficult exercise, often resulting in a schema that application programmers find difficult to work with. And if the technical difficulties of designing a unified schema aren’t enough, there are also severe political difficulties. If a critical application is likely to suffer delays in order to work with a unified schema, then often there is irresistible pressure to separate. Human conflicts between departments often exacerbate this problem.
Another, harder limit to Shared Database is external packages. Most packaged applications won’t work with a schema other than their own. Even if there is some room for adaptation, it’s likely to be much more limited than integrators would like. Adding to the problem, software vendors usually reserve the right to change the schema with every new release of the software.

This problem also extends to integration after development. Even if you can organize all your applications, you still have an integration problem should a merger of companies occur.

Multiple applications using a Shared Database to frequently read and modify the same data can turn the database into a performance bottleneck and can cause deadlocks as each application locks others out of the data. When applications are distributed across multiple locations, accessing a single, shared database across a wide-area network is typically too slow to be practical. Distributing the database as well allows each application to access the database via a local network connection, but confuses the issue of which computer the data should be stored on. A distributed database with locking conflicts can easily become a performance nightmare.

To integrate applications’ functionality rather than their data, use Remote Procedure Invocation (50). To enable frequent exchanges of small amounts of data using a format per datatype rather than one universal schema, use Messaging (53).
An enterprise has multiple applications that are being built independently, with different languages and platforms. The enterprise needs to share data and processes in a responsive way.

How can I integrate multiple applications so that they work together and can exchange information?

File Transfer (43) and Shared Database (47) enable applications to share their data, which is an important part of application integration, but just sharing data is often not enough. Changes in data often require actions to be taken across different applications. For example, changing an address may be a simple change in data, or it may trigger registration and legal processes to take into account different rules in different legal jurisdictions. Having one application invoke such processes directly in others would require applications to know far too much about the internals of other applications.

This problem mirrors a classic dilemma in application design. One of the most powerful structuring mechanisms in application design is encapsulation, where modules hide their data through a function call interface. In this way, they can intercept changes in data to carry out the various actions they need to perform when the data is changed. Shared Database (47) provides a large, unencapsulated data structure, which makes it much harder to do this. File Transfer (43) allows an application to react to changes as it processes the file, but the process is delayed.

The fact that Shared Database (47) has unencapsulated data also makes it more difficult to maintain a family of integrated applications. Many changes in any application can trigger a change in the database, and database changes have a considerable ripple effect through every application. As a result, organizations that use Shared Database (47) are often very reluctant to change the database, which means that the application development work is much less responsive to the changing needs of the business.
What is needed is a mechanism for one application to invoke a function in another application, passing the data that needs to be shared and invoking the function that tells the receiver application how to process the data.

Remote Procedure Invocation applies the principle of encapsulation to integrating applications. If an application needs some information that is owned by another application, it asks that application directly. If one application needs to modify the data of another, it does so by making a call to the other application. This allows each application to maintain the integrity of the data it owns. Furthermore, each application can alter the format of its internal data without affecting every other application.

A number of technologies, such as CORBA, COM, .NET Remoting, and Java RMI, implement Remote Procedure Invocation (also referred to as Remote Procedure Call, or RPC). These approaches vary as to how many systems support them and their ease of use. Often these environments add additional capabilities, such as transactions. For sheer ubiquity, the current favorite is Web services, using standards such as SOAP and XML. A particularly valuable feature of Web services is that they work easily with HTTP, which is easy to get through firewalls.

The fact that there are methods that wrap the data makes it easier to deal with semantic dissonance. Applications can provide multiple interfaces to the same data, allowing some clients to see one style and others a different style. Even updates can use multiple interfaces. This provides a lot more ability to support multiple points of view than can be achieved by relational views. However, it is awkward for integrators to add transformation components, so each application has to negotiate its interface with its neighbors.
Since software developers are used to procedure calls, *Remote Procedure Invocation* fits in nicely with what they are already used to. Actually, this is more of a disadvantage than an advantage. There are big differences in performance and reliability between remote and local procedure calls. If people don’t understand these, then *Remote Procedure Invocation* can lead to slow and unreliable systems (see [Waldo], [EAA]).

Although encapsulation helps reduce the coupling of the applications by eliminating a large shared data structure, the applications are still fairly tightly coupled together. The remote calls that each system supports tend to tie the different systems into a growing knot. In particular, sequencing—doing certain things in a particular order—can make it difficult to change systems independently. These types of problems often arise because issues that aren’t significant within a single application become so when integrating applications. People often design the integration the way they would design a single application, unaware that the rules of the engagement change dramatically.

To integrate applications in a more loosely coupled, asynchronous fashion, use *Messaging* (53) to enable frequent exchanges of small amounts of data, ones that are perhaps used to invoke remote functionality.
An enterprise has multiple applications that are being built independently, with different languages and platforms. The enterprise needs to share data and processes in a responsive way.

How can I integrate multiple applications so that they work together and can exchange information?

*File Transfer* (43) and *Shared Database* (47) enable applications to share their data but not their functionality. *Remote Procedure Invocation* (50) enables applications to share functionality, but it tightly couples them as well. Often the challenge of integration is about making collaboration between separate systems as timely as possible, without coupling systems together in such a way that they become unreliable either in terms of application execution or application development.

*File Transfer* (43) allows you to keep the applications well decoupled but at the cost of timeliness. Systems just can’t keep up with each other. Collaborative behavior is way too slow. *Shared Database* (47) keeps data together in a responsive way but at the cost of coupling everything to the database. It also fails to handle collaborative behavior.

Faced with these problems, *Remote Procedure Invocation* (50) seems an appealing choice. But extending a single application model to application integration dredges up plenty of other weaknesses. These weaknesses start with the essential problems of distributed development. Despite that RPCs look like local calls, they don’t behave the same way. Remote calls are slower, and they are much more likely to fail. With multiple applications communicating across an enterprise, you don’t want one application’s failure to bring down all of the other applications. Also, you don’t want to design a system assuming that calls are fast, and you don’t want each application knowing the details about other applications, even if it’s only details about their interfaces.

What we need is something like *File Transfer* (43) in which lots of little data packets can be produced quickly and transferred easily, and the receiver application is automatically notified when a new packet is available for consumption.
The transfer needs a retry mechanism to make sure it succeeds. The details of any disk structure or database for storing the data needs to be hidden from the applications so that, unlike Shared Database (47), the storage schema and details can be easily changed to reflect the changing needs of the enterprise. One application should be able to send a packet of data to another application to invoke behavior in the other application, like Remote Procedure Invocation (50), but without being prone to failure. The data transfer should be asynchronous so that the sender does not need to wait on the receiver, especially when retry is necessary.

Asynchronous messaging is fundamentally a pragmatic reaction to the problems of distributed systems. Sending a message does not require both systems to be up and ready at the same time. Furthermore, thinking about the communication in an asynchronous manner forces developers to recognize that working with a remote application is slower, which encourages design of components with high cohesion (lots of work locally) and low adhesion (selective work remotely).

Messaging systems also allow much of the decoupling you get when using File Transfer (43). Messages can be transformed in transit without either the sender or receiver knowing about the transformation. The decoupling allows integrators to choose between broadcasting messages to multiple receivers, routing a message to one of many receivers, or other topologies. This separates integration decisions from the development of the applications. Since human issues tend to separate application development from application integration, this approach works with human nature rather than against it.

The transformation means that separate applications can have quite different conceptual models. Of course, this means that semantic dissonance will occur.
However, the messaging viewpoint is that the measures used by Shared Database (47) to avoid semantic dissonance are too complicated to work in practice. Also, semantic dissonance is going to occur with third-party applications and with applications added as part of a corporate merger, so the messaging approach is to address the issue rather than design applications to avoid it.

By sending small messages frequently, you also allow applications to collaborate behaviorally as well as share data. If a process needs to be launched once an insurance claim is received, it can be done immediately by sending a message when a single claim comes in. Information can be requested and a reply made rapidly. While such collaboration isn’t going to be as fast as Remote Procedure Invocation (50), the caller needn’t stop while the message is being processed and the response returned. And messaging isn’t as slow as many people think—many messaging solutions originated in the financial services industry where thousands of stock quotes or trades have to pass through a messaging system every second.

This book is about Messaging, so you can safely assume that we consider Messaging to be generally the best approach to enterprise application integration. You should not assume, however, that it is free of problems. The high frequency of messages in Messaging reduces many of the inconsistency problems that bedevil File Transfer (43), but it doesn’t remove them entirely. There are still going to be some lag problems with systems not being updated quite simultaneously. Asynchronous design is not the way most software people are taught, and as a result there are many different rules and techniques in place. The messaging context makes this a bit easier than programming in an asynchronous application environment like X Windows, but asynchrony still has a learning curve. Testing and debugging are also harder in this environment.

