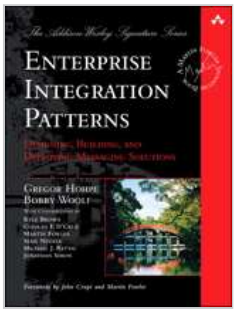


View Larger Image

Add To My Wish List

Register your product to gain access to bonus material or receive a coupon.

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions

By Gregor Hohpe, Bobby Woolf

Published Oct 10, 2003 by Addison-Wesley Professional. Part of the Addison-Wesley Signature Series (Fowler) series.

Best Value Purchase

Book + eBook Bundle

Add to cart

FREE SHIPPING!

Your Price: $79.44
List Price: $136.98
Includes EPUB and PDF
About eBook Formats

More Purchase Options

Book

Add to cart

FREE SHIPPING!

Your Price: $55.99
List Price: $69.99
Usually ships in 24 hours.

eBook (Watermarked)

Add to cart

Your Price: $53.59
List Price: $66.99
Includes EPUB and PDF
About eBook Formats

Description | Sample Content | Updates

Copyright 2004
Dimensions: 7" x 9-1/4"
Pages: 736
Edition: 1st

Book
ISBN-10: 0-321-20068-3
ISBN-13: 978-0-321-20068-6

Enterprise Integration Patterns provides an invaluable catalog of sixty-five patterns, with real-world solutions that demonstrate the formidable of messaging and help you to design effective messaging solutions for your enterprise.

The authors also include examples covering a variety of different integration technologies, such as JMS, MSMQ, TIBCO ActiveEnterprise, Microsoft BizTalk, SOAP, and XSL. A case study describing a bond trading system illustrates the patterns in practice, and the book offers a look at emerging standards, as well as insights into what the future of enterprise integration might hold.

InformIT Promotional Mailings & Special Offers

I would like to receive exclusive offers and hear about products from InformIT and its family of brands. I can unsubscribe at any time.
Privacy Notice

Email Address

Submit

Request an Instructor or Media review copy.
Corporate, Academic, and Employee Purchases
International Buying Options

Other Things You Might Like

Generative Analysis: The Power of Generative AI for Object-Oriented Software Engineering with UML
By Jim Arlow, Ila Neustadt
Book $51.99

Object-Oriented Thought Process, The, 5th Edition
By Matt Weisfeld
Book $31.99

Object-Oriented Thought Process, The, 5th Edition
By Matt Weisfeld
eBook (Watermarked) $30.39

This book provides a consistent vocabulary and visual notation framework to describe large-scale integration solutions across many technologies. It also explores in detail the advantages and limitations of asynchronous messaging architectures. The authors present practical advice on designing code that connects an application to a messaging system, and provide extensive information to help you determine when to send a message, how to route it to the proper destination, and how to monitor the health of a messaging system. If you want to know how to manage, monitor, and maintain a messaging system once it is in use, get this book.

The Addison-Wesley Signature Series

A Martin Fowler Signature Book

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

Forewords by John Crupi and Martin Fowler

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

List of Patterns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Message Expiration (176) How can a sender indicate when a message should be considered stale and thus shouldn't be processed?

Message Filter (237) How can a component avoid receiving uninteresting messages?

Message History (551) How can we effectively analyze and debug the flow of messages in a loosely coupled system?

Message Router (78) How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?

Message Sequence (170) How can messaging transmit an arbitrarily large amount of data?

Message Store (555) How can we report against message information without disturbing the loosely coupled and transient nature of a messaging system?

Message Translator (85) How can systems using different data formats communicate with each other using messaging?

Message (66) How can two applications connected by a message channel exchange a piece of information?

Messaging Bridge (133) How can multiple messaging systems be connected so that messages available on one are also available on the others?

Messaging Gateway (468) How do you encapsulate access to the messaging system from the rest of the application?

Messaging Mapper (477) How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?

Messaging (53) How can I integrate multiple applications so that they work together and can exchange information?

Normalizer (352) How do you process messages that are semantically equivalent but arrive in a different format?

Pipes and Filters (70) How can we perform complex processing on a message while maintaining independence and flexibility?

Point-to-Point Channel (103) How can the caller be sure that exactly one receiver will receive the document or perform the call?

Polling Consumer (494) How can an application consume a message when the application is ready?

Process Manager (312) How do we route a message through multiple processing steps when the required steps may not be known at design time and may not be sequential?

Publish-Subscribe Channel (106) How can the sender broadcast an event to all interested receivers?

Recipient List (249) How do we route a message to a dynamic list of recipients?

Remote Procedure Invocation (50) How can I integrate multiple applications so that they work together and can exchange information?

Request-Reply (154) When an application sends a message, how can it get a response from the receiver?

Resequencer (283) How can we get a stream of related but out-of-sequence messages back into the correct order?

Return Address (159) How does a replier know where to send the reply?

Routing Slip (301) How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design time and may vary for each message?

Scatter-Gather (297) How 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

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

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

The publisher offers 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

Copyright © 2004 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN: 0-321-20068-3

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

Fifteenth printing, May 2011

Chapter 3

Messaging Systems

Introduction

In Chapter 2, "Integration Styles," we discussed the various options for connecting applications with one another, including *Messaging* (53). Messaging makes applications loosely coupled by communicating asynchronously, which also makes the communication more reliable because the two applications do not have to be running at the same time. Messaging makes the messaging system responsible for transferring data from one application to another, so the applications can focus on what data they need to share as opposed to how to share it.

Basic Messaging Concepts

Like most technologies, *Messaging* (53) involves certain basic concepts. Once you understand these concepts, you can make sense of the technology even before you understand all of the details about how to use it. The following are the basic messaging concepts.

Channels—Messaging applications transmit data through a *Message Channel* (60), a virtual pipe that connects a sender to a receiver. A newly installed messaging system typically doesn't contain any channels; you must determine how your applications need to communicate and then create the channels to facilitate it.

Messages—A *Message* (66) is an atomic packet of data that can be transmitted on a channel. Thus, to transmit data, an application must break the data into one or more packets, wrap each packet as a message, and then send the message on a channel. Likewise, a receiver application receives a message and must extract the data from the message to process it. The message system will

IPR2025-00200 / 722 Patent Page 9 of 49 EX1021

try repeatedly to deliver the message (e.g., transmit it from the sender to the receiver) until it succeeds.

Pipes and Filters—In the simplest case, the messaging system delivers a message directly from the sender's computer to the receiver's computer. However, certain actions often need to be performed on the message after it is sent by its original sender but before it is received by its final receiver. For example, the message may have to be validated or transformed because the receiver expects a message format different from the sender's. The *Pipes and Filters* (70) architecture describes how multiple processing steps can be chained together using channels.

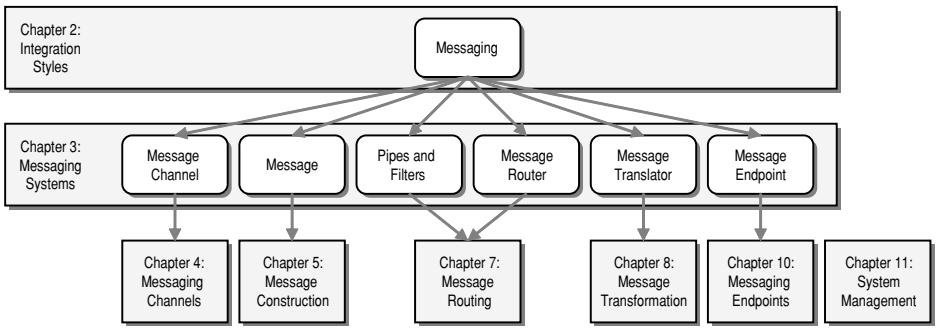
Routing—In a large enterprise with numerous applications and channels to connect them, a message may have to go through several channels to reach its final destination. The route a message must follow may be so complex that the original sender does not know what channel will get the message to the final receiver. Instead, the original sender sends the message to a *Message Router* (78), an application component that takes the place of a filter in the *Pipes and Filters* (70) architecture. The router then determines how to navigate the channel topology and directs the message to the final receiver, or at least to the next router.

Transformation—Various applications may not agree on the format for the same conceptual data; the sender formats the message one way, but the receiver expects it to be formatted another way. To reconcile this, the message must go through an intermediate filter, a *Message Translator* (85), which converts the message from one format to another.

Endpoints—Most applications do not have any built-in capability to interface with a messaging system. Rather, they must contain a layer of code that knows both how the application works and how the messaging system works, bridging the two so that they work together. This bridge code is a set of coordinated *Message Endpoint*s (95) that enable the application to send and receive messages.

Book Organization

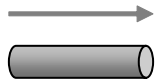
The patterns in this chapter provide you with the basic vocabulary and understanding of how to achieve enterprise integration using *Messaging* (53). Each subsequent chapter builds on one of the base patterns in this chapter and covers that particular topic in more depth.



Relationship of Root Patterns and Chapters

You can read this chapter straight through for an overview of the main topics in *Messaging* (53). For more details about any one of these topics, skip ahead to the chapter associated with that particular pattern.

