Table of Content

3. Messaging Systems

Introduction

In *Introduction to Integration Styles*, we discussed the various options for connecting applications with one another, including *Messaging*. 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 but not worry so much about how to share it.

Basic Messaging Concepts

Like most technologies, *Messaging* 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. These basic messaging concepts are:

Channels — Messaging applications transmit data through a *Message Channel*, a virtual pipe that connects a sender to a receiver. A newly installed messaging system 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* 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 try repeatedly to deliver the message (e.g., transmit it from the sender to the receiver) until it succeeds.

Multi-step delivery — In the simplest case, the message system delivers a message directly from the sender's computer to the receiver's computer. However, 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 different message format than the sender. A *Pipes and Filters* 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*, an application component and filter in the pipes-and-filters architecture, which

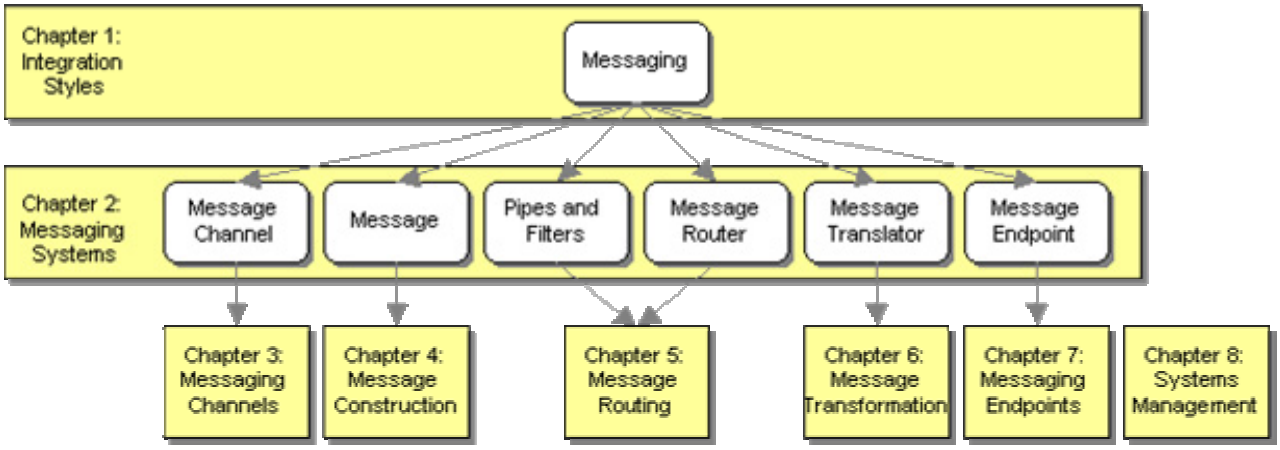
will determine how to navigate the channel topology and direct the message to the final receiver, or at least to the next router.

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

Endpoints — An application does not have some built-in capability to interface with a messaging system. Rather, it 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 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*. All subsequent chapters build upon the base patterns in this chapter.



Relationship of Root Patterns and Chapters

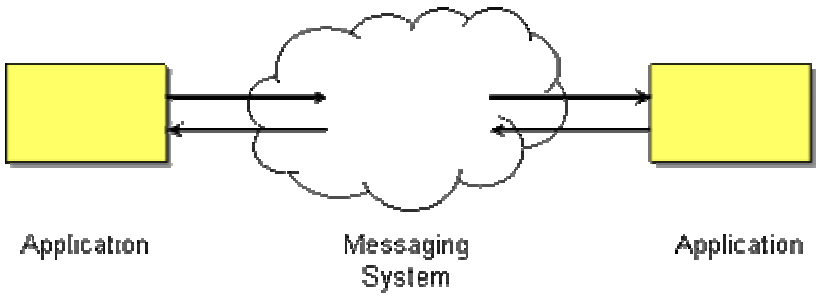
This chapter provides a broad overview of *Messaging* by introducing the main messaging topics. For more details about one of these topics, skip ahead to the chapter that contains more patterns which cover that topic in greater depth.

Message Channel

An enterprise has two separate applications that need to communicate, preferably by using *Messaging*.

How does one application communicate with another using messaging?

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

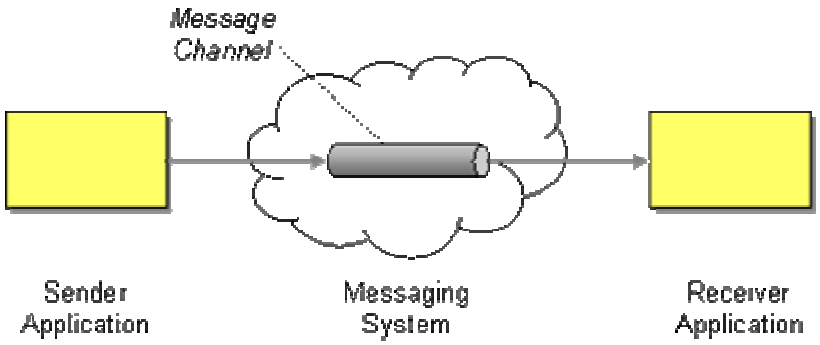


Applications Magically Connected

Likewise, it's not like an application just randomly throws information out into the messaging system while other applications just randomly grab whatever information they run across. (Even if this would work, it'd be very inefficient.) 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 sorts 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 enable 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, it 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 adding info doesn't necessarily know what particular application will end up retrieving the info, but it can be assured that whatever application retrieves the info, that application will be interested in the info. 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 info to any channel available; it adds the info 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* 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 will be needed 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. While 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 have to know about the new channel so that 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 *Introduction to Messaging Channels*.)

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 JMS code that includes calling the method `createQueue`, or .NET code that includes `new MessageQueue`, but neither of these bits of code actually allocates a new queue resource in the messaging system. Rather, these pieces of code provide access to a resource that already exists in the messaging system and was already created in the messaging system separately using its administration tools.

Another issue to 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 has unlimited memory and disk space, any messaging system implementation usually has 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. *Datatype Channel* helps you determine when you need another channel. *Selective Consumer* makes one physical channel act logically like multiple channels.

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. Another popular term is *Producer* and *Consumer*. Equally popular is *Publisher* and *Subscriber* — they are more geared towards *Publish-Subscribe Channel*s, but are often times used in generic form. Sometimes we also say that an application *listens* on a channel that another application *talks* to. In the world of Web services, we generally talk about a *Requester* and *Provider*. These terms usually imply that the requester sends a message to the provider and receives a response back. In the old days we called these *Client* and *Server* — the terms are equivalent, but saying client and server is less cool. Now it gets gets confusing: when dealing with Web services, sometimes the application that sends a message to the provider is considered a *Consumer* of the service. We can think of it in such a way that consumer sends a message to the provider and then consumes the response. Luckily, usage of the term in this way occurs only in RPC 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*.

Channel Names

If channels are logical addresses, what do these addresses look like? Like 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 and *Publish-Subscribe Channel*s. Mixing different data types on the same channel causes a lot of confusion; to avoid this confusion, use separate *Datatype Channel*s. Applications that use messaging often benefit from a special channel for invalid messages, an *Invalid Message Channel*. Applications that wish to use *Messaging* but do not have access to a messaging client can still connect to the messaging system using *Channel Adapter*s. A well designed set of channels forms a *Message Bus* 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 ones to 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 then 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 loookup 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)
```

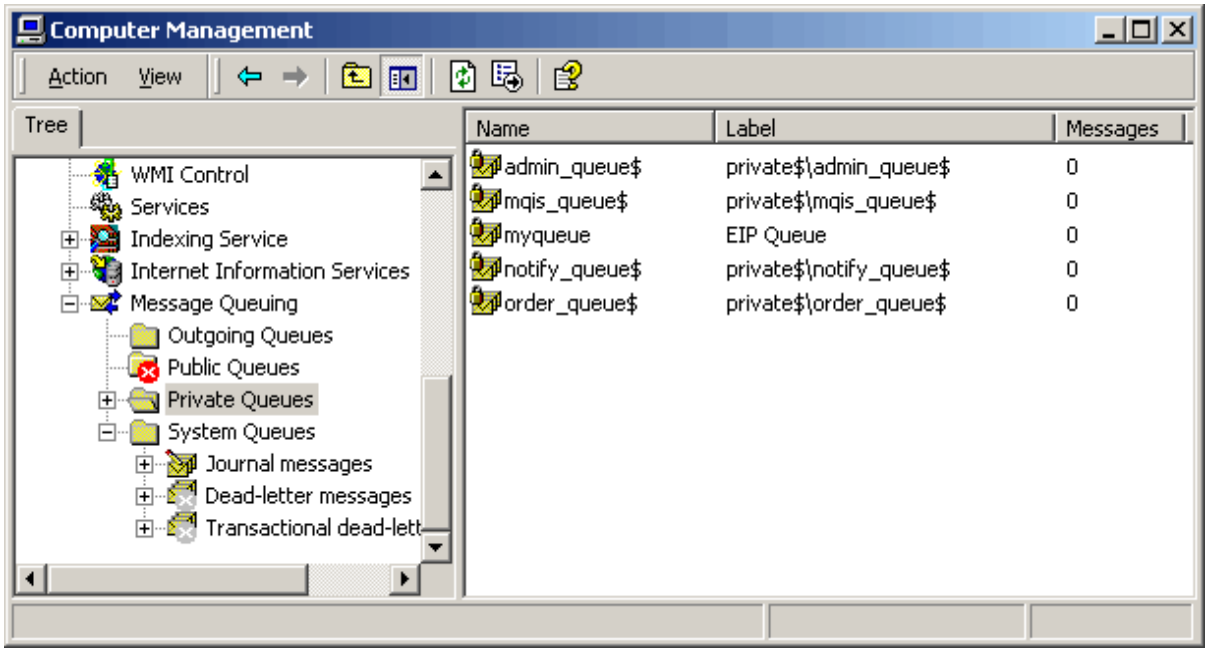
Once that queue exists in WebSphere MQ, an application can then access the queue.

WebSphere MQ, without the full WebSphere Application Server, does not include a JNDI implementation, so we cannot use JNDI to lookup 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 picture). 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:

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

Related patterns: *Channel Adapter*, *Datatype Channel*, *Invalid Message Channel*, *Message Bus*, *Message Endpoint*, *Selective Consumer*, *Messaging*, *Introduction to Messaging Channels*, *Point-to-Point Channel*, *Publish-Subscribe Channel*

Message

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

How can two applications connected by a message channel exchange a piece of information?

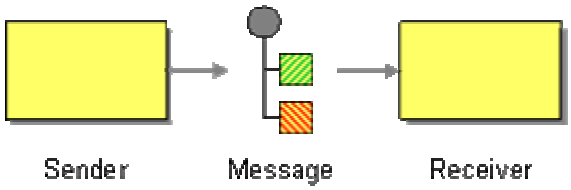
A *Message Channel* can often be thought of as a pipe, a conduit from one application to another. It might stand to reason then that data could be shoved in one end, like water, and it would come flowing out the other end. But data isn't one continuous stream; it is units, such as records, objects, database 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, which means that the first process must *marshal* the data into byte form, copy it from the first process to the second one, which will then *unmarshal* the data back into its original form, a copy of the original data in the first process. Marshalling is how an RPC sends arguments to the remote process, and how the process returns the result.

So messaging transmits discrete units of data, and does so by marshalling the data from the sender and unmarshalling 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; 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, i.e., different application styles of use. Use a *Command Message* to invoke a procedure in another application. Use a *Document Message* to pass a set of data to another application. Use

an *Event Message* to notify another application of a change in this application. If the other application should send a reply back, use *Request-Reply*.

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*. If the data is only useful for a limited amount of time, specify this use-by time as a *Message Expiration*. 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*.

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.

- `TextMessage` – The most common type of message. The body is a `String`, such as a text file or an XML document. `textMessage.getText()` returns a `String`.
- `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.
- `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`.
- `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.
- `MapMessage` – The body acts like a `java.util.Map`, where the keys are `String`s. 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`, is an `int` that specifies the type of data the body contains, such as a string, a date, a currency, or a number.

Example: SOAP Message

In the SOAP protocol [SOAP 1.1], a SOAP message is an example of this Message pattern. 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 recurssive nature of messages, because a SOAP message can be transmitted via a messaging system, which means that a messaging system message (such as `javax.jms.Message` or `System.Messaging.Message`) contains the SOAP message (the XML `SOAP-ENV:Envelope` document). In this scenario, the transport protocol isn't HTTP, it's the messaging system (which in turn may be using HTTP or some other network protocol to transmit the data, but the messaging system makes the transmission reliable).

Related patterns: *Canonical Data Model*, *Command Message*, *Document Message*, *Event Message*, *Message Channel*, *Message Expiration*, *Message Sequence*, *Messaging*, *Request-Reply*

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 stream of possibly duplicated, encrypted messages containing extra authentication data into a stream 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?

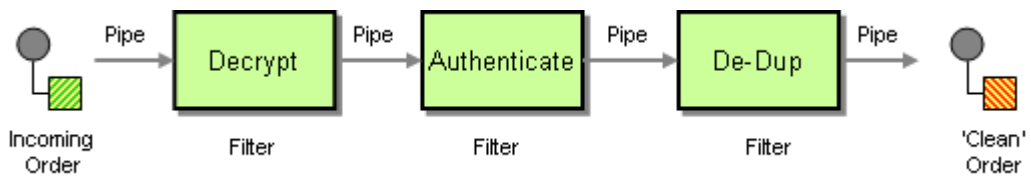
Implementing all functions inside a single component also reduces opportunities for reuse. Creating smaller, well-defined components allows us to reuse them in other processes. For example, order status messages may be encrypted but do not need to be de-duped because duplicate status requests are generally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

Integration solutions are typically a collection of heterogeneous systems. As a result, different processing steps may need to execute on different physical machines, for example because individual processing steps can only execute on a 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 while 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, on 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 component 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 *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 picture). 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 for a total of four pipes.

Pipes and Filters describe 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, e.g. routing and transformation patterns, are based on this *Pipes and Filters* architectural style. This allows us to 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 the *Message Channel* we just described at the beginning of this chapter. Most *Message Channel*s provide 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* 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* or an alternative implementation such as an in-memory queue. The *Messaging Gateway* 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 as 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.

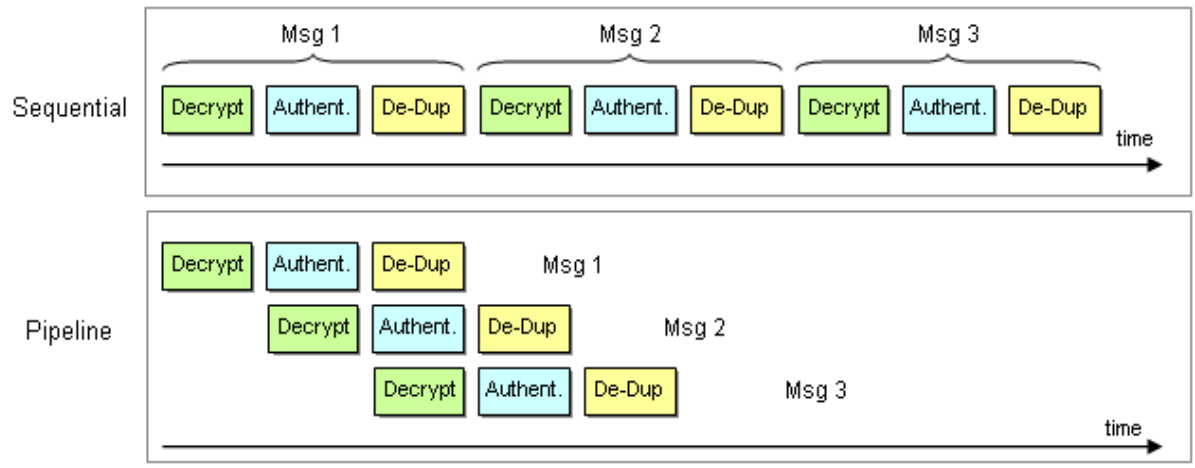
The pure form of *Pipes and Filters* allows each filter to have only a single input port and a single output port. When dealing with *Messaging* we can relax this property somewhat. A component may consume messages off more than one channel and also output messages to more than one

channel (for example, a *Message Router*). Likewise, multiple filter components can consume messages off a single *Message Channel*. A *Point-to-Point Channel* 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 steps by passing a *Test Message* to the component and comparing the results 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

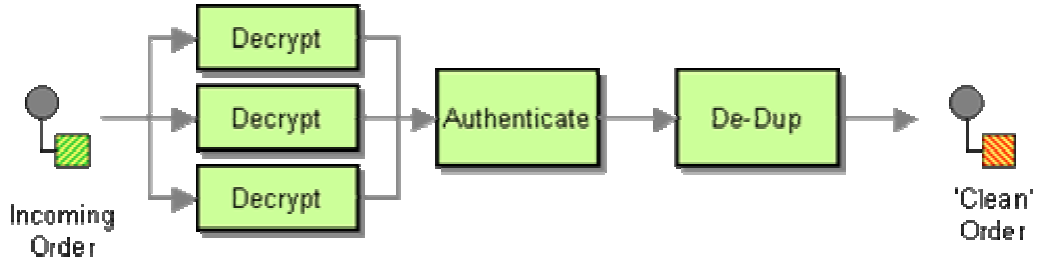
Connecting components with asynchronous *Message Channel*s allows each unit in the chain to operate in its own thread or its own process. When a unit has completed processing one message it can send the message to the output channel and immediately start processing another message. It does not have to 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 picture). 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. To improve throughput we can deploy multiple parallel instances of that process to improve throughput. In this scenario, a *Point-to-Point Channel* with *Competing Consumers* 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. However, this configuration can cause messages to be processed out of order. If the sequence of messages is critical, we can only run one instance of each component or use a *Resequencer*.



For example, if we assume that decrypting a message is much slower than authenticating it, we can use the above configuration (see picture), running three parallel instances of the decryption component. Parallelizing filters works best if each filter is stateless, i.e. it returns to the previous state after a message has been processed. This means that we cannot easily run multiple parallel de-dup 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 [Kahn]. [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 Buschmann 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 have to be 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*.

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.

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* and the *Content Filter*. 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 be renaming the *Pipes and Filters* architectural style. However, we felt that *Pipes and Filters* are 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 try to make it clear whether we are talking about a generic filter a la *Pipes and Filters* or a *Message Filter* / *Content Filter* to filter messages. In places where we felt that there is still room for confusion, we generally termed the generic filter as 'component' which is a generic enough (and often enough abused) term that should not get us into trouble.

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, e.g. transform the message content or route it to different output channels.

You notice that the `Processor` requires references to an input and output channel in order to be instantiated. The class is not tied to specific channels nor any other filter. This allows us to instantiate multiple filters and 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 System.Messaging.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 a *Event-Driven Consumer*. 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* for a solution to this problem.

Related patterns: *Competing Consumers*, *Content Filter*, *Event-Driven Consumer*, *Message Filter*, *Message Channel*, *Message Router*, *Messaging*, *Messaging Gateway*, *Point-to-Point Channel*, *Resequencer*, *Test Message*, *Transactional Client*

Message Router

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

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* architectural style connects filters directly to one another with fixed pipes. This makes sense because many applications of the *Pipes and Filters* 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 least. Message-based integration solutions, on the other hand, deal with individual messages which 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* decouples the sender and the receiver of a *Message*. This means that multiple applications can publish *Message*s to a *Message Channel*. As a result, a message channel 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* for each message type (a concept explained in more detail later as a *Datatype Channel*) 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, so that they can publish the message to the correct channel. It could also lead to an explosion of the number of *Message Channel*s. Also, 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 changes by 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 provide a very basic form of routing capabilities. An application publishes a *Message* to a *Message Channel* and has no further knowledge of that *Message*'s destination. Therefore, the path of the *Message* can change depending on which component subscribes to the *Message Channel*. However, this type of 'routing' does not take into account the properties of individual messages. Once a components subscribes to a *Message Channel* it will by default consume all messages from that channel regardless of the individual messages' specific properties. This behavior is similar to the use of the pipe symbol in Unix. It allows you to

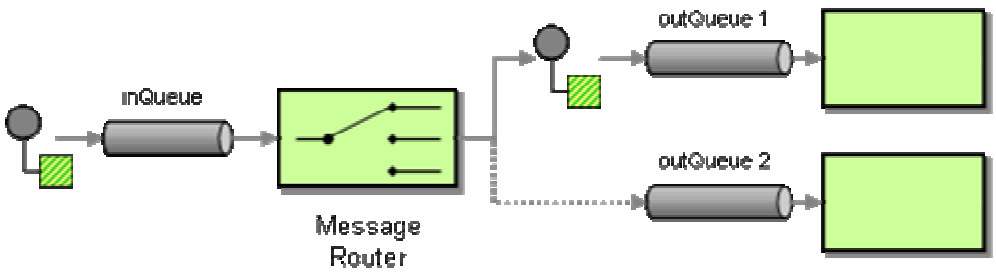
compose processes into a *Pipes and Filters* chain but for the lifetime of the chain all lines of text undergo the same steps.

We could solve this problem by making the receiving component itself responsible for determining whether it should process the message or not. 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 or not. However, this is not a general solution and will also tie 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* model.

Many of these alternatives assume that we can modify the participating components. In most integration solutions, however, the building blocks ('components') are large applications which 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.

An advantage of the *Pipes and Filters* is the composability of the individual components. It allows us to insert additional steps into the chain without having to change existing components. This opens up the option of decoupling two filters by inserting another filter in between that determines what step to execute next.

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



The *Message Router* differs from the most basic notion of *Pipes and Filters* in that it connects to multiple output channels. Thanks to the *Pipes and Filters* architecture the components surrounding the *Message Router* are completely unaware of the existence of a *Message Router*. A key property of the *Message Router* is that it does not modify the message contents. It only concerns itself 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 the routing rules change, we need to change only the *Message Router*

logic and 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, using a *Message Router* can actually cause the opposite effect. The *Message Router* needs to have knowledge of all possible message destinations in order to send the message to the correct channel. In some situations, the list of possible destinations may change frequently and turn the *Message Router* 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* and an array of *Message Filter*s. We contrast these two alternatives by calling them *predictive routing* and *reactive filtering* (for more detail see *Message Filter*).

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.

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, i.e. 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 which way messages actually flow. This can complicate testing and debugging and maintenance. A number of tools can help alleviate this problem. First, we can use the *Message History* to inspect messages at runtime and see which components they traversed. Alternatively, we can compile a list of all channels that each component in the system subscribes or publishes to. 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. Or we may be relaying messages between multiple integration solutions. In most cases, a fixed router will be combined with a *Message Translator* or a *Channel Adapter* 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*. This type of router is so common that the *Content-Based Router* 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 Routers*. 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 re-route message to another processing component and thus provide fail-over capability. Other routers split messages evenly across multiple channels to achieve parallel processing similar to a load balancer. A *Message Channel* already provides basic load balancing capabilities without the use of a *Message Router* because multiple competing consumers 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*, i.e. they only look at 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, we can envision 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* 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* 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* configures itself dynamically based on control messages from each potential recipient.

Chapter *Introduction to Message Routing* introduces further variants of the *Message Router*.

Example: Commercial EAI Tools

The notion of a *Message Router* is central to the concept of a *Message Broker*, 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 brokers between the applications. This is a key function in EAI because most applications to be connected are packaged or legacy applications and the integration has to happen non-intrusively, i.e. without changing the application code. This requires the middleware to incorporate all routing logic so the applications do not have to. The *Message Broker* 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.

```
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. The example implements an event-driven consumer 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 messages evenly between `outQueue1` and `outQueue2`. In order to keep the code to a minimum, this simple router is not transactional, i.e. if the router crashes after it consumed a message from the input channel and before it published it to the output channel, we would lose a message. Later chapters will explain how to make endpoints transactional (see *Transactional Client*).

Related patterns: *Channel Adapter*, *Content-Based Router*, *Control Bus*, *Datatype Channel*, *Dynamic Router*, *Message Filter*, *Message*, *Message Channel*, *Message History*, *Introduction to Message Routing*, *Message Translator*, *Pipes and Filters*, *Publish-Subscribe Channel*, *Transactional Client*

Message Translator

The previous patterns describe 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 which other entities a *Customer* is related to. For example, the accounting system may be more interested in the customer's tax payer ID numbers while 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 a programming interface (API) -- those entities that an integration solution has to interface with. As a result, the applications expect 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 times interact with standardized data formats that seek to be independent from specific applications. There are a number of consortia and standards bodies that define these protocols, such as 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 while 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*). 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

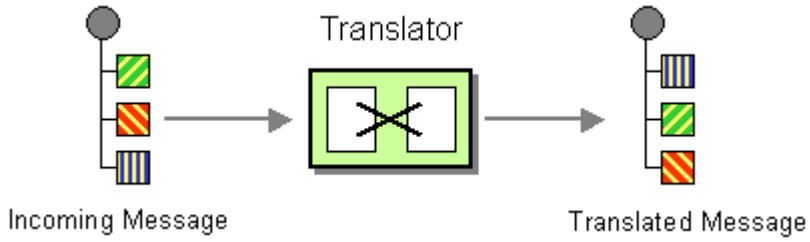
economically feasible. We may all remember the effort related to the Y2K retrofits where the scope of the change was limited to the size of a single field!

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, while the other application uses COBOL copybooks.

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*). 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*. This way, all applications would publish and consume messages in a common format as opposed to the application 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 a 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 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 US 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 etc., EAI vendor formats) Character sets (ASCII, UniCode, EBCDIC) Encryption / compression	Parse data representation and render in a different format. Decrypt/ encrypt as necessary.	XML Parsers, EAI parser / renderer tools Custom APIs
Transport	Communications Protocols:TCP/IP sockets, http, SOAP, JMS, TIBCO RendezVous	Move data across protocols without affecting message content.	*Channel Adapter* 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) while other integration technologies leverage TCP/IP protocols (e.g. SOAP). Translation between different transport layers can be provided by the *Channel Adapter* 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 can only transport character or byte streams. This means that complex data structures have to be converted into a character string. Common formats include XML, fixed-length fields (e.g. EDI records) or proprietary formats. In many cases, data is also compressed or encrypted, carries check digits or digital certificates. In order to interface systems with different data representations, data may have to be decrypted, uncompressed and parsed, then the new data format rendered, and possibly compressed and encrypted as well.

The *Data Types* layer defines the application data types that the application (domain) model is based on. Here we deal with decisions 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, etc. We may also consider whether the field *Postal Code* denotes only a US ZIP code or can contain Canadian postal codes. In case of a US 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 4 regions: Western, Central, Southern and Eastern, identified by the letters 'W", 'C', 'S' and 'E'. Another department may differentiate the Pacific Region from the Mountain Region and distinguishes the Northeast from the Southeast. Each region is identifies by a unique two-digit number. What number does the letter 'E' correspond to?

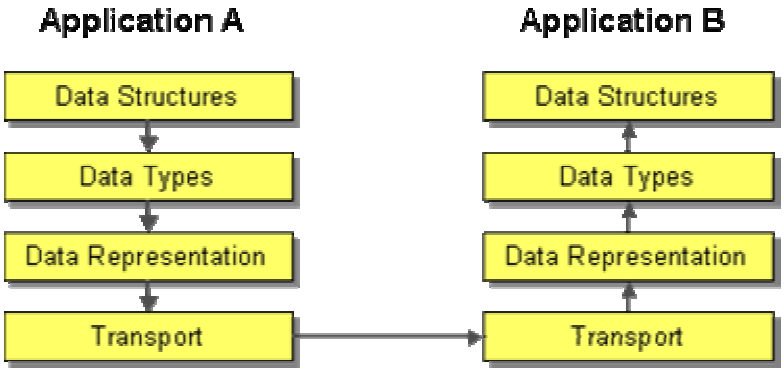
The *Data Structures* describes the data at the level of the application domain model. It is therefore also referred to as the Application Layer. This layer defines the logical entities that the application deals with, such as *Customer*, *Address* or *Account*. It also defines the relationships between these entities: Can one customer have multiple accounts? Can a customer have multiple addresses? Can customers share and 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

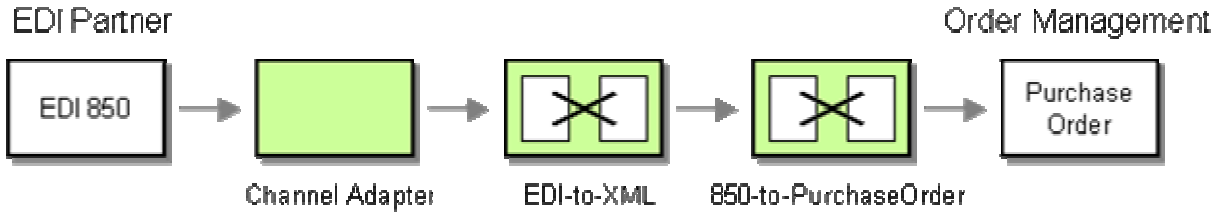
Many of the design trade-offs in integration are driven by the need to decouple components or applications. Decoupling is an essential tool to enable the management of change. Integration typically connects existing applications and has to accommodate changes to these applications. *Message Channel*s decouple applications from having to know each other's location. A *Message Router* can even decouple applications from having to agree on a common *Message Router*. However, this form of decoupling only achieves limited independence between applications if they depend on each other's data formats. A *Message Translator* can help remove this 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 a file to HTTP, the data format changes from fixed-format to XML, 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 we can treat one layer without regard to the lower layers and therefore can choose to work at different levels of abstraction. Correspondingly, we can talk about transformation at each layer of abstraction (see picture).



Chaining multiple *Message Translator* units using *Pipes and Filters* results in the following architecture (see picture). Creating one *Message Translator* for each layer allows us to reuse these components in other scenarios. For example, the *Channel Adapter* and the *EDI-to-XML Message Translator* can be generic enough to deal with any incoming EDI document.



This approach also makes individual layers interchangeable. 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. A *Content Enricher* augments the information inside a message while the *Content Filter* removes information. The *Claim Check* removes information but stores it for later retrieval. The *Normalizer* can convert a number of different message formats into a consistent format. Lastly, the *Canonical Data Model* 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). The *Messaging Bridge* performs a translation of the transport layer by connecting multiple messaging systems to each other.

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 show a simple example (for all the gory detail 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 Representation 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 far fetched, keep in mind that one of the most popular pieces of enterprise software 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>
```

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 numbers, 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 `<xsl:template>` 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">
```

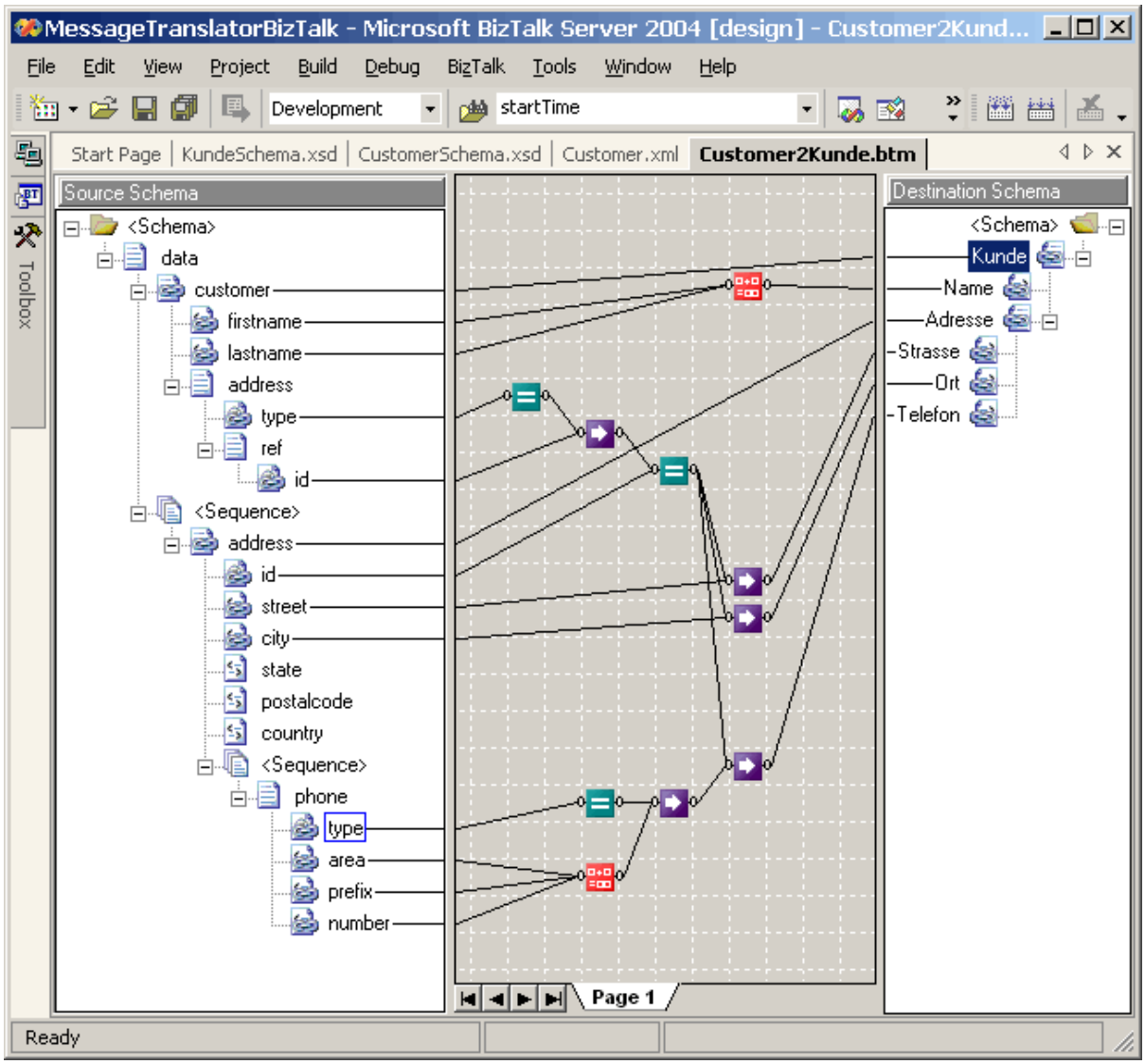
causes the subsequent lines to be executed for each `<customer>` element in the source document. The next statements concatenate first and last name and output it inside the `<Name>` element. Getting the address is a little trickier. The XSL code looks up the correct instance of the `<address>` element and calls the "subroutine" `getaddr`. `getaddr` extracts the address and phone number from the original `<address>` element. It uses the cell cell phone number if one is present and 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 visual transformation editors that displays the structure of the two document formats on the left-hand side and right-hand side of the screen, respectively. The users can then drag and drop between the two sides to associate elements between the formats. This can be a lot

simpler than coding XSL. Some vendors specialize entirely in transformation tools, for example Contivo, Inc..

The following screen shots 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 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 tool.

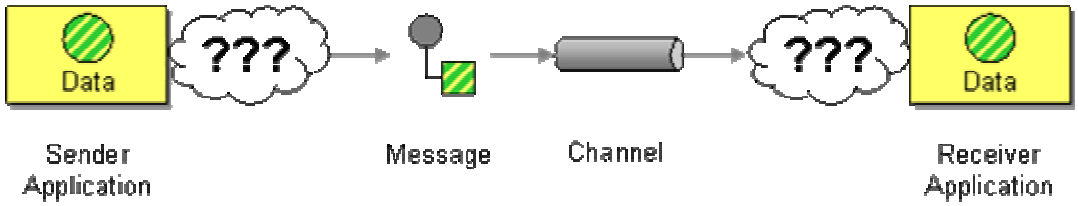
Related patterns: *Canonical Data Model*, *Channel Adapter*, *Content Filter*, *Content Enricher*, *Message Channel*, *Message Endpoint*, *Message Router*, *Message Translator*, *Messaging Bridge*, *Normalizer*, *Pipes and Filters*, *Shared Database*, *Claim Check*

Message Endpoint

Applications are communicating by sending *Message*s to each other via *Message Channel*s.

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.



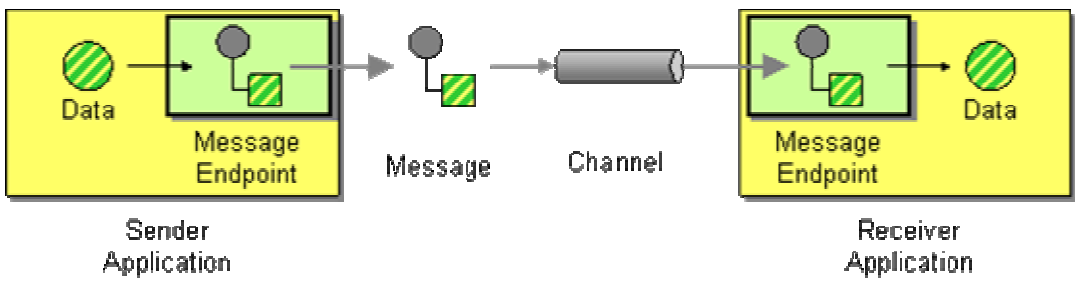
Applications disconnected from a message channel

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; it is domain-specific, where messaging is the domain.

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.

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 as something else, 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 more than one endpoint to interface to a single channel, usually to support multiple concurrent threads.

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

A *Message Endpoint* should be designed as a *Messaging Gateway* to encapsulate the messaging code and hide the message system from the rest of the application. It can employ a *Messaging Mapper* to transfer data between domain objects and messages. It can be structured as a *Service Activator* 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*.

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

Example: .NET MessageQueue

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

Related patterns: *Channel Adapter*, *Competing Consumers*, *Durable Subscriber*, *Event-Driven Consumer*, *Idempotent Receiver*, *Message*, *Message Channel*, *Message Dispatcher*, *Selective Consumer*, *Service Activator*, *Messaging Gateway*, *Messaging Mapper*, *Polling Consumer*, *Transactional Client*

4. Messaging Channels

Introduction

In *Introduction to Messaging Systems*, we discussed *Message Channel*. When two applications wish to exchange data, they do so by sending the data through a channel that connects the two. The application sending the data may not know which application will receive the data, but by selecting a particular channel to send the data on, the sender knows that the receiver will be one that is looking for that sort of data by looking for it on that channel. In this way, the applications that produce shared data have a way to communicate with those that wish to consume it.

Message Channel Themes

Deciding to use a *Message Channel* is the simple part; if an application has data to transmit or data it wishes to receive, it will have to use a channel. The challenge is knowing what channels your applications will need and what to use them for.

Fixed set of channels — One theme in this chapter is that the set of *Message Channel*s available to an application tends to be static. When designing an application, a developer has to know where to put what types of data to share that data with other applications, and likewise where to look for what types of data coming from other applications. These paths of communication cannot be dynamically created and discovered at runtime; they need to be agreed upon at design time so that the application knows where its data is coming from and where the data is going to. (While it is true that most channels must be staticly defined, there are exceptions to this theme, cases where dynamic channels are practical and useful. One exception is the reply channel in *Request-Reply*. The requestor can create or obtain a new channel the replier knows nothing about, specify it as the *Return Address* of a request message, and then the replier can make use of it. Another exception is messaging system implementations that support hierarchical channels. A receiver can subscribe to a parent in the hierarchy, then a sender can publish to a new child channel the receiver knows nothing about, and the subscriber will still receive the message. These relatively unusual cases notwithstanding, channels are usually defined at deployment-time and applications are designed around a known set of channels.)

Determining the set of channels — A related issue is: Who decides what *Message Channel*s are available, the messaging system or the applications? That is to say: Does the messaging system define certain channels and require the applications to make due with those? Or do the applications determine what channels they need and require the messaging system to provide them? There is no simple answer; designing the needed set of channels is iterative. First the applications determine the channels the messaging system will need to provide. Subsequent applications will try to design their communication around the channels that are available, but when this is not practical, they will require that additional channels be added. When a set of

applications already use a certain set of channels, and new applications wish to join in, they too will use the existing set of channels. When existing applications add new functionality, they may require new channels.

Unidirectional channels — Another common source of confusion is whether a *Message Channel* is unidirectional or bidirectional. Technically, it's neither; a channel is more like a bucket that some applications add data to and other applications take data from (albeit a bucket that is distributed across multiple computers in some coordinated fashion). But because the data is in messages that travel from one application to another, that gives the channel direction, making it unidirectional. If a channel were bidirectional, that would mean that an application would both send messages to and receive messages from the same channel, which—while technically possible—makes little sense because the application would tend to keep consuming its own messages, the messages it's supposed to be sending to other applications. So for all practical purposes, channels are unidirectional. As a consequence, for two applications to have a two-way conversation, they will need two channels, one in each direction (see *Request-Reply* in the next chapter).

Message Channel Decisions

Now that we understand what *Message Channel*s are, let's consider the decisions involved in using them:

One-to-one or one-to-many — When your application shares a piece of data, do you want to share it with just one other application or with any other application that is interested? To send the data to a single application, use a *Point-to-Point Channel*. This does not guarantee that every piece of data sent on that channel will necessarily go to the same receiver, because the channel might have multiple receivers; but it does ensure that any one piece of data will only be received by one of the applications. If you want all of the receiver applications to be able to receive the data, use a *Publish-Subscribe Channel*. When you send a piece of data this way, the channel effectively copies the data for each of the receivers.

What type of data — Any data in any computer memory has to conform to some sort of *type*: a known format or expected structure with an agreed upon meaning. Otherwise, all data would just be a bunch of bytes and there would be no way to make any sense of it. Messaging systems work much the same way; the message contents must conform to some type so that the receiver understands the data's structure. *Datatype Channel* is the principle that all of the data on a channel has to be of the same type. This is the main reason why messaging systems need lots of channels; if the data could be of any type, the messaging system would only need one channel (in each direction) between any two applications.

Invalid and dead messages — The message system can ensure that a message is delivered properly, but it cannot guarantee that the receiver will know what to do with it. The receiver has expectations about the data's type and meaning; when it receives a message that doesn't meet these expectations, there's not much it can do. What it can do, though, is put the strange message on a specially designated *Invalid Message Channel*, in hopes that some utility monitoring the

channel will pick up the message and figure out what to do with it. Many messaging systems have a similar built-in feature, a *Dead Letter Channel* for messages which are successfully sent but ultimately cannot be successfully delivered. Again, hopefully some utility monitoring the channel will know what to do with the messages that could not be delivered.

Crash proof — If the messaging system crashes or is shut down for maintence, what happens to its messages? When it is back up and running, will its messages still be in its channels? By default, no; channels store their messages in memory. However, *Guaranteed Delivery* makes channels persistent so that their messages are stored on disk. This hurts performance but makes messaging more reliable, even when the messaging system isn't.

Non-messaging clients — What if an application cannot connect to a messaging system but still wants to participate in messaging? Normally it would be out of luck; but if the messaging system can connect to the application somehow—through its user interface, its business services API, its database, or through a network connection such as TCP/IP or HTTP—then a *Channel Adapter* on the messaging system can be used to connect a channel (or set of channels) to the application without having to modify the application and perhaps without having to have a messaging client running on the same machine as the application. Sometimes the "non-messaging client" really is a messaging client, just for a different messaging system. In that case, an application that is a client on both messaging systems can build a *Messaging Bridge* between the two, effectively connecting them into one composite messaging system.

Communications backbone — As more and more of an enterprise's applications connect to the messaging system and make their functionality available through messaging, the messaging system becomes a centeralized point of one-stop-shopping for functionality in the enterprise. A new application simply needs to know which channels to use to request functionality and which others to listen on for the results. The messaging system itself essentially becomes a *Message Bus*, a backbone providing access to all of the enterprise's various and ever-changing applications and functionality. You can achieve this integration nirvana more quickly and easily by specifically designing for it from the beginning.

So as you can see, getting applications set up for *Messaging* involves more than just connecting them to the messaging system so that they can send messages. The messages must have *Message Channel*s to transmit on. Slapping in some channels doesn't get the job done either. They have to be designed with a purpose, based on the data type being shared, the sort of application making the data available, and the sort of application receiving the data. This chapter will explain the decisions that go into designing these channels.

To help illustrate the patterns, each one has an example from a ficticious, simplified "stock trading" domain. While none of these examples should be used as the basis for implementing a real trading system, they do serve as quick and specific examples of how the patterns can be used.

Point-to-Point Channel

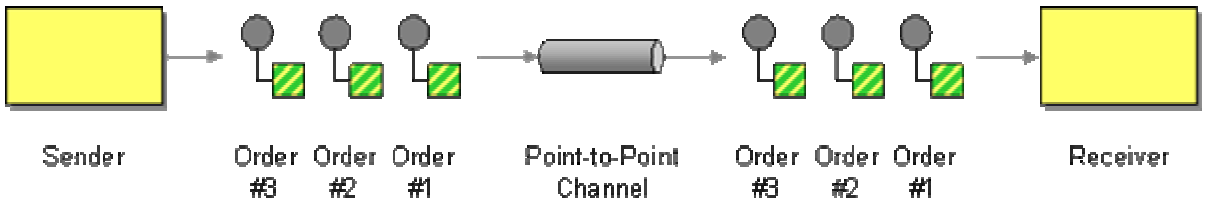
An application is using *Messaging* to make remote procedure calls (RPC's) or transfer documents.

How can the caller be sure that exactly one receiver will receive the document or perform the call?

One advantage of an RPC is that it's invoked on a single remote process, so either that receiver performs the procedure or it does not (and an exception occurs). And since the receiver was only called once, it only performs the procedure once. But with messaging, once a call is packaged as a *Message* and placed on a *Message Channel*, potentially many receivers could see it on the channel and decide to perform the procedure.

The messaging system could prevent more than one receiver from monitoring a single channel, but this would unnecessarily limit callers that wish to transmit data to multiple receivers. All of the receivers on a channel could coordinate to ensure that only one of them actually performs the procedure, but that would be complex, create a lot of communications overhead, and generally increase the coupling between otherwise independent receivers. Multiple receivers on a single channel may be desirable so that multiple messages can be consumed concurrently, but any one receiver should consume any single message.

Send the message on a *Point-to-Point Channel*, which ensures that only one receiver will receive a particular message.



A *Point-to-Point Channel* ensures that only one receiver consumes any given message. If the channel has multiple receivers, only one of them can successfully consume a particular message. If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other. The channel can still have multiple receivers to consume multiple messages concurrently, but only a single receiver consumes any one message.

When a *Point-to-Point Channel* only has one consumer, the fact that a message only gets consumed once is not surprising. When the channel has multiple consumers, then they become *Competing Consumers*, and the channel ensures that only one of the consumers receives each message. The effort to consume messages is highly scalable because that work can be load-balanced across multiple consumers running in multiple applications on multiple computers.

Whereas a *Point-to-Point Channel* sends a message to only one of the available receivers, to send a message to all available receivers, use a *Publish-Subscribe Channel*. To implement RPC's using messaging, use *Request-Reply* with a pair of *Point-to-Point Channel*s. The call is a *Command Message* whereas the reply is a *Document Message*.

Example: Stock Trading

In a stock trading system, the request to make a particular trade is a message that should be consumed and performed by exactly one receiver, so the message should be placed on a *Point-to-Point Channel*.

Example: JMS Queue

In JMS, a point-to-point channel implements the `Queue` interface. The sender uses a `QueueSender` to send messages; each receiver uses its own `QueueReceiver` to receive messages. [JMS11, pp.75-78], [Hapner, p.18]

An application uses a `QueueSender` to send a message like this:

```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory factory = // obtain the connection factory via JNDI
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender(queue);

Message message = session.createTextMessage("The contents of the message.");

sender.send(message);
```

An application uses a `QueueReceiver` to receive a message like this:

```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory factory = // obtain the connection factory via JNDI
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(queue);

TextMessage message = (TextMessage) receiver.receive();
String contents = message.getText();
```

Note: JMS 1.1 unifies the client API's for the point-to-point and publish/subscribe domains, so the code shown here can be simplified to use `Destination`, `ConnectionFactory`, `Connection`, `Session`, `MessageProducer`, and `MessageConsumer`, rather than their `Queue`-specific counterparts.

Example: .NET MessageQueue

In .NET, the `MessageQueue` class implements a point-to-point channel. [SysMsg] MSMQ, which implements .NET messaging, only supported point-to-point messaging prior to version 3.0, so point-to-point is what .NET supports. Whereas JMS seperates the responsibilities of the connection factory, connection, session, sender, and queue, a `MessageQueue` does it all.

Send a message on a `MessageQueue` like this:

```
MessageQueue queue = new MessageQueue("MyQueue");
queue.Send("The contents of the message.");
```

Receive a message on a `MessageQueue` like this:

```
MessageQueue queue = new MessageQueue("MyQueue");
Message message = queue.Receive();
String contents = (String) message.Body();
```

Related patterns: *Command Message*, *Competing Consumers*, *Document Message*, *Message*, *Message Channel*, *Messaging*, *Publish-Subscribe Channel*, *Request-Reply*

Publish-Subscribe Channel

An application is using *Messaging* to announce events.

How can the sender broadcast an event to all interested receivers?

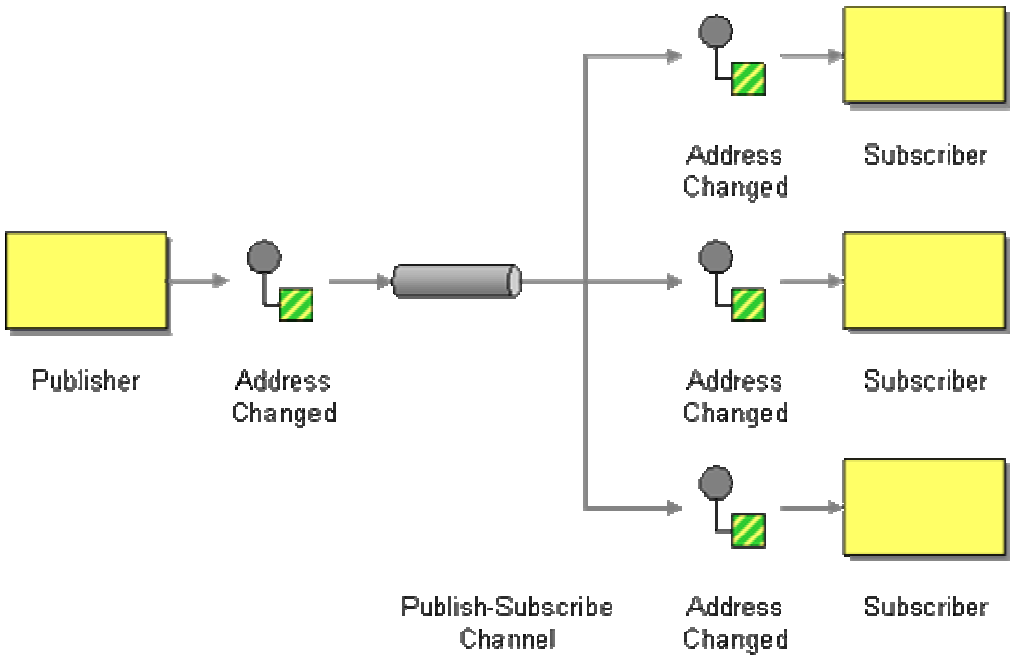
Luckily, there are well-established patterns for implementing broadcasting. The Observer pattern [GoF] describes the need to decouple observers from their subject so that the subject can easily provide event notification to all interested observers no matter how many observers there are (even none). The Publisher-Subscriber pattern [POSA] expands upon Observer by adding the notion of an event channel for communicating event notifications.

That's the theory, but how does it work with messaging? The event can be packaged as a *Message* so that messaging will reliably communicate the event to the observers (subscribers). Then the event channel is a *Message Channel*. But how will a messaging channel properly communicate the event to all of the subscribers?

Each subscriber needs to be notified of a particular event once, but should not be notified repeatedly of the same event. The event cannot be considered consumed until all of the subscribers have been notified. But once all of the subscribers have been notified, the event can be considered consumed and should disappear from the channel. Yet having the subscribers coordinate to determine when a message is consumed violates the decoupling of the Observer

pattern. Concurrent consumers should not be considered to compete, but should be able to share the event message.

Send the event on a *Publish-Subscribe Channel*, which delivers a copy of a particular event to each receiver.



A *Publish-Subscribe Channel* works like this: It has one input channel that splits into multiple output channels, one for each subscriber. When an event is published into the channel, the *Publish-Subscribe Channel* delivers a copy of the message to each of the output channels. Each output channel has only one subscriber, which is only allowed to consume a message once. In this way, each subscriber only gets the message once and consumed copies disappear from their channels.

A *Publish-Subscribe Channel* can be a useful debugging tool. Even though a message is destined to only a single receiver, using a *Publish-Subscribe Channel* allows you to eavesdrop on a message channel without disturbing the existing message flow. Monitoring all traffic on a channel can be tremendously helpful when debugging messaging applications. It can also save you from inserting a ton of print statements into each application that participates in the messaging solution. Creating a program that listens for messages on all active channels and logs them to a file can realize many of the same benefits that a *Message Store* brings.

The ability to eavesdrop on a *Publish-Subscribe Channel* can also turn into a disadvantage. If your messaging solution transmits payroll data between the payroll system and the accounting system, you may not want to allow anyone to write a simple program to listen to the message traffic. *Point-to-Point Channel*s alleviate the problem to some extent because the eavesdropper would consume the messages and the situation would be detected very quickly. However, some implementations of message queues implementing *peek* functions that let consumers look at messages inside a queue without consuming any of the messages. As a result, subscribing to a

Message Channel is an operation that should be restricted by security policies. Many (but not all) commercial messaging implementations implement such restrictions. In addition, creating a monitoring tool that logs active subscribers to *Message Channel*s can be a useful systems management tool.

For more details on how to implement Observer using messaging, see *JMS Publish/Subscribe Example*.

> **Wildcard Subscribers**
>
> Many messaging systems allow subscribers to *Publish-Subscribe Channel*s to specify special wild card characters. This is a powerful technique to allow subscribers to subscribe to multiple channels at once. For example, if an application publishes messages to the channels `MyCorp/Prod/OrderProcessing/NewOrders` and `MyCorp/Prod/OrderProcessing/CancelledOrders` an application could subscribe to `MyCorp/Prod/OrderProcessing/*` and receive all messages related to order processing. Another application could subscribe to `MyCorp/Dev/**` to receive all messages sent by all applications in the development environment. Only subscribers are allowed to use wildcards, publishers are always required to publish a message to a specific channel. The specific capabilities and syntax for wildcard subscribers vary between the different messaging vendors.

An *Event Message* is usually sent on a *Publish-Subscribe Channel* because multiple dependents are often interested in an event. A subscriber can be durable or non-durable—see *Durable Subscriber*. If notifications should be acknowledged by the subscribers, use *Request-Reply*, where the notification is the request and the acknowledgement is the reply.

Example: Stock Trading

In a stock trading system, many systems may need to be notified of the completion of a trade, so make them all subscribers of a *Publish-Subscribe Channel* that publishes trade completions.

Example: JMS Topic

In JMS, a *Publish-Subscribe Channel* implements the `Topic` interface. The sender uses a `TopicPublisher` to send messages; each receiver uses its own `TopicSubscriber` to receive messages. [JMS11, pp.79-85], [Hapner, p.18]

An application uses a `TopicPublisher` to send a message like this:

```
Topic topic = // obtain the topic via JNDI
TopicConnectionFactory factory = // obtain the connection factory via JNDI
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);
TopicPublisher publisher = session.createPublisher(topic);

Message message = session.createTextMessage("The contents of the message.");
```

```
publisher.publish(message);
```

An application uses a `TopicSubscriber` to receive a message like this:

```
Topic topic = // obtain the topic via JNDI
TopicConnectionFactory factory = // obtain the connection factory via JNDI
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber subscriber = session.createSubscriber(topic);

TextMessage message = (TextMessage) subscriber.receive();
String contents = message.getText();
```

Note: JMS 1.1 unifies the client API's for the point-to-point and publish/subscribe domains, so the code shown here can be simplified to use `Destination`, `ConnectionFactory`, `Connection`, `Session`, `MessageProducer`, and `MessageConsumer`, rather than their `Topic`-specific counterparts.

Example: MSMQ One-to-Many Messaging

A new feature in MSMQ 3.0 [MSMQ01] is a *one-to-many messaging model*, which has two different approaches:

1. **Real-Time Messaging Multicast** – This most closely matches publish-subscribe, but its implementation is entirely dependent on IP multicasting via the Pragmatic General Multicast (PGM) protocol.
2. **Distribution Lists and Multiple-Element Format Names** – A Distribution List enables the sender to explicitly send a message to a list of receivers (but this violates the spirit of the Observer pattern). A Multiple-Element Format Name is a symbolic channel specifier that dynamically maps to multiple real channels, which is more the spirit of the publish-subscribe pattern but still forces the sender to choose between real and not-so-real channels.

The .NET CLR does not provide direct support for using the one-to-many messaging model. However, this functionality can be accessed through the COM interface [MDMSG], which can be embedded in .NET code.

Related patterns: *Durable Subscriber*, *Event Message*, *Message*, *Message Channel*, *Message Store*, *Messaging*, *JMS Publish/Subscribe Example*, *Point-to-Point Channel*, *Request-Reply*

Datatype Channel

An application is using *Messaging* to transfer different types of data, such as different types of documents.

How can the application send a data item such that the receiver will know how to process it?

All messages are just instances of the same message type, as defined by the messaging system, and the contents of any message are ultimately just a byte array. While this simple structure--a bundle of bytes--is specific enough for a messaging system to be able to transmit a message, it is not specific enough for a receiver to be able to process a message's contents.

A receiver must know the message content's data structure and data format. The structure could be character array, byte array, serialized object, XML document, etc. The format could be the record structure of the bytes or characters, the class of the serialized object, the DTD of the XML document, etc. All of this knowledge is loosely referred to as the message's type, meaning the structure and format of the message's contents.

The receiver must know what type of messages it's receiving, or it doesn't know how to process them. For example, a sender may wish to send different objects such as purchase orders, price quotes, and queries. Yet a receiver will probably take different steps to process each of those, so it has to know which is which. If the sender simply sends all of these to the receiver via a message channel, the receiver will not know how to process each one.

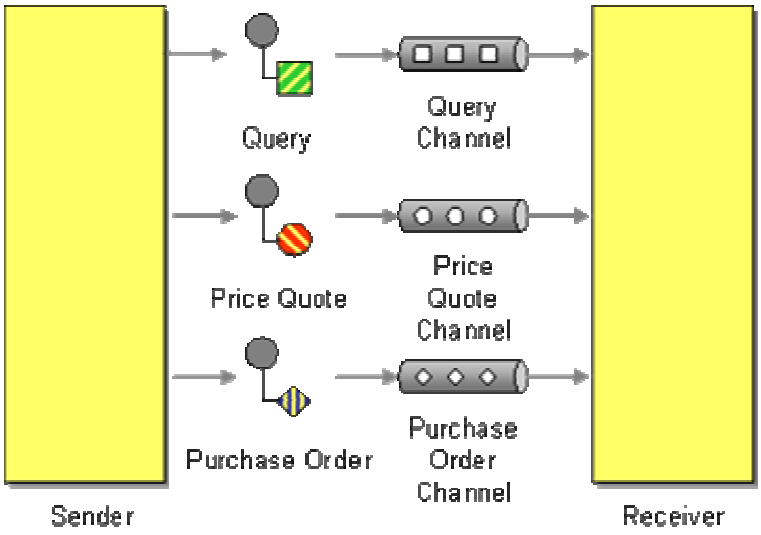
Mixed Data Types

The sender knows what message type it's sending, so how can this be communicated to the receiver? The sender could put a flag in the message's header (see *Format Indicator*), but then the receiver will need a case statement. The sender could wrap the data in a *Command Message* with a different command for each type of data, but that presumes to tell the receiver what to do with the data when all that the message is trying to do is transmit the data to the receiver.

A similar problem that is completely separate from messaging occurs using non-array collections: collections can be heterogeneous (each item can be of any object type), but as a practical matter collections need to be homogeneous (each item should be of the same object type, meaning that they all implement the same abstract class or interface). Homogeneous collections are much more useful because an iterator on the collection knows what type each item will be and can manipulate each item using the methods that type understands.

The same principle applies to messaging because in this context, a channel is like a collection and a receiver is like an iterator. Although a particular channel doesn't require that all of its messages be of the same type, they ought to be so that a receiver on that channel knows what type of message it's receiving.

Use a separate *Datatype Channel* for each data type, so that all data on a particular channel is of the same type.



By using a separate *Datatype Channel* for each type of data, all of the messages on a given channel will contain the same type of data. The sender, knowing what type the data is, will need to select the appropriate channel to send it on. The receiver, knowing what channel the data was received on, will know what its type is.

As discussed in *Message Channel*, channels are cheap but not free. An application may need to transmit many different data types, too many to create a separate *Datatype Channel* for each. In this case, multiple data types can share a single channel by using a different *Selective Consumer* for each type. This makes a single channel act like multiple data type channels. Whereas *Datatype Channel* explains why all messages on a channel must be of the same format, *Canonical Data Model* explains how all messages on all channels in an enterprise should follow a unified data model.

A *Message Dispatcher*, besides providing concurrent message consumption, can be used to process a generic set of messages in type-specific ways. Each message must specify its type; the dispatcher detects the message's type and dispatches it to a type-specific performer for processing. The messages on the channel are still all of the same type, but that type is the more general one that the dispatcher supports, not the more specific ones that the various performers require.

Example: Stock Trading

In a stock trading system, if the format of a quote request is different from that of a trade request, the system should use a separate *Datatype Channel* for communicating each kind of request. Likewise, a change-of-address announcement may have a different format from a change-of-portfolio-manager announcement, so each kind of announcement should have its own *Datatype Channel*.

Example: Purchasing System

Reconsidering our earlier example, since the sender wants to send three different types of data (purchase orders, price quotes, and queries), it should use three different channels. When sending an item, the sender must select the appropriate *Datatype Channel* for that item. When receiving an item, the receiver knows the item's type because of which datatype channel it received the item on.

Related patterns: *Canonical Data Model*, *Command Message*, *Format Indicator*, *Message Channel*, *Message Dispatcher*, *Selective Consumer*, *Messaging*

Invalid Message Channel

An application is using *Messaging* to receive *Message*s.

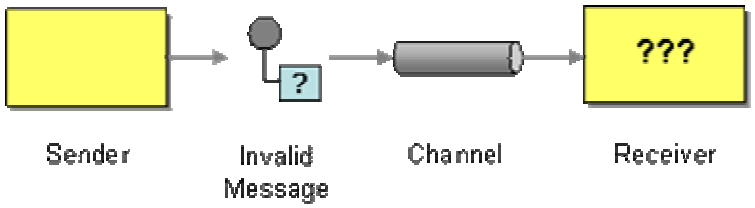
How can a messaging receiver gracefully handle receiving a message that makes no sense?

In theory, everything on a *Message Channel* is just a message and message receivers just process messages. However, to process a message, a receiver must be able to interpret its data and understand its meaning. This is not always possible: the message body may cause parsing errors, lexical errors, or validation errors. The message header may be missing needed properties, or the property values may not make sense. A sender might put a perfectly good message on the wrong channel, transmitting it to the wrong receiver. A malicious sender could purposely send an incorrect message just to mess-up the receiver. A receiver may not be able to process all messages it receives, so it needs to have some other way to handle messages it does not consider valid.

A *Message Channel* is a *Datatype Channel*, where each of the messages on the channel is supposed to be of the proper datatype for that channel. If a sender puts a message on the channel that is not of the proper datatype, the messaging system will transmit the message successfully, but the receiver will not recognize the message and will not know how to process it.

An example of a message with an improper datatype or format is a byte message on a channel that is supposed to contain text messages. Another example is a message whose format is not correct, such as an XML document that is not well-formed, or that is not valid for the agreed-upon DTD or schema. There's nothing wrong with these messages, as far as the messaging system is concerned, but the receiver will not be able to process them, so they are invalid.

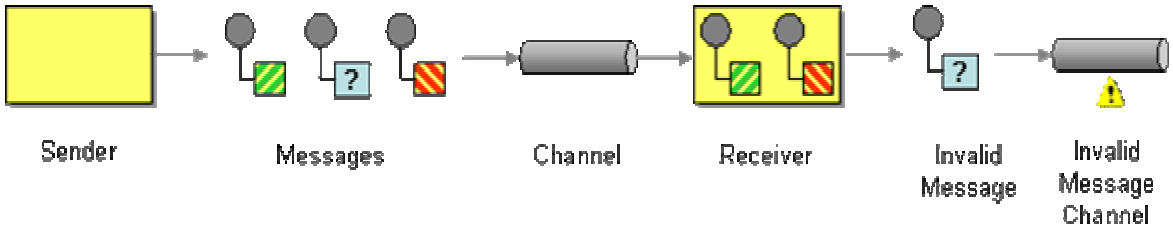
Messages that do not contain the header field values that the receiver expects are also invalid. If a message is supposed to have header properties such as a *Correlation Identifier*, *Message Sequence* identifiers, a *Return Address*, etc., but the message is missing the properties, then the messaging system will deliver the message properly but the receiver will not be able to process it successfully.



Invalid Message

When the receiver discovers that the message it's trying to process is not valid, what should it do with the message? It could put the message back on the channel, but then the message will just be re-consumed by the same receiver or another like it. Meanwhile, invalid messages that are being ignored will clutter the channel and hurt performance. The receiver could consume the invalid message and throw it away, but that would tend to hide messaging problems that need to be detected. What the system needs is a way to clean improper messages out of channels and put them somewhere out of the way, yet in a place where these invalid messages can be detected to diagnose problems with the messaging system.

The receiver should move the improper message to an *Invalid Message Channel*, a special channel for messages that could not be processed by their receivers.



When designing a messaging system for applications to use, the administrator will need to define one or more *Invalid Message Channel*s for the applications to use.

The *Invalid Message Channel* will not be used for normal, successful communication, so its being cluttered with improper messages will not cause a problem. An error handler that wants to diagnose improper messages can use a receiver on the invalid channel to detect messages as they become available.

A *Invalid Message Channel* is like an error log for messaging. When something goes wrong in an application, it's a good idea to log the error. When something goes wrong processing a message, it's a good idea to put the message on the channel for invalid messages. If it won't be obvious to anyone browsing the channel why this message is invalid, the application should also log an error with more details.

Keep in mind that a message is neither inherently valid or invalid--it is the receiver's context and expectations that make this determination. A message that may be valid for one receiver may be invalid for another receiver; two such receivers should not share the same channel. A message that is valid for one receiver on a channel should be valid for all other receivers on that channel;

likewise, if one receiver considers a message invalid, all other receivers should as well. It is the sender's responsibility to make sure that a message it sends on a channel will be considered valid by the channel's receivers; otherwise, the receivers will ignore the sender's messages by rerouting them to the *Invalid Message Channel*.

A similar but separate problem is when a message is structured properly, but its contents are semantically incorrect. For example, a *Command Message* may instruct the receiver to delete a database record that does not exist. This is not a messaging error, but an application error. As such, while it may be tempting to move the message to the *Invalid Message Channel*, there is nothing wrong with the message, so treating it as invalid is misleading. Rather, an error like this should be handled as an invalid application request, not an invalid message.

This differentiation between message-processing errors and application errors becomes simpler and clearer when the receiver is implemented as a *Service Activator* or *Messaging Gateway*. These patterns separate message-processing code from the rest of the application. If an error occurs while processing the message, the message is invalid and should be moved to the *Invalid Message Channel*. If it occurs while the application processes the data from the message, that is an application error that has nothing to do with messaging.

An *Invalid Message Channel* whose contents are ignored is about as useful as an error log that is ignored. Messages on the *Invalid Message Channel* indicate application integration problems, so those messages should not be ignored; rather, they should be analyzed to determine what went wrong so that the problem can be fixed. Ideally, this would be an automated process that consumed invalid messages, determined their cuause, and fixed the underlying problems. However, the cause is often a coding or configuration error which requires a developer or system analyst to evaluate and repair. At the very least, applications which use messaging and *Invalid Message Channel*s should have a process that monitors the *Invalid Message Channel* and alerts system administrators whenever the channel contains messages.

A similar concept implemented by many messaging systems is a *Dead Letter Channel*. Whereas an *Invalid Message Channel* is for messages that can be delivered and received but not processed, a *Dead Letter Channel* is for messages that the messaging system cannot deliver properly.

Example: Stock Trading

In a stock trading system, an application for executing trade requests might receive a request for a current price quote, or a trade request that does not specify what security to buy or how many shares, or a trade request that does not specify who to send the trade confirmation to. In any of these cases, the application has received an invalid message--one that does not meet the minimum requirements necessary for the application to be able to process the trade request. Once the application determines the message to be invalid, it should resend the message onto the *Invalid Message Channel*. The various applications that send trade requests may wish to monitor the *Invalid Message Channel* to determine if their requests are being discarded.

Example: JMS Specification

In JMS, the specification suggests that if a `MessageListener` gets a message it cannot process, a well-behaved listener should divert the message "to some form of application-specific 'unprocessable message' destination." [JMS11, p.69] This unprocessable message destination is an *Invalid Message Channel*.

Example: Simple Messaging

JMS Request/Reply Example and *.NET Request/Reply Example* show an example of how to implement receivers that reroute messages they cannot process to an *Invalid Message Channel*.

Related patterns: *Command Message*, *Correlation Identifier*, *Datatype Channel*, *Dead Letter Channel*, *Message*, *Message Channel*, *Message Sequence*, *Messaging*, *Service Activator*, *Messaging Gateway*, *JMS Request/Reply Example*, *.NET Request/Reply Example*, *Return Address*

Dead Letter Channel

An enterprise is using *Messaging* to integrate applications.

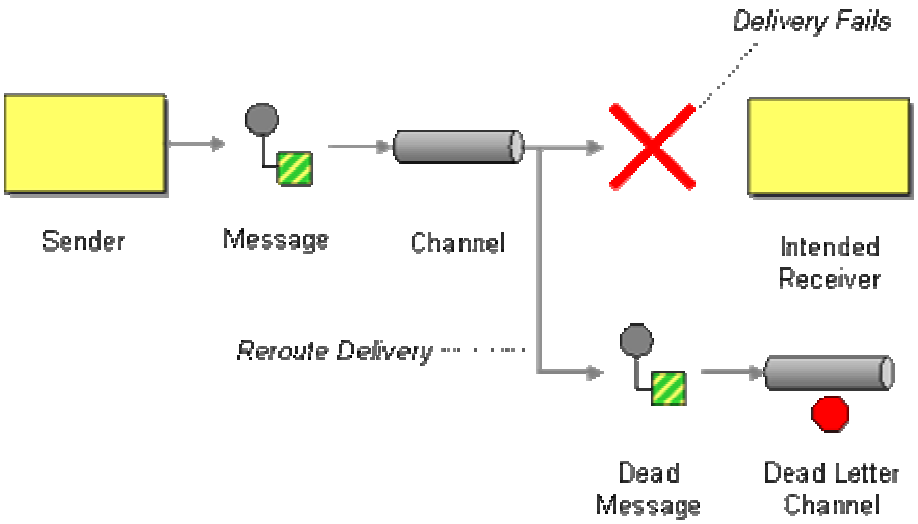
What will the messaging system do with a message it cannot deliver?

If a receiver receives a message it cannot process, it should move the invalid message to an *Invalid Message Channel*. But what if the messaging system cannot deliver the message to the receiver in the first place?

There are a number of reasons the messaging system may not be able to deliver a message. The messaging system may not have the message's channel configured properly. The message's channel may be deleted after the message is sent but before it can be delivered or while it is waiting to be received. The message may expire before it can be delivered (see *Message Expiration*). A message without an explicit expiration may nevertheless timeout if it cannot be delivered for a very long time. A message with a *Selective Consumer* that everyone ignores will never be read and may eventually die. A message could have something wrong with its header that prevents it from being delivered successfully.

Once the messaging system determines that it cannot deliver a message, it has to do something with the message. It could just leave the message wherever it is, cluttering up the system. It could try to deliver the message back to the sender, but the sender is not a receiver and cannot detect deliveries. It could just delete the message and hope no one misses it, but this may well cause a problem for the sender that has successfully sent the message and expects it to be delivered (and received and processed).

When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a *Dead Letter Channel*.



The specific way a *Dead Letter Channel* works depends on the specific messaging system's implementation, if it provides one at all. The channel may be called a "dead message queue" [Monson-Haefel, p.125] or "dead letter queue." [MQSeries], [Dickman, pp.28-29] Typically, each machine the messaging system is installed on has its own local *Dead Letter Channel* so that whatever machine a message dies on, it can be moved from one local queue to another without any networking uncertainties. This also records what machine the message died on. When the messaging system moves the message, it may also record the original channel the message was supposed to be delivered on.

The difference between a dead message and an invalid one is that the messaging system cannot successfully deliver what it then deems a dead message, whereas an invalid message is properly delivered but cannot be processed by the receiver. Determining if a message should be moved to the *Dead Letter Channel* is an evaluation of the message's header performed by the messaging system; whereas the receiver moves a message to an *Invalid Message Channel* because of the message's body or particular header fields the receiver is interested in. To the receiver, determination and handling of dead messages seems automatic, whereas the receiver must handle invalid messages itself. A developer using a messaging system is stuck with whatever dead message handling the messaging system provides, but can design his own invalid message handling, including handling for seemingly dead messages that the messaging system doesn't handle.

Example: Stock Trading

In a stock trading system, an application that wishes to perform a trade can send a trade request. To make sure that the trade is received in a reasonable amount of time (less than five minutes, perhaps), the requestor sets the request's *Message Expiration* to five minutes. If the messaging system cannot deliver the request in that amount of time, or if the trading application does not

receive the message (e.g., read it off of the channel) in time, then the messaging system will take the message off of the trade request channel and put the message on the *Dead Letter Channel*. The trading system may wish to monitor the system's *Dead Letter Channel*s to determine if it is missing trades.

Related patterns: *Invalid Message Channel*, *Message Expiration*, *Selective Consumer*, *Messaging*

Guaranteed Delivery

An enterprise is using *Messaging* to integrate applications.

How can the sender make sure that a message will be delivered, even if the messaging system fails?

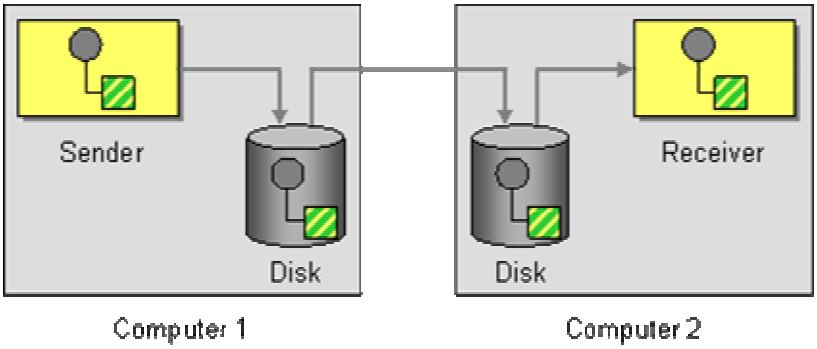
One of the main advantages of asynchronous messaging over RPC is that the sender, the receiver, and network connecting the two don't all have to be working at the same time. If the network is not available, the messaging system has to store the message until the network becomes available. If the receiver is unavailable, the messaging system has to store the message and retry delivery until the receiver becomes available. This is the *store and forward* process that messaging is based on. So where should the message be stored before it is forwarded?

By default, the messaging system stores the message in memory until it can successfully forward the message to the next storage point. This works as long as the messaging system is running reliably, but if the messaging system crashes (for example, because one of its computers loses power or the messaging process aborts unexpectedly), all of the messages stored in memory are lost.

Most applications have to deal with similar problems. All data that is stored in memory is lost if the application crashes. To prevent this, applications use files and databases to persist data to disk so that it survives system crashes.

Messaging systems need a similar way to persist messages more permanently so that no message gets lost even if the system crashes.

Use *Guaranteed Delivery* to make messages persistent so that they are not lost even if the messaging system crashes.



With *Guaranteed Delivery*, the messaging system uses a built-in data store to persist messages. Each computer the messaging system is installed on has its own data store so that the messages can be stored locally. When the sender sends a message, the send operation does not complete successfully until the message is safely stored in the sender's data store. Subsequently, the message is not deleted from one data store until it is successfully forwarded to and stored in the next data store. In this way, once the sender successfully sends the message, it is always stored on disk on at least one computer until is successfully delivered to and acknowledged by the receiver.

Persistence increases reliability, but at the expense of performance. Thus if it's OK to loose messages when the messaging system crashes or is shut down, avoid using *Guaranteed Delivery* so that messages will move through the messaging system faster.

Also consider that *Guaranteed Delivery* can consume a large amount of disk space in high-traffic scenarios. If a producer generates hundreds or thousands of messages per second, then a network outage that lasts multiple hours could use up a huge amount of disk space. Because the network is unavailable, the messages have to be stored on the producing computer's local disk drive which may not be designed to hold this much data. For these reasons, some messaging systems allow you to configure a *retry timeout* parameter that specifies how long messages are buffered inside the messaging system. In some high-traffic applications (e.g., streaming stock quotes to terminals), this timeout may have to be set to a short time span, for example a few minutes. Luckily, in many of these applications, messages are used as *Event Message*s and can safely be discarded after a short amount of time elapses (see *Message Expiration*).

It can also be useful to turn off *Guaranteed Delivery* during testing and debugging. This makes it easy to purge all message channels by stopping and restarting the messaging server. Messages that are still queued up can make it very tedious to debug even simple messaging programs. For example, you may have a sender and a receiver connected by a *Point-to-Point Channel*. If a message is still stored on the channel, the receiver will process that message before any new message that the sender produces. This is a common debugging pitfall in asynchronous, guaranteed messaging. Many commercial messaging implementations also allow you to purge queues individually to allow a fresh restart during testing.

How guaranteed is guaranteed messaging?

It is important to keep in mind that reliability in computer systems tends to be measured in the "number of 9s", e.g. 99.9%. This

tells us that something is rarely 100% reliable, with the cost already increasing exponentially to move from 99.9% to 99.99%. The same caveats apply to *Guaranteed Delivery*. There is always going to be a scenario where a message can get lost. For example, if the disk that stores the persisted messages fails, messages may get lost. You can make your disk storage more reliable by using redundant disk storage to reduce the likelihood of failure. This will possibly add another '9' to the reliability rating, but likely not make it a true 100%. Also, if the networks is unavailable for a long time, the messages that have to be stored may fill up the computer's disk, resulting in lost messages. In summary, *Guaranteed Delivery* is designed to make the message delivery resilient against expected outages, such as machine failures or network failures, but it is usually not 100% bullet-proof.

With .NET's MSMQ implementation, for a channel to be persistent, it must declared transactional, which means senders usually have to be *Transactional Client*s. In JMS, with *Publish-Subscribe Channel*, *Guaranteed Delivery* only assures that the messages will be delivered to the active subscribers. To assure that a subscriber receives messages even when it's inactive, the subscriber will need a *Durable Subscriber*.

Example: Stock Trading

In a stock trading system, trade requests and trade confirmations should probably be sent with *Guaranteed Delivery*, to help ensure that none are lost. Likewise, change-of-address announcements should probably be sent with *Guaranteed Delivery*. On the other hand, price updates probably do not require *Guaranteed Delivery*; loosing some of them is not significant, and their frequency makes the overhead of *Guaranteed Delivery* prohibitive.