The ability to transform messages has the nice benefit of allowing applications to be much more decoupled from each other than in Remote Procedure Invocation (50). But this independence does mean that integrators are often left with writing a lot of messy glue code to fit everything together.

Once you decide that you want to use Messaging for system integration, there are a number of new issues to consider and practices you can employ.

*How do you transfer packets of data?*

A sender sends data to a receiver by sending a Message (66) via a Message Channel (60) that connects the sender and receiver.

*How do you know where to send the data?*

If the sender does not know where to address the data, it can send the data to a Message Router (78), which will direct the data to the proper receiver.
How do you know what data format to use?

If the sender and receiver do not agree on the data format, the sender can direct the data to a Message Translator (85) that will convert the data to the receiver’s format and then forward the data to the receiver.

If you’re an application developer, how do you connect your application to the messaging system?

An application that wishes to use messaging will implement Message Endpoints (95) to perform the actual sending and receiving.
Index

A
Abstract pipes, 72
ACID (atomic, consistent, isolated, and durable), 484
ACT (Asynchronous Completion Token), 418, 472
ActiveEnterprise, see TIBCO
ActiveEnterprise
Activity diagrams, 21–22, 319
connecting to existing systems, 344
database, 344
message bus, 139
messaging systems, 102
Web services, 132
Address Change message, 31
Addresses, 30–32
Aggregate interface, 280
Aggregating
loan broker system strategies, 368
responses to single message, 298–300
Aggregation algorithm, 270
Aggregator class, 276–279
aggregation algorithm, 270
collect data for later evaluation algorithm, 273
completeness condition, 270
composed message processor, 296
condense data algorithm, 273
correlation, 270
correlation identifiers, 270–271
evendriven consumers, 278
External event strategy, 273
first best strategy, 273
implementation, 270–272
initialized, 274
JMS (Java Messaging Service), 276–282
listing active aggregates, 270
listing closed out aggregates, 271
loan broker, 275
loan broker system, 363, 368
loan broker system (ActiveEnterprise), 446–447, 458
loan broker system (MSMQ), 402, 422, 424
as missing message detector, 275–276
out-of-order messages, 284
parameters and strategy, 274
Publish-Subscribe Channel pattern, 22
scatter-gatherers, 300
selecting best answer algorithm, 273
sequentially numbered child messages, 262
splitters and, 274
stateful, 269
strategies, 272–274
timeout strategy, 272
timeout with override strategy, 273
wait for all strategy, 272
Apache Axis, 371, 376–378
Application integration
application coupling, 39–40
criteria, 39–41
data formats, 40
data sharing, 40
data timeliness, 40
capsulation, 51–52
Application integration, continued
file transfer, 41, 43–46
intrusiveness, 40
messaging, 41, 53–56
options, 41–42
reliability, 41
remote communication, 41
remote procedure invocation, 41, 50–52
shared database, 41, 47–49
sharing functionality, 40
technology selection, 40
Application layer, 88
Applications
automatically consuming messages, 498
brokering between, 82–83
channels, 61
client for each messaging system, 134
as client of messaging server, 95–96
collaborating behaviorally, 55
communicating with messaging, 60–66
communicating with simple protocol, 127
connecting to messaging system, 56
consistency, 48
consuming messages, 494
coupling, 39–40
data integrity, 51
data types, 88
deadlocks, 49
decoupling, 88–89
deleting files, 45
design and encapsulation, 50
different conceptual models, 54
errors, 117
exchanging data, 127
explicitly communicating with other applications, 323
file-naming conventions, 45
files, 44
handling different aspects of enterprise, 43
integration problems, 117
invalid request, 117
invoking procedures in, 145–146
logical entities, 88
messages, 67
much more decoupled, 55
multiple interfaces to data, 51
operating independently, 137–138
physical representation of data, 86
proprietary data models and data formats, 85
semantic dissonance, 47, 54
sharing databases, 47–49
sharing information, 43, 50–52
specific core function, 2
spreading business functions across, 2
as standalone solution, 127
tightly coupled, 39–40
transferring data between, 147–150
transmitting events between, 151–153
two-way conversation, 100
Application-specific messages, 20
AppSpecific property, 288, 290, 405, 424
Architectural patterns, 225, 228
Aspects, 219, 221
Asynchronous callback, 155–156
Asynchronous Completion Token pattern, 167, 472
Asynchronous message channels, 27
Asynchronous messaging, 71
AsyncRequestReplyService class, 408–409
Attach() method, 207–209, 212
Auction class, 276, 279–280
Auction versus distribution, 366–368
AuctionAggregate class, 276, 278–280
Auction-style scatter-gathers, 298
Axis server, 376
B
BAM (Business Activity Monitoring), 537
BankConnection class, 422–423
BankConnectionManager class, 438
BankGateway class, 426
BankName parameter, 410
BankQuoteGateway class, 379
BeginReceive method, 234, 292
Bid class, 276
Bidirectional channels, 100
Big-endian format, 12–13
Billing addresses, 30
BitConverter class, 12
BizTalk Mapper editor, 93
BizTalk Orchestration Manager, 320–321
Blocking gateways, 470–471
Body, 67
Body property, 68
BodyStream property, 68, 591
BodyType property, 68
BPEL4WS (Business Process Execution Language for Web Services), 318, 634
Bridges, 134–136
Broadcasting
document messages, 148
messages, 106–110
messages to multiple recipients, 298–300
Buffers, 286–288
Business applications, 1, 129
Business logic adapters, 129
Business object identifier, 166
Business tasks, 166
Business-to-business integration, 9
Byte streams, 12, 66
BytesMessage subtype, 68

