Technische Hochschule Ingolstadt

Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

IT-Integrations- und Migrationstechnologien

Enterprise Integration Patterns

Prof. Dr. Bernd Hafenrichter 01.10.2014

Motivation

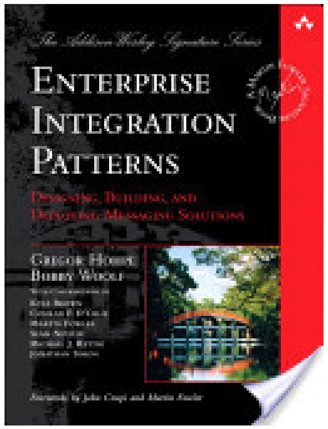
- Die Implementierung einer Integrationslösung erfordert den bewussten Umgang mit verschiedensten Technischen Problemstellungen.
- Bespiele hierfür sind:
  - Handhabung Kopplung (Zeit, Wissen, Vertrauen)
  - Realisierung der technischen Infrastruktur
- Ein Ansatz für die Umsetzung basiert auf der Idee des Messagings.
  - Grundlegendes Element sind Nachrichten die zwischen Systemen ausgetauscht werden.

Motivation

- Die folgenden Ausführungen beruhen auf folgendem Buch:

Enterprise Integrations Patterns

Designing, Building, and Deploying Messaging Solutions
Addison Wesley



Messaging Systems – Basic Elements

Basic Messaging Concepts
Like most technologies, Messaging involves certain basic concepts. Once you understand these concepts, you can make sense of the technology even before you understand all of the details about how to use it. These basic messaging concepts are:

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

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

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

Messaging Systems – Basic Elements

Basic Messaging Concepts
Like most technologies, *Messaging* involves certain basic concepts. Once you understand these concepts, you can make sense of the technology even before you understand all of the details about how to use it. These basic messaging concepts are:

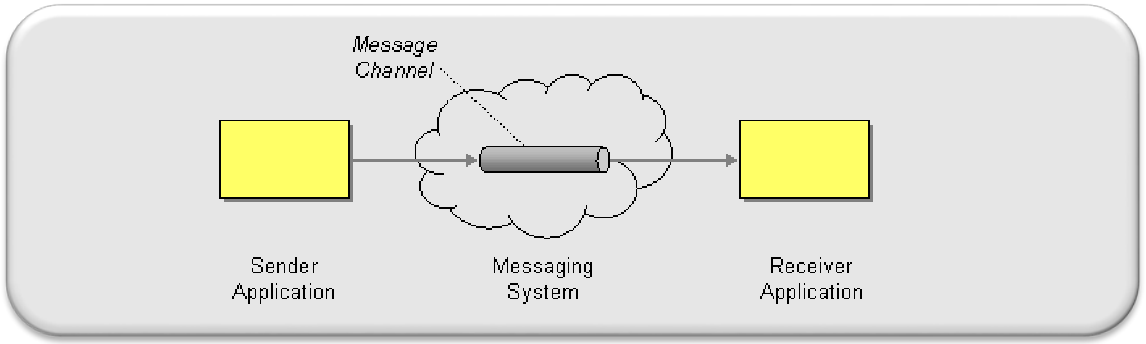
Routing — In a large enterprise with numerous applications and channels to connect them, a message may have to go through several channels to reach its final destination. The route a message must follow may be so complex that the original sender does not know what channel will get the message to the final receiver. Instead, the original sender sends the message to a *Message Router*, an application component and filter in the pipes-and-filters architecture, which will determine how to navigate the channel topology and direct the message to the final receiver, or at least to the next router.

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

Endpoints — An application does not have some built-in capability to interface with a messaging system. Rather, it must contain a layer of code that knows both how the application works and how the messaging system works, bridging the two so that they work together. This bridge code is a set of coordinated *Message Endpoints* that enable the application to send and receive messages.