In *Durable Subscriber*, the stock trading example says that some price-change subscribers may wish to be durable. If so, then perhaps the price-change channel should guarantee delivery as well. Yet other subscribers may not need to be durable nor want to suffer the overhead of *Guaranteed Delivery*. How can these different needs be met? The system may wish to implement two price-change channels, one with *Guaranteed Delivery* and another without. Only subscribers that require all updates should subscribe to the persistent channel, and their subscriptions should be durable. The publisher may wish to publish updates less frequently on the persistent channel because of its increased overhead.

Example: JMS Persistent Messages

In JMS, message persistence can be set on a per-message basis. In other words, some messages on a particular channel may be persistent while others might not be. [JMS11, pp.71-72], [Hapner, pp.58-59]

When a JMS sender wants to make a message persistent, it uses its `MessageProducer` to set the message's `JMSDeliveryMode` to `PERSISTENT`. The sender can set persistency on a per-message basis like this:

```
Session session = // obtain the session
```

```
Destination destination = // obtain the destination
Message message = // create the message
MessageProducer producer = session.createProducer(destination);
producer.send(
        message,
        javax.jms.DeliveryMode.PERSISTENT,
        javax.jms.Message.DEFAULT_PRIORITY,
        javax.jms.Message.DEFAULT_TIME_TO_LIVE);
```

If the application wants to make all of the messages persistent, it can set that as the default for the message producer:

```
producer.setDeliveryMode(javax.jms.DeliveryMode.PERSISTENT);
```

(And, in fact, the default delivery mode for a message producer is persistent). Now, messages sent by this producer are automatically persistent, so they can simply be sent:

```
producer.send(message);
```

Meanwhile, messages sent by other message producers on the same channel may be persistent, depending on how those producers configure their messages.

Example: IBM WebSphere MQ

In WebSphere MQ, *Guaranteed Delivery* can be set on a per-channel basis or a per-message basis. If the channel is not persistent, the messages cannot be persistent. If the channel is persistent, the channel can be configured such that all messages sent on that channel are automatically persistent, or such that an individual message can be sent persistently or non-persistently.

A channel is configured to be persistent (or not) when it is created in the messaging system. For example, the channel can be configured so that all of its messages will be persistent:

```
DEFINE Q(myQueue) PER(PERS)
```

Or, the channel can be configured so that the message sender can specify with each message whether the message is persistent or transient:

```
DEFINE Q(myQueue) PER(APP)
```

If the channel is set to allow the sender to specify persistency, then a JMS `MessageProducer` can set that delivery-mode property as described earlier. If the channel is set to make all messages persistent, then the delivery-mode settings specified by the `MessageProducer` are ignored. [WSMQ, pp.45-56]

Example: .NET Persistent Messages

With .NET, persistent messages are created by making a `MessageQueue` transactional:

```
MessageQueue.Create("MyQueue", true);
```

All messages sent on this queue will automatically be persistent. [Dickman, p.257]

Related patterns: *Durable Subscriber*, *Event Message*, *Message Expiration*, *Messaging*, *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Transactional Client*

Channel Adapter

Many enterprises use *Messaging* to integrate multiple, disparate applications.

How can you connect an application to the messaging system so that it can send and receive messages?

Most applications were not designed to work with a messaging infrastructure. There are a number of reasons for this. Many applications were developed as self-contained, stand-alone solutions but contain data or functionality that can be leveraged by other systems. For example, many mainframe applications were designed as a one-in-all application that does not need to interface with other applications.

Many message-oriented middleware systems expose proprietary API's so that an application developer or vendor would have to provide multiple implementations of a messaging interface for the application.

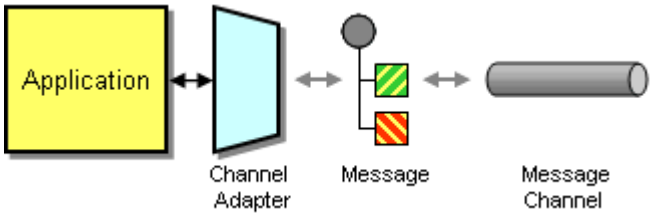
If applications need to exchange data with other applications, they often are designed to use more generic interface mechanisms such as file exchange or database tables. Reading and writing files is a basic operating system function and does not depend on vendor-specific API's. Likewise, most business applications already persists data into a database, so little extra effort is required to store data destined for other systems in a database table. Or an application can expose internal functions in a generic API that can be used by any other integration strategy, including messaging.

Other applications may be capable of communicating via a simple protocols like HTTP or TCP/IP. However, these protocols do not provide the same reliability as a *Message Channel* and the data format used by the application is usually specific to the application and not compatible with a common messaging solution.

In the case of custom applications, we could add code inside the application to send and receive messages. However, this can introduce additional complexity into the application and we need to be careful not to introduce any undesired side-effects when making these changes. Also, this

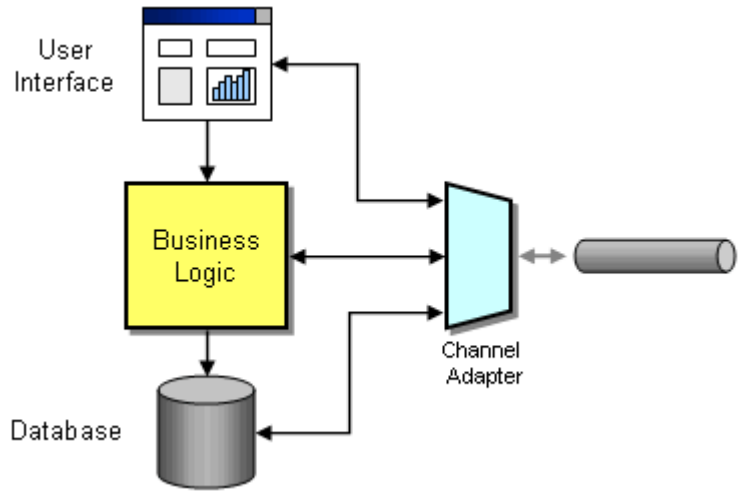
approach requires developers who are skilled with both the application logic and the messaging API. If we deal with a packaged application that we purchased from a third-party software vendor, we may not even have the option of changing the application code.

Use a *Channel Adapter* that can access the application's API or data and publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application.



The adapter acts as a messaging client to the messaging system and invokes applications functions via an application-supplied interface. This way, any application can connect to the messaging system and be integrated with other applications as long as it has a proper *Channel Adapter*.

The *Channel Adapter* can connect to different layers of the application's architecture, depending on that architecture and the data the messaging system needs to access.



A Channel Adapter Connecting to Different Layers of an Application

- **User Interface Adapter**. Sometimes disparagingly called "screen scraping," these types of adapters can be very effective in many situations. For example, an application may be implemented on a platform that is not supported by the messaging system. Or, the owner of the application may have little interest in supporting the integration. This eliminates the option of running the *Channel Adapter* on the application platform. However, the user interface is usually available from other machines and platforms (e.g. 3270 terminals). Also, the surge of Web-based thin-client architectures has caused a certain revival of user interface integration. HTML-based user interfaces make it very easy to make an HTTP request and parse out the results. Another advantage of user-interface integration is that no direct access to the application internals is needed. In some

cases, it may not be desirable or possible to expose internal functions of a system to the integration solution. Using a user-interface adapter, other applications have the exact same access to the application as a regular user. The downside of user interface adapters is the potential brittleness and low speed of the solution. The application has to parse "user" input and render a screen in response, just so that the *Channel Adapter* can parse the screen back into raw data. This process involves many unnecessary steps and can be slow. Also, user interfaces tend to change more frequently than the core application logic. Every time the user interface changes, the *Channel Adapter* is likely to have to be changed as well.

- **Business Logic Adapter**. Most business applications expose their core functions as an API. This interface may be a set of component (e.g. EJB's, COM objects, CORBA components) or a direct programming API (e.g., a C++, C#, or Java library). Since the software vendor (or developer) exposes these API's expressly for access by other applications, they tend to be more stable than the user interface. In most cases, accessing the API is also more efficient. In general, if the application exposes a well-defined API, this type of *Channel Adapter* is likely to be the best approach.
- **Database Adapter**. Most business applications persist their data inside a relational database. Since the information is already in the database, *Channel Adapter* can extract information directly from the database without the application ever noticing, which is very non-intrusive. The *Channel Adapter* can even add a trigger to the relevant tables and send messages every time the data in these tables changes. This type of *Channel Adapter* can also be very efficient and is quite universal, aided by the fact that only two or three database vendors dominate the market for relational databases. This allows us to connect to many applications with a relatively generic adapter. The downside of a database adapter is that we are poking around deep in the internals of an application. This may not be as risky if we simply read data, but making updates directly to the database can be very dangerous. Also, many application vendors consider the database schema "unpublished," meaning that they reserve the right to change it at will, which can make a database adapter solution brittle.

An important limitation of *Channel Adapter*s is that they can convert messages into application functions, but require message formatting that closely resembles the implementation of the components being adapted. For example, a database adapter typically requires the message field names of incoming messages to be the same as the names of tables and fields in the application database. This kind of message format is driven entirely by the internal structure of the application and is not a good message format to use when integration with other applications. Therefore, most *Channel Adapter*s require the combination with a *Message Translator* to convert the application-specific message into a message format that complies with the *Canonical Data Model*.

*Channel Adapter*s can often times run on a different computer than the application or the database itself. The *Channel Adapter* can the connect to the application logic or the database via protocols such as HTTP or ODBC. While this setup allows us to avoid installing additional software on the application or database serer, these protocols do not provide the same quality-of-service that a messaging channel provides, such as guaranteed delivery.

Some *Channel Adapter*s may be unidirectional. For example, if a *Channel Adapter* connects to an application via HTTP, it may only be able to consume messages and invoke functions on the application, but it may not be able to detect changes in the application data.

An interesting variation of the *Channel Adapter* is the *Metadata Adapter*, sometimes called *Design-Time Adapter*. This type of data does not invoke application functions, but extracts metadata, data that describes the internal data formats of the application. This metadata can then be used to configure *Message Translator*s or to detect changes in the application data formats (see *Introduction to Message Transformation*). Many application interfaces support the extraction of metadata. For example, most commercial databases provide a system tables that contain a description of the application tables. Likewise, most component frameworks (e.g. J2EE, .NET) provide special "reflection" functions that allow a component to enumerate methods provided by another component.

A special form of the *Channel Adapter* is the *Messaging Bridge*. The *Messaging Bridge* connects the messaging system to another messaging system as opposed to a specific application. Typically, a *Channel Adapter* is implemented as a *Transactional Client* to ensure that each piece of work the adapter does succeeds in both the messaging system and the other system being adapted.

Example: Stock Trading

A stock trading system may wish to keep a log of all of a stock's prices in a database table. The messaging system may include a relational database adapter that logs each message from a channel to a specified table and schema. This channel-to-RDBMS adapter is a *Channel Adapter*. The system may also be able to recieve external quote requests from the Internet (TCP/IP or HTTP) and send them on its internal quote-request channel with the internal quote requests. This Internet-to-channel adapter is a *Channel Adapter*.

Example: Commercial EAI Tools

Commercial EAI vendors provide a collection of *Channel Adapter*s as part of their offerings. Having adapters to all major application packages available simplifies development of an integration solution greatly. Most vendors also provide more generic database adapters as well as software development kits (SDK's) to develop custom adapters.

Example: Legacy Platform Adapters

A number of vendors provide adapters from common messaging system to legacy systems executing on platforms such as as UNIX, MVS, OS/2, AS/400, Unisys, and VMS. Most of these adapters are specific to a certain messaging system. For example, Envoy Technologies' EnvoyMQ is a *Channel Adapter* that connects many legacy platforms with MSMQ. It consists of a client component that runs on the legacy computer and a server component that runs on a Windows computer with MSMQ.

Example: Web Services Adapters

Many messaging systems provide *Channel Adapter*s to convert SOAP messages between HTTP transport and the messaging system. This way, SOAP messages can be transmitted over an intranet using the messaging system, and over the global Internet (and through firewalls) using HTTP. One example is the Web Services Gateway for IBM's WebSphere Application Server.

Related patterns: *Canonical Data Model*, *Message Channel*, *Introduction to Message Transformation*, *Message Translator*, *Messaging*, *Messaging Bridge*, *Transactional Client*

Messaging Bridge

An enterprise is using *Messaging* to enable applications to communicate. However, the enterprise uses more than one messaging system, which confuses the issue of which messaging system an application should connect to.

How can multiple messaging systems be connected so that messages available on one are also available on the others?

A common problem is an enterprise that uses more than one messaging system. This can occurr because of a merger or aquisition between two different companies that have standardized around different messaging products. Sometimes a single enterprise that uses one messaging system to integrate their mainframe/legacy systems chooses another for their J2EE or .NET web application servers, and then needs to integrate the two messaging systems. Another common occurrance is an application that participates as part of multiple enterprises, such as a B2B client that wants to be a bidder in multiple auctioning systems; if the various auction clusters use different messaging systems, the bidder applications within an enterprise may wish to consolodate the messages from several external messaging systems onto a single internal messaging system. Another example: An extremely large enterprise with a huge number of *Message Channel*s and *Message Endpoint*s may require more than one instance of the messaging system, which means those instances must be connected somehow.

If the messages on one system are of no interest to the applications using the other messaging system, then the systems can remain completely separate. But because the applications are part of the same enterprise, often some applications using one messaging system will be interested in messages being transmitted on another messaging system.

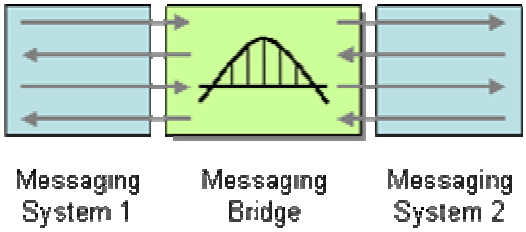
A common misconception is that a standardized messaging API such as JMS solves this problem; it does not. JMS makes two complient messaging systems look the same to a client application, but it does nothing to make the two messaging systems work with each other. For the messaging systems to work together, they need to be interoperable, meaning that they use the same message format and transmit a message from one message store to the next in the same way. Messaging

systems from two different vendors are rarely interoperable; a message store from one vendor can only work with other message stores from the same vendor.

Each application in the enterprise could choose to implement a client for each messaging system in the enterprise, but that would increase complexity and duplication in the messaging layer. This redundancy would become especially apparent if the enterprise added yet another messaging system and all of the applications had to be modified. On the other hand, each application could choose to only interface with one messaging system and ignore data on the other messaging systems. This would make the application simpler but could cause it to ignore a great deal of enterprise data.

What is needed is a way for messages on one messaging system that are of interest to applications on another messaging system to be made available on the second messaging system as well.

Use a *Messaging Bridge*, a connection between messaging systems, to replicate messages between systems.



Typically, there is no practical way to connect two complete messaging systems, so instead we connect individual, corresponding channels between the messaging systems. The *Messaging Bridge* is a set of *Channel Adapter*s, where the non-messaging client is a actually another messaging system, and where each pair of adapters connects a pair of corresponding channels. The bridge acts as map from one set of channels to the other, and also transforms the message format of one system to the other. The connected channels may be used to transmit messages between traditional clients of the messaging system, or strictly for messages intended for other messaging systems.

You may need to implement the *Messaging Bridge* for your enterprise yourself. The bridge is a specialized *Message Endpoint* application that is a client of both messaging systems. When a message is delivered on a channel of interest in one messaging system, the bridge consumes the message and sends another with the same contents on the corresponding channel in the other messaging system.

Many messaging system vendors have product extensions for bridging to messaging systems from other vendors. Thus you may be able to buy a solution rather than build it yourself.

If the other "messaging system" is really a simpler protocol, such as HTTP, apply the *Channel Adapter* pattern.

Messaging Bridge is necessary because different messaging system implementations have their own proprietary approaches for how to represent messages and how to forward them from one store to the next. Web services may be standardizing this, such that two messaging system installs, even from different vendors, may be able to act as one by transferring messaging using web services standards. See the discussion of WS-Reliability and WS-ReliableMessaging in *Emerging Standards and Futures in Enterprise Integration*.

Example: Stock Trading

A brokerage house may have one messaging system that the applications in its various offices use to communicate. A bank may have a different messaging system that the applications in its various branches use to communicate. If the brokerage and the bank decide to merge into a single company that offers bank accounts and investment services, which messaging system should the combined company use? Rather than redesigning half of the company's applications to use the new messaging system, the company can use a *Messaging Bridge* to connect the two messaging systems. This way, for example, a banking application and a brokerage application can coordinate to transfer money between a savings account and a securities trading account.

Example: MSMQ Bridges

MSMQ defines an architecture based on connector servers that enables connector applications to send and receive messages using other (non-MSMQ) messaging systems. An MSMQ application using a connector server can perform the same opperations on channels from other messaging systems that it can perform on MSMQ channels. [Dickman, pp.42-45]

Microsoft's Host Integration Server product contains an MSMQ-MQSeries Bridge service that makes the two messaging systems work together. It lets MSMQ applications send messages via MQSeries channels and vise versa, making the two messaging systems act as one.

Envoy Tecnologies, licenser of the MSMQ-MQSeries Bridge, also has a related product called Envoy Connect. It connects MSMQ and BizTalk servers with messaging servers running on non-Windows platforms, especially the J2EE platform, coordinating J2EE and .NET messaging within an enterprise.

Example: SonicMQ Bridges

Sonic Software's SonicMQ has SonicMQ Bridge products that support IBM MQSeries, TIBCO TIB/Rendezvous, and JMS. This enables messages on Sonic channels to be transmitted on other messaging systems' channels as well.

Related patterns: *Channel Adapter*, *Emerging Standards and Futures in Enterprise Integration*, *Message Channel*, *Message Endpoint*, *Messaging*

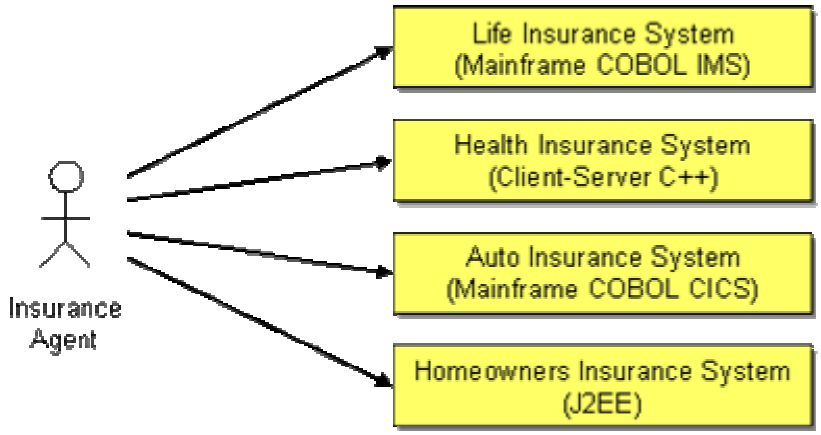
Message Bus

An enterprise contains several existing systems that must be able to share data and operate in a unified manner in response to a set of common business requests.

What is an architecture that enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others?

An enterprise often contains a variety of applications that operate independently, yet need to work together in a unified manner. Enterprise Application Integration (EAI) describes a solution to this problem but doesn't describe how to accomplish it.

For example, consider an insurance company that sells different kinds of insurance products (life, health, auto, home, etc.). As a result of corporate mergers, and of the varying winds of change in IT development, the enterprise consists of a number of separate applications for managing the company's various products. An insurance agent trying to sell a customer several different types of policies must log into a separate system for each policy, wasting effort and increasing the opportunity for mistakes.



Insurance Company EAI Scenario

The agent needs a single, unified application for selling customers a portfolio of policies. Other types of insurance company employees, such as claims adjusters and customer service representatives, need their own applications for working with the insurance products, but also want their applications to present a unified view. The individual product applications need to be able to work together, perhaps to offer a discount will purchasing more than one policy, and perhaps to process a claim that is covered by more than one policy.

The IT department could rewrite the product applications to all use the same technology and work together, but the amount of time and money to replace systems that already work (even though they don't work together) is prohibitive. IT could create a unified application for the agents, but this application needs to connect to the systems that actually manage the policies.

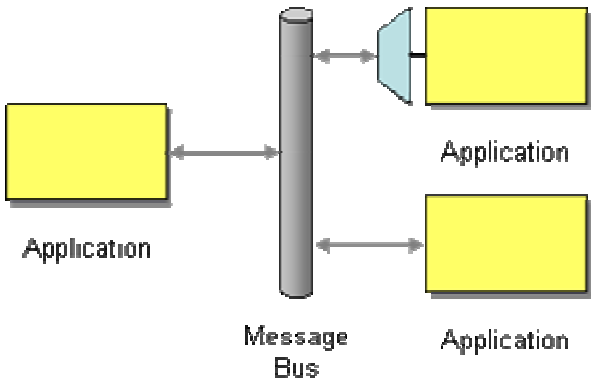
Rather than unifying the systems, this new application creates one more system that doesn't integrate with the others.

The agent application could integrate with all of these other systems, but that would make it much more complex. The complexity would be duplicated in the applications for claims adjusters and customer service representatives. Furthermore, these unified user applications would not help the product applications integrate with each other.

Even if all of these applications could be made to work together, any change to the enterprise's configuration could make it all stop working. Not all applications will be available all of the time, yet the ones that are running need to be able to continue with minimal impact from those that are not running. Over time, applications will need to be added to and removed from the enterprise, with minimal impact on the other applications.

What is needed is an integration architecture that enables the product applications to coordinate in a loosely coupled way, and for user applications to be able to integrate with them.

Structure the connecting middleware between these applications as a *Message Bus* that enables them to work together using messaging.

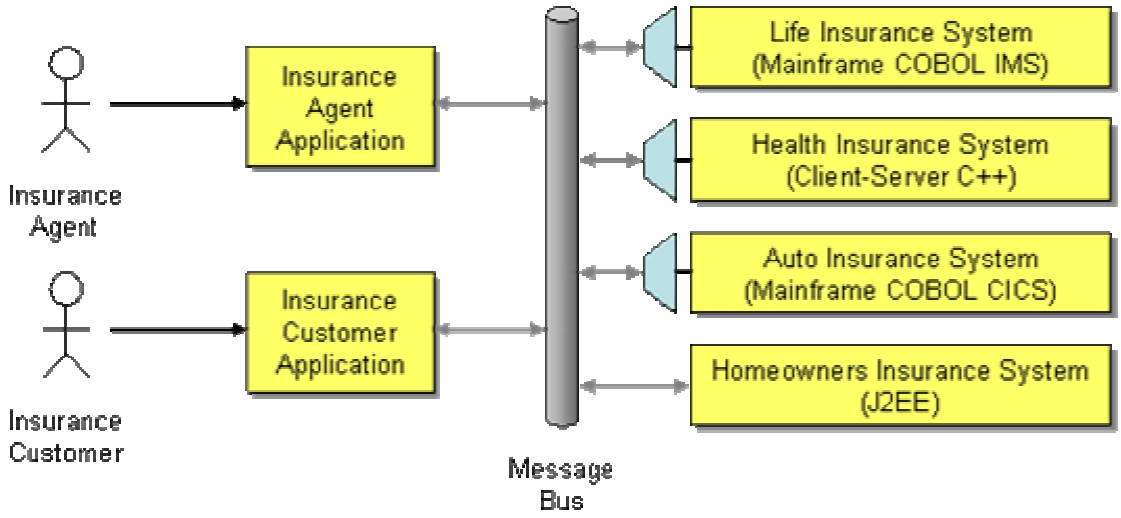


A *Message Bus* is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces. This is analogous to a communications bus in a computer system, which serves as the focal point for communication between the CPU, main memory, and peripherals. Just as in the hardware analogy, there are a number of pieces that come together to form the message bus:

- **Common communication infrastructure** — Just as the physical pins and wires of a PCI bus provide a common, well-known physical infrastructure for a PC, a common infrastructure must serve the same purpose in a message bus. Typically, a messaging system is chosen to serve as the physical communications infrastructure, providing a cross-platform, cross-language universal adapter between the applications. The infrastructure may include *Message Router* capabilities to facilitate the correct routing of messages from system to system. Another common option is to use *Publish-Subscribe Channel*s to facilitate sending messages to all receivers.

- **Adapters** — The different systems must find a way to interface with the message bus. Most commonly, this is done with commercial or custom *Channel Adapter*s and *Service Activator*s that can handle things like invoking CICS transactions with the proper parameters, or representing the general data structures flowing on the bus in the specific and particular way they should be represented inside each system. This also requires a *Canonical Data Model* that all systems can agree on.
- **Common Command Structure** — Just like PC architectures have a common set of commands to represent the different operations possible on the physical bus (read bytes from an address, write bytes to an address), there needs to be common commands that are understood by all the participants in the *Message Bus*. *Command Message* illustrates how this feature works. Another common implementation for this is the *Datatype Channel*, where a *Message Router* makes an explicit decision as to how to route particular messages (like Purchase Orders) to particular endpoints. It is at the end that the analogy breaks down, since the level of the messages carried on the bus are much more fine-grained than the "read/write" kinds of messages carried on a physical bus.

In our EAI example, a *Message Bus* could serve as a universal connector between the various insurance systems, and as a universal interface for client applications that wish to connect to the insurance systems.



Insurance Company Message Bus

Here we have a two GUI's that only know about the *Message Bus*—they are entirely unaware of the complexities of idiosyncrasies of the underlying systems. The bus is responsible for routing *Command Message*s to the proper underlying systems. In some cases, the best way to handle the command messages is to build an adapter to the system that interprets the command and then communicates with the system in a way it understands (invoking a CICS transaction, for instance, or calling a C++ API). In other cases, it may be possible to build the command-processing logic directly into the existing system as an additional way to invoke current logic.

Once the *Message Bus* has been developed for the agent GUI, it is easy to reuse for other GUIs such as those for claims processors, customer service representatives, and a web interface for

customer to browse their own accounts. The features and security control of these GUI applications differ, but their need to work with the backend applications is the same.

A *Message Bus* forms a simple, useful *service-oriented architecture* for an enterprise. Each service has at least one request channel that accepts requests of an agreed-upon format, and probably a corresponsing reply channel that supports a specified reply format. Any participant application can make use of these services by making requests and waiting for replies. The request channels, in effect, act as a directory of the services available.

A *Message Bus* requires that all of the applications using the bus use the same *Canonical Data Model*. Applications adding messages to the bus may need to depend on *Message Router*s to route the messages to the appropriate final destinations. Applications not designed to interface with a messaging system may require *Channel Adapter*s and *Service Activator*s.

Example: Stock Trading

A stock trading system may wish to offer a unified suite of services including stock trades, bond auctions, price quotes, portfolio management, etc. This may require several separate back-end systems that have to coordinate with each other. To unify the services for a front-end customer GUI, the system could employ an intermediate application that offered all of these services and delegated their performane to the back-end systems. The back-end systems could even coordinate through this intermediary application. However, the intermediary application would tend to become a bottleneck and a single point of failure.

Rather than an intermediary application, a better approach might be a *Message Bus* with channels for requesting various services and getting their responses. This bus could also enable to back-end systems to coordinate with each other. A front-end system could simply connect to the bus and use it to invoke services. The bus could relatively easily be distributed across multiple computers to provide load distribution and fault tolerance.

Once the *Message Bus* is in place, connecting front-end GUI's would be relatively easy; they each just need to send and receive messages from the proper channels. One GUI might enable a retail broker to manage his customers' portfolios. Another web-based GUI could enable any customer with a web browser to manage his own portfolio. Another non-GUI front-end might support personal finance programs like Intuit's Quicken and Microsoft's Money, enabling customers using those programs to download trades and current prices. Once the *Message Bus* is in place, developing new user applications is much simpler.

Likewise, the trading system may want to take advantage of new back-end applications such as switching one trading application for another or spreading price quote requests across multiple applications. Implementing a change like this is a simple as adding and removing applications from the *Message Bus*. Once the new applications are in place, none of the other applications have to change; they just keep sending messages on the bus' channels as usual.

Related patterns: *Canonical Data Model*, *Channel Adapter*, *Command Message*, *Datatype Channel*, *Message Router*, *Service Activator*, *Publish-Subscribe Channel*

5. Message Construction

Introduction

In *Introduction to Messaging Systems*, we discussed *Message*. When two applications wish to exchange a piece of data, they do so by wrapping it in a message. Whereas a *Message Channel* cannot transmit raw data per se, it can transmit the data wrapped in a message.

Deciding to create a *Message* and send it raises several other issues:

Message intent — Messages are ultimately just bundles of data, but the sender can have different intentions for what it expects the receiver to do with the message. It can send a *Command Message*, specifying a function or method on the receiver that the sender wishes to invoke. The sender is telling the receiver what code to run. It can send a *Document Message*, enabling the sender to transmit one of its data structures to the receiver. The sender is passing the data to the receiver, but not specifying what the receiver should necessarily do with it. Or it can send an *Event Message*, notifying the receiver of a change in the sender. The sender is not telling the receiver how to react, just providing notification.

Returning a response — When an application sends a message, it often expects a response confirming that the message has been processed and providing the result. This is a *Request-Reply* scenario. The request is usually a *Command Message*, and the reply is a *Document Message* containing a result value or an exception. The requestor should specify a *Return Address* in the request to tell the replier what channel to use to transmit the reply. The requestor may have multiple requests in process, so the reply should contain a *Correlation Identifier* that specifies which request this reply corresponds to. [JMS11, pp.27-28]

There are two common *Request-Reply* scenarios worth noting that both involve a *Command Message* request and a corresponding *Document Message* reply. In the first scenario, *Messaging RPC*, the requestor not only wants to invoke a function on the replier, but also wants the return value from the function. This is how applications perform an RPC (Remote Procedure Call) using *Messaging*. In the other scenario, *Messaging Query*, the requestor is performing a query that the replier will execute and return the results in the reply. This is how applications use messaging to perform a query remotely.

Huge amounts of data — Sometimes applications want to transfer a really large data structure, one that may not fit comfortably in a single message. In this case, break the data into more managable chunks and send them as a *Message Sequence*. The chunks have to be sent as a sequence, and not just a bunch of messages, so that the receiver can reconstruct the original data structure.

Slow messages — A concern with messaging is that the sender often does not know how long it will take for the receiver to receive the message. Yet the message contents may be time-sensitive, such that if the message isn't received by a deadline, it should just be ignored and discarded. In this situation, the sender can use *Message Expiration* to specify an expiration date. If the messaging system cannot deliver a message by its expiration, it should discard the message. If a receiver gets a message after its expiration, it should discard the message.

So it's not enough to decide to use a *Message*; anytime data needs to be transferred, it will be done through a message. This chapter will explain the other decisions that go into making messages work.

Command Message

An application needs to invoke functionality provided by other applications. It would typically use *Remote Procedure Invocation*, but would like to take advantage of the benefits of using *Messaging*.

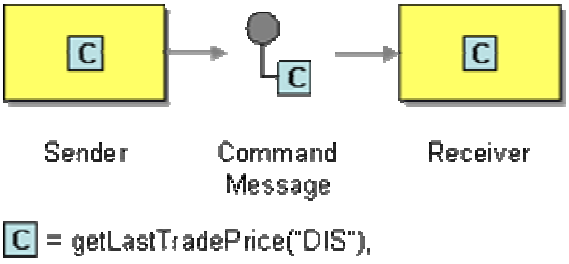
How can messaging be used to invoke a procedure in another application?

The advantage of *Remote Procedure Invocation* is that it's synchronous, so the call is performed immediately while the caller's thread blocks. But that's also a disadvantage. If the call cannot be made right now—either because the network is down or because the remote process isn't running and listening—then the call doesn't work. If the call were asynchronous, it could keep trying until the procedure in the remote application is successfully invoked.

Messaging is asynchronous. If the procedure were somehow invoked by a *Message*, the procedure would not even have to be remotely accessible, it could just be a local procedure invoked within its own process. So the question is how to make a procedure call into a message.

Luckily, there's a well-established pattern for how to encapsulate a request as an object. The Command pattern [GoF] shows how to turn a request into an object that can be stored and passed around. If this object were a message, then it could be stored in and passed around through a *Message Channel*. Likewise, the command's state (if any) can be stored in the message's state.

Use a *Command Message* to reliably invoke a procedure in another application.



There is no specific message type for commands; a *Command Message* is simply a regular message that happens to contain a command. In JMS, the command message could be any type of message; examples include an `ObjectMessage` containing a `Serializable` command object, a `TextMessage` containing the command in XML form, etc. In .NET, a command message is a `Message` with a command stored in it. A Simple Object Access Protocol (SOAP) request is a command message.

*Command Message*s are usually sent on a *Point-to-Point Channel* so that each command will only be consumed and invoked once.

Example: SOAP and WSDL

With the SOAP protocol [SOAP 1.1] and WSDL service description [WSDL 1.1], when using RPC-style SOAP messages, the request message is an example of this *Command Message* pattern. With this usage, the SOAP message body (an XML document) contains the name of the method to invoke in the receiver and the parameter values to pass into the method. This method name must be the same as one of the message names defined in the receiver's WSDL.

This example from the SOAP spec invokes the receiver's `GetLastTradePrice` method with a single parameter called `symbol`:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In a SOAP command, one might expect the method name to be the value of some standard `<method>` element; actually, the method name is the name of the method element, prefixed by the `m` namespace. Having a separate XML element type for each method makes validating the XML data much more precise, because the method element type can specify the parameters' names, types, and order.

Related patterns: *Remote Procedure Invocation*, *Message*, *Message Channel*, *Messaging*, *Point-to-Point Channel*

Document Message

An application would like to transfer data to another application. It could do so using *File Transfer* or *Shared Database*, but those approaches have shortcomings. The transfer might work better using *Messaging*.

How can messaging be used to transfer data between applications?

This is a classic problem in distributed processing: One process has data another one needs.

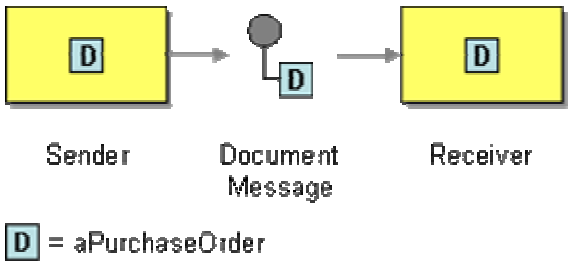
File Transfer is easy to use, but doesn't coordinate applications very well. A file written by one application may sit unused for quite a while before another another application reads it. If several applications are supposed to read it, it'll be unclear who should take responsibilty for deleting it.

Shared Database requires adding new schema to the database to accomdate the data, or force-fitting the data into the existing schema. Once the data is in the database, there's the risk that other applications which should not have access to the data now do. Triggering the receiver of the data to now come read it can be difficult, and coordinating multiple readers confuses who should delete the data.

Remote Procedure Invocation can be used to send the data, but then the caller is also telling the receiver—via the procedure being invoked—what to do with the data. Likewise, a *Command Message* would transfer the data, but would be overly specific about what the receiver should do with the data.

Yet we do want to use *Messaging* to transfer the data. *Messaging* is more reliable than an RPC. A *Point-to-Point Channel* can be used to make sure that only one receiver gets the data (no duplication), or a *Publish-Subscribe Channel* can be used to make sure that any receiver who wants the data gets a copy of it. So the trick is to take advantage of *Messaging* without making the *Message* too much like an RPC.

Use a *Document Message* to reliably transfer a data structure between applications.



Whereas a *Command Message* tells the receiver to invoke certain behavior, a *Document Message* just passes data and lets the receiver decide what, if anything, to do with the data. The data is a single unit of data, a single object or data structure which may decompose into smaller units.

*Document Message*s can seem very much like *Event Message*s; the main difference is a matter of timing and content. The important part of a *Document Message* is its content, the document. Successfully transferring the document is important; the timing of when it is sent and received is less important. *Guaranteed Delivery* may be a consideration; *Message Expiration* probably is not.

A *Document Message* can be any kind of message in the messaging system. In JMS, the document message may be an `ObjectMessage` containing a `Serializable` data object for the document, or it may be a `TextMessage` containing the data in XML form. In .NET, a document message is a `Message` with the data stored in it. A Simple Object Access Protocol (SOAP) reply message is a document message.

*Document Message*s are usually sent using a *Point-to-Point Channel* to move the document from one process to another without duplicating it. *Messaging* can be used to implement simple workflow by passing a document to an application that modifies the document and then passes it to another application. In some cases, a document message can be broadcast via a *Publish-Subscribe Channel*, but this creates multiple copies of the document. Either the copies need to be read-only, or if the receivers change the copies, there will be multiple copies of the document in the system that contain different data. In *Request-Reply*, the reply is usually a *Document Message* where the result value is the document.

Example: Java and XML

The following example (drawn from the example XML schema in [Graham02]) shows how a simple purchase order can be represented as XML and sent as a message using JMS.

```
Session session = // Obtain the session
Destination dest = // Obtain the destination
MessageProducer sender = session.createProducer(dest);
String purchaseOrder =
"       <po id=\"48881\" submitted=\"2002-04-23\">
                <shipTo>
                        <company>Chocoholics</company>
                        <street>2112 North Street</street>
                        <city>Cary</city>
                        <state>NC</state>
                        <postalCode>27522</postalCode>
                </shipTo>
                <order>
                        <item sku=\"22211\" quantity=\"40\">
                                <description>Bunny, Dark Chocolate,
Large</description>
                        </item>
                </order>
        </po>";
```

```
TextMessage message = session.createTextMessage();
message.setText(purchaseOrder);
sender.send(message);
```

Example: SOAP and WSDL

With the SOAP protocol [SOAP 1.1] and WSDL service description [WSDL 1.1], when using document-style SOAP messages, the SOAP message is an example of this pattern. The SOAP message body is an XML document (or some kind of data structure that has been converted into an XML document), and the SOAP message transmits that document from the sender (e.g., client) to the receiver (e.g., server).

When using RPC-style SOAP messages, the response message is an example of this pattern. With this usage, the SOAP message body (an XML document) contains the return value from the method that was invoked.

This example from the SOAP spec returns the answer from invoking the `GetLastTradePrice` method:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
          <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Related patterns: *Command Message*, *Remote Procedure Invocation*, *Event Message*, *File Transfer*, *Guaranteed Delivery*, *Message*, *Message Expiration*, *Messaging*, *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Request-Reply*, *Shared Database*

Event Message

Several applications would like to use event-notification to coordinate their actions, and would like to use *Messaging* to communicate those events.

How can messaging be used to transmit events from one application to another?

Sometimes an event occurs in one object that another object needs to know about. The classic example is a model that changes its state and must notify its views so that they can redraw themselves. Such change notification can also be

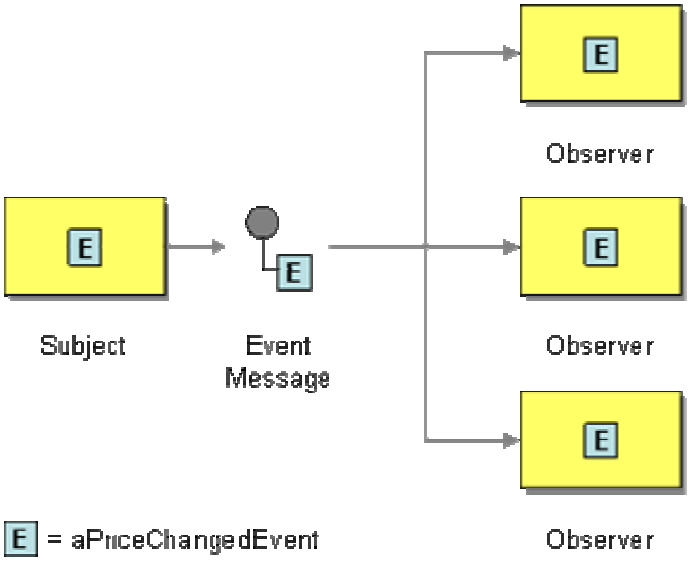
useful in distributed systems. For example, in a B2B system, one business may need to notify others of price changes or a whole new product catalog.

A process can use *Remote Procedure Invocation* to notify other applications of change events, but that requires that the receiver accept the event immediately, even if it doesn't want events right now. RPC also requires that the announcing process know every listener process and invoke an RPC on each listener.

The Observer pattern [GoF] describes how to design a subject that announces events and observers that consume events. A subject notifies an observer of an event by calling the observer's `Update()` method. `Update()` can be implemented as an RPC, but that would have all of RPC's shortcomings.

It would be better to send the event notification asynchronously, as a *Message*. This way, the subject can send the notification when it's ready and each observer can receive the notification if and when it's ready.

Use an *Event Message* for reliable, asynchronous event notification between applications.



When a subject has an event to announce, it will create an event object, wrap it in a message, and send it on a channel. The observer will receive the event message, get the event, and process it. Messaging does not change the event notification, just makes sure that the notification gets to the observer.

A *Event Message* can be any kind of message in the messaging system. In Java, an event can be an object or data such as an XML document. Thus they can be transmitted through JMS as an `ObjectMessage`, `TextMessage`, etc. In .NET, an event message is a `Message` with the event stored in it.

The difference between an *Event Message* and a *Document Message* is a matter of timing and content. An event's contents is typically less important. Many events are empty; their mere occurrance tells the observer to react. An event's timing is very important; the subject should issue an event as soon as a change occurs, and the observer should process it quickly while it's still relevant. *Guaranteed Delivery* is usually not very helpful with events because they're frequent and need to be delivered quickly. *Message Expiration* can be very helpful to make sure that an event is processed quickly or not at all.

Our B2B example could use *Event Message*s, *Document Message*s, or a combination of the two. If a message says that the price for computer disk drives has changed, that's an event. If the message provided information about the disk drive, including its new price, that's a document being sent as an event. Another message that announces the new catalog and its URL is an event, whereas a similar message that actually contains the new catalog is an event that contains a document.

Which is better? The Observer pattern describes this as a trade-off between a push model and a pull model. The *push model* sends information about the change as part of the update, whereas the *pull model* sends minimal information and observers that want more information request it by sending `GetState()` to the subject. The two models relate to messaging like this:

Push model – The message is a combined document/event message; the message's delivery announces that the state has occurred and the message's contents are the new state. This is more efficient if all observers want these details, but otherwise can be the worst of both worlds: A large message that is sent frequently and often ignored by many observers.

Pull model – There are three messages:

- **Update** – An *Event Message* that notifies the observer of the event.
- **State Request** – A *Command Message* an interested observer uses to request details from the subject.
- **State Reply** – A *Document Message* the subject uses to send the details to the observer.

The advantage of the pull model is that the update messages are small, only interested observers request details, and potentially each interested observer can request the details it specifically is interested in. The disadvantage is the channels needed and traffic caused by three messages instead of one.

For more details on how to implement Observer using messaging, see *JMS Publish/Subscribe Example*.

There is usually no reason to limit an event message to a single receiver via a *Point-to-Point Channel*; the message is usually broadcast via a *Publish-Subscribe Channel* so that all interested processes receive notification. Whereas a *Document Message* needs to be consumed so that the document is not lost, a receiver of *Event Message*s can often ignore the messages when it's too busy to process them, so the subscribers can often be non-durable (not *Durable Subscriber*s). *Event Message* is a key part of implementing the Observer pattern using messaging.

Related patterns: *Command Message*, *Document Message*, *Durable Subscriber*, *Remote Procedure Invocation*, *Guaranteed Delivery*, *Message*, *Message Expiration*, *Messaging*, *JMS Publish/Subscribe Example*, *Point-to-Point Channel*, *Publish-Subscribe Channel*

Request-Reply

When two applications communicate via *Messaging*, the communication is one-way. The applications may want a two-way conversation.

When an application sends a message, how can it get a response from the receiver?

Messaging provides one-way communication between applications. *Message*s travel on a *Message Channel* in one direction, from the sender to the receiver. This asynchrnous transmission makes the delivery more reliable and decouples the sender from the receiver.

The problem is that communication between components often needs to be two-way. When a program calls a function, it receives a return value. When it executes a query, it receives query results. When one component notifies another of a change, it may want to receive an acknowledgement.

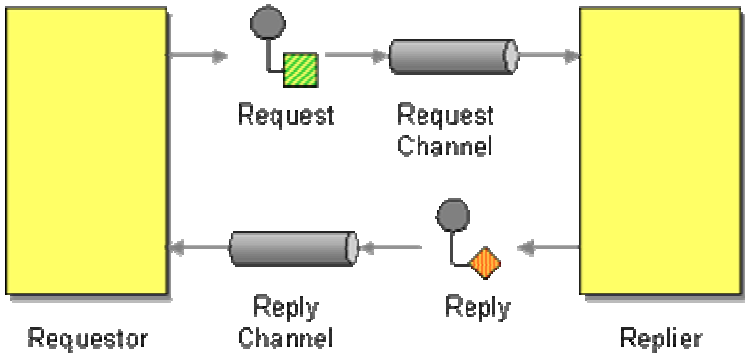
How can messaging be two-way?

Perhaps a sender and receiver could share a message simultaneously. Then each application could add information to the message for the other to consume. But that is not how messaging works. A message is first sent, then received, such that the sender and receiver cannot both access the message at the same time.

Perhaps the sender could keep a reference to the message. Then, once the receiver placed its response into the message, the sender could pull the message back. This may work for notes clipped to a clothesline, but it is not how a *Message Channel* works. A channel transmits messages in one direction.

What is needed is a two-way message on a two-way channel.

Send a pair of *Request-Reply* messages, each on its own channel.



Request-Reply has two participants:

1. **Requestor** – Sends a request message and waits for a reply message.
2. **Replier** – Receives the request message and responds with a reply message.

The request channel can be a *Point-to-Point Channel* or a *Publish-Subscribe Channel*. The difference is whether the request should be broadcast to all interested parties or should only be processed by a single consumer. The reply channel, on the other hand, is almost always point-to-point, because it usually makes no sense to broadcast replies–they should only be returned to the requestor.

When a caller performs a *Remote Procedure Invocation*, the caller's thread must block while it waits for the response. With *Request-Reply*, the requestor has two approaches for receiving the reply:

1. **Synchronous Block** – A single thread in the caller sends the request message, blocks (as a *Polling Consumer*) to wait for the reply message, then processes the reply. This is simple to implement, but if the requestor crashes, it will have difficulty re-establishing the blocked thread. The request thread awaiting the response implies that there is only one outstanding request, or that the reply channel for this request is private for this thread.
2. **Asynchronous Callback** – One thread in the caller sends the request message and sets up a callback for the reply. A separate thread listens for reply messages. When a reply message arrives, the reply thread invokes the appropriate callback, which re-establishes the caller's context and processes the reply. This approach enables multiple outstanding requests to share a single reply channel, and a single reply thread to process replies for multiple request threads. If the requestor crashes, it can recover by simply restarting the reply thread. An added complexity, however, is the callback mechanism that must re-establish the caller's context.

By itself, two applications sending requests and replies to each other are not very helpful. What is interesting is what the two messages represent.

1. **Messaging RPC** – This is how to implement *Remote Procedure Invocation* using messaging. The request is a *Command Message* that describes the function the replier should invoke. The reply is a *Document Message* that contains the function's return value or exception.
2. **Messaging Query** – This is how to perform a remote query using messaging. The request is a *Command Message* containing the query, and the reply is the results of the query, perhaps a *Message Sequence*.
3. **Notify/Acknowledge** – This provides for event notfication with acknowledgement using messaging. The request is an *Event Message* that provides notification and the reply is a *Document Message* acknowledging the notification. The acknowledgement may itself be another request, one seeking details about the event.

The request is like a method call. As such, the reply is one of three possibilities:

1. **Void** – Simply notifies the caller that the method has finished so that the caller can proceed.
2. **Result value** – A single object that is the method's return value.
3. **Exception** – A single exception object indicating that the method aborted before completing successfully, and indicating why.

The request should contain a *Return Address* to tell the replier where to send the reply. The reply should contain a *Correlation Identifier* that specifies which request this reply is for.

Example: SOAP 1.1 Messages

SOAP messages come in *Request-Reply* pairs. A SOAP request message indicates a service the sender wishes to invoke on the receiver, whereas a SOAP response message contains the result of

the service invocation. The response message either contains a result value or a fault–the SOAP equivalent of an exception. [SOAP 1.1]

Example: SOAP 1.2 Response Message Exchange Pattern

Whereas SOAP 1.1 has response messages and they are loosely described, SOAP 1.2 introduces an explicit Request-Response Message Exchange Pattern. [SOAP 1.2 Part 2] This patterns describes a separate, potentially asynchronous response to a SOAP request.

Example: JMS Requestor Objects

JMS includes a couple of features that can be used to implement *Request-Reply*.

A `TemporaryQueue` is a `Queue` that can be created programatically and that only lasts as long as the `Connection` used to create it. Only `MessageConsumer`s created by the same connection can read from the queue, so effectively it is private to the connection. [JMS11, pp.61-62]

How are `MessageProducer`s going to know about this newly created, private queue? A requestor will create a temporary queue and specify it in the reply-to property of a request message. (See *Return Address*.) A well-behaved replier will send the reply back on the specified queue, one that the replier wouldn't even know about if it weren't a property of the request message. This is a simple approach the requestor can use to make sure that the replies always come back to it.

The downside with temporary queues is that when their `Connection` closes, the queue and any messages in it are deleted. Likewise, temporary queues cannot provide *Guaranteed Delivery*; if the messaging system crashes, then the connection is lost, so the queue and its messages are lost.

JMS also provides `QueueRequestor`, a simple class for sending requests and receiving replies. A requestor contains a `QueueSender` for sending requests and a `QueueReceiver` for receiving replies. Each requestor creates its own temporary queue for receiving replies and specifies that in the request's reply-to property. [JMS11, p.78] A requestor makes sending a request and receiving a reply very simple:

```
QueueConnection connection = // obtain the connection
Queue requestQueue = // obtain the queue
Message request = // create the request message
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueRequestor requestor = new QueueRequestor(session, requestQueue );
Message reply = requestor.request(request);
```

One method–`request`–sends the request message and blocks until it receives the reply message.

`TemporaryQueue`, used by `QueueRequestor`, is a *Point-to-Point Channel*. Its *Publish-Subscribe Channel* equivalents are `TemporaryTopic` and `TopicRequestor`.

Related patterns: *Command Message*, *Correlation Identifier*, *Document Message*, *Remote Procedure Invocation*, *Event Message*, *Guaranteed Delivery*, *Message*, *Message Channel*, *Message Sequence*, *Messaging*, *Point-to-Point Channel*, *Polling Consumer*, *Publish-Subscribe Channel*, *Return Address*

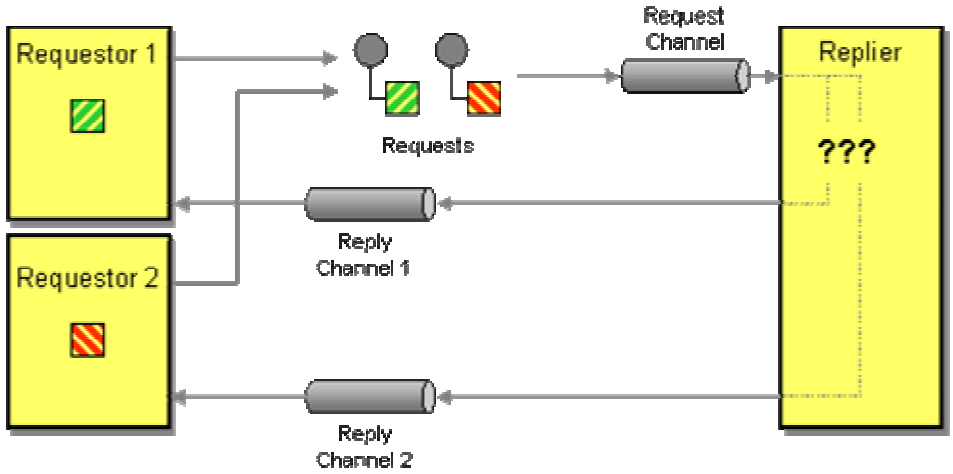
Return Address

My application is using *Messaging* to perform a *Request-Reply*.

How does a replier know where to send the reply?

Messages are often thought of as completely independent, such that any sender sends a message on any channel whenever it likes. However, messages are often associated, such as *Request-Reply* pairs, two messages which appear independent but where the reply message has a one-to-one correspondence with the request message that caused it. Thus the replier that processes the request message cannot simply send the reply message on any channel it wishes, it must send it on the channel the requestor expects the reply on.

Each receiver could automatically know which channel to send replies on, but hard coding such assumptions makes the software less flexible and more difficult to maintain. Furthermore, a single replier could be processing calls from several different requestors, so the reply channel is not the same for every message; it depends on what requestor sent the request message.



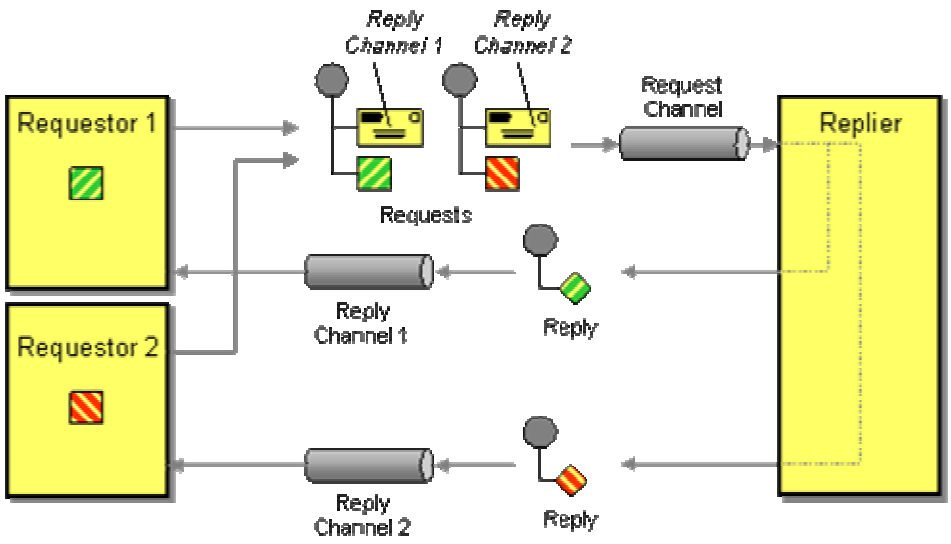
Uncertain Where to Send Replies

A requestor potentially may not want a reply sent back to itself. Rather, it may have an associated callback processor to process replies, and the callback processor may monitor a different channel than the requestor does (or the requestor may not monitor any channels at all). The requestor could have multiple callback processors such that replies for different requests from the same requestor should be sent to different processors.

The reply channel will not necessarily transmit replies back to the requestor; it will transmit them to whomever the requestor wants to process the replies, because it's listening to the channel the requestor specified. So knowing what requestor sent a request or what channel it was sent on does not necessarily tell the replier what channel to send the reply on. Even if it did, the replier would still have to infer which reply channel to use for a particular requestor or request channel. It's easier for the request to explicitly specify which reply channel to use.

What is needed is a way for the requestor to tell the replier where and how to send back a reply.

The request message should contain a *Return Address* that indicates where to send the reply message.



This way, the replier does not need to know where to send the reply, it can just ask the request. If different messages to the same replier require replies to different places, the replier knows where to send the reply for each request. This encapsulates the knowledge of what channels to use for requests and replies within the requestor so those decisions do not have to be hard coded within the replier. A *Return Address* is put in the header of a message because it's not part of the data being transmitted.

A message's *Return Address* is analogous to the reply-to field in an e-mail message. The "reply-to" e-mail address is usually the same as the "from" address, but the sender can set it to a different address to receive replies in a different account than the one used to send the original message.

When the reply is sent back the channel indicated by the *Return Address*, it may also need a *Correlation Identifier*. The *Return Address* tells the receiver what channel to put the reply message on; the correlation identifier tells the sender which request a reply is for.

Example: JMS Reply-To Property

JMS messages have a predefined property for *Return Addresses*, `JMSReplyTo`. Its type is a `Destination` (a `Topic` or `Queue`), rather than just a string for the destination name, which ensures that the destination (e.g., *Message Channel*) really exists, at least when the request is sent. [JMS11, p.33], [Monson-Haefel, pp.192-193]

A sender that wishes to specify a reply channel that is a queue would do so like this:

```
Queue requestQueue = // Specify the request destination
Queue replyQueue = // Specify the reply destination
Message requestMessage = // Create the request message
requestMessage.setJMSReplyTo(replyQueue);
MessageProducer requestSender =
        session.createProducer(requestQueue);
requestSender.send(requestMessage);
```

Then the receiver would send the reply message like this:

```
Queue requestQueue = // Specify the request destination
MessageConsumer requestReceiver =
        session.createConsumer(requestQueue);
Message requestMessage = requestReceiver.receive();
Message replyMessage = // Create the reply message
Destination replyQueue = requestMessage.getJMSReplyTo();
MessageProducer replySender = session.createProducer(replyQueue);
replySender.send(replyMessage);
```

Example: .NET Response-Queue Property

.NET messages also have a predefined property for *Return Addresses*, `ResponseQueue`. Its type is a `MessageQueue`, the queue that the application should send a response message to. [SysMsg], [Dickman, p.122]

Example: Web Services Request/Response

SOAP 1.2 incorporates the Request-Response Message Exchange Pattern [SOAP 1.2 Part 2], but the address to send the reply to is unspecified and therefore implied. This SOAP pattern will need to support an optional *Return Address* to truely make SOAP messages asynchronous and delink the responder from the requestor.

The emerging WS-Addressing standard helps address this issue by specifying how to identify a web service endpoint and what XML elements to use. Such an address can be used in a SOAP message to specify a *Return Address*. See the discussion of WS-Addressing in *Emerging Standards and Futures in Enterprise Integration*.

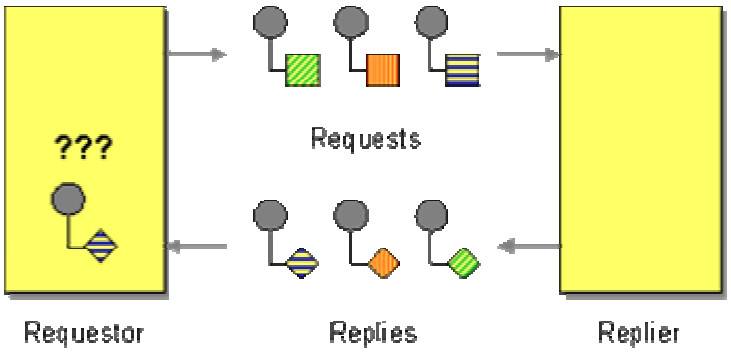
Related patterns: *Correlation Identifier*, *Emerging Standards and Futures in Enterprise Integration*, *Message Channel*, *Messaging*, *Request-Reply*

Correlation Identifier

My application is using *Messaging* to perform a *Request-Reply* and has received a reply message.

How does a requestor that has received a reply know which request this is the reply for?

When one process invokes another via *Remote Procedure Invocation*, the call is synchronous, so there is no confusion about which call produced a given result. But *Messaging* is asynchronous, so from the caller's point of view, it makes the call, then sometime later a result appears. The caller may not even remember making the request, or may have made so many requests that it no longer knows which one this is the result for. With confusion like this, when the caller finally gets the result, it may not know what to do with it, which defeats the purpose of making the call in the first place.



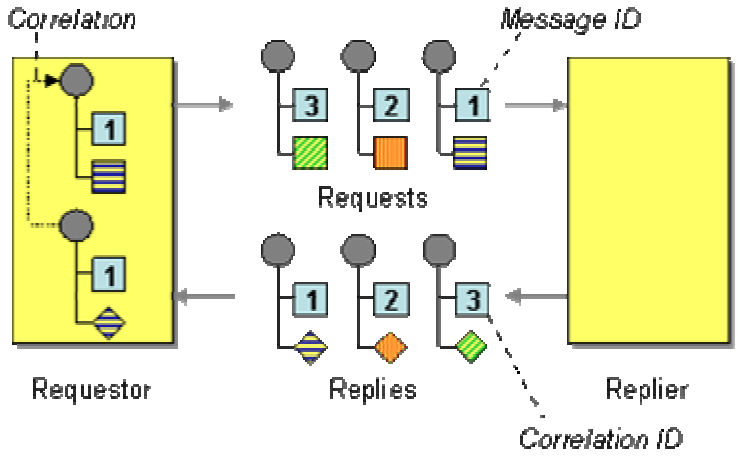
Cannot Match Reply to Request

There are a couple of approaches the caller can use to avoid this confusion. It can make just one call at a time, waiting for a reply before sending another request, so there is at most one outstanding request at any given time. This will greatly slow processing throughput, however. The call could assume that it'll receive replies in the same order it sent requests, but messaging does not guarantee what order messages are delivered in and all requests may not take the same amount of time to process, so the caller's assumption would be faulty. The caller could design its requests such that they do not need replies, but this constraint would make messaging useless for many purposes.

What the caller needs is for the reply message to have a pointer or reference to the request message, but messages do not exist in a stable memory space such that they can be referenced by

variables. However, a message could have some sort of foreign key, a unique identifier like the key for a row in a relational database table. Such a unique identifier could be used to identify the message from other messages, clients that use the message, etc.

Each reply message should contain a *Correlation Identifier*, a unique identifier that indicates which request message this reply is for.



There are six parts to *Correlation Identifier*:

1. **Requestor** — An application that performs a business task by sending a request and waiting for a reply.
2. **Replier** — Another application that receives the request, fulfills it, then sends the reply. It gets the request ID from the request and stores it as the correlation ID in the reply.
3. **Request** — A *Message* sent from the requestor to the replier containing a request ID.
4. **Reply** — A *Message* sent from the replier to the requestor containing a correlation ID.
5. **Request ID** — A token in the request that uniquely identifies the request.
6. **Correlation ID** — A token in the reply that has the same value as the request ID in the request.

This is how a *Correlation Identifier* works: When the requestor creates a request message, it assigns the request a request ID—an identifier that is different from those for all other currently outstanding requests (e.g., requests that do not yet have replies). When the replier processes the request, it saves the request ID and adds that ID to the reply as a correlation ID. When the requestor processes the reply, it uses the correlation ID to know which request the reply is for. This is called a correlation identifier because of the way the caller uses the identifier to correlate (e.g., match; show the relationship) each reply to the request that caused it.

As is often the case with messaging, the requestor and replier must agree on several details. They must agree on the name and type of the request ID property, and they must agree on the name and type of the correlation ID property. Likewise, the request and reply message formats must define those properties or allow them to be added as custom properties. For example, if the requestor stores the request ID in a first-level XML element named `request_id` and the value is an

integer, the replier has to know this so that it can find the request ID value and process it properly. The request ID value and correlation ID value are usually of the same type; if not, the requestor has to know how the replier will convert the request ID to the reply ID.

This pattern is a simpler, messaging-specific version of the Asynchronous Completion Token pattern. [POSA2] The requestor is the Initiator, the replier is the Service, the consumer in the requestor that processes the reply is the Completion Handler, and the *Correlation Identifier* that consumer uses to match the reply to the request is the Asynchronous Completion Token.

A correlation ID (and also the request ID) is usually put in the header of a message rather than the body. The ID is not part of the command or data the requestor is trying to communicate to the replier. In fact, the replier does not really use the ID at all; it just saves the ID from the request and adds it to the reply for the requestor's benefit. Since the message body is the content being transmitted between the two systems, and the ID is not part of that, the ID goes in the header.

The gist of the pattern is that the reply message contains a token (the correlation ID) that identifies the corresponding request (via its request ID). There are several different approaches for achieving this.

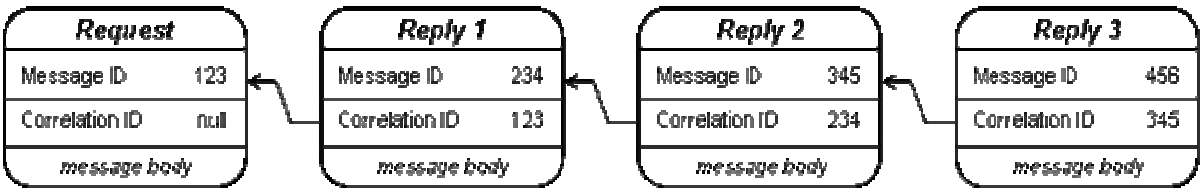
The simplest approach is for each request to contain a unique ID, such as a message ID, and for the response's correlation ID to be the request's unique ID. This relates the reply to its corresponding request. However, when the requestor is trying to process the reply, knowing the request message often isn't very interesting. What the requestor really wants is a reminder of what business task caused it to send the request in the first place, so that the requestor can complete the businees task using the data in the reply.

The business task, such as needing to execute a stock trade or ship a purchase order, probably has its own unique business object identifier (such as an order ID), so that business task's unique ID can be used as the request-reply correlation ID. Then when the requestor gets the reply and its correlation ID, it can bypass the request message and go straight to the business object whose task caused the request in the first place. In this case, rather than use the messages' built-in request message ID and reply correlation ID properties, the requestor and replier should use a custom business object ID property in the request and the reply that identifies the business object whose task this request-reply message pair is performing.

A compromise approach is for the requestor to keep a map of request ID's and business object ID's. This is especially useful when the requestor wants to keep the object ID's private, or when the requestor has no control over the replier's implementation and can only depend on the replier copying the request's message ID into the reply's correlation ID. In this case, when the requestor gets the reply, it looks up the correlation ID in the map to get the business object ID, then uses that to resume performing the business task using the reply data.

Messages have separate message ID and correlation ID properties so that request-reply message pairs can be chained. This occurs when a request causes a reply, and the reply is in turn another request that causes another reply, and so on. A message's message ID uniquely identifies the

request it represents; if the message also has a correlation ID, then the message is also a reply for another request message, as identified by the correlation ID.



Request-Reply Chaining

Chaining is only useful if an application wants to retrace the path of messages from the latest reply back to the original request. Often all the application wants to know is the original request, regardless of how many reply-steps occurred in between. In this situation, once a message has a non-null correlation ID, it is a reply and all subsequent replies caused by it should also use the same correlation ID.

Correlation Identifier is a simple version of an Asynchronous Completion Token [POSA2], where the token is simply a primitive value. Both help a caller to process the responses generated by asynchronous requests.

While a *Correlation Identifier* is used to match a reply with its request, the request may also have a *Return Address* that states what channel to put the reply on. Whereas a correlation identifier is used to match a reply message with its request, a *Message Sequence*'s identifiers are used to specify a message's position within a series of messages from the same sender.

Example: JMS Correlation-ID Property

JMS messages have a predefined property for correlation identifiers, `JMSCorrelationID`, which is typically used in conjunction with another predefined property, `JMSMessageID`. [JMS11, p.32], [Monson-Haefel, pp.194-195] A reply message's correlation ID is set from the request's message ID like this:

```
Message requestMessage = // Get the request message
Message replyMessage = // Create the reply message
String requestID = requestMessage.getJMSMessageID();
replyMessage.setJMSCorrelationID(requestID);
```

Example: .NET Correlation-Id Property

Each Message in .NET has a `CorrelationId` property, a string in an acknowledgement message that is usually set to the `Id` of the original message. `MessageQueue` also has a special peek and receive methods, `PeekByCorrelationId(string)` and `ReceiveByCorrelationId(string)`, for peeking

at and consuming the message on the queue (if any) with the specified correlation ID. (See *Selective Consumer* [SysMsg], [Dickman, pp.147-149]

Example: Web Services Request/Response

Web services standards, as of SOAP 1.1 [SOAP 1.1], do not provide very good support for asynchronous messaging, but SOAP 1.2 starts to plan for it. SOAP 1.2 incorporates the Request-Response Message Exchange Pattern [SOAP 1.2 Part 2], a basic part of asynchronous SOAP messaging. However, the request/response pattern does not mandate support for "multiple ongoing requests," so it does not define a standard *Correlation Identifier* field, even an optional one.

As a practical matter, service requestors often do require multiple outstanding requests. "Web Services Architecture Usage Scenarios" [WSAUS] discusses several different asynchronous web services scenarios. Four of them—Request/Response, Remote Procedure Call (where the transport protocol does not support [synchronous] request/response directly), Multiple Asynchronous Responses, and Asynchronous Messaging—use message-id and response-to fields in the SOAP header to correlate a response to its request. This is the request/response example:

Example: SOAP request message containing a message identifier

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
      ........
  </env:Body>
</env:Envelope>
```

Example: SOAP response message containing correlation to original request

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
```

```
    ........
  </env:Body>
</env:Envelope>
```

Like the JMS and .NET examples, in this SOAP example, the request message contains a unique message identifier, and the response message contains a response to (e.g., a correlation ID) field whose value is the message identifier of the request message.

Related patterns: *Remote Procedure Invocation*, *Message*, *Selective Consumer*, *Message Sequence*, *Messaging*, *Request-Reply*, *Return Address*

Message Sequence

My application needs to send a huge amount of data to another process, more than may fit in a single message. Or my application has made a request whose reply contains too much data for a single message.

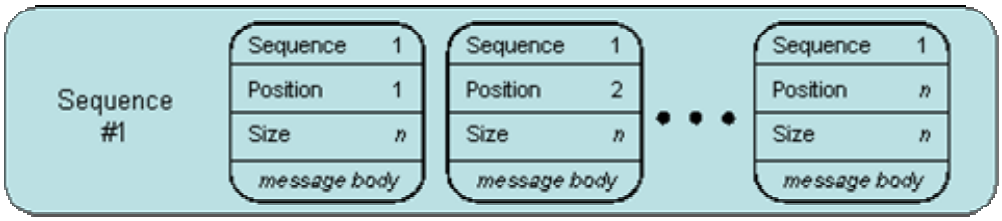
How can messaging transmit an arbitrarily large amount of data?

It's nice to think that messages can be arbitrarily large, but there are practical limits to how much data a single message can hold. Some messaging implementations place an absolute limit on how big a message can be. Other implementations allow messages to get quite big, but large messages nevertheless hurt performance. Even if the messaging implementation allows large messages, the message producer or consumer may place a limit on the amount of data it can process at once. For example, many COBOL- and mainframe-based systems will only consume or produce data in 32 Kb chunks.

So how do you get around this? One approach is to limit your application to never need more data that what the messaging layer can handle. This is an arbitrary limit, though, which can prevent your application from producing the desired functionality. If the large amount of data is the result of a request, the caller could issue multiple requests, one for each result chunk, but that assumes the caller even knows how many result chunks will be needed. The receiver could listen for data chunks until there are not anymore (but how does it know there are not anymore?), then try to figure out how to reassemble the chunks into the original, large piece of data, but that would be error-prone.

Inspiration comes from the way a mail order company sometimes ships an order in multiple boxes. If there are three boxes, the shipper will mark them as "1 of 3," "2 of 3," and "3 of 3" so that the receiver will know which ones he's received and whether he has all of them. The trick is to apply the same technique to messaging.

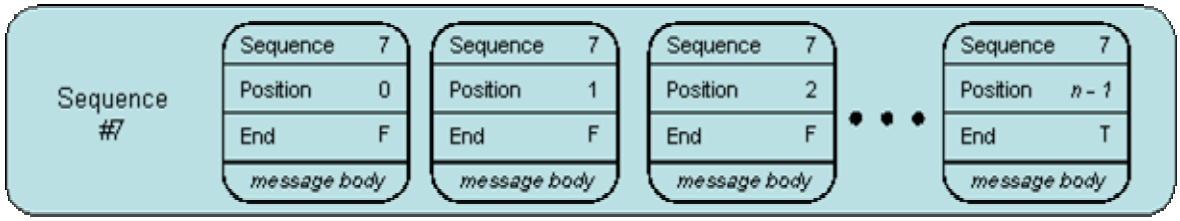
Whenever a large set of data may need to be broken into message-size chunks, send the data as a *Message Sequence* and mark each message with sequence identification fields.



The three *Message Sequence* identification fields are:

1. **Sequence identifier** – Distinguishes this cluster of messages from others.
2. **Position identifier** – Uniquely identifies and sequentially orders each message in a sequence.
3. **Size** or **End indicator** – Specifies the number of messages in the cluster, or marks the last message in the cluster (whose position identifier then specifies the size of the cluster).

The sequences are typically designed such that each message in a sequence indicates the total size of the sequence, e.g. the number of messages in that sequence. As an alternative, you can design the sequences such that each message indicates whether it is the last message in that sequence.



Message Sequence with End Indicator

Let's say a set of data needs to be sent as a cluster of three messages. The sequence identifier of the three-message cluster will be some unique ID. The position identifier for each message will be different—either 1, 2, or 3 (assuming that numbering starts from 1, not 0). If the sender knows the total number of messages from the start, the sequence size for each message is 3. If the sender does not know the total number of messages until it runs out of data to send (e.g., the sender is streaming the data), each message except the last will have a "sequence end" flag that is false; when the sender is ready to send the final message in the sequence, it will set that message's sequence end flag is true. Either way, the position identifiers and sequence size/end indicator will give the receiver enough information to reassemble the parts back into the whole, even if the parts are not received in sequential order.

If the receiver expects a *Message Sequence*, then every message sent to it should be sent as part of a sequence, even if it is only a sequence of one. Otherwise, when a single-part message is sent without the sequence identification fields, the receiver may become confused by the missing fields and may conclude that the message is invalid (see *Invalid Message Channel*).

If a receiver gets some of the messages in a sequence but never does get all of them, it should reroute the ones it did receive to the *Invalid Message Channel*.

An application may wish to use a *Transactional Client* for sending and receiving sequences. The sender can send all of the messages in a sequence using a single transaction. This way, none of the messages will be delivered until all of them have been sent. Likewise, a receiver may wish to use a single transaction to receive the messages so that it does not truly consume any of the messages until it receives all of them. If any of the messages in the sequence are missing, the receiver can choose to rollback the transaction so that the messages can be consumed later. In many messaging system implementations, if a sequence of messages are sent in one transaction, they will be received in the order they are sent, which simplifies the receiver's job of putting the data back together.

When the *Message Sequence* is the reply message in a *Request-Reply*, the sequence identifier and the *Correlation Identifier* are usually the same thing. They would be separate if the application sending the request expected multiple responses to the same request and one or more of the responses could be in multiple parts. When only one response is expected, then uniquely identifying the response and its sequence is permissible, but redundant.

Message Sequence tends not to be compatible with *Competing Consumers* nor *Message Dispatcher*. If different consumers/performers receive different messages in a sequence, none of the receivers will be able to reassemble the original data without exchanging message contents with each other. Thus a message sequence should be transmitted either via a *Message Channel* with a single consumer.

An alternative to *Message Sequence* is to use a *Claim Check*. Rather than transmitting a large document between two applications, if the applications both have access to a common database or file system, store the document and just transmit the receipt in a single message.

Example: Large Document Transfer

Imagine a sender needs to send an extremely large document to a receiver, so large that it will not fit within a single message, or is impractical to send all at once. Then break the large document into parts, each of which can be sent as a message. Each message needs to indicate its position in the sequence and indicate how many messages total to expect.

For example, the maximum size of an MSMQ message is 4 MB. [Dickman, pp.169-172] discusses how to send a multipart message sequence in MSMQ.

Example: Multi-Item Query

Consider a query that requests a list of all books by a certain author. Because this could be a very large list, the messaging design might choose to return each match as a separate message. Then each message needs to indicate the query this reply is for, the message's position in the sequence, and how many messages total to expect.

Example: Distributed Query

Consider a query that is performed in parts by multiple receivers. If the parts have some order to them, this will need to be indicated in the reply messages so that the complete reply can be assembled properly. Each receiver will need to know its position in the overall order and will need to indicate that position is the reply's message sequence.

Example: JMS and .NET

Neither JMS nor .NET have built-in properties for supporting message sequences. Therefore, messaging application must implement their own sequence fields. In JMS, an application can define its own properties in the header, so that is an option. .NET does not provide application-defined properties in the header. The fields could also be defined in the message body. Keep in mind that if a receiver of the sequence needs to filter for messages based on their sequence, such filtering is much simpler to do if the field is stored in the header rather than the body.

Example: Web Services Architecture Usage Scenarios

Web services standards currently do not provide very good support for asynchronous messaging, but the W3C has started to think about how it could. "Web Services Architecture Usage Scenarios" [WSAUS] discusses several different asynchronous web services scenarios. One of them—Multiple Asynchronous Responses—use message-id and response-to fields in the SOAP header to correlate a responses to their request, and sequence-number and total-in-sequence fields in the body to sequentially identify the responses. This is the multiple responses example:

Example: SOAP request message containing a message identifier

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

Example: First SOAP response message containing sequencing and correlation to original request

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <!-- MessageId will be unique for each response message -->
      <!-- ResponseTo will be constant for each response message in the sequence-->
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
    <s:Sequence xmlns:s="http://example.org/sequence">
      <s:SequenceNumber>1</s:SequenceNumber>
      <s:TotalInSequence>5</s:TotalInSequence>
    </s:Sequence>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

Example: Final SOAP response message containing sequencing and correlation to original request

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <!-- MessageId will be unique for each response message -->
      <!-- ResponseTo will be constant for each response message in the sequence-->
      <n:MessageId>uuid:40195729-sj20-pso3-1092-p20dj28rk104</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
    <s:Sequence xmlns:s="http://example.org/sequence">
      <s:SequenceNumber>5</s:SequenceNumber>
      <s:TotalInSequence>5</s:TotalInSequence>
    </s:Sequence>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

The message-id in the header is used as the sequence identifier in the responses. The sequence-number and total-in-sequence in each response are a position identifier and side indicator, respectively.

Related patterns: *Competing Consumers*, *Correlation Identifier*, *Invalid Message Channel*, *Message Channel*, *Message Dispatcher*, *Request-Reply*, *Claim Check*, *Transactional Client*

Message Expiration

My application is using *Messaging*. If a *Message*'s data or request is not received by a certain time, it is useless and should be ignored.

How can a sender indicate when a message should be considered stale and thus shouldn't be processed?

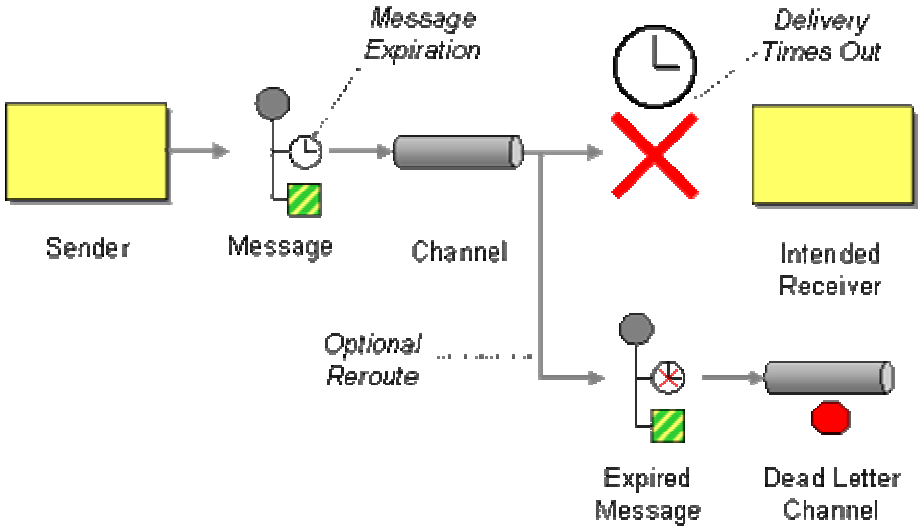
Messaging practically guarantees that the *Message* will eventually be delivered to the receiver. What it cannot guarantee is how long the delivery may take. For example, if the network connecting the sender and receiver is down for a week, then it could take a week to deliver a message. Messaging is highly reliable, even when the participants (sender, network, and receiver) are not, but messages can take a very long time to transmit in unreliable circumstances. (For more details, see *Guaranteed Delivery*.)

