

Enterprise Integration Patterns

Gregor Hohpe
Sr. Architect, ThoughtWorks
gregor@hohpe.com

July 23, 2002

Introduction

Integration of applications and business processes is a top priority for many enterprises today. Requirements for improved customer service or self-service, rapidly changing business environments and support for mergers and acquisitions are major drivers for increased integration between existing “stovepipe” systems. Very few new business applications are being developed or deployed without a major focus on integration, essentially making integratability a defining quality of enterprise applications.

Many different tools and technologies are available in the marketplace to address the inevitable complexities of integrating disparate systems. Enterprise Application Integration (EAI) tool suites offer proprietary messaging mechanisms, enriched with powerful tools for metadata management, visual editing of data transformations and a series of adapters for various popular business applications. Newer technologies such as the JMS specification and Web Services standards have intensified the focus on integration technologies and best practices.

Architecting an integration solution is a complex task. There are many conflicting drivers and even more possible ‘right’ solutions. Whether the architecture was in fact a good choice usually is not known until many months or even years later, when inevitable changes and additions put the original architecture to test. There is no cookbook for enterprise integration solutions. Most integration vendors provide methodologies and best practices, but these instructions tend to be very much geared towards the vendor-provided tool set and often lack treatment of the bigger picture, including underlying guidelines and principles. As a result, successful enterprise integration architects tend to be few and far in between and have usually acquired their knowledge the hard way.

What we need are enterprise integration patterns, similar to the architecture and design patterns documented in the realm of application architecture and design. These patterns would communicate our collective experience in designing solutions to recurring design problems and help establish a common vocabulary for integration architects.

This paper introduces a set of integration patterns harvested from multiple years of hands-on enterprise integration work with a variety of organizations. The collection of patterns form a textual as well as a visual pattern language that can help design and describe integration solutions. Depending on their primary purpose, the patterns are grouped into message routing patterns, message transformation patterns and message management patterns.

The Need for Integration

Enterprises are typically comprised of hundreds if not thousands of applications that are custom-built, acquired from a third-party, part of a legacy system, or a combination thereof, operating in

multiple tiers of different operating system platforms. We may ask how businesses allow themselves to get into such a mess.

First of all, writing business applications is hard. Creating a single, big application to run a complete business is next to impossible. The ERP vendors have had some success at creating larger-than-ever business applications. The reality, though, is that even the heavyweights like SAP, Oracle, Peoplesoft and the like only perform a fraction of the business functions required in a typical enterprise. We can see this easily by the fact that ERP systems are one of the most frequently accessed integration points in today's enterprises.

Spreading the responsibilities across multiple applications gives an enterprise a fair amount of flexibility. It is then up to the business to select the accounting package, customer relationship management or order processing system that best suits the business' needs. One-stop-shopping for enterprise applications is usually not what IT organizations are interested in, or is not possible given the number of applications or specific business requirements.

Vendors have learned to cater to this preference and offer focused applications that provide functionality around a certain core function. However, there is definitely some functionality spillover amongst business applications. For example, many billing systems provide some customer care functionality and some accounting functionality. Likewise, the customer care software maker takes a stab at implementing simple billing functions such as disputes or adjustments. Nevertheless, a robust enterprise solution needs a best-of-breed customer relationship management system and a highly functional billing system.

On the other hand, customers, business partners and internal business prefer to interact with an enterprise by business process, regardless of the number and nature of the internal systems. A calling customer may want to change his or her address and see whether the last payment was received. In many enterprises, this simple request already spans across customer care and billing systems. Imagine a customer placing a new order. We need to validate the customer, verify the customer's good standing, verify that we have inventory, fulfill the order, get a shipping quote, compute sales tax, send a bill, etc. This process can easily span across five or six different systems. From the customer's perspective, it is a single process.

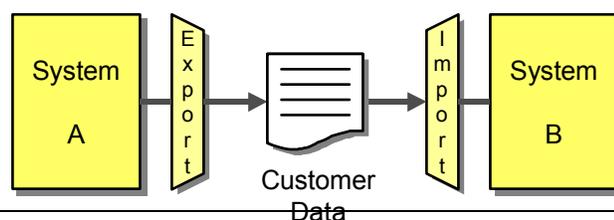
In order to support common business processes and data sharing across applications, these applications need to be integrated. Application integration needs to provide efficient, reliable and secure data exchange between multiple enterprise applications. Effective integration also requires central monitoring and management of potentially hundreds of systems and components, plus the ability to deploy changes on a global scale. Integration solutions present unique challenges due to their sheer size as well the number of layers they span – from high-level business process management down to network packet routing and systems management.

Integration Strategies

Since application integration is not a new requirement, a number of different approaches evolved over the past two or three decades. This section discusses the most commonly used techniques, highlighting their respective strengths and weaknesses.

Batch Data Exchange

As soon as businesses started to use more than a single computer system, the need for integration became apparent. Most initial efforts to achieve integration across multiple computer systems were based on



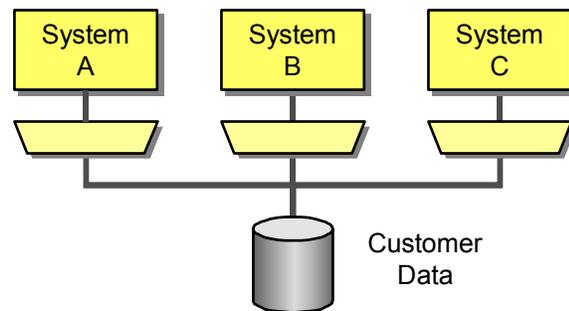
transfer of data in files. Mainframe systems tended to operate nightly batch cycles, which presented an opportunity to extract information from one system at the beginning of the nightly cycle, move the file to the other system and load the data into the target system before the end of the night.

While this approach seems simplistic, it does have some advantages. Using files affords good physical decoupling between the different processes. If the target system is not immediately available to receive the file, it can be stored until the system becomes ready. Also, files are essentially platform and language independent as long as they use a common character set.

However, batch file transfers also present a number of challenges. Data changes in one system may not be available in the other system until the next day. This may be confusing to users, and cause data integrity problems if the (outdated) data is updated in the other system. Additionally, batch exchanges tend to extract all available information from one system and replicate it to the other system. If only few changes were made, this approach will cause large amounts of unnecessary data transmission.

Shared Database

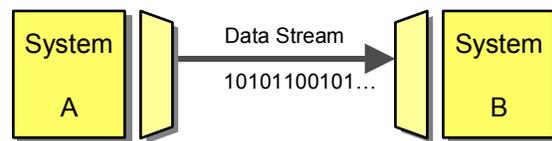
In an attempt to eliminate data synchronization issues and the replication of huge amounts of data, some enterprises went to create a shared database resource that would be shared by all systems. Having the data in a single place would eliminate the need for duplication of data. Synchronization issues and contention can be addressed by the locking and protection mechanisms provided by most database systems.



The key drawback of this solution lies in the difficulty of defining a data model that fits all applications. Many applications are built or purchased with proprietary data models and many attempts to morph all these models into a single, universal data model turned out to be a never-ending task. Also, accessing a single database may cause a serious performance bottleneck. In addition, this solution still simply exchanges data items, rather than managing business functions across systems.

Raw Data Exchange

Over time, it became clear that a single data store was not a viable solution, especially as a larger number of specialized packaged applications made their way into the enterprise, becoming part of a so-called “best-of-breed”



solution. Another way to interchange information on a more timely basis was found in direct data exchange through network data transfer protocols, such as TCP/IP Sockets. These mechanisms enable direct data exchange between two systems in (near) real time.

The advantage of the direct data exchange is that information can be propagated right as the data is modified in the source system. This real-time replication reduces the chance of data getting out of synch and avoids mass replication in nightly cycles.

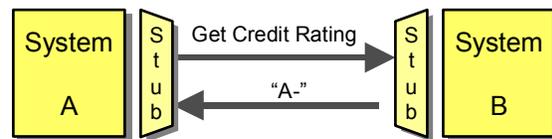
Direct data exchange mechanisms also brought new challenges. This form of exchange occurs synchronously, i.e. while data is sent through the socket (or pipe), the receiving system has to be actively receiving. Meanwhile, the originating system is waiting for a response from the target

system. If the target system or the network is unavailable, no data exchange can occur, and the originating system is stuck waiting until a time-out occurs. In many occasions, developers have created sophisticated buffer and retry mechanisms to provide reliable communication across sockets.

Also, raw data exchange mechanisms are built on the exchange of byte-by-byte, unstructured data streams and are unable to represent complex data types. Therefore, developers are responsible for creating code to convert all source data into character streams and to convert them back into data structures at the receiving end. Without a management infrastructure, this approach is error-prone and difficult to maintain.

Remote Procedure Calls

Remote Procedure Call (RPC) mechanisms isolate the application from the raw data exchange mechanisms through an additional layer. This layer manages the marshalling of complex data types into byte streams as required by the transport layer. As a result, RPC mechanisms allow an application to transparently invoke a function implemented in another application. The marshalling mechanisms rely on stubs generated from the interface specification in a language-neutral Interface Definition Language (IDL).

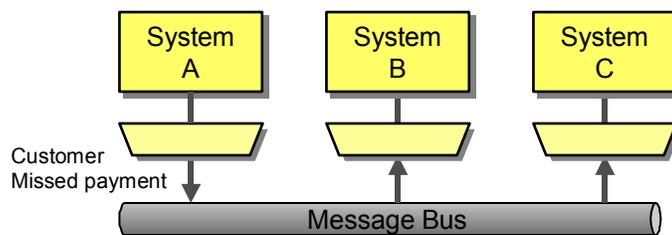


While RPC-based systems made dealing with the integration much simpler, these systems still share the drawbacks of unreliable, synchronous communication with the raw data exchange methods. RPC-based systems also imply brittle point-to-point connections between senders and receivers that become difficult to maintain as the number of participants increases. Furthermore, most vendor-provided RPC implementations are not interoperable with other products.

One interesting twist is the fact that the recent efforts around SOAP and XML-RPC imply a renaissance of this architecture with all its associated challenges.

Messaging

Messaging systems try to overcome the drawbacks of the previous solutions by providing reliable, asynchronous data transfer. An application can publish data to the integration layer and can be assured that the data will be delivered to the recipient(s). The originating system is not required to wait for an acknowledgement and can continue the primary process flow. Messaging systems also incorporate addressing schemes that avoid the difficult to maintain point-to-point connections that are typical in RPC systems.