Messaging Systems – Basic Elements

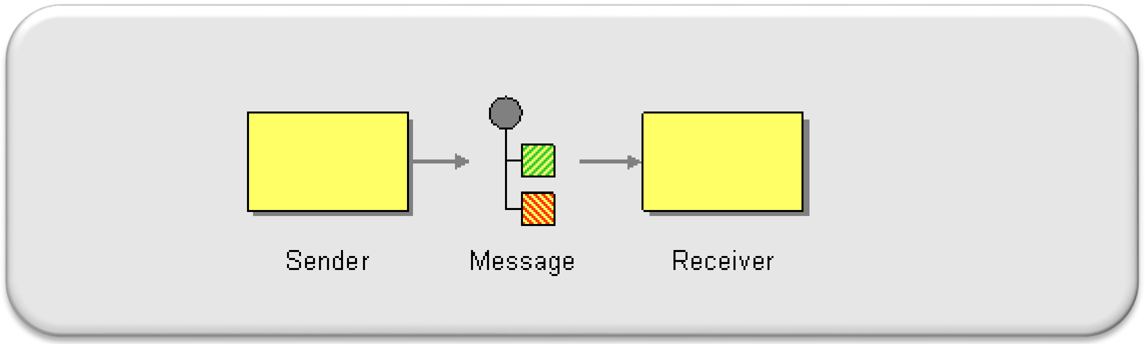
***Message Channel**:* How does one application communicate with another using messaging?



Connect the applications using a *Message Channel*, where one application writes information to the channel and the other one reads that information from the channel.
When an application has information to communicate, it doesn't just fling the information into the messaging system, it adds the information to a particular *Message Channel*. An application receiving information doesn't just pick it up at random from the messaging system; it retrieves the information from a particular *Message Channel*.

Messaging Systems – Basic Elements

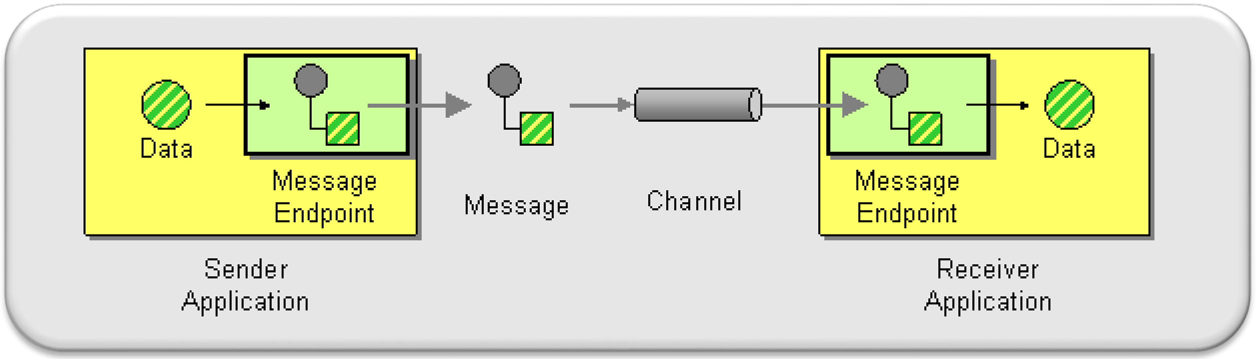
Message: How can two applications connected by a message channel exchange a piece of information?



Package the information into a *Message*, a data record that the messaging system can transmit through a message channel.
Thus any data that is to be transmitted via a messaging system must be converted into one or more messages that can be sent through messaging channels. A message consists of a message header and a message body. The body contains the payload. The header contains meta information about the message.

Messaging Systems – Basic Elements

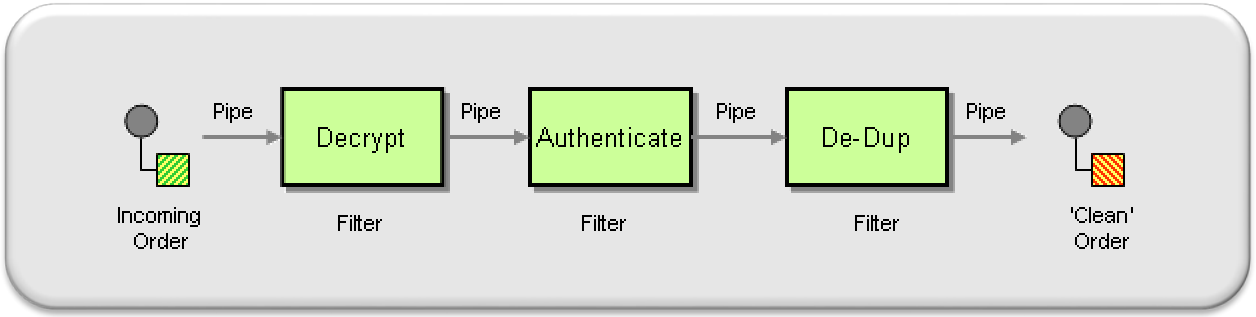
Message Endpoint: How does an application connect to a messaging channel to send and receive messages?