Often, a message's contents have a practical limit for how long they're useful. A caller issuing a stock quote request probably looses interest if it does not receive an answer within a minute or so. That means the request should not take more than a minute to transmit, but also that the answer had better transmit back very quickly. A stock quote reply more than a minute or two old is probably too old and therefore unreliable.

Once the sender sends a message and does not get a reply, it has no way to cancel or recall the message. Likewise, a receiver could check when a message was sent and reject the message if it's too old, but different senders under different circumstances may have different ideas about how long is too long, so how does the receiver know which messages to reject?

What is needed is a way for the sender to specify the message's lifetime.

Set the *Message Expiration* to specify a time limit how long the message is viable.



Once the time for which a message is viable passes, and the message still has not been consumed, then the message will expire. The messaging system's consumers will ignore an expired message; they treat the message as if it where never sent in the first place. Most messaging system implementations reroute expired messages to the *Dead Letter Channel*, while others simply discard expired messages; this may be configurable.

A *Message Expiration* is like the expiration date on a carton of milk—after that date, you're not supposed to drink the milk. If the expiration date passes while the milk is on the grocery store shelf, the grocer is supposed to pull the milk off the shelf and not sell it. If you end up with a carton on milk that expires, you're supposed to pour it out. Likewise, when a message expires, the messaging system should no longer deliver it. If a receiver nevertheless receives a message but cannot process it before the expiration, the receiver should throw away the message.

A *Message Expiration* is a timestamp (date and time) that specifies how long the message will live or when it will expire. The setting can be specified in relative or absolute terms. An absolute setting specifies a date and time when the message will expire. A relative setting specifies how long the message should live before it expires; the messaging system will use the time when the message is sent to convert the relative setting into an absolute one. The messaging system is responsible for adjusting the timestamp for receivers in different timezones from the sender, for adjustments in daylight savings times, and any other issues that can keep two different clocks from agreeing on what time it is.

The message expiration property has a related property, *sent time*, which specifies when the message was sent. A message's absolute expiration timestamp must be later than its sent timestamp (or else the message will expire immediately). To avoid this problem, senders usually specify expiration times relatively, in which case the messaging system calculates the expiration timestamp by adding the relative timeout to the sent timestamp (expiration time = sent time + time to live).

When a message expires, the messaging system may simply discard it or may move it to a *Dead Letter Channel*. A receiver that finds itself in possession of an expired message should move it to the *Invalid Message Channel*. With a *Publish-Subscribe Channel*, each subscriber gets its own copy of the message; some copies of a message may reach their subscribers successfully while other copies of the same message expire before their subscribers consume them. When using *Request-Reply*, a reply message with an expiration may not work well—if the reply expires, the sender of the request will never know whether the request was ever received in the first place. If reply expirations are used, the request sender has to be designed to handle the case where expected replies are never received.

Example: JMS Time-To-Live Parameter

Message expiration is what the JMS spec calls "message time-to-live." [JMS11, p.71], [Hapner, pp.59-60] JMS messages have a predefined property for message expiration, `JMSExpiration`, but a sender should not set it via `Message.setJMSExpiration(long)` because the JMS provider will override that setting when the message is sent. Rather, the sender should use its `MessageProducer` (`QueueSender` or `TopicPublisher`) to set the timeout for all messages it sends; the method for this setting is `MessageProducer.setTimeToLive(long)`. A sender can also set the time-to-live on an individual message using the `MessageProducer.send(Message message, int deliveryMode, int priority, long timeToLive)` method, where the forth parameter is the time-to-live in milliseconds. Time-to-live is a relative setting specifying how long after the message is sent it should expire.

Example: .NET Time-To-Be-Received and Time-To-Reach-Queue Property

A .NET Message has two properties for specifying expiration: `TimeToBeReceived` and `TimeToReachQueue`. The reach queue setting specifies how long the message has to reach its destination queue, after which the message might sit in the queue indefinitely. The be received setting specifies how long the message has to be consumed by a receiver, which limits the total time for transmitting the message to its destination queue plus the amount of time the message can spend sitting on the destination queue. `TimeToBeReceived` is equivalent to JMS's `JMSExpiration` property. Both time settings have a value of type `System.TimeSpan`, a length of time. [SysMsg], [Dickman, pp.56-58]

Related patterns: *Dead Letter Channel*, *Guaranteed Delivery*, *Invalid Message Channel*, *Message*, *Messaging*, *Publish-Subscribe Channel*, *Request-Reply*

Format Indicator

Several applications are communicating via *Message*s that follow an agreed upon data format, perhaps an enterprise-wide *Canonical Data Model*. However, that format may need to change over time.

How can a message's data format be designed to allow for possible future changes?

Even when you design a data format that works for all participating applications, future requirements may change. New applications may be added that have new format requirements, new data may need to be added to the messages, or developers may find better ways to structure the same data. Whatever the case, designing a single enterprise data model is difficult enough; designing one that will never need to change in the future is darn near impossible.

When an enterprise's data format changes, there would be no problem if all of the applications change with it. If every application stopped using the old format and started using the new format, and all did so at exactly the same time, then conversion would be simple. The problem is that some applications will be converted before others, while some less-used applications may never be converted at all. Even if all applications could be converted at the same time, all messages would have to be consumed so that all channels are empty before the conversion could occur.

Realistically, applications are going to have to be able to support the old format and the new format simultaneously. To do this, applications will need to be able to tell which messages follow the old format and which use the new.

One solution might be to use a separate set of channels for the messages with the new format. That, however, would lead to a huge number of channels, duplication of design, and configuration complexity as each application has to be configured for an ever-expanding assortment of channels.

A better solution is for the messages with the new format to share the same channels the old-format messages have already been using. This means that receivers will need a way to differentiate messages from the same channel that have different formats. The message must specify what format it is using. Each message needs a simple way to indicate its format.

Design a data format that includes a *Format Indicator*, so that the message specifies what format it is using.

The format indicator enables the sender to tell the receiver the format of the message. This way, a receiver expecting several possible formats knows which one a message is using and therefore how to interpret the message's contents.

There are three main alternatives for implementing a format indicator:

1. **Version Number** – A number or string that that uniquely identifies the format. Both the sender and receiver must agree on which format is designated by a particular indicator.
2. **Foreign Key** – A unique ID—such as a filename, a database row key, a home primary key, or an Internet URL—that specifies a format document. The sender and receiver must agree on the mapping of keys to documents, and the format of the schema document.
3. **Format Document** – A schema that describes the data format. The schema document does not have to be retrieved via a foreign key or inferred from a version number, it is embedded in the message. The sender and the receiver must agree on the format of the schema.

A version number or foreign key can be stored in a header field that the senders and receivers agree upon. Receivers that are not interested in the format version can ignore the field. A format document may be too long or complex to store in a header field, in which case the message body will need to have a format that contains two parts, the schema and the data.

Example: XML

XML documents have examples of all three approaches. One example is an XML declaration, like this:

```
<?xml version="1.0"?>
```

Here, `1.0` is a version number that indicates the document's conformance to that version of the XML specification. Another example is the document type declaration, which can take two forms. It can be an external ID containing a system identifier like this:

```
<!DOCTYPE greeting SYSTEM "hello.dtd">
```

The system identifier, `hello.dtd`, is a foreign key that indicates the file containing the DTD document that describes this XML document's format. The declaration can also be included locally, like this:

```
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
```

The markup declaration, `[<!ELEMENT greeting (#PCDATA)>]`, is a format document, an embedded schema document that describes the XML's format. [XML 1.0]

Related patterns: *Canonical Data Model*, *Message*

7. Message Routing

Introduction

In the Chapter 2, we discussed how a *Message Router* can be used to decouple a message source from the ultimate destination of the message. This chapter elaborates on specific types of *Message Router*s to explain how to provide routing and brokering ability to an integration solution. Most patterns are refinements of the *Message Router* pattern while others combine multiple *Message Router*s to solve more complex problems. Therefore, we can categorize the Message Routing patterns into the following groups:

- **Simple Routers** are variants of the *Message Router* and route messages from one inbound channel to one or more outbound channels.
- **Composed Routers** combine multiple simple routers to create more complex message flows.
- **Architectural Patterns** describe architectural styles based on *Message Router*s.

Simple Routers

The *Content-Based Router* inspects the content of a message and routes it to another channel based on the content of the message. Using such a router enables the message producer to send messages to a single channel and leave it to the *Content-Based Router* to inspect messages and route them to the proper destination. This alleviates the sending application from this task and avoids coupling the message producer to specific destination channels.

A *Message Filter* is a special form of a *Content-Based Router*. It examines the message content and passes the message to another channel if the message content matches certain criteria. Otherwise, it discards the message. *Message Filter*s can be used with *Publish-Subscribe Channel* to route a message to all possible recipients and allow the recipients to filter out irrelevant messages. A *Message Filter* performs a function that is very similar to that of a *Selective Consumer* with the key difference being that a *Message Filter* is part of the messaging system, routing qualifying messages to another channel, whereas a *Selective Consumer* is built into a *Message Endpoint*.

A *Content-Based Router* and a *Message Filter* can actually solve a similar problem. A *Content-Based Router* routes a message to the correct destination based on the criteria encoded in the *Content-Based Router*. Equivalent behavior can be achieved by using a *Publish-Subscribe Channel* and an array of *Message Filter*s, one for each potential recipient. Each *Message Filter* eliminates the messages that to not match the criteria for the specific destination. The *Content-Based Router* routes predictively to a single channel and therefore has total control, but is also dependent on the list of all possible destination channels. The *Message Filter* array filters reactively, spreading the routing logic across many *Message Filter*s but avoiding a single component that is dependent

on all possible destinations. The trade-off between these solutions is described in more detail in the *Message Filter* pattern.

A basic *Message Router* uses fixed rules to determine the destination of an incoming message. Where we need more flexibility, a *Dynamic Router* can be very useful. This router allows the routing logic to be modified by sending control messages to a designated control port. The dynamic nature of the *Dynamic Router* can be combined with most forms of the *Message Router*.

Chapter 3 introduced the concept of a *Point-to-Point Channel* and a *Publish-Subscribe Channel*. Sometimes, you need to send a message to more than one recipient, but want to maintain control over the recipients. The *Recipient List* allows you do just that. In essence, a *Recipient List* is a *Content-Based Router* that can route a single message to more than one destination channel.

Some messages contain lists of individual items. How do you process these items individually? Use a *Splitter* to split the large message into individual messages. Each message can then be routed further and processed individually.

However, you may need to recombine the messages that the *Splitter* created back into a single message. This is one of the functions an *Aggregator* performs. An *Aggregator* can receive a stream of messages, identify related messages and combine them into a single message. Unlike the other routing patterns, the *Aggregator* is a stateful *Message Router* because it has to store messages internally until specific conditions are fulfilled. This mean that an *Aggregator* can consume a number of messages before it publishes a message.

Because we use messaging to connect applications or components running on multiple computers, multiple messages can be processed in parallel. For example, more than one process may consume messages off a single channel. One of these processes may execute faster than another, causing messages to be processed out of order. However, some components depend on the correct sequence of individual messages, for example ledger-based systems. The *Resequencer* puts out-of-sequence messages back into sequence. The *Resequencer* is also a stateful *Message Router* because it may need to store a number of messages internally until the message arrives that completes the sequence. Unlike the *Aggregator*, though, the *Resequencer* ultimately publishes the same number of messages it consumed.

The following table summarizes the properties of the *Message Router* variants (we did not include the *Dynamic Router* as a separate alternative because any router can be implemented as a dynamic variant):

Pattern	Number of Msgs Consumed	Number of Msgs Published	Stateful?	Comment
Content-Based Router	1	1	No (mostly)	
Message Filter	1	0 or 1	No (mostly)	

Recipient List	1	multiple (incl. 0)	No	
Splitter	1	multiple	No	
Aggregator	multiple	1	Yes	
Resequencer	multiple	multiple	Yes	Publishes same number it consumes.

Composed Routers

A key advantage of the *Pipes and Filters* architecture is the fact that we can compose multiple filters into a larger solution. *Composed Message Processor* or an *Scatter-Gather* combine multiple *Message Router* variants to create more comprehensive solutions. Both patterns allow us to retrieve information from multiple sources and recombine it into a single message. While the *Composed Message Processor* maintains control over the possible sources the *Scatter-Gather* uses a *Publish-Subscribe Channel* so that any interested component can participate in the process.

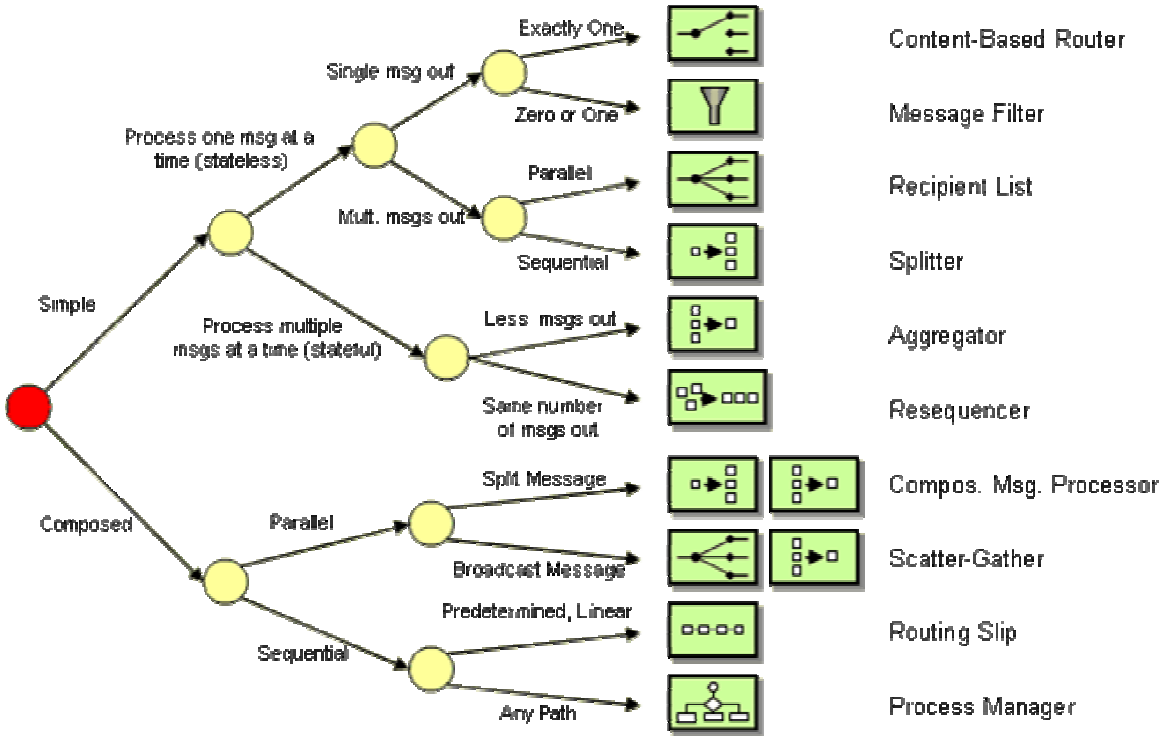
Both the *Composed Message Processor* and the *Scatter-Gather* route a single message to a number of participants concurrently and reassemble the replies into a single message. We can say that these patterns manage the *parallel routing* of a message. Two more patterns manage the *sequential routing* of a message, i.e. routing a message through a sequence of individual steps. If we want to control the path of a message from a central point we can use a *Routing Slip* to specify the path the message should take. This pattern works just like the routing slip attached to office documents to pass them sequentially by a number of recipients. Alternatively, we can use a *Process Manager* which gives us more flexibility but requires the message to return to a central component after each function.

Architectural Patterns

*Message Router*s enable us to architect an integration solution using a central *Message Broker*. As opposed to the different message routing design patterns, this pattern describes a *hub-and-spoke* architectural style.

The Right Router for the Right Purpose

This chapter contains 10 patterns. How can we make it easy to find the right pattern for the right purpose? The following decision chart helps you find the right pattern for the right purpose by matter of simple yes/no decisions. For example, if you are looking for a simple routing pattern that consumes one message at a time but publishes multiple messages in sequential order, you should use a *Splitter*. The diagram also helps illustrate how closely the individual patterns are related. For example, a *Routing Slip* and a *Process Manager* solve similar problems while a *Message Filter* does something rather different.



Content-Based Router

Assume that we are building an order processing system. When an incoming order is received, we first validate the order and then verify that the ordered item is available in the warehouse. This function is performed by the inventory system. This sequence of processing steps is a perfect candidate for the *Pipes and Filters* style. We create two filters, one for the validation step and one for the inventory system, and route the incoming messages through both filters. However, in many enterprise integration scenarios more than one inventory system exists with each system being able to handle only specific items.

How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?

Integration solutions connect existing applications so that they work together. Because many of these applications were developed without integration in mind, integration solutions rarely find an ideal scenario where a business function is well encapsulated inside a single system. For example, acquisitions or business partnerships often result in multiple systems performing the same business function. Also, many businesses that act as aggregators or resellers typically interface with multiple systems that perform the same functions (e.g. check inventory, place order etc). To make matters more complicated, these systems may be operated within the company or may be under the control of business partners or affiliates. For example, large e-tailers like Amazon allow you to order anything from books to chain saws and clothing. Depending on the type of item, the order may be processed by a different "behind-the-scenes" merchant's order processing systems.

Let us assume that the company is selling widgets and gadgets and has two inventory systems, one for widgets and one for gadgets. Let's also assume that each item is identified by a unique item number. When the company receives an order, it needs to decide which inventory system to pass the order to based on the type of item ordered. We could create separate channels for incoming orders based on the type of item ordered. However, this would require the customers to know our internal system architecture when in fact they may not even be aware that we distinguish between widgets and gadgets. Therefore, we should hide the fact that the implementation of the business function is spread across multiple systems from the remainder of the integration solution, including customers. Therefore, we need to expect messages for different items arriving on the same channel.

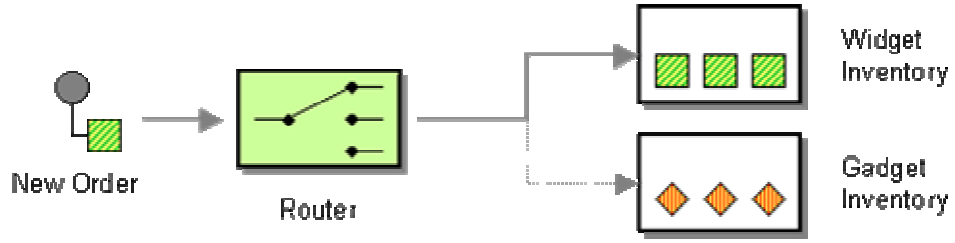
We could forward the order to all inventory systems (using a *Publish-Subscribe Channel*), and let each system decide whether it can handle the order. This approach makes the addition of new inventory systems easy because we do not have to change any of the existing components when a new inventory system comes on-line. However, this approach assumes distributed coordination across multiple systems. What happens if the order cannot be processed by any system? Or if more than one system can process the order? Will the customer receive duplicate shipments? Also, in many cases an inventory system will treat an order for an item that it cannot handle as an error. If this is the case, each order would cause errors in all inventory systems but one. It would be hard to distinguish these errors from 'real' errors such as an invalid order.

An alternative approach would be to use the item number as a channel address. Each item would have its dedicated channel and the customers could simply put publish the order to the channel associated with the item's number without having to know about any internal distinctions between widgets and gadgets. The inventory systems could listen on all the channels for those items that it can process. This approach leverages the channel addressability to route messages to the correct inventory system. However, a large number of items could quickly lead to an explosion of the number of channels, burdening the system with run-time and management overhead. Creating new channels for each item that is offered would quickly result in chaos.

We should also try to minimize message traffic. For example we could route the order message through one inventory system after the other. The first system that can accept the order consumes the message and processes the order. If it cannot process the order it passes the order message to the next system. This approach eliminates the danger of orders being accepted by multiple systems simultaneously. Also, we know that the order was not processed by any system if the last system passes it back. The solution does require, however, that the systems know enough about each other in order to pass the message from one system to the next. This approach is similar to the *Chain of Responsibility* pattern described in [GoF]. However, in the world of message-based integration passing messages through a chain of systems could mean significant overhead. Also, this approach would require collaboration of the individual systems, which may not be feasible if some systems are maintained by external business partners and are therefore not under our control.

In summary, we need a solution that encapsulates the fact that the business function is split across systems, is efficient in its usage of message channels and message traffic, and ensures that the order is handled by exactly one inventory system.

Use a *Content-Based Router* to route each message to the correct recipient based on message content.



The *Content-Based Router* examines the message content and routes the message onto a different channel based on data contained in the message. The routing can be based on a number of criteria such as existence of fields, specific field values etc. When implementing a *Content-Based Router*, special caution should be taken to make the routing function easy to maintain as the router can become a point of frequent maintenance. In more sophisticated integration scenarios, the *Content-Based Router* can take on the form of a configurable rules engine that computes the destination channel based on a set of configurable rules.

Reducing Dependencies

Content-Based Router is a frequently used form of the more generic *Message Router*. It uses *predictive routing*, i.e. it incorporates knowledge of the capabilities of all other systems. This makes for efficient routing because each outgoing message is sent directly to the correct system. The downside is that the *Content-Based Router* has to have knowledge of all possible recipients and their capabilities. As recipients are added, removed or changed, the *Content-Based Router* has to be changed every time. This can become a maintenance nightmare.

We can avoid the dependency of the *Content-Based Router* on the individual recipients if the recipients assume more control over the routing process. These options can be summarized as *reactive filtering* because they allow each participant to filter relevant messages as they come by. The distribution of routing control eliminates the need for a *Content-Based Router* but the solution is generally less efficient. These solutions and associated trade-offs are described in more detail in the *Message Filter* and *Routing Slip*.

The *Dynamic Router* describes a compromise between the *Content-Based Router* and the reactive filtering approach by having each recipient inform the *Content-Based Router* of its capabilities. The *Content-Based Router* maintains a list of each recipient's capabilities and routes incoming messages accordingly. The price we pay for this flexibility is the complexity of the solution and the difficulty of debugging such a system when compared to a simple *Content-Based Router*.

Example: Content-Based Router Router with C# and MSMQ

This code example demonstrates a very simple *Content-Based Router* that routes messages based on the first character in the message body. If the body text starts with 'W', the router routes the message to the `widgetQueue`, if it starts with 'G', it goes to the `gadgetQueue`. If it is neither, the router sends it to the `dunnoQueue`. This queue is actually an example of a *Invalid Message Channel*. This router is *stateless*, i.e. it does not "remember" any previous messages when making the routing decision.

```
class ContentBasedRouter
{
   protected MessageQueue inQueue;
   protected MessageQueue widgetQueue;
   protected MessageQueue gadgetQueue;
   protected MessageQueue dunnoQueue;

   public ContentBasedRouter(MessageQueue inQueue, MessageQueue widgetQueue,
MessageQueue gadgetQueue, MessageQueue dunnoQueue)
   {
      this.inQueue = inQueue;
      this.widgetQueue = widgetQueue;
      this.gadgetQueue = gadgetQueue;
      this.dunnoQueue = dunnoQueue;

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

   private void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
      Message message = mq.EndReceive(asyncResult.AsyncResult);

      if (IsWidgetMessage(message))
         widgetQueue.Send(message);
      else if (IsGadgetMessage(message))
         gadgetQueue.Send(message);
      else
         dunnoQueue.Send(message);
      mq.BeginReceive();
   }
```

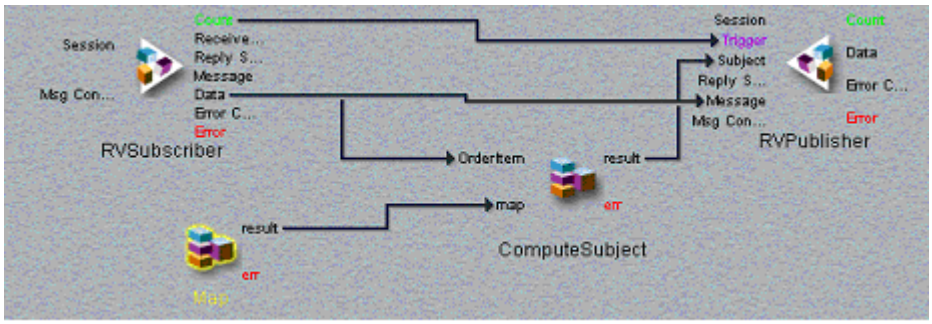
```
    protected bool IsWidgetMessage (Message message)
    {
        String text = (String)message.Body;
        return (text.StartsWith("W"));
    }

    protected bool IsGadgetMessage (Message message)
    {
        String text = (String)message.Body;
        return (text.StartsWith("G"));
    }
}
```

The example uses an event-driven message consumer by registering 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`. The message queue `Formatter` property tells the framework what type of message we expect. In our example, we only deal with simple string messages. `OnMessage` figures out where to route the message and tells .NET that it is ready for the next message by calling the `BeginReceive` method on the queue. In order to keep the code to a minimum, this simple router is not transactional, i.e. if the router crashes after it consumed a message from the input channel and before it published it to the output channel, we would lose a message. Later chapters will explain how to make endpoints transactional (see *Transactional Client*).

Example: TIBCO MessageBroker

Message routing is such a common need that most EAI tool suites provide built-in tools to simplify the construction of the routing logic. For example, in the C# example we had to code the logic to read a message off the incoming queue, deserialize it, analyze it and republish it to the correct outgoing channel. In many EAI tools this type of logic can be implemented with simple drag-and-drop operations instead of writing code. The only code to write is the actual decision logic for the *Content-Based Router*.

One such EAI tool that implements message routing is the TIBCO ActiveEnterprise suite. The suite includes TIB/MessageBroker which is designed to create simple message flows that include transformation and routing. The widget router that routes incoming messages based on the first letter of the item number looks like this when implemented in TIB/MessageBroker:



We can read the message flow from left to right. The component on the left (represented by a triangle pointing to the right) is the subscriber component that consumes messages off the channel `router.in`. The channel name is specified in a properties box not shown in this picture. The message content is directed to the message publisher (represented by the triangle on the right side of the screen) The direct line from the `Data` output of the subscriber to the `Message` input of the publisher represents the fact that a *Content-Based Router* does not modify the message. In order to determine the correct output channel, the function `ComputeSubject` (in the middle) analyzes the message content. The function uses a so-called *dictionary* (labeled as 'Map' in the picture) as a translation table between message contents and the destination channel name. The dictionary is configured with the following values:

Item Code	Channel Name
G	gadget
W	widget

The `ComputeSubject` functions uses the first letter of the incoming message's order item number to look up the destination channel from the dictionary. To form the complete name of the output channel, it appends the dictionary result to the string "router.out", to form a channel name like "router.out.widget". The result of this computation is passed to the publisher component on the right to be used as the name of the channel. As a result, any order item whose item number starts with a 'G' is routed to the channel `router.out.gadget`, whereas any item whose item number starts with a 'W' is routed to the channel `router.out.widget`.

The TIBCO implementation of the `ComputeSubject` function looks like this:

```
concat("router.out.",DGet(map,Upper(Left(OrderItem.ItemNumber,1))))
```

The function extracts the first letter of the order number (using the `Left` function) and converts it to uppercase (using the `Upper` function). The function uses the result as the key to the dictionary to retrieve the name of the outgoing channel (using the `DGet` function).

This example demonstrates the strengths of commercial EAI tools. Instead of a few dozen lines of code we only need to code a single function to implement the widget router. Plus, we get features like transactionality, thread management, systems management etc. for free. But this example also highlights the difficulties of presenting a solution created with UI tools. We had to relegate

to screen shots to describe the solution. Many important settings are hidden in property fields that are not shown on the screen. This can make it difficult to document a solution built using UI tools.

Related patterns: *Dynamic Router*, *Message Filter*, *Invalid Message Channel*, *Message Router*, *Pipes and Filters*, *Publish-Subscribe Channel*, *Routing Slip*, *Transactional Client*

Message Filter

Continuing with the order processing example, let's assume that company management publishes price changes and promotions to large customers. Whenever a price for an item changes, we send a message notifying the customer. We do the same if we are running a special promotion, e.g. all widgets are 10% off in the month of November. Some customers may be interested in receiving price updates or promotions only related to specific items. If I purchase primarily gadgets, I may not be interested in knowing whether widgets are on sale or not.

How can a component avoid receiving uninteresting messages?

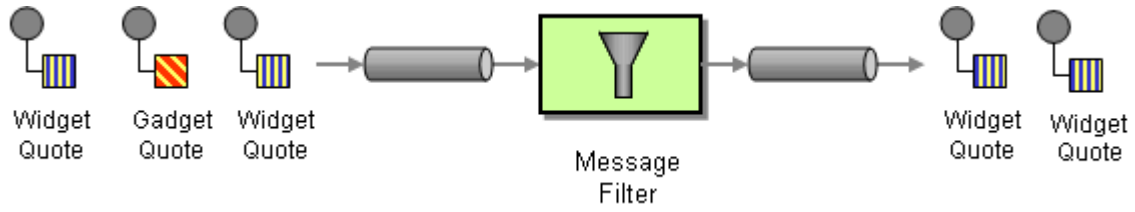
The most basic way for a component to receive only relevant messages is to subscribe only to those channels that carry relevant messages. This options leverages the inherent routing abilities of *Publish-Subscribe Channel*s. A component receives only those messages that travel through channels to which the component subscribes. For example, we could create one channel for widget updates and another one for gadget updates. Customers would then be free to 'subscribe' to one or the other channel or both. This has the advantage that new subscribers can join in without requiring any changes to the system. However, subscription to *Publish-Subscribe Channel* is generally limited to a simple binary condition: if a component subscribes to a channel, it receives all messages on that channel. The only way to achieve finer granularity it to create more channels. If we are dealing with a combination of multiple parameters, the number of channels can quickly explode. For example, if we want to allow consumers to receive all messages that announce all price cuts of widgets or gadgets by more than 5%, 10% or 15%, we already need 6 (2 item types multiplied by 3 threshold values) channels. This approach would ultimately become difficult to manage and will consume significant resources due to the large number of allocated channels. So we need to look for a solution that allows for more flexibility than channel subscription.

We also need a solution than can accommodate frequent change. For example, me could modify a *Content-Based Router* to route the message to more than one destination (a concept described in the *Recipient List*). This predictive router sends only relevant messages to each recipient so that the recipient does not have to take any extra steps. However, now we burden the message originator with maintaining the preferences for each and every subscriber. If the list of recipients or their preferences change quickly this solution would prove to be a maintenance nightmare.

We could simply broadcast the changes to all components and expect each component to filter out the undesirable messages. However, this approach assumes that we have control over the

actual component. In many integration scenarios this is not the case because we deal with packaged applications, legacy applications or applications that not under the control of our organization. Also, incorporating filtering logic inside the component makes the component dependent on a specific type of message. For example, if a customer uses a generic price watch component he or she may want to use it for both widgets and gadgets, but with different criteria.

Use a special kind of Message Router, a *Message Filter*, to eliminate undesired messages from a channel based on a set of criteria.



The *Message Filter* has only a single output channel. If the message content matches the criteria specified by the *Message Filter*, the message is routed to the output channel. If the message content does not match the criteria, the message is discarded.

In our example we would define a single *Publish-Subscribe Channel* that each customer is free to listen on. The customer can then use a *Message Filter* to eliminate messages based on criteria of his or her choosing, such as the type of item or the magnitude of the price change.

The *Message Filter* can be portrayed as a special case of a *Content-Based Router* that routes the message either to the output channel or the *null channel*, a channel that discards any message published to it. Such a channel would be similar to /dev/null present in many operating systems or the Null Object .

Stateless vs. Stateful Message Filters

The widget and gadget example described a stateless *Message Filter*, i.e., the *Message Filter* inspects a single message and decides whether to pass it on or not based solely on information contained in that message. Therefore, the *Message Filter* does not need to maintain state across messages and is considered stateless. Stateless components have the advantage that they allow us run multiple instances of the component in parallel to speed up processing. However, a *Message Filter* does not have to be stateless. For example, there are situations where the *Message Filter* needs to keep track of the message history. A common example is the use of a *Message Filter* to eliminate duplicate messages. Assuming that each message has a unique message identifier, the *Message Filter* would store the identifiers of past messages to that it can recognize a duplicate message by comparing each message's identifier with the list of stored identifiers.

Filtering Functions Built Into Messaging Systems

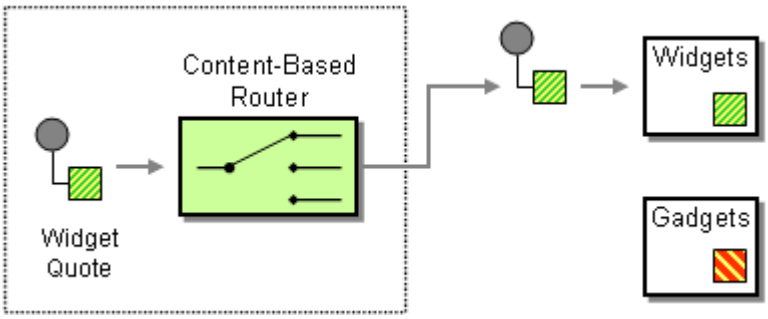
Some messaging systems incorporate aspects of a *Message Filter* inside the messaging infrastructure. For example, some publish-subscribe systems allow you to define a hierarchical structure for *Publish-Subscribe Channel*s (many publish-subscribe systems including most JMS implementations allow this). For example, one can publish promotions to the channel 'wgco.update.promotion.widget'. A subscriber can then use wildcards to subscribe to a specific subset of messages, e.g. if a subscriber listens to the topic 'wgco.update.*.widget' he would receive all updates (promotions and price changes) related to widgets. Another subscriber may listen to 'wgco.update.promotion.*', which would deliver all promotions related to widgets and gadgets, but no price changes. The channel hierarchy lets us refine the semantics of a channel by appending qualifying parameters, so that instead of a customer subscribing to all updates, customers can filter messages by specifying additional criteria as part of the channel name. However, the flexibility provided by the hierarchical channel naming is still limited when compared to a *Message Filter*. For example, a *Message Filter* could decide to only pass on 'price change' message only if the price changed by more than 11.5%, something that would be hard to express by means of channel names.

Other messaging systems provide API support for *Selective Consumer*s inside the receiving application. Message Selectors are expressions that evaluate header or property elements inside an incoming message before the application gets to see the message. If the condition does not evaluate to true the message is ignored and not passed on to the application logic. A message selector acts as a *Message Filter* that is built into the application. While the use of a message selector still requires you to modify the application (something that is often not possible in EAI), the execution of the selection rules is built into the messaging infrastructure. One important difference between a *Message Filter* and a *Selective Consumer* is that a consumer using a *Selective Consumer* does not consume messages that do not match the specified criteria. On the other hand, a *Message Filter* removes all messages from the input channel, publishing only those to the output channel that match the specified criteria.

One advantage of registering the filter expression with the messaging infrastructure is the fact that the infrastructure is able make smart internal routing decisions based on the filter criteria. Let's assume that the message receiver sits on a different network segment from the message originator (or even across the Internet). It would be rather wasteful to route the message all the way to the *Message Filter* just to find out that we want to discard the message. On the other hand, we want to use a *Message Filter* mechanism so that the recipients have control over the message routing instead of a central *Message Router*. If the *Message Filter* is part of the API that the messaging infrastructure provides to the message subscriber, the infrastructure is free to propagate the filter expression closer to the source. This will maintain the original intent of keeping control with the message subscriber, but allows the messaging infrastructure to avoid unnecessary network traffic. This behavior resembles that of a dynamic *Recipient List*.

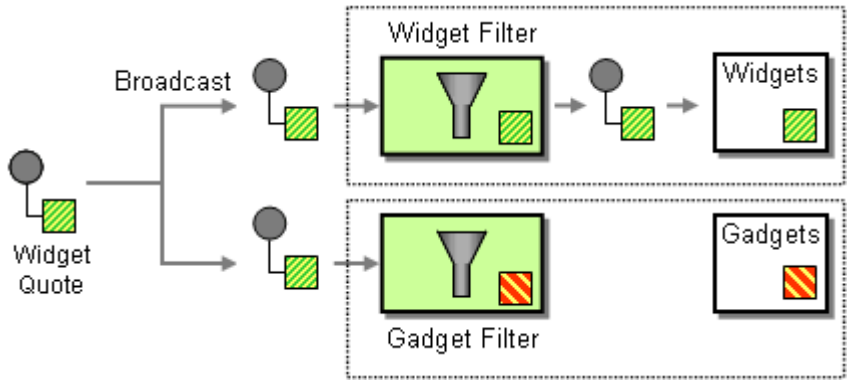
Using Message Filters to Implement Routing Fucntionality

We can use use a broadcast channel that routes a message to a set of *Message Filter*s who eliminate unwanted messages to implement functionality equivalent to that of a *Content-Based Router*. The following diagrams illustrate the two options:



Option 1: Using a Content-Based Router

In this simple example, we have 2 receivers: receiver `Gadget` is only interested in gadget messages while receiver `widget` is only interested in widget messages. The *Content-Based Router* evaluates each message's content and routes it predictively to the appropriate receiver.



Option 2: Using a broadcast channel and a set of Message Filters

The second option broadcasts the message to a *Publish-Subscribe Channel*. Each recipient is equipped with a *Message Filter* to eliminate unwanted messages. For example, the `Widget` receiver employs a widget filter that lets only widget messages pass.

The following table characterizes some of the differences between the solutions:

Content-Based Router	**Pub-Sub Channel with *Message Filter***
Exactly one consumer receives each message.	More than one consumer can consume a message.
Central control and maintenance -- predictive routing.	Distributed control and maintenance -- reactive filtering.
Router needs to know about participants. Router may need to be updated if participants are added or removed.	No knowledge of participants required. Adding or removing participants is easy.

Often used for business transactions, e.g. orders.	Often used for event notifications / informational messages.
Generally more efficient with queue-based channels.	Generally more efficient with publish-subscribe channels.

How do we decide between the two options? In some cases, the decision is driven by the required functionality, e.g. if we need the ability for multiple recipients to process the same message, we need to use a Pub-Sub Channel with *Message Filter*s. In most cases though, we decide by which party has control over (and needs to maintain) the routing decision. Do we want to keep central control or farm it out to the recipients? If message contain sensitive data that is only to be seen by certain recipients we need to use a *Content-Based Router* -- we would not want to trust the other recipients to filter out messages. For example, let's assume we offer special discounts to out premium customers we would not send those to our non-premium customers and expect them to ignore these special offers.

Network traffic considerations can drive the decision as well. If we have an efficient way to broadcast information (e.g. using IP multicast on an internal network), using filters can be very efficient and avoids the potential bottleneck of a single router. However, if this information is routed over the Internet, we are limited to point-to-point connections. In this case a router is much more efficient as it avoids sending individual messages to all participants. If we want to pass control to the recipients but need to use a router for reasons of network efficiency we can employ a *dynamic* *Recipient List*. This *Recipient List* allows recipients to express their preferences and stores them in a database or a rule base. When a message arrives the *Recipient List* forwards the message to all interested recipients whose criteria match the message.

Related patterns: *Content-Based Router*, *Message Router*, *Selective Consumer*, *Publish-Subscribe Channel*, *Recipient List*

Dynamic Router

You are using a *Message Router* to route messages between multiple destinations.

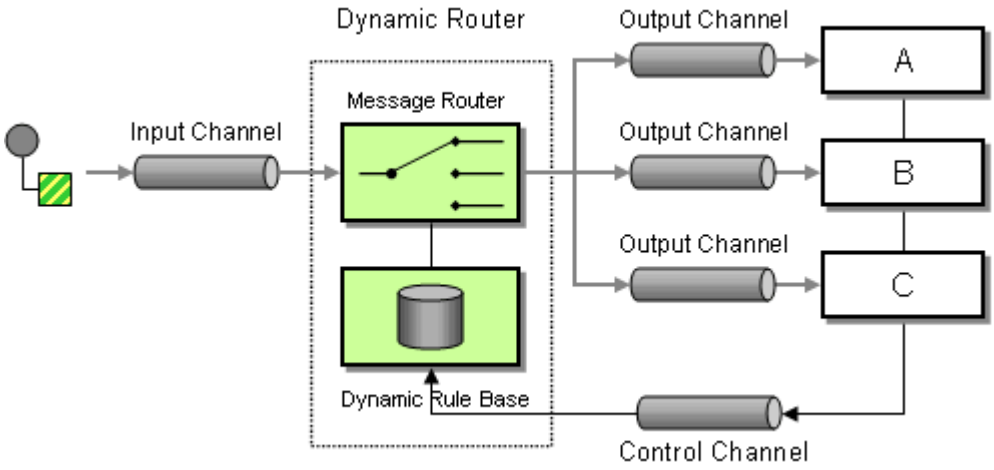
How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?

A *Message Router* is very efficient because it can route a message directly to the correct destination. Other solutions to message routing, especially reactive filtering solutions (see *Message Filter* and *Routing Slip*) are less efficient because they use a trial-and error approach: they route each message to the first possible destination. If that destination is the correct one, it accepts the message, otherwise the message is passed to the second possible destination and so on.

Distributed routing solutions also suffer from the risk that there are multiple recipients of a message or none at all. Both situations can go undetected unless we use a central routing element.

In order to achieve this accuracy, the *Message Router* has to incorporate knowledge about each destination and the rules for routing messages to the destination. This can turn the *Message Router* into a maintenance burden if the list of possible destinations changes frequently.

Use a *Dynamic Router*, a Router that can self-configure based on special configuration messages from participating destinations.



Besides the usual input and output channels the *Dynamic Router* uses an additional *control channel*. During system start-up, each potential recipient sends a special message to the *Dynamic Router* on this control channel, announcing its presence and listing the conditions under which it can handle a message. The *Dynamic Router* stores the 'preferences' for each participant in a rule base. When a message arrives, the *Dynamic Router* evaluates all rules and routes the message to the recipient whose rules are fulfilled. This allows for efficient, predictive routing without the maintenance dependency of the *Dynamic Router* on each potential recipient.

In the most basic scenario each participant announces its existence and routing preferences to the *Dynamic Router* on start-up time. This requires each participant to be aware of the control queue used by the *Dynamic Router*. It also requires the *Dynamic Router* to store the rules in a persistent way. Otherwise, if the *Dynamic Router* fails and has to restart it would not be able to recover the routing rules. Alternatively, the *Dynamic Router* could send a broadcast message to all possible participants to trigger them to reply with the control message. This configuration is more robust but requires the use of an additional *Publish-Subscribe Channel*.

It might make sense to enhance the control channel to allow participants to send both 'subscribe' and 'unsubscribe' messages to the *Dynamic Router*. This would allow recipients to add or remove themselves from the routing scheme during runtime.

Because the recipients are independent from each other, the *Dynamic Router* has to deal rules conflicts, i.e. multiple recipients announcing interest in the same type of message. The *Dynamic Router* can employ a number of different strategies to resolve such conflicts:

- Ignore control messages that conflict with existing messages. This option assures that the routing rules are free of conflict. However, the state of the routing table may depend on the sequence in which the potential recipients start up. If all recipients start up at the same time, this may lead to unpredictable behavior because all recipients would announce their preferences at the same time to the control queue.
- Send the message to the first recipient whose criteria match. This option allows the routing table to contain conflicts, but resolves them as messages come in.
- Send the message to all recipients whose criteria match. This option is tolerant of conflicts but turns the *Dynamic Router* into a *Recipient List*. generally, the behavior of a *Content-Based Router* implies that is publishes one output message for each input message. This strategy violates that rule.

The main liability of the *Dynamic Router* is the complexity of the solution and the difficulty of debugging a dynamically configured system.

A *Dynamic Router* is another example where message-based middleware performs similar functions to lower level IP networking. A *Dynamic Router* works very similar to the dynamic routing tables used in IP routing to route IP packets between networks. The protocol used by the recipients to configure the *Dynamic Router* is analogous to the IP Routing Information Protocol (RIP -- for more information see [Stevens]).

A common use of the *Dynamic Router* is dynamic service discovery in service-oriented architectures. If a client application wants to access a service it sends a message containing the name of the service to the *Dynamic Router*. The *Dynamic Router* maintains a service directory, a list of all services with their name and the channel they listen on. The *Dynamic Router* matches the name of the requested service to the service directory and routes the message to the correct channel. This setup allows services to be provided by more than one provider. The client application can continue to send command messages to a single channel without having to worry about the nature or location of the specified service provider.

[POSA] describes the Client-Dispatcher-Server pattern as a way for a client to request a specific service without knowing the physical location of the service provider. The dispatcher uses a list of registered services to establish a connection between the client and the physical server implementing the requested service. The *Dynamic Router* is different from the Dispatcher in that it can be more intelligent than a simple table lookup.

Example: Dynamic Router using C# and MSMQ

This example builds on the example presented in the *Content-Based Router* and enhances it to act as a *Dynamic Router*. The new component listens on two channels, the `inQueue` and the `controlQueue`. The control queue can receive messages of the format "X:QueueName", causing the *Dynamic Router* to route all messages whose body text begins with the letter `X` to the queue `QueueName`.

```
class DynamicRouter
{
   protected MessageQueue inQueue;
   protected MessageQueue controlQueue;
   protected MessageQueue dunnoQueue;

   protected IDictionary routingTable = (IDictionary)(new Hashtable());


   public DynamicRouter(MessageQueue inQueue, MessageQueue controlQueue, MessageQueue
dunnoQueue)
   {
      this.inQueue = inQueue;
      this.controlQueue = controlQueue;
      this.dunnoQueue = dunnoQueue;

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

      controlQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnControlMessage);
      controlQueue.BeginReceive();
   }

   protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
      Message message = mq.EndReceive(asyncResult.AsyncResult);

      String key = ((String)message.Body).Substring(0, 1);

      if (routingTable.Contains(key))
      {
         MessageQueue destination  = (MessageQueue)routingTable[key];
         destination.Send(message);
      }
      else
         dunnoQueue.Send(message);
      mq.BeginReceive();
   }

   // control message format is X:QueueName as a single string
```

```
   protected void OnControlMessage(Object source, ReceiveCompletedEventArgs
asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
      Message message = mq.EndReceive(asyncResult.AsyncResult);

      String text = ((String)message.Body);
      String [] split = (text.Split(new char[] {':'}, 2));
      if (split.Length == 2)
      {
         String key = split[0];
         String queueName = split[1];
         MessageQueue queue = FindQueue(queueName);
         routingTable.Add(key, queue);
      }
      else
      {
         dunnoQueue.Send(message);
      }
      mq.BeginReceive();
   }

   protected MessageQueue FindQueue(string queueName)
   {
      if (!MessageQueue.Exists(queueName))
      {
         return MessageQueue.Create(queueName);
      }
      else
         return new MessageQueue(queueName);
   }
}
```

This example uses a very simple conflict resolution mechanism -- last one wins. If two recipients express interest in receiving messages that start with the letter 'X', only the second recipient will receive the message because the hashmap stores only one queue for each key value. Also note that the `dunnoQueue` can now receive two types of messages: incoming messages that have no matching routing rules or control messages that do not match the required format.

Related patterns: *Content-Based Router*, *Message Filter*, *Message Router*, *Publish-Subscribe Channel*, *Recipient List*, *Routing Slip*

Recipient List

A *Content-Based Router* allows us to route a message to the correct system based on message content. This process is transparent to the original sender in the sense that the originator simply sends the message to a channel, where the router picks it up and takes care of everything.

In some cases, though, we may want to specify one or more recipients for the message. A common analogy are the recipient lists implemented in most e-mail systems. For each e-mail message, the sender can specify a list of recipients. The mail system then ensures transport of the message content to each recipient. An example from the domain of enterprise integration would be a situation where a function can be performed by one or more providers. For example, we may have a contract with multiple credit agencies to assess the credit worthiness of our customers. When a small order comes in we may simply route the credit request message to one credit agency. If a customer places a large order, we may want to route the credit request message to multiple agencies and compare the results before making a decision. In this case, the list of recipients depends on the dollar value of the order.

In another situation, we may want to route an order message to a select list of suppliers to obtain a quote for the requested item. Rather than sending the request to all vendors, we may want to control which vendors receive the request, possibly based on user preferences

How do we route a message to a list of dynamically specified recipients?

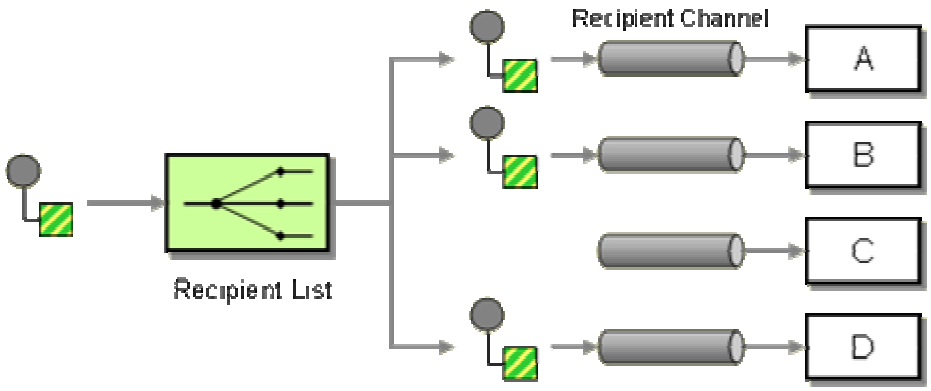
Because this problem is an extension to the problem that a *Content-Based Router* solves, some of the same forces and alternatives described in that pattern come into play here as well.

Most messaging systems provide *Publish-Subscribe Channel*s, which send a copy of a published messages to each recipient who subscribes to the channel. The set of recipients is based on subscription to the specific channel or subject. However, the list of active subscribers to a channel is somewhat static and cannot change on a message-by-message basis.

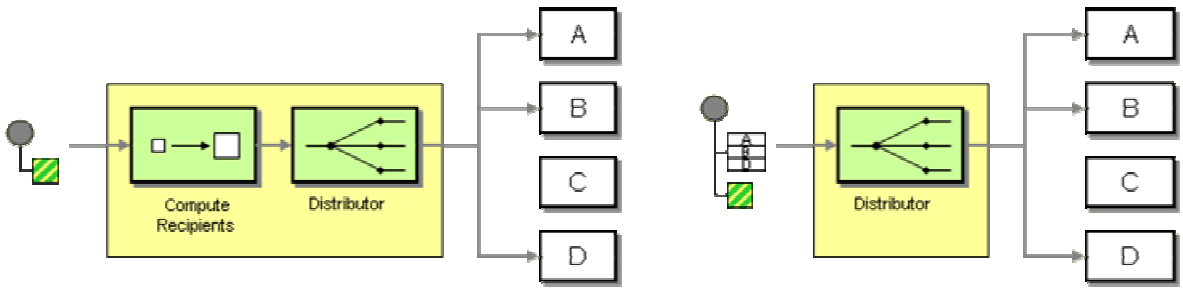
Because subscription to a *Publish-Subscribe Channel*s is binary (you are either subscribed to all messages on the channel or none), each potential recipient would have to filter messages incoming messages based on message content, most likely using a *Message Filter* or *Selective Consumer*. This distributes the logic over who receives the message to the individual subscribers. Going down this route we could maintain a central point of maintenance for the list of recipients by attaching a list of intended recipients to the message. When the message is broadcast to all possible recipients, each recipient would then look in the recipient list associated with the message. If the recipient is not part of the recipient list, it will discard the message. The problem with either approach its inefficiency by requiring each potential recipient to process every message just to possibly discard it. The configuration also relies on a certain 'honor' system on part of the recipients as we cannot really prevent a recipient from processing the message. This is definitely not desirable in situations where we forward a request for quote to a select subset of suppliers and expect others to ignore the message they are receiving.

We could also require the message originator to publish the message individually to each desired recipient. In that case, though we would place the burden of delivery to all recipients on the message originator. If we originator is a packaged application, this is generally not an option. Also, it would embed decision logic inside the application which would couple the application more tightly to the integration infrastructure. In many cases, the applications that are being integrated are unaware of the fact that they even participate in an integration solution, so expecting the application to contain message routing logic is not realistic.

Define a channel for each recipient. Then use a *Recipient List* to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list.



The logic embedded in a *Recipient List* can be pictured as two separate parts even though the implementation is often coupled together. The first part computes a list of recipients. The second part simply traverses the list and sends a copy of the received message to each recipient. Just like a *Content-Based Router*, the *Recipient List* usually does not modify the message contents.



A Recipient List Can Compute the Recipients (left) or Have Another Component Provide A List (right)

The recipient list can be derived from a number of sources. The creation of the list can be external to the *Recipient List* so that the message originator or another component attaches the list to the incoming message. The *Recipient List* only has to iterate through this ready-made list. In this situation, the *Recipient List* usually removes the recipient list from the message to reduce the size of the outgoing messages and prevent individual recipient from seeing who else is on the list. Providing the list with the incoming message makes sense if the destinations of each message are based on user selection.

In most cases, the *Recipient List* computes the list of recipients cased on the content of the message and a set of rules embedded in the *Recipient List*. The rules may be hard-coded or configurable (see below).

The *Recipient List* is subject to the same considerations regarding coupling as discussed under the *Message Router*, pattern. Routing messages predictively to individual recipients can lead to tighter coupling between components because a central has to have knowledge of a series of other components.

In order for the *Recipient List* to control flow of information we need to make sure that recipients cannot subscribe directly to the input channel into the *Recipient List*, bypassing any control the *Recipient List* exercises.

Robustness

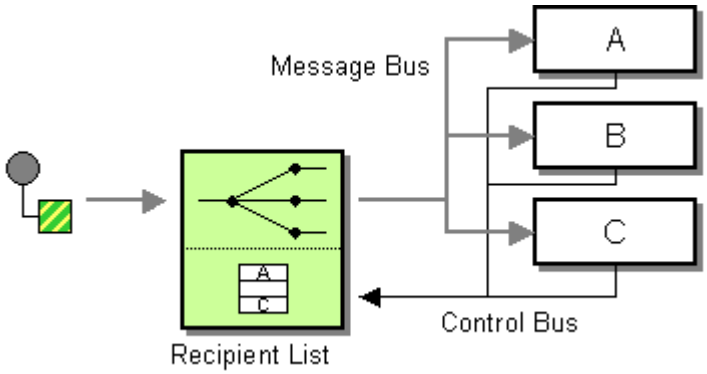
The *Recipient List* component is responsible for sending the incoming message to each recipient specified in the recipient list. A robust implementation of the *Recipient List* must be able to process the incoming message but only 'consume' it after all outbound messages have been successfully sent. As such, the Recipient List component has to ensure that the complete operation is atomic. If the *Recipient List* fails, it needs to be restartable. This can be accomplished in multiple ways:

- *Single transaction* - The *Recipient List* can use transactional channels and places the message on the outbound channels as part of a single transaction. It does not commit the messages until all messages are placed on the channels. This guarantees that either all or no messages are sent.
- *Persistent recipient list* - The *Recipient List* can "remember" which messages it already sent so that on failure and restart can send messages to the remaining recipients. The recipient list could be stored on disk or a database so that it survives a crash of the *Recipient List* component.
- *Idempotent receivers* Alternatively, the *Recipient List* could simply resend all messages on restart. This options requires all potential recipients to be *idempotent* (see *Idempotent Receiver*). Idempotent functions are those that do not change the state of the system if they are applied to themselves, i.e. the state of the component is not affected if the same message is processed twice. Messages can be inherently idempotent (e.g. the messages "All Widgets on Sale until May 30" or "get me a Quote for XYZ widgets" are unlikely to do harm if they are received twice) or the receiving component can be made idempotent by inserting a special *Message Filter* that eliminates duplicate messages. Idempotence is very handy because it allows us to simply resend messages when we are in doubt whether the recipient has received it. The TCP/IP protocol uses a similar mechanism to ensure reliable message delivery without unnecessary overhead (see [Stevens]).

Dynamic Recipient List

Even though the intent of the *Recipient List* is to maintain control, it can be useful to let the recipients themselves configure the rule set stored in the *Recipient List*, for example if recipients want to subscribe to specific messages based on rules that can not easily be represented in form

of publish-subscribe channel topics. We mentioned these types of subscription rules under the *Message Filter* Pattern, for example "accept the message if the price is less than $48.31". To minimize network traffic we would still want to send the messages only to interested parties as opposed to broadcasting it and letting each recipient decide whether to process the message or not. To implement this functionality, recipients can send their subscription preferences to the *Recipient List* via a special control channel. The *Recipient List* stores the preferences in a rules base and uses it to compile the recipient list for each message. This approach gives the subscribers control over the message filtering but leverages the efficiency of the *Recipient List* to distribute the messages. This solution combines the properties of a *Dynamic Router* with a *Recipient List* to create a *Dynamic Recipient List* (see picture).



A Dynamic Recipient List Is Configured by the Recipients via a Control Channel

This approach would work well for the 'price update' example discussed in the *Message Filter* pattern. Since it assigns control to the individual recipients it is not suitable for the bidder example mentioned at the beginning of this pattern, though.

Network (In)Efficiencies

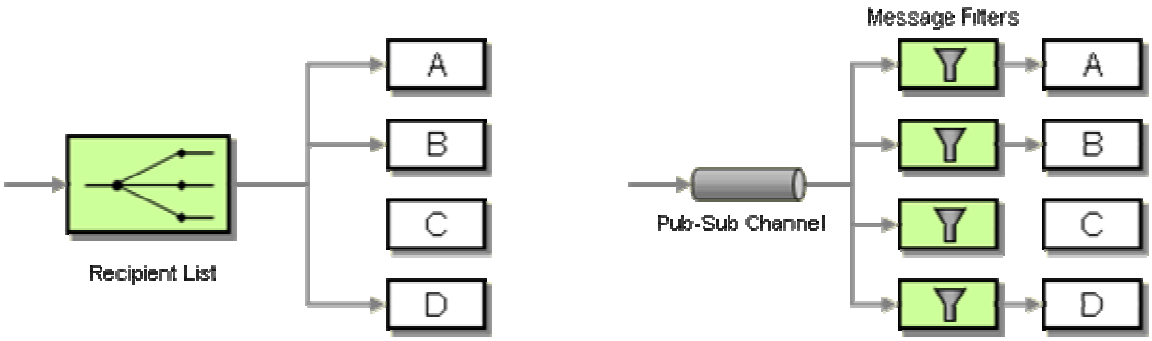
Whether it is more efficient to send one message to all all possible recipients who then filter the message or to send individual messages to each recipient depends very much on the implementation of the messaging infrastructure. Generally, we can assume that the more recipients a message has, the more network traffic it causes. However, there are exceptions. Some publish-subscribe messaging systems are based on IP Multicast functionality and can route messages to multiple recipients with a single network transmission (requiring retransmission only for lost messages). IP Multicast takes advantage of Ethernet's bus architecture. When an IP packet is sent across the network, all network adapters (NIC) on the same Ethernet segment receive the packet. Normally, the NIC verifies the intended recipient of the packet and ignores it if the packet is not addressed to the IP address the NIC is associated with. Multicast routing allows all receivers that are part of a specified multicast group to read the packet of the bus. This results in a single packet being able to be received by multiple NIC's who then pass the data to the respective application associated with the network connection. This approach can be very efficient on local networks due to the Ethernet bus architecture. It does not work across the Internet where point-to-point TCP/IP connections are required. In general, we can say that the

further apart the recipients are, the more efficient it is to use a *Recipient List* vs. a *Publish-Subscribe Channel*.

Whether a broadcast is more efficient depends not only on the network infrastructure, but also on the proportion between the number of recipients that are supposed to process the message over all recipients. If on average, most recipients are in the recipient list, it may be more efficient to simply broadcast the message and have the (few) non-participants filter the message out. If however, on average only a small portion of all possible recipients are interested in a particular message, the *Recipient List* is almost guaranteed to be more efficient.

Recipent List vs. Pub-Sub and Filters

A number of times we have contrasted implementing the same functionality using predictive routing with a *Recipient List* or using reactive filtering using a *Publish-Subscribe Channel* and an array of *Message Filter*s. Some of the decision criteria equal those of the the comparison between the *Content-Based Router* and the *Message Filter* array. However, in case of a *Recipient List*, the message can travel to multiple recipients, making the "filter" option more attractive.



Recipient List vs. MessageFilter Array

The following table compares the two solutions:

Content-Based Router	***Publish-Subscribe Channel*** **with** ***Recipient List***
Central control and maintenance -- predictive routing.	Distributed control and maintenance -- reactive filtering.
Router needs to know about participants. Router may need to be updated if participants are added or removed (unless using dynamic router, but at expense of losing control).	No knowledge of participants required. Adding or removing participants is easy.
Often used for business transactions, e.g. request for quote.	Often used for event notifications / informational messages.
Generally more efficient if limited to queue-based channels.	Can be more efficient with publish-subscribe channels (depends on infrastructure).

If we send a message to multiple recipients we may need to reconcile the results later. For example, if we send a request for a credit score to multiple credit agencies we should wait until all results come back so that we can compare the results and choose the best alternative . With other less critical functions we may just take the first available response to optimize message throughput. These types of strategies are typically implemented inside an *Aggregator*. *Scatter-Gather* describes situations where we start with a single message, send it to multiple recipients and re-combine the responses into a single message.

A *dynamic Recipient List* can be used to implement a *Publish-Subscribe Channel* if a messaging system provides only *Point-to-Point Channel*s but no *Publish-Subscribe Channel*. The *Recipient List* would keep a list of all *Point-to-Point Channel*s that are subscribed to the 'topic', represented by this specific instance of the *Recipient List*. This solution can also be useful if we need to apply special criteria to allow a recipient to subscribe to a source of data. The *Recipient List* could easily implement logic that controls access to the source data as long as the messaging system can ensure that the recipients don't have direct access to the input channel into the *Recipient List*.

Example: Loan Broker

The composed messaging example in the interlude at the end of this chapter (see *Introduction to Composed Messaging Examples*) uses a *Recipient List* to route a loan quote request only to qualified banks. The interlude shows implementations of the *Recipient List* in Java, C# and TIBCO.

Example: Dynamic Recipient List in C# and MSMQ

This example builds on the *Dynamic Router* example to turn it into a *dynamic Recipient List*. The code structure is very similar. The `DynamicRecipientList` listens on two input queues, one for incoming messages (`inQueue`) and a control queue (`controlQueue`) where recipients can hand in their subscription preferences. Messages on the control queue have to be formatted as a string consisting of two parts separated by a colon (':'). The first part is a list of characters that indicate the subscription preference of the recipient. The recipient expresses that it wants to receive all messages starting with one of the specified letters. The second part of the control message specifies the name of the queue that the recipient listens on. For example, the control message "W:WidgetQueue" tells the `DynamicRecipientList` to route all incoming messages that begin with "W" to the queue `WidgetQueue`. Likewise, the message "WQ:WidgetGadgetQueue" instructs the `DynamicRecipientList` to route messages that start with either "W" or "G" to the queue `DynamicRecipientList`.

```
class DynamicRecipientList
{
   protected MessageQueue inQueue;
   protected MessageQueue controlQueue;

   protected IDictionary routingTable = (IDictionary)(new Hashtable());
```

```
   public DynamicRecipientList(MessageQueue inQueue, MessageQueue controlQueue)
   {
      this.inQueue = inQueue;
      this.controlQueue = controlQueue;

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

      controlQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnControlMessage);
      controlQueue.BeginReceive();
   }

   protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
      Message message = mq.EndReceive(asyncResult.AsyncResult);

      if (((String)message.Body).Length > 0)
      {
         char key = ((String)message.Body)[0];

         ArrayList destinations  = (ArrayList)routingTable[key];
         foreach (MessageQueue destination in destinations)
         {
            destination.Send(message);
            Console.WriteLine("sending message " + message.Body + " to " +
destination.Path);
         }
      }
      mq.BeginReceive();
   }

   // control message format is XYZ:QueueName as a single string
   protected void OnControlMessage(Object source, ReceiveCompletedEventArgs
asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
```

```
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        String text = ((String)message.Body);
        String [] split = (text.Split(new char[] {':'}, 2));
        if (split.Length == 2)
        {
            char[] keys = split[0].ToCharArray();
            String queueName = split[1];
            MessageQueue queue = FindQueue(queueName);
            foreach (char c in keys)
            {
                if (!routingTable.Contains(c))
                {
                    routingTable.Add(c, new ArrayList());
                }
                ((ArrayList)(routingTable[c])).Add(queue);
                Console.WriteLine("Subscribed queue " + queueName + " for message " + c);
            }
        }
        mq.BeginReceive();
    }

    protected MessageQueue FindQueue(string queueName)
    {
        if (!MessageQueue.Exists(queueName))
        {
            return MessageQueue.Create(queueName);
        }
        else
            return new MessageQueue(queueName);
    }
}
```

The `DynamicRecipientList` uses a bit more clever (read complicated) way to store the recipient's preferences. To optimize processing of incoming messages, the `DynamicRecipientList` maintains a `Hashtable` keyed by the first letter of incoming messages. Unlike the *Dynamic Router* example, the Hashtable contains not a single destination, but an `Arraylist` of all subscribed destinations. When the `DynamicRecipientList` receives a message it locates the correct destination list from the `Hashtable` and then iterates over the list to send one message to each destination.

This example does not use a `dunnoChannel` (see *Content-Based Router* or *Dynamic Router*) for incoming messages that do not match any criteria. Typically, a *Recipient List* does not consider it an error of there are zero recipients for a message.

This implementation does not allow recipients to unsubscribe. It also does not detect duplicate subscription. For example, if a recipient subscribes twice for the same message type it will receive duplicate messages. This is different from the typical publish-subscribe semantics where a specific recipient can subscribe to one channel only once.The `DynamicRecipientList`could easily be changed to disallow duplicate subscriptions if that is desired.

Related patterns: *Aggregator*, *Scatter-Gather*, *Introduction to Composed Messaging Examples*, *Content-Based Router*, *Dynamic Router*, *Message Filter*, *Idempotent Receiver*, *Message Router*, *Selective Consumer*, *Point-to-Point Channel*, *Publish-Subscribe Channel*

Splitter

Many messages passing through an integration solution consist of multiple elements. For example, an order placed by a customer consists of more than just a single line item. As outlined in the description of the *Content-Based Router*, each line item may need to be handled by a different inventory system. Thus, we need to find an approach to process a complete order, but treat each order item contained in the order individually.

How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?

The solution to this routing problem should be generic enough so that it can deal with varying numbers and types of elements. For example, an order can contain any number of items, so we would not want to create a solution that assumes a fixed number of items. Nor would we want to make too many assumptions about what type of items the message contains. For example, if the Widget & Gadget company starts selling books tomorrow, we want to minimize the impact on the overall solution.

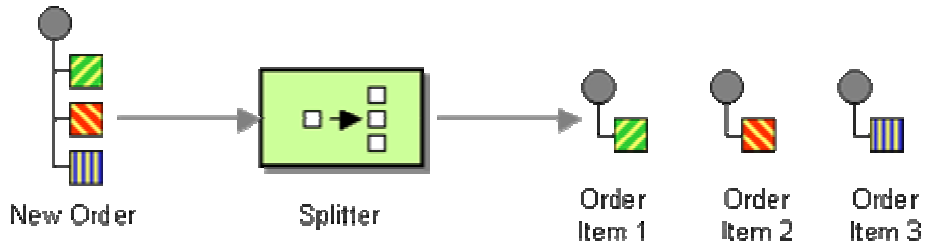
We also want to maintain control over the order items and avoid duplicated or lost processing. For example, we could send the complete order to each order management system using a *Publish-Subscribe Channel* and let it pick out the items that it can handle. This approach has the same disadvantages described in the *Content-Based Router*. It would be very difficult to avoid missing or duplicate shipment of individual items.

The solution should also be efficient in its usage of network resources. Sending the complete order message to each system that may only process a portion of the order can cause additional message traffic, especially as the number of destinations increases.

To avoid sending the complete message multiple times we could split the original message into as many messages as there are inventory systems. Each message would then contain only the line items that can be handled by the specific system. This approach is similar to a *Content-Based Router* except we are splitting the message and the routing the individual messages. This approach would be efficient but ties the solution to knowledge about the specific item types and associated destinations. What if we want to change the routing rules? We would now have to

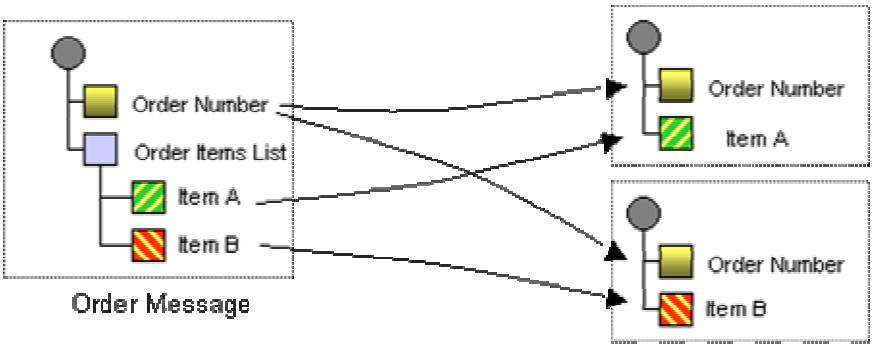
change this more complex "itemrouter" component. We use the *Pipes and Filters* architectural to break out processing into well-defined, composable components as opposed to lumping multiple functions together, so we should be able to take advantage of this architecture here as well.

Use a *Splitter* to break out the composite message into a series of individual messages, each containing data related to one item.



use a *Splitter* that consumes one message containing a list of repeating elements, each of which can be processed individually. The *Splitter* publishes a one message for each single element (or a subset of elements) from the original message.

In many cases, we want to repeat some common elements in each resulting message. These extra elements are required to make the resulting child message self-contained and therefore enables state-less processing of the child message. It also allows reconciliation of associated child messages later on. For example, each order item message should contain a copy of the order number so we can properly associate the order item back to the order and all associated entities such as the customer placing the order (see picture).



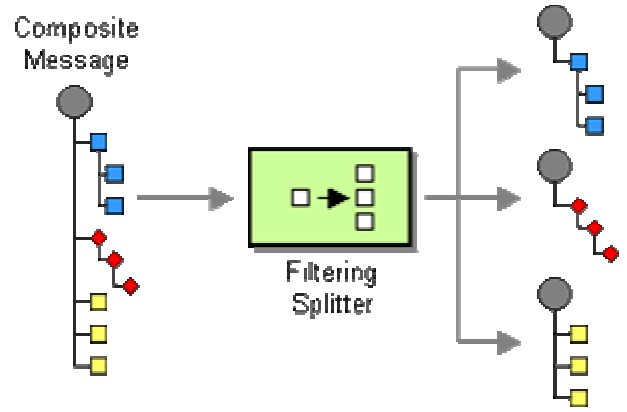
Iterating Splitters

As mentioned earlier, many enterprise integration systems store message data in a tree structure. The beauty of a tree structure is that it is recursive. Each child node underneath a node is the root of another subtree. This allows us to extract pieces of a message tree and process them further as a message tree on their own. If we use message trees , the *Splitter* can be easily be configured to iterate through all children under a specified node and send one message for each child node. Such a *Splitter* implementation would be completely generic because it does not make any

assumptions about the number and type of child elements. Many commercial EAI tools provide this type of functionality under the term *Iterator* or *Sequencer*. Since we are trying to avoid vendor vocabulary to reduce potential for confusion, we call this style of *Splitter* an *Iterating Splitter*.

Static Splitters

Using a *Splitter* is not limited to repeating elements, though. A large message may be split into individual messages to simplify processing. For example, a number of B2B information exchange standards specify very comprehensive message formats. These huge messages are often a result of design-by-committee and large portions of the messages may rarely be used. In many instances it is helpful to split these mega-messages into individual messages, each centered around a specific portion of the large message. This makes subsequent transformations much easier to develop and can also save network bandwidth since we can route smaller messages to those components that deal only with a portion of the mega-message. The resulting messages are often published to different channels rather than the same channel because they represent messages of different sub-types. In this scenario, the number of resulting messages is generally fixed whereas in the more general *Splitter* assumes a variable number of items. To distinguish this style of *Splitter* we call it *Static Splitter*. A *Static Splitter* is functionally equivalent to using a broadcast channel followed by a set of *Content Filter*s.



Ordered or Unordered Child Messages

In some cases it is useful to equip child messages with sequence numbers to improve massage traceability and simplify the task of an *Aggregator*. Also, it is a good idea to equip each message with a reference to the original (combined) message so that processing results from the individual messages can be correlated back to the original message. This reference functions as a *Correlation Identifier*.

If message envelopes are used (see *Envelope Wrapper*), each new message should be supplied with its own message envelope to make it compliant with the messaging infrastructure. For example, if the infrastructure requires a message to carry a timestamp in the message header, we would propagate the timestamp of the original message to each message's header.

Example: Splitting an XML Order Document in C#

Many messaging systems use XML messages. For example, let's assume an incoming order look as follows:

```
<order>
   <date>7/18/2002</date>
   <ordernumber>3825968</ordernumber>
   <customer>
      <id>12345</id>
      <name>Joe Doe</name>
   </customer>
   <orderitems>
      <item>
         <quantity>3.0</quantity>
         <itemno>W1234</itemno>
         <description>A Widget</description>
      </item>
      <item>
         <quantity>2.0</quantity>
         <itemno>G2345</itemno>
         <description>A Gadget</description>
      </item>
   </orderitems>
</order>
```

We want the *Splitter* to split the order into individual order items. For the example document the *Splitter* should generate the following two messages:

```
<orderitem>
   <date>7/18/2002</date>
   <ordernumber>3825968</ordernumber>
   <customerid>12345</customerid>
   <quantity>3.0</quantity>
   <itemno>W1234</itemno>
   <description>A Widget</description>
</orderitem>
<orderitem>
   <date>7/18/2002</date>
   <ordernumber>3825968</ordernumber>
   <customerid>12345</customerid>
   <quantity>2.0</quantity>
   <itemno>G2345</itemno>
   <description>A Gadget</description>
```

```
</orderitem>
```

Each `orderitem` message is being enriched with the order date, the order number, and the customer ID. The inclusion of the customer ID and the order date make the message self-contained and keeps the message consumer from having to store context across individual messages. This is important if the messages are to be processed by stateless servers. The addition of the `ordernumber` field is necessary for later re-aggregation of the items (see *Aggregator*. In this example we assume that the specific order of items is not relevant for completion of the order, so we did not have to include an item number.

Let's see what the *Splitter* code looks like in C#.

```
class XMLSplitter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue;

    public XMLSplitter(MessageQueue inQueue, MessageQueue outQueue)
    {
        this.inQueue = inQueue;
        this.outQueue = outQueue;

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

        outQueue.Formatter = new ActiveXMessageFormatter();
    }

    protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new ActiveXMessageFormatter();
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        XmlDocument doc = new XmlDocument();
        doc.LoadXml((String)message.Body);

        XmlNodeList nodeList;
        XmlElement root = doc.DocumentElement;

        XmlNode date = root.SelectSingleNode("date");
        XmlNode ordernumber = root.SelectSingleNode("ordernumber");
        XmlNode id = root.SelectSingleNode("customer/id");
        XmlElement customerid = doc.CreateElement("customerid");
        customerid.InnerText = id.InnerXml;
```

```
        nodeList = root.SelectNodes("/order/orderitems/item");

        foreach (XmlNode item in nodeList)
        {
            XmlDocument orderItemDoc = new XmlDocument();
            orderItemDoc.LoadXml("<orderitem/>");
            XmlElement orderItem = orderItemDoc.DocumentElement;

            orderItem.AppendChild(orderItemDoc.ImportNode(date, true));
            orderItem.AppendChild(orderItemDoc.ImportNode(ordernumber, true));
            orderItem.AppendChild(orderItemDoc.ImportNode(customerid, true));

            for (int i=0; i < item.ChildNodes.Count; i++)
            {
                orderItem.AppendChild(orderItemDoc.ImportNode(item.ChildNodes[i],
true));
            }

            outQueue.Send(orderItem.OuterXml);
        }

        mq.BeginReceive();
    }

}
```

Most of the code centers around the XML processing. The `XMLSplitter` uses the same *Event-Driven Consumer* structure as the other routing examples. Each incoming message invokes the method `OnMessage`. `Onmessage` converts the message body into an XML document for manipulation. First, we extract the relevant values from the order document. Then, we iterate over each `<item>` child element. We do this by specifying the XPath expression `/order/orderitems/item`. A simple XPath expression is very similar to a file path -- it descends down the document tree, matching the element names specified in the path. For each `<item>` we assemble a new XML document, copying the fields carried over from the order and the item's child nodes.

Example: Splitting an XML Order Document in C# and XSL

Instead of manipulating XML nodes and elements manually, we can also create an XSL document to transform the incoming XML into the desired format and then create output messages from the transformed XML document. That is more maintainable when the document format is likely to change. All we have to do is change the XSL transformation without any changes to the C# code.