At this time, it appears that messaging systems provide the best basis for an integration solution. However, there are still some challenges to be overcome. Messaging systems are still comparatively new and the body of knowledge around architecting these systems is still evolving. Asynchronous systems require a different approach to systems architecture and design than synchronous systems do. Furthermore, the asynchronous nature of the interaction can make testing and debugging more difficult as well.

The following section describes the defining properties of a messaging solution in more detail.

Message-Oriented Middleware

As described in the previous section, many different implementations of integration systems or ‘middleware’ exist. Even within the realm of messaging, many different approaches and technologies exist. For example, many different EAI vendors provide ‘messaging’ solutions, as do JMS implementations or solutions based on web services protocols such as SOAP. In order to stay independent of specific technical implementations, we define Message-Oriented Middleware (MOM) based on the following defining qualities:

Communication between disparate systems

Message-Oriented Middleware needs to enable communication between systems that are written in different programming languages, run on different computing platforms in different locations.

Transport discrete packets of information (messages)

In Message-Oriented Middleware, information is sent in discrete messages as opposed to streams of data. Each message is self-contained and can be routed and processed independently.

Uses addressable conduits

Messages are transported via conduits. Messages can be placed into conduits based on an addressing scheme. The addressing scheme can be based on a number of principles, with flat or hierarchical schemes being the most common. Physical implementations typically employ a form of Universal Resource Identifiers (URIs) such as queue names, channel names or message subjects as addresses.

Synchronous or Asynchronous communication

The conduits may queue the messages if a system is temporarily unavailable (yielding asynchronous communication) or not (yielding synchronous communication). The conduits may also provide for multicast distribution (multiple recipients for the same message) or single-point distribution (a single recipient per message).

In summary, we can say:

A Messaging system is one that allows communication between separate systems via discrete packets of information (called messages) transmitted through individually addressable conduits. The conduits may queue the messages (yielding asynchronous communication) or not (yielding synchronous communication). The conduits may provide for multicast distribution (one message to many receivers) or single-point distribution (one message to one receiver). The conduits may also provide different quality of service levels, ranging from best effort to guaranteed delivery.

Pattern Categories

Enterprise integration spans many problem domains and levels of abstraction. Therefore, it makes sense to divide enterprise integration patterns into categories that reflect the scope and abstraction of the patterns.

Message Routing Patterns

Message Routing Patterns discuss mechanisms to direct messages from a sender to the correct receiver. Message-based integration uses addressable conduits to send messages between systems. Senders and receivers agree on a common addressing scheme so that a message sent to a certain address is received by the correct recipient (or set of recipients). In many cases, the message recipient is unknown to the sender or is derived dynamically based on a set of conditions. Also, messages may be split up, routed separately and re-merged later. Generally,

message routing patterns leave the content of a message (largely) unaffected, but move messages from one addressable conduit to another.

Message Transformation Patterns

Message Transformation Patterns change the information content of a message. In many cases, a message format needs to be changed due to different needs of the sending or the receiving system. Data may have to be added, taken away or existing data may have to be rearranged. These tasks are accomplished by message transformers.

Message Management Patterns

Message Management Patterns provide the tools to keep a complex message-based system running. A message-based integration solution can process thousands or even millions of messages in a day. Messages are generated, routed, transformed and consumed. The solution has to deal with error conditions, performance bottlenecks and changes in the participating systems. Message management patterns address these requirements.

Pattern Description and Notation

Pattern descriptions should follow a consistent format. A number of pattern formats have been established in the pattern community (e.g., [PatternForms]). Christopher Alexander defines patterns at the highest level using a context-problem-solution structure [Alexander], augmented by pictures, diagrams, and pointers to related patterns. The patterns in this paper are intended to follow the Alexandrian form of pattern presentation. Each pattern consists of the pattern name, a brief description of the context, the problem statement, an elaboration of the forces, the proposed solution followed by a diagram, plus a discussion of related patterns and an optional example. I opted not to spell out the headings for each of the subsections in the pattern description because I believe that it hinders the flow of the prose somewhat and makes it look more like a reference manual than a paper. Whenever possible, I discuss relationships between patterns, but I have not yet been able to order patterns consistently in a way that Alexander achieves. However, I attempted to order patterns within each pattern category by complexity and dependencies between patterns to facilitate overall readability and flow.

Integration architects rely on diagrams as much (and often times more) as they do on written language. Therefore, I attempted to provide a visual pattern language in addition to the “regular” pattern language. Consequently, each pattern receives a name as well as a symbol that characterizes the pattern and distinguishes it from other patterns.

Not all patterns are represented using a single symbol. Many patterns are composed of other patterns. In that case, the image of the derived pattern is a composition of other pattern symbols connected by conduits. This enables the reader to get a quick glance at pattern dependencies.

In order to define the visual pattern language, I defined a consistent notation that I use for all patterns. Each illustration consists of three main elements (see Figure 1):

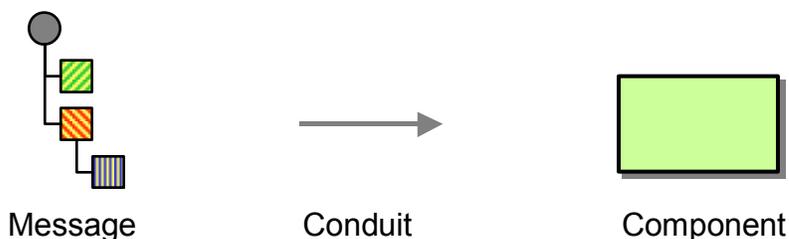


Figure 1: Notation

Messages represent data packets traveling through conduits. In most enterprise integration systems, messages are tree-like structures of data elements. These messages can take the format of an XML document, or a vendor specific implementation. Each message contains a root node (represented as a circle) and a tree of nested data elements (represented as squares). To indicate how messages are transformed, I use colors and patterns to distinguish individual data elements. Any similarities with the Funkengruven people are purely coincidental.

The conduit represents the transport for messages, similar to the Forward-Receiver pattern in [POSA]. Conduits can be implemented using queuing, multicast or broadcast mechanisms. The patterns in this paper assume a single recipient per conduit unless a broadcast mechanism is explicitly indicated. As per our earlier definition, conduits have to be addressable in a way that each conduit is described by a unique address or URI. In the proposed notation, conduit addresses are not spelled out – different conduits are indicated by using separate arrows.

Components receive, process and send messages. These components can be implemented many different ways, including (but not limited to) Java programs processing JMS messages, .Net components receiving a SOAP request, EAI message broker or transformation engines, or existing business applications that are connected to the messaging system via adapters.

The resulting diagrams resemble UML collaboration diagrams (see [UML]). However, since so many patterns are concerned with data transformations, I chose to enrich the notation by message symbols to visualize some of the transformation to the message content. The *UML For EAI Finalization Tasks Force* (FTF) of the OMG has recently published a *UML Profile for EAI* ([UMLEAI]). This document augments UML Collaboration Diagrams with new stereotypes and constraints to model interactions amongst EAI components. The intent is to define a common metadata interchange standard for EAI specifications. This specification includes a vocabulary of common EAI components. While the intent is somewhat different from that of the pattern language presented in this paper, there are a number of similarities between the resulting languages. The specification is in its final stages of acceptance, so I will keep a close eye on this specification and potentially rename or rework some patterns to be consistent with this specification.

Enterprise integration is a large space. This paper discusses a small set of patterns and is far from comprehensive. I started with patterns that I have seen being applied most frequently in practice. I hope that these patterns will motivate those people involved in enterprise integration to collect additional patterns and to add to the body of understanding. I am sure there are many more enterprise integration patterns (and pattern categories) waiting to be discovered.

Examples

Providing examples for enterprise integration patterns is not without challenges. Enterprise integration systems typically consist of a number of heterogeneous components, spread across multiple systems. Examples that fit within the limitations of a paper may be so trivial as to be meaningless. Furthermore, the majority of integration solutions are constructed using proprietary EAI suites from vendors such as IBM, TIBCO, WebMethods, SeeBeyond and the like. These tools generally require the reader to be familiar with the vendor-provided development tools. To make things more challenging, many of these tools provide only rudimentary print functions, leaving screen shots as the only effective way of sharing code examples.

In an attempt to provide simple, but meaningful examples, I drew examples from the following technologies: TIBCO ActiveEnterprise, JMS and XSL. TIBCO is used as an arbitrary example of a proprietary EAI suite. The examples are based on the TIBCO ActiveEnterprise 4.0 suite of products. Most other vendors provide comparable features, at least at the level of these simple

examples. JMS is quickly becoming an accepted “standard” for messaging in Java-based environments. The simple examples I show require a JMS 1.02-compatible messaging provider. Lastly, I used XML and XSL to illustrate some message transformation patterns.

Most of the examples are stripped down to illustrate the point without filling pages and pages with code. As such, they should be seen as illustrative tools only and not as a starting point for development of an integration solution. Almost all examples lack any form of error checking or robustness.

Message Routing Patterns

Message-based systems use addressable conduits to send messages from message senders to receivers. Senders publish messages to a specific conduit and receivers listen on one or more specific conduits. Addressing schemes help organize and separate streams of messages. Receivers are able to subscribe only to those conduits that are of interest, avoiding having to sift through a pile of messages, discarding all irrelevant messages. Depending on the specific implementation, addressing schemes can be linear or hierarchical.

Message Routing Patterns are concerned with the addressing of messages. Most message routing patterns receive a message from one conduit and republish the message on a different conduit. Routing patterns may affect the message structure (e.g. separation or aggregation), but do not significantly alter the information content itself.

Pipes and Filters

In many enterprise integration scenarios, a single event triggers a sequence of processing steps, each of which must fulfill specific requirements. For example, a new order is placed by a customer in the form of a message. One requirement is 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. As a result of these requirements, the incoming order message has to undergo a series of transformation and processing steps. We receive a possibly duplicated, encrypted message that contains extra authentication data, but the order management system may expect a simple plain-text order message without all the extraneous data fields.

To meet these requirements a solution must be defined that passes incoming messages through a sequence of individual processing steps. The sequence of the steps may depend on the message source, the message structure or other factors, and may change over time. Additionally, the individual processing units should not make any assumptions about the processing sequence.