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

Messaging Systems

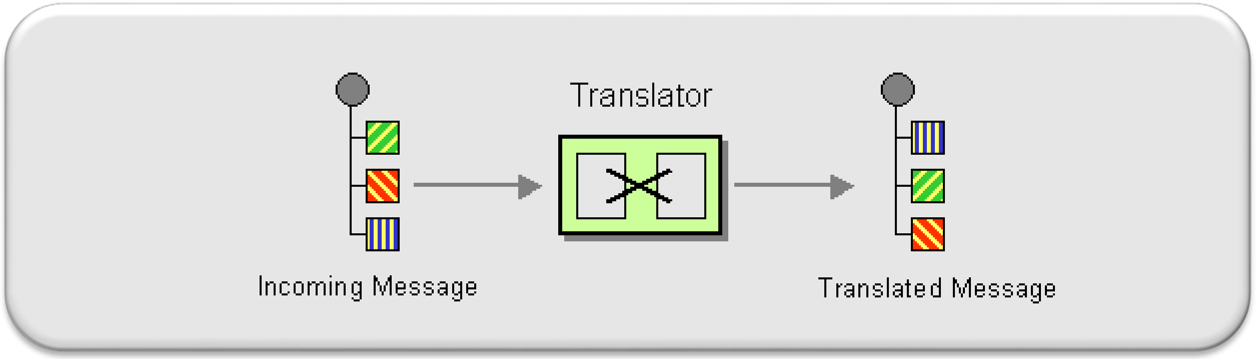
***Pipes and Filters*: How can we perform complex processing on a message while maintaining independence and flexibility?**



Use the *Pipes and Filters* architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes).
Each filter exposes a very simple interface: it receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe. The pipe connects one filter to the next, sending output messages from one filter to the next. Because all component use the same external interface they can be *composed* into different solutions by connecting the components to different pipes. We can add new filters, omit existing ones or rearrange them into a new sequence -- all without having to change the filters themselves. The connection between filter and pipe is sometimes called *port*. In the basic form, each filter component has one input port and one output port.

Messaging Systems

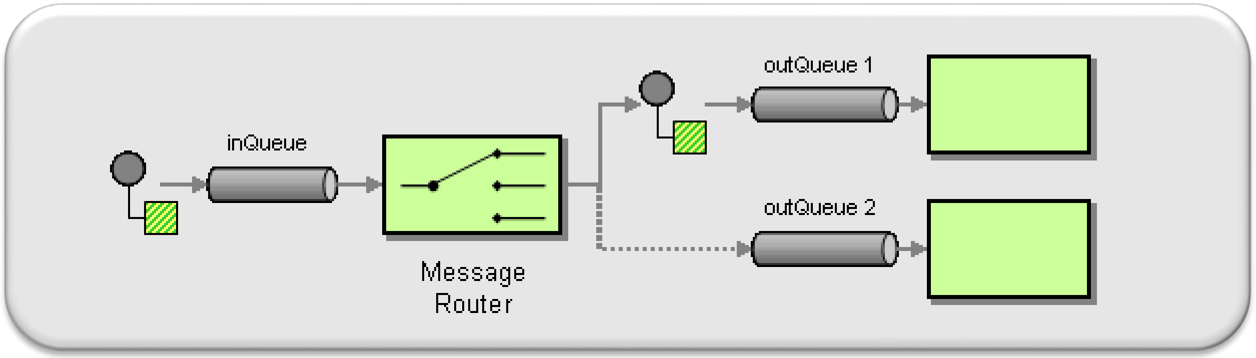
Message Translator: How can systems using different data formats communicate with each other using messaging?