The new code uses the `Transform` method provided by the `XslTransform` class to convert the input document into an intermediate document format. The intermediate document format has one child element `orderitem` for each resulting message. The code simply traverses all child elements and publishes one message for each element.

```
class XSLSplitter
{
   protected MessageQueue inQueue;
   protected MessageQueue outQueue;

   protected String styleSheet = "..\\..\\Order2OrderItem.xsl";
   protected XslTransform xslt;

   public XSLSplitter(MessageQueue inQueue, MessageQueue outQueue)
   {
      this.inQueue = inQueue;
      this.outQueue = outQueue;

      xslt = new XslTransform();
      xslt.Load(styleSheet, null);

      outQueue.Formatter = new ActiveXMessageFormatter();

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

   protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new ActiveXMessageFormatter();
      Message message = mq.EndReceive(asyncResult.AsyncResult);

      try
      {
         XPathDocument doc = new XPathDocument(new
StringReader((String)message.Body));

         XmlReader reader = xslt.Transform(doc, null, new XmlUrlResolver());

         XmlDocument allItems = new XmlDocument();
         allItems.Load(reader);

         XmlNodeList nodeList =
allItems.DocumentElement.GetElementsByTagName("orderitem");
```

```
            foreach (XmlNode orderItem in nodeList)
            {
                outQueue.Send(orderItem.OuterXml);

            }
        }
        catch (Exception e) { Console.WriteLine(e.ToString()); }
        mq.BeginReceive();
    }
}
```

We read the XSL document from a separate file to make it easier to edit and test. Also, it allows us to change the behavior of the *Splitter* without recompiling the code.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

    <xsl:template match="/order">
        <orderitems>
            <xsl:apply-templates select="orderitems/item"/>
        </orderitems>
    </xsl:template>

    <xsl:template match="item">
        <orderitem>
            <date>
                <xsl:value-of select="parent::node()/parent::node()/date"/>
            </date>
            <ordernumber>
                <xsl:value-of select="parent::node()/parent::node()/ordernumber"/>
            </ordernumber>
            <customerid>
                <xsl:value-of select="parent::node()/parent::node()/customer/id"/>
            </customerid>
            <xsl:apply-templates select="*"/>
        </orderitem>
    </xsl:template>

    <xsl:template match="*">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()"/>
        </xsl:copy>
    </xsl:template>
```

```
</xsl:stylesheet>
```

XSL is a declarative language, so it is not easy to make sense of unless you have written a fair bit of XSL yourself (or read a good XSL book like [Tennison]). This XSL transform looks for any occurrence of the `order` element (there is one in our document). Once it finds this element it creates a new root element for the output document (all XML documents have to have a single root element) and goes on to process all `item` elements inside the `orderitems` element of the input document. The XSL specifies a new 'template' for each `item` that is found. This template copies the `date`, `ordernumber` and `customerid` from `order` element (which is the `item`'s parent's parent) and then appends any element from the `item`. The resulting document has one `orderitem` element for each `item` element in the input document. This make it easy for the C# code to iterate over the elements and publish them as messages

We were curious as how to the two implementations would perform. We decided to run a real quick, non-scientific performance test. We simply piped 5000 order messages into the input Queue, started the *Splitter* and measured the time it took for 10,000 item messages to arrive on the output queue. We executed this all inside a single program on one machine using local message queues. We measured 7 seconds for the `XMLSplitter` that uses the DOM to extract elements and 5.3 seconds for the XSL-based *Splitter*. To establish a baseline, a dummy processor that consumes one message of the input queue and publishes the same message twice on the output queue took just under 2 seconds for 5000 messages. This time includes the dummy processor consuming 5,000 messages and publishing 10,000, and the test harness consuming the 10,000 messages the processor published. So it looks like the XSL manipulation is a little more efficient than moving elements around 'by hand' (if we subtract the baseline, the XSL is about 35% faster). We are sure that either program could be tuned for maximum performance, but it was interesting to see them execute side-by-side.

Related patterns: *Aggregator*, *Content-Based Router*, *Content Filter*, *Correlation Identifier*, *Envelope Wrapper*, *Event-Driven Consumer*, *Pipes and Filters*, *Publish-Subscribe Channel*

Aggregator

A *Splitter* is useful to break out a single message into a sequence of sub-messages that can be processed individually. Likewise, a *Recipient List* or a *Publish-Subscribe Channel* is useful to forward a request message to multiple recipients in parallel in order to get multiple responses to choose from. In most of these scenarios, the further processing depends on successful processing of the sub-messages. For example, we want to select the best bid from a number of vendor responses or we want to bill the client for an order after all items have been pulled from the warehouse.

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

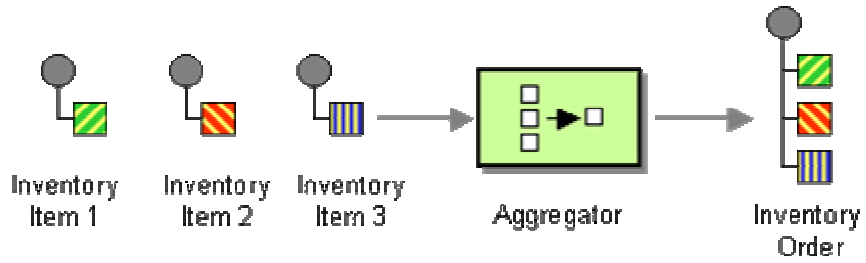
The asynchronous nature of a messaging system makes collecting information across multiple messages challenging. How many messages are there? If we broadcast a message to a broadcast channel, we may not know how many recipients listened to that channel and therefore cannot know how many responses to expect.

Even if we use a *Splitter*, the response messages may not arrive in the same sequence they were created in. As individual messages can be routed through different network paths, the messaging infrastructure can usually guarantee the delivery of each message, but may not be able to guarantee the order in which the individual messages are delivered. In addition, the individual messages may be processed by different parties with different processing speeds. As a result, response messages may be delivered out of order (see the *Resequencer* for a more detailed description of this problem).

In addition, most messaging infrastructures operate in a "guaranteed, ultimately" delivery mode. That means, that messages are guaranteed to be delivered to the intended recipient, but there are no guarantees as to when the message will be delivered. How long should we wait for a message? If we wait too long, we may delay subsequent processing. If we decide to move ahead without the missing message, we have to find a way to work with incomplete information. Even so, what should we do when the missing message (or messages) finally arrives? In some cases we may be able to process the message separately, but in general other cases that may lead to duplicate processing. On the other hand, if we ignore the late-comer messages, we permanently lose the information content contained in these messages.

All these issues can complicate the combined processing of multiple, but related messages. It would be much easier to implement the business logic if a separate component could take care of these complexities and pass a single message to the subsequent processing business that depends on the presence of all individual sub-messages.

Use a stateful filter, an *Aggregator*, to collect and store individual messages until a complete set of related messages has been received. Then, the *Aggregator* publishes a single message distilled from the individual messages.



The *Aggregator* is a special *Filter* that receives a stream of messages and identifies messages that are correlated. Once a complete set of messages has been received (more on how to decide when a set is 'complete' below), the *Aggregator* collects information from each correlated message and publishes a single, aggregated message to the output channel for further processing.

Unlike most of the previous routing patterns, the *Aggregator* is a *stateful* component. Simple routing patterns like the *Content-Based Router* are often *stateless*, which means the component processes incoming messages one-by-one and does not have to keep any information between messages. After processing a message, the component is in the same state as it was before the message arrived. Therefore, we call such a component stateless. The *Aggregator* cannot be stateless since it needs to store each incoming message until all the messages that belong together have been received. Then, it needs to distill the information associated with each message into the aggregate message. The *Aggregator* does not necessarily have to store each incoming message in its entirety. For example, if we are processing incoming auction bids, we may only need to keep the highest bid and the associated bidder ID without having to keep the history of all individual bid messages. Still, the *Aggregator* has to store information across messages and is therefore stateful.

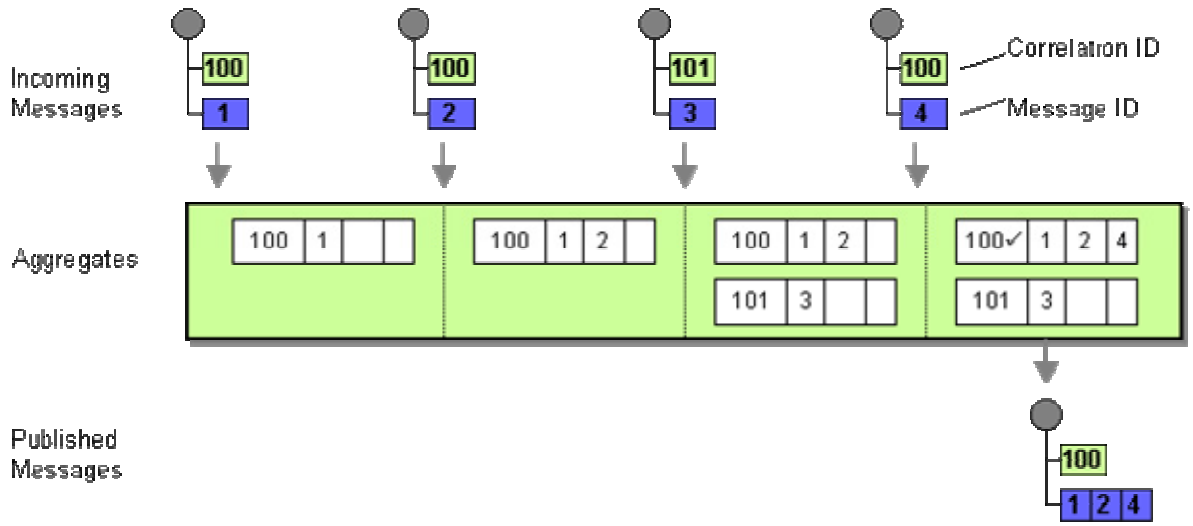
When designing an *Aggregator*, we need to specify the following items:

- **Correlation** - which incoming messages belong together?
- **Completeness Condition** - when are we ready to publish the result message?
- **Aggregation Algorithm** - how do we combine the received messages into a single result message?

Correlation is typically achieved by either the type of the incoming messages or an explicit *Correlation Identifier*. Common choices for the completeness condition and aggregation algorithm are described below.

Implementation Details

Due to the event-driven nature of a messaging system, the *Aggregator* may receive related messages at any time and in any order. To associate messages, the *Aggregator* maintains a list of active aggregates, i.e. aggregates for which the *Aggregator* has received some messages already. When the *Aggregator* receives a new message, it needs to check whether the message is part of an already existing aggregate. If no aggregate related to this message exists, the *Aggregator* assumes that this is the first message of a a set and creates a new aggregate. It then adds the message to the new aggregate. If an aggregate already exists, the *Aggregator* simply adds the message to the aggregate. After adding the message, the *Aggregator* evaluates the completeness condition for the aggregate (described in more detail below). If the condition evaluates to true, a new aggregate message is formed from the aggregate and published to the output channel. If the completeness condition evaluates to false, no message is published and the *Aggregator* keeps the aggregate active for additional messages to arrive. The following diagram illustrates this strategy. In this simple scenario, we assume an aggregate to be complete whenever it contains at least three messages.



This strategy creates a new aggregate whenever it receives a message that cannot be associated to an existing aggregate. Therefore, the *Aggregator* does not need prior knowledge of the aggregates that is may produce. Accordingly, we call this variant a *Self-starting Aggregator*.

Depending on the aggregation strategy the *Aggregator* may have to deal with the situation that an incoming message belongs to an aggregate that has already been closed out, i.e. after the aggregate message has been published. In order to avoid starting a new aggregate, the *Aggregator* needs to keep a list of aggregates that have been closed out. We need to provide a mechanism to purge this list periodically so that it does not grow indefinitely. This assumes that we can make some basic assumptions about the time frame in which related messages will arrive. Since we do not need to store the complete aggregate, but just the fact that it has been closed, we can store the list of closed aggregates quite efficiently and build a sufficient safety margin into the purge algorithm. We can also use *Message Expiration* to ignore messages that have been delayed for an inordinate amount of time.

In order to increase the robustness of the overall solution we can also allow the *Aggregator* to listen on a specific control channel which allows the manual purging of all active aggregates or a specific one. This feature can be useful if we want to recover from an error condition without having to restart the *Aggregator* component. Along the same lines, allowing the *Aggregator* to publish a list of active aggregates to a special channel upon request can be a very useful debugging feature. Both functions are excellent examples of the kind of features typically incorporated into a *Control Bus*.

Aggregation Strategies

There are a number of strategies for aggregator completeness conditions. The available strategies primarily depend on whether we know how many messages to expect or not. The *Aggregator* could know the number of sub-messages to expect because it received a copy of the original composite message or because each individual message contains the total count (as described in

the *Splitter* example). Depending on how much the *Aggregator* knows about the message stream, the most common strategies are as follows:

- **"Wait for All"** Wait until all responses are received. This scenario is most likely in the order example we discussed earlier. An incomplete order may not be meaningful. So if not all items are received within a certain time-out period an error condition should be raised by the *Aggregator*. This approach may give us the best basis for decision-making, but may also be the slowest and most brittle (plus we need to know how many messages to expect). A single missing or delayed message will prevent further processing of the whole aggregate. Resolving such error conditions can be a complicated matter in loosely-coupled asynchronous systems because the asynchronous flow of messages makes it hard to reliably detect error conditions (how long should we wait before a message is "missing"?). One way to deal with missing messages is to re-request the message. However, this approach requires the *Aggregator* to know the source of the message, which may introduce additional dependencies between the *Aggregator* and other components.
- **"Time Out"** Wait for a specified length of time for responses and then make a decision by evaluating those responses received within that time limit. If no responses are received, the system may report an exception or retry. This heuristic is useful if incoming responses are scored and only the message (or a small number of messages) with the highest score is used. This approach is common in "bidding" scenarios.
- **"First Best"** Wait only until the first (fastest) response is received and ignore all other responses. This approach is the fastest, but ignores a lot of information. It may be practical in a bidding or quoting scenario where response time is critical.
- **"Time Out with Override"** Wait for a specified amount of time or until a message with a preset minimum score has been received. In this scenario, we are willing to abort early if we find a very favorable response; otherwise, we keep on going until time is up. If no clear winner was found at that point, rank ordering among all the messages received so far occurs.
- **"External Event"** Sometimes the aggregation is concluded by the arrival of an external business event. For example, in the financial industry, the end of the trading day may signal the end of an aggregation of incoming price quotes. Using a fixed timer for such an event reduces flexibility because it does not other variability. Also, a designated business even in form of an *Event Message* allows for central control of the system. The *Aggregator* can listen for the *Event Message* on a special control channel or receive a specially formatted message that indicates the end of the aggregation.

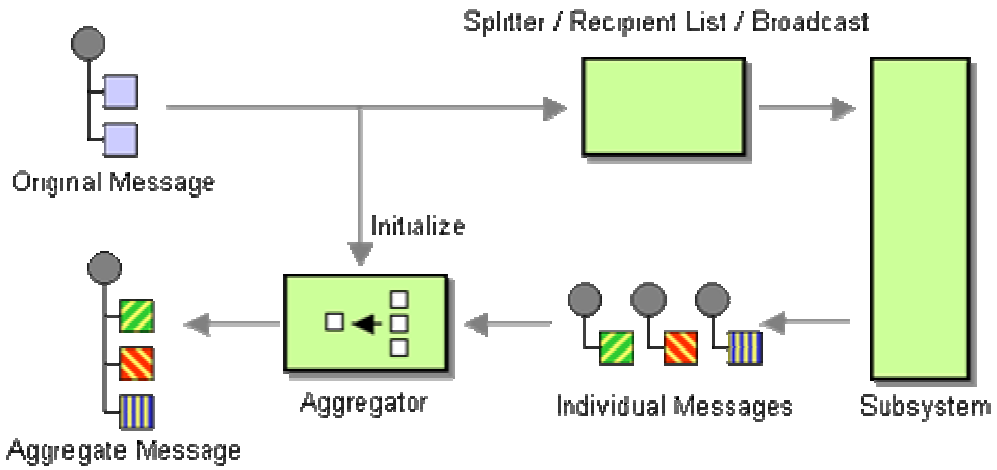
Closely tied to the selection of a completeness condition is the selection of the aggregation algorithm. The following strategies are common to condense multiple messages into a single message:

- **Select the "best" answer**. This approach assumes that there is a single best answer, e.g. the lowest bid for an identical item. This makes it possible for the *Aggregator* to make the decision and only pass the "best" message on. However, in real life, selection criteria are

rarely this simple. For example, the "best" bid for an item may depend on time of delivery, the number of available items, whether the vendor is on the preferred vendor list etc.

- **Condense data**. An *Aggregator* can be used to reduce message traffic from a high-traffic source. In these cases it may make sense to compute an average of individual messages or add numeric fields from each message into a single message. This works best if each message represents a numeric value, for example, the number of orders received.
- **Collect data for later evaluation**. it is not always possible to for an *Aggregator* to make the decision of how to select the best answer. In those cases it makes still sense to use an *Aggregator* to collect the individual messages and combine them into a single message. This message may simply be a compilation of the individual's messages data. The aggregation decision may be made later by a separate component or a human being.

In many instances, the aggregation strategy is driven by parameters. For example, a strategy that waits for a specified amount of time can be configured with the maximum wait time. Likewise, if the strategy is to wait until an offer exceeds a specific threshold we will most likely let the *Aggregator* know in advance what the desired threshold is. If these parameters are configurable at run-time, an *Aggregator* may feature an additional input that can receive control messages such as these parameter settings. The control messages may also contain information such as the number of correlated messages to expect, which can help the *Aggregator* implement more effective completion conditions. In such a scenario, the *Aggregator* does not simply start a new aggregate when the first message arrives, but rather receives up-front information related to an expected series of messages. This information can be a copy of the original request message (e.g., an *Scatter-Gather* message), augmented by any necessary parameter information. The *Aggregator* then allocates a new aggregate and stores the parameter information with the aggregate (see figure). When the individual messages come in, they are associated with the corresponding aggregate. We call this variation an *Initialized Aggregator* as opposed to the Self-starting *Aggregator*. This configuration is obviously only possible if we have access to the originating message, which may not always be the case.



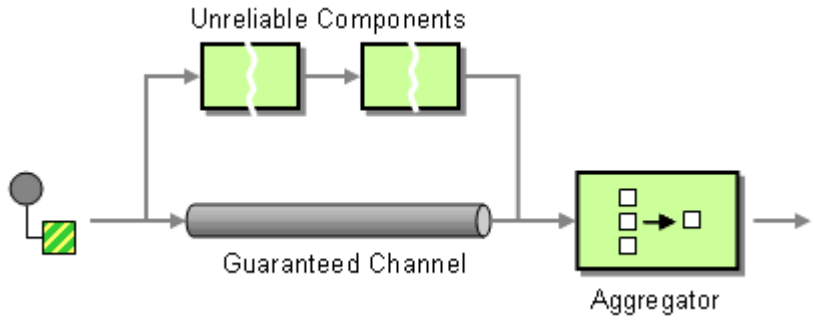
*Aggregator*s are useful in many applications. The *Aggregator* is often coupled with a *Splitter* or a *Recipient List* to form a composite pattern. See *Composed Message Processor* and *Scatter-Gather* for a more detailed description of these composite patterns.

Example: Loan Broker

The composed messaging example in the interlude at the end of this chapter (see *Introduction to Composed Messaging Examples*) uses an *Aggregator* to select the best loan quote from the loan quote messages returned by the banks. The loan broker example uses an *initialized Aggregator* -- the Recipient List informs the *Aggregator* of the number of quote messages to expect. The interlude shows implementations of the *Aggregator* in Java, C# and TIBCO.

Example: Aggregator as Missing Message Detector

Joe Walnes showed me a creative use of an *Aggregator*. His system sends a message through a sequence of components, which are unfortunately quite unreliable. Even using *Guaranteed Delivery* will not correct this problem because typically the systems themselves fail after the consumed a message. Because the applications are not *Transactional Client*s, the message-in-progress is lost. To help remedying this situation, Joe routes an incoming through two parallel paths -- once through the required, but unreliable components and once around the components using *Guaranteed Delivery*. An *Aggregator* recombines the messages from the two paths (see picture).



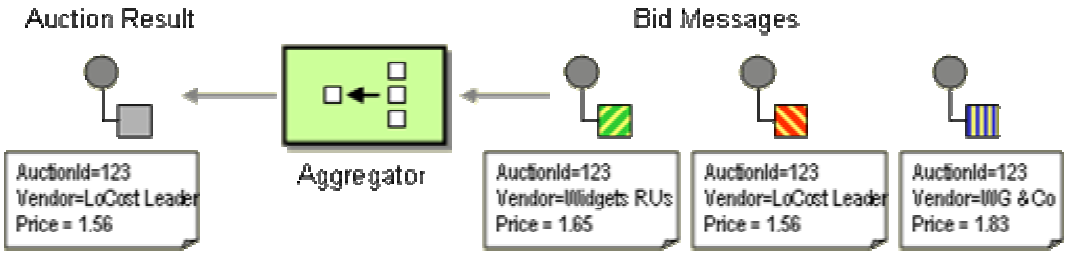
An Aggregator with Time-out Detects Missing Messages

The *Aggregator* uses a "Time Out with Override" completeness condition, which means that the *Aggregator* completes if either the time-out is reached or the two associated message have been received. The aggregation algorithm depends on which condition is fulfilled first. If two messages are received, the processed message is passed on without modification. If the time-out event occurs, we know that one of the components failed and "ate" the message. As a result, we instruct the *Aggregator* to publish an error message that alerts the operators that one of the components has failed. Unfortunately, the components have to be restarted manually, but a more sophisticated configuration could likely restart the component and re-send any lost messages.

Example: Aggregator in JMS

This example show the implementation of an *Aggregator* using the Java Messaging Service (JMS) API. The *Aggregator* receives bid messages on one channel, aggregates all related bids and

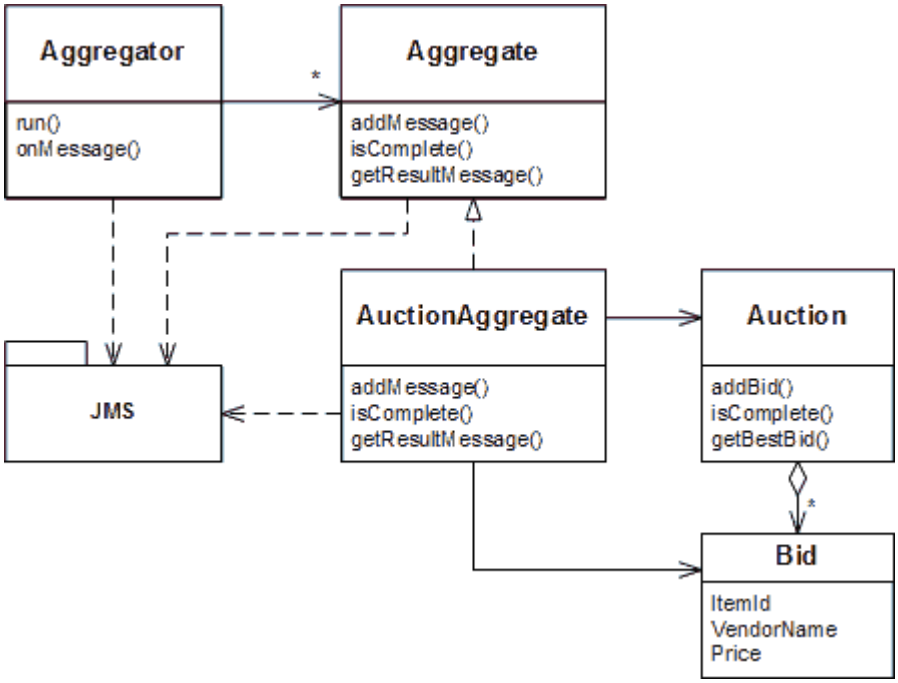
publishes a message with the lowest bid to another channel. Bids are correlated through an Auction ID property that acts as a *Correlation Identifier* for the messages. The aggregation strategy is to receive a minimum of 3 bids. The *Aggregator* is self-starting and does not require external initialization.



The Aggregator Example Selects the Lowest Bid

The solution consists of the following main classes:

- **Aggregator** - contains logic to receive messages, aggregate them and send result messages. Interfaces with aggregates via the Aggregate interface.
- **AuctionAggregate** - implements the Aggregate interface. This class acts as an *Adapter* (see [GoF]) between the Aggregate interface and the Auction class. This setup allows the Auction class to be free of references to the JMS API.
- **Auction** - a collection of related bids that have been received. The Auction class implements the aggregation strategy, e.g. finding the lowest bid and determining when the aggregate is complete.
- **Bid** - is a convenience class that holds the data items associated with a bid. We convert incoming message data into a bid object so that we can access the bid data through a strongly-typed interface, making the the Auction logic completely independent from the JMS API.



The code of the solution is the `Aggregator` class. This class requires two JMS *destinations*, an input destination and an output destination. Destination is the JMS abstraction for a queue or a topic (*Publish-Subscribe Channel*). This abstraction allows us to write JMS code independent from the type of channel. This feature can be very useful for testing and debugging. For example, during testing we may use publish-subscribe topics so that we can easily "listen in" on the message traffic. In production may want to switch to queues.

```
public class Aggregator implements MessageListener
{
    static final String PROP_CORRID = "AuctionID";

    Map activeAggregates = new HashMap();

    Destination inputDest = null;
    Destination outputDest = null;
    Session session = null;

    MessageConsumer in = null;
    MessageProducer out = null;

    public Aggregator (Destination inputDest, Destination outputDest, Session session)
    {
        this.inputDest = inputDest;
        this.outputDest = outputDest;
        this.session = session;
    }
```

```
    public void run()
    {
        try {

            in = session.createConsumer(inputDest);
            out = session.createProducer(outputDest);
            in.setMessageListener(this);
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString());
        }
    }

    public void onMessage(Message msg)
    {
        try {
            String correlationID = msg.getStringProperty(PROP_CORRID);
            Aggregate aggregate = (Aggregate)activeAggregates.get(correlationID);
            if (aggregate == null) {
                aggregate = new AuctionAggregate(session);
                activeAggregates.put(correlationID, aggregate);
            }
            //--- ignore message if aggregate is already closed
            if (!aggregate.isComplete()) {
                aggregate.addMessage(msg);
                if (aggregate.isComplete()) {
                    MapMessage result = (MapMessage)aggregate.getResultMessage();
                    out.send(result);
                }
            }
        } catch (JMSException e) {
            System.out.println("Exception occurred: " + e.toString());
        }
    }
}
```

The `Aggregator` is a *Event-Driven Consumer* and implements the `MessageListener` interface which requires it to expose the `onMessage` method. Setting the current instance of the `Aggregator` as the message listener for the `MessageConsumer` causes JMS to invoke the method `onMesssage` every time a new message is received on the destination specified by the `MessageConsumer`. For each incoming message the `Aggregator` extracts the correlation ID (stored as a message property) and checks whether an active aggregate exists for this correlation ID. If no aggregate is found, the `Aggregator` instantiates a new `AuctionAggregate`. The `Aggregator`then checks whether the aggregate is still active (i.e. not complete). If the aggregate is no longer active, it discard the incoming message. If the aggregate is active, it adds the message to the aggregate and tests

whether the termination condition has been fulfilled. If so, it gets the best bid entry and publishes it.

The `Aggregator` code is very generic and depends on this specific example application only in two lines of code. First, it assumes that the correlation ID is stored in the message property `AuctionID`. Second, it creates an instance of the class `AuctionAggregate`. We could avoid this reference if we used a factory that returns an object of type `Aggregate` and internally creates an instance of type `AuctionAggregate`. Since this is a book on enterprise integration and not on object-oriented design, we kept things simple and let this dependency pass.

The `AuctionAggregate` class needs to implement the `Aggregate` interface. The interface is rather simple, specifying only three methods. One to add a new message (`addMessage`), one to determine whether the aggregate is complete (`isComplete`) and one to get the best result (`getBestMessage`).

```
public interface Aggregate {
    public void addMessage(Message message);
    public boolean isComplete();
    public Message getResultMessage();
}
```

Instead of implementing the aggregation strategy inside the `AuctionAggreagte` class, we decided to create a separate class `Auction` that implements the aggregation strategy but is not dependent on the JMS API:

```
public class Auction
{
    ArrayList bids = new ArrayList();

    public void addBid(Bid bid)
    {
        bids.add(bid);
        System.out.println(bids.size() + " Bids in auction.");
    }

    public boolean isComplete()
    {
        return (bids.size() >= 3);
    }

    public Bid getBestBid()
    {
        Bid bestBid = null;

        Iterator iter = bids.iterator();
        if (iter.hasNext())
```

```
            bestBid = (Bid) iter.next();
        while (iter.hasNext()) {
            Bid b = (Bid) iter.next();
            if (b.getPrice() < bestBid.getPrice()) {
                bestBid = b;
            }
        }
        return bestBid;
    }
}
```

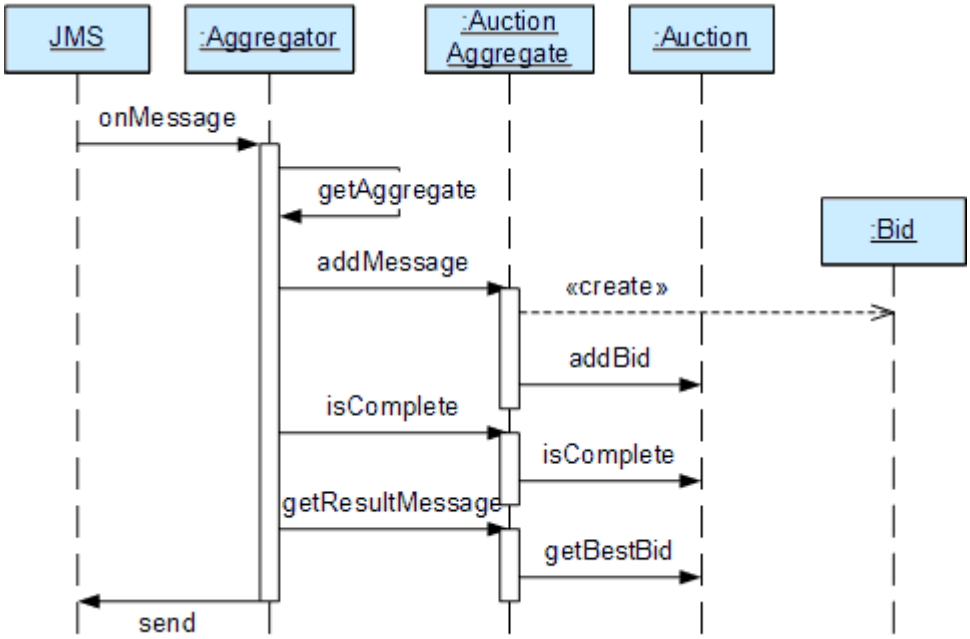
The `Auction` is actually quite simple. It provides three methods similar to the `Aggregate` interface, but the method signatures differ in that they use the strongly typed `Bid` class instead of the `Message` class. For this example, the aggregation strategy is very simple, simply waiting until three bids have been received. However, by separating the aggregation strategy from the `Auction` class and the JMS API is is easy to enhance the `Auction` class to incorporate more sophisticated logic.

The `AuctionAggregate` class acts as an *Adapter* between the `Aggregate` interface and the `Auction` class. An adapter is a class that converts the interface of a class into another interface.

```
public class AuctionAggregate implements Aggregate {
    static String PROP_AUCTIONID = "AuctionID";
    static String ITEMID = "ItemID";
    static String VENDOR = "Vendor";
    static String PRICE = "Price";

    private Session session;
    private Auction auction;

    public AuctionAggregate(Session session)
    {
        this.session = session;
        auction = new Auction();
    }

    public void addMessage(Message message) {
        Bid bid = null;
        if (message instanceof MapMessage) {
            try {
                MapMessage mapmsg = (MapMessage)message;
                String auctionID = mapmsg.getStringProperty(PROP_AUCTIONID);
                String itemID = mapmsg.getString(ITEMID);
                String vendor = mapmsg.getString(VENDOR);
                double price = mapmsg.getDouble(PRICE);
```

```
            bid = new Bid(auctionID, itemID, vendor, price);
            auction.addBid(bid);
         } catch (JMSException e) {
            System.out.println(e.getMessage());
         }
      }
   }

   public boolean isComplete()
   {
      return auction.isComplete();
   }

   public Message getResultMessage() {
      Bid bid = auction.getBestBid();
      try {
         MapMessage msg = session.createMapMessage();
         msg.setStringProperty(PROP_AUCTIONID, bid.getCorrelationID());
         msg.setString(ITEMID, bid.getItemID());
         msg.setString(VENDOR, bid.getVendorName());
         msg.setDouble(PRICE, bid.getPrice());
         return msg;
      } catch (JMSException e) {
         System.out.println("Could not create message: " + e.getMessage());
         return null;
      }
    }
}
```

The following sequence diagram summarizes the interaction between the classes:



This simple example assumes that Auction IDs are universally unique. This allows us to not worry about cleaning up the open auction list -- we just let it grow. In a real-life application we would need to decide when to purge old auction records to avoid memory leaks.

Because this code only references JMS destinations we can run it with wither topics or queues. In a production environment, this application may be more likely to employ a *Point-to-Point Channel* (equivalent to a JMS queue) because there should only be a single recipient for a bid, the *Aggregator*. As described in *Publish-Subscribe Channel*, topics can simplify testing and debugging. It is very easy to add an additional listener to a topic without affecting the flow of messages. Many times when I debug a messaging application, I run a separate 'listener' window that tracks all messages. Many JMS implementations allow you to use wildcards in topic names so that a listener can simply subscribe to all topics by specifying a topic name of '*'. It is very handy to have a simple listener tool that displays all messages traveling on a topic and also logs the messages into a file for later analysis.

Related patterns: *Scatter-Gather*, *Introduction to Composed Messaging Examples*, *Content-Based Router*, *Control Bus*, *Correlation Identifier*, *Composed Message Processor*, *Event-Driven Consumer*, *Event Message*, *Guaranteed Delivery*, *Message Expiration*, *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Recipient List*, *Resequencer*, *Splitter*, *Transactional Client*

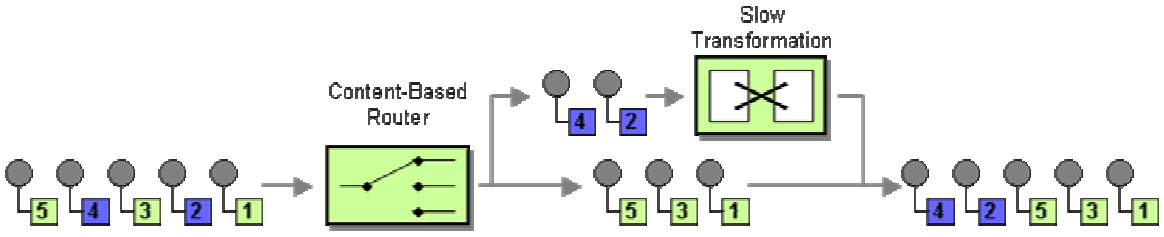
Resequencer

A *Message Router* can route messages from one channel to different channels based on message content or other criteria. Because individual messages may follow different routes, some messages are likely to pass through the processing steps sooner than others, resulting in the messages getting out of order. However, some subsequent processing steps do require in-sequence processing of messages, for example to maintain referential integrity.

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

The obvious solution to the out-of-sequence problem is to keep messages in sequence in the first place. Keeping things in order is in fact easier than getting them back in order. That's why many university libraries like to prevent readers from putting books back into the (ordered) bookshelf. By controlling the insert process, correct order is (almost) guaranteed at any point in time. But keeping things in sequence when dealing with an asynchronous messaging solution can be about as difficult as convincing a teenager that keeping their room in order is actually the more efficient approach.

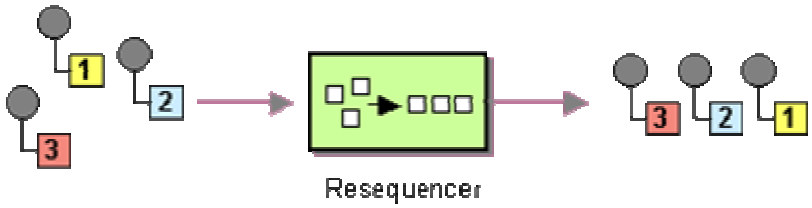
One common way things get out of sequence is the fact that different messages may take different processing paths. Let's look at a simple example. Let's assume we are dealing with a numbered sequence of messages. If all even numbered messages have to undergo a special transformation whereas all odd numbered messages can be passed right through, then odd numbered messages will appear on the resulting channel while the even ones queue up at the transformation. If the transformation is quite slow, all odd messages may appear on the output channel before a single even message makes it, bringing the sequence completely out of order (see picture).



To avoid getting the messages out of order, we could introduce a loop-back (acknowledgment) mechanism that makes sure that only one message at a time passes through the system. The next message will not be sent until the last one is done processing. This conservative approach will resolve the issue, but has two significant drawbacks. First, it can slow the system significantly. If we have a large number of parallel processing units, we would severely underutilize the processing power. In many instances, the reason for parallel processing is that we need to increase performance, so throttling traffic to one message at a time would complete erase the purpose of the solution. The second issue is that this approach requires us to have control over messages being sent into the processing units. However, often we find ourselves at the receiving end of an out-of-sequence message stream without having control over the message origin.

An *Aggregator* can receive a stream of messages, identify related messages and aggregate them into a single message based on a number of strategies. During this process, the *Aggregator* also needs to be able to deal with the fact that individual messages can arrive at any time and in any order. The *Aggregator* solves this problem by storing messages until all related messages arrive before it publishes a result message.

Use a stateful filter, a *Resequencer*, to collect and re-order messages so that they can be published to the output channel in a specified order.



The *Resequencer* can receive a stream of messages that may not arrive in order. The *Resequencer* contains in internal buffer to store out-of-sequence messages until a complete sequence is obtained. The in-sequence messages are then published to the output channel. It is important that the output channel is order-preserving so messages are guaranteed to arrive in order at the next component. Like most other routers, a *Resequencer* usually does not modify the message contents.

Sequence Numbers

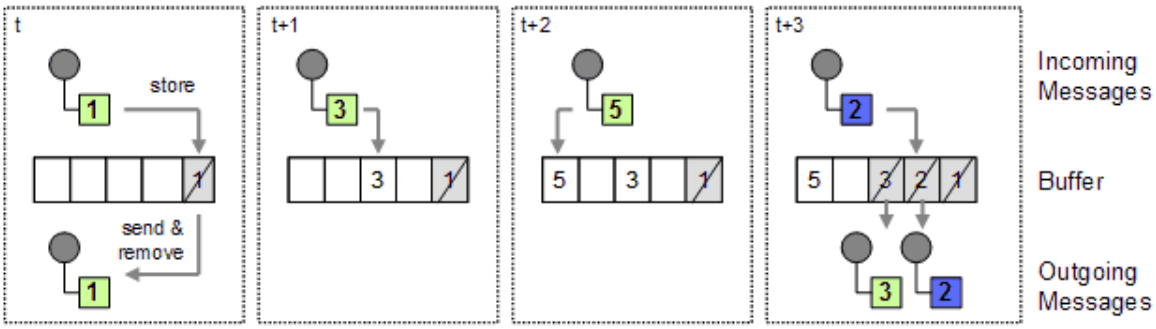
For the *Resequencer* to function, each message has to have a unique Sequence Number (see *Message Sequence*). This sequence number is different from a Message Identifier or Correlation Identifier. A Message Identifier is a special attribute that uniquely identifies each message. However, in most cases Message Identifiers are not comparable, they are basically random values and often times not even numeric. Even if they happen to numerical values it is generally a bad idea to overload the Sequence Number semantics over an existing Message Identifier element. Correlation Identifiers are designed to match incoming messages to original outbound requests. The only requirement for Correlation Identifiers is uniqueness, they do not to have to be numeric or in sequence. So if we need to preserve the order of a series of messages we should define a separate field to track the Sequence Number. Typically, this field can be part of the Message Header.

Generating Sequence Numbers can be more time-consuming than generating unique identifiers. Often times unique identifiers can be generated in a distributed fashion by combining unique location information (e.g., the MAC address of the network interface card) and current time. Most GUID (Globally Unique Identifier) algorithms work this way. To generate in-sequence numbers we generally need a single counter that assigns numbers across the system. In most cases, it is not sufficient for the numbers to be simply in ascending order, but they need to be consecutive as well. Otherwise it will be difficult to identify missing messages. If we are not careful, this Sequence Number generator could easily become a bottleneck for the message flow. If the individual messages are the result of using a *Splitter* it is best to incorporate the numbering right into the *Splitter*. The *Identify Field* pattern in [EAA] contains a useful discussion on how to generate keys and sequence numbers.

Internal Operation

Sequence Numbers ensure that the *Resequencer* can detect messages arriving out of sequence. But what should the *Resequencer* do when a out-of-sequence message arrives? An out-of-sequence

message implies that a message with a higher sequence number arrives before a message with a lower sequence number. The *Resequencer* has to store the message with the higher sequence number until it receives all the "missing" messages with lower sequence numbers. Meanwhile, it may receive other out-of-sequence messages as well, which have to be stored as well. Once the buffer contains a consecutive sequence of messages, the *Resequencer* sends this sequence to the output channel and then removes the sent messages from the buffer (see picture).

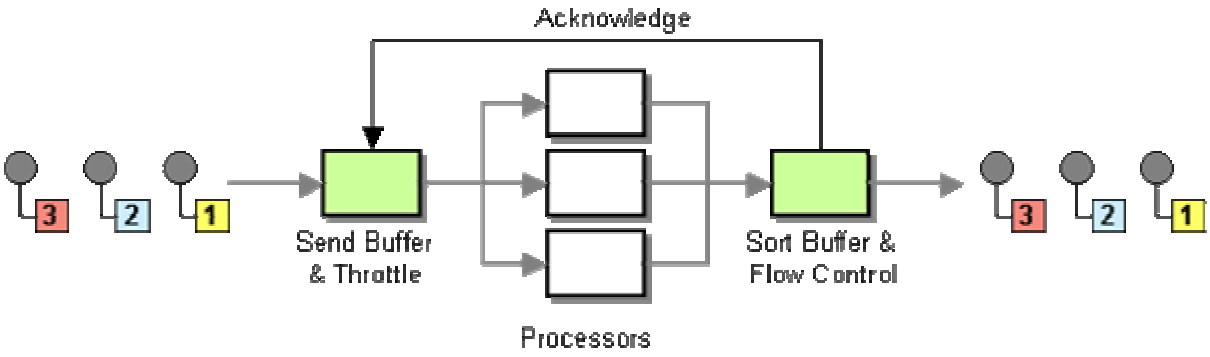


In this simple example, the *Resequencer* receives messages with the sequence numbers 1, 3 ,5 ,2. We assume that the sequence starts with 1, so the first message can be sent right away and removed from the buffer. The next message has the sequence number 3, so we are missing message 2. Therefore, we store message 3 until we have a proper sequence of messages. We do the same with the next message, which has a sequence number of 5. Once message 2 comes in, the buffer contains a proper sequence of the messages 2 and 3. Therefore, the *Resequencer* publishes these messages and removes them from the buffer. Message 5 remains in the buffer until the remaining "gap" in the sequence is closed.

Avoiding Buffer Overrun

How big should the buffer be? If we are dealing with a long stream of messages the buffer can get rather large. Worse yet, let's assume we have a configuration with multiple processing units each of which deals with a specific message type. If one processing unit fails, we will get a long stream of out-of-sequence messages. A buffer-overrun is almost certain. In some cases we can use the message queue to absorb the pending messages. This works only if the messaging infrastructure allows us to read messages from the queue based on selection criteria as opposed to always reading the oldest message first. That way we can poll the queue and see whether the first 'missing' message has come in yet without consuming all the messages in between. At some point, though, even the storage allocated to the message queue will fill up.

One robust way to avoid buffer overruns is to throttle the message producer by using active acknowledgement (see picture).



As we discussed above, sending only a single message at a time is very inefficient. So we need to be a little smarter than that. One way we can be more efficient is for the *Resequencer* to tell the producer how many slots it has available in its buffer. The message throttle can then fire off that many messages since even if they get completely out of order the *Resequencer* will be able to hold all of them in the buffer and re-sequence them. This approach presents a good compromise between efficiency and buffer requirements. However, it does require that we have access to the original in-sequence message stream in order to insert the send buffer and throttle.

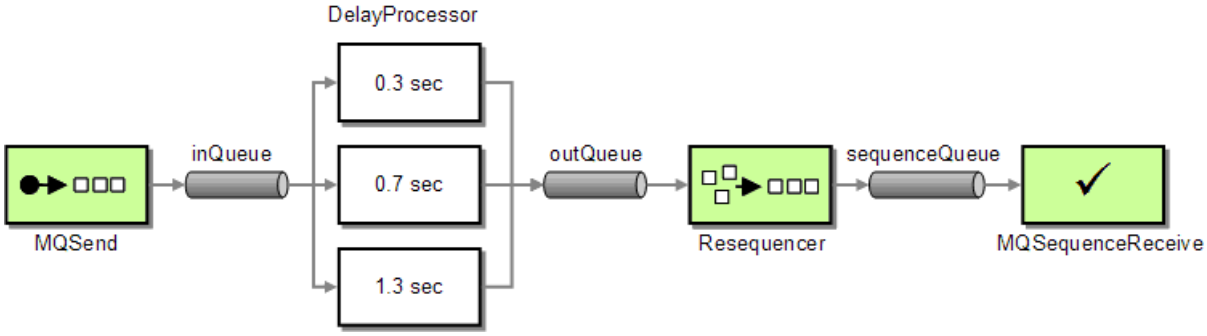
This approach is very similar to the way the TCP/IP network protocol works. One of the key features of the TCP protocol is to ensure in-sequence delivery of packets over the network. In reality, each packet may be routed through a different network path so that out-of-sequence packets occur quite frequently. The receiver maintains a circular buffer that is used as a sliding window. Receiver and sender negotiate on the number of packets to send before each acknowledgement. Because the sender waits for an acknowledgment from the receiver, a fast sender cannot outpace the receiver or cause the buffer to overflow. Specific rules also prevent the so-called Silly Window Syndrome where sender and receiver could fall into a very inefficient one-packet-at-a-time mode.

Another solution to the buffer overrun problem is to compute stand-in messages for the missing message. This works if the recipient is tolerant towards "good enough" message data and does not require precise data for each message or if speed is more important than accuracy. For example, in voice over IP transmissions implement filling in a blank packet results in a better user experience than issuing a re-request for a lost packet.

Most of us application developers take reliable network communication for granted. When designing messaging solutions, it is actually helpful to look into some of the internals of TCP because at its core, IP traffic is asynchronous and unreliable and has to deal with many of the same issues enterprise integration solutions do. For a thorough treatment of IP protocols see [Stevens] and [Wright].

Example: Resequencer in Microsoft .NET with MSMQ

To demonstrate the function of a *Resequencer* in a real-life scenario, we use the following setup:

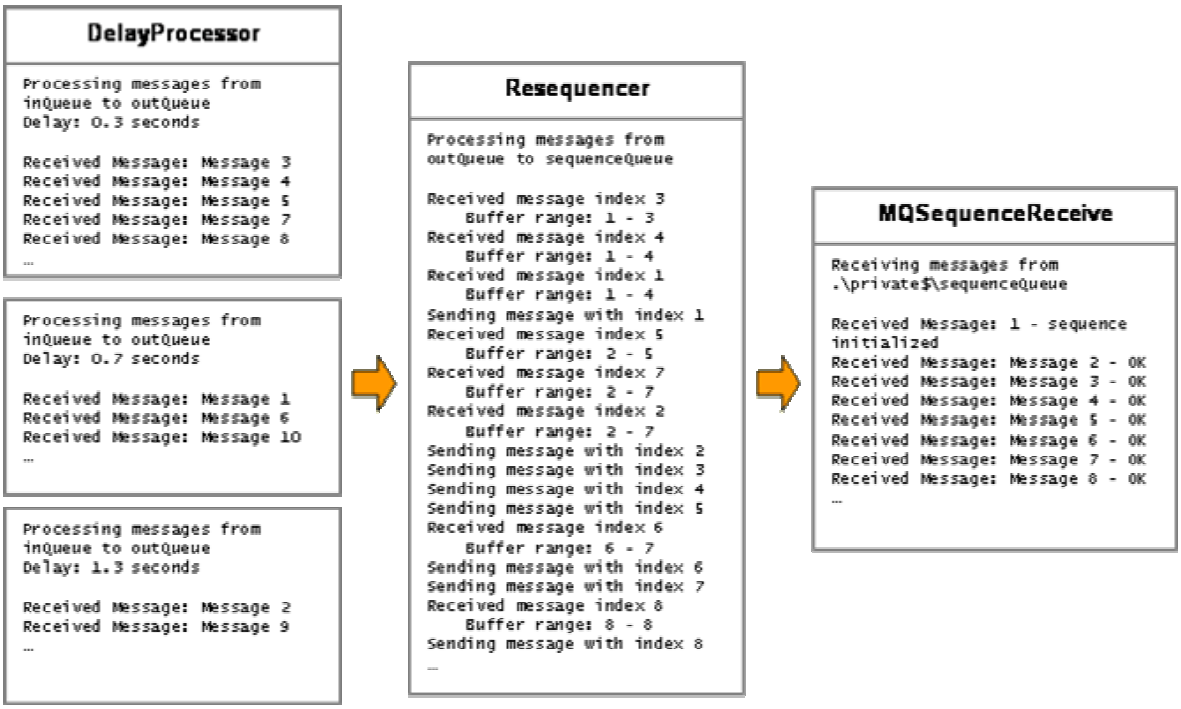


Resequencer Test Configuration

The test setup consists of four main components, each implemented as a C# class. The components communicate via MSMQ message queues, provided by the Message queuing service that is part of Windows 2000 and Windows XP.

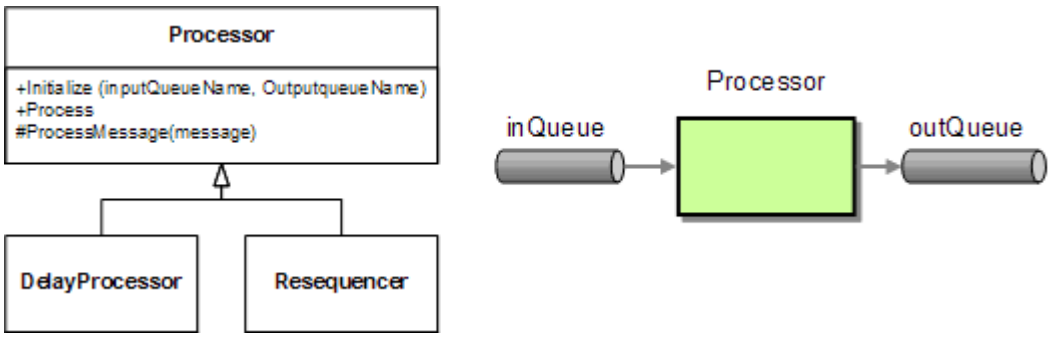
- **MQSend** acts as the *Test Message* generator. The message body contains a simple text string. MQSend equips each message with a sequence number inside the AppSpecific property of each message. The sequence starts with 1 and the number of messages can be passed in from the command line. MQSend publishes the messages to the private queue **inQueue**.
- **DelayProcessor** reads messages off the **inQueue**. The only 'processing' consists of a timed delay before the identical message is republished to the **outQueue**. We use three DelayProcessors in parallel to simulate a load balanced processing unit. The processors act as *Competing Consumers*, so that each message is consumed by exactly one processor. All processors publish messages to the **outQueue**. Because of the different processing speed, messages on the outQueue are out of sequence.
- The **Resequencer** buffers incoming out-of-sequence messages and republishes them in sequence to the **sequenceQueue**.
- **MQSequenceReceive** reads messages off the **sequenceQueue** and verifies that the sequence numbers in the AppSpecific property are in ascending order.

If we fire up all components, we see debug output similar to the following picture. From the size of the processor output windows we can see the different speeds at which the processors are working. As expected, the messages arriving at the *Resequencer* are not in sequence (in this run, the messages arrived as 3, 4, 1, 5, 7, 2 ...). We can see from the *Resequencer* output how the *Resequencer* buffers the incoming messages if a message is missing. As soon as the missing message arrives, the *Resequencer* publishes the now completed sequence.



Output from the test components

Looking at the test setup, we realize that both the DelayProcessor and the Resequencer have a few things in common: they both read messages from an input queue and publish them to an output queue. The only difference is in what happens in between -- the actual processing of the message. Therefore, we created a common base class that encapsulates the basic functionality of this generic *Filter* (see *Pipes and Filters*. It contains convenience and template methods for queue creation and asynchronous receiving, processing and sending of messages. We call this base class `Processor` (see picture).



Both the DelayProcessor and the Resequencer inherit from the common Processor class

The default implementation of the `Processor` simply copies messages from the input queue to the output queue. To implement the `Resequencer`, we have to override the default implementation of the `ProcessMessage` method. In the case of the `Resequencer`, the `processMessage` method adds the received message in the buffer, which is implemented as a `Hashtable`. The messages in the buffer are keyed by the message sequence number which is stored in the `AppSpecific` property. Once the new message is added, the method `SendConsecutiveMessages` checks whether we have a consecutive sequence starting with the next outstanding messages. If so, the method sends all consecutive messages and removes them from the buffer.

Resequencer.cs

```
using System;
using System.Messaging;
using System.Collections;
using MsgProcessor;

namespace Resequencer
{

    class Resequencer : Processor
    {
        private int startIndex = 1;
        private IDictionary buffer = (IDictionary)(new Hashtable());
        private int endIndex = -1;

        public Resequencer(MessageQueue inputQueue, MessageQueue outputQueue) : base
(inputQueue, outputQueue) {}

        protected override void ProcessMessage(Message m)
        {
            AddToBuffer(m);
            SendConsecutiveMessages();
        }

        private void AddToBuffer(Message m)
        {
            Int32 msgIndex = m.AppSpecific;
            Console.WriteLine("Received message index {0}", msgIndex);
            if (msgIndex < startIndex)
            {
                Console.WriteLine("Out of range message index! Current start is: {0}",
startIndex);
            }
            else
            {
                buffer.Add(msgIndex, m);
                if (msgIndex > endIndex)
                    endIndex = msgIndex;
            }
            Console.WriteLine("    Buffer range: {0} - {1}", startIndex, endIndex);
        }

        private void SendConsecutiveMessages()
        {
```

```
            while (buffer.Contains(startIndex))
            {
                Message m = (Message)(buffer[startIndex]);
                Console.WriteLine("Sending message with index {0}", startIndex);
                outputQueue.Send(m);
                buffer.Remove(startIndex);
                startIndex++;
            }
        }
    }

}
```

As you can see, the `Resequencer` assumes that the message sequence starts with 1. This works well if the message producer also starts the sequence from 1 and the two components maintain the same sequence over the lifetime of the components. To make the *Resequencer* more flexible, the message producer should negotiate a sequence start number with the *Resequencer* first before sending the first message of the sequence. This process is analogous to the SYN messages exchanged during the connect sequence of the TCP protocol (see [Stevens]).

The current implementation also has no provisions for a buffer overrun. Let's assume, a DelayProcessor aborts or malfunctions and 'eats' a message. The *Resequencer* will wait indefinitely for the missed message until the buffer overflows. In hi-volume scenarios the message and the *Resequencer* need to negotiate a window size describing the maximum number of messages the *Resequencer* can buffer. Once the buffer is full, an error handler has to determine how to deal with the missing message. For example, the producer could resend the message or a 'dummy' message could be injected.

The `Processor` is relatively simple. It uses asynchronous message processing by using the `BeginReceive` and `EndReceive`methods. Because it is easy to forget to call `BeginReceive` at the end of the message processing, we used a *template method* that incorporates this step. Subclasses can the override the `ProcessMessage` method without having to worry about the asynchronous processing.

Processor.cs

```
using System;
using System.Messaging;
using System.Threading;

namespace MsgProcessor
{
    public class Processor
    {
        protected MessageQueue inputQueue;
```

```
        protected MessageQueue outputQueue;

        public Processor (MessageQueue inputQueue, MessageQueue outputQueue)
        {
            this.inputQueue = inputQueue;
            this.outputQueue = outputQueue;
            inputQueue.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
            inputQueue.MessageReadPropertyFilter.ClearAll();
            inputQueue.MessageReadPropertyFilter.AppSpecific = true;
            inputQueue.MessageReadPropertyFilter.Body = true;
            inputQueue.MessageReadPropertyFilter.CorrelationId = true;
            inputQueue.MessageReadPropertyFilter.Id = true;
            Console.WriteLine("Processing messages from " + inputQueue.Path + " to " +
outputQueue.Path);
        }

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

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

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

            ProcessMessage(m);

            mq.BeginReceive();
        }

        protected virtual void ProcessMessage(Message m)
        {
            string body = (string)m.Body;
            Console.WriteLine("Received Message: " + body);
            outputQueue.Send(m);
        }
```

```
    }
}
```

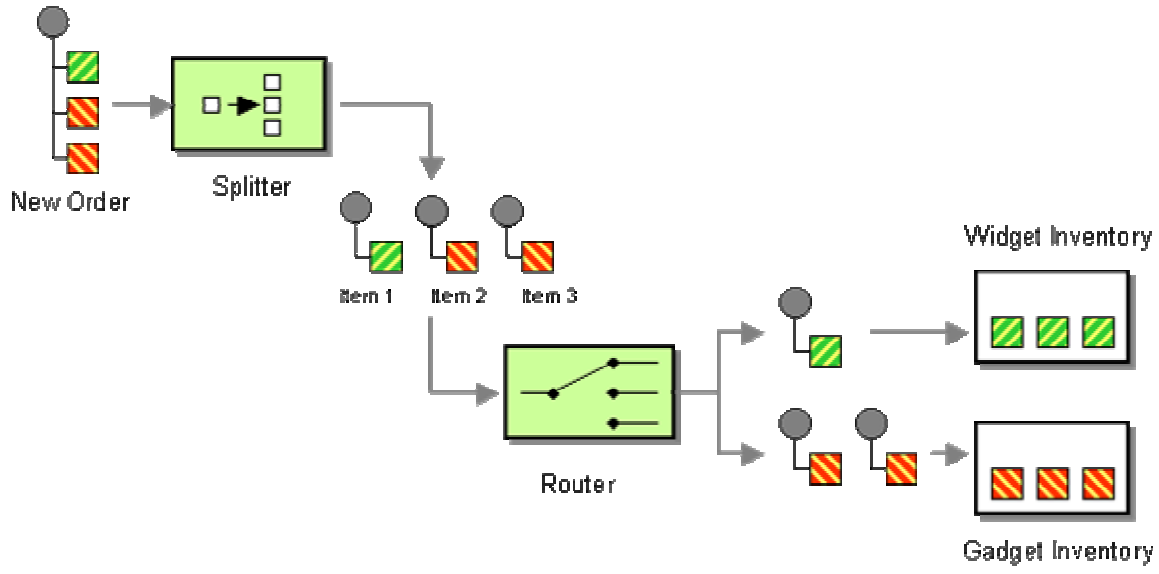
Related patterns: *Aggregator*, *Competing Consumers*, *Message Router*, *Message Sequence*, *Pipes and Filters*, *Splitter*, *Test Message*

Composed Message Processor

The order-processing example presented in the *Content-Based Router* and *Splitter* patterns processes an incoming order consisting of individual line items. Each line item requires an inventory check with the respective inventory system. After all items have been verified we want to pass the validated order message to the next processing step.

How you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?

This problem seems to contain elements of multiple patterns we have already defined. A *Splitter* can split a single message into multiple parts. A *Content-Based Router* could then route individual sub-messages through the correct processing steps based on message content or type. The *Pipes and Filters* architectural style allows us to chain together these two patterns so that we can route each item in the composed message to the appropriate processing steps:



In our example, this means that each order item is routed to the proper inventory system to be verified. The inventory systems are decoupled from each other and each system receives only items that can be processed by it.

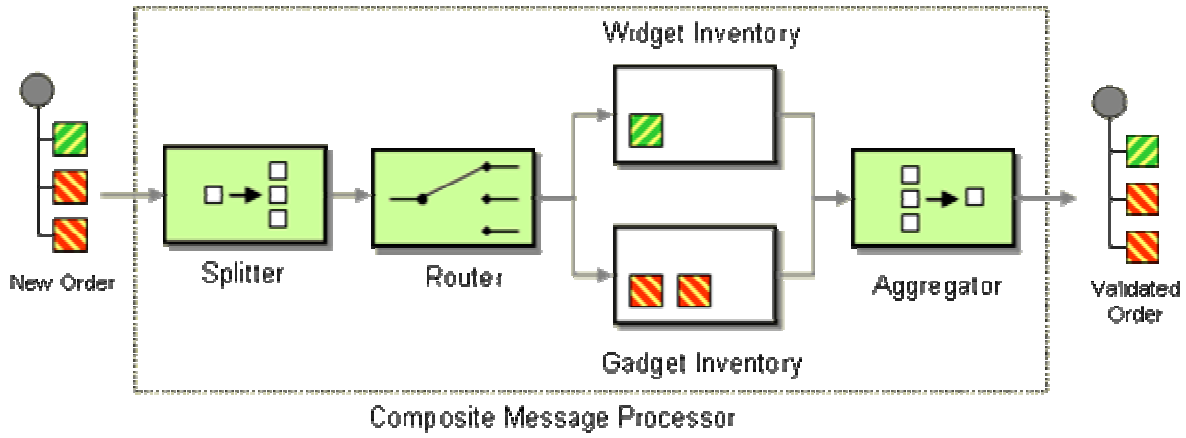
The shortcoming of the setup so far is that we cannot find out whether all items that have been ordered are actually in stock and can be shipped. We also need to retrieve the prices for all items (factoring volume discounts) and assemble them into a single invoice. This requires us to continue processing as if the order is still a single message even though we just chopped it up into many sub-messages

One approach would be to just reassemble all those items that are passing through a specific inventory system into a separate order. This order can be processed as a whole from this point on: the order can be fulfilled and shipped, a bill can be sent. Each sub-order is treated as an independent process. In some instances, lack of control over the downstream process may make this approach the only available solution. For example, Amazon follows this approach for a large portion of the goods it sells. Orders are routed to different fulfillment houses and managed from there.

However, this approach may not provide the best customer experience. The customer may receive more than one shipment and more than one invoice. Returns or disputes may be difficult to accommodate. This is not a big issue with consumers ordering books, but may prove difficult if individual order items depend on each other. Let's assume that the order consists of furniture items that make up a shelving system. The customer would not be pleased to receive a number of huge boxes containing furniture elements just to find out that the required mounting hardware is temporarily unavailable and will be shipped at a later time.

The asynchronous nature of a messaging system makes distribution of tasks more complicated than synchronous method calls. We could dispatch each individual order item and wait for a response to come back before we check the next item. This would simplify the temporal dependencies, but would make the system very inefficient. We would like to take advantage of the fact that each system can process orders simultaneously.

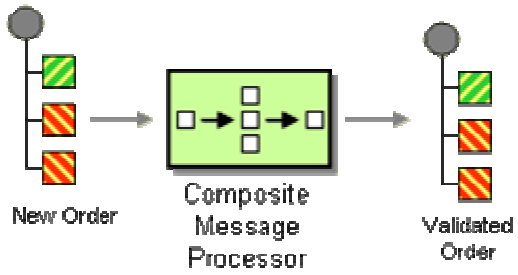
Use *Composed Message Processor* to process a composite message. The *Composed Message Processor* splits the message up, routes the sub-messages to the appropriate destinations and re-aggregates the responses back into a single message.



The *Composed Message Processor* uses an *Aggregator* to reconcile the requests that were dispatched to the multiple inventory systems. Each processing unit sends a response message to the aggregator stating the inventory on hand for the specified item. The Aggregator collects the individual responses and processes them based on a predefined algorithm as described under *Aggregator*.

Because all sub-messages originate from a single message, we can pass additional information, such as the number of submessages to the *Aggregator* to define a more efficient aggregation strategy. Nevertheless, the *Composed Message Processor* still has to deal with issues around missing or delayed messages. If an inventory system is unavailable, do we want to delay processing of all orders that include items from that system? Or should we route them to an exception queue for a human to evaluate manually? If a single response is missing, should we re-send the inventory request message? For a more detailed discussion of these trade-offs see the *Aggregator* pattern.

This patterns demonstrates the composability of individual patterns into a larger patterns. We can combine individual patterns into a larger pattern. To the rest of the system, the *Composed Message Processor* appears like a simple filter with a single input channel and a single output channel. As such, it proivdes an effective abstraction of the more complex internal workings.



The Composite Message Processor as a Single Filter

Related patterns: *Aggregator*, *Content-Based Router*, *Pipes and Filters*, *Splitter*

Scatter-Gather

In the order processing example introduced in the previous patterns , each order item that is not currently in stock could be supplied by one of multiple external suppliers. However, the suppliers may or may not have the respective item in stock themselves, they may charge a different price and may be able to supply the part by a different date. To fill the order in the best way possible, we should request quotes from all suppliers and decide which one provides us with the best term for the requested item.

How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?

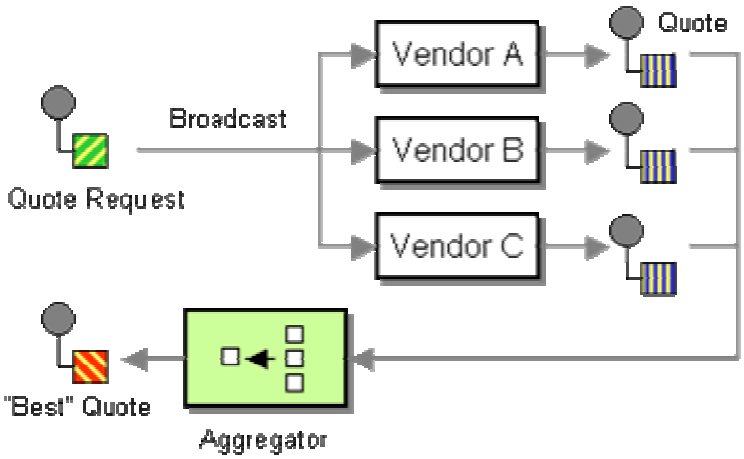
The solution should allow for flexibility in determining the recipients of the message. We can either determine the list of approved suppliers centrally or we can let any interested supplier participate in the 'bid'. Since we have no (or little) control over the recipients, we must be prepared to receive responses from some, but not all recipients. Such changes to the bidding rules should not impact the structural integrity of the solution.

The solution should hide the number and identity of the individual recipients from any subsequent processing. Encapsulating the distribution of the message locally keeps other components independent from the route of the individual messages.

We also need to coordinate the subsequent message flow. The easiest solution might be for each recipient to post the reply to a channel and let subsequent components deal with the resolution of the individual messages. However, this would require subsequent components to be aware of the message being sent to multiple recipients. It might also be harder for subsequent components to process the individual messages without having any knowledge over the routing logic that has been applied.

It makes sense to combine the routing logic, the recipients and the post-processing of the individual messages into one logical component.

Use a *Scatter-Gather* that broadcasts a message to multiple recipients and re-aggregates the responses back into a single message.



The *Scatter-Gather* routes a request message to the a number of recipients. It then uses an *Aggregator* to collect the responses and distill them into a single response message.

There are two variants of the *Scatter-Gather* that use different mechanisms to send the request messages to the intended recipients:

- **Distribution** via a *Recipient List* allows the *Scatter-Gather* to control the list of recipients but requires the *Scatter-Gather* to be aware of each recipient's message channel.
- **Auction**-style *Scatter-Gather*s use a *Publish-Subscribe Channel* to broadcast the request to any interested participant. This option allows the *Scatter-Gather* to use a single channel but at the same time relinquishes control.

The solution shares similarities with the *Composed Message Processor*. Instead of using a *Splitter*, we broadcast the complete message to all involved parties using a *Publish-Subscribe Channel*. We will most likely add a *Return Address* so that all responses can be processed through a single channel. As with the *Composed Message Processor*, responses are aggregated based on defined business rules. In our example, the *Aggregator* might take the best bids from suppliers that can fill the order. Aggregating the responses can be more difficult with a *Scatter-Gather* as compared to the *Composed Message Processor* because we may not know how many recipients participate in the interaction.

Both the *Scatter-Gather* and the *Composed Message Processor* route a single message to multiple recipients and combine the individual reply messages back into a single message by using an *Composed Message Processor*. The *Composed Message Processor*performs the task of synchronizing multiple parallel activities. If the individual activities take widely varying amounts of time this results in subsequent processing being held up even though many subtasks (or even all but one) have been completed. This consideration needs to be weighed against the simplicity and

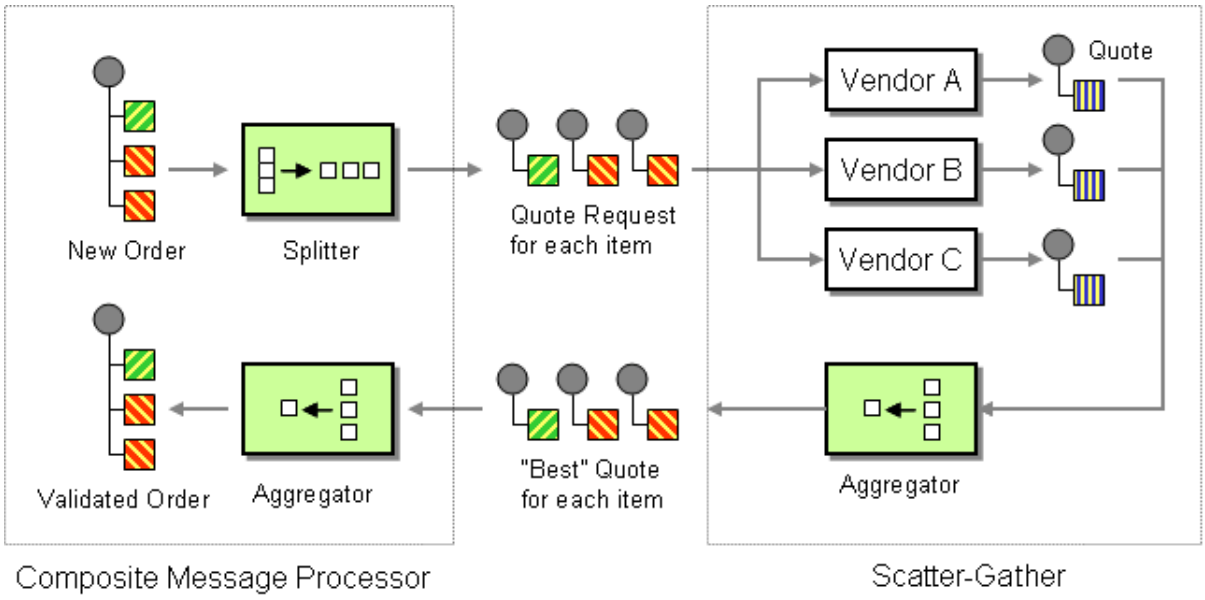
encapsulation the *Scatter-Gather* brings. A compromise between the two choices may be a cascading *Aggregator*. This design allows subsequent tasks to be initiated with only a subset of the results being available.

Example: Loan Broker

The Loan Broker example (see *Introduction to Composed Messaging Examples*) uses a *Scatter-Gather* to route requests for a loan quote to a number of banks and select the best offer from the incoming responses. The example implementations demonstrate both a solution based on a *Recipient List* (see *Asynchronous Implementation with MSMQ* and a *Publish-Subscribe Channel* (see *Asynchronous Implementation with TIBCO ActiveEnterprise*).

Example: Combining Patterns

We can now use the *Scatter-Gather* to implement the widget and gadget order processing example. We can combine the *Scatter-Gather* with the *Composed Message Processor* to process each incoming order, sequence it into individual items, then pass each item up for a bid, then aggregate the bids for each item into a combined bid response, and lastly aggregate all bid responses into a complete quote. This is a very real example how multiple integration patterns can be combined into a complete solution. The composition of individual patterns into larger patterns allows us to discuss the solution at a higher level of abstraction. It also allows us to modify details of the implementation without affecting other components.



Combining a Scatter-Gather and a Composite Message Processor

This example also shows the versatility of the *Aggregator*. The solution uses two *Aggregator*s for quite different purposes. The first *Aggregator* (part of the *Scatter-Gather*) chooses the best bid from a number of vendors. This aggregator may not require a response from all vendors (speed may

be more important than a low price) but may require a complex algorithm to combine responses. For example, the order may contain 100 widgets and the lowest price supplier has only 60 widgets in stock. The *Aggregator* needs to be able to decide whether to accept this offer and fill the remaining 40 items from another supplier. The second *Aggregator* (part of the *Composed Message Processor*) might be simpler because it simply concatenates all responses received from the first *Aggregator*. However, this *Aggregator* needs to make sure that all responses are in fact received and needs to deal with error conditions such as missing item responses.

Related patterns: *Aggregator*, *Introduction to Composed Messaging Examples*, *Asynchronous Implementation with MSMQ*, *Asynchronous Implementation with TIBCO ActiveEnterprise*, *Composed Message Processor*, *Publish-Subscribe Channel*, *Recipient List*, *Return Address*, *Splitter*

Routing Slip

Most of the routing patterns presented in this section route incoming messages to one or more destinations based on a set of rules. Sometimes, though, we need to route a message not just to a single component, but through a whole series of components. Let's assume, for example, that we use a *Pipes and Filters* architecture to process incoming messages that have to undergo a sequence of processing steps and business rule validations. Since the nature of the validations varies widely and may depend on external systems (e.g., credit card validations), we implement each type of step as a separate filter. Each filter inspects the incoming message, and applies the business rule(s) to the message. If the message does not fulfill the conditions specified by the rules it is routed to an exception channel. The channels between the filters determine the sequence of validations that the message needs to undergo.

Now let's assume, though, that the set of validations to perform against each message depends on the message type (for example, purchase order request do not need credit card validation or customers who send orders over a VPN may not require decryption and authentication). To accommodate this requirement we need to find a configuration that can route the message through a different sequence of filters depending on the type of the message.

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?

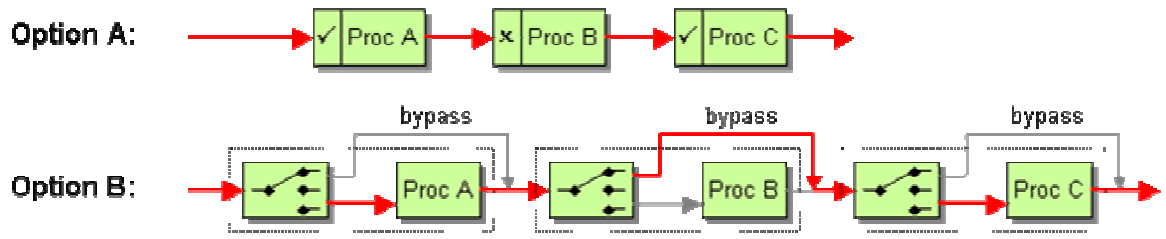
The *Pipes and Filters* architectural style give us an elegant approach to represent a sequence of processing steps as independent filters, connected by pipes (channels). In its default configuration, the filters are connected by fixed pipes. If we want to allow message to be routed to different filters dynamically, we can use special filters that act as *Message Router*s. The routers dynamically determine the next filter to route the message to.

The key requirements to a good solution to our problem can be summarized as follows:

- **Efficient message flow** - Messages should only flow through the required steps and avoid unnecessary components.

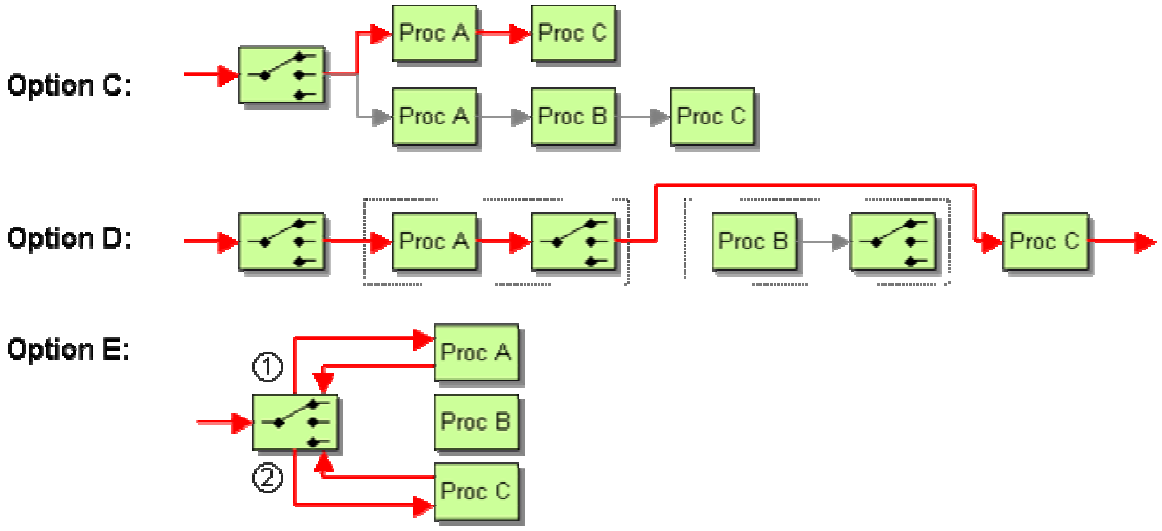
- **Efficient use of resources** - The solution should not use a huge amount of channels, routers and other resources.
- **Flexible** - The route that individual messages that should be easy to change.
- **Simple to maintain** - If a new type of message needs to be supported, we would like to have a single point of maintenance to avoid introducing errors.

The following diagram illustrates our alternative solutions to the problem. We are assuming that the system offers three separate processing steps A, B, and C and that the current message is required to pass only through steps A and C. The actual flow for this example message is marked with thick, red arrows.



We could form one long *Pipes and Filters* chain of all possible validation steps and add code to each router to bypass the validation if the step is not required for the type of message being passed through (see Option A). This option employs the reactive filtering approach described in the *Message Filter*. While the simplicity of this solution is appealing, the fact that the components blend both business logic (the validation) and routing logic (deciding whether to validate) will make them harder to reuse. Also, it is conceivable that two types of messages undergo similar processing steps but in different order. This hard-wired approach would not easily support this requirement.

In order to improve separation of concerns and increase the composability of the solution we should replace the 'gating' logic inside each component with *Content-Based Router*s. We would then arrive at a chain of all possible validation steps, each prefixed by a *Content-Based Router* (see Option B). When a message arrives at the router, it checks the type of the message and determines whether this type of message requires the validation step at hand. If the step is required, the router routes the message through the validation. If the step is not required, the router bypasses the validation and routes the message directly to the next router. This configuration works quite well in cases where each step is independent from any other step and the routing decision can be made locally at each step. On the downside, this approach ends up routing the message though a long series of routers even though only a few validation steps may be executed. In fact, each message will be transmitted across a channel at a rate of 2 times the number of possible components. If we have a large component library, this will cause an enormous amount of message flow for a rather simple function. Also, the routing logic is distributed across many filters, making it hard to understand which validation steps a message of a specific type will actually undergo. Likewise, if we introduce a new message type we may have to update each and every router. Lastly, this option suffers from the same limitation as Option A in that messages are tied to executing steps in a common order.



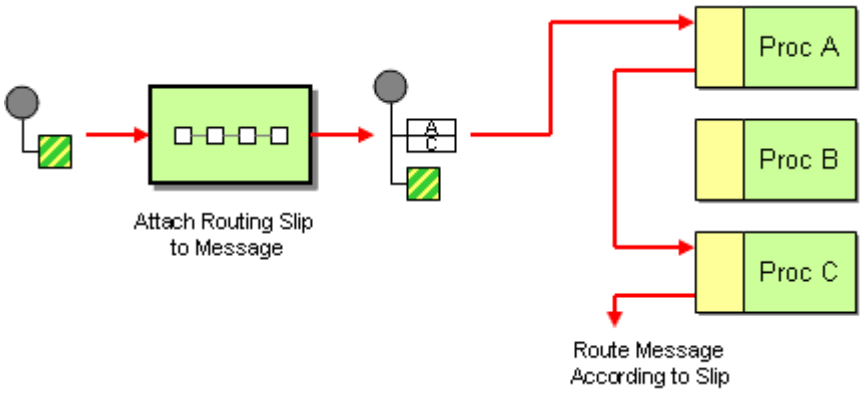
If we desire a central point of control, an up-front *Content-Based Router* tended to be a good choice in our prior pattern discussions. We could envision a solution where we setup individual *Pipes and Filters* chains for each message type. The chain would contain the sequence of validations relevant to the specific type. We would then use a *Content-Based Router* to route the incoming message to the correct validation chain based on message type (see Option C). This approach routes messages only through the relevant steps (plus the initial router). Thus, is is the most efficient approach so far because we only add a single routing step in order to implement the desired functionality. The solution also highlights the path that a message of a specific type will take quite nicely. However, it requires us to hard-wire any possible combination of validation rules. Also, the same component may be used in more than one path. This approach would require us to run multiple instances of such components which leads to unnecessary duplication. For a large set of message types, this approach could result in a maintenance nightmare because of the large number of component instances and associated channels that we have to maintain. In summary, this solution is very efficient at the expense of maintainability.

If we want to avoid hard-wiring all possible combinations of validation steps, we need to insert a *Content-Based Router* between each validation step (see Option D). In order not to run into the same issues associated with the reactive filtering approach (presented in Option B), we would insert the *Content-Based Router* *after* each step instead of before (we need one additional router in front of the very first step to get started). The routers would be smart enough to relay the message directly to the next *required* validation step instead of routing it blindly to the next *available* step in the chain. In the abstract, this solution looks similar to the reactive filtering approach because the message traverses an alternating set of routers and filters. However, in this case the routers possess more intelligence than a simple yes/no decision which allows us to eliminate unnecessary steps. For example, in our simple scenario, the message passes only through 2 routers as opposed to 3 with Option B. This option provides efficiency and flexibility, but does not solve our goal of obtaining central goal-- we still have to maintain a potentially large number of routers because the routing logic is spread out across a series of independent routers .

To address this last shortcoming we could combine all routers into a single 'super-router' (See Option E). After each validation step, the message would be routed back to the super-router who

would determine the next validation step to be executed. Since all the routing decisions are now incorporated into a single router, we need to devise a mechanism to remember which steps we already finished processing. Therefore, the super-router would have to be stateful or each filter would have to attach a tag to the message telling the super-router the name of the last filter the message went through. Also, we are still dealing with the fact that each validation step requires the message to be passed through two channels: to the component and the back to the super-router. This results in about two times as much traffic as Option C.

Attach a *Routing Slip* to each message, specifying the sequence of processing steps. Wrap each component with a special message router that reads the *Routing Slip* and routes the message to the next component in the list.



We insert a special component into the beginning of the process that computes the list of required steps for each message. It then attaches the list as a *Routing Slip* to the message and starts the process by routing the message to the first processing step. After successful processing, each processing step looks at the *Routing Slip* and passes the message to the next processing step specified in the routing table.

This pattern works very similar to the routing slip attached to a magazine for circulation in a department. The only difference is that the *Routing Slip* has a defined sequence of components it traverses whereas in most companies you can hand the magazine after reading it to any person on the list who has not read it (of course, the boss usually comes first).

The *Routing Slip* combines the central control of the 'super-router' approach (Option E) with the efficiency of the hard-wired solutions (Option C). We determine the complete routing scheme up-front and attach it to the message so we do not have to return to the central router for further decision making. Each component is augmented by simple routing logic. In the proposed solution we assume that this routing logic is built into the processing component itself. If we look back at Option A, we remember that we dismissed this approach partway because we had to hard-code some logic into each component. How is the *Routing Slip* better? The key difference is that the router used in the *Routing Slip* is generic and does not have to change with changes in the routing logic. The routing logic incorporated into each component is similar to a *Return Address* where the return address is selected from a list of addresses. Similar to the *Return Address*, the components retain their reusability and composability even though a piece of routing logic is

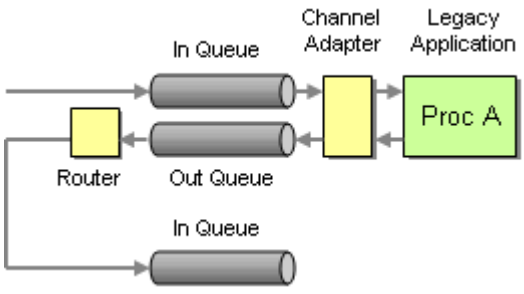
built into the component. Additionally, the computation of the routing table can now be done in a central place without ever touching the code inside any of the processing components.

As always, there is no free lunch. So we can expect the *Routing Slip* to have some limitations. First, the message size increases slightly. In most cases this should be insignificant, but we need to realize that we are now carrying process state (which steps have been completed) inside the message. This can cause other side effects. For example, if we lose a message we lose not only the message data but also the process data (i.e., where the message was going to go next). In many cases it may be useful to maintain the state of all messages in a central place to perform reporting or error recovery.

Another limitation of the *Routing Slip* is the fact that the path of a message cannot be changed once it is under way. This implies that the message path cannot depend on intermediate results generated by a processing step along the way. In many real-life business processes the message flow does change based on intermediate results, though. For example, depending on the availability of the ordered items (as reported by the inventory system) we may want to continue with a different path. This also means that a central entity has to be able to determine all steps a message should undergo in advance. This can lead to some brittleness in the design, similar to the concerns about using a *Content-Based Router*.

Implementing a Routing Slip with Legacy Applications

The *Routing Slip* assumes that we have the ability to augment the individual components with the router logic. If we are dealing with legacy applications or packaged applications we may not be able to influence the functionality of the component itself. Rather, we need to use an external router that communicates with the component via messaging. This inevitably increases the number of channels and components in usage. However, the *Routing Slip* still provides the best trade-off between our goals of efficiency, flexibility and maintainability.



Implementing a Routing Slip with Legacy Applications

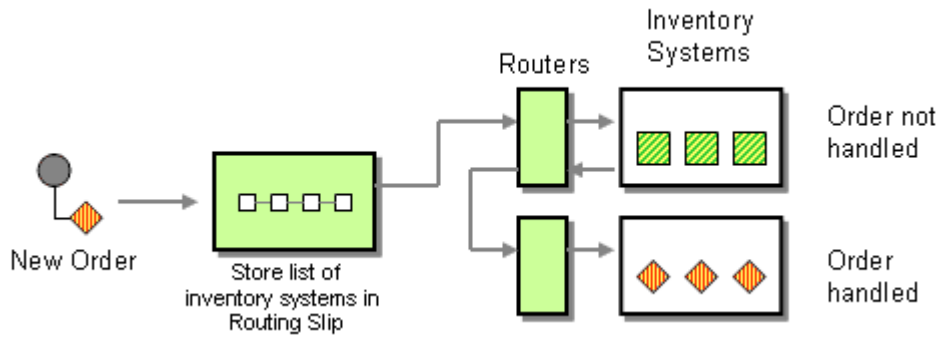
Common Usage

The *Routing Slip* is most useful in the following scenarios:

- A sequence of binary validation steps. By not adding information to the message, the limitation that we cannot change the routing once the message is underway is no longer a factor. We still appreciate the flexibility to change the sequence of validation steps by reconfiguring the central *Routing Slip*. Each component has the choice between aborting the sequence due to error or to pass the message on to the next step.
- Each step is a stateless transformation. For example, let's assume that we receive orders from a variety of business partners. All orders arrive on a common channel. Depending in the partner, the message may require different transformation steps. Messages from some partners may require decryption, others may not. Some may require transformation or enrichment while others may not. Keeping a *Routing Slip* for each partner gives us an easy way to reconfigure the steps for each partner in a central location.
- Each step gathers data, but makes no decisions. In some cases, we receive a message that contains reference identifiers to other data. For example, if we receive an order for a DSL line, the message may contain only the home phone number of the applicant. We need to go to external sources to determine the customer's name, the central office servicing the line, the distance from the central office etc. Once we have a complete message with all relevant data we can decide what package to offer to the customer. In this scenario the decision is postponed until the end so we can use a *Routing Slip*. In this scenario we need to assess though whether we really require the flexibility of the *Routing Slip*. Otherwise a simple hard-wired chain of *Pipes and Filters* may be sufficient.

Implementing a Simple Router with a Routing Slip

One of the downsides of a *Content-Based Router* was that it has to incorporate knowledge about each possible recipient and the routing rules associated with that recipient. Under the spirit of loose coupling it may be undesirable to have a central component that incorporates knowledge about many other components. An alternative solution to the *Content-Based Router* was a *Publish-Subscribe Channel* combined with an array of *Message Filter*s. This solution allows each recipient to decide which messages to process but suffered from risk of duplicate message processing. Another option to enable individual recipients to decide whether to process a given message is to use a modified version of a *Routing Slip* acting as a *Chain of Responsibility* as described in [GoF]. The *Chain of Responsibility* allows each component to accept a message or route it to the next component in the list. The *Routing Slip* is a static list of all participants. This still implies that a central component has to have knowledge of all possible recipients. However, the component does not need to know which messages each component consumes.



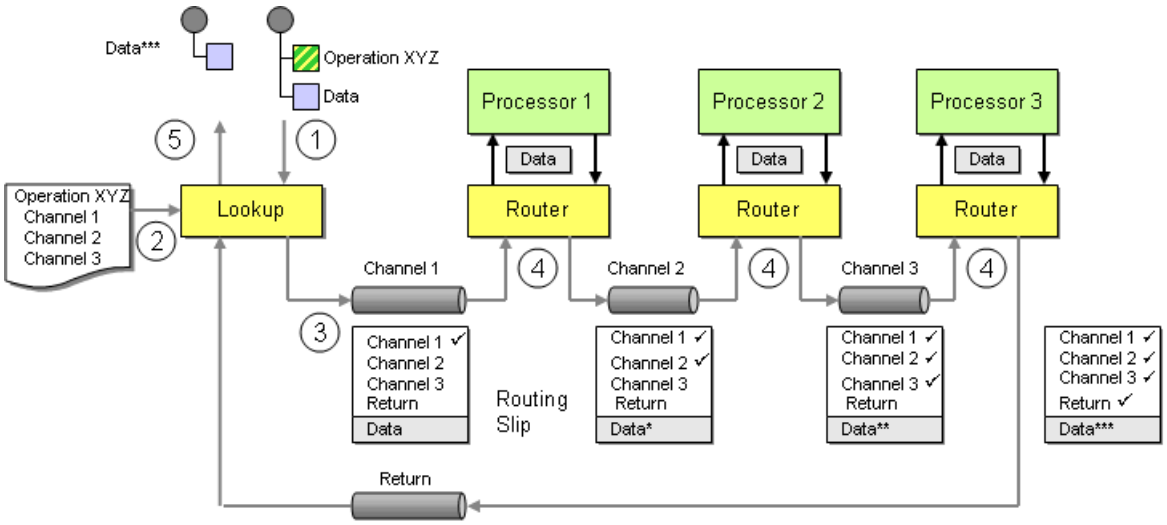
Using a *Routing Slip* avoids the risk of duplicate message processing. Likewise it is easy to determine if a message was not processed by any component. The main trade-off is the slower processing and increased network traffic. While a *Content-Based Router* publishes a single message regardless of the number of systems, the *Routing Slip* approach publishes an average number of messages equal to 1/2 the number of systems. We can reduce this number if we can arrange the systems in such a way that the first systems to receive the message have a higher chance of handling the message, but the number of messages will likely remain higher than with a predictive *Content-Based Router*.

There are cases where we need more control than a simple sequential list or we need to change the flow of a message based on intermediate results. The *Process Manager* can fulfill these requirements because it supports branching conditions, forks and joins. In essence, the *Routing Slip* is a special case of a dynamically configured business process. The trade-offs between using a *Routing Slip* and using a central *Process Manager* should be carefully examined. A dynamic *Routing Slip* combines the benefits of a central point of maintenance with the efficiency of a hard-wired solution. However, as the complexity grows, analyzing and debugging the system may become increasingly difficult as the routing state information is distributed across the messages. Also, as the semantics of the process definition begin to include constructs such as decisions, forks and joins, the configuration file may become hard to understand and maintain. We could include conditional statements inside the routing table and augment the routing modules in each component to interpret the conditional commands to determine the next routing location. We need to be careful, though, to not overburden the simplicity of this solution with additional functionality. Once we require this type of complexity it may be a good idea to give up the run-time efficiency of a *Routing Slip* and to start using a much more powerful *Process Manager*.

Example: Routing Slip as a Composed Service

When creating a service-oriented architecture, a single logic function is often composed of multiple independent steps. This situation occurs commonly for two primary reasons. First, packaged applications tend to expose fine-grained interfaces based on their internal APIs. When integrating these packages into an integration solution, we want to work at a higher level of abstraction. For example, the operation "New Account" may require multiple steps inside a billing system: create a new customer, select a service plan, set address properties, verify credit data etc. Second, a single logical function may be spread across more than one system. We want

to hide this fact from other systems so we have the flexibility to reassign responsibilities between systems without affecting the the rest of the integration solution. We can easily use a *Routing Slip* to execute multiple internal steps for a single request message. The *Routing Slip* gives us the flexibility to execute different requests from the same channel. The *Routing Slip* executes the sequence of individual steps but appears to the outside like a single step (see picture).



1. The incoming request message, specifying the intended operation and any necessary data, is sent to the lookup component.
2. The lookup component retrieves the list of required processing steps associated with the intended operation from a service directory. It adds the list of channels (each channel equals one fine-grained operation) to the message header. The lookup component adds the return channel to the list so that completed messages are returned to the lookup component.
3. The lookup component publishes the message to the channel for the first activity.
4. Each router reads the request from the queue and passes it to the service provider. After the execution, the router marks the activity as completed and routes the message to the next channel specified in the routing table.
5. The lookup component consumes the message off the return channel and forwards it to the requestor. To the outside, this whole process appears like a simple request-reply message exchange.

Example: WS-Routing

Frequently, a Web service request has to be routed through multiple intermediaries. For this purpose, Microsoft defined the Web Services Routing Protocol (WS-Routing) specification. WS-Routing is a SOAP-based protocol for routing messages from a sender through a series of intermediaries to a receiver. The semantics of WS-Routing are richer than those of the *Routing Slip*, but a *Routing Slip* can be easily implemented in WS-Routing. The following example shows the SOAP header for a message that is routed from node A to node D via the intermediaries B and C.