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?

Individual processing steps may be implemented in multiple existing systems or may be implemented in different programming languages or on top of different operating system platforms. This makes a single module solution very complex or impossible.

Therefore, use the *Pipes and Filters* pattern to divide a larger processing task into a sequence of smaller, independent processing steps (*Filters*) that are connected by a conduit (*Pipe*). Each filter processes messages received from the inbound conduit and publishes the results to the outbound conduit. The pipe directs output from that filter to the next filter. This solution allows us to add, change, or remove specific processing steps easily without affecting other processing steps.

Drawn out in the notation we defined earlier the incoming order problem can be solved as follows:

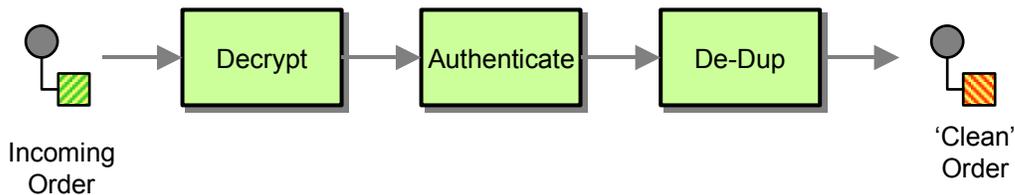


Figure 2: Pipes and Filters Example

The Pipes and Filters pattern uses messaging conduits to decouple components from each other. The messaging infrastructure provides language, platform and location independence between the filters. This affords us the flexibility to move processing steps to a different machine for maintenance or performance reasons.

The Pipe and Filter pattern describes a processing pipeline where each unit in the process can operate in its own thread or its own processor. This allows multiple items to be processed concurrently as they pass through the individual stages. For example, as the first message has been decrypted and is being authenticated, the next message can already be decrypted. However, the overall throughput is limited to the slowest process in the chain. Many messaging systems allow us to deploy multiple instances of a process. The message queue guarantees one-of-N delivery, i.e. it makes sure that each message is routed to exactly one processor of N available, identical processors. This allows us to speed up the most time-intensive process and improve overall throughput. The requirement is that each Filter is stateless, i.e. it returns to the previous state after a message has been processed.

Pipes and Filters are a common architectural pattern and are discussed in detail by Buschmann et al. in [POSA]. Almost all integration-related implementations of the Pipes and Filters pattern 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. As such, the pipes are assumed to be rather 'dumb', whereas the filters contain all the logic. 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 pattern:

Content-Based Router

Assume that we are creating an order processing system. When an incoming order has been received, we will perform some initial validation and then verify that the ordered item is available in the warehouse. This particular function is performed by the inventory system. We could use the Pipes and Filters (8) pattern to route messages first to the validation filter(s), then to the inventory system. However, in many enterprise integration scenarios, more than one inventory system exists. This situation arises frequently as a result of acquisitions or business partnerships. In many of these situations, some of the systems may be operated within the company and some

systems may be operated by business partners or affiliates. Good examples are many of the large e-tailers like Amazon.

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

We would like to abstract the fact that the implementation is spread across multiple systems from the remainder of the integration solution, so that other participants need not be concerned about implementation details. So how do we architect the system so that the incoming message is sent to the correct recipients even though all messages arrive on the same pipe? Let us assume that our company is selling widgets and gadgets. We have two inventory systems—one for widgets and one for gadgets. When we receive an order, we need to decide which inventory system to pass the order to. We could create separate channels for incoming orders based on the type of item ordered. This would require the customer to know our internal system architecture when in fact the customer may not even be aware that we distinguish between widgets and gadgets.

We could also send the order to all inventory systems, and let each system decide whether the order can be handled by it. This approach will make the addition of new inventory systems easy, but it 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? In many cases, the systems will treat an order for an item that they do not contain as an error. In this scenario, each order would cause errors in all inventory systems but one. It will be hard to distinguish these errors from ‘real’ errors such as an invalid order.

We could also consider using the item number itself as the conduit address. Each item would have its dedicated conduit. The inventory systems could listen on the conduits for those items that it can process. This approach would leverage the conduit addressability to route messages to the correct inventory system. However, it would lead to an explosion of conduit definitions. In most integration implementations conduit addressing needs to be somewhat static, due to performance and management considerations. Creating new conduits for each item that is offered would quickly result in chaos.

Therefore, use a *Content-Based Router* to route messages to the correct recipient based on message content (see Figure 3).

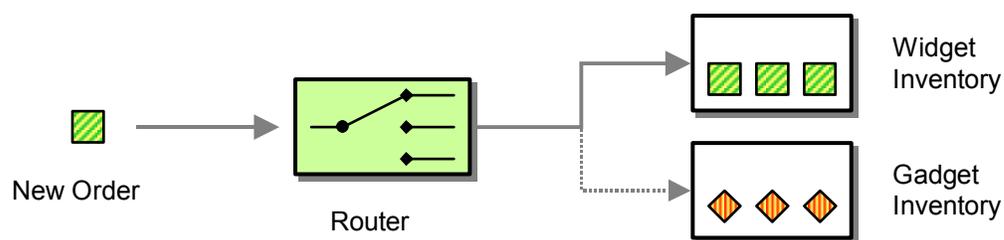


Figure 3: Content-Based Router

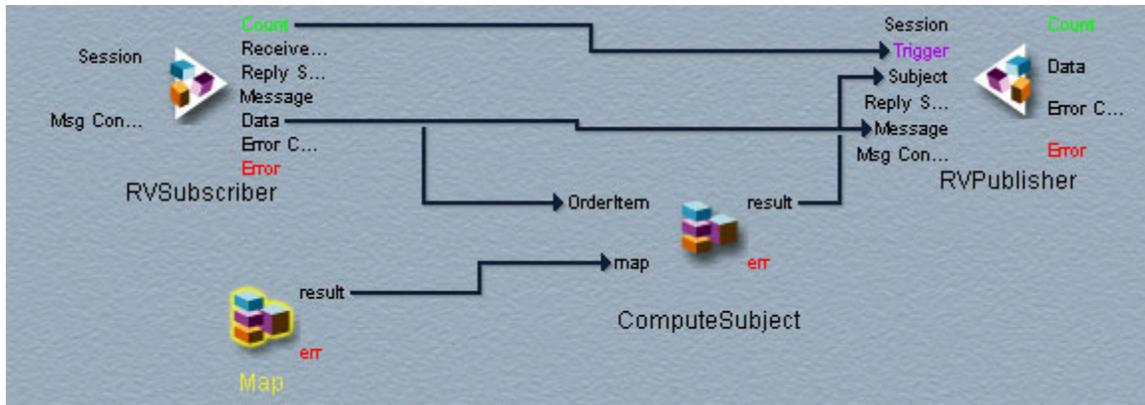
This routing algorithm may be simple or complex, depending on the type of information that is available. For example, if we can arrange item numbers so that they begin with ‘W’ for widgets and ‘G’ for gadgets, the router can be quite simple. The important point is that this logic is contained in a single component. If the naming convention changes, only the router logic is affected.

[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 Content-Based Router is different from the Dispatcher in that it can be more dynamic than a simple table lookup. In addition, the Dispatcher assumes that the client knows which service to request. In integration scenarios, the 'client' is typically only concerned with its own message processing and may not even be aware what the next processing step is. The Routing Table (19) pattern is a common extension of the Content-Based Router.

Example

A simple content-based router implemented in TIBCO MessageBroker looks like this:



The RVSubscriber receives messages from the subject router.in. The message content is sent to the message publisher as well as the function ComputeSubject. For our simple routing example, this function is defined as follows:

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

Based on the first letter of the order number, the subject name is looked up from a hashmap. The result of this function is used as the subject of the RVPublisher. For this simple example, the hashmap contains the following values:

W widget
G gadget

As a result, any order item whose item number starts with a 'G' is routed to the subject router.out.gadget, whereas any item whose item number starts with a 'W' is routed to the subject router.out.widget. More complex routing algorithms can be implemented in a similar fashion.

Sequencer

Typically, an order placed by a customer consists of more than just a single item. As outlined in the Content-Based Router (9) pattern, the order may contain a mix of items, each of which may be managed through a different inventory system. 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 the message contains multiple elements, each of which may have to be processed by a different component?

We could send the complete order to each order management system and let it pick out the items that it can handle. This approach would have the same disadvantages described in the Content-Based Router (9) pattern. It would be very difficult to avoid missing or duplicate shipment of individual items.

We could also repeat the message n times, where n represents the number of elements contained in the message. We would then add a counter field that indicates which element is currently to be processed. This approach would address the issue, but could be very confusing, as the messages look nearly identical. We are also generating an enormous amount of traffic in situations where an order contains main items.

Therefore, use a *Sequencer* to break out the composite message into individual messages, each containing data related to one item (Figure 4).

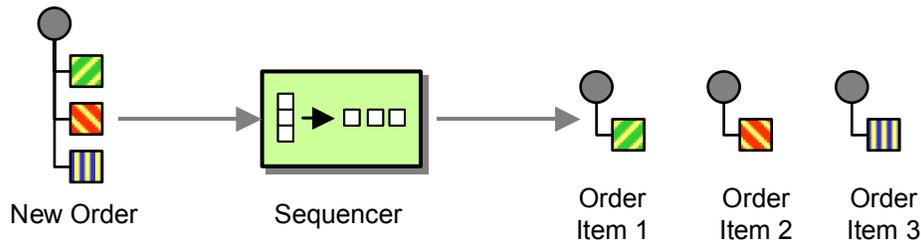


Figure 4: Sequencer

The sequencer receives one message which contains a list of repeating elements that can be processed individually. The sequencer publishes a sequence of new messages each of which contains a single element from the original message.

As mentioned earlier, many enterprise integration systems represent message data in a tree format. Naturally, trees can contain repeating elements (leaves or sub-trees) underneath a node. The sequencer iterates through all children under a specified node and packages each of them into a separate message. Typically, the resulting ‘child’ message contains elements underneath the child node, augmented by common elements that reside at the document root level that are needed to make the message context-free. For example, each order item message will contain the customer number and order number.

Child messages may also contain a sequence number to improve message traceability and simplify the task of an Aggregator (13). The message may also contain a reference to the original (combined) message so that processing results from the individual messages can be correlated back to the original message. This use of a common reference is similar to the Correlation Identifier pattern presented in [BrownWoolf].