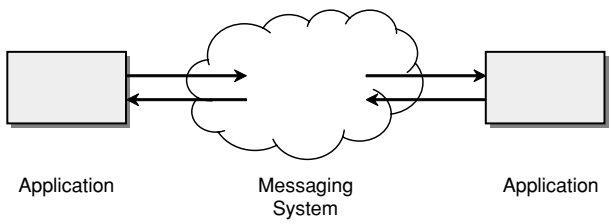
Message Channel



An enterprise has two separate applications that need to communicate by using *Messaging* (53).

How does one application communicate with another using messaging?

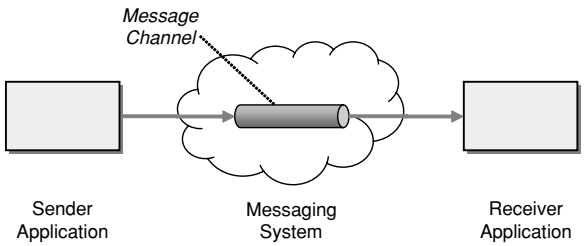
Once a group of applications has a messaging system available, it's tempting to think that any application can communicate with any other application anytime you want it to. Yet, the messaging system does not magically connect all of the applications.



Applications Magically Connected

Likewise, it's not as though an application just randomly throws out information into the messaging system while other applications just randomly grab whatever information they run across. (Even if this worked, it wouldn't be very efficient.) Rather, the application sending out the information knows what sort of information it is, and the applications that would like to receive information aren't looking for just any information but for particular types of information they can use. So the messaging system isn't a big bucket that applications throw information into and pull information out of. It's a set of connections that enables applications to communicate by transmitting information in predetermined, predictable ways.

Connect the applications using a *Message Channel*, where one application writes information to the channel and the other one reads that information from the channel.



When an application has information to communicate, it doesn't just fling the information into the messaging system but adds the information to a particular *Message Channel*. An application receiving information doesn't just pick it up at random from the messaging system; it retrieves the information from a particular *Message Channel*.

The application sending information doesn't necessarily know what particular application will end up retrieving it, but it can be assured that the application that retrieves the information is interested in the information. This is because the messaging system has different *Message Channel*s for different types of information the applications want to communicate. When an application sends information, it doesn't randomly add the information to any channel available; it adds it to a channel whose specific purpose is to communicate that sort of information. Likewise, an application that wants to receive particular information doesn't pull info off some random channel; it selects what channel to get information from based on what type of information it wants.

Channels are logical addresses in the messaging system. How they're actually implemented depends on the messaging system product and its implementation. Perhaps every *Message Endpoint* (95) has a direct connection to every other endpoint, or perhaps they're all connected through a central hub. Perhaps several separate logical channels are configured as one physical channel that nevertheless keeps straight which messages are intended for which destination. The set of defined logical channels hides these configuration details from the applications.

A messaging system doesn't automatically come preconfigured with all of the message channels the applications need to communicate. Rather, the developers

designing the applications and the communication between them have to decide what channels they need for the communication. Then the system administrator who installs the messaging system software must also configure it to set up the channels that the applications expect. Although some messaging system implementations support creating new channels while the applications are running, this isn't very useful because other applications besides the one that creates the channel must know about the new channel so they can start using it too. Thus, the number and purpose of channels available tend to be fixed at deployment time. (There are exceptions to this rule; see the introduction to Chapter 4, "Messaging Channels.")

Message Channel

A Little Bit of Messaging Vocabulary

So what do we call the applications that communicate via a *Message Channel*? There are a number of terms out there that are largely equivalent. The most generic terms are probably *sender* and *receiver*; an application sends a message to a *Message Channel* to be received by another application. Other popular terms are *producer* and *consumer*. You will also see *publisher* and *subscriber*, but they are geared more toward *Publish-Subscribe Channel*s (106) and are often used in generic form. Sometimes we say that an application *listens* on a channel to which another application *talks*. In the world of Web services, we generally talk about a *requester* and a *provider*. These terms usually imply that the requester sends a message to the provider and receives a response back. In the olden days we called these *client* and *server* (the terms are equivalent, but saying "client" and "server" is not cool).

Now it gets confusing. When dealing with Web services, the application that sends a message to the service provider is often referred to as the *consumer* of the service even though it sends the request message. We can think of it in such a way that the consumer sends a message to the provider and then consumes the response. Luckily, use of the term with this meaning is limited to *Remote Procedure Invocation* (50) scenarios. An application that sends or receives messages may be called a *client* of the messaging system; a more specific term is *endpoint* or *message endpoint*.

Something that often fools developers when they first get started with using a messaging system is what exactly needs to be done to create a channel. A developer can write Java code that calls the method `createQueue` defined in the JMS

API or .NET code that includes the statement `new MessageQueue`, but neither code actually allocates a new queue resource in the messaging system. Rather, these pieces of code simply instantiate a runtime object that provides access to a resource that was already created in the messaging system using its administration tools.

There is another issue you should keep in mind when designing the channels for a messaging system: Channels are cheap, but they're not free. Applications need multiple channels for transmitting different types of information and transmitting the same information to lots of other applications. Each channel requires memory to represent the messages; persistent channels require disk space as well. Even if an enterprise system had unlimited memory and disk space, any messaging system implementation usually imposes some hard or practical limit to how many channels it can service consistently. So plan on creating new channels as your application needs them, but if it needs thousands of channels or needs to scale in ways that may require thousands of channels, you'll need to choose a highly scalable messaging system implementation and test that scalability to make sure it meets your needs.

Channel Names

If channels are logical addresses, what do these addresses look like? As in so many cases, the detailed answer depends on the implementation of the messaging system. Nevertheless, in most cases channels are referenced by an alphanumeric name, such as `MyChannel`. Many messaging systems support a hierarchical channel-naming scheme, which enables you to organize channels in a way that is similar to a file system with folders and subfolders. For example, `MyCorp/Prod/OrderProcessing/NewOrders` would indicate a channel that is used in a production application at `MyCorp` and contains new orders.

There are two different kinds of message channels: *Point-to-Point Channel*s (103) and *Publish-Subscribe Channel*s (106). Mixing different data types on the same channel can cause a lot of confusion; to avoid this, use separate *Datatype Channels* (111). *Selective Consumer* (515) makes one physical channel act logically like multiple channels. Applications that use messaging often benefit from a special channel for invalid messages, an *Invalid Message Channel*. Applications that wish to use *Messaging* (53) but do not have access to a messaging client can still connect to the messaging system using *Channel Adapter*s (127). A

well-designed set of channels forms a *Message Bus* (137) that acts like a messaging API for a whole group of applications.

Example: *Stock Trading*

When a stock trading application makes a trade, it puts the request on a *Message Channel* for trade requests. Another application that processes trade requests will look for those it can process on that same message channel. If the requesting application needs to request a stock quote, it will probably use a different *Message Channel*, one designed for stock quotes, so that the quote requests stay separate from the trade requests.

Example: *J2EE JMS Reference Implementation*

Let's look at how to create a *Message Channel* in JMS. The J2EE SDK ships with a reference implementation of the J2EE services, including JMS. The reference server can be started with the `j2ee` command. Message channels have to be configured using the `j2eeadmin` tool. This tool can configure both queues and topics.

```
j2eeadmin -addJmsDestination jms/mytopic topic
j2eeadmin -addJmsDestination jms/myqueue queue
```

Once the channels have been administered (created), they can be accessed by JMS client code.

```
Context jndiContext = new InitialContext();
Queue myQueue = (Queue) jndiContext.lookup("jms/myqueue");
Topic myTopic = (Topic) jndiContext.lookup("jms/mytopic");
```

The JNDI lookup doesn't create the queue (or topic); it was already created by the `j2eeadmin` command. The JNDI lookup simply creates a `Queue` instance in Java that models and provides access to the queue structure in the messaging system.

Example: *IBM WebSphere MQ*

If your messaging system implementation is IBM's WebSphere MQ for Java, which implements JMS, you'll use the WebSphere MQ JMS administration tool to create destinations. This will create a queue named `myQueue`.

```
DEFINE Q(myQueue)
```

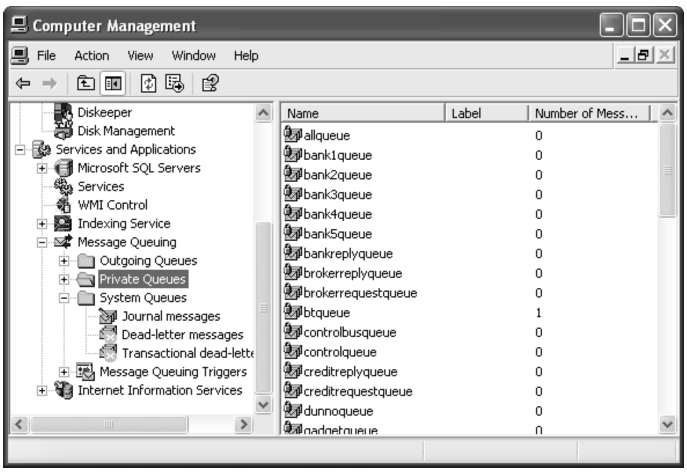
Once that queue exists in WebSphere MQ, an application can access the queue.