C
C#
content-based routers, 233–234
delegates, 84
dynamic recipient lists, 256–258
dynamic routers, 246–248
filters, 76–77
routers, 83–84
smart proxies, 561–568
splitting XML order document, 262–267
CanHandleLoanRequest method, 422
Canonical Data Model pattern, 67, 90, 130, 355–360
Canonical data models, 20, 31, 113
ActiveEnterprise, 360
data format dependencies, 359
designing, 358–359
double translation, 358
indirection between data formats, 356
message bus, 140
multiple applications, 357
transformation options, 357–358
WSDL, 359–360
Canonical messages, 20
Chain of Responsibility pattern, 231, 308
Chaining
envelope wrappers, 332
gateways, 414, 472–473
request-reply message pairs, 166–167
transformations, 89–90
Change notification, 151
Channel Adapter pattern, 63, 86, 97, 102, 127–132, 134–135, 139
Channel Purger pattern, 572–575
Channels, 14–15, 20, 26, 57, 60–66, 359
acting like multiple channels, 63
adapters, 127–132
asynchronous, 27
bidirectional, 100
concurrent threading, 213
cost of, 63
crash proof, 101–102
data types, 101, 112, 220–221
datatyped, 111–114
dead letter, 119–121
dead messages, 101
decisions about, 101–102
defining for recipients, 250–251
deployment time, 62
design in Publish-Subscribe example, 219–222
designing, 62–63
determining set of, 100
dynamic, 100
for each aspect, 221
eliminating dependences, 327–328
FIFO (First-In, First-Out), 74
fixed set of, 99–100
hierarchical, 100
input, 107
invalid messages, 63, 101, 115–118
item number as address, 231
JMS, 64
message priorities, 113
message sequences, 172
message types, 78
messages, 15
mixing data types, 63
MSMQ, 63
Channels, continued
multiple consumer coordination, 508–509
multiple data types sharing, 113
multiple receivers, 103–104
names, 63
non-messaging clients, 102
notifying subscriber about event once, 106
number needed, 220–222
one-to-one or one-to-many relationships, 101–102
one-way, 154
output, 107
persistent, 63, 102
Pipes and Filters architecture, 72
planning, 61–62
point-to-point, 103–105
practical limit to, 63
preventing more than one receiver monitoring, 103
publish-subscribe, 106–110
quality-of-service, 113
queuing requests, 14
routing, 79
separating data types, 63–64
static, 99
subscribing to multiple, 108
subscribing to relevant, 237
themes, 99–100
TIB/RendezVous Transport, 448
transmitting units of data, 66
two-way, 154
unidirectional, 100
Channel-specific endpoints, 96–97
Channel-to-RDBMS adapters, 131
Child messages, 262
Claim Check pattern, 27, 90, 173, 346–351
Class diagrams, 88
Client-Dispatcher-Server pattern, 246
Clients, 62
   concurrently processing messages, 502
   non-messaging, 102
   transaction control, 484–485
CLR (Common Language Runtime), 110
Coarse-grained interfaces, 32
COBOL, 44
Collect data for later evaluation algorithm, 273
Collections and data types, 112
Command Message pattern, 23, 67, 104, 112, 117, 139, 143–147, 153, 156
   invoking behavior, 148
   JMS, 146
   loan broker system (ActiveEnterprise), 452
   routing, 140
   SOAP (Simple Object Access Protocol), 146
Commands, common structure of, 139
Commercial EAI tools
   channel adapters, 131
   content-based routers, 234–236
   Message Broker pattern, 82–83
   message brokers, 326
   message stores, 557
   Message Translator pattern, 445
   process Manager pattern, 445
   Common command structure, 139
   Common communication infrastructure, 139
Communications
   assumptions, 13–14
   availability of components, 14
   big-endian format, 12–13
   data formats, 14
   little-endian format, 12–13
   local method invocation, 10–11
   location of remote machine, 13
   loose coupling, 10
   platform technology, 13
   reducing assumptions about, 10
   strict data format, 13
   TCP/IP, 12
   tight coupling, 10
Communications backbone, 102
Competing Consumers pattern, 74, 97, 104, 172, 502–507
   JMS, 505–507
   processors, 289
Components
   decoupling, 72, 88–89
   dependencies between, 71
filtering out undesirable messages, 238
receiving only relevant messages, 237–238
two-way communication, 154
Composed Message Processor pattern, 25, 28, 227–228, 294–296
Composed routers, 225, 227–228
Composed service, 309–310
Composite messages, processing, 295–296
Computations and content enrichers, 339
ComputeBankReply method, 412
Computer systems
communications bus, 139
reliability, 124
ComputeSubject method, 235–236
Concurrency, 368–369
Concurrent threading, 213
Condense data algorithm, 273
Conflict resolution, 248
Consumers, 62, 515–516
Content, 111
Content Enricher pattern, 24–25, 90, 336–341
loan broker system, 363
loan broker system (ActiveEnterprise), 447
loan broker system (Java), 372
Content Filter pattern, 75, 90, 342–345
Content-Based Router pattern, 22–24, 81–82, 114, 225–226, 230–236
C#, 233–234
commercial EAI tools, 234–236
implementing functionality with filters, 240–242
knowledge about every recipient, 308
modifying for multiple destinations, 237–238
MSMQ, 233–234
reducing dependencies, 232–233
routing messages to correct validation chain, 303–305
routing messages to dynamic list of recipients, 249–250
routing rules associated with recipient, 308
special case of, 238
TIB/MessageBroker, 234–236
Context-Based Router, 82
ContextBasedRouter class, 594
Contivo, 93
Control Box pattern, 82
Control Bus pattern, 35, 407, 540–544
Control channels and dynamic routers, 244
ControlReceiver class, 594–595
Conway’s Law, 3
CORBA, 4, 10
Correlation
aggregators, 270
process managers, 315–316
Correlation Identifier pattern, 115, 143, 161, 163–169, 172, 197, 206
loan broker system (ActiveEnterprise), 457, 459–469
loan broker system (MSMQ), 405, 420–421, 439
replier, 195, 205
reply, 156
CorrelationId property, 195, 205
CreditAgencyGateway class, 379
CreditBureauGateway class, 476
CreditBureauGatewayImp class, 442
CreditBureauRequest struct, 414
CreditBureauReply struct, 418
Criteria and application integration, 39–41
CSPs (Communicating Sequential Processes), 75–76
Custom applications, sending and receiving messages, 127–128

D
Data
byte stream, 66
changes to, 50
frequent exchanges of small amounts of, 52
inconsistencies, 47
knowing where to send, 55–56
as message sequence, 171–179
moving between domain objects and infrastructure, 477–480
Data, continued
multiple interfaces to, 51
sharing, 53
storage schema and details easily changed, 54
storing in tree structure, 260–261
transferring between applications, 147–150
transformations, 327–329
transmitting large amounts, 170–171
units, 66
wrapping and unwrapping in envelope, 331–335
Data formats, 56
application integration, 40
developing content, 111
dependency, 359
designing for changes, 180–181
detecting, 353–354
distinguishing different, 180–181
evolution and extensibility, 40
foreign key, 181
format document, 181–182
format indicator, 181–182
integration, 16
internal, 16
minimizing dependencies, 355–356
not enforcing, 47
proprietary, 85
rate of change, 352
standardized, 85
transformations, 14
translation, 16, 86
translators, 353
version number, 181
Data models, proprietary, 85
Data packets, 55, 57
Data replication, 7, 31
Data Representation layer, 87–88, 90
Data sharing and latency, 40
Data structure content, 111
Data Structures layer, 87–88
Data transfer mechanism, 44, 54
Data types, 88
channels, 101, 112, 220–221
collections, 112
Datatype Channel pattern, 222
multiple sharing channel, 113
Data Types layer, 87
Database adapters, 129–130
Databases
adapters with content filters, 344–345
adding trigger to relevant tables, 129
changes to, 50
extracting information directly from, 129–130
performance bottleneck, 49
sharing, 47–49
suitable design for shared, 48
Datatype Channel pattern, 20, 63, 78, 101, 111–114, 139, 196, 353
data types, 220–222
Message Channel pattern, 115
request channel, 205
stock trading, 114
DCOM, 10
Dead Letter Channel pattern, 101, 117–121, 144
expired messages, 177
messaging systems, 120
Dead letter queue, 120
Dead message queue, 120
Dead messages, 101, 117–118, 120
Deadlocks, 49
Debugging Guaranteed Delivery pattern, 123–124
Decoupling, 88–89
DelayProcessor class, 288–290, 292
Detach() method, 207, 208
Detour pattern, 545–546
Detours, 545–546
Direct translation, 358
Dispatchers, 509–512
Java, 513–514
.NET, 512–513
Distributed environment and Observer pattern, 208–209
Distributed query message sequences, 173
Distributed systems
  asynchronous messaging, 54
  change notification, 151
Distribution versus auction, 366–368
DNS (Dynamic Naming Service), 13
Document binding, 375
Document/event messages, 153
Double translation, 358
Duplicate messages and receivers, 528–529
Durable Subscriber pattern, 108, 124, 522–527
  JMS, 525–527
  observers, 213
  stock trading, 125, 525
Dynamic channels, 100
Dynamic Recipient List pattern, 34, 252–253
  C#, 256–258
  MSMQ, 256–258
Dynamic Router pattern, 226, 233, 242–248
Dynamic routing slips, 309
DynamicRecipientList class, 256, 258

E
EAI (Enterprise Application Integration)
  applications operating independently, 137–138
  one-minute, 11
  process manager component, 317–318
  suites, 2–3
  ebXML, 85
E-mail
  data as discrete mail messages, 67
Encapsulation, 50–52
  reply-to field, 161
  encodingStyle attribute, 374
Endpoints, 19, 58, 62, 84
  channel-specific, 96–97
  customizing messaging API, 96
  encapsulating messaging system from rest of application, 96
  message consumer patterns, 464–466
  message endpoint themes, 466–467
  send and receive patterns, 463–464
  sending and receiving messages, 95–97
  transactional, 84
EndReceive method, 204, 292
Enterprises
  challenges to integration, 2–4
  loose coupling, 9–11
  need for integration, 1
  services, 8
Entity-relationship diagrams, 88
Envelope Wrapper pattern, 69, 90, 330–335
  adding information to raw data, 332
  chaining, 332
  headers, 331–332
  postal system, 334–335
  process of wrapping and unwrapping messages, 331
  SOAP messages, 332–333
  TCP/IP, 333–334
Envoy Connect, 136
EnvoyMQ, 131
ERP (Enterprise Resource Planning)
  vendors, 1
Errors, 117
Event Message pattern, 67, 123, 143, 148, 151–153, 156
  Observer pattern, 153
  Publish-Subscribe Channel, 108
Event-Driven Consumer pattern, 77, 84, 97, 498–501
  aggregators, 278
  gateways, 212
  JMS MessageListener interface, 500–501
  loan broker system (MSMQ), 417–418
  .NET ReceiveCompletedEventHandler delegate, 501
  pull model, 217
  replier, 195, 204
Event-driven gateways, 471–472
Events
  content, 152
  Guaranteed Delivery pattern, 152
  Message Expiration pattern, 152
  notify/acknowledge, 156
Events, continued
notifying subscriber once about, 106
timing, 152
transmitting between applications, 151–153
Exceptions, 156, 473
Expired messages, 176–179
External event strategy, 273
External packages and schemas, 49