If message envelopes are used (see Envelope Wrapper / Unwrapper (28)), each new message is generally supplied with its own message envelope.

Example

Many messaging systems use XML messages. For example, an incoming order may look as follows:

```
<order>
  <date>7/18/2002</date>
  <customer>
    <id>12345</id>
    <name>Joe Doe</name>
  </customer>
  <orderitems>
    <item>
      <quantity>1.0</quantity>
      <itemno>W1234</itemno>
      <description>A Widget</description>
    </item>
  </orderitems>
</order>
```

```
<item>
  <quantity>2.0</quantity>
  <itemno>G2345</itemno>
  <description>A Gadget</description>
</item>
</orderitems>
</order>
```

For this order the sequencer could generate the following two child messages:

```
<orderitem>
  <sequence>1</sequence>
  <total>2</total>
  <date>7/18/2002</date>
  <customerid>12345</customerid>
  <quantity>1.0</quantity>
  <itemno>W1234</itemno>
  <description>A Widget</description>
</orderitem>
```

```
<orderitem>
  <sequence>2</sequence>
  <total>2</total>
  <date>7/18/2002</date>
  <customerid>12345</customerid>
  <quantity>2.0</quantity>
  <itemno>G2345</itemno>
  <description>A Gadget</description>
</orderitem>
```

Each orderitem received a sequence number and the total number of items. This will facilitate the re-aggregation of items later. Each orderitem also contains the customer ID and the order date so that the message does not require any context assumptions. This is important if the messages are processed by stateless servers. For convenience, the customer hierarchy has been eliminated, a common strategy employed in the Content Filter (24) pattern.

Aggregator

The Sequencer (11) pattern tells us how we can break out a message into a sequence of sub-messages that can be processed individually. In many cases, we want to continue processing the original (composite) message, after all the sub-messages have been processed. We may also be sending a message to a broadcast channel, which results in multiple recipients receiving the same message. Each recipient may respond to the message, generating a series of response messages.

How do we create a mechanism to aggregate the results of individual, but related messages back into a single message?

The asynchronous nature of a messaging system makes collection of information from 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 sequencer, the response messages may not arrive in the same sequence. As messages can be processed by multiple components distributed across the network, the messaging infrastructure guarantees the delivery of a message, but does not deal with the order in which the messages are processed.

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? Should we hold up processing because a single message is late?

Therefore, use the *Aggregator* pattern to manage the reconciliation of multiple, related messages into a single message.

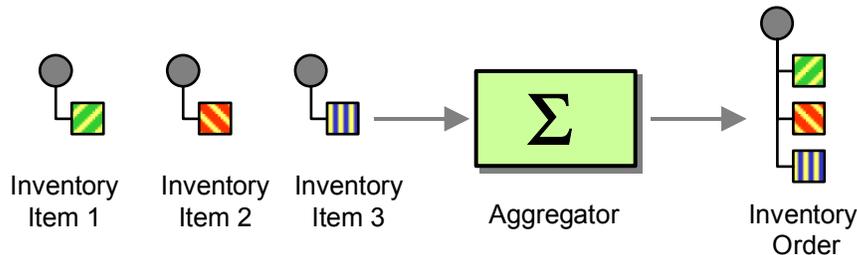


Figure 5: Aggregator

For each message that the Aggregator receives, it checks whether the message is part of an existing aggregate. If no aggregate exists for the message, the Aggregator starts a new aggregate and adds the message. If an aggregate already exists, the Aggregator simply adds the message to the aggregate. In either case, the Aggregator evaluates the completeness condition for the aggregate. If the condition evaluates to true, a new message is formed from the aggregate and published to the output channel. If the completeness condition evaluates to false, no message is published.

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 Sequencer example). Depending on the knowledge of the Aggregator, the most common strategies are as follows:

- Wait until all responses are received. This scenario is most likely in the order example. An incomplete order may not be meaningful. If not all items are received within a certain time-out period an error condition is raised by the Aggregator. This approach may give us the best basis for decision-making, but may also be the slowest and most brittle.
- 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.
- 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.
- 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.

Distribution with Aggregate Response

The order-processing example presented in the Content-Based Router (9) and Sequencer (11) 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, the original order message is ready to be passed on to the next processing step.

How do we split a message into individual processing units and continue processing the main message once all the sub-messages have been processed?

To address this processing requirement, we could chain a Sequencer (11) pattern and a Content-Based Router (9) pattern using the Pipes and Filters (8) architectural pattern (Figure 6).

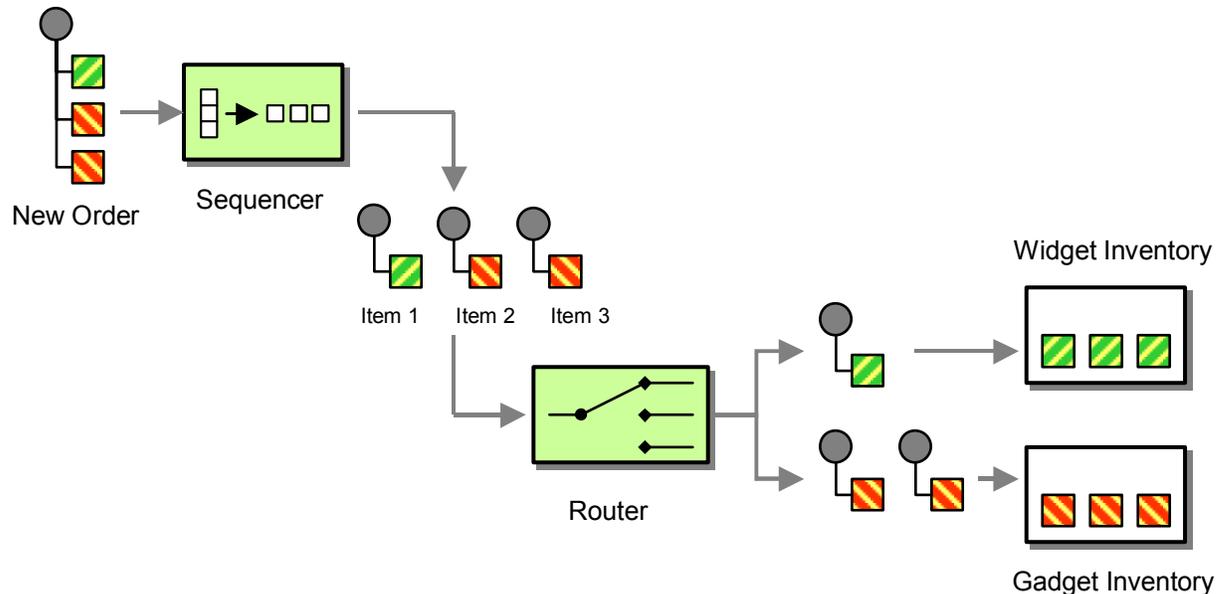


Figure 6: Sequencer and Content-Based Router

The Sequencer (11) splits an incoming order into the individual line items, each of which is routed to the proper inventory system to be checked. The inventory systems are decoupled from each other and each system receives only items that can be processed by it.

The major 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.

In many cases where a sequencer is used to process individual items contained in a message, we need to define a mechanism that can aggregate the processing results and compile them into a single response message.

One approach would be to just reassemble all items that have been passed to 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.

Therefore, use *Distribution with Aggregate Response* to process a composite message when the results of each element are required to continue processing.

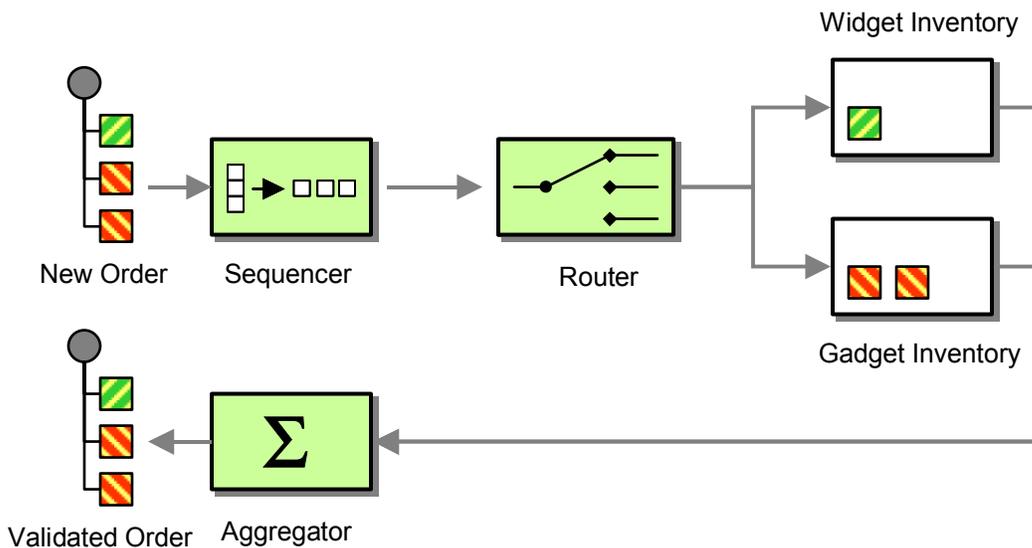


Figure 7: Distribution with Aggregate Response

The Distribution with Aggregate Response uses an Aggregator (13) to reconcile the requests that were dispatched to the multiple inventory systems. Each processing unit will send 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 (see Figure 7) as described under Aggregator (13).

Broadcast with Aggregate Response

Let's try to improve upon the order-processing example from the previous patterns. Assume that each order item that is not currently in stock could be supplied by one of multiple external suppliers. Suppliers may or may not have the item in stock themselves, they may charge a different price and may be able to supply the part by a different date. We should request quotes from all suppliers and see which one provides us with the best term for the requested item.

We need to find an approach that lets us send a message simultaneously to a set of recipients, and process responses from the individual recipients. The list of recipients should be dynamic in the sense that new recipients should be able to enter the 'bid' without changes to the underlying mechanisms. In addition, we must be prepared to receive responses from some, but not all recipients.

This problem is a variation of the Distribution with Aggregate Response (13) pattern. Instead of using a Sequencer (11), we broadcast the message to all involved parties using the Publish-Subscribe Message pattern (see [BrownWoolf]). We will most likely supply each recipient with a Reply Specifier (see [BrownWoolf]) so that all responses can be processed through a single channel. As with the Distribution pattern, responses are aggregated based on defined business rules (see Figure 8).