Use a special filter, a *Message Translator*, between other filters or applications to translate one data format into another.
The *Message Translator* is the messaging equivalent of the *Adapter* pattern described in [GoF]. An adapter converts the interface of a component into a another interface so it can be used in a different context.

Messaging Routing

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

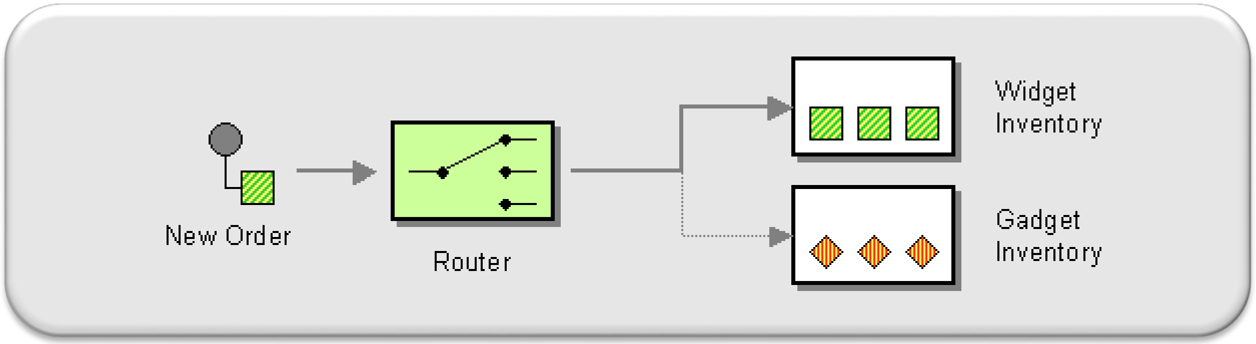


Insert a special filter, a *Message Router*, which consumes a *Message* from one *Message Channel* and republishes it to a different *Message Channel* channel depending on a set of conditions.

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

Messaging Routing

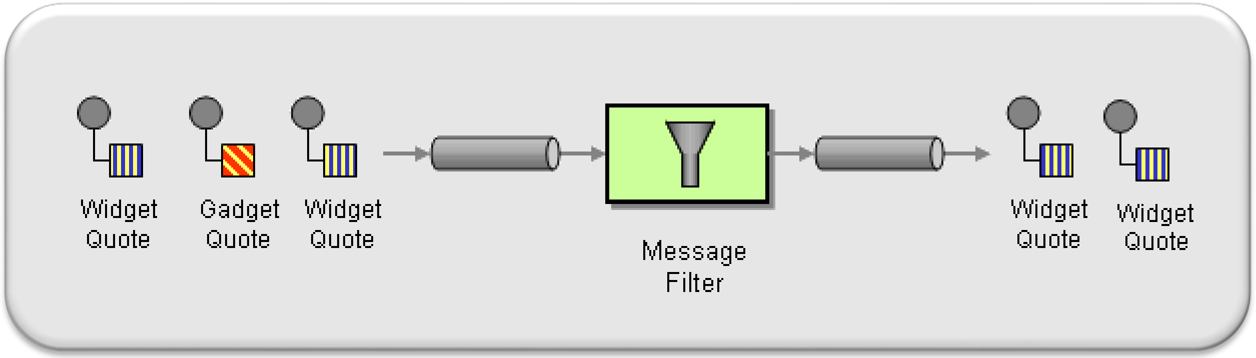
Content-Based Router: How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?



