INTRODUCTION TO MESSAGE ROUTING

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

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

Simple Routers

The *Content-Based Router* inspects the content of a message and routes it to another channel based on the content of the message.

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

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

A *Content-Based Router* and a *Message Filter* can actually solve a similar problem. A *Content-Based Router* routes a message to the correct destination based on the criteria encoded in the *Content-Based Router*. Equivalent behavior can be achieved by using a *Publish-Subscribe Channel* and an array of *Message Filter*s, one for each potential recipient. Each *Message Filter* eliminates the messages that to not match the criteria for the specific destination.

The *Content-Based Router* routes predictively to a single channel and therefore has total control, but is also dependent on the list of all possible destination channels. The *Message Filter* array filters reactively, spreading the routing logic across many *Message Filter*s but avoiding a single component that is dependent on all possible destinations. The trade-off between these solutions is described in more detail in the *Message Filter* pattern.

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

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

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

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

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

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

Pattern	**Number of Msgs Consumed**	**Number of Msgs Published**	**Stateful?**	**Comment**
Content-Based Router	1	1	No (mostly)	
Message Filter	1	0 or 1	No (mostly)	
Recipient List	1	multiple (incl. 0)	No	
Splitter	1	multiple	No	
Aggregator	multiple	1	Yes	
Resequencer	multiple	multiple	Yes	Publishes same number it consumes.

Composed Routers

A key advantage of the *Pipes and Filters* architecture is the fact that we can compose multiple filters into a larger solution. *Composed Message Processor* or an *Scatter-Gather* combine multiple *Message Router* variants to create more comprehensive solutions. Both patterns allow us to retrieve information from multiple sources and recombine it into a single message.

While the *Composed Message Processor* maintains control over the possible sources the *Scatter-Gather* uses a *Publish-Subscribe Channel* so that any interested component can participate in the process.

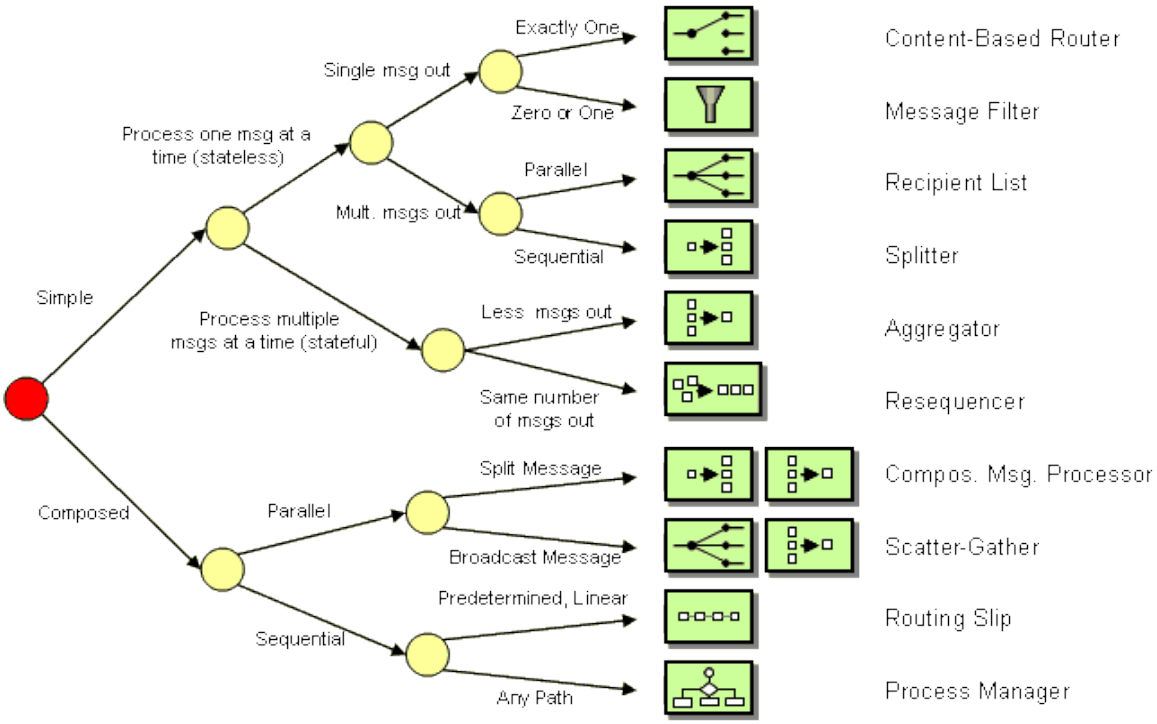
Both the *Composed Message Processor* and the *Scatter-Gather* route a single message to a number of participants concurrently and reassemble the replies into a single message. We can say that these patterns manage the *parallel routing* of a message. Two more patterns manage the *sequential routing*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

Architectural Patterns

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

The Right Router for the Right Purpose

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



Source:
http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRoutingIntro.html