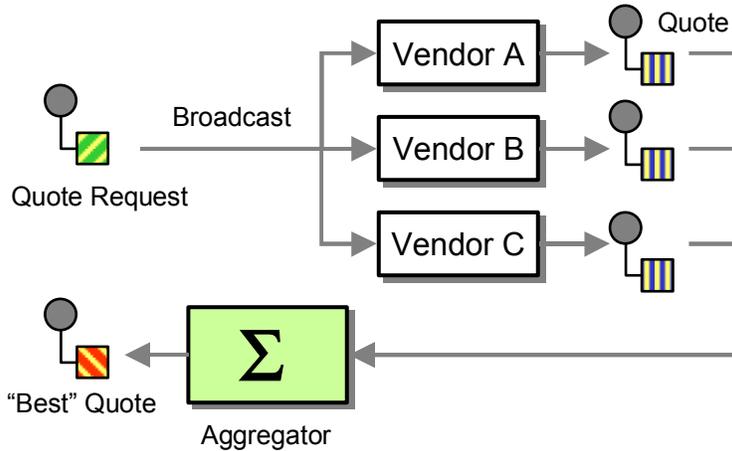
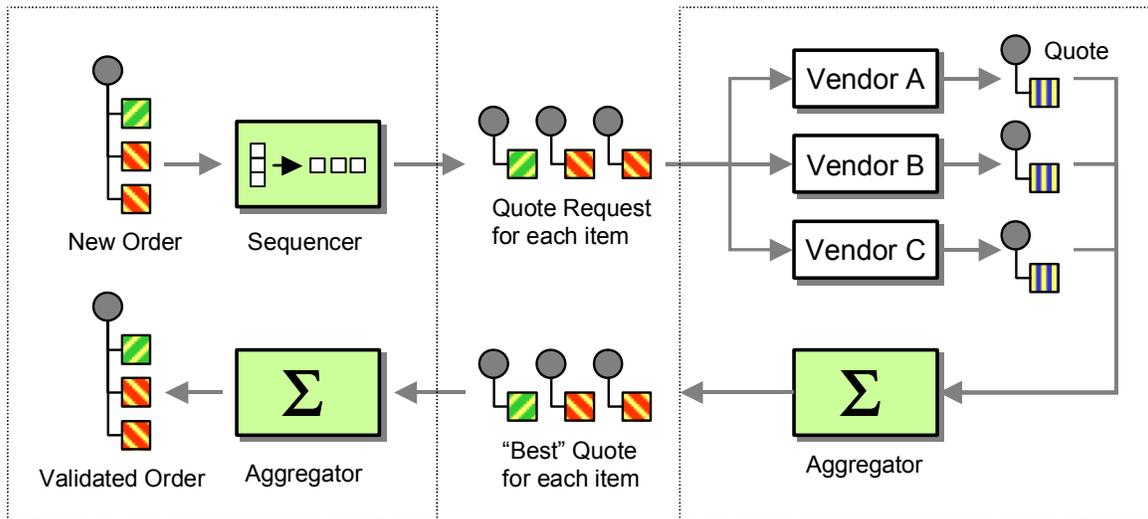


Figure 8: Broadcast with Aggregate Response

The aggregator generally consists of two parts:

1. Define which answers will be accepted (e.g. subjecting message to time-out conditions).
2. Determine which answer of the ones accepted is the “best” one. This is usually done by scoring each response and rank-ordering them.

Example



Distribution with Aggregate Response

Broadcast with Aggregate Response

Figure 9: Complex Quote Example

To follow through with our widget and gadget company, we could combine the aforementioned patterns 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, then aggregate all bid responses into a complete quote (see Figure 9). This is a very real example how multiple integration patterns can be combined into a complete solution.

Recipient List

The Content-Based Router (9) pattern 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 the router, which takes care of everything.

In some cases, though, the sender may want to specify more than one recipient for the message. A common example of this functionality is 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 ensures transport of the message content to each recipient. Another example would be a variation of the Broadcast with Aggregate Response (16) example. Rather than sending the request to all vendors, we may want to allow the user to select to which vendors to send the request, based on user preferences. In this case, we would offer the user a list of all vendors, and then allow the user to compile a recipient list.

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

Most messaging systems have built-in publish-subscribe capability (see [BrownWoolf]), which sends messages to a defined set of recipients. However, the set of recipients is determined by the number of subscribers to the specific channel or subject. While subscribers can come and go, the list of subscribers cannot change on a message-by-message basis.

We could consider having the originator add a list of intended recipients to the message. The message could then be 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. This approach is inefficient in that it requires each potential recipient to process a message. Also, depending on the messaging infrastructure, sending the message to all possible recipients may cause additional network traffic. This solution could be viable if on average a large subset of the recipients is chosen.

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. What happens if the originator process crashes after sending only a portion of the messages?

Therefore, use a *Recipient List* if the message originator specifies the list of recipients for the message.

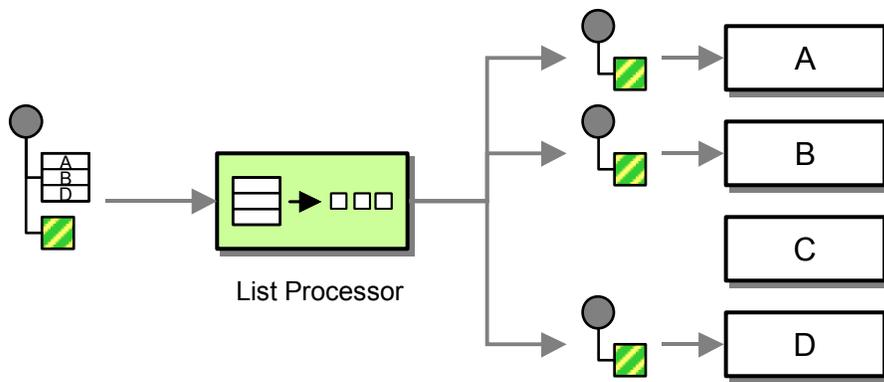


Figure 10: Recipient List

The originator includes a list of recipients with the message. Generally, the list would be included in the Message Header (27). The list processor traverses the list and sends an identical message to each recipient. In many cases, the list processor strips the recipient list from the message header to reduce the size of the outgoing messages.

The list processor component is responsible to send the incoming message to each recipient specified in the recipient list. The list processor needs to be able to store the incoming message and only 'consume' the incoming message after all outbound messages have been successfully sent. As such, the Recipient List component has to ensure that the complete operation is atomic.

Routing Table

The Pipes and Filters (8) pattern gives us an elegant solution to dividing multiple processing steps into separate Filters, connected by Pipes. This approach works well as long as each message that passes through the system should undergo the same processing steps. However, this is not always the case.

Assume a system where incoming requests have to undergo a sequence of business rule validations. Each of these validations can be seen as a filter. The filter will inspect the incoming message, apply the business rule and route the message to an exception channel if one of the conditions was not fulfilled. Each of these business rules may depend on external systems, so the Pipes and Filters patterns gives us a clean way to separate the processing steps.

Now let's assume, though, that the set of validations to perform against each message depends on the message type. We need to define a system that can route the message through different filters depending on the type of message that was received.

How do we route a message through a series of processing steps when the sequence of required steps is not known at design-time and may depend on the type of the incoming message?

We could include case statements in each filter that examine the message type and bypass the filter if this rule is not applicable to the specific message type. This approach would cause quite a bit of overhead because the message would be routed to each and every possible rule just to find out that many of the rules do not apply. It would also make the maintenance of the rule set difficult. If a new message type is introduced, code changes may have to be made to every filter.

We could also setup an individual Pipes and Filters (8) pattern for each applicable validation sequence. Then we could use a Content-Based Router (9) to route the message to the correct validation chain based on message type. This approach would make good use of existing

patterns, but requires us to hard-wire any possible combination of validation rules. For a large set of message types, this would certainly result in a maintenance nightmare.

Therefore, we define a *Routing Table* to specify the sequence of processing steps through which a message will be routed.

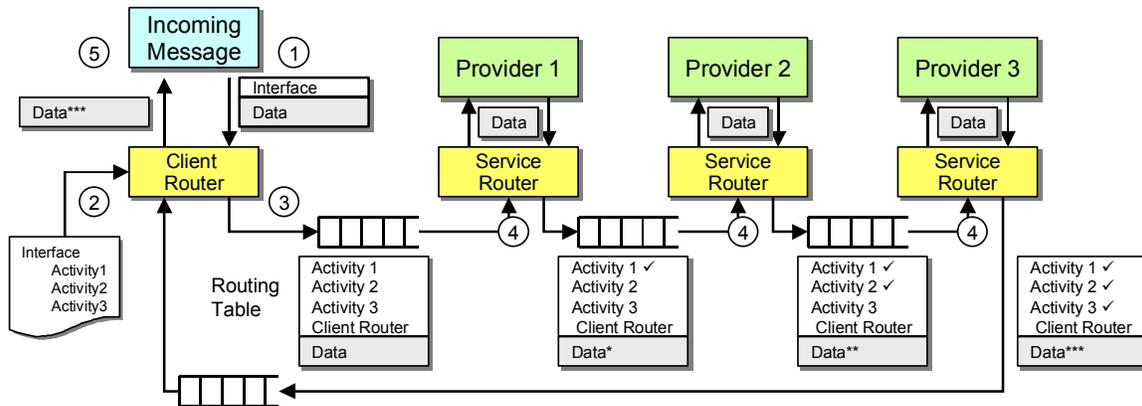


Figure 11: Routing Table

At the beginning of the process, we initialize the routing table based on a configuration document and route the message to the first processing step. After successful processing, each processing step will look at the routing table and send the message to the next processing step specified in the routing table (see Figure 11).

The pattern consists of two key elements:

The *Client Router* retrieves the routing table based in the message type (or any other property contained in the message body). The routing tables associated with each type can be retrieved from any external data store, e.g. an XML file, a database, a properties file or a routing service that is accessed via a channel.

A *Service Router* is chained in front of each service provider. The Service Router passes the message to the service provider for processing. Once the processing is complete, the service router looks up the next processing step in the routing table and sends the message to the channel specified in the routing table.