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

Messaging Routing

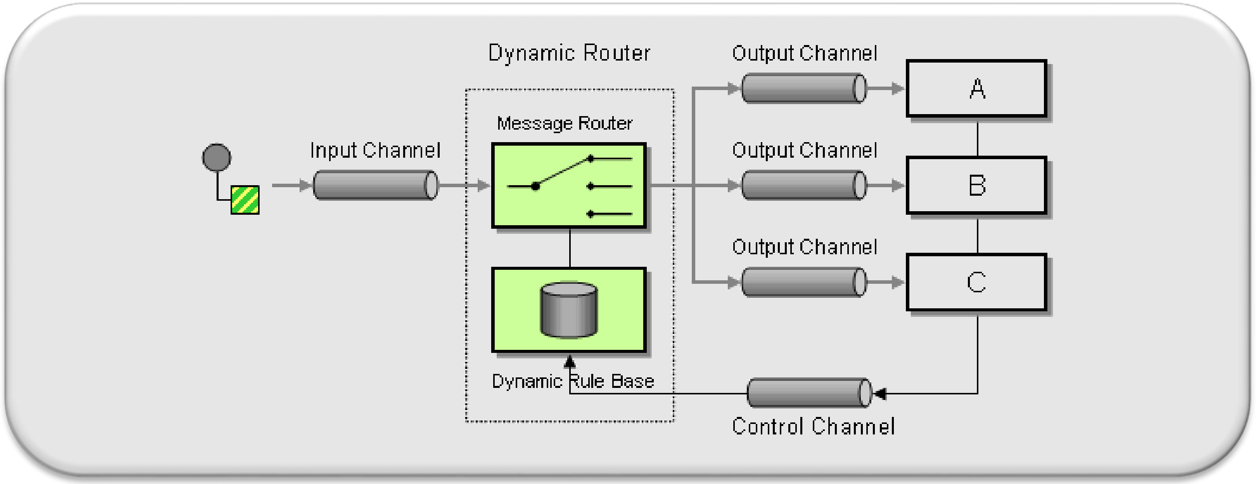
Message Filter: How can a component avoid receiving uninteresting messages?



Use a special kind of Message Router, a *Message Filter*, to eliminate undesired messages from a channel based on a set of criteria.
The *Message Filter* has only a single output channel. If the message content matches the criteria specified by the *Message Filter*, the message is routed to the output channel. If the message content does not match the criteria, the message is discarded.

Messaging Routing

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

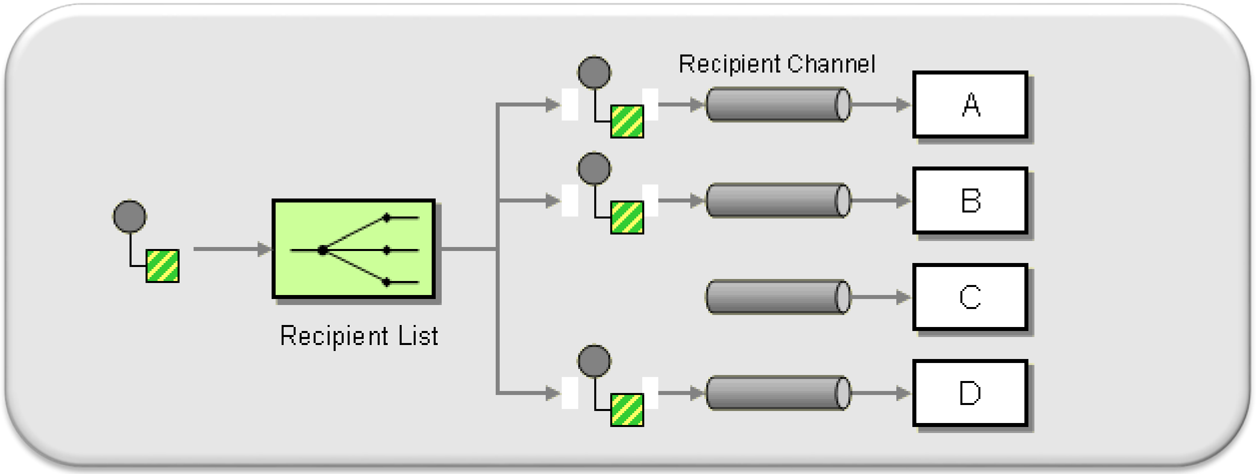


Use a *Dynamic Router*, a Router that can self-configure based on special configuration messages from participating destinations.