F
FailOverHandler class, 599–600
FIFO (First-In, First-Out) channels, 74
File formats, standard, 44
File transfer, 33, 41, 43–46
File Transfer pattern, 50, 147
decoupling, 53–54
multiple data packets, 53–54
not enforcing data format, 47
reacting to changes, 50
sharing data, 47, 53
Files, 44–45
Filtering
built-in messaging system functions, 239–240
messaging, 71
reactive, 233
splitters, 344
Filters, 58, 71–72, 226, 238
aggregators, 269–270
combining, 227
composability, 312
connection with pipes, 72
decoupling, 79
directly connecting, 78
eliminating messages not meeting criteria, 226
generic, 75
implementing router functionality, 240–242
loan broker system, 367
multiple channels, 72–73
parallelizing, 74
versus recipient lists, 254–255
sequence of processing steps as independent, 301–302
single input port and output port, 72
stateful, 239
stateless, 239
Fine-grained interfaces, 32
First best strategy, 273
Fixed routers, 81
Foreign key, 181
Format document, 181–182
Format Indicator pattern, 112, 114, 180–182
Formatter property, 234

G
Gateways, 469
abstracting technical details, 403
asynchronous loan broker gateway
(MSMQ), 475–476
blocking, 470–471
chaining, 414, 472–473
event-driven, 471–472
Event-Driven Consumer pattern, 212
exceptions, 473
generating, 473–474
between observer and messaging system, 212
pull model, 215–217
sending replies, 217–218
between subject and messaging system, 211
testing, 475
generateGUID method, 457
Generic filters, 75
getaddr method, 93
GetCreditHistoryLength method, 441–442
GetCreditScore method, 414, 420, 441–442, 477
GetLastTradePrice method, 146, 150
GetRequestBodyType method, 407, 409, 412
GetState() method, 207–209, 214, 218
getStateRequestor method, 218, 219
GetTypedMessageBody method, 407
Guaranteed delivery
built-in datastore, 123
debugging, 123–124
events, 152
large amount of disk space, 123
redundant disk storage, 124
stock trading, 124–125
testing, 123–124
WebSphere MQ, 126
Guaranteed Delivery pattern, 102, 113, 122–126, 176
GUIDs (globally unique identifiers), 285
GUIs and message bus, 140

H
Half-Sync/Half-Async pattern, 472, 534
Header, 67
Hierarchical channels, 100
Host Integration Server, 135–136
HTTP Web services, 51
Hub-and-spoke architecture, 228, 313–314, 324–326

I
ICreditBureauGateway interface, 442
Idempotent Receiver pattern, 97, 528–531
Idempotent receivers, 252, 529–531
IMessageReceiver interface, 403, 442
IMessageSender interface, 403, 442
Incoming messages output channel criteria, 81
Information Portal scenario, 32
Information portals, 6
Initialized aggregators, 274
Integration
application, 39–56
big-endian format, 12–13
broad definition, 5
business-to-business, 9
challenges, 2–4
channels, 14–5
data formats, 16
data replication, 7
distributed business processes, 8–9
distributed XML Web services standards, 4
far-reaching implications on business, 3
existing XML Web services standards, 4
information portals, 6
limited amount of control over participating applications, 3
little-endian format, 12–13
location of remote machine, 13
loosely coupled solution, 15–16
message-oriented middleware, 15
messages, 15
middleware, 15
need for, 1–2
patterns, 4–5
redundant functionality, 7
remote data exchange into semantics as local method call, 10
removing dependencies, 14–15
routing, 16
semantic differences between systems, 4
shared business functions, 7–8
significant shift in corporate politics, 3
skill sets required by, 4
SOAs (service-oriented architectures), 8
standard data format, 14
standards, 3–4
systems management, 16
tightly coupled dependencies, 11–14
user interfaces, 129
Integrators and files, 44–45
Interfaces, 32
loan broker system (Java), 371–372
Internal data formats, 16
Invalid application request, 117
Invalid Message Channel pattern, 101, 115–118, 196–197, 205–206
loan broker system (MSMQ), 405
messages out of sequence, 172
queues, 233
Invalid messages, 23, 63, 101, 115–118, 120
application integration problems, 117
ignoring, 116
JMS specification, 118
monitoring, 117
receiver context and expectations, 117
receivers, 120
Request-Reply example, 196–197
stock trading, 118
InvalidMessenger class, 196, 205
Inventory Check message, 26
Inventory systems, 22–23
IsConditionFulfilled method, 84
Iterating splitters, 260–261
Iterator, 261
**J**

**J2EE**
- EJBs (Enterprise JavaBeans), 535
- messaging systems, 64

**Java**
- dispatchers, 513–514
- document messages, 149
- event messages, 152
- loan broker system, 371–400

**Java RMI**, 10

**JAX-RPC specification**, 375

**JMS (Java Messaging Service)**
- aggregators, 276–282
- channel purgers, 574–575
- channels, 64
- command message, 146
- competing consumers, 505–507
- correlation identifiers, 167
- Correlation-ID property, 167
- document messages, 148
- Durable subscribers, 525–527
- event messages, 152
- expired messages, 178
- invalid messages, 118
- mappers, 483
- message selector, 521
- message sequences, 174
- MessageListener interface, 500–501
- messages, 68
- multiple message systems, 133
- persistent messages, 125–126
- point-to-point channels, 104–105
- producer and consumer, 97
- Publish-Subscribe example, 207–208
- Publish-Subscribe Channel pattern, 109, 124, 186
- receive method, 496
- Reply-To property, 161
- requestor object, 157–158
- Request-Reply example, 118, 187–197
- Request/Reply pattern, 157–158
- return addresses, 161
- Time-To-Live parameter, 178
- transacted session, 489

**JndiUtil JNDI identifiers**, 191

**JWS (Java Web Service) file**, 378

**K**

**Kahn Process Networks**, 74

**Kaye, Doug**, 9

**L**

**Large document transfer message sequences**, 173

**Legacy application routing slips implementation**, 306

**Legacy platform and adapters**, 131

**LenderGateway class**, 379

**Listens**, 62

**little-endian format**, 12–13

**Loan broker system**
- ActiveEnterprise, 445–462
- addressing, 366–368
- aggregating strategies, 368
- Aggregator pattern, 363, 368
- aggregators, 275
- asynchronous timing, 364–366
- bank component, 578
- Content Enricher pattern, 363
- control buses, 544
- credit bureau component, 578
- credit bureau failover, 579, 592–595
- designing message flow, 362–364
- distribution versus auction, 366–368
- enhancing management console, 595–602
- instrumenting, 578–579
- Java, 371–400
- loan broker component, 578
- loan broker quality of service, 578–587
- management console, 578, 579
- managing concurrency, 368–369
- Message Channel pattern, 367–368
- Message Filter pattern, 367
- Message Translators pattern, 364
- MSMQ, 401–444
- normalizer pattern, 364
- obtaining loan quote, 361–362
- patterns, 363
- Point-to-Point pattern, 368
- process managers, 320
- Publish-Subscribe Channel pattern, 363, 366–368
- Recipient List pattern, 366–367
recipient lists, 256
Scatter-Gather pattern, 363, 366
scatter-gatherers, 299
Selective Consumer pattern, 367
sequencing, 364–366
synchronous implementation with Web
services, 371–400
synchronous timing, 364–366
system management, 577–602
test client component, 578
verifying credit bureau operation, 579,
587–592
wire taps, 549
XML Web services, 371–400
Loan broker system (ActiveEnterprise)
Aggregator pattern, 446–447, 458
architecture, 445–447
Command Message pattern, 452
Content Enricher pattern, 447
Correlation Identifier pattern, 457,
459–460
design considerations, 455
execution, 460–461
implementing synchronous services,
452–454
interfaces, 451–452
managing concurrent auctions,
459–460
Message Translator pattern, 457–458
process model implementation,
456–459
Publish-Subscribe pattern, 446
Request-Reply pattern, 446, 452
Return Address pattern, 452
Loan broker system (Java), 379–381
accepting client requests, 378–384
Apache Axis, 376–378
Bank1.java file, 393–394
Bank1WS.jws file, 395
Bank.java file, 391–392
BankQuoteGateway.java file, 390–391,
396
client application, 396–397
Content Enricher pattern, 372
CreditAgencyGateway.java file,
385–386
CreditAgencyWS.java file, 386–388
implementing banking operations,
394–3945
interfaces, 371–372
JWS (Java Web Service) file, 378
LenderGateway.java file, 389–390
Message Translators pattern, 372
Normalizer pattern, 372
obtaining quotes, 388–3889
performance limitations, 399–400
Recipient List pattern, 372
running solution, 397–399
Service Activator pattern, 372, 379
service discovery, 379
solution architecture, 371–372
Web services design considerations,
372–376
Loan broker system (MSMQ), 401
accepting requests, 428–431
Aggregator pattern, 402, 422, 424
bank design, 410–412
bank gateway, 421–428
Bank.cs file, 411–412
base classes, 405–409
Control Bus pattern, 407
Correlation Identifier pattern, 405,
420–421, 439
credit bureau design, 412–413
credit bureau gateway, 414–421
CreditBureau.cs file, 413
CreditBureauGateway.cs file, 418–420
designing, 413–431
Event-Driven Consumer pattern,
417–418
external interfaces, 401–402
IMessageSender.cs file, 403–404
improving performance, 435–540
Invalid Message Channel pattern, 405
limitations, 443–444
LoanBroker.cs file, 430–431
Message Translator pattern, 402
message types for bank, 410
Messaging Gateway pattern, 402–405
MQService.cs file, 406–409
Process Manager pattern, 402, 434
Recipient List pattern, 402, 422,
424–425
refactoring, 431–434
Loan broker system (MSMQ), continued
  Return Address pattern, 405
  Scatter-Gather pattern, 402, 422
  Service Activator pattern, 412
testing, 440–443
LoanBroker class, 428–431
LoanBrokerPM class, 433–434
LoanBrokerProcess class, 432–433
LoanBrokerProxy class, 582–583
LoanBrokerProxyReplyConsumer class, 584–585
LoanBrokerProxyRequestConsumer class, 584
LoanBrokerWS class, 379
Local invocation, 145
Local method invocation, 10–11
Local procedure calls, 52
Logical entities, 88
Loose coupling, 9–11