In the above example, the incoming message undergoes the following steps (see Figure 11):

1. The incoming message is sent to the router.
2. The router retrieves list required processing steps associated with the message type and places the routing table in the message header. The requestor itself is added to the list as the return address.
3. The client router publishes the message to the channel for the first activity.
4. The service 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 consumer receives the resulting message. If this is a one-way (asynchronous) message, the message may not have to be routed back to the originator but could be routed to the next logical processing step.

The Routing Table pattern is an example of a dynamically configured business process. In this case, the process is a simple, linear process, which is sufficient for many real-world scenarios. In

some cases, a more complex business process, including branching conditions, forks and joins has to be executed. In this case, the trade-offs between using a Routing Table and using a central process manager should be carefully examined. A dynamic Routing Table gives a central point of maintenance, but analyzing and debugging the system may become increasingly difficult as the execution of the business process is spread across many entities. 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. Some of the current efforts around the Web Services Flow Language ([WSFL]) may help address some of these concerns. If we do have to deal with complex business process flows, a Content-Based Router (9) combined with hard-wired business processes may be a vital alternative. Now we have to maintain multiple flows, but each individual flow is much easier to test and debug. As in many cases, the decision between these two approaches should be made on architectural drivers, such as required rate of change, required flexibility etc.

Message Transformation Patterns

Message Transformation Patterns work at the message content level. In most cases, enterprise integration solutions provide communication between existing systems. These systems could be packaged applications, custom-built applications, legacy systems, or external parties. Typically, each of these systems defines its own, proprietary data format or interface. As a result, an enterprise integration system needs the ability to transform messages from one data format to another. Data elements may have to be added to the message, elements may have to be taken away or existing data elements may have to be rearranged. Message transformation patterns describe reusable solutions to common data transformation problems in message-based enterprise integration systems.

Data Enricher

It is common in enterprise integration systems for a 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. 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, or has the items in a different format?

We need to use a *Data Enricher* to augment the incoming message with the missing information (Figure 12).

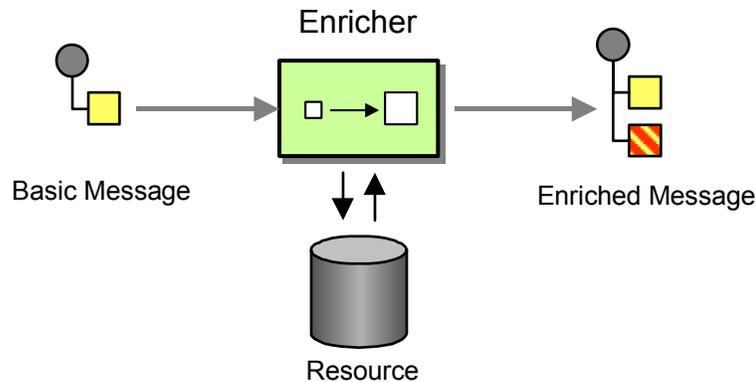


Figure 12: Data Enricher

In some cases, the missing element can be derived by using an algorithm (e.g. determine a state code based on the ZIP code). In most cases, the Data Enricher will need access to an external data resource to retrieve the missing information. This data resource could be implemented as a database, a file, an LDAP directory, a system, or a user who enters missing data. The resource may be local to the Data Enricher or may be situated on another system or even outside the enterprise. Accordingly, the communication between the Data Enricher and the resource can occur via message channels or via any other communication mechanism.

After the required data is retrieved from the resource, the original data element can be replaced with the new data or the new data elements can be appended to the message, preserving the original data element (as indicated in Figure 12).

The Data Enricher pattern 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 Data 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 enricher to restore some of the original message content.

Store in Library

Our 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?

The *Store in Library* pattern extracts data items from an incoming message, stores them in a persistent store or “library” and replaces the data elements with a reference to the item just inserted into the library (see Figure 13).

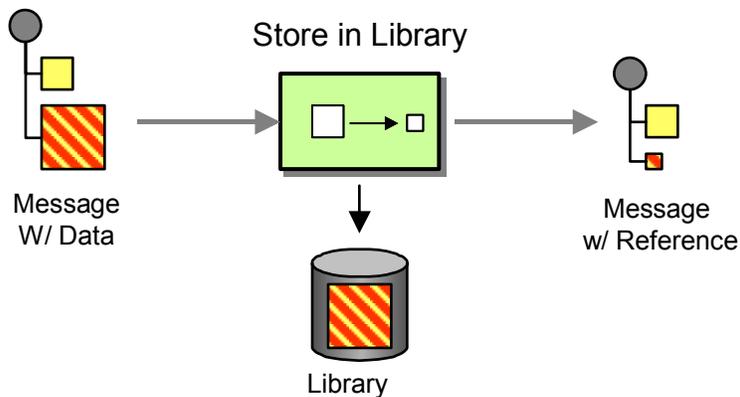


Figure 13: Store in Library

A Data Enricher (21) pattern can later be used to re-insert the information into the message. The reference can be established via a unique key that may already be contained in the incoming message. If there is no suitable key available, a unique key can be generated to associate the reference with the data stored in the library.

The implementation of a library 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 library just as well. It is important that the library is reachable by message transformations in other parts of the integration solution so that these parts can reconstruct the original message.

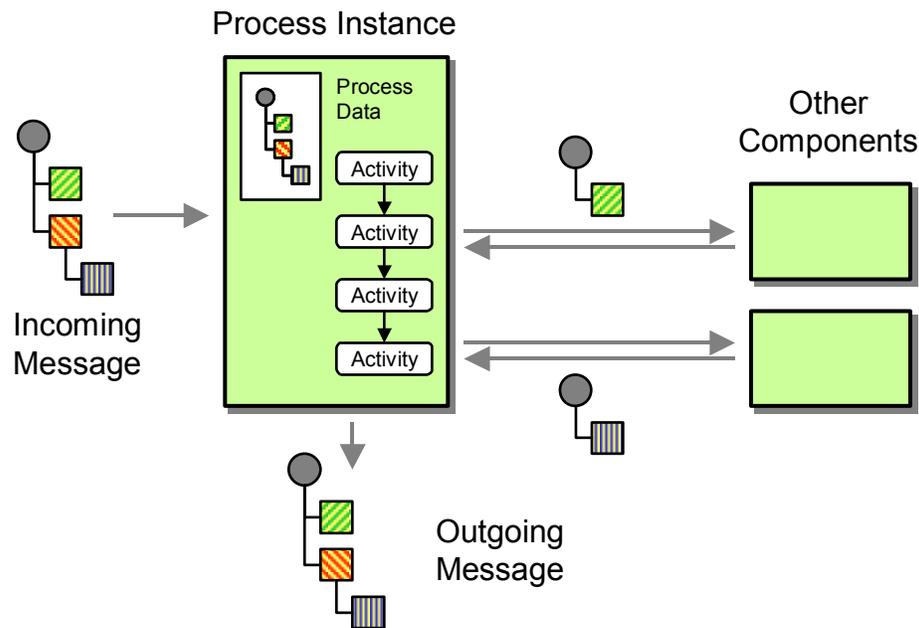


Figure 14: Process Instance as Library

A special case of the Store in Library pattern can be found in business process management tools. These tools create persistent process instances (sometimes called ‘tasks’ or ‘jobs’) when a message arrives. Most systems allow the developer to associate data items with the process instance. In effect, the process engine now serves as the library. As the process progresses, messages may be sent to other components of the messaging systems. These messages do not

have to carry all the information contained in the original message since that information is kept with the process itself (see Figure 14).

Content Filter

The Data Enricher (21) pattern helps us solve situations where a message receiver requires more – or different – data elements than the message creator can provide. There are surprisingly many situations where the opposite effect is desired: removing data elements from a message.

What do we do if we have to deal with a very complicated message, but 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.

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 standard message formats. 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. These documents are difficult to work with for internal message exchange. 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 element enriches the usefulness of such a message, because redundant and irrelevant fields are removed, leaving a more meaningful message.

Therefore, use the *Content Filter* pattern to eliminate data elements from a message..

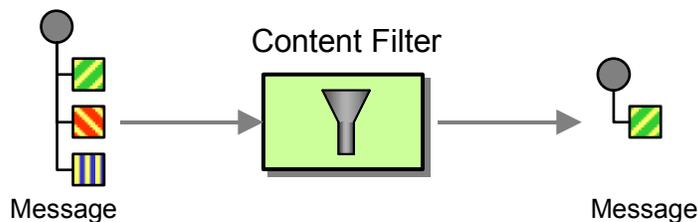


Figure 15: Content Filter

Content Filter is different from Store in Library (22) in that it really discards unneeded elements whereas the Store in Library stores them for later use. Do we really want to discard data elements for good? In many cases, an incoming comprehensive message is routed to multiple components or sub-processes. Each component may only be interested in specific aspects of the original message. Therefore, it uses a Content Filter to get rid of extra baggage. In aggregate, though, all data elements may be used by some sub-process.

The Content Filter does not necessarily just discard data elements. The content filter may also simplify the structure of the message. Typically, messages are represented as tree structures. Many messages originating from external systems or packaged applications contain many levels of nested, repeating groups. Frequently, constraints and assumptions make this level of nesting

superfluous and Content Filters ‘flatten’ the hierarchy into a simple list of elements than can be easily processed by other systems (see Figure 16).

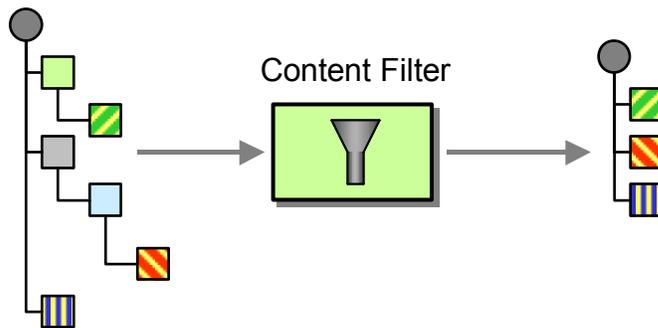


Figure 16: Flattening a Message Hierarchy

Message Management Patterns

A message-based enterprise integration system can process thousands or even millions of messages in a day. Messages are generated, routed, transformed and consumed. Error conditions can occur at any stage in this chain. Performance bottlenecks may occur. Message Management Patterns tackle the problems of keeping a complex message-based system running.