WebSphere MQ, without the full WebSphere Application Server, does not include a JNDI implementation, so we cannot use JNDI to look up the queue as we did in the J2EE example. Rather, we must access the queue via a JMS session, like this.

```
Session session = // create the session
Queue queue = session.createQueue("myQueue");
```

Example: *Microsoft MSMQ*

MSMQ provides a number of different ways to create a message channel, called a queue. You can create a queue using the Microsoft Message Queue Explorer or the Computer Management console (see figure). From here you can set queue properties or delete queues.



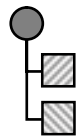
Alternatively, you can create the queue using code.

```
using System.Messaging;
...
MessageQueue.Create("MyQueue");
```

Once the queue is created, an application can access it by creating a `MessageQueue` instance, passing the name of the queue.

```
MessageQueue mq = new MessageQueue("MyQueue");
```

Message



An enterprise has two separate applications that are communicating via *Messaging* (53), using a *Message Channel* (60) that connects them.

How can two applications connected by a Message Channel exchange a piece of information?

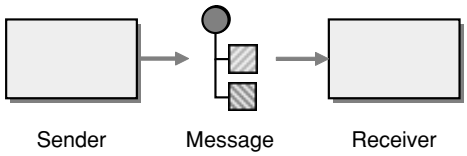
A *Message Channel* (60) can often be thought of as a pipe, a conduit from one application to another. It might stand to reason then that data could be poured into one end, like water, and it would come flowing out of the other end. But most application data isn't one continuous stream; it consists of units, such as records, objects, database rows, and the like. So a channel must transmit units of data.

What does it mean to "transmit" data? In a function call, the caller can pass a parameter by reference by passing a pointer to the data's address in memory; this works because both the caller and the function share the same memory heap. Similarly, two threads in the same process can pass a record or object by passing a pointer, since they both share the same memory space.

Two separate processes passing a piece of data have more work to do. Since they each have their own memory space, they have to copy the data from one memory space to the other. The data is usually transmitted as a byte stream, the most basic form of data. This means that the first process must *marshal* the data into byte form, and then copy it from the first process to the second one; the second process will *unmarshal* the data back into its original form, such that the second process then has a copy of the original data in the first process. Marshaling is how a Remote Procedure Call (RPC) sends arguments to the remote process and how the process returns the result.

So messaging transmits discrete units of data, and it does so by marshaling the data from the sender and unmarshaling it in the receiver so that the receiver has its own local copy. What would be helpful would be a simple way to wrap a unit of data such that it is appropriate to transmit the data on a messaging channel.

Package the information into a *Message*, a data record that the messaging system can transmit through a Message Channel.



Thus, any data that is to be transmitted via a messaging system must be converted into one or more messages that can be sent through messaging channels.

A message consists of two basic parts.

1. **Header**—Information used by the messaging system that describes the data being transmitted, its origin, its destination, and so on.
2. **Body**—The data being transmitted, which is generally ignored by the messaging system and simply transmitted as is.

This concept is not unique to messaging. Both postal service mail and e-mail send data as discrete mail messages. An Ethernet network transmits data as packets, as does the IP part of TCP/IP such as the Internet. Streaming media on the Internet is actually a series of packets.

To the messaging system, all messages are the same: some body of data to be transmitted as described by the header. However, to the applications programmer, there are different types of messages—that is, different application styles of use. Use a *Command Message* (145) to invoke a procedure in another application. Use a *Document Message* (147) to pass a set of data to another application. Use an *Event Message* (151) to notify another application of a change in this application. If the other application should send back a reply, use *Request-Reply* (154).

If an application wishes to send more information than one message can hold, break the data into smaller parts and send the parts as a *Message Sequence* (170). If the data is only useful for a limited amount of time, specify this use-by time as a *Message Expiration* (176). Since all the various senders and receivers of messages must agree on the format of the data in the messages, specify the format as a *Canonical Data Model* (355).

Example: *JMS Message*

In JMS, a message is represented by the type `Message`, which has several subtypes. In each subtype, the header structure is the same; it's the body format that varies by type.

1. `TextMessage`—The most common type of message. The body is a string, such as literal text or an XML document. `textMessage.getText()` returns the message body as a `String`.
2. `BytesMessage`—The simplest, most universal type of message. The body is a byte array. `bytesMessage.readBytes(byteArray)` copies the contents into the specified byte array.
3. `ObjectMessage`—The body is a single Java object, specifically one that implements `java.io.Serializable`, which enables the object to be marshaled and unmarshaled. `objectMessage.getObject()` returns the `Serializable`.
4. `StreamMessage`—The body is a stream of Java primitives. The receiver uses methods like `readBoolean()`, `readChar()`, and `readDouble()` to read the data from the message.
5. `MapMessage`—The body acts like a `java.util.Map`, where the keys are `Strings`. The receiver uses methods like `getBoolean("isEnabled")` and `getInt("numberOfItems")` to read the data from the message.

Example: *.NET Message*

In .NET, the `Message` class implements the message type. It has a property, `Body`, which contains the contents of the message as an `Object`; `BodyStream` stores the contents as a `Stream`. Another property, `BodyType`, specifies the type of data the body contains, such as a string, a date, a currency, a number, or any other object.

Example: *SOAP Message*

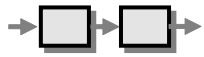
In the SOAP protocol [SOAP 1.1], a SOAP message is an example of *Message*. A SOAP message is an XML document that is an envelope (a root `SOAP-ENV:Envelope` element) that contains an optional header (a `SOAP-ENV:Header` element) and required body (a `SOAP-ENV:Body` element). This XML document is an atomic data record that can be transmitted (typically the transmission protocol is HTTP) so it is a message.

Here is an example of a SOAP message from the SOAP spec that shows an envelope containing a header and a body.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
       <t:Transaction
           xmlns:t="some-URI"
           SOAP-ENV:mustUnderstand="1">
               5
       </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DEF</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP also demonstrates the recursive nature of messages, because a SOAP message can be transmitted via a messaging system, which means that a messaging system message object (e.g., an object of type `javax.jms.Message` in JMS or `System.Messaging.Message` in .NET) contains the SOAP message (the XML `SOAP-ENV:Envelope` document). In this scenario, the transport protocol isn't HTTP but the messaging system's internal protocol (which in turn may be using HTTP or some other network protocol to transmit the data, but the messaging system makes the transmission reliable). For more information on transporting a message across a different messaging system, see *Envelope Wrapper* (330).

Pipes and Filters



In many enterprise integration scenarios, a single event triggers a sequence of processing steps, each performing a specific function. For example, let's assume a new order arrives in our enterprise in the form of a message. One requirement may be that the message is encrypted to prevent eavesdroppers from spying on a customer's order. A second requirement is that the messages contain authentication information in the form of a digital certificate to ensure that orders are placed only by trusted customers. In addition, duplicate messages could be sent from external parties (remember all the warnings on the popular shopping sites to click the Order Now button only once?). To avoid duplicate shipments and unhappy customers, we need to eliminate duplicate messages before subsequent order processing steps are initiated. To meet these requirements, we need to transform a series of possibly duplicated, encrypted messages containing extra authentication data into a series of unique, simple plain-text order messages without the extraneous data fields.

How can we perform complex processing on a message while maintaining independence and flexibility?

One possible solution would be to write a comprehensive "incoming message massaging module" that performs all the necessary functions. However, such an approach would be inflexible and difficult to test. What if we need to add a step or remove one? For example, what if orders can be placed by large customers who are on a private network and do not require encryption?

Implementing all functions inside a single component also reduces opportunities for reuse. Creating smaller, well-defined components allows us to reuse them in other processes. For example, order status messages may be encrypted but do not need to be de-duped because duplicate status requests are generally not harmful. Separating the decryption function into a separate module allows us to reuse this function for other messages.

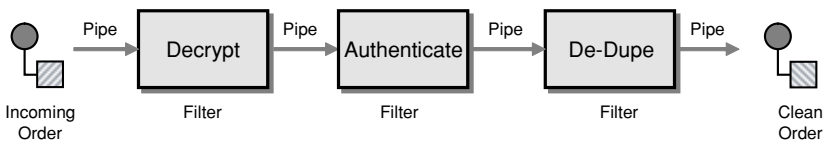
Integration solutions typically connect a collection of heterogeneous systems. As a result, different processing steps may need to execute on different physical

machines, such as when individual processing steps can only execute on specific systems. For example, it is possible that the private key required to decrypt incoming messages is only available on a designated machine and cannot be accessed from any other machine for security reasons. This means that the decryption component has to execute on this designated machine, whereas the other steps may execute on other machines. Likewise, different processing steps may be implemented using different programming languages or technologies that prevent them from running inside the same process or even on the same computer.