M
ManagementConsole class, 597–598
MapMessage subtype, 68
Mapper pattern, 480
Mapper task, 457–458
Mappers, 480–483
match attribute, 93
MaxLoanTerm parameter, 410
Mediator pattern, 509
Mediators, 481
Message Broker pattern, 228, 322–326
brokering between applications, 82–83
central maintenance, 324
commercial EAI tools, 82–83, 326
hierarchy, 325
stateless, 324–325
translating message data between applications, 325–326
Message bus, 102, 139–141
Message Bus pattern, 64, 137–141
Message Channel pattern, 19, 55, 57, 62,
73, 78, 106
Apache Axis, 377
availability, 100
Datatype Channel pattern, 115
decisions about, 101–102
decoupling applications, 89
fixed set of, 99–100
load-balancing capabilities, 82
loan broker system, 367–368
monitoring tool, 108
as pipe, 66
security policies, 108
unidirectional or bidirectional, 100
Message class, 68
Message Consumer patterns, 464–466
Message Dispatcher pattern, 97, 113,
508–514
Message dispatchers, 172
Message Endpoint pattern, 56, 58, 61,
173
Apache Axis, 376
data format translation, 86
Selective Consumer pattern, 226
Message endpoints, 16, 62, 95–97,
134–135
Message Expiration pattern, 67, 108, 119,
123, 144, 176–179
Message Filter pattern, 75, 80, 237–242
Publish-Subscribe Channel pattern, 226
Message History pattern, 81, 551–554
Message ID, 166
Message identifiers, 285
Message pattern, 57–58, 78
Message Router pattern, 34, 58, 75, 89,
139, 225, 228
capabilities, 139
Content-Based Router pattern, 232
Message Filter pattern, 238
Message Sequence pattern, 67, 115, 144,
156, 170–175
Message sequences, 171–179
channels, 172
Competing Consumers pattern, 172
distributed query, 173
dern indicator field, 171
identification fields, 171
identifiers, 167
JMS, 174
large document transfer, 173
Message Dispatcher pattern, 172
multi-item query, 173
.NET, 174
position identifier field, 171
Request-Reply pattern, 172
sending and receiving, 172
sequence identifier field, 171
size field, 171
Message Store pattern, 555–557
Message stores, 26–27, 34, 556–557
Message Translator pattern, 58
Channel Adapters, 130
commercial EAI products, 445
data in incoming message, 336
loan broker system, 364
loan broker system (ActiveEnterprise),
457–458
loan broker system (Java), 372
loan broker system (MSMQ), 402
metadata, 130
MessageConsumer class, 191, 278,
562–563
MessageConsumer type, 97
MessageGateway, 414
message-id property, 195, 205
MessageListener interface, 195, 212, 217,
500–501
Message-oriented middleware, 15
Message-processing errors, 117
MessageProducer class, 125, 191, 195
MessageProducer type, 97
MessageQueue class, 97, 105, 126, 167,
201, 204
MessageQueue instance, 65
MessageReceiverGateway class, 404
Messages, 14–15, 66, 159
aggregating, 24
applications, 67
application-specific, 20
augmenting with missing information,
338–341
authentication information, 70
body, 67
breaking data into smaller parts, 67
broadcasting, 106–110
canonical, 20
channels, 78
checking in data for later use, 27
collecting and storing, 269–270
combining related to process as whole,
268–269
common format, 86
conforming to data types, 101
containing commands, 146
contents are semantically incorrect, 117
correlation ID, 166
data formats, 56
data packets, 57
dead, 101, 117–118
decoupling destination of, 322–323
decrypting, 70–71
delivering, 57–58
demultiplexing, 113
destination of, 80
different types of, 67
directing, 56
document/event, 153
duplicate, 70
elements requiring different processing,
259–260, 294–295
encrypted, 70
endpoints, 58
expired, 176–179
format data, 67
formatting in proprietary formats, 31
guaranteed delivery, 122–126
header, 67
high frequency of, 55
huge amounts of data, 144
improper datatype or format, 115
“incoming message massaging module,”
70
intent, 143
invalid, 101, 115–118, 120
JMS, 68
large amounts of data, 170–171
message ID, 166
messaging system, 67
missing properties, 115
monitoring, 34–36
multiple recipients with multiple replies,
297
.NET, 68
order ID, 24–25
out-of-sequence, 227, 283–284
peek functions, 108
persistent, 122–126
private, 358
Messages, continued
processing in type-specific ways, 113–114
processing steps, 71
public, 358
recombining, 226–227
recursive nature, 69
reducing data volume, 346
removing unimportant data from, 343–345
removing valuable elements from, 342–343
reordering, 284–293
response, 143–144
retry timeout parameter, 123
return address, 161
routing, 58, 80, 85
routing slips, 305–306
routing to correct recipient based on content, 232–236
semantically equivalent in different format, 352–353
sending and receiving, 95–97
sent time, 178
sequence numbers, 285
simplifying structure, 343
slow, 144
SOAP, 68–69
splitting, 24, 226, 260–267
state reply, 153
state request, 153
storing data between, 28
storing data in central database, 27
storing data in tree structure, 260–261
testing, 34–36
timestamp, 177–178
transformation, 54, 58, 327–329
transformation levels, 87–88
two-way, 154
types, 68, 78
unable to deliver, 118–121
update, 153
Wire Tap, 27
MessageSenderGateway class, 404
Messaging, 41, 53–56
asynchronous, 54, 71
basic concepts, 57–58
filtering, 71
invoking procedure in another application, 145–146
one-way communication, 154
remote procedure invocation, 156
remote query, 156
transferring data between applications, 147–150
transmitting discrete units of data, 66
Messaging API, 96
Messaging Bridge pattern, 102, 131, 133–136
Messaging Gateway pattern, 19–20, 72, 97, 117, 211, 468–476
loan broker system (MSMQ), 402–405
Messaging Mapper pattern, 97, 477–483
Messaging mappers, 357–358
Messaging pattern, 45–46, 49, 52, 57–58, 163
Messaging server, applications as clients of, 95–96
Messaging services, 8
dynamic discovery, 245
invoking with messaging and non-messaging technologies, 532
request-reply, 28–29
reuse, 29
shared business functions as, 28
Messaging systems
adapters, 102
applications communicating with, 60–66
built-in datastore, 123
channel adapters, 63
communicating without required data items, 336–338
communications backbone, 102
connecting application to, 56
connecting multiple, 133–136
connections, 60–61
Dead Letter Channel, 120
decoupling, 54
delivering messages, 57–58
encapsulating access to, 468–469
filtering built-in functions, 239–240
filters, 58
hierarchical channel-naming scheme, 63
implementation of single function spread across, 230–232
inconsistency, 55
interoperability, 133–134
invalid messages, 330
J2EE, 64
logical addresses, 61
managing channels, 95
messages, 67
pipes, 58
Pipes and Filters architecture, 70–77
planning channels, 61–62
receivers inspecting message properties, 79
reducing data volume of messages, 346
sending and receiving messages, 95–97
specific messaging requirements, 330–331
store-and-forward process, 122
uneconomical or impossible to adjust
components, 79
valid messages, 330
WebSphere MQ for Java, 64–65
Metadata
management and transformations, 328–329
Message Translators pattern, 130
Metadata adapter, 130–131
MetricsSmartProxy class, 566
Meunier, Regine, 74
Middleware, 15
MIDL (Microsoft Interface Definition Language), 531
Missing messages
aggregators as detector of, 275–276
stand-in messages for, 287–288
MockQueue, 404
Model-View-Controller architecture, 151
Monitor class, 589–592
MonitorStatusHandler class, 598
MQSend class, 288
MQSequenceReceive class, 289
MQService class, 405–409, 412
MSMQ (Microsoft Messaging Queuing Service)
asynchronous loan broker gateway, 475–476
bridges, 135–136
content-based routers, 233–234
distribution lists, 110
dynamic recipient lists, 256–258
dynamic routers, 246–248
filters, 76–77
loan broker system, 401–444
maximum message size, 173
message channels, 65
multiple-element format names, 110
one-to-many messaging model, 109
persistent channels, 124
queues, 65
real-time messaging multicast, 109
resequencers, 288–293
routers, 83–84
smart proxies, 561–568
splittering order document, 264–267
Transactional Clients pattern, 124
transactional filter, 490–493
Multi-item queries and message sequences, 173
Multiple asynchronous responses, 174
Multiplexing, 113