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 administer a system that is distributed across multiple platforms and a wide geographic area? How can we know whether all components are up and running? How do we identify bottlenecks? How do we change configurations remotely?

We could use techniques widely used for non-distributed systems such as property files and error logs. We could then process the error logs and send e-mail messages when something goes wrong. This approach works well if there is a single system, or a small number of systems. Property files will be hard to distribute and maintain across multiple machines. We may not want to give access to the file system on every machine just to allow property file distribution.

We could use other networking protocols, such as http to create a management infrastructure. This will address the problem, but now we are using two different technologies inside the messaging system: the message flow occurs over addressable conduits and pipes, while configuration information is handled through another protocol (e.g., http).

Therefore, use a *Control Bus* pattern to manage an enterprise integration system (Figure 17). 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.

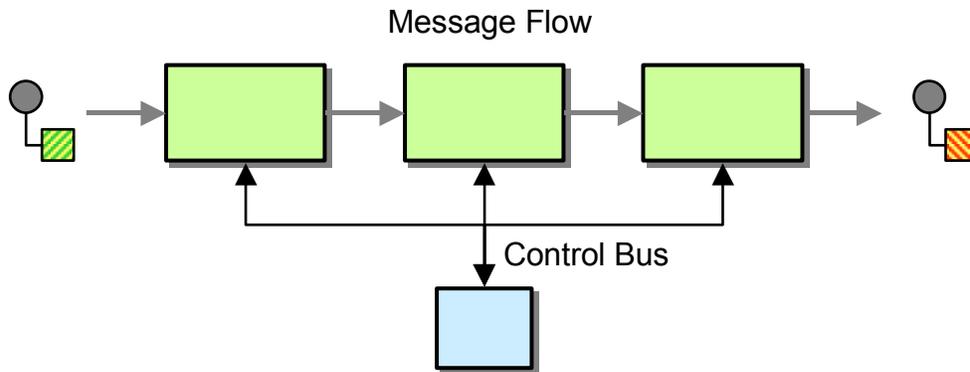


Figure 17: Control Bus

Typical uses of a control bus include:

- **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 the reconfiguration of the integration solution at run-time. For example, the routing table inside a Content-Based Router (9) may need to be updated dynamically based on system conditions, such as overload or component failure.
- **Heartbeat** – each component may send a ‘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 (32)).
- **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.
- **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 control bus functions resemble typical network management functions, based on underlying network protocols. A control bus allows us to implement equivalent management functions at the messaging system level -- essentially elevating them from the ‘dumb’ network

level to the richer messaging level. This concept has been labeled ‘Enterprise Nervous System’ and has caught the attention of enterprises implementing message-based integration solutions.

When we design message processing components, we architect the core processor around three interfaces (see Figure 18). 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.

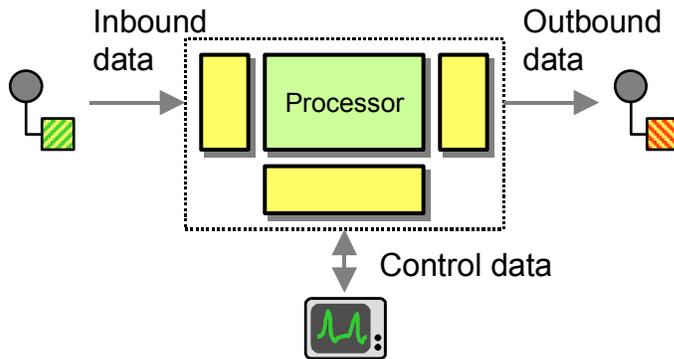


Figure 18: Message Processing Component Architecture

Message Header

As described in the Control Bus (25) pattern, it makes sense to separate the flow of control information from the flow of messages. In some cases, though we might want to transmit some control information with a message. We may want to be able to trace a message using a unique ID. We may want to know where the message has been and how long it took the message to get here. Messages may “expire” at a certain time and may be discarded because the information contained in them is no longer up-to-date.

How do we best transmit control information along with a message?

Obviously, we can just add fields to the message definition. However, this would become problematic as we start to blur application data and control data. What if we need to add an additional field, e.g. ‘Sent time’ to the message? Would we have to change all message definitions? What if the control information can be understood only by some components and not by others? The other components may consider this unexpected information erroneous and flag an exception.

Therefore, we should use a *Message Header* pattern to separate message content from additional control information.

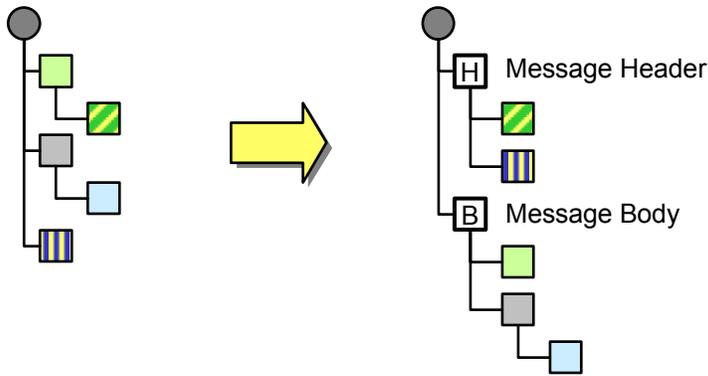


Figure 19: Message Header

The message header is a separate section of the message that contains data related to the messaging infrastructure as opposed to data that is going to be processed by the participating systems.

In many occasions, the message header is closely tied to the use of the Control Bus. The Control Bus may collect statistical information based on the data that is contained in the message header. For example, the control bus could collect message run-time information based on unique message IDs contained in the message header of each message.

The Message Header pattern also supports a number of message patterns described in [BrownWoolf], such as Reply Specifier, Correlation Identifier, Sequence Identifier, or Message Expiration.

Envelope Wrapper / Unwrapper

As we concluded in the Message Header (27) pattern, message headers perform a number of very useful functions. 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 specified by the application. On the other hand, messaging components may require the header fields and would consider a message invalid if it does not contain the proper header fields.

Assume our messaging system is using a security scheme. A valid message would have to contain security credentials for the message to be accepted for processing by other messaging components. This would prevent unauthorized users from feeding messages into the system. Additionally, the message content may be encrypted to prevent eavesdropping by unauthorized listeners. This is a particularly important issue with publish-subscribe mechanisms. However, a typical integration solution integrates existing applications that do not really know too much about the concepts of user identity or message encryption. As a result, we need to translate 'raw' messages into messages that comply with the rules of the messaging system.

How do we define a mechanism that lets existing systems participate in a messaging exchange that places specific requirements on the message format, such as message header fields or encryption?

We employ an *Envelope Wrapper / Unwrapper* pattern to transform messages into a format that is required by the messaging infrastructure and back to the original message format (see Figure 20).

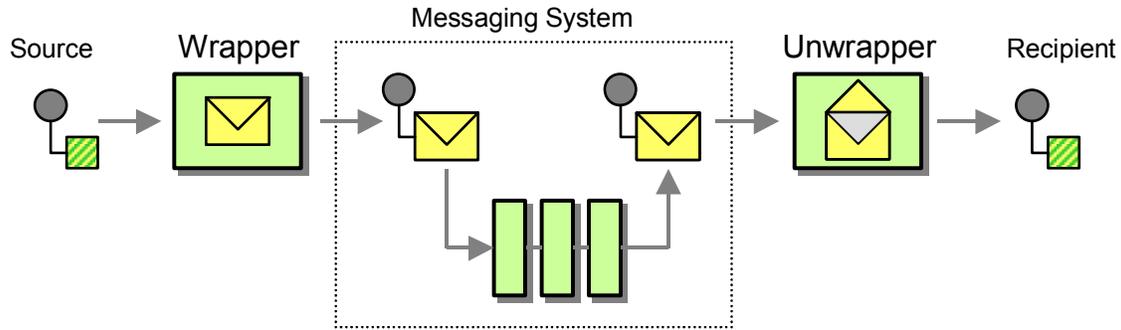


Figure 20: Message Wrapper and Unwrapper

The Envelope Wrapper / Unwrapper consists of five key parts:

- 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.

The Message Wrapper / Unwrapper pattern can be compared to the postal system (see Figure 21). 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.

The postal system example illustrates the very common practice of chaining wrappers and unwrappers using the Pipes and Filters (8) 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 flexibility.

The approach used in the postal example is also referred as “tunneling”. Tunneling describes the technique of wrapping messages that are based on one protocol into another protocol so that they can be transported over new communication channels. Postal mail may be “tunneled” through air freight just like UDP multicast packets may be tunneled over a TCP/IP connection in order to reach a different WAN segment.

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. What distinguishes wrappers is that they 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), it can be extracted from the infrastructure (e.g. retrieval of a security context) or it may be contained in the raw message and split out into the message header (e.g. a timestamp contained in the ‘raw’ message).

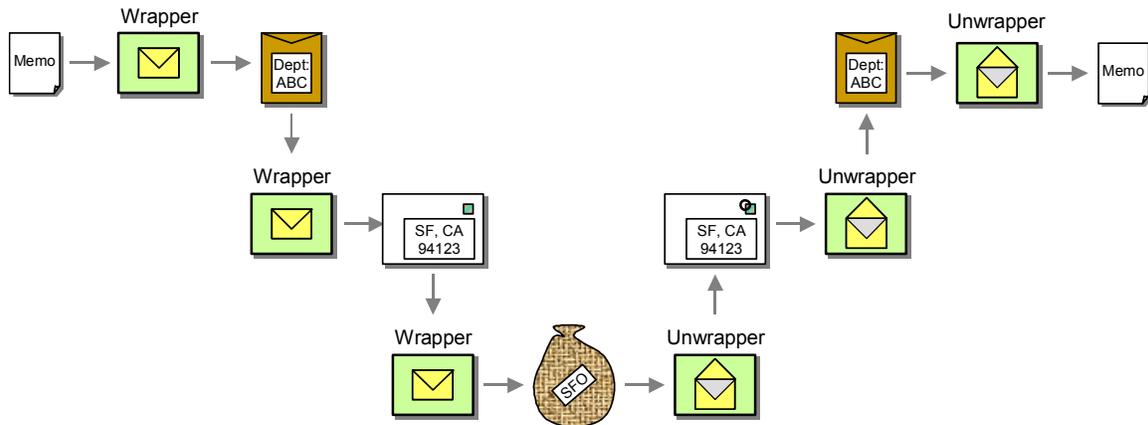


Figure 21: Chained Wrapper/ Unwrapper Example

Message History

One of the fundamental properties of message-based systems is the fact that they are loosely coupled. This means that the message sender and recipient make few assumptions about each other’s identity. If a message recipient retrieves a message from a message queue or a channel, it really does not care who put the message in the queue. 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.