Implementing each function in a separate component can still introduce dependencies between components. For example, if the decryption component calls the authentication component with the results of the decryption, we cannot use the decryption function without the authentication function. We could resolve these dependencies if we could "compose" existing components into a sequence of processing steps in such a way that each component is independent from the other components in the system. This would imply that components expose generic external interfaces so that they are interchangeable.

If we use asynchronous messaging, we should take advantage of the asynchronous aspects of sending messages from one component to another. For example, a component can send a message to another component for further processing without waiting for the results. Using this technique, we could process multiple messages in parallel, one inside each component.

Use the *Pipes and Filters* architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (filters) that are connected by channels (pipes).



Each filter exposes a very simple interface: It receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe. The pipe connects one filter to the next, sending output messages from one filter to the next. Because all components use the same external interface, they can be *composed* into different solutions by connecting the components to different pipes. We can add new filters, omit existing ones, or rearrange them

into a new sequence—all without having to change the filters themselves. The connection between filter and pipe is sometimes called a *port*. In the basic form, each filter component has one input port and one output port.

When applied to our example problem, the *Pipes and Filters* architecture results in three filters connected by two pipes (see figure). We need one additional pipe to send messages to the decryption component and one to send the clear-text order messages from the de-duper to the order management system. This makes a total of four pipes.

Pipes and Filters

Pipes and Filters describes a fundamental architectural style for messaging systems: Individual processing steps (filters) are chained together through the messaging channels (pipes). Many patterns in this and the following sections, such as routing and transformation patterns, are based on this *Pipes and Filters* architectural style. This lets you easily combine individual patterns into larger solutions.

The *Pipes and Filters* style uses abstract pipes to decouple components from each other. The pipe allows one component to send a message into the pipe so that it can be consumed later by another process that is unknown to the component. The obvious implementation for such a pipe is a *Message Channel* (60). Typically, a *Message Channel* (60) provides language, platform, and location independence between the filters. This affords us the flexibility to move a processing step to a different machine for dependency, maintenance, or performance reasons. However, a *Message Channel* (60) provided by a messaging infrastructure can be quite heavyweight if all components can in fact reside on the same machine. Using a simple in-memory queue to implement the pipes would be much more efficient. Therefore, it is useful to design the components so that they communicate with an abstract pipe interface. The implementation of that interface can then be swapped out to use a *Message Channel* (60) or an alternative implementation such as an in-memory queue. The *Messaging Gateway* (468) describes how to design components for this flexibility.

One of the potential downsides of a *Pipes and Filters* architecture is the larger number of required channels. First, channels may not be an unlimited resource, since channels provide buffering and other functions that consume memory and CPU cycles. Also, publishing a message to a channel involves a certain amount of overhead because the data has to be translated from the application-internal format into the messaging infrastructure's own format. At the receiving end, this process has to be reversed. If we are using a long chain of filters, we are paying for the gain in flexibility with potentially lower performance due to repeated message data conversion.

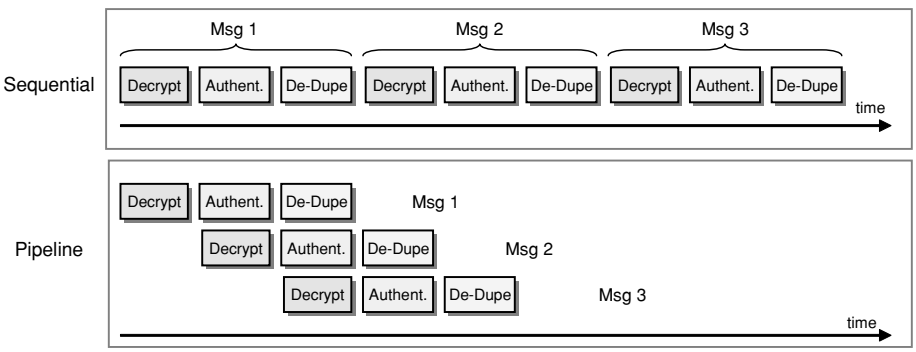
The pure form of *Pipes and Filters* allows each filter to have only a single input port and a single output port. When dealing with *Messaging* (53), we can relax this property somewhat. A component may consume messages off more

than one channel and also output messages to more than one channel (for example, a *Message Router* [78]). Likewise, multiple filter components can consume messages off a single *Message Channel* (60). A *Point-to-Point Channel* (103) ensures that only one filter component consumes each message.

Using *Pipes and Filters* also improves testability, an often overlooked benefit. We can test each individual processing step by passing a *Test Message* (66) to the component and comparing the result message to the expected outcome. It is more efficient to test and debug each core function in isolation because we can tailor the test mechanism to the specific function. For example, to test the encryption/decryption function we can pass in a large number of messages containing random data. After we encrypt and decrypt each message we compare it with the original. On the other hand, to test authentication, we need to supply messages with specific authentication codes that match known users in the system.

Pipeline Processing

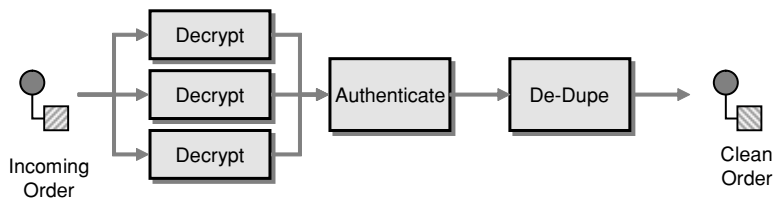
Connecting components with asynchronous *Message Channel*s (60) allows each unit in the chain to operate in its own thread or its own process. When a unit has completed processing one message, it can send the message to the output channel and immediately start processing another message. It does not have to wait for the subsequent components to read and process the message. This allows multiple messages to be processed concurrently as they pass through the individual stages. For example, after the first message has been decrypted, it can be passed on to the authentication component. At the same time, the next message can already be decrypted (see figure). We call such a configuration a *processing pipeline* because messages flow through the filters like liquid flows through a pipe. When compared to strictly sequential processing, a processing pipeline can significantly increase system throughput.



Pipeline Processing with Pipes and Filters

Parallel Processing

However, the overall system throughput is limited by the slowest process in the chain. We can deploy multiple parallel instances of that process to improve throughput. In this scenario, a *Point-to-Point Channel* (103) with *Competing Consumers* (502) is needed to guarantee that each message on the channel is consumed by exactly one of *N* available processors. This allows us to speed up the most time-intensive process and improve overall throughput. We need to be aware, though, that this configuration can cause messages to be processed out of order. If the sequence of messages is critical, we can run only one instance of each component or we must use a *Resequencer* (283).



Increasing Throughput with Parallel Processing

For example, if we assume that decrypting a message is much slower than authenticating it, we can use the configuration shown in the figure, running three parallel instances of the decryption component. Parallelizing filters works best if each filter is stateless—that is, it returns to the previous state after a message has been processed. This means that we cannot easily run multiple parallel de-dupe components because the component maintains a history of all messages that it already received and is therefore not stateless.

History of Pipes and Filters

Pipes and Filters architectures are by no means a new concept. The simple elegance of this architecture combined with the flexibility and high throughput makes it easy to understand the popularity of *Pipes and Filters* architectures. The simple semantics also allow formal methods to be used to describe the architecture.

[Kahn] described Kahn Process Networks in 1974 as a set of parallel processes that are connected by unbounded FIFO (First-In, First-Out) channels. [Garlan] contains a good chapter on different architectural styles, including *Pipes and Filters*. [Monroe] gives a detailed treatment of the relationships between architectural styles and design patterns. [PLoPD1] contains Regine

Meunier's "The Pipes and Filters Architecture," which formed the basis for the *Pipes and Filters* pattern included in [POSA]. Almost all integration-related implementations of *Pipes and Filters* follow the "Scenario IV" presented in [POSA], using active filters that pull, process, and push independently from and to queuing pipes. The pattern described by [POSA] assumes that each element undergoes the same processing steps as it is passed from filter to filter. This is generally not the case in an integration scenario. In many instances, messages are routed dynamically based on message content or external control. In fact, routing is such a common occurrence in enterprise integration that it warrants its own patterns, the *Message Router* (78).

Vocabulary

When discussing *Pipes and Filters* architectures, we need to be cautious with the term *filter*. We later define two additional patterns, the *Message Filter* (237) and the *Content Filter* (342). While both of these are special cases of a generic filter, so are many other patterns in this pattern language. In other words, a pattern does not have to involve a filtering function (e.g., eliminating fields or messages) in order to be a filter in the sense of *Pipes and Filters*. We could have avoided this confusion by renaming the *Pipes and Filters* architectural style. However, we felt that *Pipes and Filters* is such an important and widely discussed concept that it would be even more confusing if we gave it a new name. We are trying to use the word *filter* cautiously throughout these patterns and trying to be clear about whether we are talking about a generic filter as in *Pipes and Filters* or a *Message Filter* (237)/*Content Filter* (342) that filters messages. If we thought there might still be confusion, we called the generic filter a *component*, which is a generic enough (and often abused enough) term that it should not get us into any trouble.