N
.NET
CLR (Common Language Runtime), 110
correlation identifiers, 167–168
Correlation-Id property, 167–168
delegates, 418
dispatchers, 512–513
document messages, 148
event messages, 152
expired messages, 179
message sequences, 174
MessageQueue class, 97
messages, 68
persistent messages, 126
point-to-point channels, 105
Receive method, 496–497
ReceiveCompletedEventHandler
delegate, 501
Request-Reply example, 118, 198–206
resequencers, 288–293
Response-Queue property, 162
return addresses, 162
.NET, continued
  selective consumers, 521
  serialization and deserialization, 416
  Time-To-Be-Received property, 179
  Time-To-Reach-Queue property, 179
  transactional queue, 490
.NET Framework, 404–405
.NET Framework SDK, 415
.NET Remoting, 10
Networks, inefficiencies and recipient lists,
  253–254
Neville, Sean, 375
New Order message, 22, 27, 30
Normalizer pattern, 90, 352–354
  loan broker system, 364
  loan broker system (Java), 372
Normalizers, 353–354
Notify() method, 207, 208, 211, 213
notifyNoState() method, 217
Null Object, 238

O
OAGIS, 85
ObjectMessage class, 196
ObjectMessage subtype, 68
Objects, notifying dependents of change,
  207–208
Observer pattern, 106, 110, 151
  distributed environment, 208–209
  Event Message pattern, 153
  implementing, 209–212
  JMS Publish-Subscribe example,
    207–208
  .NET Framework, 404
  pull model, 153
  push model, 153
ObserverGateway class, 212, 218
Observers, 207–208
  concurrent threading, 213
  Durable Subscriber pattern, 213
  implementing, 209–213
  losing notification, 209
  multiple aspects, 219
  receiving messages, 213
  reply channels, 214–215
  subscribing and unsubscribing from
    channels, 213
OnBestQuote method, 431
OnCreditReply method, 431
OnCreditReplyEvent delegate, 476
OnCreditReplyEvent event, 420
One-minute EAI (Enterprise Application
  Integration) suites, 11
One-way channels, 154
OnMessage event, 404
onMessage method, 84, 195, 197, 212,
  217, 234, 264, 278, 407–408
OnMsgEvent delegate, 404
OnReceiveCompleted method, 77, 204
Operating environment, 339
ORB (object request broker) environment,
  208
Order ID, 24–25
Order Item Aggregator, 25
Order Item messages, 24–25
Order message, 24–25, 263
Ordered or unordered child messages,
  262
Orders, checking status, 26–29
Out-of-order messages, 283–284
Out-of-sequence messages, 227

P
Parallelizing filters, 74
Pattern matching, 93
Patterns
  combining with scatter-gatherers,
    299–300
  comparing Process Manager pattern
    with, 319–320
  loan broker system, 363
  pattern form, xlii–xlvi
 Peek functions, 108
  PeekByCorrelationId() method, 168
 Persistence, 123
 Persistent channels, 63, 102, 126
 Persistent messages, 122–126
  JMS, 125–126
 .NET, 126
 Persistent recipient lists, 252
 Persistent store, 29, 347–348
 PGM (Pragmatic General Multicast),
  109
 Pipeline processing, 73–74
INDEX 675