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

Messaging Routing

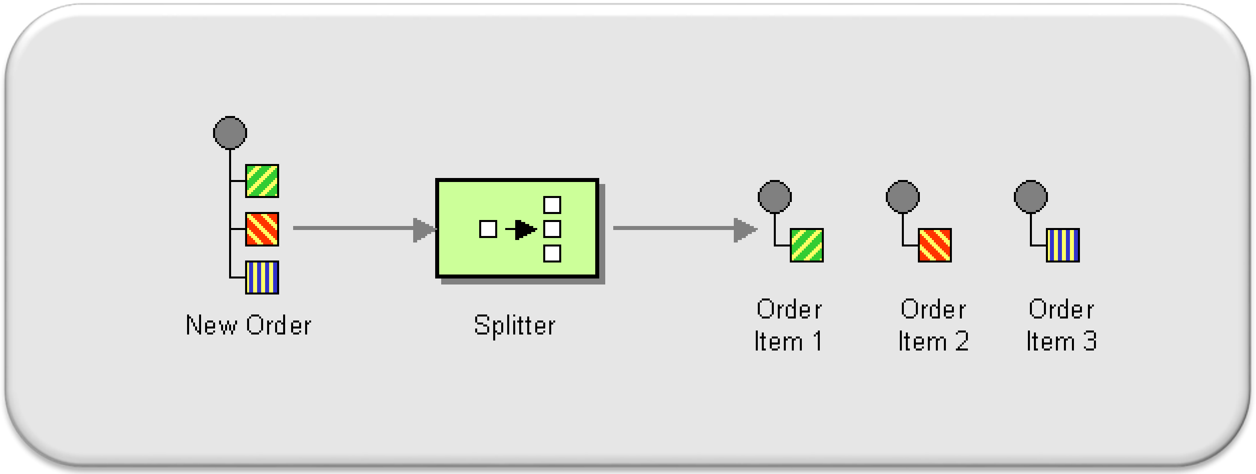
Recipient List **:** How do we route a message to a list of dynamically specified recipients?



Define a channel for each recipient. Then use a *Recipient List* to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list.
The logic embedded in a *Recipient List* can be pictured as two separate parts even though the implementation is often coupled together. The first part computes a list of recipients. The second part simply traverses the list and sends a copy of the received message to each recipient. Just like a *Content-Based Router*, the *Recipient List* usually does not modify the message contents.

Messaging Routing

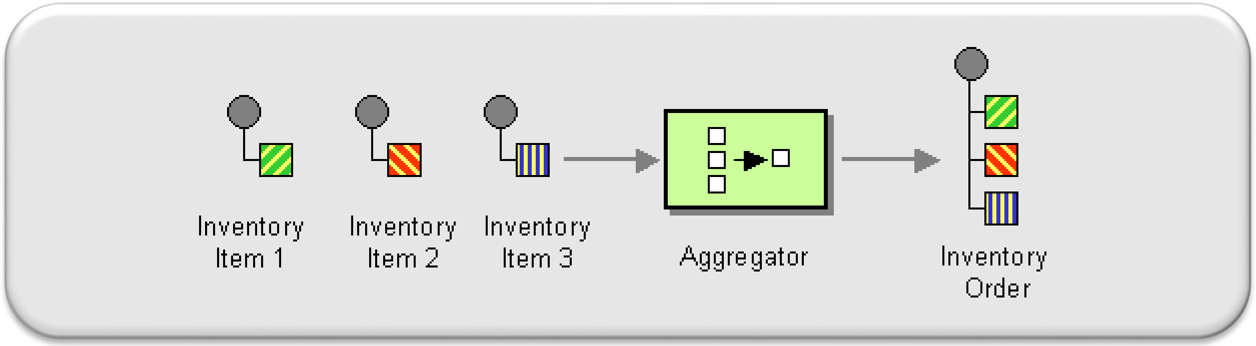
Splitter: How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?



Use a *Splitter* to break out the composite message into a series of individual messages, each containing data related to one item.
use a *Splitter* that consumes one message containing a list of repeating elements, each of which can be processed individually. The *Splitter* publishes a one message for each single element (or a subset of elements) from the original message.