Pipes and Filters share some similarities with the concept of Communicating Sequential Processes (CSPs). Introduced by Hoare in 1978 [CSP], CSPs provide a simple model to describe synchronization problems that occur in parallel processing systems. The basic mechanism underlying CSPs is the synchronization of two processes via input-output (I/O). I/O occurs when process A indicates that it is ready to output to process B, and process B states that it is ready to input from process A. If one of these happens without the other being

true, the process is put on a wait queue until the other process is ready. CSPs are different from integration solutions in that they are not as loosely coupled, nor do the "pipes" provide any queuing mechanisms. Nevertheless, we can benefit from the extensive treatment of CSPs in the academic world.

Example: *Simple Filter in C# and MSMQ*

The following code snippet shows a generic base class for a filter with one input port and one output port. The base implementation simply prints the body of the received message and sends it to the output port. A more interesting filter would subclass the `Processor` class and override the `ProcessMessage` method to perform additional actions on the message—that is, transform the message content or route it to different output channels.

You notice that the `Processor` receives references to an input and output channel during instantiation. Thus, the class is tied to neither specific channels nor any other filter. This allows us to instantiate multiple filters and to chain them together in arbitrary configurations.

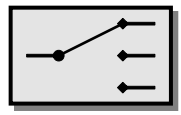
```
using System;
using System.Messaging;

namespace PipesAndFilters
{
    public class Processor
    {
        protected MessageQueue inputQueue;
        protected MessageQueue outputQueue;

        public Processor (MessageQueue inputQueue, MessageQueue outputQueue)
        {
            this.inputQueue = inputQueue;
            this.outputQueue = outputQueue;
        }

        public void Process()
        {
            inputQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnReceiveCompleted);
            inputQueue.BeginReceive();
        }

        private void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs asyncResult)
        {
            MessageQueue mq = (MessageQueue)source;

            Message inputMessage = mq.EndReceive(asyncResult.AsyncResult);
            inputMessage.Formatter =  new XmlMessageFormatter
                                      (new String[] {"System.String,mscorlib"});
```

```
            Message outputMessage = ProcessMessage(inputMessage);

            outputQueue.Send(outputMessage);

            mq.BeginReceive();
        }

        protected virtual Message ProcessMessage(Message m)
        {
            Console.WriteLine("Received Message: " + m.Body);
            return (m);
        }
    }
}
```


This implementation is an *Event-Driven Consumer* (498). The `Process` method registers for incoming messages and instructs the messaging system to invoke the method `OnReceiveCompleted` every time a message arrives. This method extracts the message data from the incoming event object and calls the virtual method `ProcessMessage`.

This simple filter example is not transactional. If an error occurs while processing the message (before it is sent to the output channel), the message is lost. This is generally not desirable in a production environment. See *Transactional Client* (484) for a solution to this problem.

Message Router



Message Router

Multiple processing steps in a *Pipes and Filters* (70) chain are connected by *Message Channel*s (60).

How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?

The *Pipes and Filters* (70) architectural style connects filters directly to each other with fixed pipes. This makes sense because many applications of the *Pipes and Filters* (70) pattern (e.g., [POSA]) are based on a large set of data items, each of which undergoes the same sequential processing steps. For example, a compiler will always execute the lexical analysis first, the syntactic analysis second, and the semantic analysis last. Message-based integration solutions, on the other hand, deal with individual messages that are not necessarily associated with a single, larger data set. As a result, individual messages are more likely to require a different series of processing steps.

A *Message Channel* (60) decouples the sender and the receiver of a *Message* (66). This also means that multiple applications can publish *Messages* (66) to a *Message Channel* (60). As a result, a *Message Channel* (60) can contain messages from different sources that may have to be treated differently based on the type of the message or other criteria. You could create a separate *Message Channel* (60) for each message type (a concept explained in more detail later as a *Datatype Channel* [111]) and connect each channel to the required processing steps for that message type. However, this would require the message originators to be aware of the selection criteria for different processing steps in order to publish the message to the correct channel. It could also lead to an explosion of the number of *Message Channel*s (60). Furthermore, the decision on which steps the message undergoes may not just depend on the origin of the message. For example, we could imagine a situation where the destination of a message

varies depending on the number of messages that have passed through the channel so far. No single originator would know this number and would therefore be unable to send the message to the correct channel.

*Message Channel*s (60) provide a very basic form of routing capabilities. An application publishes a *Message* (66) to a *Message Channel* (60) and has no further knowledge of that *Message*'s (66) destination. Therefore, the path of the *Message* (66) can change depending on which component subscribes to the *Message Channel* (60). However, this type of "routing" does not take into account the properties of individual messages. Once a component subscribes to a *Message Channel* (60), it will by default consume all messages from that channel regardless of the individual message's specific properties. This behavior is similar to the use of the pipe symbol in UNIX to process text files. It allows you to compose processes into a *Pipes and Filters* (70) chain, but for the lifetime of the chain, all lines of text undergo the same steps.

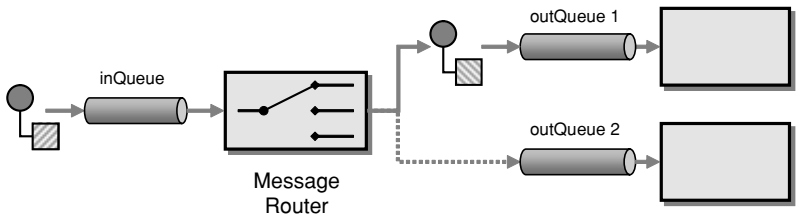
We could make the receiving component itself responsible for determining whether it should process a message that arrives on a common *Message Channel* (60). This is problematic, though, because once the message is consumed and the component determines that it does not want the message, it can't just put the message back on the channel for another component to check out. Some messaging systems allow receivers to inspect message properties without removing the message from the channel so that it can decide whether to consume the message. However, this is not a general solution and also ties the consuming component to a specific type of message because the logic for message selection is now built right into the component. This would reduce the potential for reuse of that component and eliminate the composability that is the key strength of the *Pipes and Filters* (70) model.

Many of these alternatives assume that we can modify the participating components to meet our needs. In most integration solutions, however, the building blocks (components) are large applications that in most cases cannot be modified at all—for example, because they are packaged applications or legacy applications. This makes it uneconomical or even impossible to adjust the message-producing or -consuming applications to the needs of the messaging system or other applications.

One advantage of *Pipes and Filters* (70) is the composability of the individual components. This property enables us to insert additional steps into the filter chain without having to change existing components. This opens up the option of decoupling two filters by inserting between them another filter that determines what step to execute next.

Message Router

Insert a special filter, a *Message Router*, which consumes a Message from one Message Channel and republishes it to a different Message Channel, depending on a set of conditions.



The *Message Router* differs from the basic notion of *Pipes and Filters* (70) in that it connects to multiple output channels (i.e., it has more than one output port). However, thanks to the *Pipes and Filters* (70) architecture, the components surrounding the *Message Router* are completely unaware of the existence of a *Message Router*. They simply consume messages off one channel and publish them to another. A defining property of the *Message Router* is that it does not modify the message contents; it concerns itself only with the destination of the message.

The key benefit of using a *Message Router* is that the decision criteria for the destination of a message are maintained in a single location. If new message types are defined, new processing components are added, or routing rules change, we need to change only the *Message Router* logic, while all other components remain unaffected. Also, since all messages pass through a single *Message Router*, incoming messages are guaranteed to be processed one by one in the correct order.

While the intent of a *Message Router* is to decouple filters from each other, using a *Message Router* can actually cause the opposite effect. The *Message Router* component must have knowledge of all possible destination channels in order to send the message to the correct channel. If the list of possible destinations changes frequently, the *Message Router* can turn into a maintenance bottleneck. In those cases, it would be better to let the individual recipients decide which messages they are interested in. You can accomplish this by using a *Publish-Subscribe Channel* (106) and an array of *Message Filters* (237). We contrast these two alternatives by calling them *predictive routing* and *reactive filtering* (for a more detailed comparison, see the *Message Filter* (237) in Chapter 7, "Message Routing").

Because a *Message Router* requires the insertion of an additional processing step, it can degrade performance. Many message-based systems have to decode the message from one channel before it can be placed on another channel, which causes computational overhead if the message itself does not really change. This overhead can turn a *Message Router* into a performance bottleneck. By using multiple routers in parallel or adding additional hardware, this effect can be minimized. As a result, the message throughput (number of messages processed per time unit) may not be impacted, but the latency (time for one message to travel through the system) will almost certainly increase.

Like most good tools, *Message Router*s can also be abused. Deliberate use of *Message Router*s can turn the advantage of loose coupling into a disadvantage. Loosely coupled systems can make it difficult to understand the "big picture" of the solution—the overall flow of messages through the system. This is a common problem with messaging solutions, and the use of routers can exacerbate the problem. If everything is loosely coupled to everything else, it becomes impossible to understand in which direction messages actually flow. This can complicate testing, debugging, and maintenance. A number of tools can help alleviate this problem. First, we can use the *Message History* (551) to inspect messages at runtime and see which components they traversed. Alternatively, we can compile a list of all channels to which each component in the system subscribes or publishes. With this knowledge we can draw a graph of all possible message flows across components. Many EAI packages maintain channel subscription information in a central repository, making this type of static analysis easier.