Assume we implemented a publish-subscribe system that propagates address changes to multiple systems. Each address change is broadcast to all 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 and no changes to the existing messaging system are required. Assume the customer care system stores addresses in 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 will result in a database update, which will in turn trigger another ‘address changed’ message. We end up in an infinite loop of ‘address changed’ messages.

How can we create a system that is loosely coupled, but still gives participant the ability to track the origin of a message so that infinite loops and deadlocks can be avoided?

Each system could assign a unique ID to each message and keep a log of all IDs for the messages which ones originated from it. When a message is received, the system could sift through the list of published messages and discard any incoming message that originated from the system itself.

This would assume we have some persistent storage for all addresses that we send. While part of this can be provided by the Message Store (31) pattern, this would mean a significant overhead for each system that wants to participate in the address updates.

We could try to use different channels for each system's updates. The system would listen to all other update channels except its own. This would solve the infinite-loop problem. However, it will lead to a proliferation of channels and eliminate some of the advantages of the original solution. Now every system has to be aware of the existence of all other systems in order to subscribe to their updates. When a new system is added, all existing systems are affected. This is what we tried to avoid in the first place.

We could also consider comparing the incoming message data to the data in the database and not update if the data is identical to what we have in the database record already. This would eliminate the infinite loop but it requires us to make two database accesses for each address update.

Therefore, use a *Message History* pattern when the origin of a message has to be known to avoid deadlock or infinite-loop situations (see Figure 22).

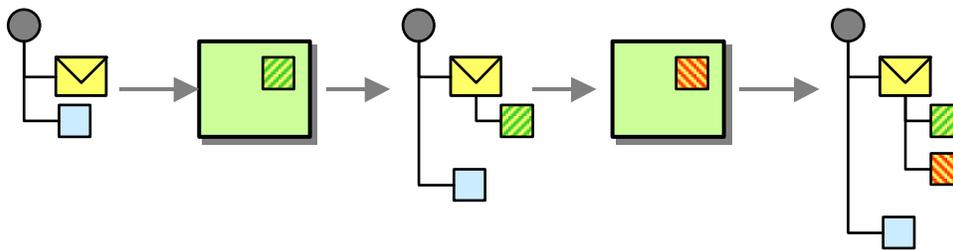


Figure 22: Message History

The message history becomes part of the message header, so that it is logically separated from the message body. Every component that processes the message (including the originator) adds an entry to the history. Thus, the messaging infrastructure can determine the history of a message passing through it while the message origin is transparent to the participating applications.

The message history pattern is also a very useful debugging tool. Any message that travels through the system carries with it a list of all the components through which it passed.

Message Store

The Message History (30) pattern illustrates the usefulness of being able to tell where a message has been. 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, which disappear once they have been consumed.

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

Therefore, use a message store to capture information about each message in a central location.

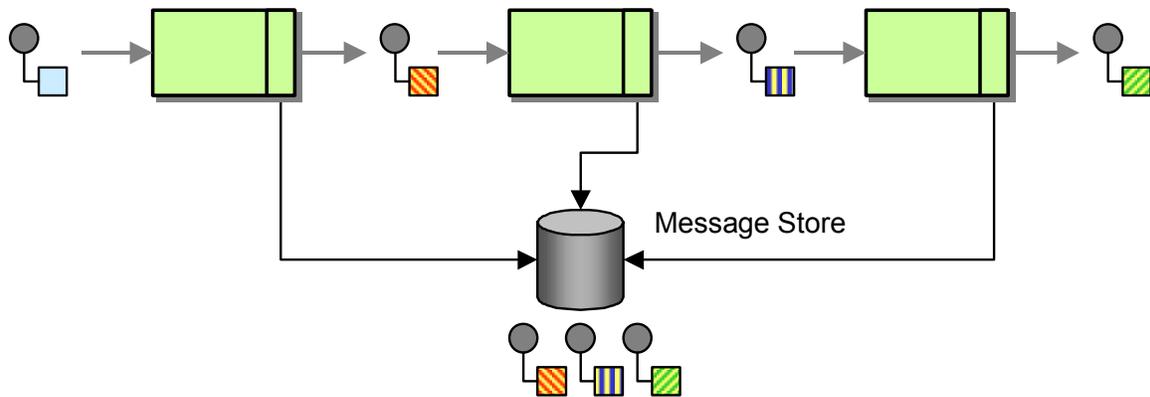


Figure 23: Message Store

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. We can consider this channel part of the Control Bus (25). Sending this message in a ‘fire-and-forget’ mode will not slow down the flow of our main messages. We do need to keep in mind network traffic, though. That’s why we may not store the complete message, but just a few key fields, 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 (see Message Header (27)), 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.

In some cases, enterprise integration tools may supply some aspects of a message store. Each message that passes through a channel can be logged and reported against.

Test Message

The Control Bus (25) 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 Messages* to assure the health of message processing components (see Figure 24).

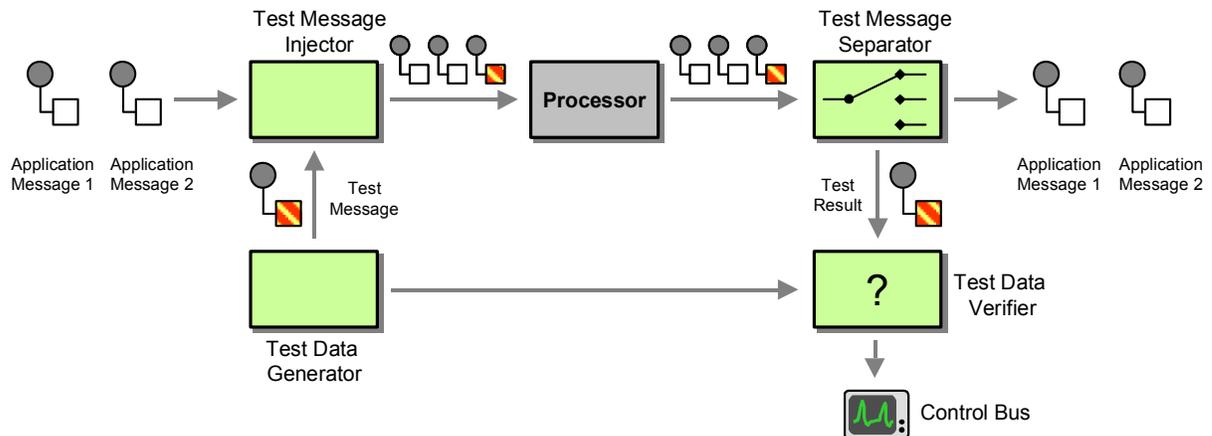


Figure 24: Test Message

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 (9).
- 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.

Test Messages are considered active monitoring mechanisms. Unlike passive mechanisms, they 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.

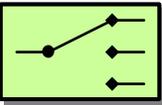
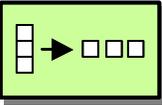
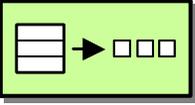
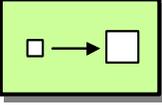
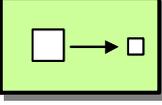
Active monitoring may also incur cost if we are being charged for the use of a component on a pay-per-use basis. This is the case for many external components, e.g. if we request credit reports for our customers from an external credit scoring agency.

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!

Visual Pattern Language

Individual patterns come to life when they are used in the context of a pattern language. Many integration designs start out as diagrams on a whiteboard, a piece of paper or the proverbial napkin. Therefore, it makes sense to provide a visual vocabulary for the pattern language. This page summarizes all pattern symbols defined in this paper.

Note that not all patterns are represented by a single symbol (e.g. Distribution with Aggregate Response). Only those patterns that are represented by a single symbol are shown in this list.

Symbol	Pattern	Uses
	Router	Use a Content-Based Router to route messages to the correct recipient based on message content.
	Sequencer	Use a Sequencer to break out the composite message into individual messages, each containing data related to one item.
	Aggregator	Use the Aggregator pattern to manage the reconciliation of multiple, related messages into a single message.
	Recipient List	Use a Recipient List if the message originator specifies the list of recipients for the message.
	Data Enricher	Use a Data Enricher to augment the incoming message with the missing information.
	Store in Library	Use the Store in Library pattern to store message data for later retrieval and thus reduce message size.
	Content Filter	Use the Content Filter to eliminate unnecessary data elements from a message.
	Envelope Wrapper	Use the Envelope Wrapper pattern to transform messages into a format that is required by the messaging infrastructure.
	Envelope Unwrapper	Use the Envelope Unwrapper pattern to transform messages from the format required by the messaging infrastructure back to the original message format.

Conclusions

Integration has become a critical part of any enterprise application development effort. Integration solutions are complex systems that span across many different technologies and levels of abstraction. Enterprise Integration Patterns can be of great assistance to integration architects in defining a flexible solution.

The field of integration patterns is still relatively new. This paper focuses only on a small set of patterns. As we explore the field of message-based integration further, we expect to discover many more patterns and pattern categories.

Bibliography

- [Alexander] C. Alexander, S. Ishikawa, M Silverstein, *A Pattern Language – Towns·Buildings·Construction*, Oxford University Press, 1977
- [BrownWoolf] Kyle Brown and Bobby Woolf, *Patterns of System Integration with Enterprise Messaging*, PLOP 2002
- [POSA] Frank Buschmann et al, *Pattern-Oriented Software Architecture*, Wiley 1996
- [PatternForms] *Pattern Forms*, Wiki-Wiki-Web, <http://c2.com/cgi-bin/wiki?PatternForms>
- [UML] *Unified Modeling language Specification (UML), Version 1.4*, Object Management Group.
<http://www.omg.org/technology/documents/formal/uml.htm>
- [UMLEAI] *UML Profile for Enterprise Application Integration*, Object Management Group 2002. <http://cgi.omg.org/cgi-bin/doc?ptc/02-02-02>
- [WSFL] IBM, *Web Services Flow Language*.
<http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>