The Addison-Wesley Signature Series

Enterprise Integration Patterns

Designing, Building, and Deploying Messaging Solutions

A Martin Fowler Signature Book

Gregor Hohpe
Bobby Woolf

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

Forewords by John Crupi and Martin Fowler

List of Patterns

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 do 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

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

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Hohpe, Gregor.
Enterprise integration patterns : designing, building, and deploying messaging solutions / Gregor Hohpe, Bobby Woolf.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-20068-3
1. Telecommunication—Message processing. 2. Management information systems. I. Woolf, Bobby. II. Title.

TK5102.5.H5882 2003
005.7'136—dc22

2003017989

ISBN: 0-321-20068-3

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

Fifteenth printing, May 2011

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

This page intentionally left blank

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

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 communication 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 implementing 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

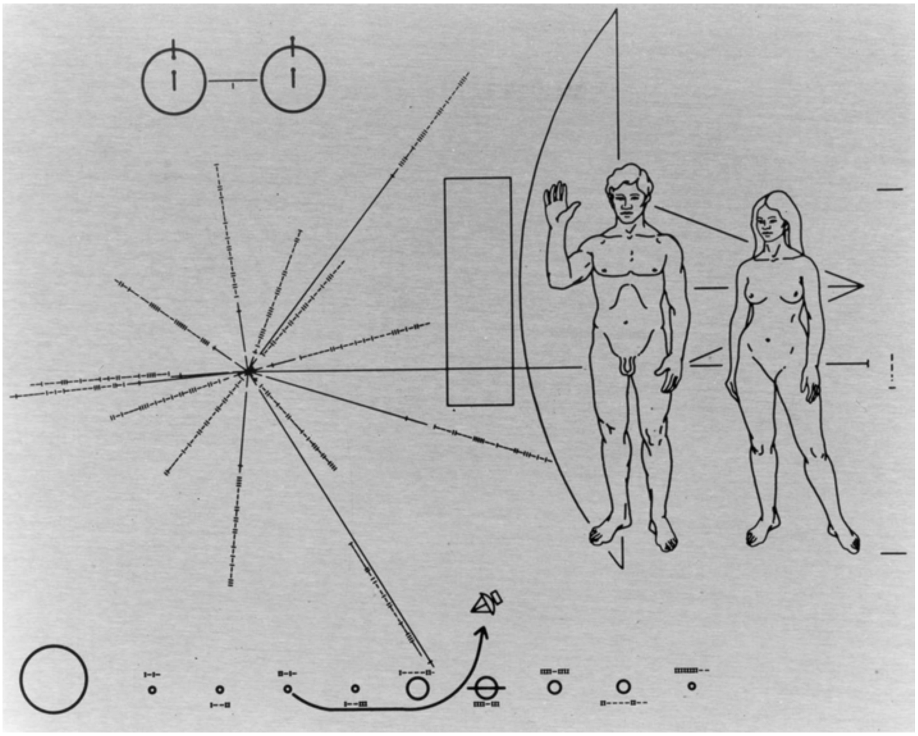
The common theme for books in the *Martin Fowler Signature Series* is a picture of a bridge. In some sense we lucked out, because what theme would make a better match for a book on integration? For thousands of years, bridges have helped connect people from different shores, mountains, and sides of the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

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.

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 him 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 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 *message*s to each other. *Channel*s, 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?

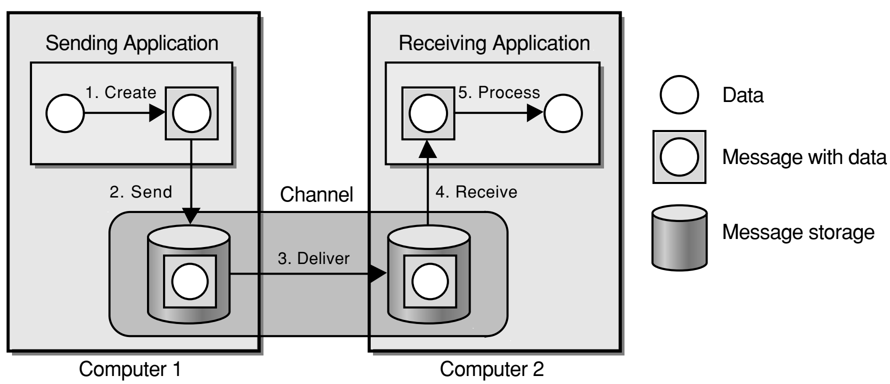
Messaging capabilities are typically provided by a separate software system called a *messaging system* or *message-oriented middleware* (MOM). A messaging system manages messaging the way a database system manages data persistence. Just as an administrator must populate the database with the schema 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:



Message Transmission Step-by-Step

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.