Message Router Variants

A *Message Router* can use any number of criteria to determine the output channel for an incoming message. The most trivial case is a fixed router. In this case, only a single input channel and a single output channel are defined. The fixed router consumes one message off the input channel and publishes it to the output channel. Why would we ever use such a brainless router? A fixed router may be useful to intentionally decouple subsystems so that we can insert a more intelligent router later. Or, we may be relaying messages between multiple integration solutions. In most cases, a fixed router will be combined with a *Message Translator* (85) or a *Channel Adapter* (127) to transform the message content or send the message over a different channel type.

Many *Message Router*s decide the message destination only on properties of the message itself—for example, the message type or the values of specific message fields. We call such a router a *Content-Based Router* (230). This type of

router is so common that the *Content-Based Router* (230) pattern describes it in more detail.

Other *Message Router*s decide the message's destination based on environment conditions. We call these routers *context-based router*s. Such routers are commonly used to perform load-balancing, test, or failover functionality. For example, if a processing component fails, the context-based router can reroute messages to another processing component and thus provide failover capability. Other routers split the flow of messages evenly across multiple channels to achieve parallel processing similar to a load balancer. Some *Message Channel*s (60) already provide basic load-balancing capabilities without the use of a *Message Router* because multiple *Competing Consumers* (502) can each consume messages off the same channel as fast as they can. However, a *Message Router* can have additional built-in intelligence to route the messages as opposed to a simple round-robin implemented by the channel.

Many *Message Router*s are *stateless*—in other words, they look at only one message at a time to make the routing decision. Other routers take the content of previous messages into account when making a routing decision. For example, the *Pipes and Filters* (70) example used a router that eliminates duplicate messages by keeping a list of all messages it already received. These routers are *stateful*.

Most *Message Router*s contain hard-coded logic for the routing decision. However, some variants connect to a *Control Bus* (540) so that the middleware solution can change the decision criteria without having to make any code changes or interrupting the flow of messages. For example, the *Control Bus* (540) can propagate the value of a global variable to all *Message Router*s in the system. This can be very useful for testing to allow the messaging system to switch from test to production mode. The *Dynamic Router* (243) configures itself dynamically based on control messages from each potential recipient.

Chapter 7, "Message Routing," introduces more variants of the *Message Router*.

Example: *Commercial EAI Tools*

The notion of a *Message Router* is central to the concept of a *Message Broker* (322), implemented in virtually all commercial EAI tools. These tools accept incoming messages, validate them, transform them, and route them to the correct destination. This architecture alleviates the participating applications from having to be aware of other applications altogether because the *Message Broker* (322) brokers between the applications. This is a key function in enterprise integration because most applications to be connected are packaged or legacy

applications and the integration has to happen nonintrusively—that is, without changing the application code. Therefore, the middleware has to incorporate all routing logic so the applications do not have to. The *Message Broker* (322) is the integration equivalent of a *Mediator* presented in [GoF].

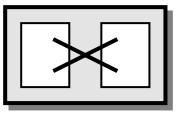
Example: *Simple Router with C# and MSMQ*

This code example demonstrates a very simple router that routes an incoming message to one of two possible output channels based on a simple condition.

```
class SimpleRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue1;
    protected MessageQueue outQueue2;

    public SimpleRouter(MessageQueue inQueue, MessageQueue outQueue1, MessageQueue outQueue2)
    {
        this.inQueue = inQueue;
        this.outQueue1 = outQueue1;
        this.outQueue2 = outQueue2;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }

    private void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        if (IsConditionFulfilled())
            outQueue1.Send(message);
        else
            outQueue2.Send(message);

        mq.BeginReceive();
    }

    protected bool toggle = false;

    protected bool IsConditionFulfilled ()
    {
        toggle = !toggle;
        return toggle;
    }

}
```

The code is relatively straightforward. Like the simple filter presented in *Pipes and Filters* (70), the `SimpleRouter` class implements an *Event-Driven Consumer* (498) of messages using C# delegates. The constructor registers the method `OnMessage` as the handler for messages arriving on the `inQueue`. This causes the .NET Framework to invoke the method `OnMessage` for every message that arrives on the `inQueue`. `OnMessage` figures out where to route the message by calling the method `IsConditionFulfilled`. In this trivial example, `IsConditionFulfilled` simply toggles between the two channels, dividing the sequence of messages evenly between `outQueue1` and `outQueue2`. In order to keep the code to a minimum, this simple router is not transactional—that is, if the router crashes after it consumes a message from the input channel and before it publishes it to the output channel, the message would be lost. *Transactional Client* (484) explains how to make endpoints transactional.

Message Translator



The previous patterns show how to construct messages and how to route them to the correct destination. In many cases, enterprise integration solutions route messages between existing applications such as legacy systems, packaged applications, homegrown custom applications, or applications operated by external partners. Each of these applications is usually built around a proprietary data model. Each application may have a slightly different notion of the Customer entity, the attributes that define a Customer, and other entities to which a Customer is related. For example, the accounting system may be more interested in the customer's taxpayer ID numbers, whereas the customer-relationship management (CRM) system stores phone numbers and addresses. The application's underlying data model usually drives the design of the physical database schema, an interface file format, or an application programming interface (API)—those entities with which an integration solution must interface. As a result, each application typically expects to receive messages that mimic the application's internal data format.

In addition to the proprietary data models and data formats incorporated in the various applications, integration solutions often interact with external business partners via standardized data formats that are independent from specific applications. A number of consortia and standards bodies define these protocols; for example, RosettaNet, ebXML, OAGIS, and many other industry-specific consortia. In many cases, the integration solution needs to be able to communicate with external parties using the "official" data formats, even though the internal systems are based on proprietary formats.

How can systems using different data formats communicate with each other using messaging?

We could avoid having to transform messages if we could modify all applications to use a common data format. This turns out to be difficult for a number of reasons (see *Shared Database* [47]). First, changing an application's data format is risky, difficult, and requires a lot of changes to inherent business

functionality. For most legacy applications, data format changes are simply not economically feasible. We may all remember the effort related to the Y2K retrofits, where the scope of the change was limited to the size of a single field!

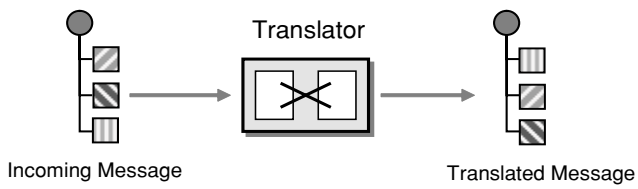
Also, while we may get multiple applications to use the same data field names and maybe even the same data types, the physical representation may still be quite different. One application may use XML documents, whereas the other application uses COBOL copybooks.

Message Translator

Furthermore, if we adjust the data format of one application to match that of another application, we are tying the two applications more tightly to each other. One of the key architectural principles in enterprise integration is loose coupling between applications (see *Canonical Data Model* [355]). Modifying one application to match another application's data format would violate this principle because it makes two applications directly dependent on each other's internal representation. This eliminates the possibility of replacing or changing one application without affecting the other application, a scenario that is fairly common in enterprise integration.

We could incorporate the data format translation directly into the *Message Endpoint* (95). This way, all applications would publish and consume messages in a common format as opposed to in the application's internal data format. However, this approach requires access to the endpoint code, which is usually not the case for packaged applications. In addition, hard-coding the format translation to the endpoint would reduce the opportunities for code reuse.

Use a special filter, a *Message Translator*, between other filters or applications to translate one data format into another.



The *Message Translator* is the messaging equivalent of the *Adapter* pattern described in [GoF]. An adapter converts the interface of a component into another interface so it can be used in a different context.

Levels of Transformation

Message translation may need to occur at a number of different levels. For example, data elements may share the same name and data types but may be used in different representations (e.g., XML file vs. comma-separated values vs. fixed-length fields). Or, all data elements may be represented in XML format but use different tag names. To summarize the different kinds of translation, we can divide it into multiple layers (loosely borrowing from the OSI Reference Model).

Layer	Deals With	Transformation Needs (Example)	Tools/ Techniques
Data Structures (Application Layer)	Entities, associations, cardinality	Condense many-to-many relationship into aggregation.	Structural mapping patterns, custom code
Data Types	Field names, data types, value domains, constraints, code values	Convert ZIP code from numeric to string. Concatenate First Name and Last Name fields to single Name field. Replace U.S. state name with two-character code.	EAI visual transformation editors, XSL, database lookups, custom code
Data Representation	Data formats (XML, name-value pairs, fixed-length data fields, EAI vendor formats, etc.)	Parse data representation and render in a different format.	XML parsers, EAI parser/ renderer tools, custom APIs
	Character sets (ASCII, UniCode, EBCDIC)	Decrypt/encrypt as necessary.	
	Encryption/compression		
Transport	Communications protocols: TCP/IP sockets, HTTP, SOAP, JMS, TIBCO RendezVous	Move data across protocols without affecting message content.	*Channel Adapter* (127), EAI adapters

The *Transport* layer at the bottom of the "stack" provides data transfer between the different systems. It is responsible for complete and reliable data transfer across different network segments and deals with lost data packets and other network errors. Some EAI vendors provide their own transport protocols (e.g., TIBCO RendezVous), whereas other integration technologies leverage

TCP/IP protocols (e.g., SOAP). Translation between different transport layers can be provided by the *Channel Adapter* (127) pattern.