```
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://www.w3.org/2001/06/soap-envelope">
  <SOAP-ENV:Header>
     <wsrp:path xmlns:wsrp="http://schemas.xmlsoap.org/rp/">
        <wsrp:action>http://www.im.org/chat</wsrp:action>
        <wsrp:to>soap://D.com/some/endpoint</wsrp:to>
        <wsrp:fwd>
           <wsrp:via>soap://B.com</wsrp:via>
           <wsrp:via>soap://C.com</wsrp:via>
        </wsrp:fwd>
        <wsrp:from>soap://A.com/some/endpoint</wsrp:from>
        <wsrp:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</wsrp:id>
     </wsrp:path>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
     ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Like most services specs, WS-Routing is likely to evolve over time and/or be merged with other specs. We included the example here as a snapshot of where the Web Services community is going with respect to routing.

Related patterns: *Content-Based Router*, *Message Filter*, *Message Router*, *Pipes and Filters*, *Process Manager*, *Publish-Subscribe Channel*, *Return Address*

Process Manager

The *Routing Slip* demonstrates how a message can be routed through a dynamic series of processing steps. The solution of the *Routing Slip* is based on two key assumptions: the sequence of processing steps has to be determined up-front and the sequence is linear. In many cases, these assumptions may not be fulfilled. For example, routing decisions might have to be made based on intermediate results. Or, the processing steps may not be sequential, but multiple steps might be executed in parallel.

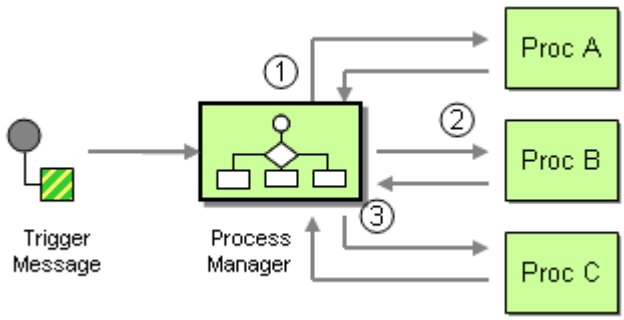
How do we route a message through multiple processing steps when the required steps may not be known at design-time and may not be sequential?

One of the primary advantages of a *Pipes and Filters* architectural style is the composability of individual processing units ("filters") into a sequence by connecting them with channels ("pipes"). Each message is then routed through the sequence of processing units (or components). If we need to be able to change the sequence for each message, we can use multiple *Content-Based Router*s. This solution provides the maximum flexibility, but has the disadvantage that the routing logic is spread across many routing components. The *Routing Slip* provides a central

point of control by computing the message path up-front, but does not provide the flexibility to re-route the message based on intermediate results or to execute multiple steps simultaneously.

We can gain flexibility and maintain a central point of control, if after each individual processing unit we return control back to a central component. That component can then determine the next processing unit(s) to be executed. Following this approach, we end up with an alternating process flow: central component, processing unit, central component, processing unit and so on. As a result, the central unit receives a message after each individual processing step. When the message arrives, the central component has to determine the next processing step(s) to be executed based on intermediate results and the current 'step' in the sequence. This would require the individual processing units to return sufficient information to the central unit to make this decision. However, this approach would make the processing units dependent on the existence of the central unit because they might have to pass through extraneous information that is not relevant to the processing unit, but only to the central component. If we want to decouple the individual processing steps and the message formats from the central unit, we need to provide the central unit with some form of 'memory' that tells it what step in the sequence was executed last.

Use a central processing unit, a *Process Manager*, to maintain the state of the sequence and determine the next processing step based on intermediate results.



First of all, let me clarify that the design and configuration of a *Process Manager* is a pretty extensive topic. We could probably fill a whole book (Volume 2, maybe?) with patterns related to the design of workflow or business process management. Therefore, this pattern is intended primarily to "round off" the topic of routing patterns and to provide a pointer into the direction of workflow and process modeling. By no means is it a comprehensive treatment of business process design.

Using a *Process Manager* results in a so-called hub-and-spoke pattern of message flow (see diagram). An incoming message initializes the *Process Manager*. We call this message the *trigger message*. Based on the rules inside the *Process Manager*, it sends a message (1) to the first processing step, implemented by Processing Unit A. After unit A completes its task it sends a reply message back to the *Process Manager*. Next, the *Process Manager* determines the next step to be executed and sends message (2) to the next processing unit. As a result, all message traffic runs through this central 'hub', hence the term hub-and-spoke. The downside of this central control element is the danger of turning the *Process Manager* into a performance bottleneck.

The versatility of a *Process Manager* is at the same time its biggest strength and weakness. A *Process Manager* can execute any sequence of steps, sequential or in parallel. Therefore, almost any integration problem can be solved with a *Process Manager*. Likewise, most of the patterns introduced in this chapter could be implemented using a *Process Manager*. In fact, many EAI vendors make you believe that every integration problem is a process problem. We think that using a *Process Manager* for every situation may be overkill. It can distract from the core design issue and also cause significant performance overhead.

Managing State

One of the key functions of the *Process Manager* is to maintain state between messages. For example, when the second processing unit returns a message to the *Process Manager*, the *Process Manager* needs to remember that this is step 2 out of a sequence of many steps. We do not want to tie this knowledge to the processing unit because the same unit may appear multiple times inside the same process. For example, Processing Unit B may be step 2 and step 4 of a single process. As a result, the same reply message sent by Processing Unit B may trigger the *Process Manager* to execute step 3 or step 5, based on the process context. To accomplish this without complicating the processing unit, the *Process Manager* needs to maintain the current position in the process execution.

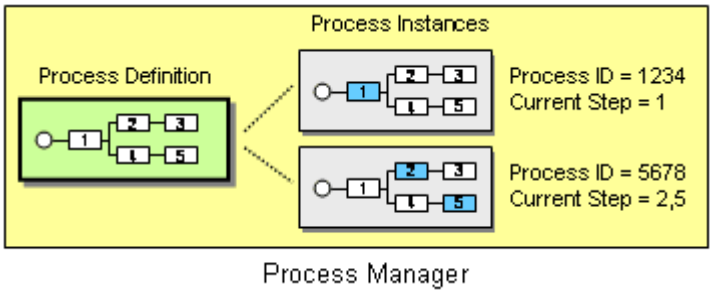
It is useful for the *Process Manager* to be able to store additional information besides the current position in the process. The *Process Manager* can store intermediate results from previous processing if it is relevant to later steps. For example, if the results of step 1 are relevant to a later step, the *Process Manager* can store this information without burdening the messages to and from subsequent processing units with this data back. This allows the individual processing steps to be independent of each other because they do not have to worry about data produced or consumed by other units. Effectively, the *Process Manager* plays the role of a *Claim Check* explained later.

Process Instances

Because the process execution can span many steps and can therefore take a long time, the *Process Manager* needs to be prepared to receive new messages while another process is still executing. In order to manage multiple parallel executions, the *Process Manager* creates a new *process instance* for each incoming trigger message. The process instance stores the state associated with the execution of the process triggered by the trigger message. The state includes the current execution step of the process and any associated data. Each process instance is identified by a unique process identifier.

It is important to separate the concepts of a *process definition* (also referred to as process template) and a process instance. The process definition is a design construct that defines the sequence of steps to be executed. The process instance is an active execution of a specific template. The diagram below shows a simple example with one process definition and two process instances.

The first instance (process identifier 1234) is currently executing step 1 while the second process instance (process identifier 5678) is executing steps 2 and 5 in parallel.

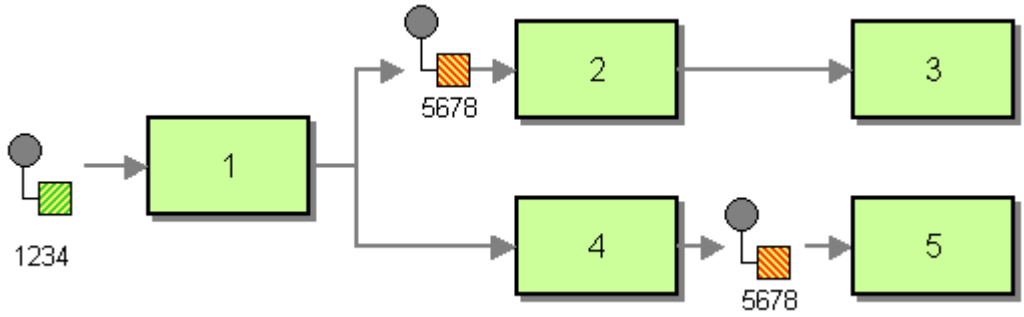


Multiple Process Instances based on one Process Definition

Because multiple process instances may be executing simultaneously, the *Process Manager* needs to be able to associate an incoming message with the correct instance. For example, if the *Process Manager* in the above example receives a message from a processing unit, which process instance is the message meant for? Multiple instances may be executing the same step so the *Process Manager* cannot derive the instance from the channel or the type of message. The requirement to associate an incoming message with a process instance reminds us of the *Correlation Identifier*. The *Correlation Identifier* allows a component to associate an incoming reply message with the original request by storing a unique identifier in the reply message that correlates it to the request message. Using this identifier the component can match up the reply with the correct request even if the component has sent multiple requests and the replies arrive out of order. The *Process Manager* requires a similar mechanism. When the *Process Manager* receives a message from a processing unit, it needs to be able to associate the message with the process instance that sent the message to the processing unit. The *Process Manager* needs to include a *Correlation Identifier* inside messages that it sends to processing units. The component needs to return this identifier in the reply message as a *Correlation Identifier*.

Keeping State in Messages

It is apparent that state management is an important feature of the *Process Manager*. How, then, did the previous patterns get away without managing state? In a traditional *Pipes and Filters* architecture, the pipes (i.e., the *Message Channel*s) manage the state. To continue the example above, if we were to implement the process with hard-wired components connected by *Message Channel*, it would look like the picture below. If we assume that this system is in the same state as the example above (i.e. two process instances), it equates to one message with the identifier 1234 sitting in a channel waiting to be processed by component (1) and two messages with the identifier 5678 waiting to be processed by the components (2) and (5) respectively. As soon as component (1) consumes the message and completes its processing tasks, it broadcasts a new message to components (2) and (4) -- exactly the same behavior as the *Process Manager* in the above example.



Keeping State in Channels

It is striking how much the message flow notation used for this example resembles an UML activity diagram that is often used to model the behavior of *Process Manager* components. Effectively, we can use an abstract notation to model the behavior of the system during design, and then decide whether we want to implement the behavior as a distributed pipes-and-filters architecture or as a hub-and-spoke architecture using a central *Process Manager*. Even though we don't have room in this book to dive too deeply into the design of process models, many of the patterns in this language do apply when designing a process model.

As with most architectural decisions, implementing a central *Process Manager* or a distributed pipes-and-filters architecture is not a simple yes/no decision. In many cases, it makes most sense to use multiple *Process Manager* components, each of which houses a particular aspect of a larger process. The *Process Manager* components can then communicate with each other through a pipes-and-filters architecture.

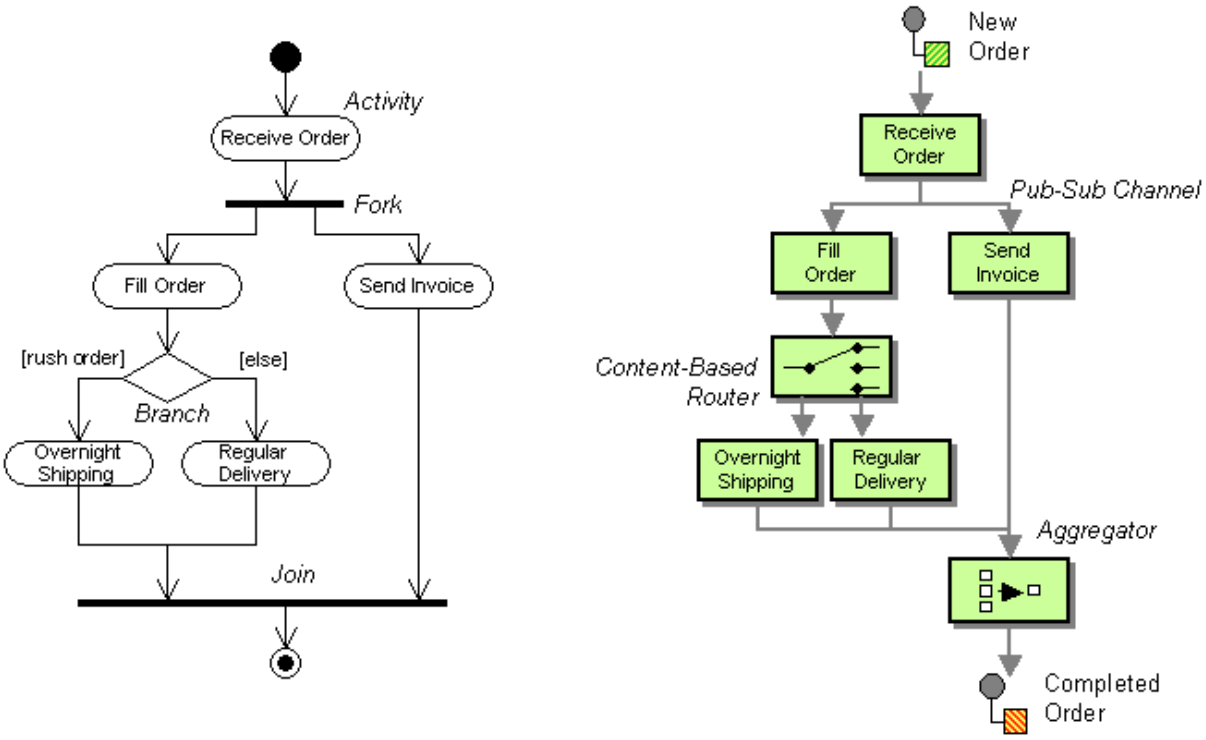
Managing state explicitly inside a *Process Manager* may require a more complex component, but it allows much more powerful process reporting. For example, most implementations of a *Process Manager* provide the ability to query process instance state. This makes it easy to see how many orders are currently waiting for approval or have been put on hold because of lacking inventory. We can also tell each customer the status of his or her order. If we used hard-wired channels, we would have to inspect all channels to obtain the same information. This property of the *Process Manager* is not only important for reporting, but also for debugging. Using a central *Process Manager* makes it easy to retrieve the current state of a process and the associated data. Debugging a fully distributed architecture can be a lot more challenging and is almost impossible without the assistance of such mechanisms as the *Message History* or *Message Store*.

Creating the Process Definition

Most commercial EAI implementations include a *Process Manager* component, combined with visual tools to model the process definition. Most visual tools use a notation that resembles UML Activity Diagrams because the semantics of a *Process Manager* and those of an activity diagram are fairly similar. Also, activity diagrams are a good visual representation of multiple tasks executing in parallel. Until recently, most vendor tools converted the visual notation into an internal, vendor-proprietary process definition to be executed by the process engine. However, the push to standardize various aspects of distributed systems under the umbrella of Web

services has not ignored the important role of process definitions. Three proposed 'languages' have emerged as a result of these efforts. Microsoft defined XLANG which is supported by its family of BizTalk orchestration modeling tools. IBM drafted the WSFL, the Web Services Flow Language [WSFL]. Recently, both companies have joined forces to create the specification for BPEL4WS, the Business Process Execution Language for Web Services (see [BPEL4WS]). The BPEL4WS is a powerful language that describes a process model as an XML document. The intent is to define a standardized intermediate language between the process modeling tools and the *Process Manager* engines. This way, I could model my processes with Vendor X's product and decide to execute the process on Vendor Y's process engine implementation. For more information on the impact of Web services standards in integration, see *Emerging Standards and Futures in Enterprise Integration*.

The semantics of a process definition can be described in rather simple terms. The basic building block is an activity (sometimes called task or action). Usually, an activity can send a message to another component, wait for an incoming message or execute a specific function internally (e.g, a *Message Translator*). Activities can be connected in serial fashion or multiple activities can be executed in parallel using a *fork* and *join* construct. A fork allows multiple activities to execute at the same time. It is semantically equivalent to a *Publish-Subscribe Channel* in a hard-wired pipes-and-filters architecture. A join synchronizes multiple parallel threads of execution back into a single thread. Execution after a join can only continue if all parallel threads have completed their respective activities. In the pipes-and-filters style an *Aggregator* often serves this purpose. The process template also needs to be able to specify a *branch*, or decision point, so that the path of execution can change based on the content of a message field. This function is equivalent to a *Content-Based Router*. Many modeling tools include the ability to design a loop construct, but this is really a special case of the branch. The following picture highlights the semantic similarities between an process definition (depicted as an UML Activity Diagram) and a pipes-and-filters implementation using the patterns defined in this pattern language, even though the physical implementation is very different.



Example UML Activity Diagram and Corresponding Pipes-And-Filters Implementation

Comparing the Process Manager against Other Patterns

A number of times we have contrasted a basic *Pipes and Filters* architecture, the *Routing Slip* and the *Process Manager*. We compiled the key differences into the following table to highlight the trade-offs involved in choosing the correct architecture.

Distributed *Pipes and Filters*	***Routing Slip***	**Central *Process Manager***
Supports complex message flow	Supports only simple, linear flow	Supports complex message flow
Difficult to change flow	Easy to change flow	Easy to change flow
No central point of failure	Potential point of failure (compute routing table)	Potential point of failure (*Process Manager*)
Efficient distributed run-time architecture	Mostly distributed	Hub-and-spoke architecture may lead to bottleneck
No central point of administration and reporting	Central point of administration, but not reporting	Central point of administration and reporting

A central point of control and state management can also mean a central point of failure or a performance bottleneck. For this reason, most *Process Manager* implementations allow persistent storage of process instance state in a file or in a database. The implementation can then leverage redundant data storage mechanisms typically implemented in enterprise-class database systems. It is also common to run multiple *Process Manager* in parallel. Parallelizing *Process Manager* is generally easy because process instances are independent from each other. This allows us to

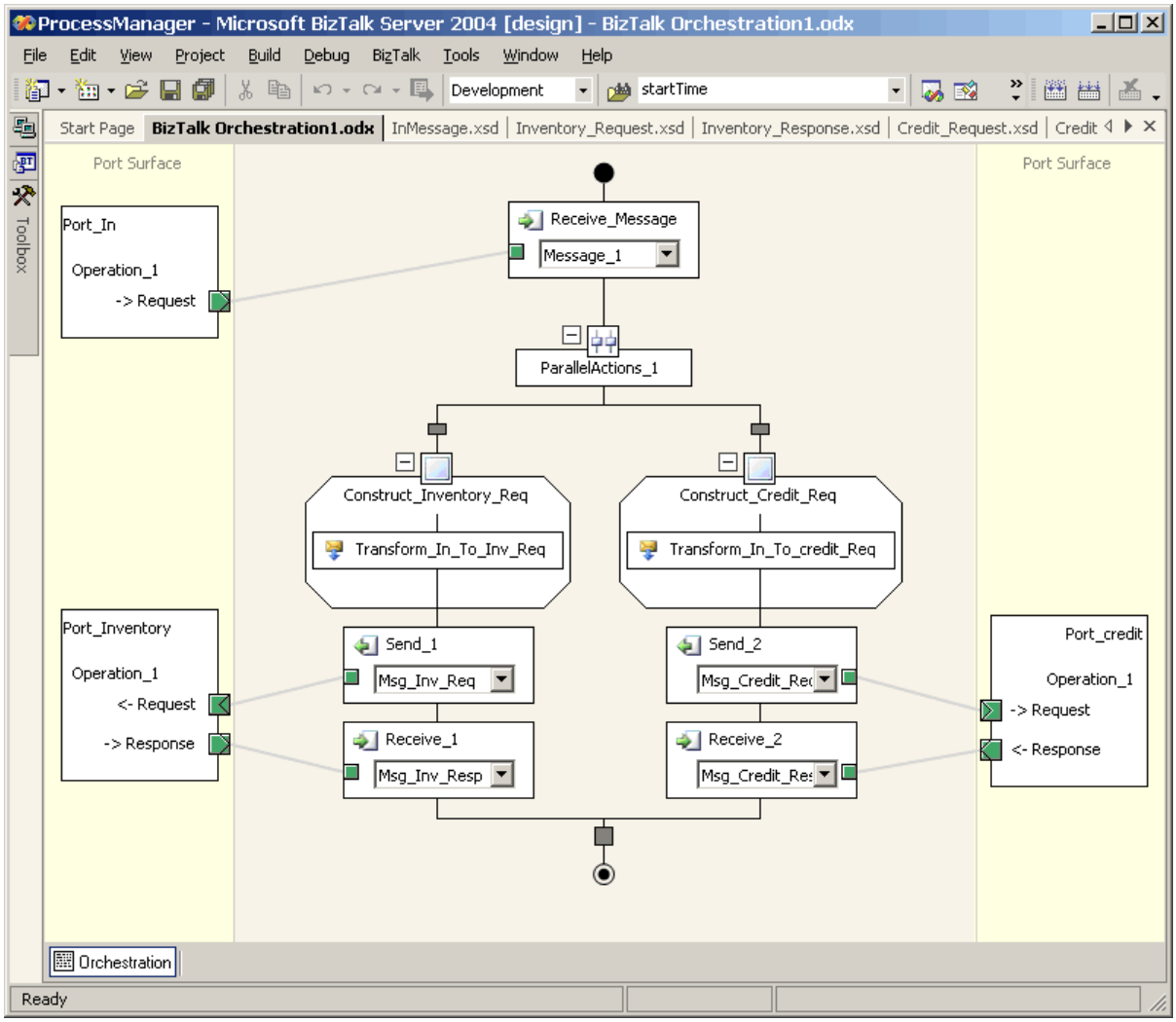
distribute process instances across multiple process engines. If the process engine persists all state information in a shared database, the system can become robust enough to survive the failure of a process engine -- another engine can simply pick up where the previous one left of. The downside of this approach is that the state of each process instance has to be persisted in a central database after each processing step. This could easily turn the database into a new performance bottleneck. As so often, the architect has to find the correct balance between between performance, robustness, cost and maintainability.

Example: Loan Broker

The MSMQ implementation of the Loan Broker example at the end of this chapter (see *Asynchronous Implementation with MSMQ*) implements a simple *Process Manager*. This example creates the *Process Manager* functionality from scratch, defining a process manager and process instances. The TIBCO implementation of the same example (see *Asynchronous Implementation with TIBCO ActiveEnterprise*) uses a commercial process management tool.

Example: Microsoft BizTalk Orchestration Manager

Most commercial EAI tools include process design and execution capabilities. For example, Microsoft BizTalk lets users design process definitions via the Orchestration Designer tool that is integrated into the Visual Studio .NET programming environment.



Microsoft BizTalk 2004 Orchestration Designer

This simple example orchestration receives an order message and executes two parallel activities. One activity creates a request message to the inventory systems and the other activity creates a request message to the credit system. Once both responses are received the process continues. The visual notation makes it easy to follow the process definition.

Related patterns: *Aggregator*, *Asynchronous Implementation with MSMQ*, *Asynchronous Implementation with TIBCO ActiveEnterprise*, *Content-Based Router*, *Correlation Identifier*, *Emerging Standards and Futures in Enterprise Integration*, *Message Channel*, *Message History*, *Message Store*, *Message Translator*, *Pipes and Filters*, *Publish-Subscribe Channel*, *Routing Slip*, *Claim Check*

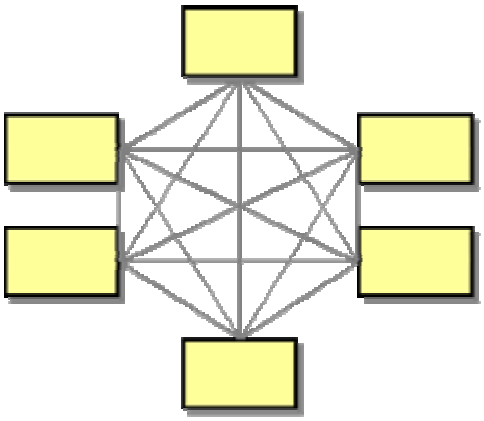
Message Broker

Many patterns in this chapter present ways to route messages to the proper destination without the originating application being aware of the ultimate destination of the message. Most of the patterns focused on specific types of routing logic. However, in aggregate, these patterns solve a bigger problem.

How can you decouple the destination of a message from the sender and maintain central control over the flow of messages?

Using a simple *Message Channel* already provides a level of indirection between sender and receiver -- the sender only knows about the channel, but not the receiver. However, if each receiver has its own channel, this level of indirection becomes less meaningful. Instead of knowing the receiver's address, the sender has to know the correct channel name that is associated with the receiver.

All but the most trivial messaging solutions connect a number of different applications. If we created individual message channels to connect each application to each other application, the number of channels in the system would quickly explode into an unmanageable number, resulting in integration spaghetti (see diagram).



Integration Spaghetti as a Result of Point-to-Point Connections

This diagram illustrates that direct channels between individual applications can lead to an explosion of the number of channels and reduce many of the benefits of routing messages through channels in the first place. These types of integration architectures are often a result of a solution that grew over times. First the customer care system had to talk to the accounting system. Then the customer care system was also expected to retrieve information from the inventory system and the shipping system was to update the accounting system with the shipping charges. It is easy to see how "adding one more piece" can quickly compromise the overall integrity of solution.

Requiring an application to explicitly communicate with each other application can quickly hamper the maintainability of the system. For example, if the customer address changes in the customer care system this system would have to send a message to all systems that maintain copies of the customer address. Every time a new system is added, the customer care system would have to know whether that system uses addresses and be changed accordingly.

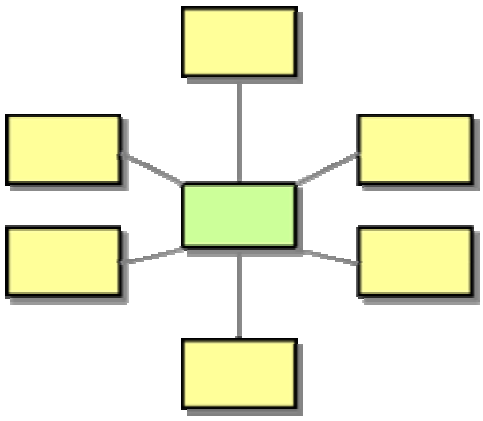
*Publish-Subscribe Channel*s provide some form of basic routing -- the message is routed to each application that subscribed to the specific channel. This works in simple 'broadcast' scenarios but often times routing rules are much more complicated. For example, an incoming order message may have to be routed to a different system based on the size or the nature of the order. To avoid

making the applications responsible for determining a message's ultimate destination, the middleware should include a *Message Router* that can route messages to the appropriate destination.

Individual message routing patterns have helped us decouple the sender from the receiver(s). For example, a *Recipient List* can help pull the knowledge about all recipients out of the sender and into the middleware layer. Moving the logic into the middleware layer helps us in two ways. First, many of the commercial middleware and EAI suites provide tools and libraries that are specialized to performing these kind of tasks. This simplifies the coding effort because we do not have to write the *Message Endpoint*-related code, such as *Event-Driven Consumer*s or thread management. Also, implementing the logic inside the middleware layer allows us to make the logic "smarter" than would be practical inside of the application. For example, using a *dynamic Recipient List* can avoid coding changes when new systems are added to the integration solution.

However, having a large number of individual *Message Router* components can be almost as hard to manage as the integration spaghetti we were trying to resolve.

Use a central *Message Broker* that can receive messages from multiple destinations, determine the correct destination and route the message to the correct channel. Implement the internals of the *Message Broker* using the design patterns presented in this chapter.



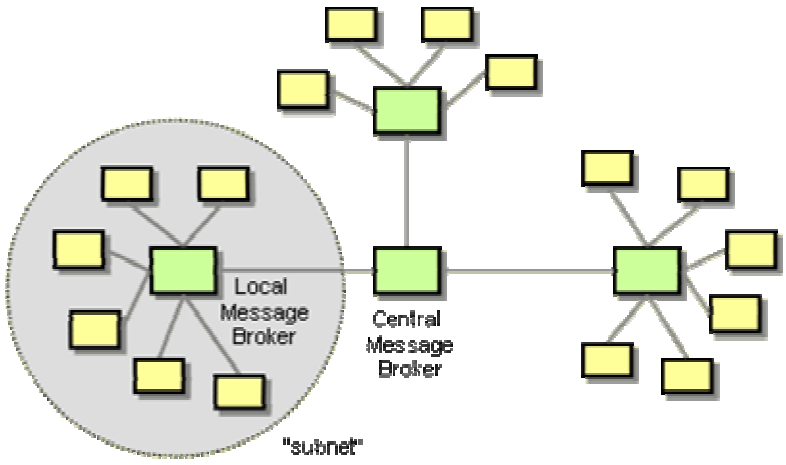
Using a central *Message Broker* is sometimes referred to as *hub-and-spoke* architectural style, which appears to be a descriptive name when looking at the diagram above.

The *Message Broker* pattern has a slightly different scope than most of the other patterns presented in this chapter. It is an architecture pattern as opposed to individual design patterns we presented in this chapter. As such, it is comparable to the *Pipes and Filters* architectural style, which gives us a fundamental way of chaining components together to form more complex message flows. Rather than just chaining individual components, the *Message Broker* concerns itself with larger solutions and helps us deal with the inevitable complexity of managing such a system.

The *Message Broker* is not a monolithic component. Internally, it uses many of the message routing patterns presented in this chapter. So once you decide to use the *Message Broker* as an

architectural pattern, you can choose the correct *Message Router* design patterns to implement the *Message Broker*.

The advantage of central maintenance of a *Message Broker* can also turn into a disadvantage. Routing all messages through a single *Message Broker* can turn the *Message Broker* into a serious bottleneck. A number of techniques can help us alleviate this problem. For example, the *Message Broker* pattern only tells us to *develop* a single entity that performs routing. It does not prescribe how many instances of this entity we deploy in the system at deployment time. If the *Message Broker* design is stateless (i.e. if it is composed only of stateless components), we can easily deploy multiple instances of the broker to improve throughput. The properties of a *Point-to-Point Channel* ensure that only one instance of the *Message Broker* consumes any incoming message. Also, as in most real-life situations, the ultimate solution ends up being a combination of a patterns. Likewise, in many complex integration solutions it may make sense to design multiple *Message Broker* components, each specializing on a specific portion of the solution. This avoids creating the über-*Message Broker* that is so complex as to become unmaintainable. The apparent flip-side is that we no longer have a single point of maintenance and could create a new form of "*Message Broker* spaghetti". One excellent architectural style to use a combination of *Message Broker*s is a *Message Broker hierarchy* (see picture). This configuration resembles a network configuration composed out of individual subnets. If a message has to travel only between two applications inside a "subnet" the local *Message Broker* can manage the routing of the message. If the message is destined for another subnet, the local *Message Broker* can pass the message to the central *Message Broker* who then determines the ultimate destination. The central *Message Broker* performs the same functions as a local *Message Broker*, but instead of decoupling individual applications it decouples whole subsystems consisting of multiple applications.



A Hierarchy of Message Brokers Provides Decoupling while Avoiding the "Über-Broker"

Because the purpose of the *Message Broker* is to reduce coupling between individual applications, it usually has to deal with translating message data formats between applications. Having a *Message Broker* abstract the routing of the message does not help the sending application if it has to format the message in the (supposedly hidden) destination's message format. The next chapter introduces a series of message transformation patterns to address these issues. In many cases, a *Message Broker* uses a *Canonical Data Model* internally to avoid the N-square problem (the number

of translators required to translate between each and every recipient in a system grows with the square of the number of participants).

Example: Commercial EAI Tools

Most commercial EAI tools provide tools to greatly simplify the creation of *Message Broker* components for integration solutions. These tool suites typically provide a number of features that support the development and deployment of *Message Broker*s:

- **Built-in Endpoint code**. Most EAI suites incorporate all code to send and receive messages to and from the message bus. The developer does not have to concern itself with writing any of the transport-related code.
- **Visual Design tools**. These tools allow the developer to compose the functionality of a *Message Broker* using visual components, such as Routers, Decision Points, Transformers. These tools make the flow of message visually intuitive and can reduce the coding effort for many of these components to single-lines of code, e.g. an evaluation function or a rule.
- **Runtime support**. Most EAI packages also provide sophisticated run-time support in both deploying the solution and monitoring the traffic flowing through the *Message Broker*.

Related patterns: *Canonical Data Model*, *Event-Driven Consumer*, *Message Channel*, *Message Endpoint*, *Message Router*, *Pipes and Filters*, *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Recipient List*

8. Message Transformation

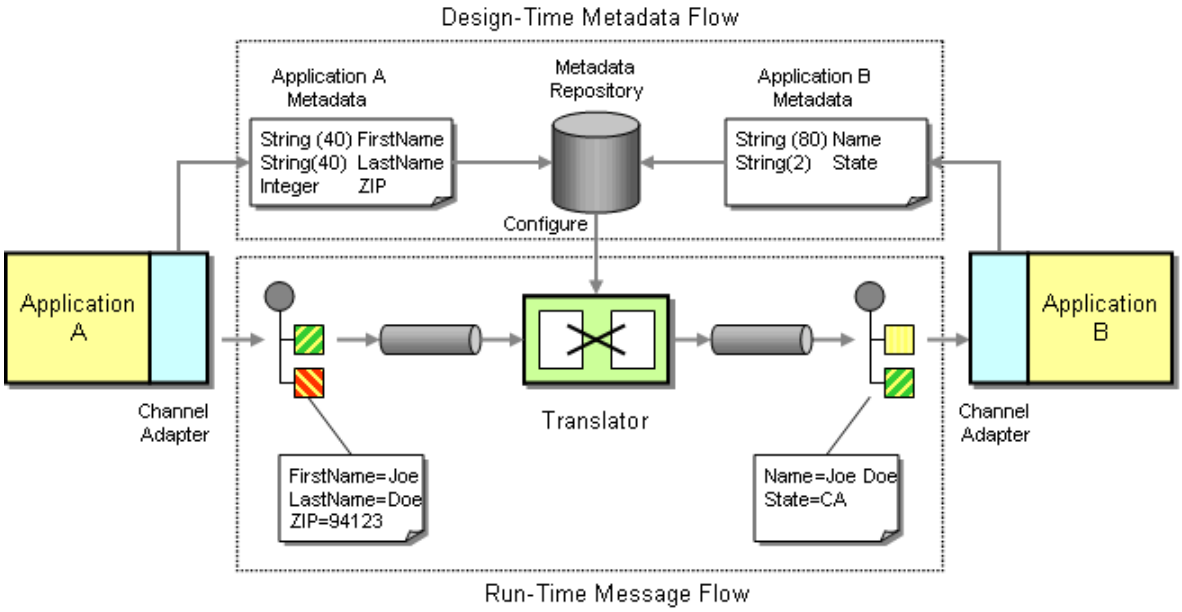
Introduction

As described in the *Message Translator*, applications that need to be integrated by a messaging system rarely agree on a common data format. For example, an accounting system is going to have a different notion of a `Customer` object than a customer relationship management system. On top of that, one system may persist data in a relational model, while another application uses flat files or XML documents. Integrating existing applications often times means that we do not have the liberty of modifying the applications to work more easily with other systems. Rather, the integration solution has to accommodate and resolve the differences between the varying systems. The *Message Translator* pattern offers a general solution to such differences in data formats. This chapter explores specific variants of the *Message Translator*.

Most messaging systems place specific requirements on the format and contents of a message header. We wrap message payload data into an *Envelope Wrapper* that is compliant with the requirements of the messaging infrastructure. Multiple *Envelope Wrapper*s can be combined if a message is passed through across different messaging infrastructures.

A *Content Enricher* is needed if the target system requires data fields that the originating system cannot supply. It has the ability to look up missing information or compute it from the available data. The *Content Filter* does the opposite -- it removes unwanted data from a message. The *Claim Check* also removes data from a message but stores it for later retrieval. The *Normalizer* translates messages arriving in many different formats into a common format.

Message transformation is a deep topic in integration. *Message Channel*s and *Message Router*s can remove basic dependencies between applications by eliminating the need for one application to be aware of the other's location. One application can send a message to a *Message Channel* and worry about what application will consume it. However, message formats impose another set of dependency. If one application has to format messages in another application's data format, the decoupling in form of the *Message Channel* is somewhat of an illusion. Any change to the receiving application or the switch from one receiving application to another still requires a change to the sending application. *Message Translator*s help remove this dependency. Due to the importance of message formats and the transformation between them, we can view any integration solution as two parallel systems. One deals with actual message data, the other one with metadata, the data that describes message data. Many of the patterns used in creation of the message flow can also be used to manage metadata. For example, a *Channel Adapter* can not only move messages in an out of a system, but it can also extract metadata from external applications and load it into a central metadata repository. Using this repository the integration developers can define transformations between the application metadata and the *Canonical Data Model*.



Metadata Integration

For example, the picture above depicts the integration between two applications that need to exchange customer information. Each system has a slightly different definition of customer data. Application A stores first and last name in two separate fields whereas Application B stores it in one field. Likewise, Application A stores the customer's ZIP code and not the state while Application B stores only the state code. Messages flowing from Application A to Application B have to undergo a transformation so that Application B can receive data in the required format. Creating the transformation is much simplified if the *Channel Adapter*s can also extract metadata, e.g. data describing the message format. This metadata can then be loaded into a repository, greatly simplifying the configuration and validation of the *Message Translator*. The metadata can be stored in a variety of formats. A common format used for XML messages are XSD's -- XML Schema Definitions. Other EAI tools implement proprietary metadata formats, but allow administrators to import and export of metadata into different formats.

Many of the principles incorporated in these patterns are applicable outside of messaging integration. For example, *File Transfer* has to perform transformation functions between systems. Likewise, *Remote Procedure Invocation* has to make requests in the data format specified by the service that is to be called even if the application's internal format is different.

This chapter describes variations of data transformation tasks. It does not go into the details of structural transformations between entities (e.g., how to you transform between two data models if one model supports many-to-many relationships between customer and address but the other model includes address fields on the customer record). One of the oldest and still most relevant books on the topic of data representation and relationships is [Kent].

Envelope Wrapper

Most messaging systems divide the message data into a header and a body. The header contains fields that are used by the messaging infrastructure to manage the flow of messages. However, most endpoint systems that participate in the integration solution generally are not aware of these extra data elements. In some cases, systems may even consider these fields as erroneous

because they do not match the message format used by the application. On the other hand, the messaging components that route the messages between the applications may require the header fields and would consider a message invalid if it does not contain the proper header fields.

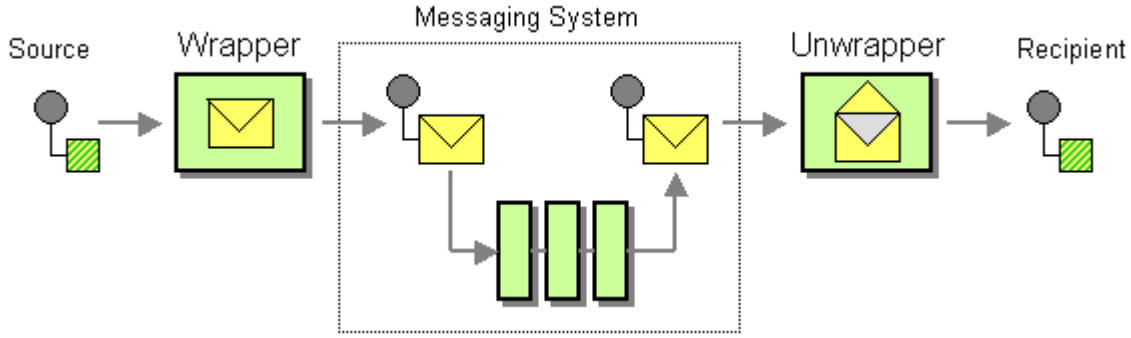
How can existing systems participate in a messaging exchange that places specific requirements on the message format, such as message header fields or encryption?

For example, assume the messaging system is using a proprietary security scheme. A valid message would have to contain security credentials for the message to be accepted for processing by other messaging components. Such a scheme is useful to prevent unauthorized users from feeding messages into the system. Additionally, the message content may be encrypted to prevent eavesdropping by unauthorized listeners -- a particularly important issue with publish-subscribe mechanisms. However, existing applications that are being integrated via the messaging systems are most likely not aware of the concepts of user identity or message encryption. As a result, 'raw' messages need to be translated into messages that comply with the rules of the messaging system.

Some large enterprises use more than one messaging infrastructure. As a result, a message may have to be routed across messaging systems using a *Messaging Bridge*. Each messaging system is likely to have different requirements for the format of the message body as well as the header. This scenario is a case where we can learn by looking at existing TCP/IP-based network protocols. In many cases, connectivity to another system is restricted to a specific protocol, for example telnet or secure shell. In order to enable communication using another protocol (for example, FTP), that protocol format has to be encapsulated into packets that conform to the supported protocol. At the other end, the packet payload can be extracted. This process is called 'tunneling'.

When one message format is encapsulated inside another, the system may lose access to the information inside the data payload. Most messaging systems allow components (for example, a *Content-Based Router*) to access only data fields that are part of the defined message header. If one message is packaged into a data field inside another message, the component may not be able to use the fields from the original message to perform routing or transformation functions. Therefore, some data fields may have to be elevated from the original message into the message header of the new message format.

Use a *Envelope Wrapper* to wrap application data inside an envelope that is compliant with the messaging infrastructure. Unwrap the message when it arrives at the destination.



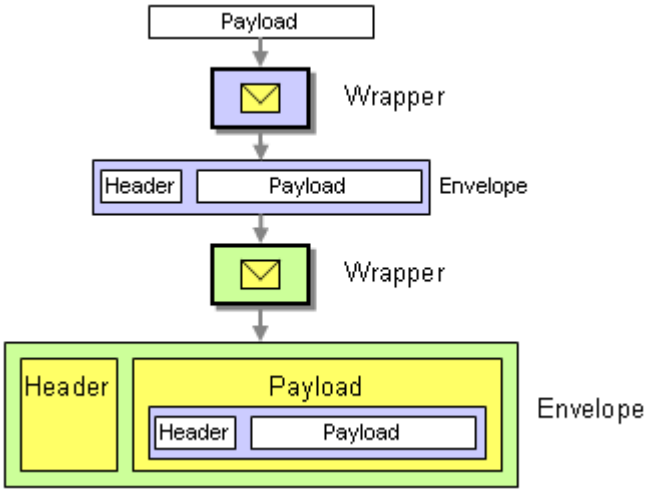
The process of wrapping and unwrapping a message consists of five steps:

- The Message Source publishes a message in a raw format. This format is typically determined by the nature of the application and does not comply with the requirements of the messaging infrastructure.
- The Wrapper takes the raw message and transforms it into a message format that complies with the messaging system. This may include adding message header fields, encrypting the message, adding security credentials etc.
- The Messaging System processes the compliant messages.
- A resulting message is delivered to the Unwrapper. The unwrapper reverses any modifications the wrapper made. This may include removing header fields, decrypting the message or verifying security credentials.
- The Message Recipient receives a 'clear text' message.

An envelope typically wraps both the message header and the message body or payload. We can think of the header as being the information on the outside of the envelope -- it is used by the messaging system to route and track the message. The contents of the envelope is the payload or body -- the messaging infrastructure does not care much about it (within certain limitations) until it arrives at the destination.

It is typical for wrappers to add information to the raw message. For example, before an internal message can be sent through the postal system, a ZIP code has to be looked up. In that sense, wrappers incorporate some aspects of a *Content Enricher*. However, wrappers do not enrich the actual information content, but add information that is necessary for the routing, tracking and handling of messages. This information can be created on the fly (e.g. creation of a unique message ID or adding a time stamp), it can be extracted from the infrastructure (e.g. retrieval of a security context), or the data may be contained in the original message body and the split by the wrapper into the message header (e.g. a key field contained in the 'raw' message). The last option is sometimes referred to as 'promotion' because a specific field is 'promoted' from being hidden inside the body to being prominently visible in the header.

Frequently, multiple wrappers and unwrappers are chained (see Postal System example below), taking advantage of the layered protocol model. This results in a situation where the payload of a message contains a new envelope, which in turn wraps a header and a payload section (see picture)



A Chain of Wrappers Creates a Hierarchical Envelope Structure

Example: SOAP Message Format

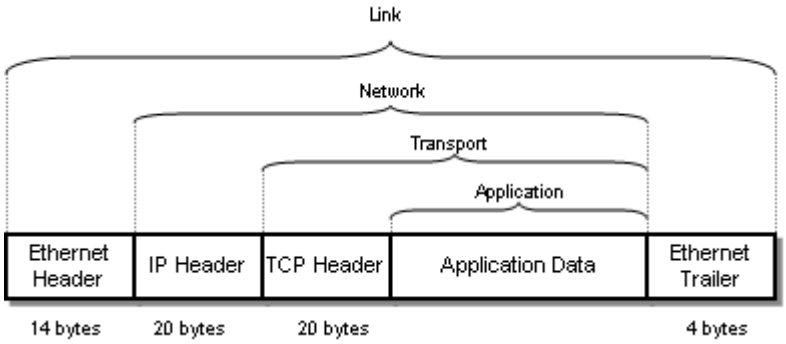
The basic SOAP message format [SOAP 1.1] is relatively simple. It specifies an envelope that contains a message header and a message body. The following example illustrates how the body can contain another envelope, which in turn contains another header and body. The combined message is sent to an intermediary who unwraps the outside message and forwards the inside message. This chaining of intermediates is very common when crossing trust boundaries. I may encode all my messages and wrap them inside another message so that no intermediary can see the message content or header (e.g. the address of a message may be confidential). The recipient then unwraps the message, decodes the payload and passes the unencoded message through the trusted environment.

```
<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
   <env:Header env:actor="http://example.org/xmlsec/Bob">
      <n:forward xmlns:n="http://example.org/xmlsec/forwarding">
         <n:window>120</n:window>
      </n:forward>
   </env:Header>
   <env:Body>
      <env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
         <env:Header env:actor="http://example.org/xmlsec/Alice"/>
         <env:Body>
            <secret xmlns="http://example.org/xmlsec/message">
     The black squirrel rises at dawn</secret>
         </env:Body>
      </env:Envelope>
   </env:Body>
</env:Envelope>
```

Example: TCP/IP

While we commonly use "TCP/IP" as one term, it actually comprises two protocols. The IP protocol provides basic addressing and routing services while TCP provides a reliable, connection-oriented protocol that is layered on top of IP. Following the OSI layer model, TCP is *Transport* protocol while IP is a *Network*protocol. Typically, TCP/IP data is transported over an Ethernet network, which implements the *Link* layer.

As a result, application data is wrapped into a TCP envelope first, which is then wrapped into an IP envelope, which is then wrapped into an Ethernet envelope. Since networks are stream-oriented an envelope can consists of both a header and a trailer. The following diagram illustrates the structure of application data traveling over the Ethernet:

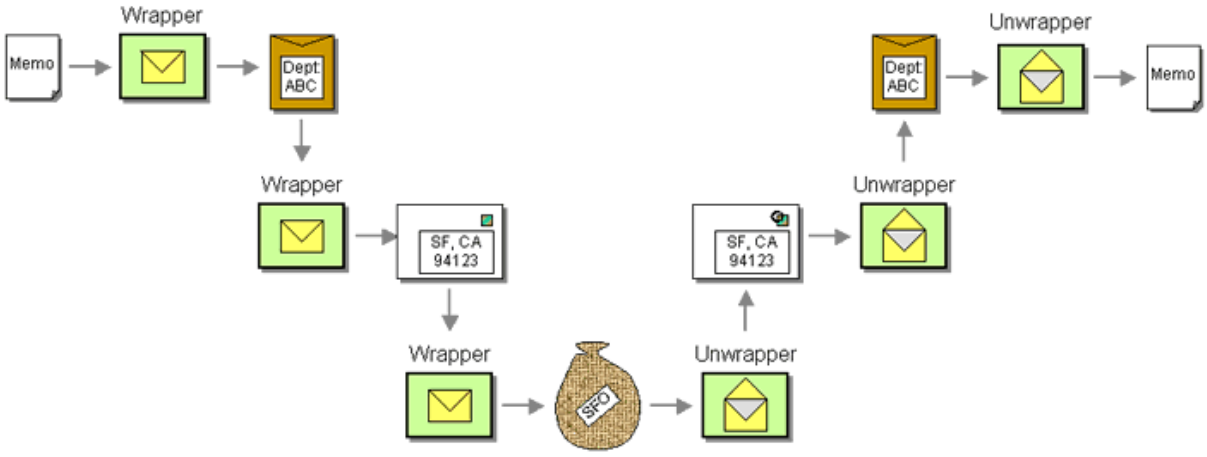


Application Data is Wrapped Inside Multiple Envelopes to be Transported over the Network

If you are interested in more details about TCP/IP, [Stevens] is guaranteed to quench your thirst for knowledge.

Example: The Postal System

The *Envelope Wrapper* pattern can be compared to the postal system (see Figure). Let's assume an employee creates an internal memo to a fellow employee. Any sheet of paper will be an acceptable format for this message. In order to deliver the memo it has to be 'wrapped' into an intra-company envelope that contains the recipient's name and department code. If the recipient works in a separate facility, this intra-company 'message' will be stuffed into a large envelope and mailed via the postal service. In order to make the new message comply with the USPS requirements, it needs to feature a new envelope with ZIP code and postage. The US Postal Service may decide to transport this envelope via air. To do so, it stuffs all envelopes for a specific region into a mailbag, which is addressed with a bar code featuring the three-letter airport code for the destination airport. Once the mailbag arrives at the destination airport, the wrapping sequence is reversed until the original memo is received by the coworker. This example illustrates the term "tunneling": Postal mail may be "tunneled" through air freight just like UDP mulitcast packets may be tunneled over a TCP/IP connection in order to reach a different WAN segment.



The postal system example illustrates the common practice of chaining wrappers and unwrappers using the *Pipes and Filters* pattern. Messages may be wrapped by more than one step and need to be unwrapped by a symmetric sequence of unwrapping steps. As laid out in the *Pipes and Filters* pattern, keeping the individual steps independent from each other gives the messaging infrastructure the flexibility to add or remove wrapping and unwrapping steps. For example, encryption may no longer be required because all traffic is routed across a VPN as opposed to the public Internet.

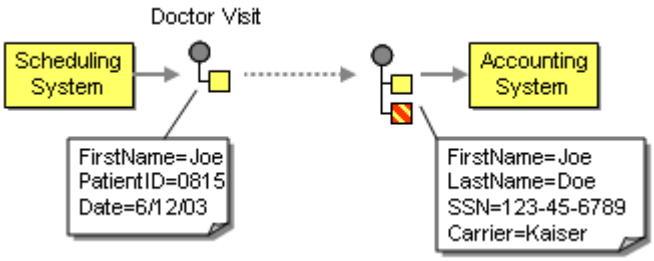
Related patterns: *Content-Based Router*, *Content Enricher*, *Messaging Bridge*, *Pipes and Filters*

Content Enricher

When sending messages from one system to another it is common for the target system to require more information than the source system can provide. For example, incoming `Address` messages may just contain the ZIP code because the designers felt that storing a redundant state code would be superfluous. Likely, another system is going to want to specify both a state code and a ZIP code field. Yet another system may not actually use state codes, but spell the state name out because it uses free-form addresses in order to support international addresses. Likewise, one system may provide us with a customer ID, but the receiving system actually requires the customer name and address. An order message sent by the order management system may just contain an order number, but we need to find the customer ID associated with that order, so we can pass it to the customer management system. The scenarios are plentiful.

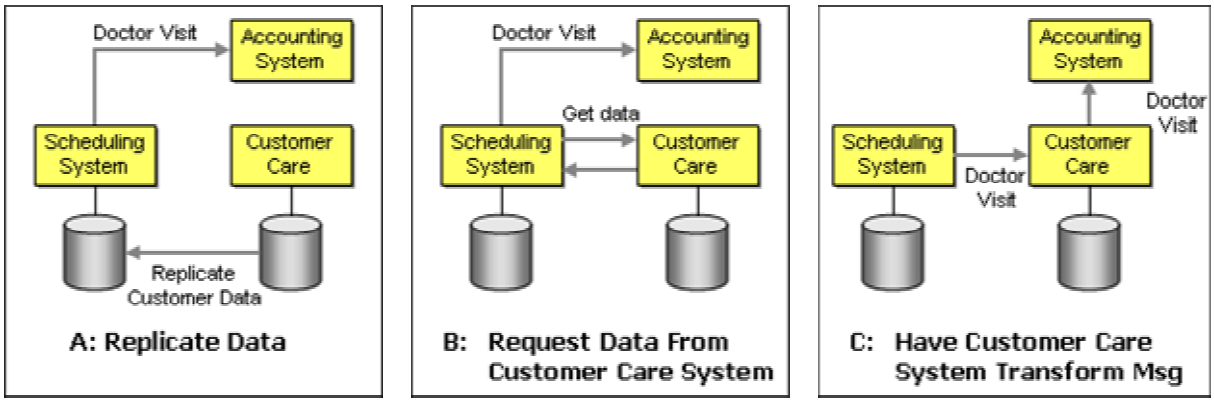
How do we communicate with another system if the message originator does not have all the required data items available?

This problem is a special case of the *Message Translator*, so some of the same considerations apply. However, this problem is slightly different from the basic examples described in the *Message Translator*. The description of the *Message Translator* assumed that the data needed by the receiving application is already contained in the incoming message, albeit in the wrong format. In this new case, it is not a simple matter of rearranging fields, we actually need to inject additional information to the message.



The Accounting System Requires More Information Than The Scheduling System Can Deliver

Let's consider the following example (see picture). A hospital scheduling system publishes a message announcing that the patient has completed a doctor's visit. The message contain's the patient's first name, his or her patient ID and the date of the visit. In order for the accounting system to log this visit and inform the insurance company, it requires the full patient name, the insurance carrier and the patient's social security number. However, the scheduling system does not store this information, it is contained in the customer care system. What are our options?



Possible solutions for the Enricher problem

Option A: We could modify the scheduling system so it can store the additional information. When the customer's information changes in the customer care system (e.g. because the patient switches insurance carriers) the changes need to be replicated to the scheduling system. The scheduling system can now send a message that includes all required information. Unfortunately, this approach has two significant drawbacks. First, it requires a modification to the scheduling system's internal structure. In most cases, the scheduling system is going to be a packaged application and may not allow this type of modification. Second, even if the scheduling system is customizable, we need to consider that we are making a change to the system based on a the specific needs of another system. For example, if we also want to send a letter to the patient confirming the visit we would have to change the scheduling system again to accommodate the customer's mailing address. The integration solution would be much more maintainable if we decouple the scheduling system from the specifics of the applications that consume the "Doctor Visit" message.

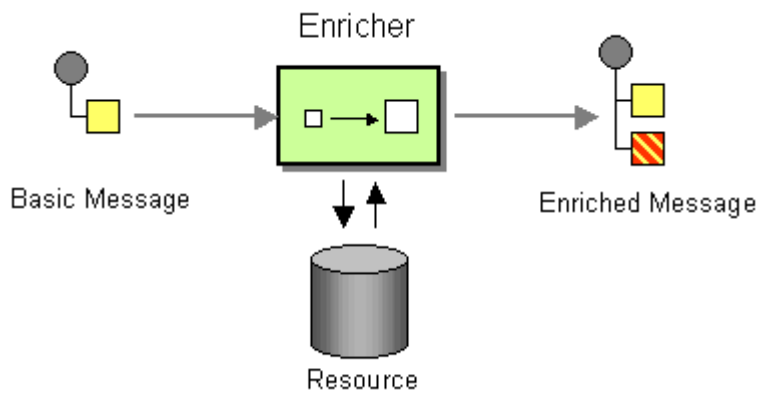
Option B: Instead of storing the customer's information inside the scheduling system, the scheduling system could request the SSN and carrier data from the customer care system just

before it is sending the 'Doctor Visit' message. This solves the first problem -- we no longer have to modify the storage of the scheduling system. However, the second problem remains: the scheduling system needs to know that the SSN and carrier information is required in order to notify the accounting system. Therefore, the semantics of the message is more similar to 'Notify Insurance' than 'Doctor Visit'. In a loosely coupled system we do not want one system to instruct the next one on what to do. We rather send an *Event Message* and let the other systems decide what to do. In addition, this solution couples the scheduling system more tightly to the customer care system because the scheduling system now needs to know where to get the missing data. This ties the scheduling system to both the accounting system and the customer care system. This type of coupling is undesirable because it leads to brittle integration solutions.

Option C: We can avoid some of these dependencies if we send the message to the customer care system first instead of the accounting system. The customer care system can then fetch all the required information and send a message with all required data to the accounting system. This decouples the scheduling system nicely from the subsequent flow of the message. However, now we implement the business rule that a the insurance company receives a bill after the patient visits the doctor inside the customer care system. This requires us to modify the logic inside the customer care system. If the customer care system is a packaged application, this modification may be difficult or impossible. Even if we can make this modification, we now make the customer care system indirectly responsible for sending bills messages. This may not be a problem if all the data items required by the accounting system are available inside the customer care system. If some of the fields have to be retrieved from other systems we are in a similar situation to where we started.

Option D (not shown): We could also modify the accounting system to only require the customer ID and retrieve the SSN and carrier information from the customer care system. This approach has two disadvantages. First, we now couple the accounting system to the customer care system. Second, this option again assumes that we have control over the accounting system. In most cases, the accounting system is going to be a packaged application with limited options for customization.

Use a specialized transformer, a *Content Enricher*, to access an external data source in order to augment a message with missing information.



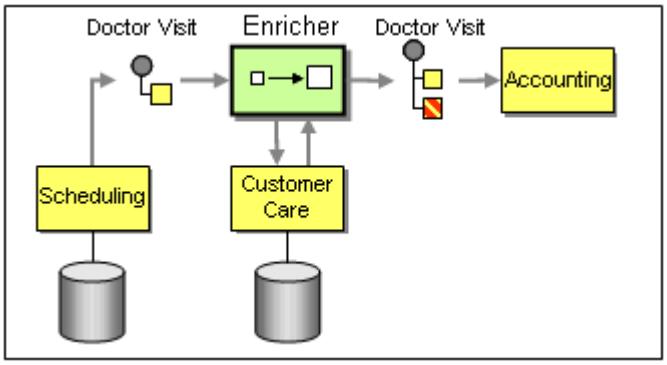
The *Content Enricher* uses information inside the incoming message (e.g. key fields) to retrieve data from an external source. After the *Content Enricher* retrieves the required data from the resource, it appends the data to the message. The original information from the incoming message may be carried over into the resulting message or may no longer be needed, depending on the specific needs of the receiving application.

The additional information injected by the *Content Enricher* has to be available somewhere in the system. The most common sources for the new data are:

- **Computation** The *Content Enricher* may be able to compute the missing information. In this case, the algorithm incorporates the additional information. For example, if the receiving system requires a state code but the incoming message only contains a ZIP code. Or, a receiving system may require a data format that specifies the total size of the message. The *Content Enricher* can add the length of all message fields and thus compute the message size. This form of *Content Enricher* is very similar to the basic *Message Translator* because it needs no external data source.
- **Environment** The *Content Enricher* may be able to retrieve the additional data from the operating environment. The most common example is a time stamp. For example, the receiving system may require each message to carry a time stamp. if the sending system does not include this field, the *Content Enricher* can get the current time from the operating system and add it to the message.
- **Another System** This option is the most common one. The *Content Enricher* has to retrieve the missing data from another system. This data resource can take on a number of forms, including a database, a file, an LDAP directory, a system, or a user who manually enters missing data.

In many cases, the external resource required by the *Content Enricher* may be situated on another system or even outside the enterprise. Accordingly, the communication between the *Content Enricher* and the resource can occur via message channels or via any other communication mechanism. Since the interaction between the *Content Enricher* and the data source is by definition synchronous (the *Content Enricher* cannot send the enriched message until the data source returns the requested data), a synchronous protocol (e.g. HTTP or an ODBC connection to a database) may result in better performance than using asynchronous messaging. The *Content Enricher* and the data source are inherently tightly coupled, so achieving loose coupling through *Message Channel*s is not as important.

Returning to our example, we can insert a *Content Enricher* to retrieve the additional data from the customer care system (see picture). This way, the scheduling system is nicely decoupled from having to deal with insurance information or the customer care system. All it has to do is publish the 'Doctor Visit' message. The *Content Enricher* component takes care of retrieving the required data. The accounting system also remains independent from the customer care system.

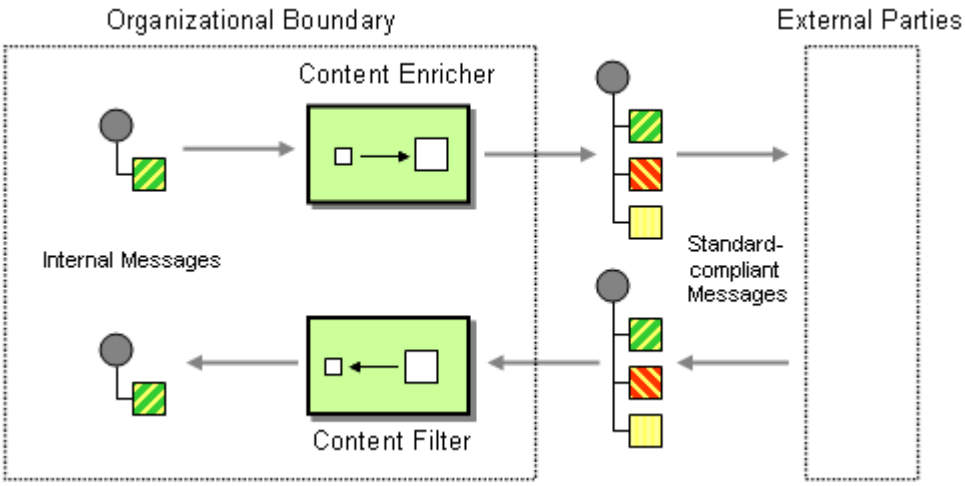


Applying the Enricher to the Patient Example

The *Content Enricher* is used in many occasions to resolve references contained in a message. In order to keep message small and easy to manage, often times we choose to pass simple references to objects rather than passing a complete object with all data elements. These references usually take the form of keys or unique IDs. When the message needs to be processed by a system, we need to retrieve the required data items based on the object references included in the original message. We use a *Content Enricher* to perform this task. There are some apparent trade-offs involved. Using references reduces the data volume in the original messages, but requires additional look-ups in the resource. Whether the use of references improves performance depends on how many components can operate simply on references versus how many components need to use an *Content Enricher* to restore some of the original message content. For example, if a message passes through a long list of intermediaries before it reaches the final recipient, using an object reference can decrease message traffic significantly. We can insert a *Content Enricher* as the last step before the final recipient to load the missing information into the message. If the message already contains data that we we might not want to carry along the way, we can use the *Claim Check* to store the data and obtain a reference to it.

Example: Communication with External Parties

A *Content Enricher* is also commonly used when communicating with external parties that require messages to be compliant with a specific message standard (e.g. ebXML). Most of these standards require large messages with a long list of data. We can usually simplify our internal operations significantly if we keep internal messages as simple as possible and then use a *Content Enricher* to add the missing fields whenever we send a message outside of the organization. Likewise, we can use a *Content Filter* to strip unnecessary information from incoming messages (see picture).



Using a Content Enricher / Content Filter Pair When Communicating with External Parties

Related patterns: *Content Filter*, *Event Message*, *Message Channel*, *Message Translator*, *Claim Check*

Content Filter

The *Content Enricher* helps us in situations where a message receiver requires more - or different - data elements than the message creator provides. There are surprisingly many situations where the opposite effect is desired: removing data elements from a message.

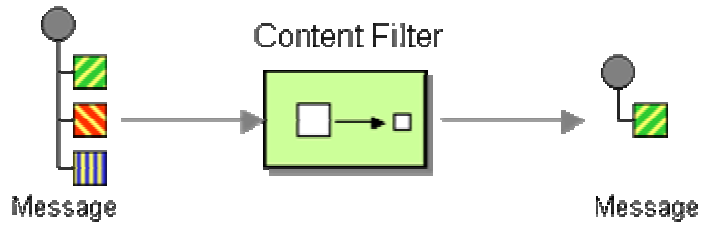
How do you simplify dealing with a large message, when you are interested only in a few data items?

Why would we want to remove valuable data elements from a message? One common reason is security. A requestor of data may not be authorized to see all data elements that a message contains. The service provider may not have knowledge of a security scheme and always return all data elements regardless of user identity. We need to add a step that removes sensitive data based on the requestor's proven identity. For example, the payroll system may expose only a simple interface that returns all data about an employee. This data may include payroll information, social security numbers and other sensitive information. If you are trying to build a service that returns an employee's start date with the company you may want to eliminate all sensitive information from the result message.

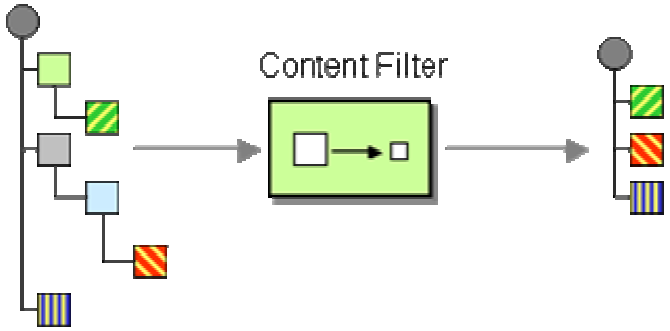
Another reason to remove data elements is to simplify message handling and to reduce network traffic. In many instances, processes are initiated by messages received from business partners. For obvious reasons, it is desirable to base communication with third parties on a standardized message format. A number of standards bodies and committees define standard XML data formats for certain industries and applications. Well-known examples are RosettaNet, ebXML, ACORD and many more. While these XML formats are useful to conduct interaction with external parties based on an agreed-upon standard, the documents can be very large. Many of the documents have hundreds of fields, consisting of multiple nested levels. Such large documents are difficult to work with for internal message exchange. For example, most visual (drag-drop

style) transformation tools become unusable if the documents to be mapped have hundreds of element. Also, debugging becomes a major nightmare. Therefore, we want to simplify the incoming documents to include only the elements we actually require for our internal processing steps. In a sense, removing elements enriches the usefulness of such a message, because redundant and irrelevant fields are removed, leaving a more meaningful message and less room for developer mistakes.