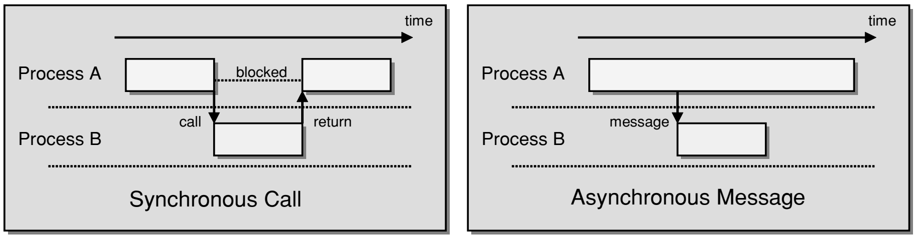
Therefore, many messaging systems need to bridge the gap between synchronous and asynchronous solutions.

- *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).



Synchronous and Asynchronous Call Semantics

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.

Enterprise Integration Patterns	**Java Message Service (JMS)**	**Microsoft MSMQ**	**WebSphere MQ**
Message Channel	Destination	MessageQueue	Queue
Point-to-Point Channel	Queue	MessageQueue	Queue
Publish-Subscribe Channel	Topic	—	—
Message	Message	Message	Message
Message Endpoint	MessageProducer, MessageConsumer		

Enterprise Integration Patterns	TIBCO	WebMethods	SeeBeyond	Vitria
Message Channel	Subject	Queue	Intelligent Queue	Channel
Point-to-Point Channel	Distributed Queue	Deliver Action	Intelligent Queue	Channel
Publish-Subscribe Channel	Subject	Publish-Subscribe Action	Intelligent Queue	Publish-Subscribe Channel
Message	Message	Document	Event	Event
Message Endpoint	Publisher, Subscriber	Publisher, Subscriber	Publisher, Subscriber	Publisher, Subscriber

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.

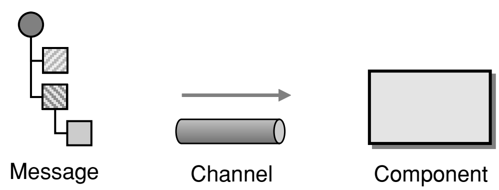
- *Problem*—This explains the difficulty you are facing, expressed as a question. You should be able to read the problem statement and quickly determine if this pattern is relevant to your work. We've formatted the problem to be one sentence delimited by horizontal rules.

- *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:



Visual Notation for Messaging Solutions

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 solution. To highlight the collaboration between multiple patterns, we included more comprehensive examples as interludes (see Chapters 6, 9, and 12). These solutions illustrate many of the trade-offs involved in designing a more comprehensive 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.

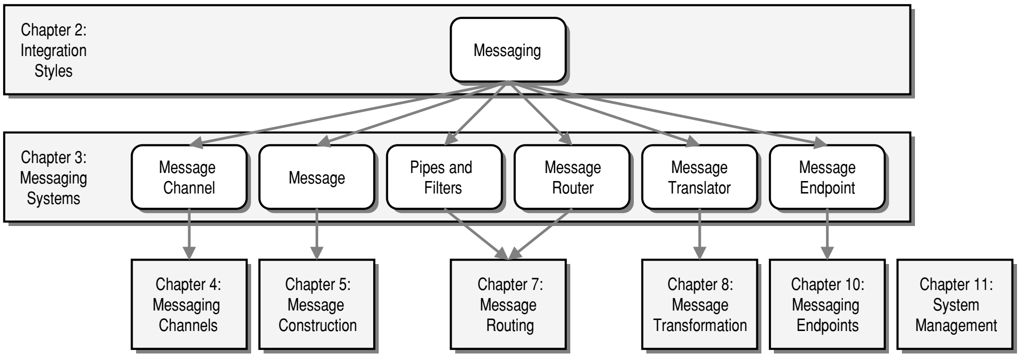
The Java examples in this book are based on the JMS 1.1 specification, which is part of the J2EE 1.4 specification. By the time this book is published, most messaging and application server vendors will support JMS 1.1. You can download Sun Microsystems' reference implementation of the JMS specification from Sun's Web site: *http://java.sun.com/j2ee*.