The *Data Representation* layer is also referred to as the *syntax layer*. This layer defines the representation of data that is transported. This translation is necessary because the transport layer typically transports only character or byte streams. This means that complex data structures have to be converted into a character string. Common formats for this conversion include XML, fixed-length fields (e.g., EDI records), and proprietary formats. In many cases, data is also compressed or encrypted and carries check digits or digital certificates. In order to interface systems with different data representations, data may have to be decrypted, uncompressed, and parsed, and then the new data format must be rendered and possibly compressed and encrypted as well.

Message Translator

The *Data Types* layer defines the application data types on which the application (domain) model is based. Here we deal with such decisions as whether date fields are represented as strings or as native date structures, whether dates carry a time-of-day component, which time zone they are based on, and so on. We may also consider whether the field *Postal Code* denotes only a U.S. ZIP code or can contain Canadian postal codes. In the case of a U.S. zip code, do we include a ZIP+4? Is it mandatory? Is it stored in one field, or two? Many of these questions are usually addressed in so-called *Data Dictionaries*. The issues related to data types go beyond whether a field is of type *string* or *integer*. Consider sales data that is organized by region. The application used by one department may divide the country into four regions: West, Central, South, and East, identified by the letters W, C, S, and E. Another department may differentiate the Pacific region from the mountain region and distinguish the Northeast from the Southeast. Each region is identified by a two-digit number. What number does the letter E correspond to?

The *Data Structures* layer describes the data at the level of the application domain model. It is therefore also referred to as the *application layer*. This layer defines the logical entities that the application deals with, such as *customer, address,* or *account*. It also defines the relationships between these entities: Can one customer have multiple accounts? Can a customer have multiple addresses? Can customers share an address? Can multiple customers share an account? Is the address part of the account or the customer? This is the domain of entity-relationship diagrams and class diagrams.

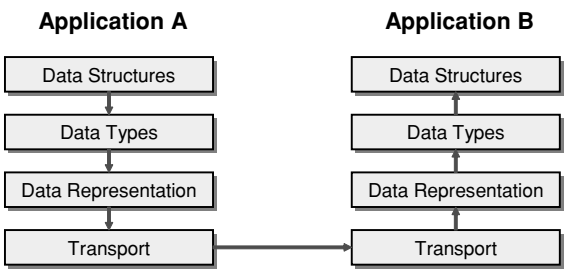
Levels of Decoupling

Many of the design trade-offs in integration are driven by the need to decouple components or applications. Decoupling is an essential tool to enable the man-

agement of change. Integration typically connects existing applications and has to accommodate changes to these applications. *Message Channels* (60) decouple applications from having to know each other's location. A *Message Router* (78) can even decouple applications from having to agree on a common *Message Channel* (60). However, this form of decoupling achieves only limited independence between applications if they still depend on each other's data formats. A *Message Translator* (85) can remove this additional level of dependency.

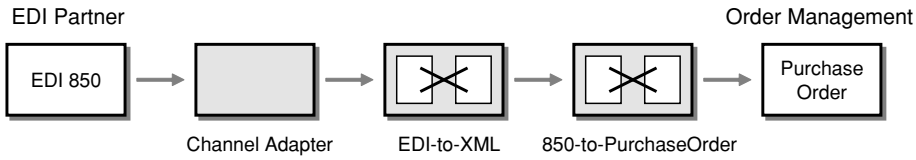
Chaining Transformations

Many business scenarios require transformations at more than one layer. For example, let's assume an EDI 850 Purchase Order record represented as a fixed-format file has to be translated to an XML document sent over HTTP to the order management system, which uses a different definition of the `Order` object. The required transformation spans all four levels: The transport changes from file transfer to HTTP, the data format changes from a fixed-field format to XML, and both data types and data formats have to be converted to comply with the `Order` object defined by the order management system. The beauty of a layered model is that you can treat one layer without worrying about the lower layers and therefore can focus on one level of abstraction at a time (see the following figure).



Mapping Across Multiple Layers

Chaining multiple *Message Translator* units using *Pipes and Filters* (70) results in the following architecture (see figure on the next page). Creating one *Message Translator* for each layer allows us to reuse these components in other scenarios. For example, the *Channel Adapter* (127) and the EDI-to-XML *Message Translator* can be implemented in a generic fashion so that they can be reused for any incoming EDI document.



Chaining Multiple Message Translators *(85)*

Message Translator

Chaining multiple *Message Translator*s also allows you to change the transformations used at an individual layer without affecting any of the other layers. You could use the same structural transformation mechanisms, but instead of converting the data representation into a fixed format, you could convert it into a comma-separated file by swapping out the data representation transformation.

There are many specializations and variations of the *Message Translator* pattern. An *Envelope Wrapper* (330) wraps the message data inside an envelope so that it can be transported across a messaging system. A *Content Enricher* (336) augments the information inside a message, whereas the *Content Filter* (342) removes information. The *Claim Check* (346) removes information but stores it for later retrieval. The *Normalizer* (352) can convert a number of different message formats into a consistent format. Last, the *Canonical Data Model* (355) shows how to leverage multiple *Message Translator*s to achieve data format decoupling. Inside each of those patterns, complex structural transformations can occur (e.g., mapping a many-to-many relationship into a one-to-one relationship).

Example: *Structural Transformation with XSL*

Transformation is such a common need that the W3C defined a standard language for the transformation of XML documents: the Extensible Stylesheet Language (XSL). Part of XSL is the XSL Transformation (XSLT) language, a rules-based language that translates one XML document into a different format. Since this is a book on integration and not on XSLT, we just present a simple example (for all the gory details, see the spec [XSLT 1.0], or to learn by reviewing code examples, see [Tennison]). In order to keep things simple, we explain the required transformation by showing example XML documents as opposed to XML schemas.

For example, let's assume we have an incoming XML document and need to pass it to the accounting system. If both systems use XML, the Data Represen-

tation layer is identical, and we need to cover any differences in field names, data types, and structure. Let's assume the incoming document looks like this.

```
<data>
    <customer>
        <firstname>Joe</firstname>
        <lastname>Doe</lastname>
        <address type="primary">
            <ref id="55355"/>
        </address>
        <address type="secondary">
            <ref id="77889"/>
        </address>
    </customer>
    <address id="55355">
        <street>123 Main</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94123</postalcode>
        <country>USA</country>
        <phone type="cell">
            <area>415</area>
            <prefix>555</prefix>
            <number>1234</number>
        </phone>
        <phone type="home">
            <area>415</area>
            <prefix>555</prefix>
            <number>5678</number>
        </phone>
    </address>
    <address id="77889">
        <company>ThoughtWorks</company>
        <street>410 Townsend</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94107</postalcode>
        <country>USA</country>
    </address>
</data>
```

This XML document contains customer data. Each customer can be associated with multiple addresses, each of which can contain multiple phone numbers. The XML represents addresses as independent entities so that multiple customers could share an address.

Let's assume the accounting system needs the following representation. (If you think that the German tag names are bit farfetched, keep in mind that one of the most popular pieces of enterprise software (SAP) is famous for its German field names!)

```
<Kunde>
    <Name>Joe Doe</Name>
    <Adresse>
        <Strasse>123 Main</Strasse>
        <Ort>San Francisco</Ort>
        <Telefon>415-555-1234</Telefon>
    </Adresse>
</Kunde>
```

Message Translator

The resulting document has a much simpler structure. Tag names are different, and some fields are merged into a single field. Since there is room for only one address and phone number, we need to pick one from the original document based on business rules. The following XSLT program transforms the original document into the desired format. It does so by matching elements of the incoming document and translating them into the desired document format.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:key name="addrlookup" match="/data/address" use="@id"/>
    <xsl:template match="data">
        <xsl:apply-templates select="customer"/>
    </xsl:template>
    <xsl:template match="customer">
        <Kunde>
            <Name>
                <xsl:value-of select="concat(firstname, ' ', lastname)"/>
            </Name>
            <Adresse>
                <xsl:variable name="id" select="./address[@type='primary']/ref/@id"/>
                <xsl:call-template name="getaddr">
                    <xsl:with-param name="addr" select="key('addrlookup', $id)"/>
                </xsl:call-template>
            </Adresse>
        </Kunde>
    </xsl:template>
    <xsl:template name="getaddr">
        <xsl:param name="addr"/>
        <Strasse>
            <xsl:value-of select="$addr/street"/>
        </Strasse>
        <Ort>
            <xsl:value-of select="$addr/city"/>
        </Ort>
        <Telefon>
            <xsl:choose>
                <xsl:when test="$addr/phone[@type='cell']">
                    <xsl:apply-templates select="$addr/phone[@type='cell']" mode="getphone"/>
                </xsl:when>
```