Use a *Content Filter* to remove unimportant data items from a message leaving only important items.



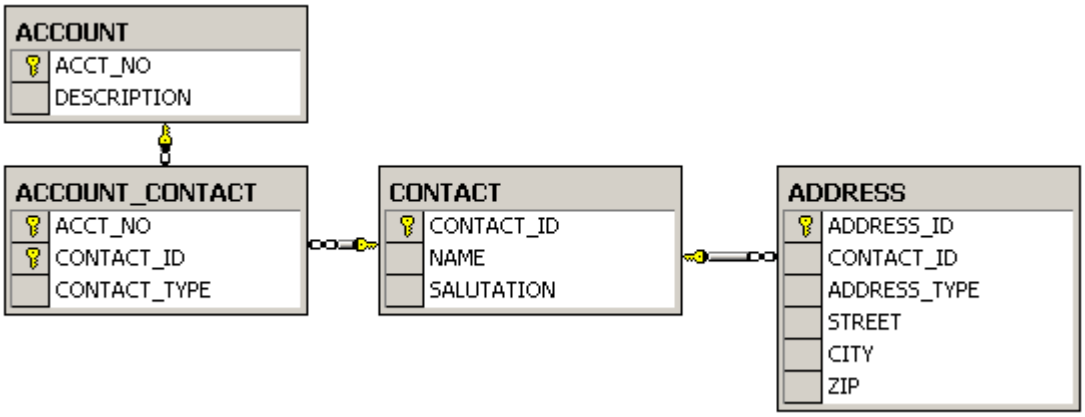
The *Content Filter* does not necessarily just remove data elements. A *Content Filter* is also useful to simplify the structure of the message. Often times, messages are represented as tree structures. Many messages originating from external systems or packaged applications contain many levels of nested, repeating groups because they are modeled after generic, normalized database structures. Frequently, known constraints and assumptions make this level of nesting superfluous and a *Content Filter* can be used to 'flatten' the hierarchy into a simple list of elements than can be more easily understood and processed by other systems.



Multiple *Content Filter*s can be used as a *Filtering Splitter* (see *Splitter*) to break one complex message into individual messages that each deal with a certain aspect of the large message.

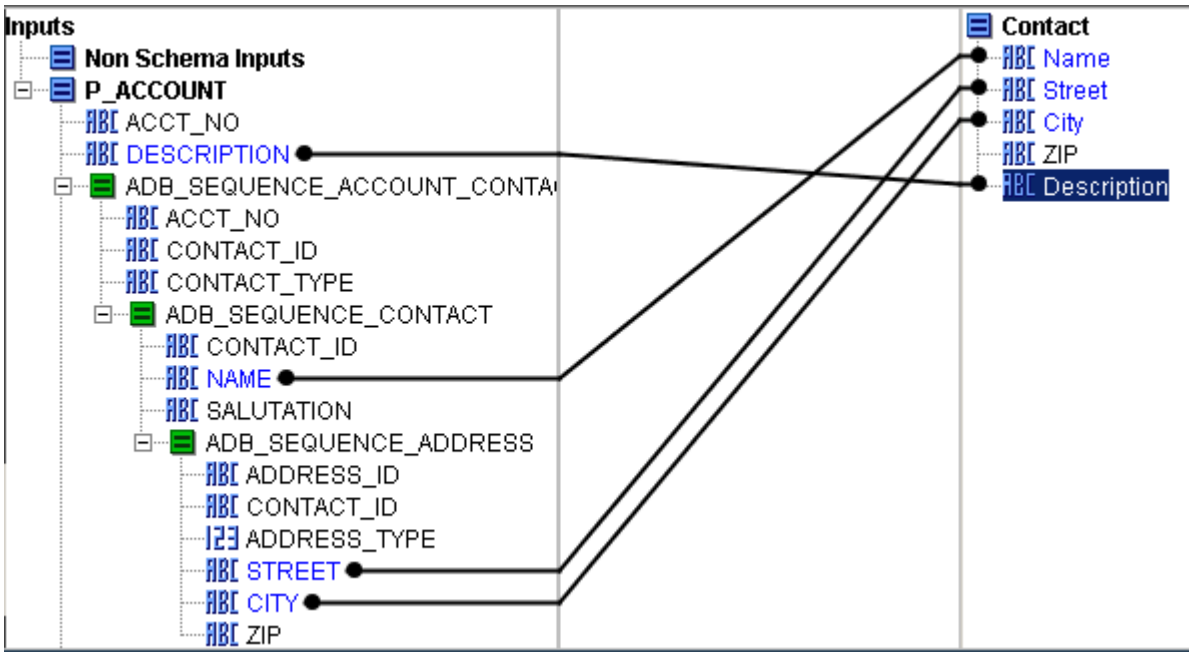
Example: Database Adapter

Many integration suites provide *Channel Adapter*s to connect to existing systems. In many cases, these adapters publish messages whose format resembles the internal structure of the application. For example, let's assume we connect a database adapter to a database with the following schema:



An Example Database Schema

It is very typical for a physical database schema to store related entities in separate tables that are linked by foreign keys and relation tables (e.g. ACCOUNT_CONTACT links the ACCOUNT and CONTACT tables). Many database adapters will translate these related tables into a hierarchical message structure that can contain additional fields such as primary and foreign keys that may not be relevant to the message receiver. In order to make processing a message easier, we can use a *Content Filter* to flatten the message structure and extract only relevant fields. The example shows the implementation of a *Content Filter* using a visual transformation tool. We can see how we reduce the message from over a dozen fields spread across multiple levels into a simple message with five fields. It will be much easier (and more efficient) for other components to work with the simpler message.



Simplifying a Message Published by a Database Adapter

A *Content Filter* is not the only solution to this particular problem. For example, we could configure a view in the database that resolves the table relationships and returns a simple result

set. This may be a simple choice if we have the ability to add views to the database. In many situations, enterprise integration aims to be as little intrusive as possible and that guideline may include not adding views to a database.

Related patterns: *Channel Adapter*, *Content Enricher*, *Splitter*

Claim Check

The *Content Enricher* tells us how we can deal with situations where our message is missing required data items. The *Content Filter* lets us remove uninteresting data items from a message. Sometimes, we want to remove fields only temporarily. For example, a message may contain a set of data items that may be needed later in the message flow, but that are not necessary for all intermediate processing steps. We may not want to carry all this information through each processing step because it may cause performance degradation and makes debugging harder because we carry so much extra data.

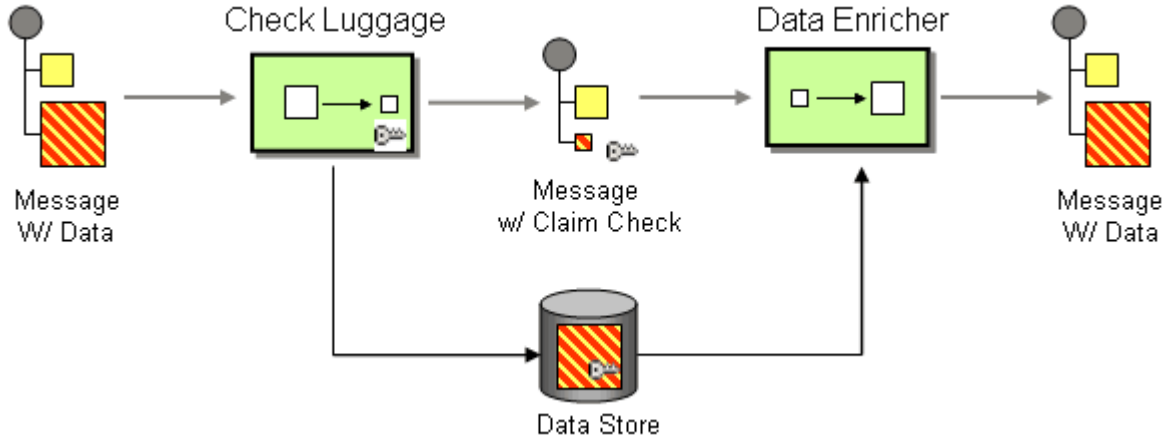
How can we reduce the data volume of message sent across the system without sacrificing information content?

Moving large amounts of data via messages may be inefficient. Some messaging systems even have hard limits as to the size of messages. Other messaging systems use an XML representation of data, which can increase the size of a message by an order of magnitude or more. So while messaging provides the most reliable and responsive ay to transmit information, it may not be the most efficient.

A simple *Content Filter* helps us reduce data volume but does no guarantee that we can restore the message content later on. Therefore, we need to store the complete message information in a way that we can retrieve it later.

Because we need to store data for each message, we need a key to retrieve the correct data items associated with a message. We could use the message ID as the key, but that would not allow subsequent components to pass the key on because the message ID changes with each message.

Store message data in a persistent store and pass a *Claim Check* to subsequent components. These components can use the *Claim Check* to retrieve the stored information.

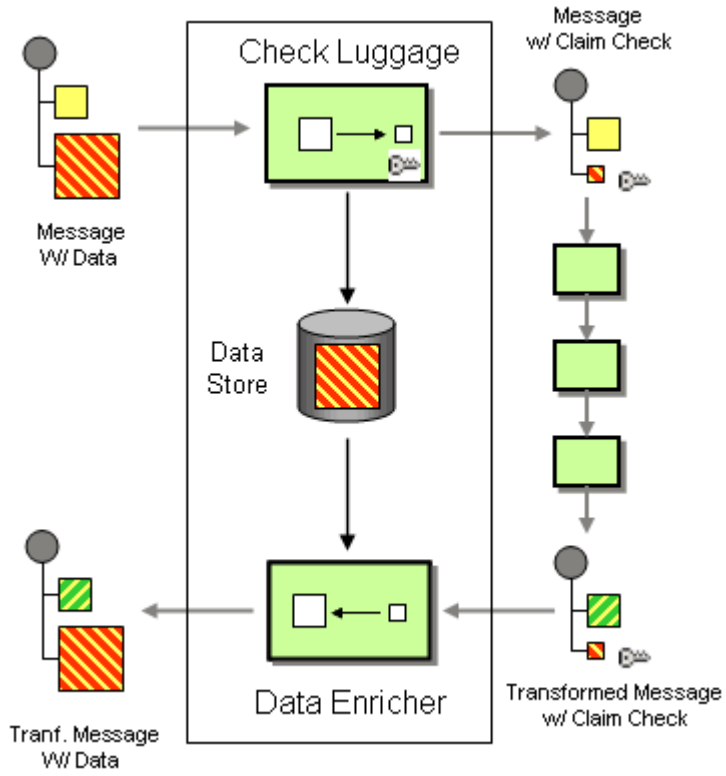


The *Claim Check* pattern consists of the following steps:

1. A message with data arrives.
2. The "Check Luggage" component generates a unique key for the information. This key will be used later as the *Claim Check*
3. The Check Luggage component extracts the data from the message and stores it in a persistent store, e.g. a file or a database. It associates the stored data with the key .
4. It removes the persisted data from the message and adds the *Claim Check*.
5. Another component can use a *Content Enricher* to retrieve the data based on the *Claim Check*.

This process is analogous to a luggage check at the airport. If you do not want to carry all your luggage with you, you simply check it with the airline counter. In return you receive a sticker on your ticket that has a reference number that uniquely identifies each piece of luggage you checked. Once you reach your final destination, you can retrieve your luggage.

As the picture illustrates, the data that was contained in the original message still needs to be "moved" to the ultimate destination. So did we gain anything? Yes, because transporting data via messaging may be less efficient than storing it in a central data store. For example, the message may undergo multiple routing steps that do not require the large amount of data. Using a messaging system, the message data would be marshaled and unmarshaled, possibly encrypted and decrypted at every step. This type of operation can be very CPU intensive and would be completely unnecessary of the data is not needed by an intermediate step but only by the final destination. The *Claim Check* also works well in a scenario where a message travels through a number of components and returns to the sender. In this case, the "Check Luggage" component and the *Content Enricher* are local to the same component and the data never has to travel across the network (see below):



The Data May Be Stored and Retrieved Locally

Choosing a Key

How should we choose a key for the data? A number of options spring to mind:

- A business key may already be contained in the message body, e.g. a Customer ID.
- The message may contain a message ID that can be used to associate the data in the data store with the message.
- We can generate a unique ID.

Reusing an existing business key seem like the easiest choice. If we have to stow away some customer detail we can reference it later by the customer ID. When we pass this key to other components we need to decide whether we want these components to be aware that the key is a customer ID as opposed to just an abstract key. Representing the key as an abstract key has the advantage that we can process all keys in the same way and can create a generic mechanism to retrieve data from the data store based on an abstract key.

Using the message ID as the key may seem convenient but is generally not a good idea. Using a message ID as a key for data retrieval results in dual semantics being attached to a single data element and can cause conflicts. For example, let's assume we need to pass the reference on to another message. The new message is supposed to be assigned a new, unique ID, but then we can't use that new ID anymore to retrieve data from the data store. The use of a message ID can be meaningful only in a circumstance where we want the data to be accessible only within the

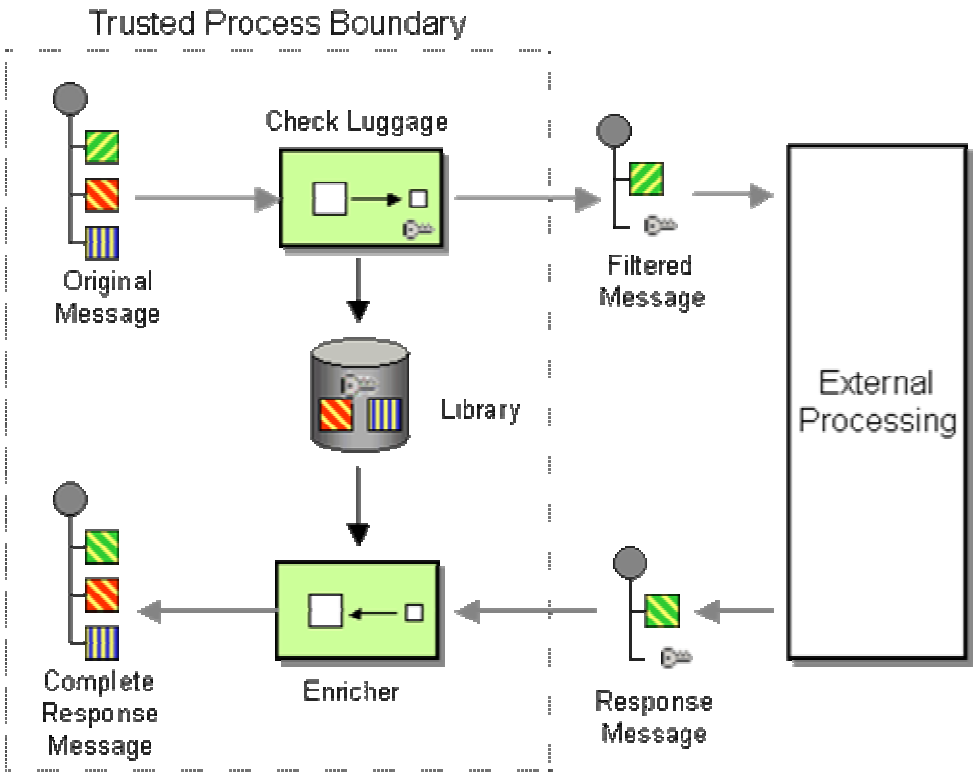
scope of the single message. Therefore, in general it is better to assign a new element to hold the key and avoid this bad form of 'element reuse'.

The data may be stored in the data store only temporarily. How do we remove unused data? We can modify the semantics of the data retrieval from the data store to delete the data when it is read. In this case we can retrieve the data only once, which may actually be desirable in some cases for security reasons. However, it does not allow multiple components to access the same data. Alternatively, we can attach an expiration date to the data and define a 'garbage collection' process that periodically removes all data over a certain age. As a third option, we may not want to remove any data. This may be the case because we use a business system as the data store (e.g. an accounting system) and need to maintain all data in that system.

The implementation of a data store can take on various forms. A database is an obvious choice, but a set of XML files or an in-memory message store can serve as a data store just as well. Sometimes, we may use an application as the data store. It is important that the data store is reachable by components in other parts of the integration solution so that these parts can reconstruct the original message.

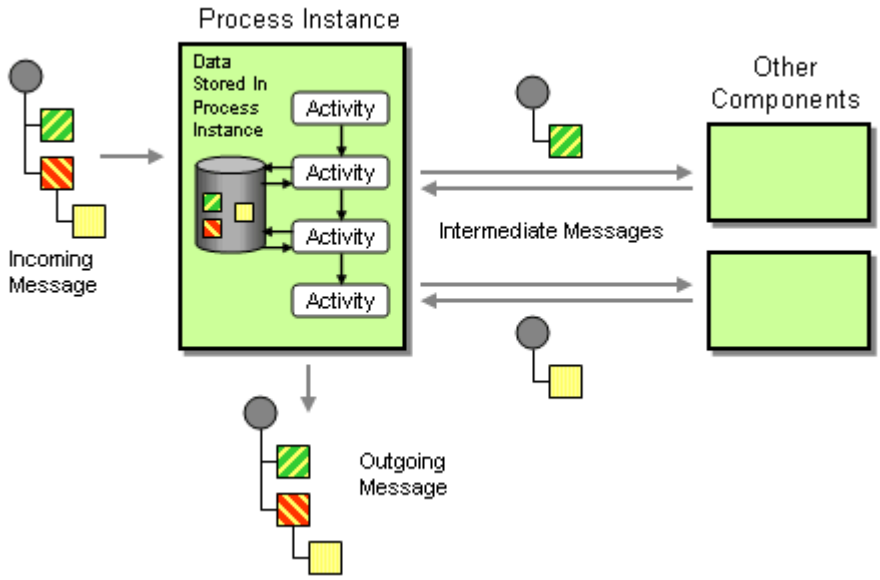
Using a Claim Check to Hide Information

While the original intent of the *Claim Check* is to avoid sending around large volumes of data, it can also serve other purposes. Often times we want to remove sensitive data before sending a message to an outside party (see picture). This accomplishes that outside parties receive data only on a 'need to know basis'. For example when we send employee data to an external party we may prefer not to reference employees by some magic unique ID and eliminate fields such as social security number. After the outside party has completed the required processing, we reconstruct the complete message by merging data from the data store and the message returned from the outside party. We may even generate special unique keys for these messages so that we restrict the actions the outside party can take by the key it possesses. This will restrict the outside party from maliciously feeding messages into our system. Messages containing an invalid (or expired or already used) key will be blocked by the *Content Enricher* attempting to retrieve message data using the key.



Using a Process Manager with a Claim Check

If we interact with more than one external party, a *Process Manager* can serve as a *Claim Check*. A *Process Manager* creates process instances (sometimes called 'tasks' or 'jobs') when a message arrives. The *Process Manager* allos additional data to be associated with each individual process instance. In effect, the process engine now serves as the data store, storing our message data. This allows the *Process Manager*to send messages to external parties that contain only the data relevant to that party. The messages do not have to carry all the information contained in the original message since that information is kept with the process' data store. When the *Process Manager*receives a response message from an external party it re-merges the new data with the data stored by the process instance.



Related patterns: *Content Filter*, *Content Enricher*, *Process Manager*

Normalizer

In a business-to-business (B2B) integration scenario it is quite common for an enterprise to receive messages from different business partners. These message may have the same meaning, but follow different formats, depending on the partners' internal systems and preferences. For example, we built a solution for a pay-per-view provider that has to accept and process viewership information from over 1700 (!) affiliates, most of which did not conform to a standard format.

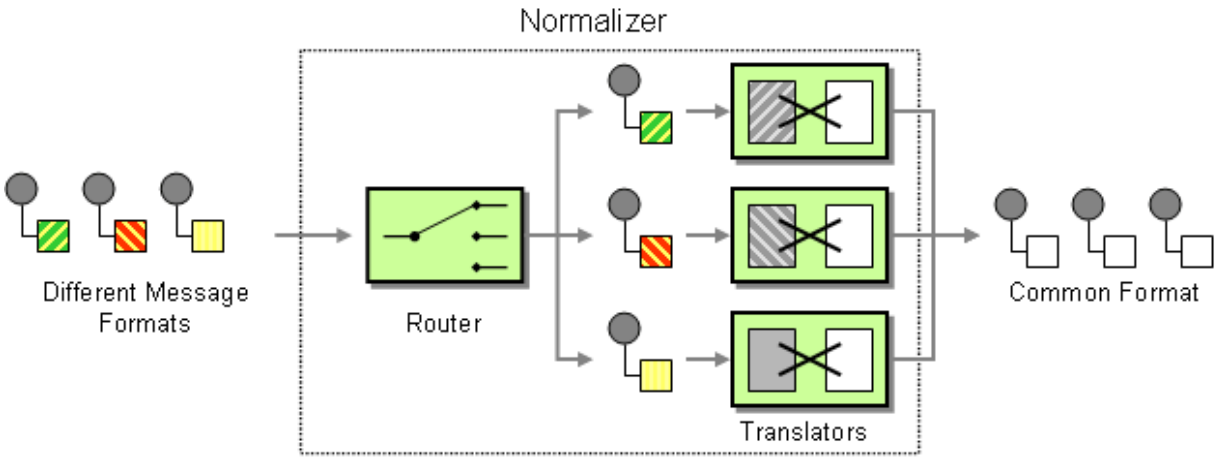
How do you process messages that are semantically equivalent, but arrive in a different format?

The easiest solution from a technical perspective may seem to dictate a uniform format on all participants. This may work if the business is a large corporation and has control over the B2B exchange or the supply channel. For example, if General Motors would like to receive order status updates from their suppliers in a common message format, we can be pretty sure that Joe's Supplier business is likely to conform to the 'guidelines'. In many other situations, however, a business is not going to have such a luxury. In the contrary, many business models position the message recipient as an 'aggregator' of information and part of the agreement with the individual participants is that a minimum of changes is required to their systems infrastructure. As a result, you find the aggregator willing to process information arriving in any data format ranging from EDI records or comma separated files to XML documents or Excel spreadsheets arriving via e-mail.

One important consideration when dealing with a multitude of partners is the rate of change. Not only may each participant prefer a different data format to begin with -- the preferred format may also change over time. In addition, new participants may join while others drop off. Even if a specific partner makes changes to the data format only once every couple of years, dealing with a few dozen partners can quickly result in monthly or weekly changes. It is important to isolate these changes from the rest of the processing as much as possible to avoid a "ripple-effect" of changes throughout the whole system.

To isolate the remainder of the system from the variety of incoming message formats, you need to transform the incoming messages into a common format. Because the incoming messages are of different types, you need a different *Message Translator* for each message data format.The easiest way to accomplish this is to use a collection of *Datatype Channel*s, one for each message type. Each *Datatype Channel* is then connected to a different *Message Translator*. The drawback of this approach is that a large number of message formats translates into an equally large number of message channels.

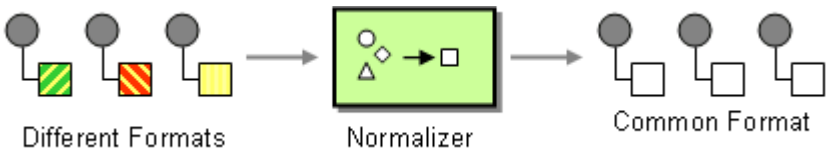
Use a *Normalizer* to route each message type through a custom *Message Translator* so that the resulting messages match a common format.



The *Normalizer* uses a *Message Router* to route the incoming message to the correct *Message Translator*. This requires the *Message Router* to detect the type of the incoming message. Many messaging systems equip each message with a type specifier field in the Message Header to make this type of task simple. However, in many B2B scenarios messages do not arrive as messages compliant with the enterprise's internal messaging system, but in diverse formats such as comma separated files or XML document without associated schema. While it is certainly best practice to equip any incoming data format with a type specifier we know all to well that the world is far from perfect. As a result, we need to think of more general ways to identify the format of the incoming message. One common way for schema-less XML documents is to use the name of the root element to assume the correct type. If multiple data formats use the same root element, you can use XPATH expressions to determine the existence of specific sub-nodes. Comma-separated files can require a little more creativity. Sometimes you can determine the type based on the number of fields and the type of the data (e.g. numeric vs. string). If the data arrives as files, the easiest way may be to use the file name or the file folder structure as a surrogate *Datatype Channel*. Each business partner can name the file with a unique naming convention. The *Message Router* can then use the file name to route the message to the appropriate *Message Translator*.

Adding a *Message Router* also allows the same transformation to be used for multiple business partners. That might be useful if multiple business partners use the same format or if a transformation is generic enough to accommodate multiple message formats. For example, x-path expressions are great at picking out elements from XML documents even if the documents vary in format.

Since a *Normalizer* is a common occurance in messaging solutions, we created a short-hand icon for it:



Related patterns: *Datatype Channel*, *Message Router*, *Message Translator*

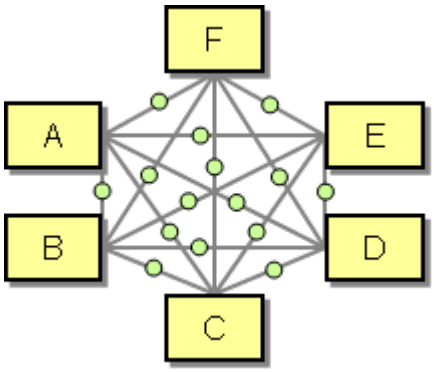
Canonical Data Model

I am designing several applications to work together through *Messaging*. Each application has its own internal data format.

How can you minimize dependencies when integrating applications that use different data formats?

Independently developed applications tend to use different data formats because each format was designed with just that application in mind. When an application is designed to send messages to or receive messages from some unknown application, the application will naturally use the message format that is most convenient for it. Likewise, commercial adapters used to integrate packaged applications using publish and consume messages in a data format that resembles the application's internal data structure.

The *Message Translator* resolves differences in message formats without changing the applications or having the applications know about each other's data formats. However, if a large number of applications communicate with each other, one *Message Translator* may be needed between each pair of communicating applications (see picture).



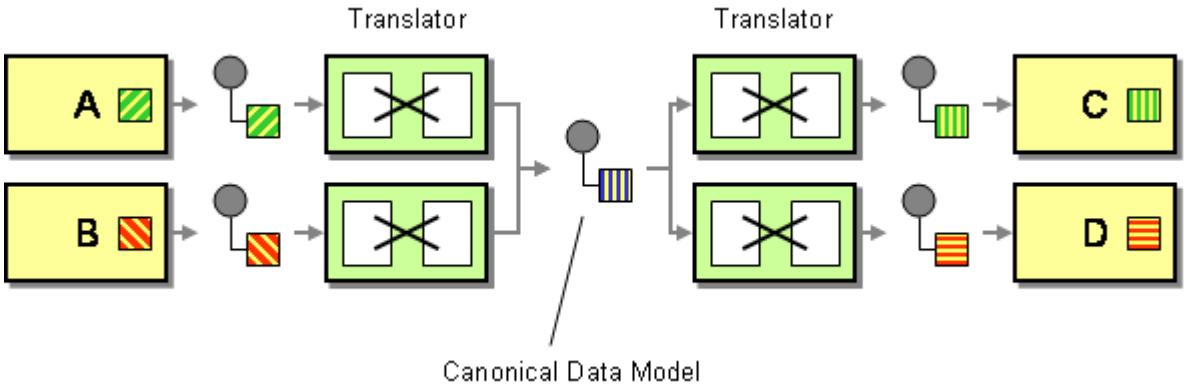
This approach requires a large number of *Message Translator*s, especially when considering that each integrated application may publish or consume multiple message types. The number of required *Message Translator*s increases exponentially with the number of integrated applications which quickly becomes unmanageable.

While the *Message Translator* provides an indirection between the message formats used by two communicating applications it is still dependent on the message formats used by either

application. As a result, if an application's data format changes, all *Message Translator*s between the changing application and all other applications that it communicates with have to change. Likewise, if a new application is added to the solution, new *Message Translator*s have to be created from each existing application to the new application in order to exchange messages. This situation creates a nightmare out of having to maintain all *Message Translator*s.

We also need to keep in mind that each additional transformation step injected into a message flow can increase latency and reduce message throughput.

Therefore, design a *Canonical Data Model* that is independent from any specific application. Require each application to produce and consume messages in this common format.



The *Canonical Data Model* provides an additional level of indirection between application's individual data formats. If a new application is added to the integration solution only transformation between the *Canonical Data Model* has to created, independent from the number of applications that already participate.

The use a *Canonical Data Model* seems complicated if only a small number of applications participate in the integration solution. However, the solution quickly pays off as the number of applications increases. If we assume that each application sends and receives message to and from each other application, a solution consisting of 2 applications would require only 2 *Message Translator*s if we translate between the applications' data formats directly whereas the *Canonical Data Model* requires 4 *Message Translator*s. A solution consisting of 3 applications requires 6 *Message Translator*s with either approach. However, a solution consisting of 6 applications requires 30(!) *Message Translator*s without a *Canonical Data Model* and only 12 *Message Translator*s when using a *Canonical Data Model*.

The use of a *Canonical Data Model* can also be very useful if an existing application is likely to be replaced by another application in the future. For example, is a number of applications interface with a legacy system that is likely to be replaced by a new system in the future, the effort of switching from one application to the other is much reduced if the concept of a *Canonical Data Model* is built into the original solution.

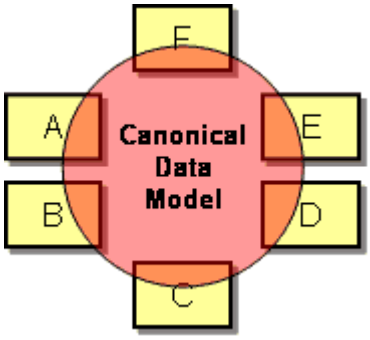
How do you make applications conform to the common format? You have three basic choices:

- **Change the applications internal data format.** This may be in theory possible, but is unlikely in a complex real-life scenario. If it was easy to just make each application to natively use the same data format, we would be better off using *Shared Database* and not *Messaging*.
- **Implement a *Messaging Mapper*** inside the application. Custom applications can use a mapper to generate the desired data format.
- **Use an external *Message Translator*.** You can use an external *Message Translator* to translate from the app-specific message format into the format specified by the *Canonical Data Model*.

Whether to use a *Messaging Mapper* or an external *Message Translator* depends on the complexity of the transformation and the maintainability of the application. Packaged applications usually eliminate the use of a *Messaging Mapper* because the source code is not available. For custom applications the choice depends on the complexity of the transformation. Many integration tool suites provide visual transformation editors that allow faster construction of mapping rules. However, these visual tools can get unwieldy if transformations are complex.

The use of a *Canonical Data Model* does introduce a certain amount of overhead into the message flow. Each message now has to undergo two translation steps instead of one -- one translation from the source application's format into the common format and from the common format into the target application's format. For this reason, the use of a *Canonical Data Model* is sometimes referred to as "double translation" (transforming directly from one application's format to the other is called "direct translation"). Each translation step causes additional latency in the flow of messages. Therefore, for very high throughput systems direct translation can be the only choice. This trade-off between maintainability and performance is common. The best advice is to use the more maintainable solution (i.e., the *Canonical Data Model*) unless performance requirements do not allow it. A mitigating factor may be that fact that many translations are stateless and therefore lend themselves to load balancing with multiple *Message Translator*s executing in parallel.

Designing a *Canonical Data Model* can be difficult. Designers should strive to make the unified model work equally well for all applications being integrated, but in reality this ideal is difficult to achieve. The chances of successfully definition a *Canonical Data Model* improve when considering that the *Canonical Data Model* does not have to model the complete data model of all applications, but only the portion that participates in messaging (see picture). This can significantly reduce the complexity of creating the *Canonical Data Model*.



Using a *Canonical Data Model* can also have political advantages. Using a *Canonical Data Model* allows developers and business users to discuss the integration solution in terms of the company's business domain, not a specific package implementation. For example, packaged applications may represent the common concept of a customer in many different internal formats, such as 'account', 'payer', 'contact' etc. Defining a *Canonical Data Model* is often the first step to resolving case of *semantic dissonance* between applications (see [Kent].

The picture in the pattern introduction showing the large number of transformers needed to translate between each and every application looks surprisingly similar to the picture in *Introduction to Message Routing*. This reminds us that dependencies between applications can exist at multiple levels. The use of *Message Channel*s provides a common transport layer between applications and removes dependencies between application's individual transport protocols. *Message Router*s can provide location-independence so that a sending application does not have to depend on the location of the receiving application. The use of a common data representation such as XML removes dependencies on any application-specific data types. Finally, the *Canonical Data Model* resolves dependencies in the data formats and semantics used by the applications.

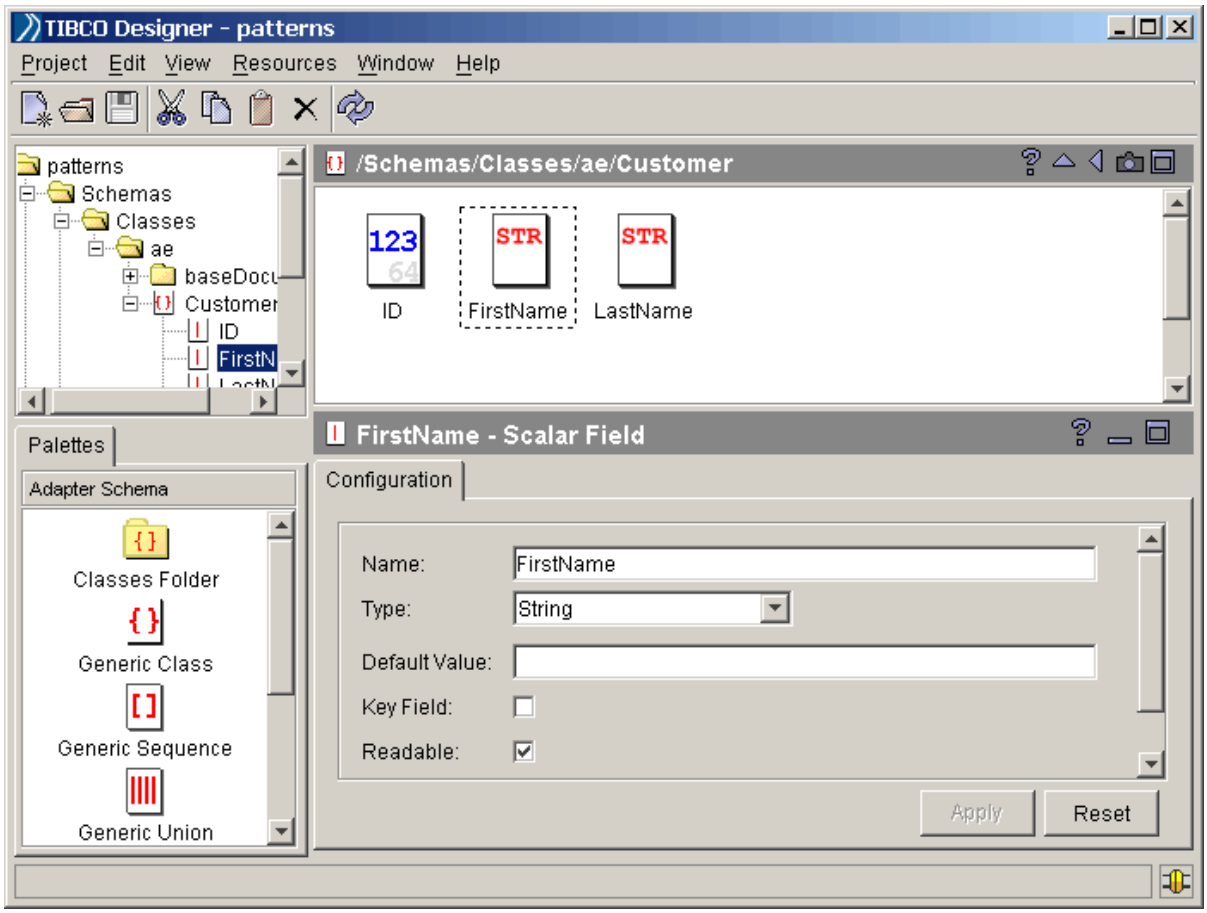
Since the canonical format is likely to change over time, it should specify a *Format Indicator*.

Example: WSDL

When accessing an external service from your application the service may already specify a *Canonical Data Model* to be used. In the world of XML Web services, the data format is specified by a WSDL document, the Web Services Definition Language (see [WSDL 1.1]. The WSDL specifies the structure of request and reply messages that the service can consume and produce. In most cases, the data format specified in the WSDL is different than the internal format of the application providing the service. Effectively, the WSDL specifies a *Canonical Data Model* to be used by both parties participating in the conversation. The double-translation consists of a *Messaging Mapper* or a *Gateway* in the service consumer and a *Remote Facade* [GoF] in the service provider.

Example: TIBCO ActiveEnterprise

Many EAI tool suites provide a complete set of tools to define and describe the canonical data format. For example, the TIBCO ActiveEnterprise suite provides the TIB/Designer that allows the user to inspect all common message definitions. Message definitions can be imported from or exported to XML schema definitions. When implementing a *Message Translator* using built-in visual tool set, the tool presents the designer with both the application specific data format and the *Canonical Data Model* stored in the central data format repository.



The TIBCO Designer - A UI Tool to Maintain a Canonical Data Model

Related patterns: *Format Indicator*, *Message Channel*, *Message Router*, *Introduction to Message Routing*, *Message Translator*, *Messaging*, *Messaging Mapper*, *Shared Database*

10. Messaging Endpoints

Introduction

In *Introduction to Messaging Systems*, we discussed *Message Endpoint*. This is how an application connects to a messaging system so that it can send and receive messages. As an application programmer, when you program to a messaging API such as JMS or the `System.Messaging` namespace, you're developing endpoint code. If you are using a commercial middleware package, most of this coding is already done for you using the libraries and tools provided by the vendor.

Send and Receive Patterns

Some endpoint patterns apply to both senders and receivers. They concern how the application relates to the messaging system in general.

Encapsulate the messaging code — In general, an application should not be aware that it is using *Messaging* to integrate with other applications. Most of the application's code should be written without messaging in mind. At the points where the application integrate with others, there should be a thin layer of code that performs the application's part of the integration. When the integration is implemented with messaging, that thin layer of code that attches the application to the messaging system is a *Messaging Gateway*.

Data translation — It's great when the internal data representation the sender and receiver applications use are the same, and when the message format uses that same representation as well. However, this is often not the case. Either the sender and receiver disagree on data format, or the messages use a different format (usually to support other senders and receivers). In this situation, use a *Messaging Mapper* to convert data between the application's format and the message's format.

Externally-controlled transactions — Messaging systems use transactions internally; externally, by default, each `send` or `receive` method call runs in its own transaction. However, message producers and consumers have the option of using a *Transactional Client* to control these transactions externally, which is useful when you need to batch together multiple messages or to coordinate messaging with other transactional services.

Message Consumer Patterns

Other endpoint patterns only apply to message receivers. Sending messages is easy. There are issues involved in deciding when a message should be sent, what it should contain, and how to

communicate its intent to the receiver—that's why we have the Message Construction patterns (see *Introduction to Message Construction*)—but once the message is built, sending it is easy. Receiving messages, on the other hand—that's tricky. So many of these patterns are about receiving messages.

An overriding theme in message consumption is *throttling*, which means the application controlling the rate at which it consumes messages. As discussed in the *Introduction*, a potential problem any server faces is that a high volume of client requests could overload the server. With *Remote Procedure Invocation*, the server is pretty much at the mercy of the rate that clients make calls. Likewise, with *Messaging*, the server cannot control the rate at which clients send requests—but the server can control the rate at which it processes those requests. The application does not have to receive and process the messages as rapidly as they're delivered by the messaging system; it can process them at a sustainable pace and let the extras queue up to be processed in some sort of first come, first serve basis. On the other hand, if the messages are piling up too much and the server has the resources to process more messages faster, the server can increase its message consumption throughput using concurrent message consumers. So use these patterns to let your application control the rate at which it consumes messages.

Many of these patterns come in pairs that represent alternatives. You can design an endpoint one way or the other. A single application may design some endpoints one way and some endpoints the other way, but a single endpoint can only implement one alternative. Alternatives from each pair can be combined, leading to a great number of choices for how to implement a particular endpoint.

Synchronous or asynchronous consumer — One alternative is whether to use a *Polling Consumer* or an *Event-Driven Consumer*. [JMS11, pp.68-69], [Hapner, p.13], [Dickman, p.29] Polling provides the best throttling because if the server is busy, it won't run the code to receive more messages, so the messages will queue up. Consumers that are event-driven tend to process messages as fast as they arrive, which could overload the server; but each consumer can only process one message at a time, so limiting the number of consumers effectively throttles the consumption rate.

Message assignment vs. message grab — Another alternative concerns how a handful of consumers process a handful of messages. If each consumer gets a message, they can process the messages concurrently. The simplest approach is *Competing Consumers*, where one *Point-to-Point Channel* has multiple consumers. Each one could potentially grab any message; the messaging system's implementation decides which consumer gets a message. If you want to control this message-to-consumer matching process, use a *Message Dispatcher*. This is a single consumer that receives a message but delegates it to a performer for processing. An application can throttle message load by limiting the number of consumers/performers. Also, the dispatcher in a *Message Dispatcher* can implement explicit throttling behavior.

Accept all messages or filter — By default, any message delivered on a *Message Channel* becomes available to any *Message Endpoint* listening on that channel for messages to consume. However, some consumers may not want to consume just any message on that channel, but only wish to consume messages of a certain type or description. Such a discriminating consumer can use a

Selective Consumer to describe what sort of message it's willing to receive. Then the messaging system will only make messages matching that description available to that receiver.

Subscribe while disconnected — An issue that comes up with *Publish-Subscribe Channel*s: What if a subscriber was interested in the data being published on a particular channel and will be again, but is currently disconnected from the network or shut down for maintenance? Will a disconnected application miss messages published while it is disconnected, even though it has subscribed? By default, yes, a subscription is only valid while the subscriber is connected. To keep the application from missing messages published inbetween connections, make it a *Durable Subscriber*.

Idempotency — Sometimes the same message gets delivered more than once, either because the messaging system is not certain the message has been successfully delivered yet, or because the *Message Channel*'s quality-of-service has been lowered to improve performance. Message receivers, on the other hand, tend to assume that each message will be delivered exactly once, and tend to cause problems when they repeat processing because of repeat messages. A receiver designed as an *Idempotent Receiver* handles duplicate messages and prevents them from causing problems in the receiver application.

Synchronous or asynchronous service — Another tough choice is whether an application should expose its services to be invoked synchronously (via *Remote Procedure Invocation*) or asynchronously (via *Messaging*). Different clients may prefer different approaches; different circumstances may require different approaches. Since it's often hard to choose just one approach or the other, let's have both. A *Service Activator* connects a *Message Channel* to a synchronous service in an application so that when a message is received, the service is invoked. Synchronous clients can simply invoke the service directly; asynchronous clients can invoke the service by sending a message.

Message Endpoint Themes

Another significant theme in this chapter is difficulty using *Transactional Client* with other patterns. *Event-Driven Consumer* usually cannot externally control transactions properly, *Message Dispatcher* must be carefully designed to do so, and *Competing Consumers* that externally manage transactions can run into significant problems. The safest bet for using *Transactional Client* is with a single *Polling Consumer*, but that may not be a very satisfactory solution.

Special mention should be made of JMS-style message-driven beans, one type of Enterprise JavaBeans (EJB). [EJB20, pp.311-326], [Hapner, pp.69-74] An MDB is a message consumer that is an *Event-Driven Consumer*, a *Transactional Client* that supports J2EE distributed (e.g., `XAResource`) transactions, and can be dynamically pooled as *Competing Consumers*, even for a *Publish-Subscribe Channel*. This is a difficult and tedious combination to implement in one's own application code, but this functionality is provided as a ready-built feature of compatible EJB containers (such as BEA's WebLogic and IBM's WebSphere). (How is the MDB framework implemented? Essentially, the container implements a *Message Dispatcher* with a dynamicly-sized pool of reusable

performers, where each performer consumes the message itself using its own session and transaction.)

Finally, keep in mind that a single *Message Endpoint* may well combine several different patterns from this chapter. A group of *Competing Consumers* may be implemented as *Polling Consumer*s that are also *Selective Consumer*s and act as a *Service Activator* on a service in the application. A *Message Dispatcher* may be an *Event-Driven Consumer* and a *Durable Subscriber* that uses a *Messaging Mapper*. Whatever other patterns an endpoint implements, it should also be a *Messaging Gateway*. So don't think of what one pattern to use, think of the combinations. That's the beauty of solving the problems with patterns.

So there are a lot of options for making an application into a *Message Endpoint*. This chapter will explain what those options are and how to make the best use of them.

Messaging Gateway

An application accesses another system via *Messaging*.

How do you encapsulate access to the messaging system from the rest of the application?

Most custom applications access the messaging infrastructures through a vendor-supplied API. While there are many different flavors of such API's, these libraries generally expose similar functions, such as "open channel", "create message", "send message". While this type of API allows the application to send any kind of message data across any kind of channel, it is sometimes hard to tell what the intent of sending the message data is.

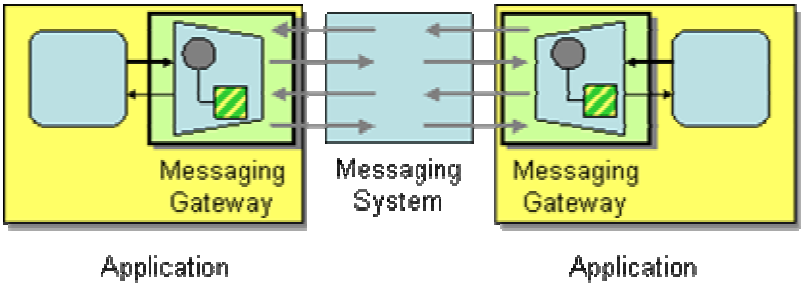
Messaging solutions are inherently asynchronous. This can complicate the code to access an external function over messaging. Instead of calling a method `GetCreditScore` that returns the numeric credit score, the application has to send the request message and expect the reply message to arrive at a later time (see *Request-Reply*). The application developer may prefer the simple semantics of a synchronous function as opposed dealing with incoming message events.

Loose coupling between applications provides architectural advantages, such as resilience to minor changes in message formats (i.e., adding fields). Usually, the loose coupling is achieved by using XML documents or other data structures that are not strongly typed like a Java or C# class. Coding against such structures is tedious and error-prone because there is no compile-type support to detect misspelled field names or mismatching data types. Therefore, we often gain the flexibility in data formats at the expense of application development effort.

Sometimes, a simple logical function to be executed via messaging requires more than one message to be sent. For example, a function to get customer information may in reality require multiple messages, one to get the address, another to get the order history and yet another to get personal information. Each of these messages my be processed by a different system. We would not want to clutter the application code with all the logic required to send and receive three

separate message. We could take some of the burden off the application by using a *Scatter-Gather* that receives a single message, sends three messages separate messages and aggregates them back into a single reply message. However, not always do we have the luxury of adding this function to the messaging middleware.

Use a *Messaging Gateway*, a class than wraps messaging-specific method calls and exposes domain-specific methods to the application.



The *Messaging Gateway* encapsulates messaging-specific code (e.g., the code required to send or receive a message) and separates it from the rest of the application code. This way, only the *Messaging Gateway* code knows about the messaging system; the rest of the application code does not. The *Messaging Gateway* exposes a business function to the rest of the application so that instead of requiring the application to set properties like `Message.MessageReadPropertyFilter.AppSpecific`, a *Messaging Gateway* exposes methods such as `GetCreditScore` that accept strongly typed parameters just like any other method. A *Messaging Gateway* is a messaging-specific version of the more general *Gateway* pattern [EAA].

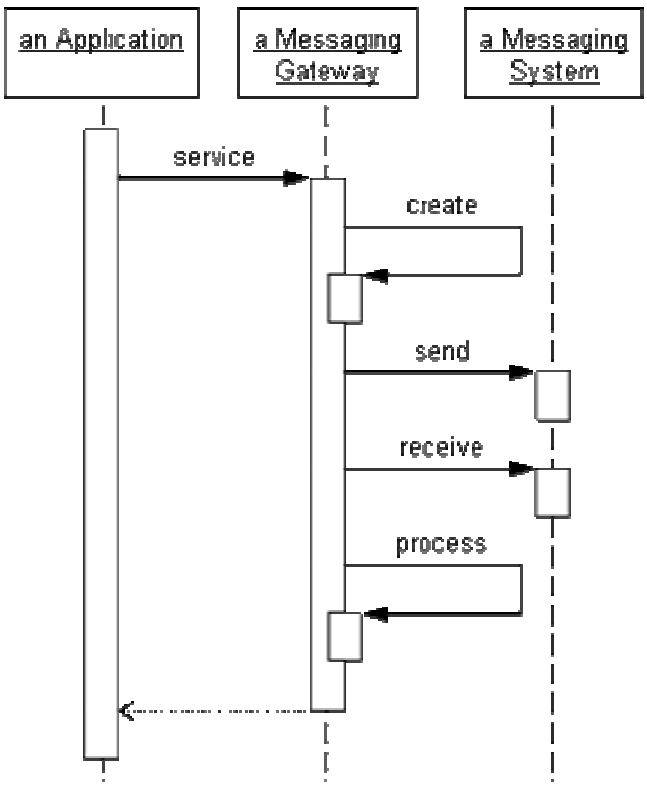
A Gateway Eliminates Direct Dependencies Between the Application and the Messaging Systems

A *Messaging Gateway* sits between the application and the messaging system and provides a domain-specific API to the application (see picture). Because the application doesn't even know that it's using a messaging system, we can swap out the gateway with a different implementation that uses another integration technology, such as remote procedure calls or Web services.

Many *Messaging Gateway*s send a message to another component and expect a reply message (*Request-Reply*). Such a *Messaging Gateway* can be implemented in two different ways:

- Blocking (Synchronous) *Messaging Gateway*
- Event-Driven (Asynchronous) *Messaging Gateway*

A blocking *Messaging Gateway*s sends out a message and waits for the reply message to arrive before returning control to the application. When the gateway receives the reply, it processes the message and returns the result to the application (see sequence diagram).



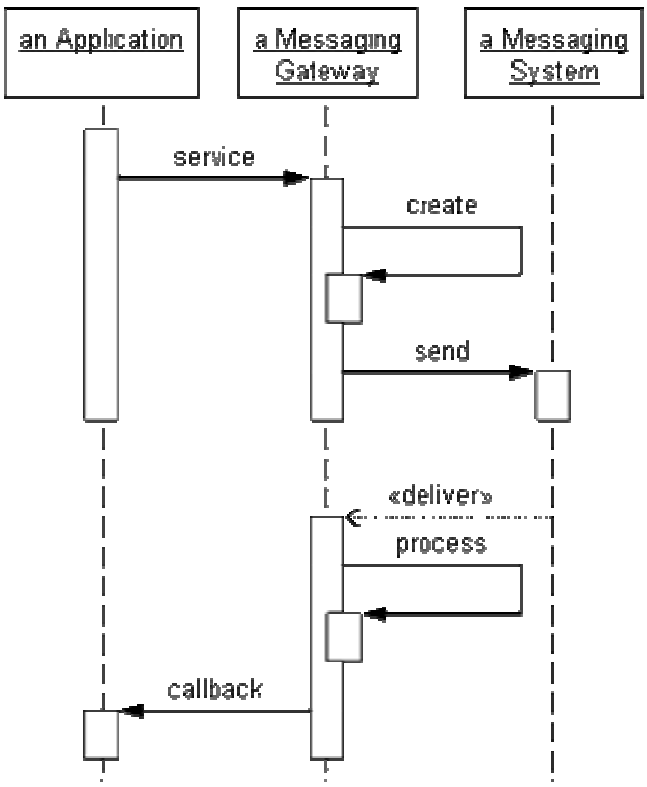
Blocking (Synchronous) Messaging Gateway

A blocking *Messaging Gateway* encapsulates the asynchronous nature of the messaging interaction, exposing a regular synchronous method to the application logic. Thus, the application is unaware of any asynchronicity in the communication. For example, a blocking gateway may expose the following method:

```
int GetCreditScore(string SSN);
```

While this approach makes writing application code against the *Messaging Gateway* very simple, it can also lead to poor performance because the application ends up spending most of its time sitting around and waiting for reply messages while it could be performing other tasks.

An event-driven *Messaging Gateway* exposes the asynchronous nature of the messaging layer to the application. When the application makes the domain-specific request to the *Messaging Gateway* it provides a domain-specific callback for the reply. Control returns immediately to the application. When the reply message arrives, the *Messaging Gateway* processes it and then invokes the callback (see sequence diagram).



Event-Driven (Asynchronous) Messaging Gateway

For example, in C# using delegates, the *Messaging Gateway* could expose the following public interface:

```
delegate void OnCreditReplyEvent(int CreditScore);
void RequestCreditScore(string SSN, OnCreditReplyEvent OnCreditResponse);
```

The method `RequestCreditScore` accepts an additional parameter that specifies the callback method to be invoked when the reply message arrives. The callback method has a parameter `CreditScore` so that the *Messaging Gateway* can pass the results to the application. Depending on the programming language or platform, the callback can be accomplished with function pointers, object references or delegates (as shown here). Note that despite the event-driven nature of this interface there is no dependency at all on a specific messaging technology.

Alternatively, the application can periodically poll to see whether the results arrived. This approach makes the higher-level interface simple without introducing blocking, essentially employing the Half-sync/Half-async pattern [POSA2]. This pattern describes the use of buffers that store incoming messages so that the application can poll at its convenience to see whether a message has arrived.

One of the challenges of using an event-driven *Messaging Gateway* is that the *Messaging Gateway* requires the application to maintain state between the request method and the callback event (the call stack takes care of this in the blocking case). When the *Messaging Gateway* invokes the callback event into the application logic, the application needs to be able to correlate the reply with the request it made earlier so that it can continue processing the correct thread of execution.

The *Messaging Gateway* can make it easier for the application to maintain state if it allows the application to pass a reference to an arbitrary set of data to the request method. The *Messaging Gateway* will then pass this data back to the application with the callback. This way, the application has all necessary data available when the asynchronous callback is invoked. This type of interaction is commonly called ACT (Asynchronous Completion Token) [POSA2].

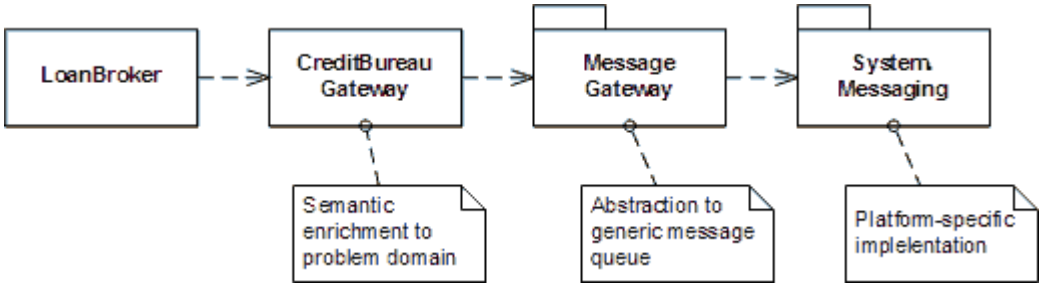
The public interface of an event-driven *Messaging Gateway* that supports an ACT may look like this:

```
delegate void OnCreditReplyEvent(int CreditScore, Object ACT);
void RequestCreditScore(string SSN, OnCreditReplyEvent OnCreditResponse, Object ACT);
```

While supporting an ACT is a very convenient feature for the application, it does introduce the danger of a memory leak if the *Messaging Gateway* maintains a reference to an object but the expected reply message never arrives.

Chaining Gateways

It can be beneficial to create more than one layer of *Messaging Gateway*s. The "lower-level" *Messaging Gateway* can simply abstract the syntax of the messaging system but maintain generic messaging semantics, e.g. "SendMessage". This *Messaging Gateway* can help shield the rest of the application when the enterprise changes messaging technologies, e.g. from MSMQ to Web Services. We wrap this basic *Messaging Gateway* with an additional *Messaging Gateway* that translates the generic messaging API into a narrow, domain-specific API, such as `GetCreditScore`. We use this configuration in the MSMQ implementation of the Loan Broker example (see *Asynchronous Implementation with MSMQ*, figure below).



A Chain of Gateways Provides Different Levels of Abstraction

Dealing With Messaging Exceptions

Besides making coding the application simpler, the intent of the *Messaging Gateway* is also to eliminate dependencies of the application code on specific messaging technologies. This is easy to do by wrapping any messaging-specific method calls behind the *Messaging Gateway* interface. However, most messaging layers tend to throw messaging-specific exceptions, e.g. the `InvalidDestinationException` exception raised by JMS. If we really want to make our application

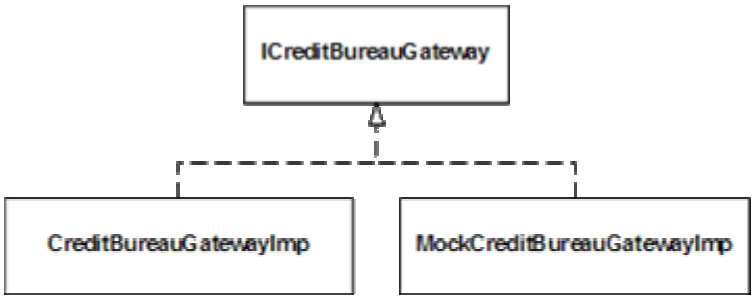
code independent from the messaging library, the *Messaging Gateway* has to catch any messaging specific exception and throw an application-specific (or a generic) exception instead. This code can get a little tedious, but it is very helpful if we ever have to switch the underlying implementations, e.g. from JMS to Web services.

Generating Gateways

In many situations, we can generate the *Messaging Gateway* code from metadata exposed by the external resource. This is common on the world of Web services. Almost every vendor or open-source platform provides a tool such as `wsdl2java` that connects to the Web Service Description Language (WSDL) exposed by an external web service. The tool generates Java (or C# or whatever language you need) classes that encapsulate all the nasty SOAP stuff and expose a simple function call. We created a similar tool that can read message schema definitions off the TIBCO repository and creates Java source code for a class that mimics the schema definition. This allows application developers to send correctly-typed TIBCO ActiveEnterprise messages without having to learn the TIBCO API.

Using Gateways for Testing

*Messaging Gateway*s make great testing vehicles. Because we wrapped all the nasty messaging code behind a narrow, domain specific interface, we can easily create a dummy implementation of this interface. We simply separate interface and implementation and provide two implementations: one "real" implementation that accesses the messaging infrastructure and a "fake" implementation for testing purposes (see picture). The "fake" implementations acts as a *Service Stub* (see [EAA]) and allows us to test the application without any dependency on messaging. A *Service Stub* can also be useful to debug an application that uses an event-driven *Messaging Gateway*. For example, a simple test stub for an event-driven *Messaging Gateway* can simply invoke the callback (or delegate) right from the request method, effectively executing both the request and the response processing in one thread. This can simplify step-by-step debugging enormously.



Gateways as a Testing Tool

Example: Asynchronous Loan Broker Gateway in MSMQ

This example shows a piece of the Loan Broker example introduced in the Composed Messaging Interlude (see *Asynchronous Implementation with MSMQ*).

```
public delegate void OnCreditReplyEvent(CreditBureauReply creditReply, Object ACT);

internal struct CreditRequestProcess
{
   public int CorrelationID;
   public Object ACT;
   public OnCreditReplyEvent callback;
}

internal class CreditBureauGateway
{
   protected IMessageSender creditRequestQueue;
   protected IMessageReceiver creditReplyQueue;

   protected IDictionary activeProcesses = (IDictionary)(new Hashtable());

   protected Random random = new Random();

   public void Listen()
   {
      creditReplyQueue.Begin();
   }

   public void GetCreditScore(CreditBureauRequest quoteRequest, OnCreditReplyEvent
OnCreditResponse, Object ACT)
   {
      Message requestMessage = new Message(quoteRequest);
      requestMessage.ResponseQueue = creditReplyQueue.GetQueue();
      requestMessage.AppSpecific = random.Next();

      CreditRequestProcess processInstance = new CreditRequestProcess();
      processInstance.ACT = ACT;
      processInstance.callback = OnCreditResponse;
      processInstance.CorrelationID = requestMessage.AppSpecific;

      creditRequestQueue.Send(requestMessage);

      activeProcesses.Add(processInstance.CorrelationID, processInstance);
   }
```

```
   private void OnCreditResponse(Message msg)
   {
      msg.Formatter =  GetFormatter();

      CreditBureauReply replyStruct;
      try
      {
         if (msg.Body is CreditBureauReply)
         {
            replyStruct = (CreditBureauReply)msg.Body;
            int CorrelationID = msg.AppSpecific;

            if (activeProcesses.Contains(CorrelationID))
            {
               CreditRequestProcess processInstance =
(CreditRequestProcess)(activeProcesses[CorrelationID]);
               processInstance.callback(replyStruct, processInstance.ACT);
               activeProcesses.Remove(CorrelationID);
            }
            else { Console.WriteLine("Incoming credit response does not match any
request"); }
         }
         else
         { Console.WriteLine("Illegal reply."); }
      }
      catch (Exception e)
      {
         Console.WriteLine("Exception: {0}", e.ToString());
      }
   }
}
```

You will notice that the public method `GetCreditScore` and the public delegate `OnCreditReplyEvent` make no references to messaging at all. This implementation allows the calling application to pass an arbitrary object reference as an Asynchronous Completion Token. The `CreditBureauGateway` stores this object reference in a dictionary indexed by the *Correlation Identifier* of the request message. When the reply message arrives, the `CreditBureauGateway` can retrieve the data that was associated with the outbound request message. The calling application does not have to worry about how the messages are correlated.

Example: Synchronous Gateway in JMS

Related patterns: *Scatter-Gather*, *Asynchronous Implementation with MSMQ*, *Correlation Identifier*, *Messaging*, *Request-Reply*

Messaging Mapper

When integrating applications using messaging, the data inside a message is often derived from domain objects inside the integrated applications. If we use a *Document Message*, the message itself may directly represent one or domain objects. If we use a *Command Message*, some of the data fields associated with the command are likely to be extracted from domain objects as well. There are some distinct differences between messages and objects. For example, most objects rely on associations in the form of object references and inheritance relationships. Many messaging infrastructures do not support these concepts because they have to be able to communicate with a range of applications, some of which may not be object-oriented at all.

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

Why can't we make our messages look exactly like the domain objects and make the problem go away? In many cases, we are not in control of the message format because it is defined by a *Canonical Data Model* or a common messaging standard (e.g. ebXML). We could still publish the message in a format that corresponds to the domain object and use a *Message Translator* inside the messaging layer to make the necessary transformation to the common message format. This approach is commonly used by adapters to third-party systems that do not allow transformation inside the application (e.g., a database adapter).

Alternatively, the domain can create and publish a message in the required format without the need for a separate *Message Translator*. This option most likely results in better performance because we do not publish an intermediate message. Also, if our domain model contains many small objects it may be beneficial to combine them into a single message first to simplify routing and improve efficiency inside the messaging layer. Even if we can afford the additional transformation step, we will run into limitations if we want to create messages that mimic domain objects. The shortcoming of this approach is that the domain must know the message format, which makes domain maintenance difficult if multiple formats can be used or if the format changes.

Most messaging infrastructures support the notion of a "Message" object as part of the application programming interface (API). This message object encapsulates the data to be sent over a channel. In most cases, this message object can contain only scalar data types such as strings, numbers, or dates, but does not support inheritance or object references. This is one of the key differences between RPC-style communications (i.e. RMI) and asynchronous messaging systems. Let's assume we send an asynchronous message containing an object reference to a

component. In order to process the message, the component would have to resolve the object reference. It would do this by requesting the object from the message source. However, request-reply interaction would defeat some of the motivations of using asynchronous messaging in the first place, i.e. loose coupling between components. Worse yet, by the time the asynchronous message is received by the subscriber, the referenced object may no longer exist in the source system.

One attempt to resolve the issue of object references is to traverse the dependency tree of an object and include all dependent objects in the message. For example, if an `Order` object references five `OrderItem` objects, we would include the five objects in the message. This will ensure that the receiver has access to all data references by the "root" object. If we use a fine-grained domain object model in which many objects are interrelated, messages can quickly explode in size. It would desirable to have more control over what is included in a message and what is not.

Let's assume for a moment that our domain object is self-contained and does not have any references to other objects. We still cannot simply stick the whole domain object into a message as most messaging infrastructures do not support objects because they have to be language independent (the JMS interface `ObjectMessage` and the `Message` class inside .NET's `System.Messaging` namespace are exceptions since these messaging systems are either language (Java) or platform (.NET CLR) specific). We could think of serializing the object into a string and store it into a string field called "data," which is supported by pretty much every messaging system. However, this approach has disadvantages as well. First, a *Message Router* would not be able to use object properties for routing purposes because this string field would be 'opaque' to the messaging layer. It would also make testing and debugging difficult, because we would have to decipher the contents of the 'data' field. Also, constructing all messages so that they just contain a single string field would not allow us to route messages by message type because to the infrastructure all messages look the same. It would also be difficult to verify the correct format of the message because the messaging infrastructure would not verify anything inside the "data" field. Lastly, we would not be able to use the serialization facilities provided by the language run-time libraries because these presentation are usually not compatible across languages. So we would have to write our own serialization code.

Some messaging infrastructures now support XML fields inside messages so that we could serialize objects into XML. This can alleviate some of the disadvantages because the messages are easier to decipher now and some messaging layers can access elements inside and XML string directly. However, we now have to deal with quite verbose messages and limited data type validation. Plus, we still have to create code that translates an object into XML and back. Depending on the programming language we use, this could be quite complex, especially if we use an older language that does not support reflection.

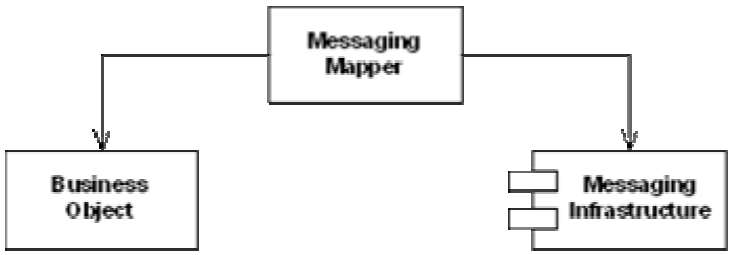
We are well advised to separate this mapping code from the domain object for a number of reasons. First of all, we may not want to blend code that concerns itself with low-level language features with application logic. In many cases, we will have a group of programmers dedicated to working with the messaging layer, while another group focuses on the domain logic. Sticking both pieces of code into one object will make it difficult for the teams to work in parallel.

Second, incorporating mapping code inside the domain object makes the domain object dependent on the messaging infrastructure because the mapping code will need to make calls into the messaging API (e.g. to instantiate the `Message` object). In most cases this dependency is not desirable because it prevents the reuse of the domain objects in another context that does not use messaging or uses another vendor's messaging infrastructure. As a result, we would seriously impede the reusability of the domain objects.

We often see people write "abstraction layers" that wrap the messaging infrastructure API, effectively making the code that deals with messaging independent from the messaging API. Such a layer provides a level of indirection because it separates the messaging interface from the messaging implementation. Therefore, we can reuse the messaging-related code even if we have to switch to another vendor's messaging layer. All we need to do is implement a new abstraction layer that translates the messaging interface to the new API. However, this approach does not resolve the dependency of the domain objects on the messaging layer. The domain objects would now contain references to the abstracted messaging interface as opposed to the vendor-specific messaging API. But we still cannot use the domain objects in a context that does not use messaging.

Many messages are composed of more than one domain object. Since we cannot pass object references through the messaging infrastructure, it is likely that we need to include fields from other objects. In some cases, we may include the whole "dependency tree" of all dependent objects inside one message. Which class should hold the mapping code? The same object may be part of multiple message types, combined with different objects, so there is no easy answer to this question.

Create a separate *Messaging Mapper* that contains the mapping logic between the messaging infrastructure and the domain objects. Neither the objects nor the infrastructure have knowledge of the *Messaging Mapper*'s existence.



The *Messaging Mapper* accesses one or more domain objects and converts them into a message as required by the messaging channel. It also performs the opposite function, creating or updating domain objects based on incoming messages. Since the *Messaging Mapper* is implemented as a separate class that references the domain object(s) and the messaging layer, neither layer is aware of the other. The layers don't even know about the *Messaging Mapper*.

The *Messaging Mapper* is a specialization of the *Mapper* pattern defined in [EAA]. It shares some analogies with the *Data Mapper* defined in the same book. Anyone who has worked on O-R

(Object-Relational) Mapping strategies will understand the complexities of mapping data between layers that use different paradigms. The issues inherent in the *Messaging Mapper* are similarly complex and a detailed discussion of all possible aspects is beyond the scope of this book. Many of the Data Source Architectural Patterns in [EAA] make a good read for anyone concerned with creating a *Messaging Mapper* layer.

A *Messaging Mapper* is different from the frequently used concept of an abstraction layer wrapped around the messaging API. In the case of an abstraction layer, the domain objects do not know about the messaging API, but they do know about the abstraction layer (the abstraction layer essentially performs the function of a *Gateway* (see [EAA]). In the case of a *Messaging Mapper*, the objects have no idea whatsoever that we are dealing with messaging.

The intent of a *Messaging Mapper* is similar to that of a Mediator ([GoF]) which is also used to separate elements. In the case of a Mediator, though, the elements are aware of the Mediator whereas neither element is aware of the *Messaging Mapper*.

If neither the domain objects nor the messaging infrastructure know about the *Messaging Mapper* how does it get invoked? In most cases, the *Messaging Mapper* is invoked through events triggered either by the messaging infrastructure or the application. Since neither one is dependent on the *Messaging Mapper* the event notification can either happen through a separate piece of code or by making the *Messaging Mapper* an *Observer* pattern (see [GoF]). For example, if we use the JMS API to interface with the messaging infrastructure we can implement the `MessageListener` interface to be notified of any incoming messages. Likewise, we can use an *Observer* to be notified of any relevant events inside the domain objects and to invoke the *Messaging Mapper*. If we have to invoke the *Messaging Mapper* directly from the application, we should define a *Messaging Mapper* interface so that the application does at least not depend on the *Messaging Mapper* implementation.

Reducing the Coding Burden

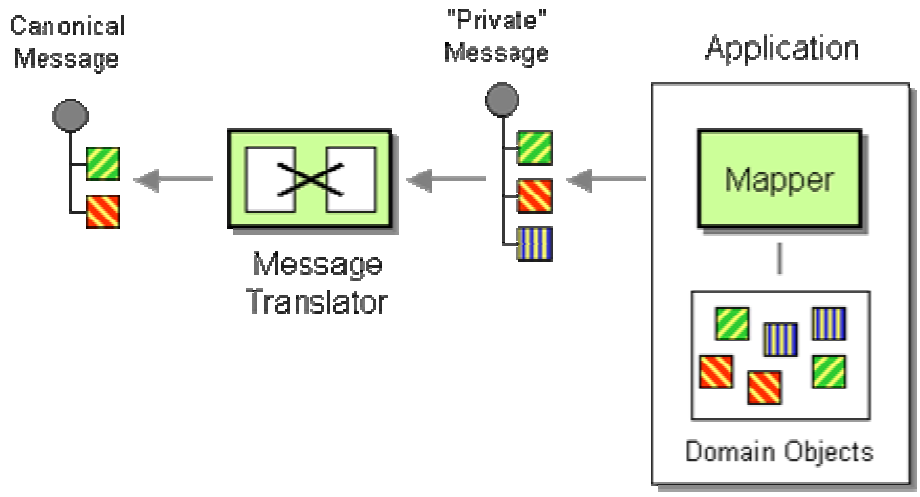
Some *Messaging Mapper* implementations may contain a lot of repetitive code: get a field from the domain object and store it in a message object. Go on to the next field and repeat until all fields are done. This can be pretty tedious and also smells suspiciously like code duplication. We have a number of tools to help us avoid this tedium. First, we can write generic a *Messaging Mapper* that uses reflection to extract fields from a domain object in a generic way. For example, it could traverse the list of all fields inside the domain object and store it in a field of the same name in the message object. Obviously, this works only if the field names match. According to our previous discussions, we need to come up with some way to resolve object references since we cannot store those in the message object. The alternative is to use a configurable code generator to generate the *Messaging Mapper* code. This allows us some more flexibility in the field naming (the message field name and the domain object field name do not have to match and we can device clever ways to deal with object references. The downside of code generators is that they can be difficult to test and debug, but if we make it generic enough we have to write it only once.

Some frameworks, such as Microsoft .NET, feature built-in object serialization of objects into XML and vice versa and take away of lot of the grunt work involved in object serialization. Even if the framework does some of the legwork of converting an object into a message, this conversion is limited to the 'syntactic' level of translation. It might be tempting to just let the framework do all the work, but it will just create messages that correspond to the domain objects one-to-one. As we explained earlier, this may not be desirable because the constraints and design criteria for messages are quite different from those for domain objects. It may make sense to define a set of 'interface objects' that correspond to the desired messages structure and let the framework do the conversion between the messages and these objects. The *Messaging Mapper* layer will then manage the translation between the true domain objects and the interface objects. These interface objects bear some resemblance to *Data Transfer Objects* ([EAA]) even though the motivations are slightly different.

Mapper vs. Translator

Even if we use a *Messaging Mapper* it still makes sense to use a *Message Translator* to translate the messages generated by the *Messaging Mapper* into messages compliant with the *Canonical Data Model*. This gives us an additional level of indirection. We can use the *Messaging Mapper* to resolve issues such as object references and data type conversions and leave structural mappings to a *Message Translator* inside the messaging layer. The price we pay for this additional decoupling is the creation of an additional component and a small performance penalty. Also, sometimes it is easier to perform complex transformations inside the application's programming language as opposed to the drag-and-drop 'doodleware' supplied by the integration vendor.

If we use both a *Messaging Mapper* and a *Message Translator* we gain an additional level of indirection between the canonical data format and the domain objects. It has been said that computer science is the area where every problem can be solved by adding just one more level of indirection, so does this rule hold true here? The additional indirection gives us the ability to compensate for changes in the canonical model inside the messaging layer without having to touch application code. It also allows us to simplify the mapping logic inside the application by leaving tedious field mappings and data type changes (e.g., a numeric 'ZIP_Code' field to an alphanumeric 'Postal_Code' field) to the mapping tools of messaging layer that are optimized for this kind of work. The *Messaging Mapper* would then primarily deal with the resolution of object references and the elimination of unnecessary domain object detail. The apparent downside of the extra level of indirection is that a change in the domain object may now require changes to both the *Messaging Mapper* and the *Message Translator*. If we did manage to generate the *Messaging Mapper* code, this issue largely goes away.



Combining Mapper and Message Translator

Example: Messaging Mapper in JMS

The `AuctionAggregate` class in the *Aggregator* JMS example acts as a *Messaging Mapper* between the JMS messaging system and the `Bid` class. The methods `addMessage` and `getResultMessage` convert between JMS messages and `Bid` objects. Neither the messaging system nor the `Bid` class have any knowledge of this interaction.

Related patterns: *Aggregator*, *Canonical Data Model*, *Command Message*, *Document Message*, *Message Router*, *Message Translator*

Transactional Client

A messaging system, by necessity, uses transactional behavior internally. It may be useful for an external client to be able to control the scope of the transactions that impact its behavior.

How can a client control its transactions with the messaging system?

A messaging system must use transactions internally. A single *Message Channel* can have multiple senders and multiple receivers, so the messaging system must coordinate the messages to make sure senders don't overwrite each other's *Message*s, multiple *Point-to-Point Channel* receivers don't receive the same message, multiple *Publish-Subscribe Channel* receivers each receive one copy of each message, etc. To manage all of this, messaging systems internally use transactions to make sure a message gets added to the channel or not, and gets read from the channel or not. Messaging systems also have to employ transactions—preferably two-phase, distributed transactions—to copy a message from the sender's computer to the receiver's computer, such that at any given time, the message is "really" only on one computer or the other.

*Message Endpoint*s sending and receiving messages are transactional, even if they don't realize it. The `send` method that adds a message to a channel does so within a transaction to isolate that message from any other messages simultaneously being added to or removed from that channel. Likewise, a `receive` method also uses a transaction, which prevents other point-to-point receivers from getting the same message, and even assures that a publish-subscribe receiver won't read the same message twice. (Transactions are often described as being ACID: atomic, consistent, isolated, and durable. Only transactions for *Guaranteed Delivery* are durable, and a message by definition is atomic. But all messaging transactions have to be consistent and isolated. A message can't be sort of in the channel, it either is or isn't. And an application's sending and receiving of messages has to be isolated from whatever other sending and receiving other threads and applications might be doing.)

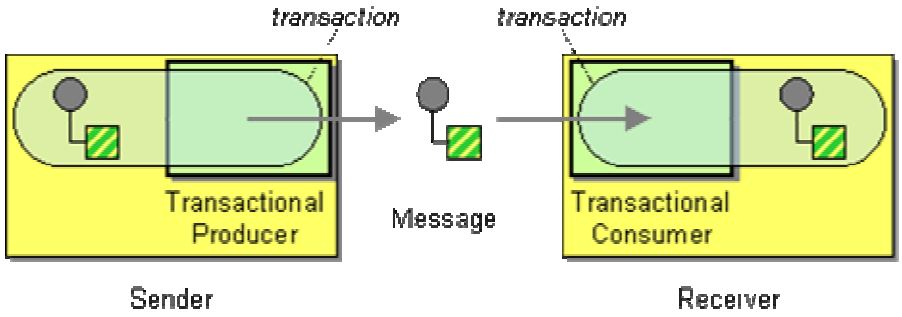
The messaging system's internal transactions are sufficient and convenient for a client that simply wants to send or receive a single message. However, an application may need a broader transaction to coordinate several messages or to coordinate messaging with other resources. Common scenarios like this include:

- **Send-Receive Message Pairs –** Receive one message and send another, such as a *Request-Reply* scenario or when implementing a message filter such as a *Message Router* or *Message Translator*.
- **Message Groups –** Send or receive a group of related messages, such as a *Message Sequence*.
- **Message/Database Coordination –** Combine sending or receiving a message with updating a database, such as with a *Channel Adapter*. For example, when an application receives and processes a message for ordering a product, the application will also need to update the product inventory database. Likewise, the sender of a *Document Message* may wish to delete a persisted document, but only when it is sent successfully; the receiver may want to persist the document before the message is truly considered to be consumed.
- **Message/Workflow Coordination –** Use a pair of *Request-Reply* messages to perform a work item, and use transactions to ensure that the work item isn't acquired unless the request is also sent, and the work item isn't completed or aborted unless the reply is also received.