The Microsoft .NET examples are based on Version 1.1 of the .NET Framework and are written in C#. You can download the .NET Framework SDK from Microsoft's Web site: *http://msdn.microsoft.com/net*.

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 fundamental 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.



Relationship of Root Patterns and Chapters

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.

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 provider 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.

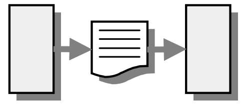
This chapter presents each style as a pattern. The four patterns share the same problem statement—the need to integrate applications—and very similar contexts. What differentiates them are the forces searching for a more elegant solution. Each pattern builds on the last, looking for a more sophisticated 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.

File Transfer

by Martin Fowler



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.

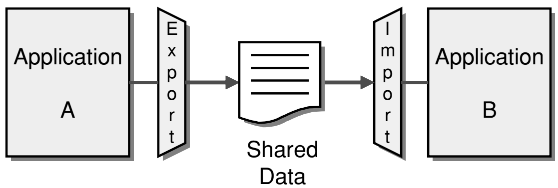
As a result, any organization has to worry about sharing information between very divergent applications. These can be written in different languages, based on different platforms, and have different assumptions about how the business operates.

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.

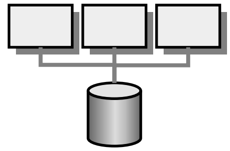
In fact, the biggest problem with staleness is often on the software development staff themselves, who frequently must deal with data that isn't quite right. This can lead to inconsistencies that are difficult to resolve. If a customer changes his address on the same day with two different systems, but 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).

Shared Database

by Martin Fowler



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.

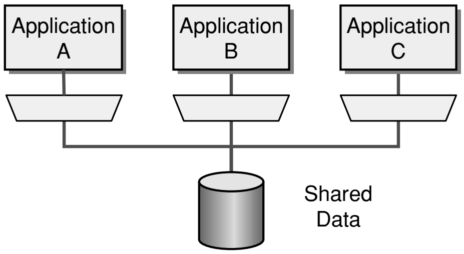
How can I integrate multiple applications so that they work together and can exchange information?

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.

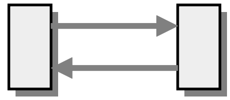
Another, harder limit to *Shared Database* is external packages. Most packaged applications won't work with a schema other than their own. Even if there is some room for adaptation, it's likely to be much more limited than integrators would like. Adding to the problem, software vendors usually reserve 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).

Remote Procedure Invocation



by Martin Fowler

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?

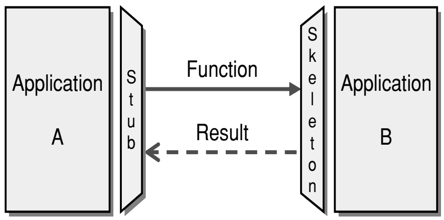
File Transfer (43) and *Shared Database* (47) enable applications to share their data, which is an important part of application integration, but just sharing data is often not enough. Changes in data often require actions to be taken across different applications. For example, changing an address may 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.

Develop each application as a large-scale object or component with encapsulated data. Provide an interface to allow other applications to interact with the running application.



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.

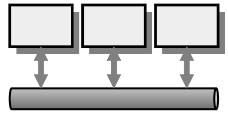
The fact that there are methods that wrap the data makes it easier to deal with semantic dissonance. Applications can provide multiple interfaces to the same data, allowing some clients to see one style and others a different style. Even updates can use multiple interfaces. This provides a lot more ability 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.

Messaging



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.

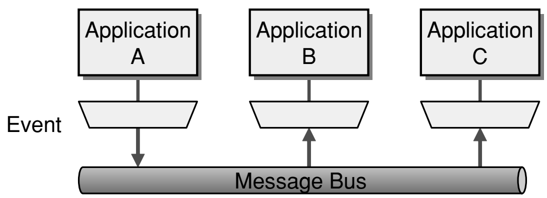
File Transfer (43) allows you to keep the applications well decoupled but at the cost of timeliness. Systems just can't keep up with each other. Collaborative behavior is way too slow. *Shared Database* (47) keeps data together in a responsive way but at the cost of coupling everything to the database.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

Use *Messaging* to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats.



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 Endpoint*s (95) to perform the actual sending and receiving.

Index

D

H

I

S