```
            <xsl:otherwise>
                <xsl:apply-templates select="$addr/phone[@type='home']" mode="getphone"/>
            </xsl:otherwise>
        </xsl:choose>
    </Telefon>
</xsl:template>
<xsl:template match="phone" mode="getphone">
    <xsl:value-of select="concat(area, '-', prefix, '-', number)"/>
</xsl:template>
<xsl:template match="*"/>
</xsl:stylesheet>
```


XSL is based on pattern matching and can be a bit hairy to read if you are used to procedural programming like most of us. In a nutshell, the instructions inside an `<xsl:template>` element are called whenever an element in the incoming XML document matches the expression specified in the `match` attribute. For example, the line

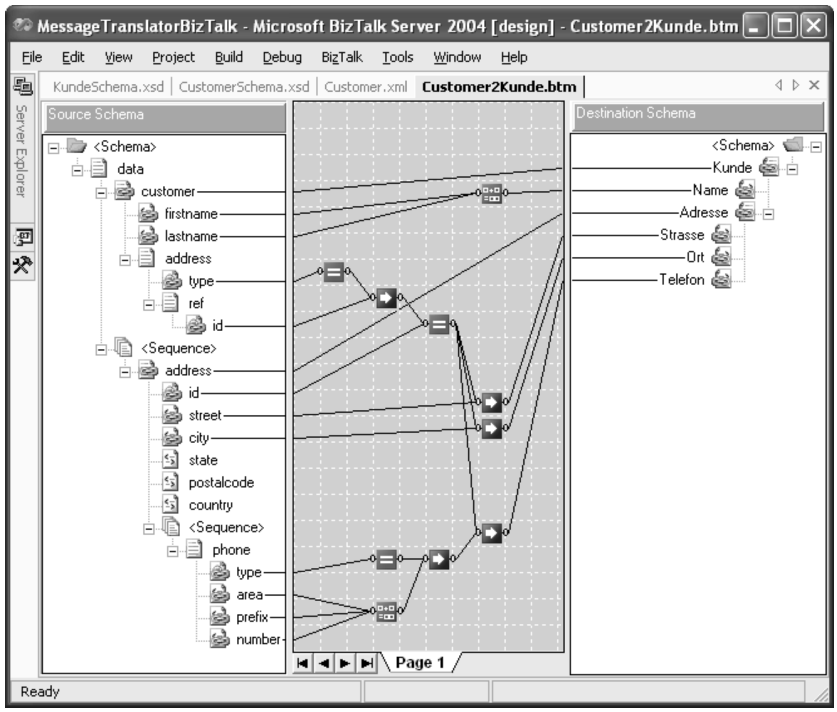
```
<xsl:template match="customer">
```

causes the subsequent lines to be executed for each `<customer>` element in the source document. The next statements concatenate first and last name and output it inside the `<Name>` element. Getting the address is a little trickier. The XSL code looks up the correct instance of the `<address>` element and calls the subroutine `getaddr`. `getaddr` extracts the address and phone number from the original `<address>` element. It uses the cell phone number if one is present, or the home phone number otherwise.

Example: *Visual Transformation Tools*

If you find XSL programming a bit cryptic, you are in good company. Therefore, most integration vendors provide a visual transformation editor that displays the structure of the two document formats on the left-hand side and the right-hand side of the screen respectively. The users can then associate elements between the formats by drawing connecting lines between them. This can be a lot simpler than coding XSL. Some vendors, such as Contivo, specialize entirely in transformation tools.

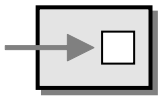
The following figure shows the Microsoft BizTalk Mapper editor that is integrated into Visual Studio. The diagram shows the mapping between individual elements more clearly than the XSL script. On the other hand, some of the details (e.g., how the address is chosen) are hidden underneath the so-called functoid icons.



Creating Transformations: The Drag-Drop Style

Being able to drag and drop transformations shortens the learning curve for developing a *Message Translator* dramatically. As so often though, visual tools can also become a liability when it comes to debugging or when you need to create complex solutions. Therefore, many tools let you switch back and forth between XSL and the visual representation.

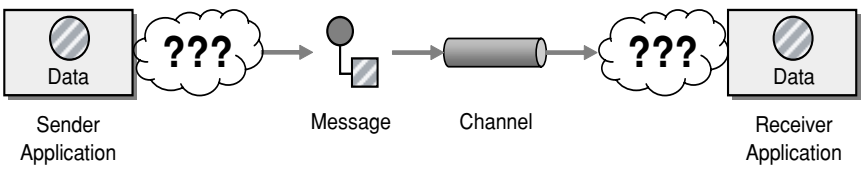
Message Endpoint



Applications are communicating by sending *Messages* (66) to each other via *Message Channels* (60).

How does an application connect to a messaging channel to send and receive Messages?

The application and the messaging system are two separate sets of software. The application provides functionally for some type of user, whereas the messaging system manages messaging channels for transmitting messages for communication. Even if the messaging system is incorporated as a fundamental part of the application, it is still a separate, specialized provider of functionality, much like a database management system or a Web server. Because the application and the messaging system are separate, they must have a way to connect and work together.



Applications Disconnected from a Message Channel

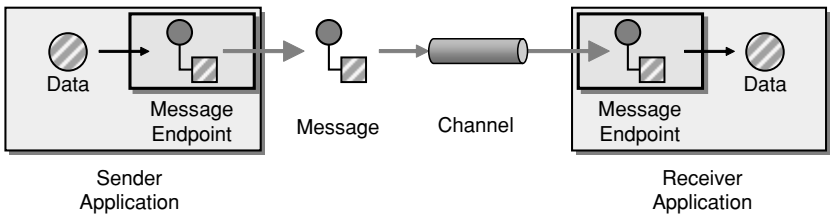
A messaging system is a type of server, capable of taking requests and responding to them. Like a database accepting and retrieving data, a messaging server accepts and delivers messages. A messaging system is a messaging server.

A server needs clients, and an application that uses messaging is a client of the messaging server. But applications do not necessarily know how to be messaging clients any more than they know how to be database clients. The messaging server, like a database server, has a client API that the application can use to interact with the server. The API is not application-specific but is

domain-specific, where the domain is messaging. The application must contain a set of code that connects and unites the messaging domain with the application to allow the application to perform messaging.

Message Endpoint

Connect an application to a messaging channel using a *Message Endpoint*, a client of the messaging system that the application can then use to send or receive Messages.



Message Endpoint code is custom to both the application and the messaging system's client API. The rest of the application knows little about message formats, messaging channels, or any of the other details of communicating with other applications via messaging. It just knows that it has a request or piece of data to send to another application, or is expecting those from another application. It is the messaging endpoint code that takes that command or data, makes it into a message, and sends it on a particular messaging channel. It is the endpoint that receives a message, extracts the contents, and gives them to the application in a meaningful way.

The *Message Endpoint* encapsulates the messaging system from the rest of the application and customizes a general messaging API for a specific application and task. If an application using a particular messaging API were to switch to another, developers would have to rewrite the message endpoint code, but the rest of the application should remain the same. If a new version of a messaging system changes the messaging API, this should only affect the message endpoint code. If the application decides to communicate with others via some means other than messaging, developers should ideally be able to rewrite the message endpoint code but leave the rest of the application unchanged.

A *Message Endpoint* can be used to send messages or receive them, but one instance does not do both. An endpoint is channel-specific, so a single application would use multiple endpoints to interface with multiple channels. An

application may use multiple endpoint instances to interface to a single channel, usually to support multiple concurrent threads.

A *Message Endpoint* is a specialized *Channel Adapter* (127) one that has been custom developed for and integrated into its application.

A *Message Endpoint* should be designed as a *Messaging Gateway* (468) to encapsulate the messaging code and hide the message system from the rest of the application. It can employ a *Messaging Mapper* (477) to transfer data between domain objects and messages. It can be structured as a *Service Activator* (532) to provide asynchronous message access to a synchronous service or function call. An endpoint can explicitly control transactions with the messaging system as a *Transactional Client* (484).

Sending messages is pretty easy, so many endpoint patterns concern different approaches for receiving messages. A message receiver can be a *Polling Consumer* (494) or an *Event-Driven Consumer* (498). Multiple consumers can receive messages from the same channel either as *Competing Consumers* (502) or via a *Message Dispatcher* (508). A receiver can decide which messages to consume or ignore using a *Selective Consumer* (515). It can use a *Durable Subscriber* (522) to make sure a subscriber does not miss messages published while the endpoint is disconnected. And the consumer can be an *Idempotent Receiver* (528) that correctly detects and handles duplicate messages.

Example: *JMS Producer and Consumer*

In JMS, the two main endpoint types are `MessageProducer`, for sending messages, and `MessageConsumer`, for receiving messages. A *Message Endpoint* uses an instance of one of these types to either send messages to or receive messages from a particular channel.

Example: *.NET MessageQueue*

In .NET, the main endpoint class is the same as the main *Message Channel* (60) class, `MessageQueue`. A *Message Endpoint* uses an instance of `MessageQueue` to send messages to or receive messages from a particular channel.