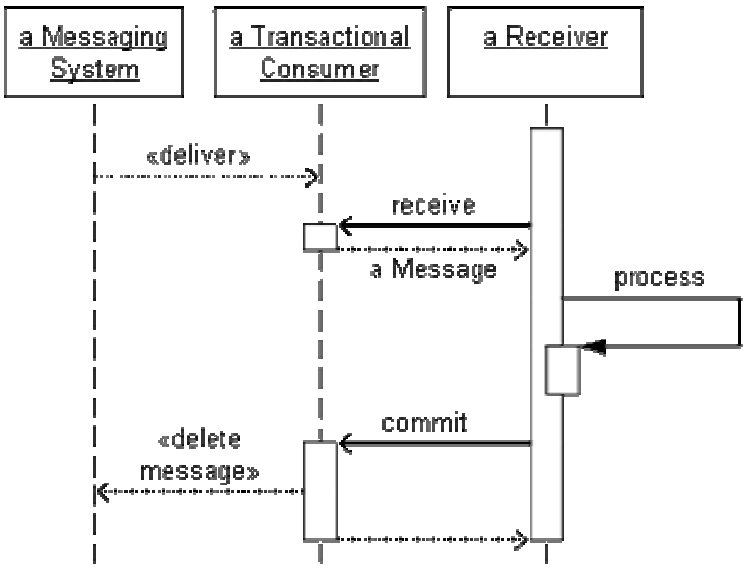
Scenarios like these require a larger atomic transaction that involve more than just a single message and may involve other transactional stores besides the messaging system. A transaction is required so that if part of the scenario works (receiving the message, for example) but another part does not (such as updating the database, or sending another message), all parts can be rolled back as if they never happened, and then the application can try again.

Yet a messaging system's internal transaction model is insufficient to allow an application to coordinate handling a message with other messages or other resources. What is needed is a way for the application to externally control the messaging system's transactions and combine them with other transactions in the messaging system or elsewhere.

Use a *Transactional Client*—make the client's session with the messaging system transactional so that the client can specify transaction boundaries.



Both a sender and a receiver can be transactional. With a sender, the message isn't "really" added to the channel until the sender commits the transaction. With a receiver, the message isn't "really" removed from the channel until the receiver commits the transaction. A sender that uses explicit transactions can be used with a receiver that uses implicit transactions, and vise versa. A single channel might have a combination of implicitly and explicitly transactional senders; it could also have a combination of receivers.



Transactional Receiver Sequence

With a transactional receiver, an application can receive a message without actually removing the message from the queue. At this point, if the application crashed, when it recovered, the message would still be on the queue; the message would not be lost. Having received the message, the application can then process it. Once the application is finished with the message and is certain it wants to consume it, the application commits the transaction, which (if successful) removes the message from the channel. At this point, if the application crashed, when it recovered, the message would no longer be on the channel, so the application had better truly be finished with the message.

How does controlling a messaging system's transactions externally help an application coordinate several tasks? Here's what the application would do in the scenarios described above:

Send-Receive Message Pairs

- **What to do**: Start a transaction, receive and process the first message, create and send the second message, then commit.
- **What this does**: This keeps the first message from being removed from its channel until the second message is successfully added to its channel.
- **Transaction type**: If the two messages are sent via channels in the same messaging system, the transaction encompassing the two channels is a simple one. However, if the two channels are managed by two separate messaging systems, such as with a *Messaging Bridge*, the transaction will be a distributed one coordinating the two messaging systems.
- **Warning**: A single transaction only works for the receiver of a request sending a reply. The sender of a request cannot use a single transaction to send a request and wait for its reply. If it tries to do this, the request will never really be sent—because the send transaction isn't committed—so the reply will never be received.

Message Groups

- **What to do**: Start a transaction, send or receive all of the messages in the group (such as a *Message Sequence*), then commit.
- **What this does**: When sending, none of the messages in the group will be added to the channel until they are all successfully sent. When receiving, none of the messages will be removed from the channel until all are received.
- **Transaction type**: Since all of the messages are being sent to or received from a single channel, that channel will be managed by a single messaging system, so the transaction will be a simple one. Also, in many messaging system implementations, sending a group of messages in a single transaction ensures that they will be received on the other end of the channel in the order they were sent.

Message/Database Coordination

- **What to do**: Start a transaction, receive a message, update the database, then commit. Or update the database and send a message to report the update to others, then commit. (This behavior is often implemented by a *Channel Adapter*.)
- **What this does**: The message will not be removed unless the database is updated (or the database change will not stick if the message cannot be sent).
- **Transaction type**: Since the messaging system and the database each has its own transaction manager, the transaction to coordinate them will be a distributed one.

Message/Workflow Coordination

- **What to do**: Use a pair of *Request-Reply* messages to perform a work item. Start a transaction, acquire the work item, send the request message, then commit. Or start

another transaction, receive the reply message, complete or abort the work item, then commit.

- **What this does**: The work item will not be committed unless the request is sent; the reply will not be removed unless the work item is updated.
- **Transaction type**: Since the messaging system and the workflow engine each has its own transaction manager, the transaction to coordinate them will be a distributed one.

In this way, the application can assure that it will not loose the messages it receives, nor forget to send a message that it should. If something goes wrong in the middle, the application can roll back the transaction and try again.

Transactional clients using *Event-Driven Consumer*s may not work as expected. The consumer typically must commit the transaction for receiving the message before passing the message to the application. Then if the application examines the message and decides it does not want to consume it, or if the application encounters an error and wants to rollback the consume action, it cannot because it does not have access to the transaction. So an event-driven consumer tends to work the same whether or not its client is transactional.

Messaging systems are capable of participating in a distributed transaction, although some implementations may not support it. In JMS, a provider can act as an XA resource and participate in Java Transaction API [JTA] transactions. This behavior is defined by the "XA" classes in the `javax.jms` package, particularly `javax.jms.XASession`, and by the `javax.transaction.xa` package. The JMS spec recommends that JMS clients not try to handle distributed transactions directly, so an application should use the distributed transaction support provided by a J2EE application server. MSMQ can also participate in an XA transaction; this behavior is exposed in .NET by the `MessageQueue.Transactional` property and the `MessageQueueTransaction` class.

As discussed earlier, transactional clients can be useful as part of other patterns like *Request-Reply*, message filters in *Pipes and Filters*, *Message Sequence*, and *Channel Adapter*. Likewise, the receiver of an *Event Message* may want to complete processing the event before removing its message from the channel completely. On the other hand, *Transactional Client*s do not work well with *Event-Driven Consumer*s nor *Message Dispatcher*s, can cause problems for *Competing Consumers*, but work well with a single *Polling Consumer*.

Example: JMS Transacted Session

In JMS, a client makes itself transactional when it creates its session. [JMS11, p.64], [Hapner, pp.64-66]

```
Connection connection = // Get the connection
Session session =
        connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

Setting the first `createSession` parameter to `true` makes the session transactional.

When a client is using a transactional session, it must explicitly commit sends and receives to make them real.

```
Queue queue = // Get the queue
MessageConsumer consumer = session.createConsumer(queue);
Message message = consumer.receive();
```

At this point, the message has only been consumed in the consumer's transactional view. But to other consumers with their own transactional views, the message is still available.

```
session.commit();
```

Now, assuming that the commit message does not throw any exceptions, the consumer's transactional view becomes the message system's, which now considers the message consumed.

Example: .NET Transactional Queue

In .NET, queues are not transactional by default. So to use a transactional client, the queue must be made transactional when it is created:

```
MessageQueue.Create("MyQueue", true);
```

Once a queue is transactional, each client action (send or receive) on the queue can be transactional or non-transactional. A transactional receive looks like this:

```
MessageQueue queue = new MessageQueue("MyQueue");
MessageQueueTransaction transaction =
        new MessageQueueTransaction();
transaction.Begin();
Message message = queue.Receive(transaction);
transaction.Commit();
```

Although the client had received the message, the messaging system did not make the message unavailable on the queue until the client committed the transaction successfully. [SysMsg]

Example: Tranactional Filter with MSMQ

The following example enhances the basic filter component introduced in *Pipes and Filters* to use transactions. This example implements the **Send-Receive Message Pair** scenario, receiving and sending a message inside the same transaction. We really have to add only a few lines of code to make the filter transactional. We use a variable of type `MessageQueueTransaction` to manage the transaction. We open a transaction before we consume the input smessage and commit after we publish the output message. If any exception occurs we abort the transaction which rolls back all

message consumption and publication actions and returns the input message to the queue to be available to other queue consumers.

```
public class TransactionalFilter
{
   protected MessageQueue inputQueue;
   protected MessageQueue outputQueue;
   protected Thread receiveThread;
   protected bool stopFlag = false;

   public TransactionalFilter (MessageQueue inputQueue, MessageQueue outputQueue)
   {
      this.inputQueue = inputQueue;
      this.inputQueue.Formatter =  new System.Messaging.XmlMessageFormatter(new
String[] {"System.String,mscorlib"});
      this.outputQueue = outputQueue;
   }

   public void Process()
   {
      ThreadStart receiveDelegate = new ThreadStart(this.ReceiveMessages);
      receiveThread = new Thread(receiveDelegate);
      receiveThread.Start();
   }

   private void ReceiveMessages()
   {
      MessageQueueTransaction myTransaction = new MessageQueueTransaction();

      while (!stopFlag)
      {
         try
         {
            myTransaction.Begin();
            Message inputMessage = inputQueue.Receive(myTransaction);
            Message outputMessage = ProcessMessage(inputMessage);
            outputQueue.Send(outputMessage, myTransaction);
            myTransaction.Commit();
         }
         catch (Exception e)
         {
            Console.WriteLine(e.Message + " - Transaction aborted ");
            myTransaction.Abort();
         }
      }
```

```
    }


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

How do we verify that our *Transactional Client* works as intended? We subclass the basic `TransactionalFilter` with the aptly named class `RandomlyFailingFilter`. For each consumed message, this filter draws a random number between 0 and 10. If the number is less than 3, it throws an an arbitrary exception (`ArgumentNullException` seemed convenient enough for an example). If we implemented this filter on top of our basic, non-transactional filter described in *Pipes and Filters*, we would lose about one in three messages.

```
public class RandomlyFailingFilter : TransactionalFilter
{
    Random rand = new Random();

    public RandomlyFailingFilter(MessageQueue inputQueue, MessageQueue outputQueue) :
base (inputQueue, outputQueue) { }

    protected override Message ProcessMessage(Message m)
    {
        string text = (string)m.Body;
        Console.WriteLine("Received Message: " + text);

        if (rand.Next(10) < 3)
        {
            Console.WriteLine("EXCEPTION");
            throw (new ArgumentNullException());
        }
        if (text == "end")
            stopFlag = true;
        return(m);
    }
}
```

To make sure that we do in fact not lose any messages with the transactional version we rigged up a really simple test harness that publishes a sequence of messages to the input queue and makes sure that it can receive all messages in the correct order from the output queue. It is important to remember that the output messages remain in sequence only if we run a single

instance of the transactional filter. If we run multiple filters in parallel, messages can (and will) get out of order (see *Resequencer*.

```
public void RunTests()
{
   MessageQueueTransaction myTransaction = new MessageQueueTransaction();

   for (int i=0; i < messages.Length; i++)
   {
      myTransaction.Begin();
      inQueue.Send(messages[i], myTransaction);
      myTransaction.Commit();
   }

   for (int i=0; i < messages.Length; i++)
   {
      myTransaction.Begin();
      Message message = outQueue.Receive(new TimeSpan(0,0,3), myTransaction);
      myTransaction.Commit();

      String text = (String)message.Body;
      Console.Write(text);
      if (text == messages[i])
         Console.WriteLine(" OK");
      else
         Console.WriteLine(" ERROR");
   }

   Console.WriteLine("Hit enter to exit");
   Console.ReadLine();
}
```

Related patterns: *Channel Adapter*, *Competing Consumers*, *Document Message*, *Event-Driven Consumer*, *Event Message*, *Guaranteed Delivery*, *Message*, *Message Channel*, *Message Dispatcher*, *Message Endpoint*, *Message Router*, *Message Sequence*, *Message Translator*, *Messaging Bridge*, *Pipes and Filters*, *Point-to-Point Channel*, *Polling Consumer*, *Publish-Subscribe Channel*, *Request-Reply*, *Resequencer*

Polling Consumer

An application needs to consume *Message*s, but it wants to control when it consumes each message.

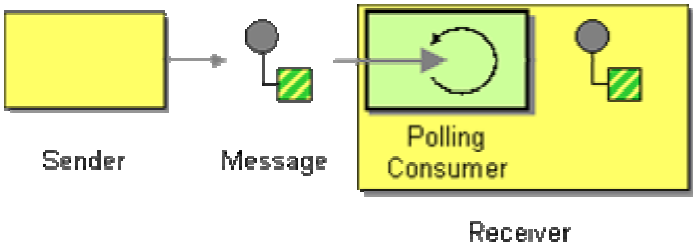
How can an application consume a message when the application is ready?

Message consumers exit for one reason—to consume messages. The messages represent work that needs to be done, so the consumer needs to consume those messages and do the work.

But how does the consumer know when a new message is available? The easiest approach is for the consumer to repeatedly check the channel to see if a message is available. When a message is available, it consumes the message, and then goes back to checking for the next one. This process is called *polling*.

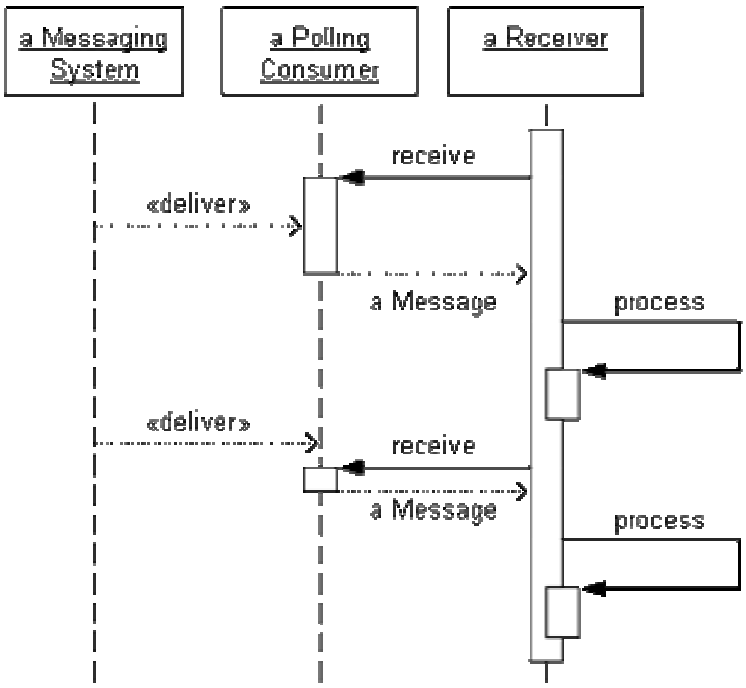
The beauty of polling is that the consumer can request the next message when it is ready for another message. So it consumes messages at the rate it wants to, rather than at the rate they arrive in the channel.

The aplication should use a *Polling Consumer*, one that explicitly makes a call when it wants to receive a message.



This is also known as a synchronous receiver, because the receiver thread blocks until a message is received. We call it a *Polling Consumer* because the receiver polls for a message, processes it, then polls for another. As a convenience, messaging API's usually provide a `receive` method that blocks until a message is delivered, in addition to methods like `receiveNoWait()` and `Receive(0)` that return immediately if no message is available. This difference is only apparent when the receiver is polling faster than messages are arriving.

A *Polling Consumer* is an object that an application uses to receive messages by explicitly requesting them. When the application is ready for another message, it polls the consumer, which in turn gets a message from the messaging system and returns it. (How the consumer gets the message from the messaging system is implementation-specific and may or may not involve polling. All the application knows is that it doesn't get the message until it explicitly asks for one.)



Polling Consumer Sequence

When the application polls for a message, the consumer blocks until it gets a message to return (or until some other condition is met, such as a time limit). Once the application receives the message, it can then process it. Once it is through processing the message and wishes to receive another, the application can poll again.

By using polling consumers, an application can control how many messages are consumed concurrently by limiting the number of threads that are polling. This can help keep the receiving application from being overwhelmed by too many requests; extra messages queue up until the receiver can process them.

A receiver-application typically uses one thread (at least) per channel that it wishes to monitor, but it can also use a single thread to monitor multiple channels, to help conserve threads when monitoring channels that are frequently empty. To poll a single channel, assuming that the thread has nothing to do until a message arrives, use a version of `receive` that blocks until a message arrives. To poll multiple channels with a single thread, or to perform other work while waiting for a message to arrive, use a version of `receive` with a timeout or `receiveNoWait()` so that if one channel is empty, the thread goes on to check another channel or perform other work.

A consumer that is polling too much or blocking threads for too long can be inefficient, in which case an *Event-Driven Consumer* may be more efficient. Multiple *Polling Consumer*s can be *Competing Consumers*. A *Message Dispatcher* can be implemented as a *Polling Consumer*. A *Polling Consumer* can be a *Selective Consumer*; it can also be a *Durable Subscriber*. A *Polling Consumer* can also be a *Transactional Client* so that the consumer can control when the message is actually removed from the channel.

Example: JMS Receive

In JMS, a message consumer uses `MessageConsumer.receive` to consume a message synchronously. [JMS11, p.69], [Hapner, p.21]

`MessageConsumer` has three different receive methods:

1. `receive()` – Blocks until a message is available, and then returns it.
2. `receiveNoWait()` – Checks once for a message, and returns it or null.
3. `receive(long)` – Blocks either until a message is available and returns it, or until the time-out expires and returns null.

For example, the code to create a consumer and receive a message is very simple:

```
Destination dest = // Get the destination
Session session = // Create the session
MessageConsumer consumer = session.createConsumer(dest);
Message message = consumer.receive();
```

Example: .NET Receive

In .NET, a consumer uses `MessageQueue.Receive` to consume a message synchronously. [SysMsg]

A `MessageQueue` client has several variations of receive. The two simplest are:

1. `Receive()` – Blocks until a message is available, and then returns it.
2. `Receive(TimeSpan)` – Blocks either until a message is available and returns it, or until the time-out expires and throws `MessageQueueException`.

The code to receive a message from an existing queue is quite simple:

```
MessageQueue queue = // Get the queue
Message message = queue.Receive();
```

Related patterns: *Competing Consumers*, *Durable Subscriber*, *Event-Driven Consumer*, *Message*, *Message Dispatcher*, *Selective Consumer*, *Transactional Client*

Event-Driven Consumer

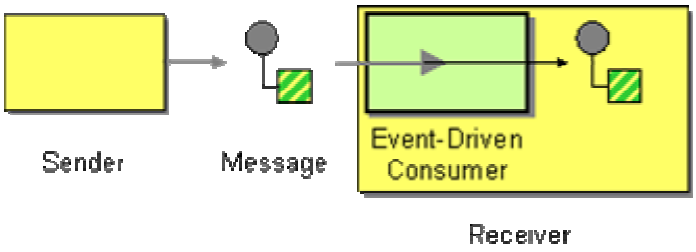
An application needs to consume *Message*s as soon as they're delivered.

How can an application automatically consume messages as they become available?

The problem with *Polling Consumer*s is that when the channel is empty, the consumer blocks threads and/or consumes process time while polling for messages that are not there. Polling enables the client to control the rate of consumption, but wastes resources when there's nothing to consume.

Rather than continuously asking the channel if it has messages to consume, it would be better if the channel could tell the client when a message is available. For that matter, instead of making the consumer poll for the message to get the message, just give the message to the consumer as soon as the message becomes available.

The application should use an *Event-Driven Consumer*, one that is automatically handed messages as they're delivered on the channel.



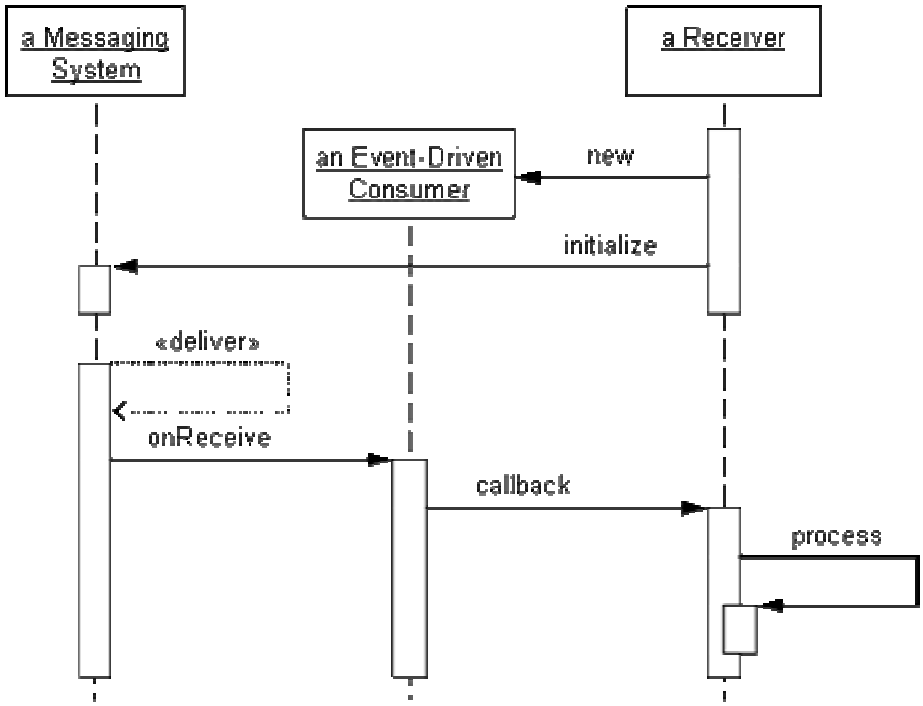
This is also known as an asynchronous receiver, because the receiver does not have a running thread until a callback thread delivers a message. We call it an *Event-Driven Consumer* because the receiver acts like the message delivery is an event that triggers the receiver into action.

An *Event-Driven Consumer* is an object that is invoked by the messaging system when a message arrives on the consumer's channel. The consumer passes the message to the application through a callback in the application's API. (How the messaging system gets the message is implementation-specific and may or may not be event-driven. All the consumer knows is that it can sit dormant with no active threads until it gets invoked by the messaging system passing it a message.)

An *Event-Driven Consumer* is invoked by the messaging system, yet it invokes an application-specific callback. To bridge this gap, the consumer has an application-specific implementation that conforms to a known API defined by the messaging system.

The code for an *Event-Driven Consumer* consists of two parts:

1. **Initialization –** The application creates an application-specific consumer and associates it with a particular *Message Channel*. After this code is run once, the consumer is ready to receive a series of messages.
2. **Consumption –** The consumer receives a message, which it and the application process. This code is run once per message being consumed.



Event-Driven Consumer Sequence

The application creates its custom consumer and associates it with the channel. Once the consumer is initialized, it (and the application) can then go dormant, with no running threads, waiting to be invoked when a message arrives.

When a message is delivered, the messaging system calls the consumer's message-received-event method and passes in the message as a parameter. The consumer passes the message to the application using the application's callback API. The application now has the message and can process it. Once the application finishes processing the message, it and the consumer can then go dormant again until the next message arrives. Typically, a messaging system will not run multiple threads through a single consumer, so the consumer can only process one message at a time.

*Event-Driven Consumer*s automatically consume messages as they become available. For more fine-grained control of the consumption rate, use a *Polling Consumer*. *Event-Driven Consumer*s can be *Competing Consumers*. A *Message Dispatcher* can be implemented as a *Event-Driven Consumer*. A *Event-Driven Consumer* can be a *Selective Consumer*; it can also be a *Durable Subscriber*. *Transactional Client*s may not work as well with event-driven consumers as they do with polling consumers (see the JMS example).

Example: JMS MessageListener

In JMS, an *Event-Driven Consumer* is a class that implements the `MessageListener` interface. [Hapner, p.22] This interface declares a single method, `onMessage(Message)`. The consumer implements `onMessage` to process the message. Here is an example of a JMS performer:

```
public class MyEventDrivenConsumer implements MessageListener {
        public void onMessage(Message message) {
                // Process the message
        }
}
```

The initializer part of an *Event-Driven Consumer* creates the desired performer object (which is a `MessageListener` instance) and associates it with a message consumer for the desired channel:

```
Destination destination = // Get the destination
Session session = // Create the session
MessageConsumer consumer = session.createConsumer(destination);
MessageListener listener = new MyEventDrivenConsumer();
consumer.setMessageListener(listener);
```

Now, when a message is delivered to the destination, the JMS provider will call `MyEventDrivenConsumer.onMessage` with the message as a parameter.

Note that in JMS, an *Event-Driven Consumer* that is also a *Transactional Client* will not work as expected. Normally a transaction is rolled back when the code in the transaction throws an exception, but the `MessageListener.onMessage` signiture does not provide for an exception being thrown (such as `JMSException`), and a runtime exception is considered programmer error. If a runtime exception occurs, the JMS provider responds by delivering the next message, so the message that caused the exception is lost. [JMS11, p.69], [Hapner, p.22] To successfully achieve transaction, event-driven behavior, use a message-driven EJB. [EJB20, pp.311-326], [Hapner, pp.69-71]

Example: .NET ReceiveCompletedEventHandler

With .NET, the performer part of an *Event-Driven Consumer* implements a method that is a `ReceiveCompletedEventHandler` delegate. This delegate method must accept two parameters: an `Object` that is the `MessageQueue`, and a `ReceiveCompletedEventArgs` that is the arguments from the `ReceiveCompleted` event. [SysMsg] The method uses the arguments to get the message from the queue and process it. Here is an example of a .NET performer:

```
public static void MyEventDrivenConsumer(Object source,
        ReceiveCompletedEventArgs asyncResult)
{
        MessageQueue mq = (MessageQueue) source;
        Message m = mq.EndReceive(asyncResult.AsyncResult);
        // Process the message
        mq.BeginReceive();
        return;
}
```

The initializer part of an event-driven client specifies that the queue should run the delegate method to handle a `ReceiveCompleted` event:

```
MessageQueue queue = // Get the queue
queue.ReceiveCompleted +=
        new ReceiveCompletedEventHandler(MyEventDrivenConsumer);
queue.BeginReceive();
```

Now, when a message is delivered to the queue, the queue will issue a `ReceiveCompleted` event, which will run the `MyEventDrivenConsumer` method.

Related patterns: *Competing Consumers*, *Durable Subscriber*, *Message*, *Message Channel*, *Message Dispatcher*, *Selective Consumer*, *Polling Consumer*, *Transactional Client*

Competing Consumers

An application is using *Messaging*. However, it cannot process messages as fast as they're being added to the channel.

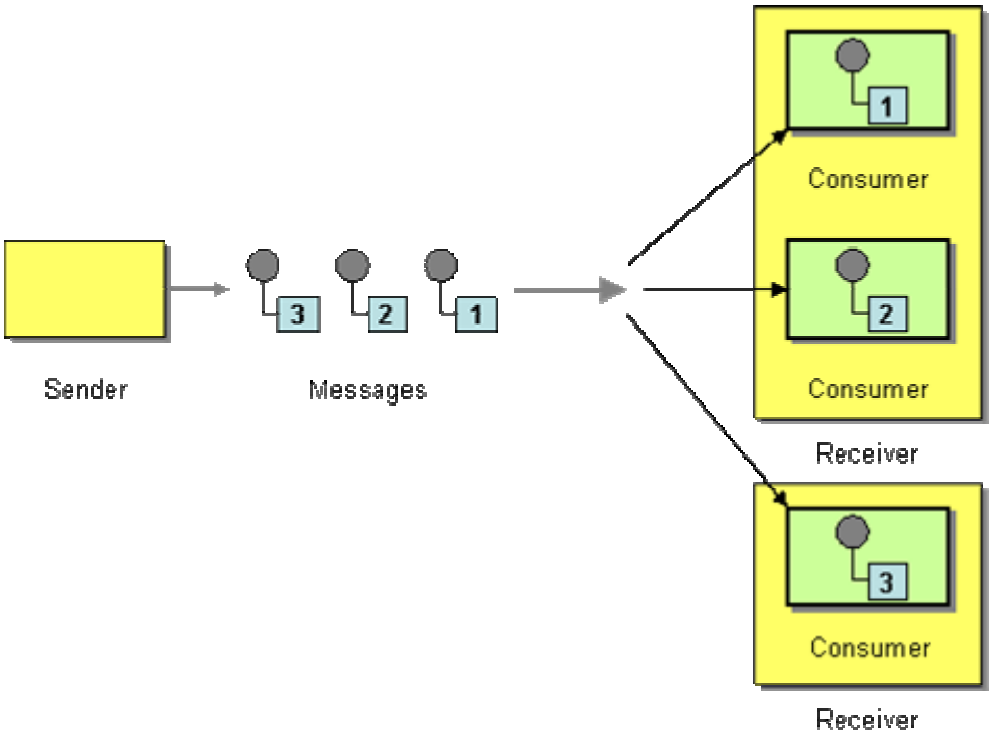
How can a messaging client process multiple messages concurrently?

*Message*s arrive through a *Message Channel* sequentially, so the natural inclination of a consumer is to process them sequentially. However, sequential consumption may be too slow and messages may pile up on the channel, which makes the messaging system a bottleneck and hurts overall throughput of the application. This can happen either because of multiple senders on the channel, because a network outage causes a backlog of messages which are then delivered all at once, because a receiver outage causes a backlog, or because each message takes significantly more effort to consume and perform than it does to create and send.

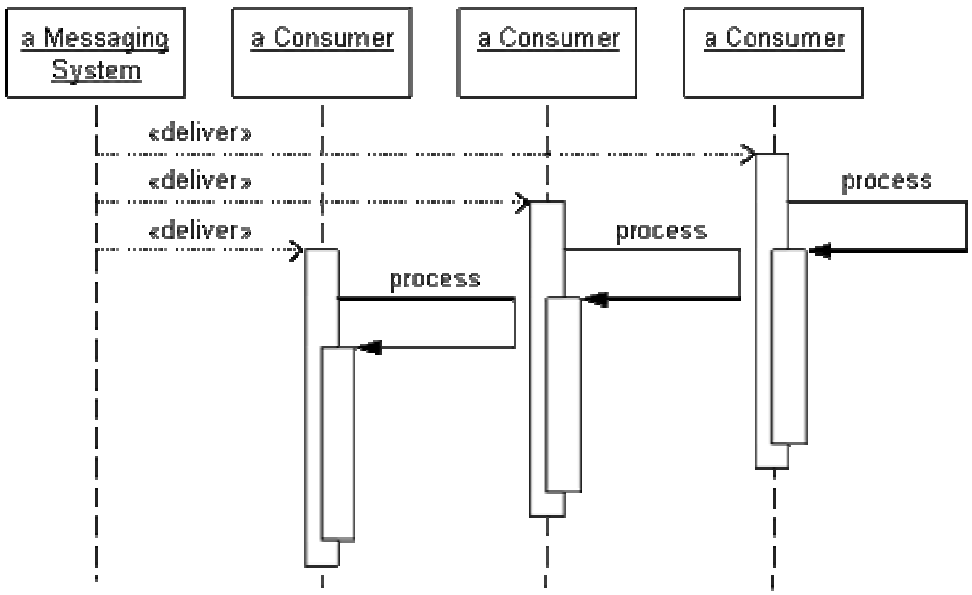
The application could use multiple channels, but one channel might become a bottleneck while another sits empty, and a sender would not know which one of equivalent channels to use. Multiple channels would have the advantage, however, of enabling multiple consumers (one per channel), processing messages concurrently. Even if this worked, though, the number of channels the application defined would still limit the throughput.

What is needed is a way for a channel to have multiple consumers.

Create multiple *Competing Consumers* on a single channel so that the consumers can process multiple messages concurrently.



Competing Consumers are multiple consumers that are all created to receive messages from a single *Point-to-Point Channel*. When the channel delivers a message, any of the consumers could potentially receive it. The messaging system's implementation determines which consumer actually receives the message, but in effect the consumers compete with each other to be the receiver. Once a consumer receives a message, it can delegate to the rest of its application to help process the message. (This solution only works with *Point-to-Point Channel*s; multiple consumers on a *Publish-Subscribe Channel* just create more copies of each message.)



Competing Consumers Sequence

Each of the *Competing Consumers* runs in its own thread so that they all can consume messages concurrently. When the channel delivers a message, the messaging system's transactional controls ensure that only one of the consumers successfully receives the message. While that consumer is processing the message, the channel can deliver other messages, which other consumers can concurrently consume and process. The channel coordinates the consumers, making sure that they each receives a different message; the consumers do not have to coordinate with each other.

Each consumer processes a different message concurrently, so the bottleneck becomes how quickly the channel can feed messages to the consumers instead of how long it takes a consumer to process a message. A limited number of consumers may still be a bottleneck, but increasing the number of consumers can alleviate that constraint as long as there are available computing resources.

To run concurrently, each consumer must run in its own thread. For *Polling Consumer*s, this means that each consumer must have its own thread to perform the polling concurrently. For *Event-Driven Consumer*s, the messaging system must use a thread per concurrent consumer; that thread will be used to hand the message to the consumer, and will be used by the consumer to process the message.

A sophisticated messaging system will detect competing consumers on a channel and internally provide a *Message Dispatcher* that ensures that each message is only delivered to a single consumer. This helps avoid conflicts that would arise if multiple consumers each thought they were the consumer of a single message. A less-sophisticated messaging system will allow multiple consumers to attempt to consume the same message. When this happens, whichever consumer commits its transaction first wins; then the other consumers will not be able to commit succesfully and will have to roll back their transactions.

A messaging system that allows multiple consumers to attempt consuming the same message can make a *Transactional Client* very inefficient. The client thinks it has a message, consumes it, spends effort processing the message, then tries to commit and cannot (because the message has already been consumed by a competitor). Frequently performing work just to roll it back hurts throughput, whereas the point of this solution is to increase throughput. Thus the performance of competing transactional consumers should be measured carefully; it could vary significantly on different messaging system implementations and configurations.

Not only can *Competing Consumers* be used to spread load across multiple consumer threads in a single application; they can also spread the consumption load across multiple applications. This way, if one application cannot consume messages fast enough, multiple consumer applications--perhaps with each employing multiple consumer threads--can attack the problem. The ability to have multiple applications running on multiple computers using multiple threads to consume messages provides virtually unlimited message processing capacity, where the only limit is the messaging system's ability to deliver messages from the channel to the consumers.

The coordination of competing consumers depends on each messaging system's implementation. If the client wants to implement this coordination itself, it should use a *Message Dispatcher*. *Competing Consumers* can be *Polling Consumer*s, *Event-Driven Consumer*s, or a combination thereof. Competing *Transactional Client*s can waste significant effort processing messages whose receive operations do not commit successfully and have to be rolled back.

Example: Simple JMS Competing Consumers

This is a simple example of how to implement a competing consumer in Java. An external driver/manager object (not shown) runs a couple of them. It runs each one in its own thread and calls `stopRunning()` to make it stop.

A JMS session must be single-threaded. [JMS11, pp.26-27], [Hapner, p.19] A single session serializes the order of message consumption. [JMS11, p.60], [Hapner, p.19] So for each competing consumer to work properly in its own thread, and for the consumers to be able to consume messages in parallel, each consumer must have its own `Session` (and therefore its own `MessageConsumer`). The JMS specification does not specify the semantics of how concurrent `QueueReceivers` (e.g., *Competing Consumers*) should work, or even require that this approach work at all. Thus applications which use this technique are not assumed to be portable and may work differently with different JMS providers. [JMS11, p.65], [Hapner, p.347]

The consumer class implements `Runnable` so that it can run in its own thread; this allows the consumers to run concurrently. All of the consumers share the same `Connection` but each creates its own `Session`, which is important since each session can only support a single thread. Each consumer repeatedly receives a message from the queue and processes it.

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.naming.NamingException;

public class CompetingConsumer implements Runnable {

        private int performerID;
        private MessageConsumer consumer;
        private boolean isRunning;

        protected CompetingConsumer() {
                super();
        }
```

```
        public static CompetingConsumer newConsumer(int id, Connection connection,
String queueName)
                throws JMSException, NamingException {
                CompetingConsumer consumer = new CompetingConsumer();
                consumer.initialize(id, connection, queueName);
                return consumer;
        }

        protected void initialize(int id, Connection connection, String queueName)
                throws JMSException, NamingException {
                performerID = id;
                Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
                Destination dispatcherQueue = JndiUtil.getDestination(queueName);
                consumer = session.createConsumer(dispatcherQueue);
                isRunning = true;
        }

        public void run() {
                try {
                        while (isRunning())
                                receiveSync();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        private synchronized boolean isRunning() {
                return isRunning;
        }

        public synchronized void stopRunning() {
                isRunning = false;
        }

        private void receiveSync() throws JMSException, InterruptedException {
                Message message = consumer.receive();
                if (message != null)
                        processMessage(message);
        }

        private void processMessage(Message message) throws JMSException,
InterruptedException {
                int id = message.getIntProperty("cust_id");
```

```
                System.out.println(System.currentTimeMillis() + ": Performer #" +
performerID + " starting; message ID " + id);
                Thread.sleep(500);
                System.out.println(System.currentTimeMillis() + ": Performer #" +
performerID + " processing.");
                Thread.sleep(500);
                System.out.println(System.currentTimeMillis() + ": Performer #" +
performerID + " finished.");
        }
}
```

So implementing a simple competing consumer is easy. The main trick is to make the consumer a `Runnable` and run it in its own thread.

Related patterns: *Event-Driven Consumer*, *Message*, *Message Channel*, *Message Dispatcher*, *Messaging*, *Point-to-Point Channel*, *Polling Consumer*, *Publish-Subscribe Channel*, *Transactional Client*

Message Dispatcher

An application is using *Messaging*. The application needs multiple consumers on a single *Message Channel* to work in a coordinated fashion.

How can multiple consumers on a single channel coordinate their message processing?

Multiple consumers on a single *Point-to-Point Channel* act as *Competing Consumers*. That's fine when the consumers are interchangeable, but it does not allow for specializing the consumers so that certain consumers are better able to consume certain messages.

Multiple consumers on a single *Publish-Subscribe Channel* won't work as intended. Rather than distribute the message load, these consumers will duplicate the effort.

*Selective Consumer*s can be used as specialized consumers. However, not all messaging systems support this feature. Even amongst those that do, they may not support selection based on values in the body of the message. Their selector value expressions may be too simple to adequately distinguish amongst the messages, or the performance of repeatedly evaluating those expressions may be slow. There may be numerous expressions which need to be carefully designed in a coordinated fashion so that they do not overlap but also do not leave any selector values unhandled. They may need to implement a default case for selector values that are not handled by other consumers or that are unexpected.

*Datatype Channel*s can be used to keep different types of messages separate and to enable consumers to specialize for those message types. But the type system may be too large and varied to justify creating a separate channel for each type. Or the types may be based on dynamically

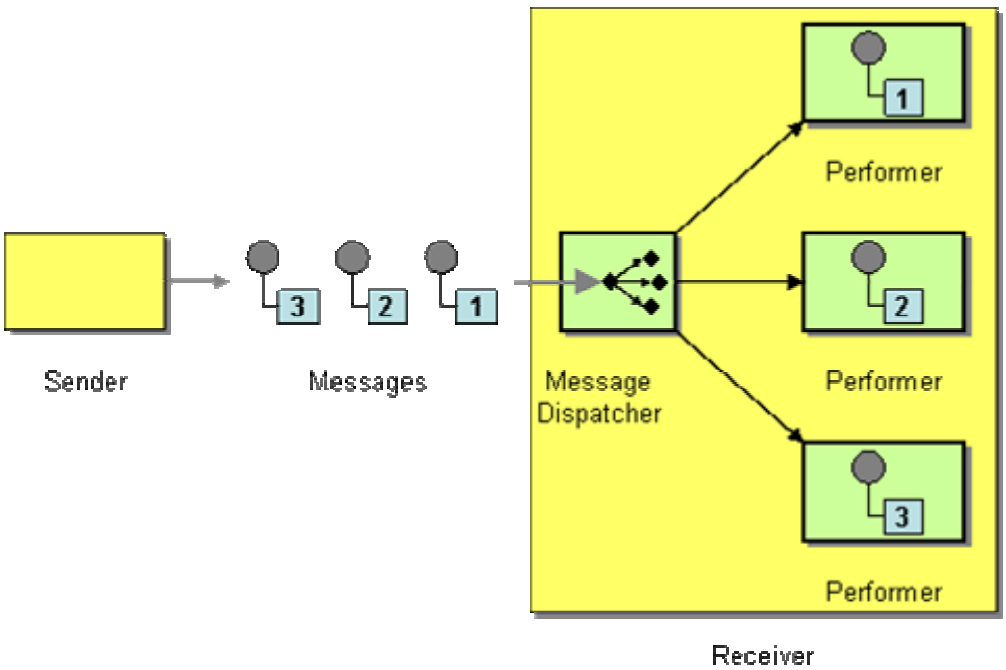
changing criteria which are difficult to handle with a static set of channels. The enterprise may already require a huge number of channels, taxing the messaging system, and multiplying many of those channels for distinct message types may simply require too many channels.

Each of these problems could be solved if consumers could work together. They could avoid duplicating work by being aware if another consumer had already processed that work. They could be specialized; if a consumer got the wrong kind of message for its speciality, it could hand off the message to another consumer with the right speciality. If an application had too many channels coming in, it could save channels by having all of its consumers share a single channel; they'd coordinate to make sure that the right messages went to the right consumers.

Alas, consumers are very independent objects that are difficult to coordinate. Making specialized consumers general enough to handle any message and hand it off would add a lot of design and processing overhead to each consumer. They would all have to know about each other so they could hand off work, and they would all need to know which of the others were busy so as not to give a consumer a message to process while it's already processing another. Making consumers work together would radically change the typical consumer design.

The Mediator pattern [GoF] offers some help. A Meditor coordinates a group of objects so that they don't need to know how to coordinate with each other. What we need for messaging is a mediator that coordinates the consumers for a channel. Then each consumer could focus on processing a particular kind of message, and the coordinator could make sure the right message gets to the right consumer.

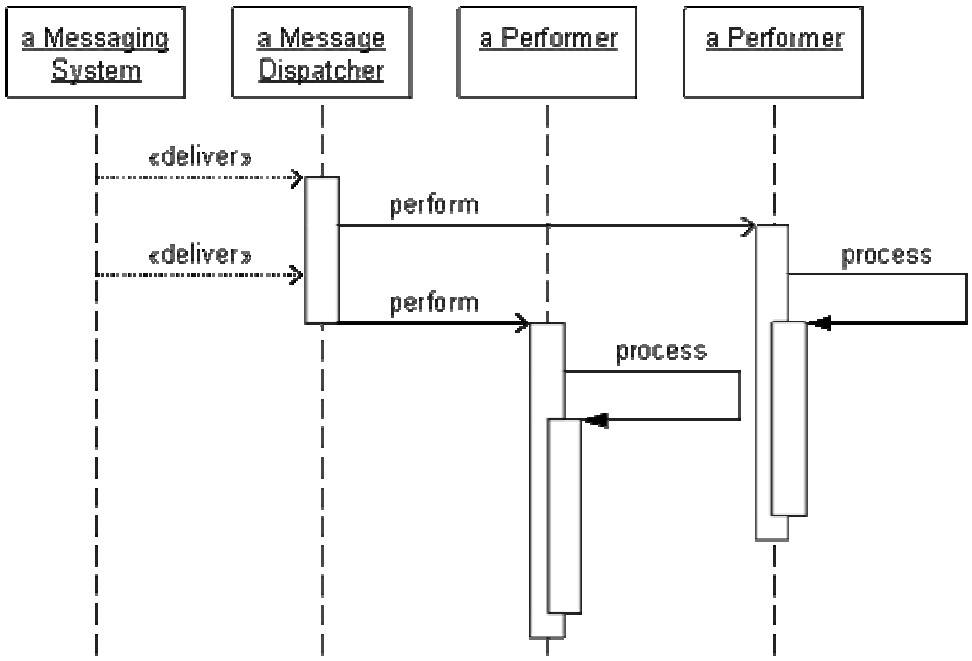
Create a *Message Dispatcher* on a channel that will consume messages from a channel and distribute them to performers.



A *Message Dispatcher* consists of two parts:

1. **Dispatcher –** The object that consumes messages from a channel and distributes each message to a performer.
2. **Performer –** The object that is given the message by the dispatcher and processes it.

When a *Message Dispatcher* receives a message, it obtains a performer and dispatches the message to the performer to process it. A performer can delegate to the rest of its application to help process its message. The performer could be newly created by the dispatcher, or could be selected from a pool of available performers. Each performer can run in its own thread to process messages concurrently. All performers may be appropriate for all messages, or the dispatcher may match a message to a specialized performer based on properties of the message.



Message Dispatcher Sequence

When the dispatcher receives a message, it delegates the message to an available performer to process it. If the performer processes the message using the dispatcher's thread, then the dispatcher blocks until the performer is finished processing the message. On the other hand, if the performer processes the message in its own thread, then once the dispatcher starts that thread, it can immediately start receiving other messages and delegating them to other performers, so that the messages are processed concurrently. This way, messages can be consumed as fast as the dispatcher can recieve and delegate them, regardless of how long each message takes to process.

A dispatcher acts as a one-to-many connection between a single channel and a group of performers. The performers do most of the work; the dispatcher just acts as a matchmaker, matching each message with an available performer, and does not block as long as the performers run in their own threads. The dispatcher receives the message and then sends it to a performer to process it. Because the dispatcher does relatively little work and does not block, it

potentially can dispatch messages as fast as the messaging system can feed them and thus avoids becoming a bottleneck.

This pattern is a simpler, messaging-specific version of the Reactor pattern [POSA2], where the message dispatcher is a Reactor and the message performers are Concrete Event Handlers. The *Message Channel* acts as the Synchronous Event Demultiplexer, making the messages available to the dispatcher one at a time. The messages themselves are like Handles, but much simpler. A true handle tends to be a reference to a resource's data, whereas a message usually contains the data directly. (However, the message does not have to store the data directly. If a message's data is stored externally and the message is a *Claim Check*, then the message contains a reference to the data, which is more like a Reactor handle.) Different types of handles select different types of concrete event handlers, whereas a *Message Channel* is a *Datatype Channel*, so all of the messages (handles) are of the same type, so there is typically only one type of concrete event handler.

On the other hand, whereas *Datatype Channel* designs a channel so that all messages are of the same type and all consumers process messages of that type, the Reactor pattern points out an opportunity to use *Message Dispatcher* to support multiple data types on the same channel and process them with type-specific performers. Each message must specify its type; the dispatcher detects the message's type and dispatches it to a type-specific performer for processing. In this way, a dispatcher with specialized performers can act as an alternative to *Datatype Channel*s and as a specialized implementation of *Selective Consumer*s.

One difference between *Message Dispatcher* and *Competing Consumers* is the ability to distribute across multiple applications. Whereas a set of competing consumers may be distributed amongst multiple applications, a set of performers typically all run in the same application as the dispatcher (even if they run in different threads). If a performer were running in a different application from its dispatcher, the dispatcher would have to communicate with the performer in a distributed, *Remote Procedure Invocation* manner, which is exactly what *Messaging* intends to avoid in the first place.

Since a dispatcher is a single consumer, it works fine with both *Point-to-Point Channel*s and *Publish-Subscribe Channel*s. With point-to-point messaging, a dispacther can be a suitable alternative to *Competing Consumers*; this alternative may be preferable if the messaging system handles multiple consumers badly or if handling of multiple consumers across different messaging system implementations is inconsistent.

A dispatcher makes the performers work much like *Event-Driven Consumer*s, even though the dispatcher itself could be event-driven or a *Polling Consumer*. As such, implementing a dispatcher as part of a *Transactional Client* can be difficult. If the client is transactional, ideally the dispatcher should allow the performer to process a message before completing the transaction. Then, only if the performer is successful should the dispatcher commit the transaction. If the performer fails to process the message, the dispatcher should rollback the transaction. Since each performer may need to rollback its individual message, the dispatcher needs a session for each performer and must use that performer's session to receive the performer's message and complete its transaction.

Since event-driven consumers often do not work well with transactional clients, the dispatcher should not be an event-driven consumer, but rather should be a polling consumer.

It can be helpful to implement performers as *Event-Driven Consumer*s. In JMS, this means implementing the performer as a `MessageListener`. A message listener has one method, `onMessage(Message)`; it accepts a message and performs whatever processing necessary. This forms a clean separation between the dispatcher and the performer. Likewise, in .NET, the performer should be a `ReceiveCompletedEventHandler` delegate, even though the dispatcher will not really issue `ReceiveCompleted` events. However, these event-driven API's may not be compatible with the API necessary to run a performer in its own thread.

To avoid the effort of implementing your own *Message Dispatcher*, consider instead using *Competing Consumers* on a *Datatype Channel*, or using *Selective Consumer*s. A *Message Dispatcher* can be a *Polling Consumer* or an *Event-Driven Consumer*. *Message Dispatcher*s do not make very good *Transactional Client*s.

Example: .NET

Usually, a *Message Dispatcher* dispatches messages to the performers (see the Java example). .NET provides another option: The dispatcher can use `Peek` to detect a message and get its message ID, then dispactch the message ID (not the full message) to the performer. The performer then uses `ReceiveById` to consume the particular message it has been assigned. In this way, each performer can take responsibility not just for processing the message, but for consuming it as well, which can help with concurrency issues, especially when the consumers are *Transactional Client*s.

Example: Simple Java Dispatcher

This is a simple example of how to implement a dispatcher and performer in Java. A more sophisticated dispather implementation might pool several performers, keep track of which ones are currently available to process messages, and make use of a thread pool. This simple example skips those details, but does run each performer in its own thread so that they can run concurrently.

The driver/manager that controls the dispatcher (not shown) will run `receiveSync()` repeatedly. Each time, the dispatcher will `receive()` the next message, instantiate a new `Performer` instance to process the message, then start the performer in its own thread.

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
```

```
import javax.naming.NamingException;

public class MessageDispatcher {

	MessageConsumer consumer;
	int nextID = 1;

	protected MessageDispatcher() {
		super();
	}

	public static MessageDispatcher newDispatcher(Connection connection, String
queueName)
		throws JMSException, NamingException {
		MessageDispatcher dispatcher = new MessageDispatcher();
		dispatcher.initialize(connection, queueName);
		return dispatcher;
	}

	protected void initialize(Connection connection, String queueName)
		throws JMSException, NamingException {
		Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
		Destination dispatcherQueue = JndiUtil.getDestination(queueName);
		consumer = session.createConsumer(dispatcherQueue);
	}

	public void receiveSync() throws JMSException {
		Message message = consumer.receive();
		Performer performer = new Performer(nextID++, message);
		new Thread(performer).start();
	}
}
```

The `Performer` must implement `Runnable` so that it can run in its own thread. The runnable's `run()` method simply calls `processMessage()`. When this is complete, the performer becomes eligable for garbage collection.

```
import javax.jms.JMSException;
import javax.jms.Message;

public class Performer implements Runnable {

	private int performerID;
	private Message message;
```

```
        public Performer(int id, Message message) {
                performerID = id;
                this.message = message;
        }

        public void run() {
                try {
                        processMessage();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        private void processMessage() throws JMSException, InterruptedException {
                int id = message.getIntProperty("cust_id");

                System.out.println(System.currentTimeMillis() + ": Performer #" +
performerID + " starting; message ID " + id);
                Thread.sleep(500);
                System.out.println(System.currentTimeMillis() + ": Performer #" +
performerID + " processing.");
                Thread.sleep(500);
                System.out.println(System.currentTimeMillis() + ": Performer #" +
performerID + " finished.");
        }
}
```

So implementing a simple dispatcher and performer is easy. The main trick is to make the performer a `Runnable` and run it in its own thread.

Related patterns: *Competing Consumers*, *Datatype Channel*, *Remote Procedure Invocation*, *Event-Driven Consumer*, *Message Channel*, *Selective Consumer*, *Messaging*, *Point-to-Point Channel*, *Polling Consumer*, *Publish-Subscribe Channel*, *Claim Check*, *Transactional Client*

Selective Consumer

An application is using *Messaging*. It consumes *Message*s from a *Message Channel*, but it does not necessarily want to consume all of the messages on that channel, just some of them.

How can a message consumer select which messages it wishes to receive?

By default, if a *Message Channel* has only one consumer, all *Message*s on that channel will be delivered to that consumer. Likewise, if there are multiple *Competing Consumers* on the channel,

any message can potentially go to any consumer, and every message will go to some consumer. A consumer normally does not get to choose which messages it consumes; it always gets whatever message is next.

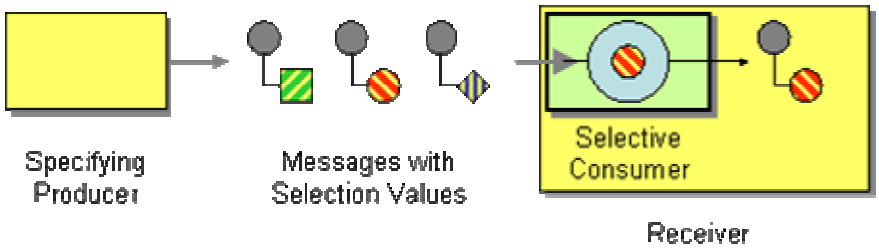
This behavior is fine as long as the consumer wants to receive any and all messages on the channel, which is normally the case. This is a problem, however, when a consumer only wants to consume certain messages, because a consumer normally has no control over which messages on a channel it receives. Why would a consumer want to receive certain messages only? Consider an application processing loan request messages; it may want to process loans for up to $100,000 differently from those over $100,000. One approach would be for the application to have two different kinds of consumers, one for small loans and another for big loans. Yet, since any consumer can receive any message, how can the application make sure that the right messages go to the right consumer?

The simplest approach might be for each to consume whatever messages it gets. If it gets the wrong kind of message, it could somehow hand that message to the appropriate kind of consumer. That's going to be difficult, though; consumer instances usually don't know about each other, and finding one that isn't already busy processing another message can be difficult. Perhaps when the consumer realizes it doesn't want the message, it could put the message back on the channel. But then it's likely to just consume the message yet again. Perhaps every consumer could get a copy of every message, and just discard the ones it doesn't want. This will work, but will cause a lot of message duplication and a lot of wasted processing on messages that are ultimately discarded.

Perhaps the messaging system could define separate channels for each type of message. Then the sender could make sure to send each message on the proper channel, and the receivers could be sure that the messages they receive off of a particular channel are the kind desired. However, this solution is not very dynamic. The receivers may change their selection criteria while the system is running, which would require defining new channels and redistributing the messages already on the channels. It also means that the senders must know what the receiver's selection criteria are and when those criteria change. The criteria need to be a property of the receivers, not of the channels, and the messages on the channel need to specify what criteria they meet.

What is needed is a way for messages fitting a variety of criteria to all be sent on the same channel, for the consumers to be able to specify what criteria they're interested in, and for each consumer to only receive the messages that meet its criteria.

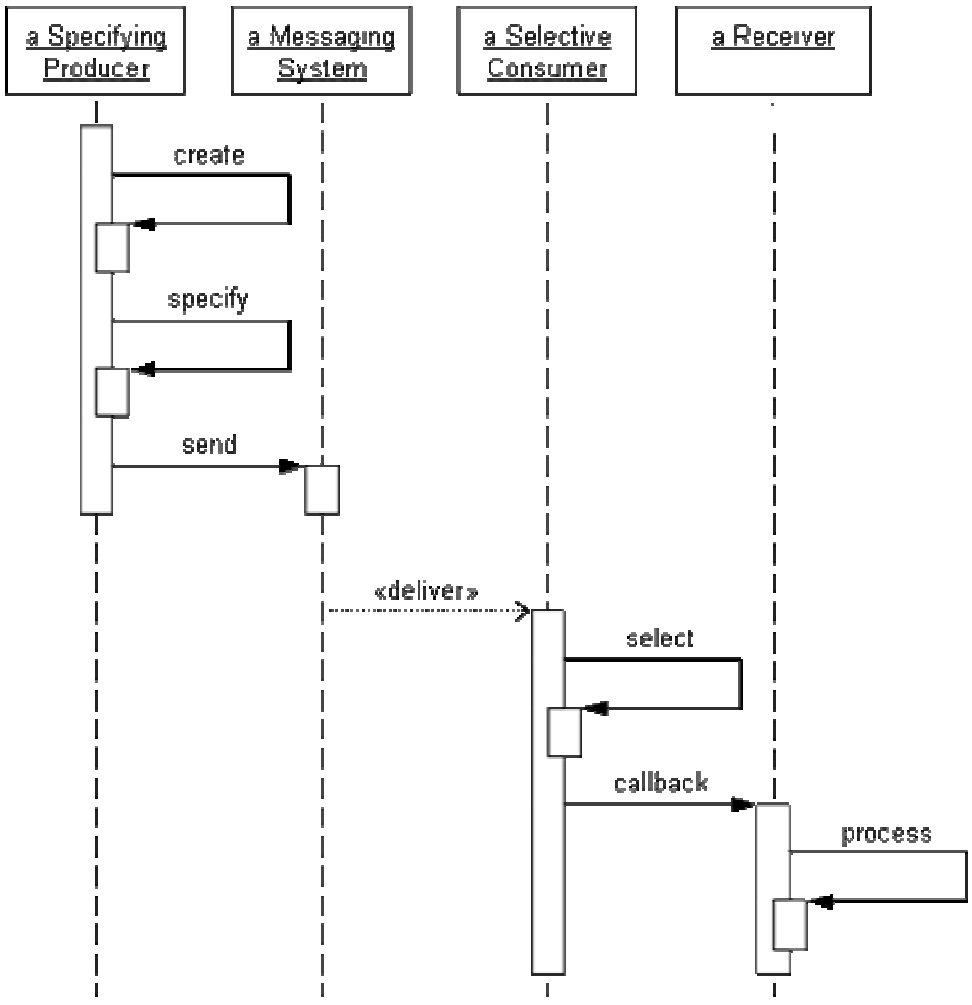
Make the consumer a *Selective Consumer*, one that filteres the messages delivered by its channel so that it only receives the ones that match its criteria.



There are three parts to this filtering process:

1. **Specifying Producer** — Specifies the message's selection value before sending it.
2. **Selection Value** — One or more values specified in the message that allow a consumer to decide whether to select the message.
3. **Selective Consumer** — Only receives messages that meet its selection criteria.

The message sender specifies each message's selection value before sending it. When a message arrives, a *Selective Consumer* tests the message's selection value to see if the value meets the consumer's selection criteria. If so, the consumer receives the message and passes it to the application for processing.



Selective Consumer Sequence

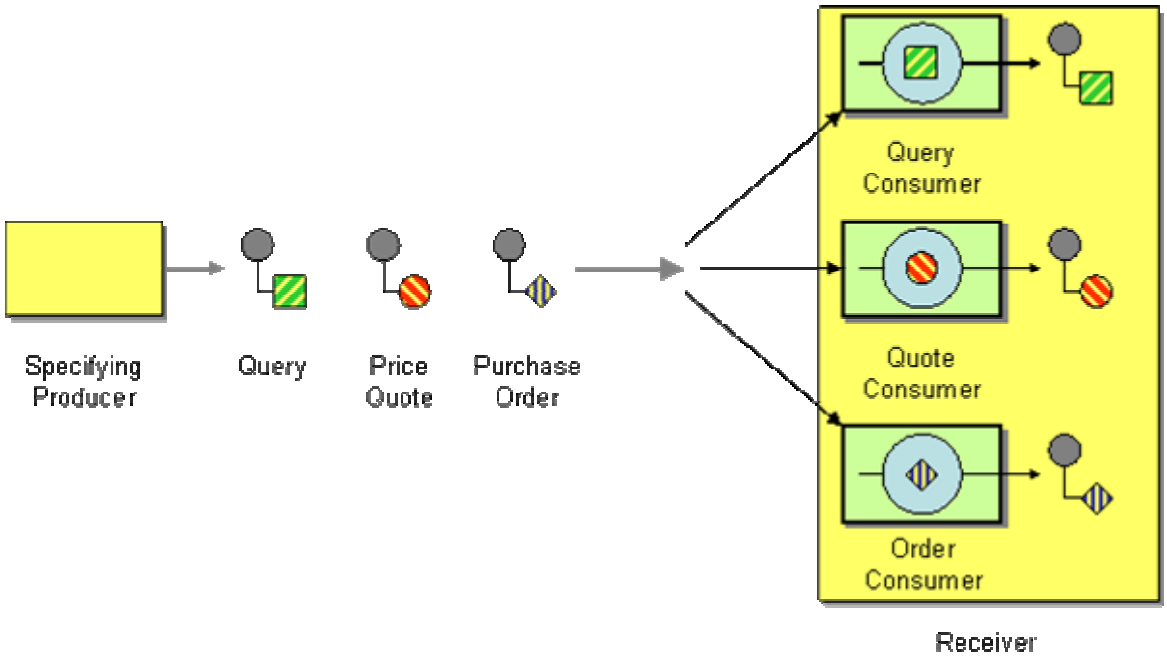
When the sender creates the message, it also sets the message's selection value; then it sends the message. When the messaging system delivers the message, the *Selective Consumer* tests the message's selection value to determine whether to select the message. If the message passes, the consumer receives the message and passes the message to the application using a callback.

*Selective Consumer*s are often used in groups—one consumer filters for one set of criteria, while another filters for a different set, and so on. For the loan processing example, one consumer would select "amount <= $100,000" while another would select "amount > $100,000". Then each consumer would only get the kinds of loans it is interested in.

When multiple *Selective Consumer*s are used with a *Point-to-Point Channel*, they effectively become *Competing Consumers* that are also selective. If two consumer's criteria overlap, and a message's selection value meets both of their criteria, either consumer can consume the message. Consumers should be designed to ensure that at least one of them is eligable to consume every valid selection value. Otherwise, a message with an unmatched selection value will never be consumed and will clutter the channel forever (or at least until *Message Expiration* occurrs).

When multiple *Selective Consumer*s are used with a *Publish-Subscribe Channel*, each message will be delivered to each subscriber, but a subscriber will simply ignore its copy of a message that does not fit its criteria. Once a consumer decides to ignore a message, the messaging system can discard the message since it has been successfully delivered and will never be consumed. A messaging system can optimize this process by not even delivering a message it knows the consumer will ignore, thereby decreasing the number of copies of a message that must be produced and transmitted. This behavior of discarding ignored messages is independent of whatever *Guaranteed Delivery*, *Durable Subscriber*, and/or *Message Expiration* settings are used.

*Selective Consumer*s make a single channel act like multiple *Datatype Channel*s. Different types of messages can have different selection values, so that a consumer that is specialized for a particular type will only receive messages of that type. This can facilitate sending a large number of types using a small number of channels. This approach can also conserve channels in an enterprise which requires more channels than a messaging system can support.



Competing, Selective Consumers

Using *Selective Consumer*s to emulate *Datatype Channel*s is not a good approach when trying to hide messages of a certain type from certain consumer applications. Whereas a messaging system can ensure only authorized applications successfully receive messages from a channel, they usually do not authorize a consumer's selection criteria, so a malicious consumer authorized to access the channel can gain access to unauthorized messages by changing its criteria. Separate datatype channels are needed to securely lock out applications.

An alternative to using *Selective Consumer*s is to use a *Message Dispatcher*. The selection criteria are built into the dispatcher, which then uses them to determine the performer for each message. If a message does not meet any of the performer's criteria, rather than leave it cluttering the channel or discarding it, the dispatcher can reroute the unmatched message to the *Invalid Message Channel*. As with the tradeoff between *Message Dispatcher* and *Competing Consumers*, the question is really whether you wish to let the messaging system do the dispatching or whether you want to implement it yourself. If a messaging system does not support *Selective Consumer* as a feature, you have no choice but to implement it youself using a *Message Dispatcher*.

As mentioned earlier, if none of the *Selective Consumer*s on a channel match the message's selector value, the message will be ignored as if the channel has no receivers. A similar problem in procedural programming is a case statement where none of the cases matches the value being tested. Thus a case statement can have a default case which matches values that aren't matched by any other case. Applying this approach to messaging, it may seem tempting to create some sort of default consumer for messages that otherwise have no matching consumer. Yet such a default consumer will not work as desired because it would require an expression that matches all selector values, so it would compete with all other consumers. Instead, to implement a default consumer, use a *Message Dispatcher* that implements a case statement with a default option for unhandled cases that uses the default consumer.

Another alternative to *Selective Consumer*s is *Message Filter*. They accomplish much the same goal, but in different ways. With a *Selective Consumer*, all of the messages are delivered to the receivers, but each receiver ignores unwanted messages. A *Message Filter* sits between a channel from the sender and a channel to the receiver and only transfers desired messages from the sender's channel to the receiver's. Thus unwanted messages are never even delivered to the receiver's channel, so the receiver has nothing to ignore. *Message Filter* is useful for getting rid of messages that no receiver wants. *Selective Consumer*s are useful when one receiver wants to ignore certain messages but other receivers want to receive those messages.

Another alternative to consider is *Content-Based Router*. This type of router, like a filter, makes sure that a channel only gets the messages that the receivers want, which can increase security and increase the performance of the consumers. *Selective Consumer* is more flexible, however, because each filtering option simply requires a new consumer (which are easy to create while the system is running), whereas each new option with a *Content-Based Router* requires a new output channel (which is not so easy to create and use while the system is running) as well as a new consumer for the new channel. Consider a requirements change where you want to process medium-size loans ($50K-$150K) differently from small and large loans. With *Content-Based Router*, you need to create a new channel for medium loans, as well as a consumer on that new channel, and adjust the way the router separates loans. You also need to worry about what happens when the change takes effect, because some messages that have already been routed onto the original channels may not have been consumed yet and may now be on the wrong channel. With *Selective Consumer*, you just replace the two types of consumers (less-than-$100K and greater-than-$100K) with three types (less-than-$50K, $50K-$150K, and greater-than-$150K). *Content-Based Router* is a much more static approach, whereas *Selective Consumer* can be much more dynamic.

Ideally, a message's selection value should be specified in its header, not its body, so that a *Selective Consumer* can process the value without having to parse (and know how to parse) the message's body.

*Selective Consumer*s make a single channel act like multiple *Datatype Channel*s. They allow messages to be available for other receivers whereas *Message Filter* prevents unwanted messages from being delivered to any receiver, and can be used more dynamically than a *Content-Based Router*. A *Selective Consumer* can be implemented as a *Polling Consumer* or *Event-Driven Consumer*, and can be part of a *Transactional Client*. To implement the filtering behavior yourself, use a *Message Dispatcher*.

Example: Separating Types

For example, a stock trading system with a limited number of channels might need to use one channel for both quotes and trades. The receiver for performing a quote is very different from that for trading, so the right receiver needs to be sure to consume the right message. So the sender would set the selector value on a quote message to QUOTE, and the *Selective Consumer* for quotes would only consume messages with that selector value. Trade messages would have their

own TRADE selector value that their senders and receivers would use. In this way, two message types can successfully share a single channel.

Example: JMS Message Selector

In JMS, a `MessageConsumer` (`QueueReceiver` or `TopicSubscriber`) can be created with a message selector string that filters messages based on their property values. [JMS11, p.41], [Hapner, p.23] First, a sender would set the value of a property in the message that the receiver could filter by:

```
Session session = // get the session
TextMessage message = session.createTextMessage();
message.setText("<quote>SUNW</quote>");
message.setStringProperty("req_type", "quote");
Destination destination = //get the destination
MessageProducer producer = session.createProducer(destination);
producer.send(message);
```

Second, a receiver set its message selector to filter for that value:

```
Session session = // get the session
Destination destination = //get the destination
String selector = "req_type = 'quote'";
MessageConsumer consumer =
        session.createConsumer(destination, selector);
```

This receiver will ignore all messages whose request type property is not set to "quote" as if those messages were never delivered to the destination at all.

Example: .NET Peek, ReceiveById, and ReceiveByCorrelationId

In .NET, `MessageQueue.Receive` does not support JMS-style message selectors per se. Rather, what a receiver can do is use `MessageQueue.Peek` to look at a message. If it meets the desired criteria, then it can use `MessageQueue.Receive` to read it from the queue. This may not work very reliably, though, since the message returned by the `Receive` call may not necessarily be the same message that was `Peek`ed. Thus use `ReceiveById`, whereby the consumer specifies the `Id` property value of the message it wishes to receive, instead of `Receive` to ensure getting the same message that was `Peek`ed.

Another option in .NET is the `ReceiveByCorrelationId` method, with which the consumer specifies the `CorrelationId` property value of the message it wants to receive. A sender of a

particular request message can use `ReceiveByCorrelationId` to receive the reply message specific to that request (see *Request-Reply* and *Correlation Identifier*).

Related patterns: *Competing Consumers*, *Content-Based Router*, *Correlation Identifier*, *Datatype Channel*, *Durable Subscriber*, *Event-Driven Consumer*, *Message Filter*, *Guaranteed Delivery*, *Invalid Message Channel*, *Message*, *Message Channel*, *Message Dispatcher*, *Message Expiration*, *Messaging*, *Point-to-Point Channel*, *Polling Consumer*, *Publish-Subscribe Channel*, *Request-Reply*, *Transactional Client*

Durable Subscriber

An application is receiving messages on a *Publish-Subscribe Channel*.

How can a subscriber avoid missing messages while it's not listening for them?

Why is this even an issue? Once a message is added to a channel, it stays there until it is either consumed, it expires (see *Message Expiration*), or the system crashes (unless you're using *Guaranteed Delivery*). This is true for a message on a *Point-to-Point Channel*, but a *Publish-Subscribe Channel* works somewhat differently.

When a message is published on a *Publish-Subscribe Channel*, the messaging system must deliver the message to each subscriber. How it does this is implementation specific: It can keep the message until the list of subscribers that have not received it is empty, or it might duplicate and deliver the message to each subscriber. Whatever the case, which subscribers receive the message is completely dependent upon who is subscribed to the channel when the message is published. If a receiver is not subscribed when the message is published, even if the receiver subscribes an instant later, it will not receive that message. (There is also a timing issue of what happens when a subscriber subscribes and a message is published on the same channel at "about" the same time. Does the subscriber receive the message? How this issue is resolved depends on the messaging system's implementation. To be safe, subscribers should be sure to subscribe before messages of interest are published.)

As a practical matter, a subscriber unsubscribes from a channel by closing its connection to the channel. Thus no explicit unsubscribe action is necessary; the subscriber just closes its connection.

Often, an application prefers to ignore messages published after it disconnects, because being disconnected means that the application is uninterested in whatever may be published. For example, a B2B/C application selling bricks may subscribe to a channel where buyers can request bricks. If the application stops selling bricks, or is temporarily out of bricks, it may decide to disconnect from the channel to avoid receiving requests it cannot fulfill anyway.

Yet this behavior can be disadvantageous, because the "you snooze, you loose" approach can cause an application to miss messages it needs. If an application crashes, or must be stopped for maintenance, it may want to know what messages it missed while it wasn't running. The whole

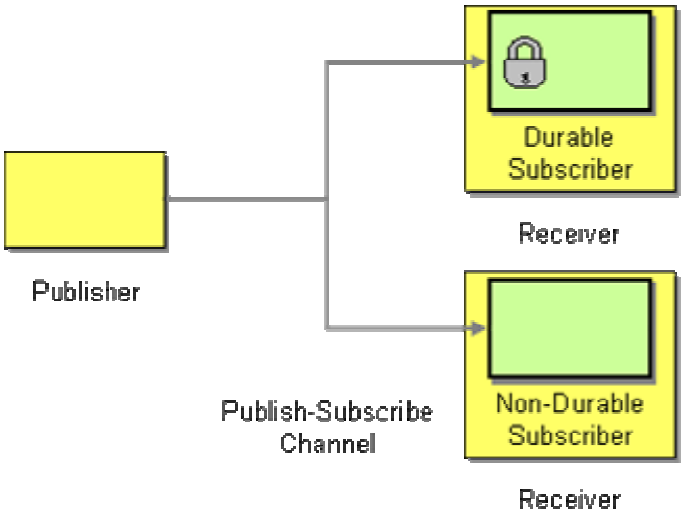
idea of messaging is to make communication reliable even if the sender and receiver applications and network aren't all working at the same time.

So, sometimes applications disconnect because they don't want messages from that channel anymore. But sometimes applications have to disconnect for a short time, but when they reconnect, they want to have access to all of the messages that were published during the connection lapse. A subscriber is normally either connected (*subscribed*) or disconnected (*unsubscribed*), but a third possibe state is *inactive*, the state of a subscriber that is disconnected but still subscribed because it wants to receive messages published while it is disconnected.

If a subscriber was connected to a *Publish-Subscribe Channel*, but is disconnected when a message is published, how does the messaging system know whether to save the message for the subscriber so that it can deliver the message when the subscriber reconnects? That is, how does the messaging system know whether a disconnected subscriber is inactive or unsubscribed? There needs to be two kinds of subscriptions, those that end when the subscriber disconnects and those that survive even when the application disconnects and are only broken when the application explicitly unsubscribes.

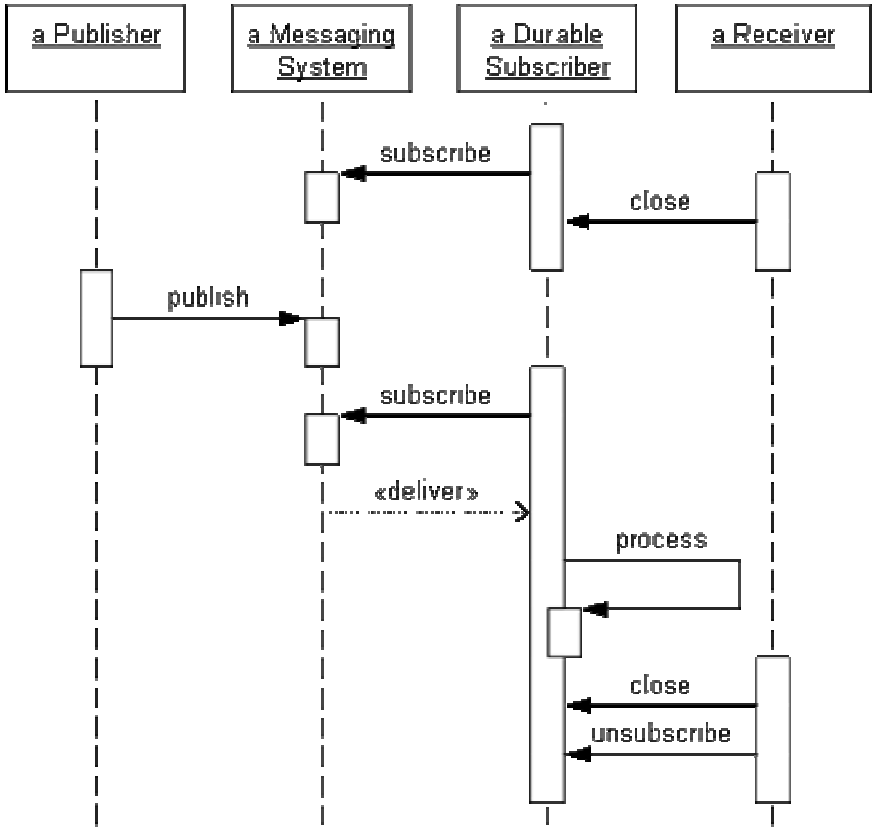
By default, a subscription only lasts as long as its connection. So what is needed it another type of subscription that survives disconnects by becoming inactive.

Use a *Durable Subscriber* to make the messaging system save messages published while the subscriber is disconnected.



A durable subscription saves messages for an inactive subscriber and deliveres these saved messages when the subscriber reconnects. In this way, a subscriber will not loose any messages even though it disconnected. A durable subscription has no effect on the behavior of the subscriber or the messaging system while the subscriber is *active* (e.g., connected). A connected subscriber acts the same whether its subscription is durable or non-durable. The difference is in how the messaging system behaves when the subscriber is disconnected.

A *Durable Subscriber* is simply a subscriber on a *Publish-Subscribe Channel*. However, when the subscriber disconnects from the messaging system, it becomes inactive and the messaging system will save any messages published on its channel until it becomes active again. Meanwhile, other subscribers to the same channel may not be durable; they're non-durable subscribers.



Durable Subscription Sequence

To be a subscriber, the *Durable Subscriber* must establish its subscription to the channel. Once it has, when it closes its connection, it becomes inactive. While the subscriber is inactive, the publisher publishes a message. If the subscriber were non-durable, it would miss this message; but because it is durable, the messaging system saves this message for this subscriber. When the subscriber resubscribes, becoming active once more, the messaging system delivers the queued message (and any others saved for this subscriber). The subscriber receives the message and process it (perhaps delegating the message to the application). Once the subscriber is through processing messages, if it does not wish to receive any more messages, it closes its connection, becoming inactive again. Since it does not want the messaging system to save messages for it anymore, it also unsubscribes.

An interesting consequence is: What would happen if a durable subscriber never unsubscribed? The inactive durable subscription would continue to retain messages—that is, the messaging system will save all of the published messages until the subscriber reconnects. But if the subscriber does not reconnect for a long time, the number of saved messages can become excessive. *Message Expiration* can help alleviate this problem. The messaging system may also wish to limit the number of messages that can be saved for an inactive subscription.

Example: Stock Trading

A stock trading system might use a *Publish-Subscribe Channel* to broadcast changes in stocks prices; each time a stocks' price changes, a message is published. One subscriber might be a GUI that displays the current prices for certain stocks. Another subscriber might be a database that stores the day's trading range for certain stocks.

Both applications should be subscribers to the price-change channel so that they're notified when a stock's price changes. The GUI's subscription can be non-durable because it is displaying the current price. If the GUI crashes a looses its connection to the channel, there is no point in saving price changes the GUI cannot display. On the other hand, the price range database should use a *Durable Subscriber*. While it is running, it can display the range thus far. If it looses its connection, when it reconnects, it can process the price changes that occurred and update the range as necessary.

Example: JMS Durable Subscription

JMS supports durable subscriptions for `TopicSubscribers`. [JMS11, pp.80-81], [Hapner, pp.61-63]

One challenge with durable subscriptions is differentiating between an old subscriber that is reconnecting vs. a completely new subscriber. In JMS, a durable subscription is identified by three criteria:

1. the topic being subscribed to
2. the connection's client ID
3. and the subscriber's subscription name

The connection's client ID is a property of its connection factory, which is set when the connection factory is created using the messaging system's administration tool. The subscription name has to be unique for each subscriber (for a particular topic and client ID).

A *Durable Subscriber* is created using the `Session.createDurableSubscriber` method:

```
ConnectionFactory factory = // obtain the factory
// the factory has the client ID
Connection connection = factory.createConnection();
// the connection has the same client ID as the factory
Topic topic = // obtain the topic
String clientID = connection.getClientID(); // just in case you're curious
String subscriptionName = "subscriber1"; // some UID for the subscription

Session session =
   connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
TopicSubscriber subscriber =
    session.createDurableSubscriber(topic, subscriptionName);
```

This subscriber is now active. It will receive messages as they are published to the topic (just like a non-durable subscriber). To make it inactive, close it, like this:

```
subscriber.close();
```

The subscriber is now disconnected and therefore inactive. Any messages published to its topic will be saved for this subscriber and delivered when it reconnects.

To make the subscription active again, you must create a new durable subscriber with the same topic, client ID, and subscription name. The code is the same as before, except that the connection factory, topic, and subscription name must be the same as before.