Pipes, 58
abstract, 72
composability, 312
connection with filters, 72
managing state, 316
Message Channel pattern, 66
simple in-memory queue to implement, 72
Pipes and Filters architecture, 58
directly connecting filters, 78
history of, 74–75
large number of required channels, 72
Pipes and Filters pattern, 227
chaining transformations, 89
composability of individual components, 79
composability of processing units, 312
distributed, 317
pipeline processing, 73–74
processing messages, 73–74
processing steps, 230
sequence of processing steps as independent filters, 301–302
testability, 73
Point-to-Point Channel pattern, 63, 73–74, 101, 124, 147, 368
Point-to-Point channels, 20, 23, 26–27, 103–105
broadcasting messages, 153
command messages, 146
document messages, 148
eavesdropping, 107–108
inspecting messages, 547–550
JMS, 104–105
.NET, 105
request channel, 155
stock trading, 104
Polling Consumer pattern, 97, 155, 494–497
Port, 72
Postal service
data as discrete mail messages, 67
envelope wrappers, 334–335
Predictive routing, 80
Private messages, 358
Procedures, invoking in another application, 145–146
Process definitions, 315
process managers creation of, 317–318
TIB/IntegrationManager Process Manager Tool, 449
Process instances, 28
process managers, 314–315
TIB/IntegrationManager Process Manager Tool, 449
Process Manager pattern, 312–321
commercial EAI products, 445
comparing with other patterns, 319–320
loan broker system (MSMQ), 402, 434
Process managers, 27–31, 309, 313
BizTalk Orchestration Manager, 320–321
central, 317
claim checks, 350–351
correlation, 315–316
hub-and-spoke pattern, 313–314
keeping state in messages, 316–317
loan broker, 320
process definition, 315, 317–318
process instances, 314–315
state maintenance, 314
storing intermediate information, 314
trigger message, 313
versatility, 314
Process method, 77
Process template, 28
Processes
marshaling and unmarshaling data, 66
passing piece of data, 66
synchronizing with IO (input-output), 75
Processing
composite messages, 295–296
orders, 20–23
Processing pipeline, 73–74
Processor class, 76, 290, 292
Processors competing with consumers, 289
Producers, 62
Protocols, tunneling, 330
Provider, 62
Public messages, 358
Publisher, 62
Publish-Subscribe Channel pattern, 62–63, 80, 101, 104, 139, 147, 153, 207, 209
loan broker system (ActiveEnterprise), 446
Publish-subscribe channels, 23, 26, 31, 33–34, 106–110, 249–250
announcing address changes, 220
basic routing, 323
as debugging tool, 107
document messages, 148
eavesdropping, 107–108
Event Message pattern, 108
filters versus recipient lists, 254–255
hierarchical structure, 239
implementing router functionality with filters, 240–242
JMS, 109, 124, 186
loan broker system, 363, 366–368
Message Filters pattern, 226
multiple output channels, 107
one input channel, 107
out-of-product announcements, 220
receiving change notification code, 211–212
request channel, 155
Scatter-Gather pattern, 228
special wildcard characters, 108
stock trading, 108–109
storing messages, 108
subscription to, 237
Publish-Subscribe example
channel design, 219–222
code to announce change, 210–211
Command Message pattern, 185
comparisons, 212–213
Datatype Channel pattern, 185
distributed notification between applications, 212–213
Document Message pattern, 185
Durable Subscriber pattern, 185
Event Message pattern, 185
Event-Driven Consumer pattern, 185
implementing observers, 209–212
Java using JMS, 186
Messaging Gateway pattern, 185
Observer pattern, 185
Publish-Subscribe Channel pattern, 185
pull model, 213–219
push model, 213–219
Request-Reply pattern, 185
Return Address pattern, 185
serialization, 213
Pull model, 153, 207–208
Event-Driven Consumer pattern, 217
gateways, 215–217
Publish-Subscribe example, 213–219
PullObserverGateway class, 218
PullSubjectGateway class, 217
Push model, 153, 207–208
Publish-Subscribe example, 213–219
Q
Quality-of-Service Channel pattern, 113
Queries, 173
Queue instance, 64
Queue interface, 104
QueueRequestor class, 192
Queues
Invalid Message Channel pattern, 233
peek functions, 108
R
RatePremium parameter, 410
Reactive filtering, 80, 233
ReceiveByCorrelationID() method, 168
ReceiveCompleted event, 404
ReceiveCompletedEventHandler class, 501
ReceiveCompletedEventHandler delegate, 501
Receive() method, 192, 202
Receivers, 62
communicating message type to, 112
content data structure and data format, 111
dead messages, 120
duplicate messages, 528–529
Event-Driven Consumer pattern, 97
idempotent receivers, 529–531
inspecting message properties, 79
invalid messages, 120
multiple on channel, 103–104
Polling Consumer pattern, 97
response from, 154–158
type of messages received, 111
ReceiveSync() method, 192, 202
Receiving sequences, 172
Recipient List pattern, 110, 226, 249–258
loan broker system (Java), 372
loan broker system (MSMQ), 402, 422, 424–425
Recipient lists, 242, 250–251
dynamic, 252–253
idempotent receivers, 252
list of recipients, 251
loan broker, 256
network inefficiencies, 253–254
persistent, 252
versus publish-subscribe channels and filters, 254–255
restartable, 252
robustness, 252
routing, 439
scatter-gathers, 298
sending copy of message to all recipients, 251
sending preferences to, 253
single transaction, 252
Recipients
broadcasting messages to multiple, 298–300
defining channel for, 250–251
list of, 251
multiple with multiple replies, 297
routing messages to dynamic list, 249–250
sending copy of message to all, 251
Recombining messages, 226–227
Redundant functionality, 7
Relational databases, SQL-based, 48
Relationships and entities, 88
Reliability of Web services, 375–376
Remote invocation, 145
Remote Procedure Call pattern, 209
Remote procedure calls, 52
Remote Procedure Invocation pattern, 46, 49, 62, 145, 147, 151
Remote procedure invocations, 41
failure of, 53–54
messaging, 156
sharing functionality, 53
synchronous, 163
two-way communication, 147
Remote query and messaging, 156
Web services, 375
Reordering messages, 284–293
Replier class, 183, 187, 198
Repliers, 155
agreeing on details, 165
correlation identifier, 164
Correlation Identifier pattern, 195, 205
Event-Driven Consumer pattern, 195, 204
Return Address pattern, 195, 204
Replies
callback processor to process, 160
correlation identifier, 164–169, 165
Correlation Identifier pattern, 156
document messages, 148
exceptions, 156
gateway sending, 217–218
from multiple recipients, 297
one-to-one correspondence with request, 159
pointer or reference to request, 164
processing, 195
reassembling multiple into one, 228
result value, 156
return address, 159–162
token, 166
void, 156
where to send, 159–162
which requests they are for, 163–169
Reply channels and observers, 214–215
Request channel, 155, 205
Requestor class, 183, 187, 198
Requestor:ReceiveSync() method, 197
Requestors, 62, 155
agreeing on details, 165
callback processor to process replies, 160
correlation identifier, 164
map of request IDs and business object IDs, 166
Requestors, continued
receiving reply messages, 192, 202
sending request messages, 192, 202
Request-replies
asynchronous callback, 155–156
chaining message pairs, 166–167
channels to transmit messages, 214
loan broker system (ActiveEnterprise), 446, 452
message sequences, 172
replier, 155
requestor, 155
synchronous block, 155
Request-Reply example
Command Message pattern, 188
Correlation Identifier pattern, 184, 189, 200
Datatype Channel pattern, 184
Document Message pattern, 184, 188
Event Driven Consumer pattern, 184
Invalid Message Channel pattern, 184
Invalid Message example, 196–197, 205–206
JMS, 187–197
JMS API in Java J2EE, 184
jms/InvalidMessages queue, 187
jms/ReplyQueue queue, 187, 188
jms/RequestQueue queue, 187
Message Channel pattern, 184
MSMQ API in Microsoft .NET using C#, 184
.NET, 198–206
Point-to-Point Channel pattern, 184
Polling Consumer pattern, 184
.private$InvalidQueue queue, 198
.private$ReplyQueue queue, 198
.private$RequestQueue queue, 198
Replier class, 183, 187, 198
Requestor class, 183, 187, 198
Request-Reply code, 189–196, 200–205
Request-Reply pattern, 184
Return Address pattern, 184, 188–189, 199, 204
JMS, 157–158
reply channel, 100
RequestReplyService class, 408–409, 412, 424
Request-Response Message Exchange pattern
return addresses, 162
SOAP 1.2, 157, 162, 168–169
Web services, 162, 168–169
Requests
correlation identifier, 156
messaging query, 156
notify/acknowledge messages, 156
pointer or reference to, 164
remote procedure invocation messages, 156
Return Address pattern, 100, 156
return addresses, 167, 195
sent and received timestamps, 199
unique ID, 166
which replies are for, 163–169
Resequencer class, 289
Resequencer pattern, 74, 164, 227, 283–293
Resequencers, 227, 284
avoiding buffer overrun, 286–288
buffers, 286
internal operations, 285–286
MSMQ, 288–293
.NET, 288–293
out-of-sequence messages, 285–286
sequence numbers, 285
stand-in messages for missing messages, 287–288
throttling message producer with active acknowledgment, 287
ResponseQueue property, 202
Responses, 143–144
aggregating to single message, 298–300
delivered out of order, 268
from receivers, 154–158
Retry timeout parameter, 123
Return Address pattern, 115, 143, 159–162
loan broker system (ActiveEnterprise), 452
loan broker system (MSMQ), 405
replier, 195, 204
request message, 100
requests, 156
Return addresses, 29, 35, 36, 159–162
  JMS, 161
  .NET, 162
Request-Response Message Exchange pattern, 162
requests, 167
RIP (Routing Information Protocol), 245
RMI (Remote Method Invocation), 10
RosettaNet, 85
Router slips, 308–309
Routers, 25, 56, 58, 73, 78–84, 140, 359
  abuse of, 81
  architectural patterns, 225, 228
  avoiding dependency, 243
  built-in intelligence, 82
  C#, 83–84
  combining variants, 228
  composed, 225, 227–228
  content-based, 81–82, 225–226,
  230–236
  context-based, 82
  Control Bus pattern, 82
  decoupling filters, 80
  degrading performance, 81
  destination based on environment conditions, 82
  destination of message, 80–82
  dynamic, 244–248
  eliminating dependencies, 327–328
  filters, 238
  fixed, 81
  fixed rules for destination of in-coming message, 226
  hard-coded logic, 82
  implementing functionality with filters, 240–242
  knowledge of all destination channels, 80
  loosely coupled systems, 81
  maintaining efficiency, 243
  maintenance bottleneck, 80
  MSMQ, 83–84
  multiple in parallel, 81
  parallel processing, 82
  performance bottleneck, 81
  selecting correct for purpose, 228–229
  self-configuring, 244–248
  simple, 225–227
  stateful, 82, 227
  stateless, 82, 233
  variants, 81–82
Routing, 16, 58
  basic form, 79
  channels, 79
  command messages, 140
  to correct recipient based on content,
  232–236
  flexibility, 302
  maintaining state of sequence, 313–321
  message flow efficiency, 302
  moving logic to middleware layer, 323
  recipient lists, 439
  resource usage efficiency, 302
  simple maintenance, 302
  unknown non-sequential processing steps, 312–313
Routing messages, 80, 85
  based on criteria, 226
  to correct translator, 353–354
  to dynamic list of recipients, 249–250
  with multiple elements, 259–260
  for system management, 545–546
  through series of unknown steps, 301–305
Routing Slip pattern, 301–311
Routing slips
  acting as chain of responsibility, 308
  binary validation steps, 307
  as composed service, 309–310
  decision postponed until end, 307
  dynamic, 309
  legacy application implementation, 306
  limitations, 306
  processing steps, 312
  stateless transformation steps, 307
  WS-Routing (Web Services Routing Protocol), 310–311
RPC (Remote Procedure Call), 10, 51, 103
  asynchronous messaging, 122
  binding, 375
  marshaling, 66
RPC-style SOAP messaging, 149
RPC-style Web services, 10
Run method, 412
S
SASE (Self-Addresses Stamped Envelope) pattern, 219
Scatter-Gather pattern, 228, 297–300
loan broker system, 363, 366
loan broker system (ActiveEnterprise), 446
loan broker system (MSMQ), 402, 422
Publish-Subscribe Channel pattern, 228
Scatter-gatherers, 298–300
Schemas, 49
Security and Web services, 375–376
Selecting best answer algorithm, 273
Selective Consumer pattern, 63, 119, 168, 222, 226, 239–240, 515–521
JMS message selector, 521
loan broker system, 367
.NET, 521
separating types, 520
Selectors, 239–240
Semantic dissonance, 47, 54–55
Semantic enrichment, 414
Send and receive patterns, 463–464
SendConsecutiveMessages method, 290
Senders, 62
communicating message type to receiver, 112
decoupling message destination from, 322–323
Send() method, 192, 202
SendReply method, 408, 433
Sent time, 178
Sequence identifier, 172
Sequence numbers, 285
Sequence, 261
Sequencing, 364–366
Serializable command object, 146
Serialization in Publish-Subscribe example, 213
Service Activator pattern, 97, 117, 139, 532–535
Service activators, 140, 533–534
Axis server, 376
loan broker system (Java), 379
loan broker system (MSMQ), 412
Service stubs, 403
Service-oriented architecture (SOA), 8, 140
Shared business functions, 7–8
Shared Database pattern, 46–50, 147
Shared databases, 29, 41, 53
avoiding semantic dissonance, 55
unencapsulated data structure, 50
Sharing data, 53
Sharing information, 43
Shipping addresses, 30
Silly Window Syndrome, 287
Simple routers, 225–227, 308–309
SimpleRouter class, 84
Slow messages, 144
Smart proxies, 29, 35, 36, 559–560
C#, 561–568
MSMQ, 561–568
Smart Proxy pattern, 558–568
SmartProxyBase class, 563
SmartProxyReplyConsumer class, 565
SmartProxyReplyConsumerMetrics class, 566
SmartProxyRequestConsumer class, 564
SOAP (Simple Object Access Protocol)
binding styles, 375
command messages, 146
document messages, 148, 149–150
encoding style, 374
messages, 68–69
recursive nature of messages, 69
transport protocol, 373
Web services, 372–373
SOAP 1.2 and Request-Response Message Exchange pattern, 157, 162, 168–169
SOAP messages
envelope wrappers, 332–333
Request-Reply pairs, 157
SOAP request messages, 168, 174
SOAP response messages
correlation to original request, 168–169
sequencing and correlation to original request, 174–175
SonicMQ Bridges, 136
Splitter pattern, 173, 226, 259–267
Splitters, 24, 25
aggregators and, 274
C# XML order document, 262–267
filtering, 344
iterating, 260–261
MSMQ XML order document, 264–267
ordered or unordered child messages, 262
static, 261
Splitting messages, 226, 260–267
SQL-based relational databases, 48
Stale information, 45
Standard file formats, 44
Standardized data formats, 85
State
aspects, 219
keeping in messages, 316–317
process manager maintenance, 314
State request messages, 153
Static channels, 99
Static splitters, 261, 343–344
Stock trading
bridges, 135
channel adapter, 131
Datatype Channel pattern, 114
dead letter channels, 121
Durable Subscriber pattern, 125
guaranteed delivery, 124–125
invalid messages, 118
message bus, 141
Publish-Subscribe Channel pattern, 108–109
Store-and-forward process, 122
StreamMessage subtype, 68
Structural transformations, 90–93
SubjectGateway class, 211, 217
Subscribers, 62
avoiding missing messages, 522–523
durable or nondurable, 108
multiple channels, 108
notifying once about event, 106
special wildcard characters, 108
Synchronous block, 155
Synchronous implementation of loan broker system, 371–400
Syntax layer, 88
System management, 537
analyzing and debugging message flow, 551–554
avoiding infinite loops, 554
internal faults, 569
leftover messages, 572–575
loan broker system, 577–602
monitoring and controlling, 538
observing and analyzing message traffic, 538
reporting against message information, 555–557
routing messages for, 545–546
testing and debugging, 539
tracking messages, 558–568
widely distributed system, 540–541
Systems
data transfer between, 87–88
management, 16
out of synchronization, 45
T
Taking orders, 18–19, 24–25
Talks, 62
TCP/IP, 12–13, 88
ensuring in-sequence delivery of messages, 287
envelope wrappers, 333–334
tightly coupled dependencies, 11–12
Tee, 547
Template methods, 292, 404
temporaryQueue class, 215
Test data generator, 36
Test data verifier, 36
Test Message pattern, 569–571
Test messages, 36, 569–571
Testing
gateways, 475
Guaranteed Delivery pattern, 123–124
loan broker system (MSMQ), 440–443
Text-based files, 44
TextMessage subtype, 68
TIBCO ActiveEnterprise
canonical data models, 360
loan broker system, 445–462
message history, 553–554
TIBCO Repository for Metadata Management Integration, 450–451
TIB/IntegrationManager Process Manager Tool, 448–450
TIB/MessageBroker, 234–236
INDEX