Because the code is the same to establish a durable subscription and to reconnect to it, only the messaging system knows whether this durable subscription had already been established or is a new one. One interesting consequence is that the application re-connecting to a subscription may not be the same application that disconnected earlier. As long as the new application uses the same topic, the same connection factory (and so the same client ID), and the same subscription name as the old application, the messaging system cannot distinguish between the two applications and will proceed to deliver all messages to the new application that weren't delivered to the old application before it disconnected.

Once an application has a durable subscription on a topic, it will have the opportunity to receive all messages published to that topic, even if the subscriber closes its connection (or if it crashes and the messaging system closes the subscriber's connection for it). To stop the messaging system from queuing messages for this inactive subscriber, the application must explicitly unsubscribe its durable subscription.

```
subscriber.close();
// subscriber is now inactive, messages will be saved
session.unsubscribe(subscriptionName);
// subscription is removed
```

Once the subscriber is unsubscribed, the subscription is removed from the topic, and messages will no longer be delivered to this subscriber.

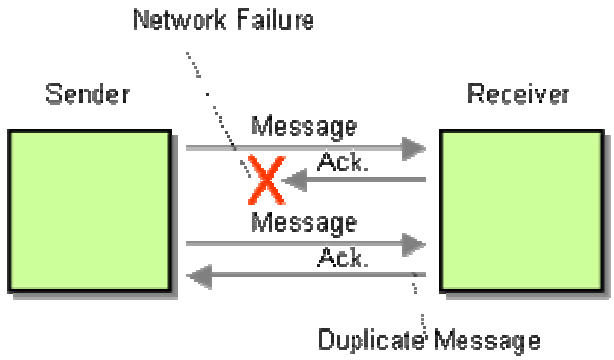
Related patterns: *Guaranteed Delivery*, *Message Expiration*, *Point-to-Point Channel*, *Publish-Subscribe Channel*

Idempotent Receiver

Even when a sender application only sends a message once, the receiver application may receive the message more than once.

How can a message receiver deal with duplicate messages?

The Channel Patterns discuss how to make messaging channels reliable by using *Guaranteed Delivery*. However, some reliable messaging implementations can produce duplicate messages. In other scenarios, *Guaranteed Delivery* may not be available because the communication relies on inherently unreliable protocols. This is the case in many B2B (business-to-business) integration scenarios where messages have to be sent over the Internet using the HTTP protocol. In these cases, message delivery can generally only be guaranteed by resending the message until an acknowledgment is returned from the recipient. However, if the acknowledgment is lost due to an unreliable connection, the sender may resend a message that the receiver had already received (see diagram).



Message duplication because of problem sending acknowledgement

Many messaging systems incorporate built-in mechanisms to eliminate duplicate messages so that the application does not have to worry about duplicates. Eliminating duplicates inside the messaging infrastructure causes additional overhead. If the receiver is inherently resilient against duplicate messages, for example, a stateless receiver that processes query-style *Command Message*s, messaging throughput can be increased if duplicates are allowed. For this reason, some messaging systems provide only "at least once" delivery and let the application deal with duplicate messages. Others allow the application to specify whether it deals with duplicates or not (for example, the JMS spec defines a DUPS_OK_ACKNOWLEDGE mode).

Another scenario that can produce duplicate messages is a failed distributed transaction. Many packaged applications that are connected to the messaging infrastructure through commercial adapters cannot properly participate in a distributed two-phase commit. When a message is sent to multiple applications and the message causes one or more of these applications to fail it may be difficult to recover from this inconsistent state. If receivers are designed to ignore duplicate messages, the sender can simply re-send the message to all recipients. Those recipients that had already received and processed the original message will simply ignore the resend. Those

applications that were not able to properly consume the original message will apply the message that was resent.

Therefore:

Design a receiver to be an *Idempotent Receiver*--one that can safely receive the same message multiple times.

The term *idempotent* is used in mathematics to describe a function that produces the same result if it is applied to itself, i.e. f(x) = f(f(x)). In *Messaging* this concepts translates into the a message that has the same effect whether it is received once or multiple times. This means that a message can safely be resent without causing any problems even if the receiver receives duplicates of the same message.

Idempotency can be achieved through two primary means:

1. Explicit "de-duping", i.e. the removal of duplicate messages.
2. Defining the message semantics to support idempotency.

The recipient can explicitly de-dup messages (let's assume this is a proper English word) by keeping track of messages that it already received. A unique message identifier simplifies this task and helps detect those cases where two legitimate messages with the same message content arrive. By using a separate field, the message identifier, we do not tie the semantics of a duplicate message to the message content. We then assign a unique message identifier to each message. Many messaging systems, such as JMS-compliant messaging tools, automatically assign unique message identifiers to each message without the applications having to worry about them.

In order to detect and eliminate duplicate messages based on the message identifier, the message recipient has to keep a buffer of already received message identifiers. One of the key design decisions is how long to keep this history of messages and whether to persist the history to permanent storage such as disk. This decision depends primarily on the contract between the sender and the receiver. In the simplest case, the sender sends one message at a time, awaiting the receiver's acknowledgment after every message. In this scenario, it is sufficient for the receiver to compare the message identifier of any incoming message to the identifier of the previous message. It will then ignore the new message if the identifiers are identical. Effectively, the receiver keeps a history of a single message. In practice, this style of communication can be very inefficient, especially if the latency (the time for the message to travel from the sender to the receiver) is significant relative to the desired message throughput. In these situations, the sender may want to send a whole set of messages without awaiting acknowledgment for each one. This implies, though, that the receiver has to keep a longer history of identifiers for already received messages. The size of the receiver's "memory" depends on the number of message the sender can send without having gotten an acknowledgment from the receiver. This problem resembles the considerations presented in the *Resequencer*.

Eliminating duplicate messages is another example where we can learn quite a bit by having a closer look at the low-level TCP/IP protocol. When IP network packets are routed across the network, duplicate packets can be generated. The TCP/IP protocol ensures elimination of duplicate packets by attaching a unique identifier to each packet. Sender and receiver negotiate a "window size" that the recipient allocates in order to detect duplicates. For a thorough discussion of how TCP/IP implements this mechanism, see [Stevens].

In some cases, it may be tempting to use a business key as the message identifier and let the persistence layer handle the de-duping. For example, let's assume that an application persists incoming orders into a database. If each order contains a unique order number, and we configure the database to use a unique key on the order number field, the insert operation into the database would fail if a duplicate order message is received. This solution appears elegant because we delegated the checking of duplicates to the database systems which is very efficient at detecting duplicate keys. But we have to be cautious because we associated dual semantics to a single field. Specifically, we tied infrastructure-related semantics (a duplicate message) to a business field (order number). Imagine that the business requirements change so that customers can amend existing orders by sending another message with the same order number (this is quite common). We would now have to make changes to our message structure since we tied the unique message identifier to a business field. Therefore, it is best to avoid overloading a single field with dual semantics.

Using a database to force de-duping is sometimes used with database adapters that are provided by the messaging infrastructure vendors. In many cases, these adapters are not capable of eliminating duplicates so that this function has to be delegated to the database.

An alternative approach to achieve idempotency is to define the semantics of a message such that resending the message does not impact the system. For example, rather than defining a message as "Add $10 to account 12345", we could change the message to "Set the balance of account 12345 to $110". Both messages achieve the same result if the current account balance is $100. The second message is idempotent because receiving it twice will not have any effect. Admittedly, this example ignores concurrency situations, for example the case where another message "Set the balance of account 12345 to $150" arrives between the original and the duplicate message.

Example: Microsoft IDL (MIDL)

The Microsoft Interface Definition Language (MIDL) supports the concept of idempotency as part of the remote call semantics. A remote procedure can be declared as idempotent by using the `[idempotent]` attribute. The MIDL specification states that the "`[idempotent]` attribute specifies that an operation does not modify state information and returns the same results each time it is performed. Performing the routine more than once has the same effect as performing it once".

```
interface IFoo;
[
   uuid(5767B67C-3F02-40ba-8B85-D8516F20A83B),
```

```
    pointer_default(unique)
]

interface IFoo
{
    [idempotent]
    bool GetCustomerName
    (
        [in] int CustomerID,
        [out] char *Name
    );
}
```

Related patterns: *Command Message*, *Guaranteed Delivery*, *Messaging*, *Resequencer*

Service Activator

An application has a service that it would like to make available to other applications.

How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

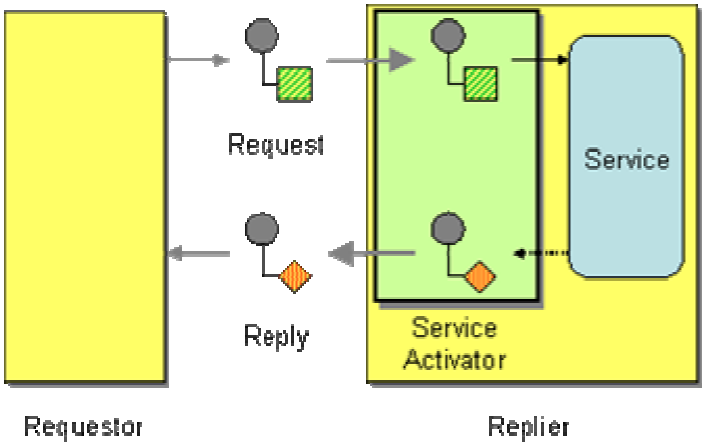
An application may not want to choose whether a service (an operation in a Service Layer [EAA]) can be invoked synchronously or asynchronously, it may want to support both approaches for the same service. Yet technologies can seem to force the choice. For example, an application implemented using EJB (Enterprise JavaBeans) may need to use a session bean to support synchronous clients, but a message-driven bean to support messaging clients. (Thanks to Mark Weitzel for this example.)

Developers designing an application to work with other applications, such as a B2B (business-to-business) application, may not know what other applications they're communicating with and how the various communication will work. There are too many different messaging technologies and data formats to try to support every one just in case it's needed. (Thanks to Luke Hohmann for this example.)

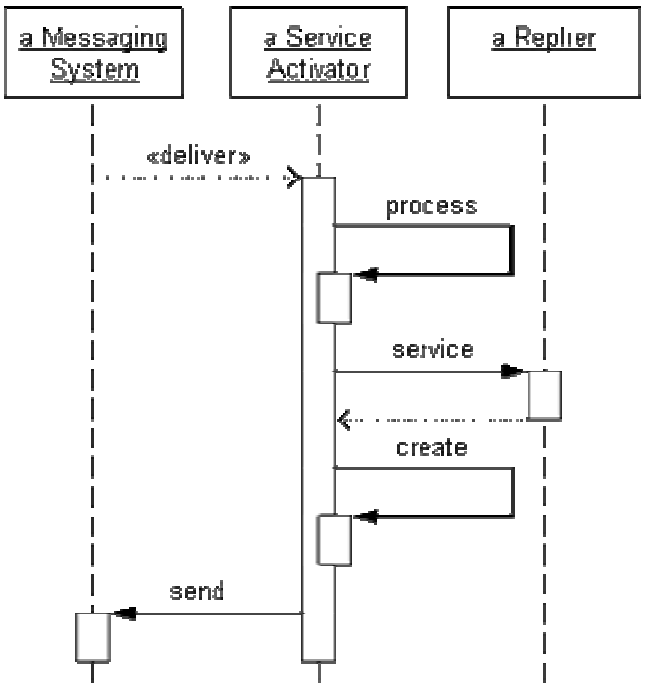
Receiving and processing a message involves a number of steps; separating these steps can be difficult and unnecessarily complex. Yet *Message Endpoint* code that mixes together these tasks—receiving the message, extracting its contents, and acting on those contents to perform work—can be difficult to reuse.

When designing clients for multiple styles of communication, it may well seem necessary to reimplement the service for each style. This makes supporting each new style cumbersome and creates the risk that each style may not produce quite the same behavior. What is needed is a way for a single service to support multiple styles of communication.

Design a *Service Activator* that connects the messages on the channel to the service being accessed.



A *Service Activator* can be one-way (request only) or two-way (*Request-Reply*). The service can be as simple as a method call—synchronous and non-remote—perhaps part of a Service Layer [EAA]. The activator can be hard-coded to always invoke the same service, or can use reflection to invoke the service indicated by the message. The activator handles all of the messaging details and invokes the service like any other client, such that the service doesn't even know it's being invoked through messaging.



Service Activator Sequence for Request-Reply

The *Service Activator* handles receiving the request message (either as a *Polling Consumer* or as an *Event-Driven Consumer*). It knows the message's format and processes the message to extract the information necessary to know what service to invoke and what parameter values to pass in. The activator then invokes the service just like any other client of the service, and blocks while the

service executes. When the service completes and returns a value, the activator can optionally create a reply message containing the value and return it to the requestor. (The reply makes the service invocation an example of *Request-Reply* messaging.)

A *Service Activator* enables a service to be written as though it's always going to be invoked synchronously. The activator receives the asynchronous message, determines what service to invoke and what data to pass it, and then invokes the service synchronously. The service is designed to work without messaging, yet the activator enables it to easily be invoked via messaging.

If the *Service Activator* cannot process the message successfully, the message is invalid and should be moved to an *Invalid Message Channel*. If the message can be processed and the service is invoked successfully, then any errors that occur as part of executing the service are semantic errors in the application and should be handled by the application.

Developers still may not be able to predict every way partners might wish to access their services, but they do at least know what services their application will provide and can implement those. Then implementing new activators for different technologies and formats as needed is relatively easy.

This *Service Activator* pattern is also documented in [CoreJ2EE], which is where the pattern was originally named. That version of the pattern is somewhat different from this one—it assumes the activator is an *Event-Driven Consumer*, and assumes that the service already exists so that the activator can be added to the service—but both versions propose the same solution to the same problem in a very similar fashion. *Service Activator* is related to the Half-Sync/Half-Async pattern [POSA2], which separates service processing into synchronous and asynchronous layers.

A *Service Activator* usually receives *Command Message*s, which describe what service to invoke. A *Service Activator* serves as a *Messaging Gateway*, separating the messaging details from the service. The activator can be a *Polling Consumer* or an *Event-Driven Consumer*. If the service is transactional, the activator should be a *Transactional Client* so that the message consumption can participate in the same transaction as the service invocation. Multiple activators can be *Competing Consumers* or coordinated by a *Message Dispatcher*. If a *Service Activator* cannot process a message successfully, it should send the message to an *Invalid Message Channel*.

Example: J2EE Enterprise JavaBeans

Consider, for example, Enterprise JavaBeans (EJBs) [EJB20] in J2EE: Encapsulate the service as a session bean, and then implement message-driven beans for various messaging scenarios: One for a JMS destination using messages of one format; another for a different destination using another format; another for a web service/SOAP message; and so on. Each message-driven bean that processes the message by invoking the service is a *Service Activator*. Clients that wish to invoke the service synchronously can access the session bean directly.

Related patterns: *Command Message*, *Competing Consumers*, *Event-Driven Consumer*, *Invalid Message Channel*, *Message Dispatcher*, *Message Endpoint*, *Messaging Gateway*, *Polling Consumer*, *Request-Reply*, *Transactional Client*

11. System Management

Introduction

While developing a messaging solution is no easy task, operating such a solution in production is equally challenging: A message-based integration solution may produce, route and transform thousands or even millions of messages in a day. We have to deal with exceptions, performance bottlenecks and changes in the participating systems. To make things ever more challenging, components are distributed across many platforms and machines that can reside at multiple locations.

Besides the inherent complexities and scale of integrating distributed packaged and custom applications, the architectural benefits of loose coupling actually make testing and debugging a system harder. Martin Fowler refers to this as the "Architect's dream, Developer's nightmare" symptom: architectural principles of loose coupling and indirection reduce the assumptions systems make about each other and therefore provide flexibility. However, testing a system where a message producer is not aware of who the consumers of this message are, can be challenging. Add to that the asynchronous and temporal aspects of messaging and things get even more complicated. For example, the messaging solution may not even be designed for the message producer to receive a reply message from the recipient(s). Likewise, the messaging infrastructure typically guarantees the delivery of the message but not the delivery time. This makes it hard to develop test cases that rely on the results of the message delivery.

When monitoring a message solution, we can track message at two different levels of abstraction. A typical **System Management** solution monitors how many messages are being sent or how long it took a message to be processed. These monitoring solutions do not inspect the message data except maybe for some fields in the message header such as the message identifier or the *Message History*. In contrast, **Business Activity Monitoring** (BAM) solutions focus on the payload data contained in the message, for example the Dollar value of all orders placed in the last hour. Many of the patterns presented in this section are general enough that they can be used for either purpose. However, because business activity monitoring is a whole new field in itself and shares many complexities with data warehousing (something we have not touched on at all), we decided to discuss the patterns in the context of system management.

System Management patterns are designed to address these requirements and provide the tools to keep a complex message-based system running. We divided this chapter into three sections:

Monitoring and Controlling

A *Control Bus* provides a single point of control to manage and monitor a distributed solution. It connects multiple components to a central management console that can display the status of

each component and monitor message traffic through the components. The console can also be used to send control commands to components, e.g. to change the message flow.

We may want to route messages through additional steps, such as validation or logging. Because these steps can introduce performance overheads, we may want to be able to switch them on and off via the control bus. A *Detour* gives us this ability.

Observing and Analyzing Message Traffic

Sometimes we need to want to inspect the contents of a message without affecting the primary message flow. A *Wire Tap* allows us to tap into message traffic.

When we debug a message-based system, is is a great aid to know where a specific message has been. The *Message History* provides this capability without introducing dependencies between components.

While the *Message History* is tied to an individual message, a central *Message Store* can provide a complete account of every message that traveled through the system. Combined with the *Message History* the *Message Store* can analyze all possible paths messages can take through the system.

The *Wire Tap*, *Message History*, and *Message Store* help us analyze the asynchronous flow of a message. In order to track messages sent to request-reply services, we need to insert a *Smart Proxy* into the message stream.

Testing and Debugging

Testing a messaging system before deploying it into production is a good idea. But testing should not stop there. You should be actively verifying that the running messaging system is functioning properly. You can do this by periodically injecting a *Test Message* into the system and verifying the results.

When a component fails or misbehaves, it is easy to end up with unwanted messages on a channel. During testing it is very useful to remove all remaining messages from a channel so that the components under test do not receive 'leftover' messages. A *Channel Purger* does that for us.

Control Bus

Naturally, enterprise integration systems are distributed. In fact, one of the defining qualities of an enterprise messaging system is to enable communication between disparate systems. Messaging systems allow information to be routed and transformed so that data can be exchanged between these systems. In most cases, these applications are spread across multiple networks, buildings, cities or continents.

How can we effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area?

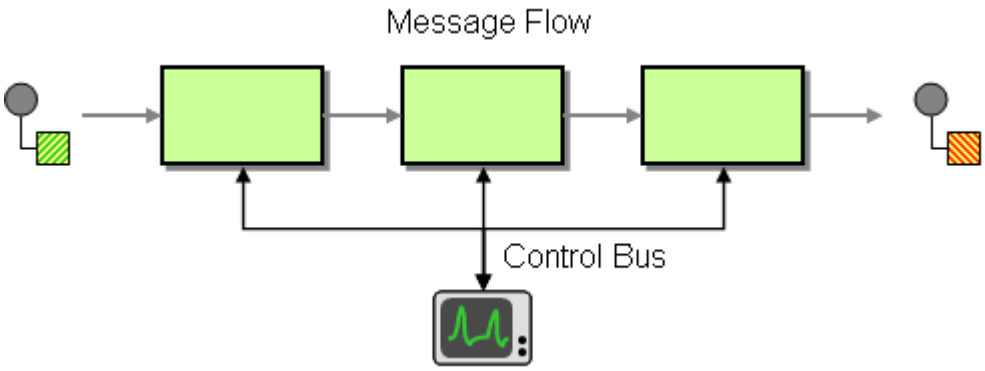
A distributed, loosely coupled architecture allows for flexibility and scalability. At the same time it poses serious challenges for administration and control of such a system. For example, how can you tell whether all components are up and running? A simple process status won't suffice because processes are distributed across many machines. Also, if you cannot obtain status from a remote machine does it mean that the remote machine is not functioning or may the communication with the remote machine be disturbed?

Besides just knowing whether a system or a component is up and running you also need to monitor the dynamic behavior of the system. What is the message throughput? Are there any unusual delays? Are channels filling up? Some of this information requires tracking of message travel times between components or through components. This requires the collection and combination of information from more than one machine.

Also, just reading information from components may not be sufficient. Often times, you need to make adjustments or change configuration settings while the system is running. For example, you may need to turn logging features on or off while the system is running. Many applications use property files and error logs to read configuration information and report error conditions. This approach tends to work well as long as the application consists of a single machine, or possibly a small number of machines. In a large, distributed solution, property files would have to be copied to remote machines using some file transfer mechanism, which requires the file system on every machine to be accessible remotely. This can pose security risks and can be challenging if the the machines are connected over the Internet or a wide-area network that may not support file mapping protocols. Also, the versions of the local property files would have to be managed carefully -- a management nightmare waiting to happen.

It seems natural to try to leverage the messaging infrastructure to perform some of these tasks. For example, we could send a message to a component to change its configuration. This control message could be transported and routed just like a regular message. This would solve most of the communication problems but also poses new challenges. Configuration messages should be subject to stricter security policies than regular application messages. For example, one wrongly formatted control message could easily bring a component down. Also, what if messages are queued up on a message channel because a component is malfunctioning? If we send a control message to reset the component, this control message would get queued up with all the other messages and not reach the component in distress. Some messaging systems support message priorities that can help move control messages to the front of the queue. However, not all systems provide this ability, and the priority may not help if a queue is filled to the limit and refuses to accept another message. Likewise, some control messages are of a lower priority than application messages. If we have components publish periodic status messages, delaying or losing a 'I am alive' control message may be a lot less troublesome and delaying or losing the 'Order for $1 Million' message.

Use a *Control Bus* to manage an enterprise integration system. The *Control Bus* uses the same messaging mechanism used by the application data, but uses separate channels to transmit data that is relevant to the management of components involved in the message flow.



Each component in the system is now connected to two messaging subsystems:

- The Application Message Flow
- The *Control Bus*

The application message flow transports all application-related messages. The components subscribe and publish to these channels just like they would in an unmanaged scenario. In addition, each component also sends and receives messages from the channels that make up the *Control Bus*. These channels connect to a central management component.

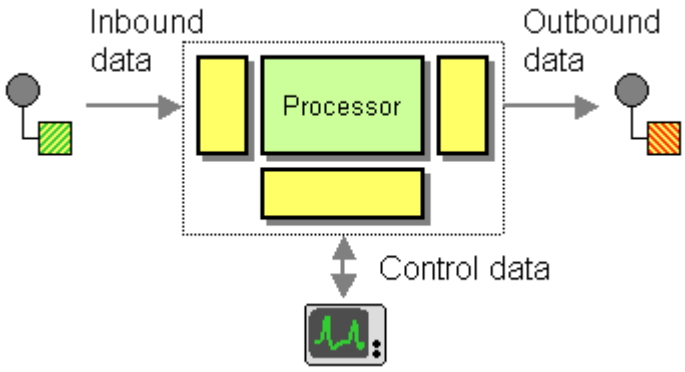
The *Control Bus* is well suited to carry the following types of messages:

- Configuration - each component involved in the message flow should have configurable parameters that can be changed as required. These parameters include channel addresses, message data formats, timeouts etc. Components use the Control Bus to retrieve this information from a central repository rather than using property files, allowing a central point of configuration and the reconfiguration of the integration solution at run-time. For example, the routing table inside a Content-Based Router may need to be updated dynamically based on system conditions, such as overload or component failure.
- Heartbeat - each component may send a periodic 'heartbeat' message on the Control Bus at specified intervals so that a central console application can verify that the component is functioning properly. This 'heartbeat' may also include metrics about the component, such as number of messages processed, the amount of available memory on the machine etc.
- Test Messages - heartbeat messages tell the Control Bus that a component is still alive, but may provide limited information on ability of the component to correctly process messages. In addition to having components publish periodic heartbeat messages to the Control Bus, we can also inject test messages into the message stream that will be processed by the components. We will then extract the message later to see whether the component processed the message correctly. As this blurs the definition of the control bus and the message bus, I decided to define a separate pattern for it (see Test Message).

- Exceptions - each component can channel exception conditions to the Control Bus to be evaluated. Severe exceptions may cause an operator to be alerted. The rules to define exception handling should be specified in a central handler.
- Statistics - Each component can collect statistics about the number of messages processed, average throughput, average time to process a message, etc. Some of this data may be split out by message type, so we can determine whether messages of a certain type are flooding the system. Since this message tends to be lower priority than other messages, it is likely that the Control Bus uses non-guaranteed or lower-priority channels for this type of data.
- Live Console - most of the functions mentioned here can be aggregated for display in a central console. From here, operators can assess the health of the messaging system and take corrective action if needed.

Many of the functions that a *Control Bus* supports resemble traditional network management functions that are used to monitor and maintain any networked solution. A *Control Bus* allows us to implement equivalent management functions at the messaging system level -- essentially elevating them from the low-level IP network level to the richer messaging level. Providing this functionality is vital to the successful operation of a messaging infrastructure, but the absence of management standards for messaging solutions makes it difficult to build enterprise-wide, reusable management solutions for messaging systems.

When we design message processing components, we architect the core processor around three interfaces (see Figure). The inbound data interface receives incoming messages from the message channel. The outbound data interface sends processed messages to the outbound channel. The control interface sends and receives control messages from and to the *Control Bus*.



Key interfaces of a messaging component.

Example: Instrumenting the Loan Broker Example

At the end of this chapter, we show how to use a *Control Bus* to instrument the *Asynchronous Implementation with MSMQ* of the Loan Broker example. This implementation includes a simple management console that displays the status of components in real-time (see *Loan Broker System Management*).

Related patterns: *Asynchronous Implementation with MSMQ*, *Content-Based Router*, *Loan Broker System Management*, *Test Message*

Detour

A *Wire Tap* is useful to inspect messages that travel across a channel. Sometimes, though, we need to modify or reroute the messages instead of simply inspecting them.

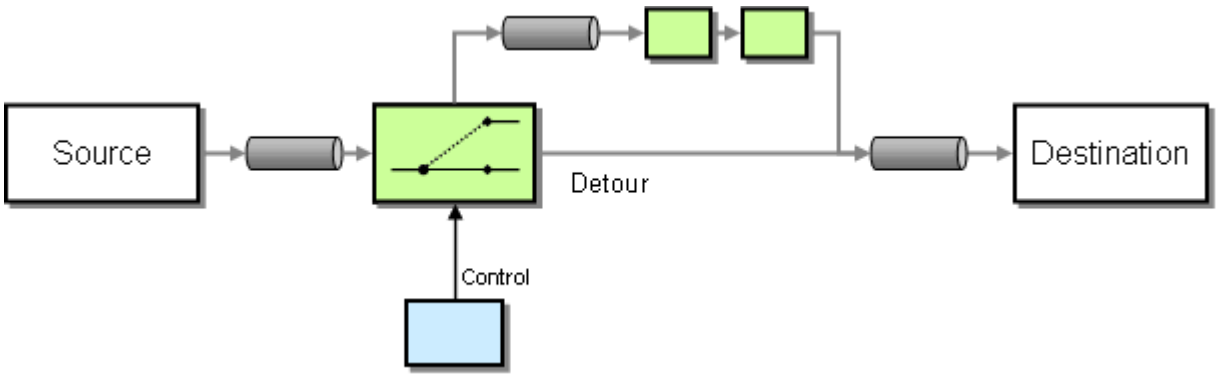
How can you route a message through intermediate steps to perform validation, testing or debugging functions?

Performing validations on messages that travel between components can be a very useful debugging tool. However, these extra steps may not always be required and would slow down the system if they are always executed.

Being able to include or skip these steps based on a central setting can be a very effective debugging or performance tuning tool. For example, while we test a system we may want to pass messages through additional validation steps. Bypassing these steps during production may improve performance. We can compare these validations to assert statements in code that are executed in the debug configuration but not in the release configuration of the executable.

Likewise, during trouble-shooting it may be useful to route messages through additional steps for logging or monitoring purposes. Being able to turn these logging steps on and off allows us to maximize message throughput under normal circumstances.

Construct a *Detour* with a context-based router controlled via the *Control Bus*. In one state the router routes incoming messages through additional steps while in the other it routes messages directly to the destination channel.



The *Detour* uses a simple context-based router with two output channels. One output channel passes the unmodified message to the original destination. When instructed by the *Control Bus*, the *Detour* routes messages to a different channel. This channel sends the message to additional components that can inspect and/or modify the message. Ultimately, these components route the message to the same destination.

If the detour route contains only a single component, it may be more efficient to combine the *Detour* switch and the component into a single filter. However, this solution assumes that the component in the detour path can be modified to include the *Control Bus*-controlled bypass logic.

The strength of controlling the *Detour* over the *Control Bus* is that multiple *Detour*s can be activated or deactivated simultaneously with a single command on the *Control Bus* using a *Publish-Subscribe Channel* from the control console to all *Detour*s.

Related patterns: *Control Bus*, *Publish-Subscribe Channel*, *Wire Tap*

Wire Tap

*Point-to-Point Channel*s are often used for *Document Message*s because they ensure that exactly one consumer will consume each message. However, for testing, monitoring or troubleshooting, it may be useful to be able to inspect all messages that travel across the channel.

How do you inspect messages that travel on a point-to-point channel?

It can be very useful to see which messages traverse a channel. For example, for simple debugging purposes or to store messages in a *Message Store*.

You can't just add another listener to the *Point-to-Point Channel* because it would consume messages off the channel and prevent the intended recipient from being able to consume the message.

Alternatively, you could make the sender or the receiver responsible to publish the message to a separate channel for inspection. However, this would force us to modify a potentially large set of components. Additionally, if we are dealing with packaged applications, we may not even be able to modify the application.

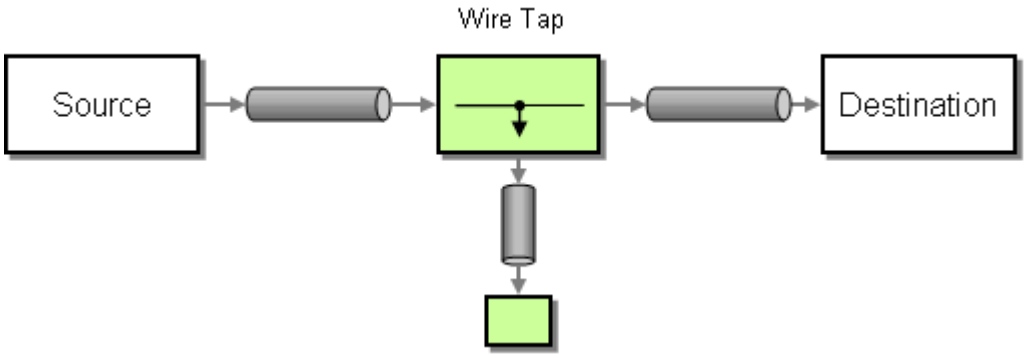
You could also consider changing the channel to a *Publish-Subscribe Channel*. This would allow additional listeners to inspect messages without disturbing the flow of messages. However, a *Publish-Subscribe Channel* change the semantics of the channel. For example, multiple *Competing Consumers* may be consuming messages off the channel, relying on the fact that only one consumer can receive a specific message. Changing the channel to the a *Publish-Subscribe Channel* would cause each consumer to receive each message. This could be very undesirable, for example if the incoming messages represent orders that now get processed multiple times. Even if only a single consumer listens on the channel, using a *Publish-Subscribe Channel* may be less efficient or less reliable than a *Point-to-Point Channel*.

Many messaging systems provide a `peek` method that allows a component to inspect a messages inside a *Point-to-Point Channel* without consuming the message. This approach has one important limitation though: once the intended consumer consumes the message, the `peek` method can no

longer see the message. Therefore, this approach does not allow us to analyze messages after they have been consumed.

You could insert a component into the channel (a form of "interceptor') that performs any necessary inspection. The component would consume a message off the incoming channel, inspect the message and pass the unmodified message to the output channel. However, frequently the type of inspection depends on messages from more than one channel (e.g. to measure message run-time) so that this function cannot be implemented by a single filter inside a single channel.

Insert a simple Recipient List into the channel that publishes each incoming message to the main channel and a secondary channel.



The *Wire Tap* is a fixed *Recipient List* with two output channels. It consumes messages off the input channel and publishes the unmodified message to both output channels. To insert the *Wire Tap* into a channel, you need to create an additional channel and change the destination receiver to consume of the second channel. Because the analysis logic is located inside a second component, we can insert a generic *Wire Tap* into any channel without any danger of modifying the primary channel behavior. This improves reuse and reduces the risk of instrumenting an existing solution.

It might be useful to make the *Wire Tap* programmable over the *Control Bus* so that the secondary channel (the "tap") can be turned on or off. This way, the *Wire Tap* can be instructed to publish messages to the secondary channel only during testing or debugging cycles.

The main disadvantage of the *Wire Tap* is the additional latency incurred by consuming and republishing a message. Many integration tool suites automatically decode a message even if it is published to another channel without modification. Also, the new message will receive a new message id and new time stamps. These operations can add up to additional overhead.

Because the *Wire Tap* publishes two messages, it is important not to correlate messages by their message ID. Even though the primary and the secondary channel receive identical messages, most messaging systems automatically assign a new message ID to each message in the system. This means that the original message and the "duplicate" message have different message IDs.

A *Message Broker* can easily be turned into a *Wire Tap* because all messages already pass through this central component.

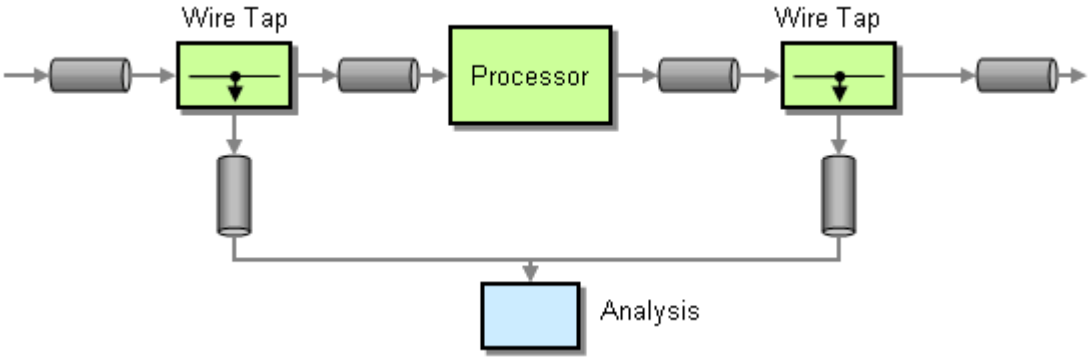
An important limitation of the *Wire Tap* is that it cannot alter the messages flowing across the channel. If you need to be able to manipulate messages, use a *Detour* instead.

Example: Loan Broker

At the end of this chapter we enhance the loan broker to include a *Wire Tap* on the request channel to the Credit Bureau to keep a log of all requests made to this external service (see *Loan Broker System Management*).

Example: Using Multiple Wire Taps to Measure Message Runtime

One of the strengths of the *Wire Tap* is that we can combine multiple *Wire Tap*s to send copies of messages to a central component for analysis. That component can be a *Message Store* or another component that analyses relationships between messages, e.g. the time interval between two related messages. (see picture)



Using a Pair of Wire Taps to Analyze Message Runtime

Related patterns: *Competing Consumers*, *Control Bus*, *Detour*, *Document Message*, *Message Broker*, *Message Store*, *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Recipient List*, *Loan Broker System Management*

Message History

One of key benefits of a message-based systems is the loose coupling between participants; the message sender and recipient make no (or few) assumptions about each other's identity. If a message recipient retrieves a message from a message channel, it generally does not know nor care which application put the message on the channel. The message is by definition

self-contained and is not associated with a specific sender. This is one of the architectural strengths of message-based systems.

However, the same property can make debugging and analyzing dependencies very difficult. If we are not sure where a message goes, how can we assess the impact of a change in the message format? Likewise, if we don't know which application published a message it is difficult to correct a problem with the message.

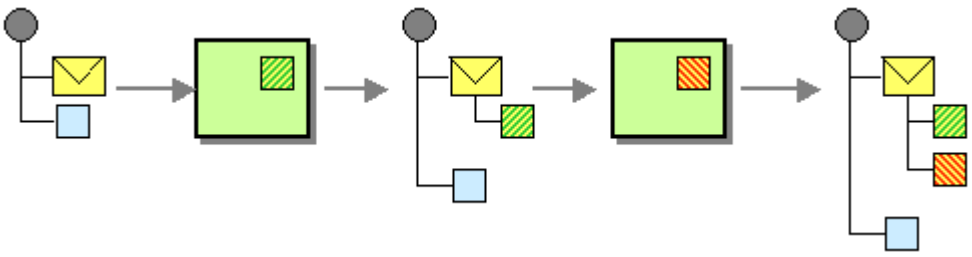
How can we effectively analyze and debug the flow of messages in a loosely coupled system?

The *Control Bus* monitors the state of each component that processes messages, but it does not concern itself with the route that an individual message takes. You could modify each component to publish the unique message identifier of each message that passes through it to the *Control Bus*. This information can then be collected in a common database, a *Message Store*. This approach requires a significant amount of infrastructure, including a separate data store. Also, if a component needs to examine the history of a message, it would have to execute a query against a central database, running the risk of turning the database into a bottle neck.

Tracking the flow of a message through a system is not as simple as it appears. It would seem natural to use the unique message ID associated with each message. However when a component (e.g. a *Message Router* processes a message and publishes it to the output channel, the resulting message will receive a new message identifier that is not associated with the message that the component consumed. Therefore, we would need to identify a new key that is copied from the incoming message to outgoing message to that the two message can be associated later. This can work reasonable well if the component publishes exactly one message for every message it consumes. However, this is not the case for many components, e.g. a *Recipient List*, an *Aggregator*, or a *Process Manager*.

Instead of identifying the path of each message by tagging the messages, the message itself could collect a list of components that it traversed. If each component in the messaging system carries a unique identifier, each component could add its identifier to each message it publishes.

Therefore, attach a *Message History* to the message. The *Message History* is a list of all applications that the message passed through since its origination.



The *Message History* maintains a list of all components that the message passed through. Every component that processes the message (including the originator) adds one entry to the list. The *Message History* should be part of the message header because it contains system-specific control

information. Keeping this information in the header separates it from the message body that contains application specific data.

Not every message that a component publishes is the result of a single message. For example,an *Aggregator* publishes a single message that carries information collected from multiple messages, each of which could have its own history. If we want to represent this scenario in the *Message History*, we have two choices. If we want to track the complete history we can enhance the *Message History* to be stored as a hierarchical tree structure. Because of the recursive nature of a tree structure we can store multiple message histories under a single 'node'. Alternatively, we can keep a simple list and only keep the history of one incoming message. This can work well if one incoming message is more important to the result than other, auxiliary messages.

The *Message History* is most useful if a series of messages flows through a number of filters to perform a specific business function or process. If it is important to manage the path that a message takes, a *Process Manager* can be useful. The *Process Manager* creates one process instance for each incoming trigger message. The flow of the message through various components is now managed centrally, alleviating the need to tag each message with it's history.

Equipping a message with its history has also another important benefit when using *Publish-Subscribe Channel*s to propagate events. Assume we implement a system that propagates address changes to multiple systems via *Publish-Subscribe Channel*s. Each address change is broadcast to all interested systems so that they may update their records. This approach is very flexible towards the addition of new systems -- the new system will automatically receive the broadcast message without requiring any changes to the existing messaging system. Assume the customer care system is one of the systems that stores addresses in the application database. Each change to the database fields causes a message to be triggered to notify all systems of the change. By nature of the publish-subscribe paradigm, all systems subscribing to the 'address changed' channel will receive the event. The customer care system itself has to subscribe to this channel, in order to receive updates made in other systems, e.g. through a self-service Web site. This means that the customer care system will receive the message that it just published. This received message would result in a database update, which will in turn trigger another 'address changed' message. We could end up in an infinite loop of 'address changed' messages. To avoid such an infinite loop, the subscribing applications can inspect the *Message History* to determine whether the message originated from the very same system and ignore the incoming message if this is the case.

Example: TIBCO ActiveEnterprise

Many EAI integration suites include support for a *Message History*. For example, the message header of every ActiveEnterprise message includes a `tracking` field that maintains a list of all components though which the message has passed. In this context it is important to note that a TIBCO ActiveEnterprise component assigns outgoing messages the same message ID as the consumed message. This makes tracking of messages through multiple component easier, but it also means that the message ID is not a system-wide unique property because multiple

individual messages share the same ID. For example, when implementing a *Recipient List* TIBCO ActiveEnterprise transfers the ID of the consumed message to each outbound message.

The following example shows the dump of a message that passed through multiple components, including two Integration Manager processes, `OrderProcess` and `VerifyCustomerStub`.

```
tw.training.customer.verify.response
{
  RVMSG_INT      2  ^pfmt^       10
  RVMSG_INT      2  ^ver^        30
  RVMSG_INT      2  ^type^       1
  RVMSG_RVMSG  108  ^data^
  {
    RVMSG_STRING  23  ^class^     "VerifyCustomerResponse"
    RVMSG_INT      4  ^idx^       1
    RVMSG_STRING   6  CUSTOMER_ID  "12345"
    RVMSG_STRING   6  ORDER_ID     "22222"
    RVMSG_INT      4  RESULT       0
  }
  RVMSG_RVMSG  150  ^tracking^
  {
    RVMSG_STRING  28  ^id^  "4OEaDEoiBIpcYk6qihzzwB5Uzzw"
    RVMSG_STRING  41  ^1^   "imed_debug_engine1-OrderProcess-Job-4300"
    RVMSG_STRING  47  ^2^   "imed_debug_engine1-VerifyCustomerStub-Job-4301"
  }
}
```

Related patterns: *Aggregator*, *Control Bus*, *Message Router*, *Message Store*, *Process Manager*, *Publish-Subscribe Channel*, *Recipient List*

Message Store

As the *Message History* describes, the architectural principle of loose coupling allows for flexibility in the solution, but can make it difficult to gain insight into the dynamic behavior of the integration solution.

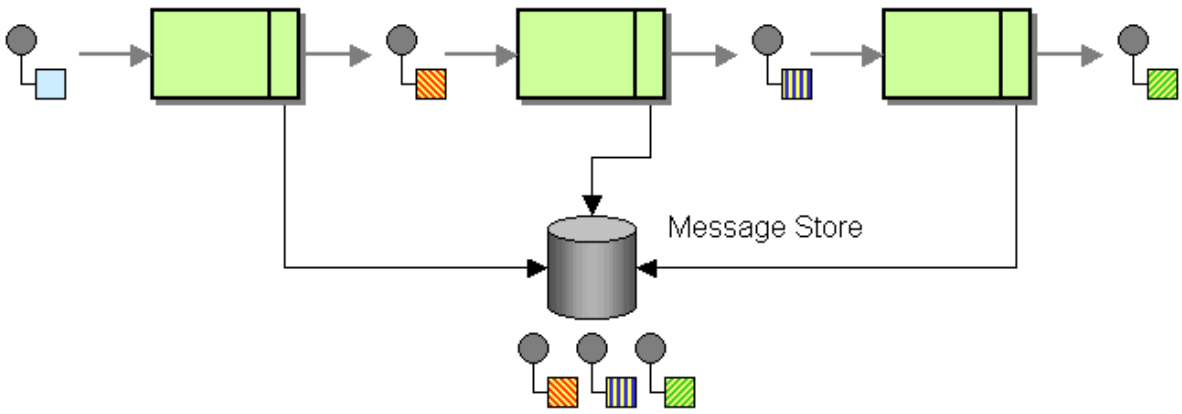
How can we report against message information without disturbing the loosely coupled and transient nature of a messaging system?

The very properties that make messaging powerful can also make it difficult to manage. Asynchronous messaging guarantees delivery, but does not guarantee when the message will be delivered. For many practical applications, though the response time of a system may be critical. It may also be important to know how many messages passed through the system within a certain time interval.

The *Message History* pattern illustrates the usefulness of being able to tell the "source" of a message. From this data, we can derive interesting message throughput and runtime statistics. The only downside is that the information is contained within each individual message. There is no easy way to report against this information since it is spread across many messages. Also, the lifetime of a message can be very short. Once the message is consumed, the *Message History* may no longer be available.

In order to perform meaningful reporting, we need to store message data persistently and in a central location.

Use a *Message Store* to capture information about each message in a central location.



When using a *Message Store*, we can take advantage of the asynchronous nature of a messaging infrastructure. When we send a message to a channel, we send a duplicate of the message to a special channel to be collected by the *Message Store*. This can be performed by the component itself or we can insert a *Wire Tap* into the channel. We can consider the secondary channel that carries a copy of the message as part of the *Control Bus*. Sending a second message in a 'fire-and-forget' mode will not slow down the flow of the main application messages. It does, however, increase network traffic. That's why we may not store the complete message, but just a few key fields that are required for later analysis, such as a message ID, or the channel on which the message was sent and a timestamp.

How much detail to store is actually an important consideration. Obviously, the more data we have about each message, the better reporting abilities we have. The counter-forces are network traffic and storage capacity of the *Message Store*. Even if we store all message data, our reporting abilities may still be limited. Messages typically share the same message header structure, but the message content is structured differently. Therefore, we may not be able to easily report against the data elements contained in a message.

Since the message data is different for each type of message, we need to consider different storage options. If we create a separate storage schema (e.g. tables) for each message type that matches that message type's data structure, we can apply indexes and perform complex searches on the message content. However, this assumes that we have a separate storage structure for each

message type. This could become a maintenance burden. We could also store the message data as unstructured data in XML format in a long character field. This allows us a generic storage schema. We could still query against header fields, but would not be able to report against fields in the message body. However, once we identified a message, we can recreate the message content based on the XML document stored in the *Message Store*.

The *Message Store* may get very large, so most likely we will need to consider introducing a purging mechanism. This mechanism could move older message logs to a back-up database or delete it altogether.

Example: Commercial EAI Tools

Some enterprise integration tools supply a *Message Store*. For example, MSMQ allows queues to automatically store sent or received messages in a *Journal Queue*. Microsoft BizTalk optionally stores all documents (messages) in a SQL Server database for later analysis.

Related patterns: *Control Bus*, *Message History*, *Wire Tap*

Smart Proxy

A pair of *Wire Tap*s can be used to track messages that flow through a component. However, this approach assumes that the component publishes messages to a fixed output channel. However, many service-style components publish reply messages to the channel specified by the *Return Address* included in the request message.

How can you track messages on a service that publishes reply messages to the Return Address specified by the requestor?

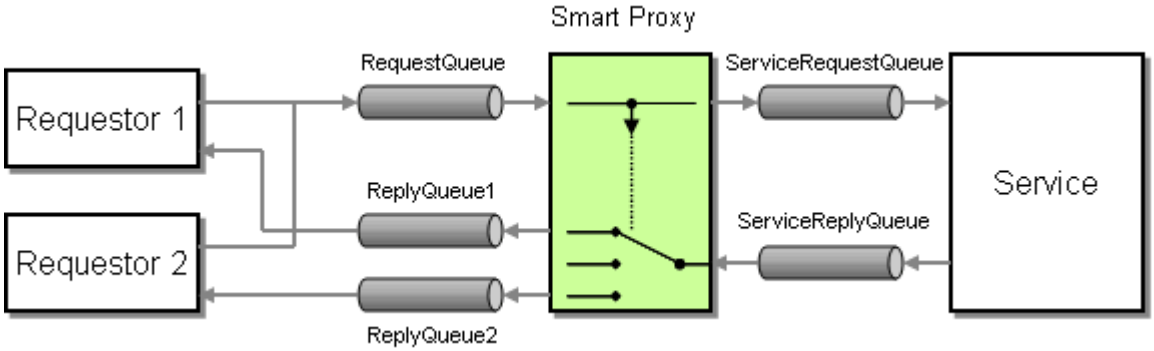
In order to track messages flowing through a service, we need to capture both request and reply messages. Intercepting a request message using a *Wire Tap* is easy enough. Intercepting reply messages is the tough part because the service publishes the reply message to different channels based on the requestor's preferred *Return Address*.

The support of a *Return Address* is required for most *Request-Reply* services so that a requestor can specify the channel that the reply message should be sent to. Changing the service to post reply messages to a fixed channel would make it hard for each requestor to extract the correct reply messages. Some messaging systems allow consumers to "peek" for specific messages inside a single reply queue but that approach is implementation specific and does not work in those instances where the reply message does not go back to the requestor but to a third party.

As discussed in the *Wire Tap*, modifying the component to inspect messages is not always feasible or practical. If we are dealing with a custom application we may not be able to modify the application code and have to implement a solution that is external to the application. Likewise,

we may not want to require each application to implement inspection logic, especially because the nature of the logic may vary depending on whether we operate in test mode or production mode. Keeping the inspection functions in a separate self-contained component improves flexibility, reuse and testability.

Use a *Smart Proxy* to store the Return Address supplied by the original requestor and replace it with the address of the *Smart Proxy*. When the service sends the reply message route it to the original Return Address.



The *Smart Proxy* intercepts messages sent on the request channel to the *Request-Reply* service. For each incoming message, the *Smart Proxy* stores the *Return Address* specified by the original sender. It then replaces the *Return Address* in the message with the channel the reply channel that the *Smart Proxy* is listening on. When a reply message comes in on that channel, the *Smart Proxy* retrieves the stored *Return Address* and uses a *Message Router* to forward the unmodified reply address to that channel.

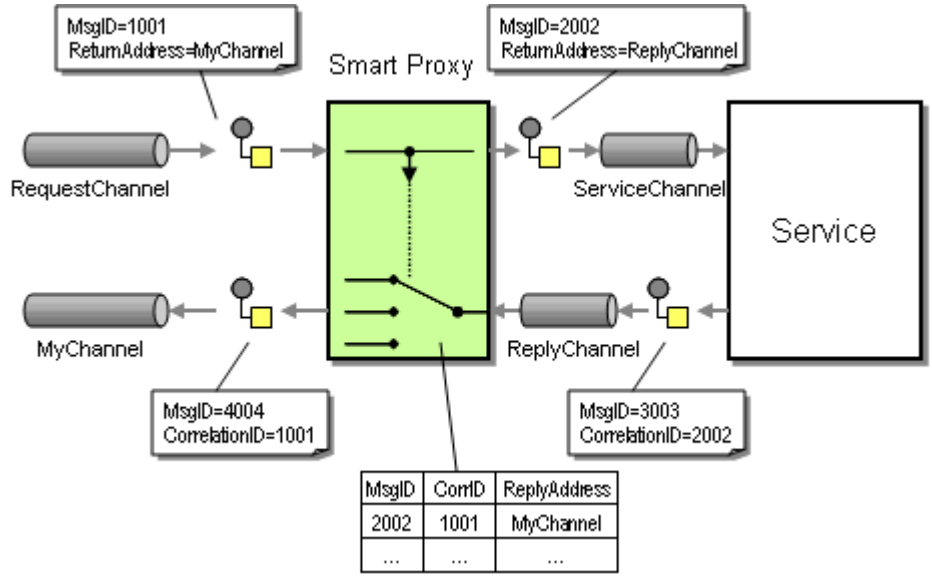
The *Smart Proxy* is also useful in cases where an external service does not support a *Return Address* but instead replies to a fixed reply channel. We can proxy such a service with a *Smart Proxy* to provide support for a *Return Address*. In this case the *Smart Proxy* performs no analytical functions, but simply forwards the reply message to the correct channel.

The *Smart Proxy* needs to store the *Return Address* supplied by the original requestor in such a way that it can correlate incoming reply messages with the *Return Address* and forward the reply message to the correct channel. The *Smart Proxy* can store this data in two places:

- Inside the Message
- Inside the *Smart Proxy*

To store the *Return Address* inside the message, the *Smart Proxy* can add a new message field with the *Return Address* to the message. The *Request-Reply* service is required to copy this field the reply message. All the *Smart Proxy* has to do is to extract the special message field from the reply message, remove the field from the message and forward the message to the channel specified by the field. This solution keeps the *Smart Proxy* simple but it requires collaboration by the *Request-Reply* service. If the *Request-Reply* service is a non-modifiable component, this option may not be available.

Alternatively, the *Smart Proxy* can store the *Return Address* in dedicated storage, for example in a memory structure or a relational database. Because the purpose of the *Smart Proxy* is to track messages between the request and reply message the *Smart Proxy* usually has to store data from the request message anyway to correlate it to the reply message and to analyze both messages in unison. This approach requires the *Smart Proxy* to be able to correlate the reply message to the response message. Most *Request-Reply* services support a *Correlation Identifier* that the service copies from the request message to the reply message. If the *Smart Proxy* cannot modify the original message format, it can (ab-)use this field to correlate request and reply messages. The *Smart Proxy* should construct its own *Correlation Identifier* because not all requestors will specify a *Correlation Identifier* and also because the *Correlation Identifier* needs to be unique only across requests made by a single requestor and not across multiple requestors. Because the service reply queue now carries messages from multiple requestors using the original *Correlation Identifier* is not reliable. Therefore, the *Smart Proxy* stores the original *Correlation Identifier* together with the original *Return Address* and replaces the original *Correlation Identifier* with its own *Correlation Identifier* so that it can retrieve the original *Correlation Identifier* and *Return Address* when the reply message arrives. Many services us the Message ID of the request message as the *Correlation Identifier* for the reply message. This introduces another problem. The service will now copy the Message ID of the request message it received from the *Smart Proxy* to the reply message to the *Smart Proxy*. The *Smart Proxy* needs to replace this *Correlation Identifier* in the reply message with the Message ID of the original request message so that the requestor can properly correlate request and reply messages. The following picture illustrates this process:

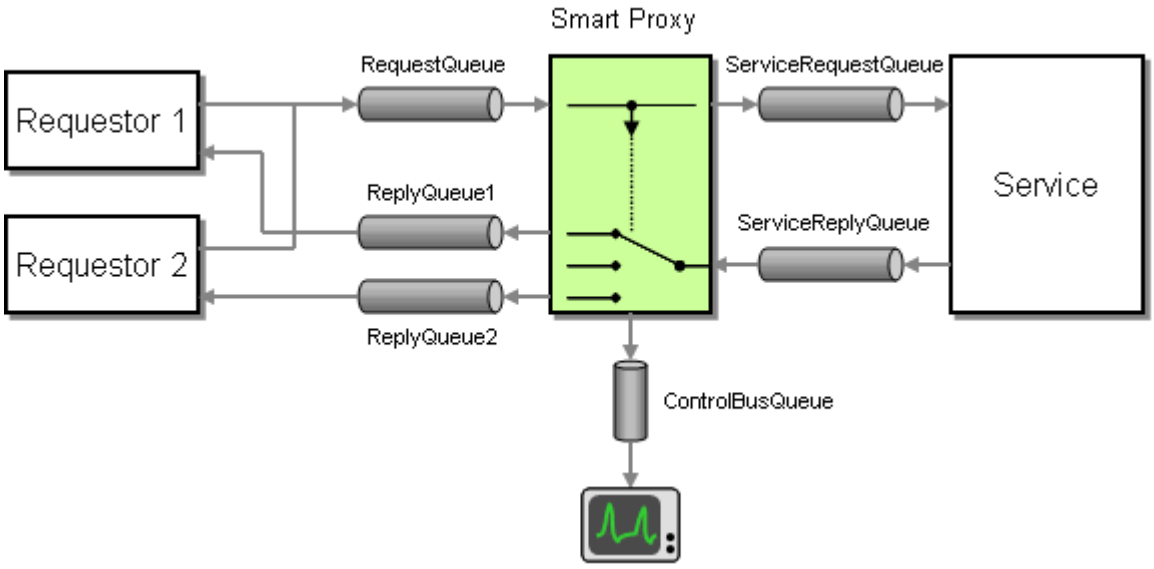


Storing and Replacing the Correlation Identifier and Return Address

Example: Simple Smart Proxy in MSMQ and C#

Implementing a *Smart Proxy* is not as complicated as it sounds. The following code implements a solution scenario consisting of two requestors, a *Smart Proxy* and a simple service. The *Smart Proxy* passes the message processing-time to the control bus for display in the console. We want

to allow the requestors to correlate by either the message Id or the numeric `AppSpecific` property provided by the `Message` object.



Simple Smart Proxy Example

For our coding convenience, we define a base class `MessageConsumer` that encapsulates a the code that is required to create an event-driven message consumer. Inheriting classes can simply overload the virtual method `ProcessMessage` to perform any necessary message handling and do not have to worry about the configuration of the message queue or the event-driven processing. Separating this code into a common base class makes it easy to create test clients and a dummy request-reply service with just a few lines of code.

MessageConsumer

```
public class MessageConsumer
{
    protected MessageQueue inputQueue;

    public MessageConsumer (MessageQueue inputQueue)
    {
        this.inputQueue = inputQueue;
        SetupQueue(this.inputQueue);
        Console.WriteLine(this.GetType().Name +  ": Processing messages from " +
inputQueue.Path);
    }

    protected void SetupQueue(MessageQueue queue)
    {
        queue.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
        queue.MessageReadPropertyFilter.ClearAll();
```

```
        queue.MessageReadPropertyFilter.AppSpecific = true;
        queue.MessageReadPropertyFilter.Body = true;
        queue.MessageReadPropertyFilter.CorrelationId = true;
        queue.MessageReadPropertyFilter.Id = true;
        queue.MessageReadPropertyFilter.ResponseQueue = true;
    }

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

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

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

        ProcessMessage(m);

        mq.BeginReceive();
    }

    protected virtual void ProcessMessage(Message m)
    {
        String text = "";
        try
        {
            text = (String)m.Body;
        }
        catch (InvalidOperationException) {};
        Console.WriteLine(this.GetType().Name + ": Received Message " + text);
    }
}
```

With the `MessageConsumer` class as a starting point, we can create a *Smart Proxy*. A *Smart Proxy* contains two `MessageConsumer`s, one for the request messages coming from the requestors and one for the reply messages returned by the request-reply service. The *Smart Proxy* also defines a Hashtable to store message data between request and reply message.

SmartProxy

```
public class SmartProxyBase
{
    protected SmartProxyRequestConsumer requestConsumer;
    protected SmartProxyReplyConsumer replyConsumer;

    protected Hashtable messageData;

    public SmartProxyBase(MessageQueue inputQueue, MessageQueue serviceRequestQueue,
MessageQueue serviceReplyQueue)
    {
        messageData = Hashtable.Synchronized(new Hashtable());
        requestConsumer = new SmartProxyRequestConsumer(inputQueue, serviceRequestQueue,
serviceReplyQueue, messageData);
        replyConsumer = new SmartProxyReplyConsumer(serviceReplyQueue, messageData);
    }

    public virtual void Process()
    {
        requestConsumer.Process();
        replyConsumer.Process();
    }
}
```

The `SmartProxyRequestConsumer` is relatively simple. It stores relevant information from the request message (message ID, the *Return Address*, the `AppSpecific` property, and the current time) in the hashtable, indexed by the message Id of the new request message sent to the actual service. The request-reply service copies this message ID to the `CorrelationID` field of the service reply message so that the *Smart Proxy* can retrieve the stored message data. The `SmartProxyRequestConsumer` also replaces the *Return Address* (the `ResponseQueue` property) with the queue that the *Smart Proxy* listens on for reply messages. We included a virtual method `AnalyzeMessage` in this class so that subclasses can perform any desired analysis.

SmartProxyRequestConsumer

```
public class SmartProxyRequestConsumer : MessageConsumer
{
    protected Hashtable messageData;
    protected MessageQueue serviceRequestQueue;
    protected MessageQueue serviceReplyQueue;

    public SmartProxyRequestConsumer(MessageQueue requestQueue, MessageQueue
serviceRequestQueue, MessageQueue serviceReplyQueue, Hashtable messageData) :
base(requestQueue)
```

```
    {
        this.messageData = messageData;
        this.serviceRequestQueue = serviceRequestQueue;
        this.serviceReplyQueue = serviceReplyQueue;
    }

    protected override void ProcessMessage(Message requestMsg)
    {
        base.ProcessMessage(requestMsg);

        MessageData data = new MessageData(requestMsg.Id, requestMsg.ResponseQueue,
requestMsg.AppSpecific);
        requestMsg.ResponseQueue = serviceReplyQueue;
        serviceRequestQueue.Send(requestMsg);
        messageData.Add(requestMsg.Id, data);
        AnalyzeMessage(requestMsg);
    }

    protected virtual void AnalyzeMessage(Message requestMsg)
    {
    }
}
```

The `SmartProxyReplyConsumer` listens on the service reply channel. The `ProcessMessage` method retrieves the message data for the associated request message stored by the `SmartProxyRequestConsumer` and calls the `AnalyzeMessage` template method. It then copies the `CorrelationID` and the `AppSpecific` properties to the new reply message and routes it to the *Return Address* specified in the original request message.

SmartProxyReplyConsumer

```
public class SmartProxyReplyConsumer : MessageConsumer
{
    protected Hashtable messageData;

    public SmartProxyReplyConsumer(MessageQueue replyQueue, Hashtable messageData) :
base(replyQueue)
    {
        this.messageData = messageData;
    }

    protected override void ProcessMessage(Message replyMsg)
    {
        base.ProcessMessage(replyMsg);
```

```
        String corr = replyMsg.CorrelationId;
        if (messageData.Contains(corr))
        {
            MessageData data = (MessageData)(messageData[corr]);

            AnalyzeMessage(data, replyMsg);

            replyMsg.CorrelationId = data.CorrelationID;
            replyMsg.AppSpecific = data.AppSpecific;

            MessageQueue outputQueue = data.ReturnAddress;
            outputQueue.Send(replyMsg);
            messageData.Remove(corr);
        }
        else
        {
            Console.WriteLine(this.GetType().Name + "Unrecognized Reply Message");
            //send message to invalid message queue
        }
    }

    protected virtual void AnalyzeMessage(MessageData data, Message replyMessage)
    {
    }
}
```

In order to collect metrics and send them to the control bus we subclass both the generic `SmartProxy` and `SmartProxyReplyConsumer` classes. The new `MetricsSmartProxy` instantiates the `SmartProxyReplyConsumerMetrics` as the consumer, which includes a simple implementation of the `AnalyzeMessage` method. The method computes the message runtime between request and response and sends this data together with the number of outstanding messages to the control bus queue. We could easily enhance this method to perform more complex computations. The control bus queue is connected to a simple file writer that writes each incoming message to a file.

MetricsSmartProxy

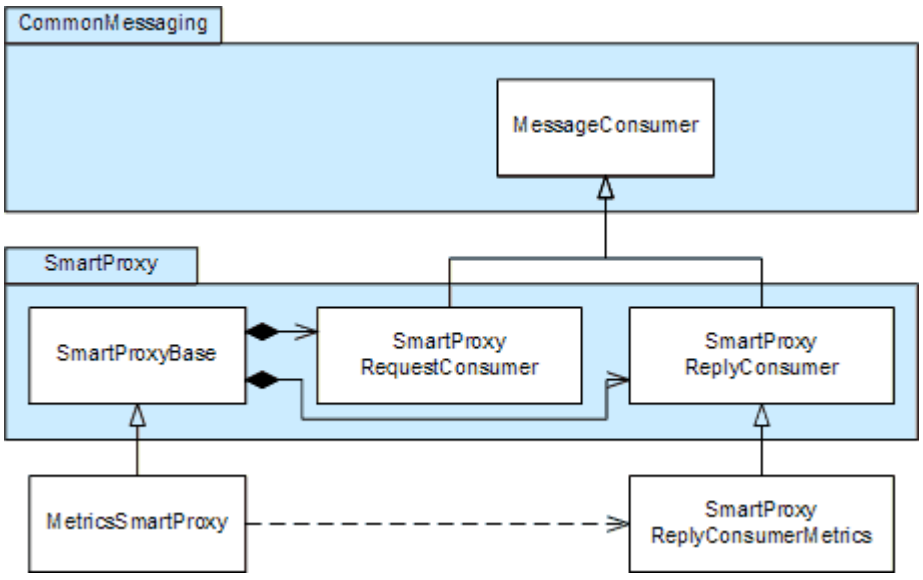
```
public class MetricsSmartProxy : SmartProxyBase
{
    public MetricsSmartProxy(MessageQueue inputQueue, MessageQueue serviceRequestQueue,
MessageQueue serviceReplyQueue, MessageQueue controlBus) :
            base (inputQueue, serviceRequestQueue, serviceReplyQueue)
    {
        replyConsumer = new SmartProxyReplyConsumerMetrics(serviceReplyQueue,
messageData, controlBus);
    }
```

```
}
```

SmartProxyReplyConsumerMetrics

```
public class SmartProxyReplyConsumerMetrics : SmartProxyReplyConsumer
{
    MessageQueue controlBus;

    public SmartProxyReplyConsumerMetrics(MessageQueue replyQueue, Hashtable
messageData, MessageQueue controlBus) : base(replyQueue, messageData)
    {
        this.controlBus = controlBus;
    }

    protected override void AnalyzeMessage(MessageData data, Message replyMessage)
    {
        TimeSpan duration = DateTime.Now - data.SentTime;
        Console.WriteLine(" processing time: {0:f}", duration.TotalSeconds);
        if (controlBus != null)
        {
            controlBus.Send(duration.TotalSeconds.ToString() + "," + messageData.Count);
        }
    }
}
```

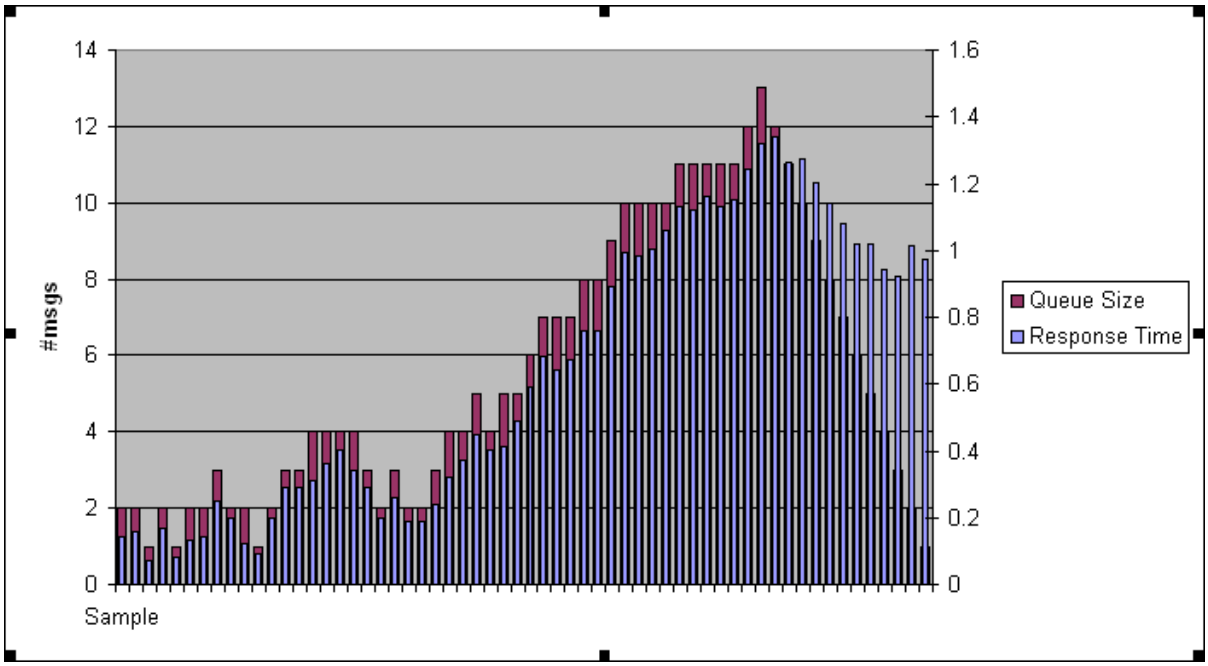
The class diagram for the solution looks as follows:



Smart Proxy Example Class Diagram

To test the proxy, we created a dummy request-reply service that does nothing but wait for a random interval between 0 and 200 ms. We feed the *Smart Proxy* from two requestors, each of

which publishes 30 messages in 100ms intervals. For demonstration purposes we loaded the resulting control bus file into a Microsoft Excel spreadsheet and created a nice looking chart:



Response Time Statistics Collected by the Smart Proxy and Visualized by the Control Bus Console

We can see that the queue size and the response time increase steadily until 13 messages are queued up. At that time, the requestors do no longer send new messages and the queue size decreases steadily. The response time decreases as well, but remains around 1 second because the message that are now being processed have been sitting in the request queue for that long.

Related patterns: *Correlation Identifier*, *Message Router*, *Request-Reply*, *Return Address*, *Wire Tap*

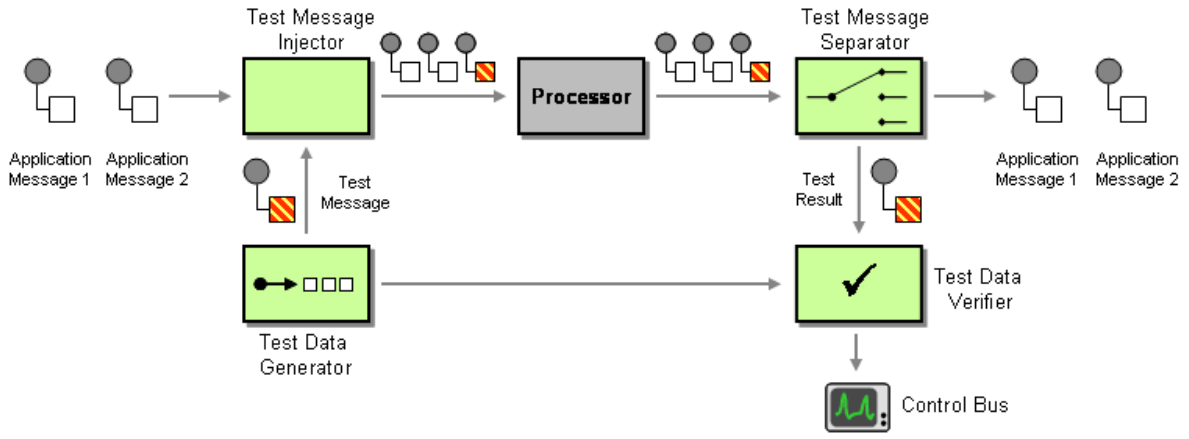
Test Message

The *Control Bus* describes a number of approaches to monitor the health of the message processing system. Each component in the system can publish periodic "heartbeat" messages to the control bus in order to keep the monitoring mechanism informed that the component is still active. The heartbeat messages can contain vital stats of the component as well, such as the number of messages processed, the average time required to process a message or the percentage of CPU utilization on the machine.

What happens, though, if a component is actively processing messages, but garbles outgoing messages due to an internal fault?

A simple heartbeat mechanism will not detect this error condition because it operates only at a component level and is not aware of application message formats.

Therefore, use *Test Message* to assure the health of message processing components (see Figure).



The *Test Message* pattern relies on the following components:

- The **Test Data Generator** creates messages to be sent to the component for testing. Test data may be constant, driven by a test data file or generated randomly.
- The **Test Message Injector** inserts test data into the regular stream of data messages sent to the component. The main role of the injector is to tag messages in order to differentiate 'real' application messages from test messages. This can be accomplished by inserting a special header field. If we have no control over the message structure, we can try to use special values to indicate test messages (e.g. OrderID = 999999). This changes the semantics of application data by using the same field to represent application data (the actual order number) and control information (this is a test message). Therefore, this approach should be used only as a last resort.
- The **Test Message Separator** extracts the results of test messages from the output stream. This can usually be accomplished by using a *Content-Based Router*.
- The **Test Data Verifier** compares actual results with expected results and flags an exception if a discrepancy is discovered. Depending on the nature of the test data, the verifier may need access to the original test data.

An explicit Test Message Separator may not needed if the component under test supports a *Return Address*. In this case the test Data Generator can include a special test channel as the *Return Address* so that test messages are not passed through the remainder of the system. Effectively, the *Return Address* acts as the tag for text messages.

Test Message is considered an active monitoring mechanism. Unlike passive mechanisms, active mechanisms do not rely on information generated by the components (e.g. log files or heartbeat messages), but actively probe the component. The advantage is that active monitoring usually achieves a deeper level of testing since data is routed through the same processing steps as the application messages. It also works well with components that were not designed to support passive monitoring.

One possible disadvantage of active monitoring is the additional load placed on the processor. We need to find a balance between frequency of test and minimizing the performance impact. Active monitoring may also incur cost if we are being charged for the use of a component on a pay-per-use basis. This is the case for many external components, e.g. if we request credit reports for our customers from an external credit scoring agency.

Active monitoring does not work with all components. Stateful components may not be able to distinguish test data from real data and may create database entries for test data. We may not want to have test orders included in our annual revenue report!

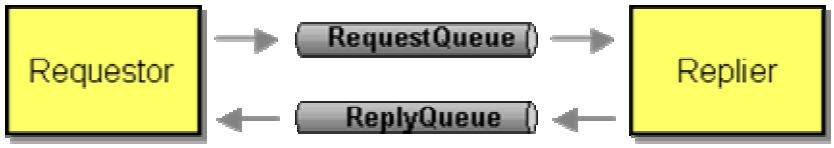
Example: Loan Broker: Testing the Credit Bureau

In the example at the end of the chapter, we use a *Test Message* to actively monitor the external Credit Bureau (see *Loan Broker System Management*).

Related patterns: *Content-Based Router*, *Control Bus*, *Return Address*, *Loan Broker System Management*

Channel Purger

When Bobby and I worked on the simple *JMS Request/Reply Example*, we ran into a simple, but interesting problem. The example consists of a Requestor which sends a message to a Replier and waits for the response. The example uses two *Point-to-Point Channel*s, `RequestQueue` and `ReplyQueue` (see picture).



We had plastered `println` statements all over the code so we can see what is going on. We started the Replier first, then the Requestor. Then a very odd thing happened. The Requestor console window claimed to have gotten a response before the Replier ever acknowledged receiving a request. A delay in the console output? Lacking any great ideas, we decided to shut the Replier down and re-ran the Requestor. Odd enough, we still received a response to our request! Magic? No, just a side effect of persistent messaging. A superfluous message was present on the `ReplyQueue`. When we started the Requestor it placed a new message on the `RequestQueue` and then immediately retrieved the extraneous reply message that was sitting on the `ReplyQueue`. We never noticed that this was not the reply to the request the Requestor just made! Once the Replier received the request message, it placed a new reply message on the `ReplyQueue` so that the 'magic' repeated during the next test. It can be amazing (or amazingly frustrating) how persistent, asynchronous messaging can play tricks on you in even the most simple scenarios!

How can you keep 'left-over' messages on a channel from disturbing tests or running systems?

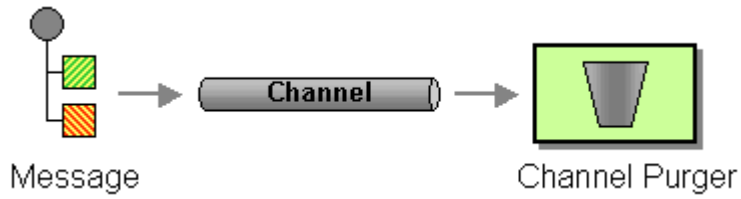
*Message Channel*s are designed to deliver messages reliably, even if the receiving component is unavailable. In order to do so, the channel has to persist messages along the way. This useful feature can cause confusing situations during testing or if one of the components misbehaves (and does not use transactional message consumption and production). We can quickly end up with extraneous messages stuck on channels as described above. These messages make it impossible to pass test data into the system until the pending messages have been consumed. If the pending messages are orders worth a few million Dollars, this is a good thing. If we are testing or debugging a system and have a channel full of query messages or of reply messages it can cause us a fair amount of headache.

In our simple example, some of our debugging pain could have been eased if we had used a *Correlation Identifier*. Using the identifier, the Requestor would have recognized that the incoming message is actually not the response to the request it just sent. It could then discard the 'old' reply message or route it to an *Invalid Message Channel*, which would effectively remove the 'stuck' message. In other scenarios, it is not as easy to detect duplicate or unwanted messages. For example, if a specific message is malformed and causes the message recipient to fail, the recipient cannot restart until the 'bad' message is removed, because it would just fail right away again. Of course, this example requires the defect in the recipient to be corrected (no malformed message should cause a component failure), but removing the message can get the system up and running quickly until the defect is corrected.

Another way to avoid left-over messages in channels is to use tomporary channels (e.g., JMS provides the method `createTemporaryQueue` for this purpose). These channels are intended for request-reply applications and lose all messages once the application closes its connection to the messaging system. But again, this approach is limited to a simple request-reply example and does not protect against other messages being left over on other channels that need to be permanent.

It may be tempting to assume that transaction management can eliminate the 'extra message' scenario because message consumption, message processing and message publication are covered in a transaction. So if a component aborts in the middle of processing a message, the message would not be considered consumed. Likewise, a reply message would not be published until the component signals the final 'commit' to send the message. We need to keep in mind though, that transactions do not protect us against programming errors. In our simple request-reply example, a programmer error may have caused the Requestor to not read a response from the `ReplyQueue` channel. As a result, despite potential transactionality, a message is stuck on that channel causing the symptoms described above.

Use a *Channel Purger* to remove unwanted messages from a channel.



A simple *Channel Purger* simply remove all the messages from a channel. This may be sufficient for test scenarios where we want to reset the system into a consistent state. If we are debugging a production system we may need to remove individual message or a set of messages based on specific criteria, such as the message ID or the values of specific message fields.

In many cases it is alright for the *Channel Purger* to simply delete the message from the channel. In other cases, we may need the *Channel Purger* to store the removed messages for later inspection or replay. This is useful if the messages on a channel cause a system to malfunction, so that we need to remove them to continue operation. However, once the problems are corrected, we want to re-inject the message(s) so that the system does not lose the contents of the message. This may also include the requirement to edit message contents before re-injecting the message. This type of function combines some of the features of a *Message Store* and a *Channel Purger*.

Example: Channel Purger in JMS

This example shows a simple *Channel Purger* implemented in Java. This example simply removes all messages on a channel. The `ChannelPurger` class references two external classes:

- `JMSEndpoint` - The base class for any JMS particpant. Provides pre-initialized instance variables for a Connection and Session instance.
- `JNDIUtil` - Implements helper functions to encapsulate the lookup of JMS objects via JNDI.

Both classes are described in more detail in *JMS Request/Reply Example*.

```
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;

public class ChannelPurger extends JmsEndpoint
{

   public static void main(String[] args)
   {

      if (args.length != 1) {
         System.out.println("Usage: java ChannelPurger <queue_name>");
         System.exit(1);
      }
```

```
        String queueName = new String(args[0]);
        System.out.println("Purging queue " + queueName);

        ChannelPurger purger = new ChannelPurger();

        purger.purgeQueue(queueName);

    }

    private void purgeQueue(String queueName)
    {
        try {
            initialize();
            connection.start();
            Queue queue = (Queue) JndiUtil.getDestination(queueName);

            MessageConsumer consumer = session.createConsumer(queue);

            while (consumer.receiveNoWait() != null)
                System.out.print(".");
            connection.stop();
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                    // ignore
                }
            }
        }
    }
}
```

Related patterns: *Correlation Identifier*, *Invalid Message Channel*, *Message Channel*, *Message Store*, *Point-to-Point Channel*, *JMS Request/Reply Example*