TIB/RendezVous Transport, 448
Tight coupling, 10, 32
Tightly coupled applications, 39–40
Tightly coupled dependencies
  integration, 11–14
  TCP/IP, 11–12
Timeout strategy, 272
Timeout with override strategy, 273
Topic interface, 109
TopicPublisher class, 109, 209
TopicSubscriber class, 109
Transactional Client pattern, 77, 84, 97,
  131, 172, 484–493
JMS transacted session, 489
message groups, 487–488
message/database coordination, 488
message/workflow coordination, 488
MSMQ, 124
.NET transactional queue, 490
send-receive message pairs, 487
transactional filter with MSMQ, 490–493
Transactions, 172, 484–485
Transform method, 265
Transformations, 54, 58
  chaining, 89–90
  changing application internal data
    format, 357
  changing at individual level, 90
  content enrichers, 338–341
Data Representation layer, 87, 88
Data Structures layer, 87, 88
Data Types layer, 87, 88
decoupling levels, 88–89
dragging and dropping, 94
eliminating dependencies, 327–328
external translators, 357–358
implementing messaging mapper,
  357–358
levels of, 87–88
metadata management, 328–329
  at multiple layers, 89
  options, 357–358
  outside of messaging, 329
structural, 90–93
Transport layer, 87–88
visual tools, 93–94
XML documents, 90–93
Translators, 20, 23, 31, 56, 85–94
  chaining multiple units, 89–90
  data formats, 353
  double translation, 358
  external, 357–358
  versus mappers, 482
  resolving data format differences,
    355–356
  routing messages to correct, 353–354
Transport protocols, 87
Tree structure, 260–261
Trigger message, 313
Tunneling, 330, 334
Two-way channels, 154
Two-way messages, 154
U
UDDI (Universal Description, Discovery
  and Integration), 379
UML (Unified Modeling Language)
  activity diagrams, 21–22
Unidirectional adapters, 130
Unidirectional channels, 100
Universal storage mechanism, 44
Update messages, 153
updateConsumer method, 218
Update() method, 151, 207–209, 213–214
updateNoState() method, 218–219
Updating files, 45
User interface adapters, 129
User interfaces, 129
V
Validated Order message, 26
Verify Customer Standing message, 27
Visual transformation tools, 93–94
Void replies, 156
W
Wait for all strategy, 272
Web services, 3
  adapters, 132
  Apache AXIS toolkit, 371
  architecture usage scenarios, 174–175
  asynchronous versus synchronous
    messaging, 373–374
discovery, 379
encoding style, 374
existing standards, 4
HTTP, 51
loan broker system (Java) design
   considerations, 372–376
reliability, 375–376
Remote Procedure Invocation pattern, 375
Request-Response Message Exchange
   pattern, 162, 168–169
security, 375–376
SOAP (Simple Object Access Protocol), 372–373
synchronous implementation of loan
   broker system, 371–400
transport protocol, 373
Web Services Gateway, 132
WebSphere Application Server, 132
WebSphere MQ for Java
   Guaranteed Delivery, 126
   messaging systems, 64–65
   persistent channels, 126
   queues, 65
WGRUS (Widgets & Gadgets ‘R Us), 17
   announcements, 17, 33–34
   changing addresses, 17, 30–32
   channels to interact with customers, 18
   checking order status, 17
   checking status, 26–29
   internal systems, 18
   inventory systems, 22–23
   processing orders, 17, 20–25
   requirements, 17
SOAs (service-oriented architectures), 8
   taking orders, 17, 18–20
   testing and monitoring, 17, 34–36
   updating catalog, 17, 32–33
Wire Tap pattern, 547–550
World Wide Web Consortium Web site, 373, 374
Wrapping and unwrapping data in enve-
   lope, 331–335
WSDD (Web Services Deployment
   Descriptor), 378
WSDL (Web Services Definition Lan-
   guage), 374
   canonical data models, 359–360
   Command Message pattern, 146
doctor messages, 149–150
WSFL (Web Services Flow Language), 318
WS-Routing (Web Services Routing
   Protocol), 310–311
X
   XLANG, 318, 634
XML, 3, 149, 182
XML documents, 68, 90–93
XML files, 44
XML schema, 374–375
XML Schema Definition Tool, 415
XML Web services, 371–400
XML Splitter class, 263–264
XmlMessageFormatter class, 201
XSL, 3, 90–93
XSLT (XSL Transformation) language, 90
XslTransform class, 265