Messaging Routing

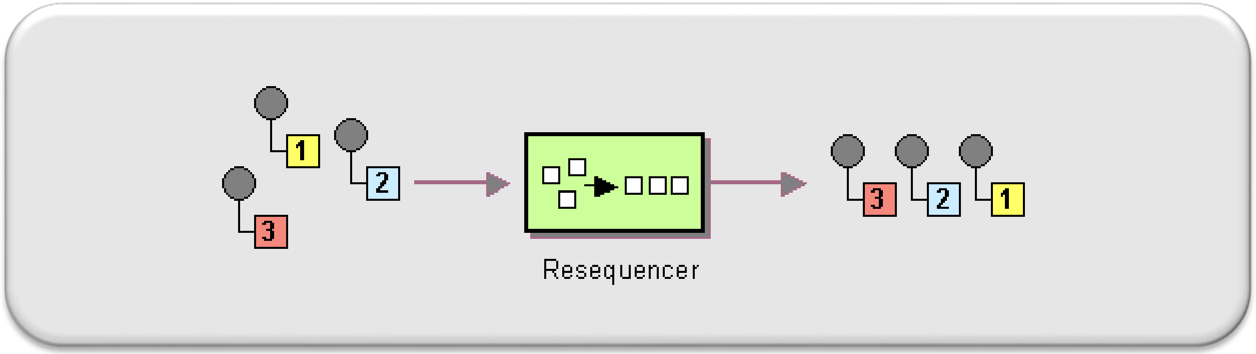
Aggregator: How do we combine the results of individual, but related messages so that they can be processed as a whole?



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

Messaging Routing

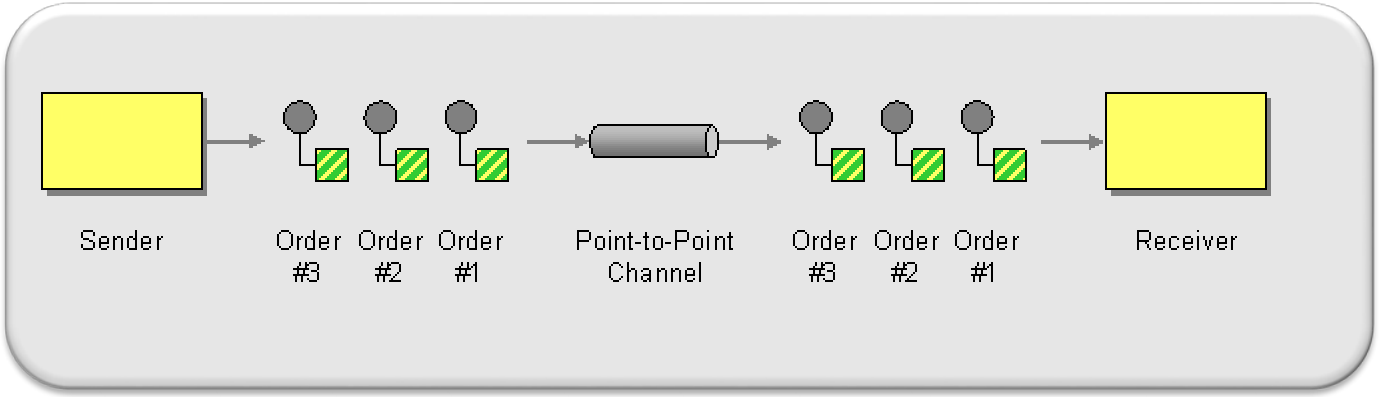
Resequencer: How can we get a stream of related but out-of-sequence messages back into the correct order?



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

Messaging Channels

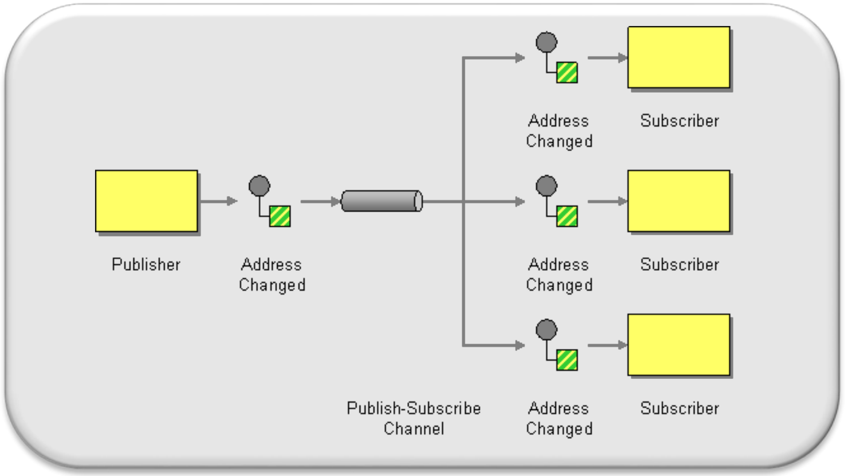
Point-to-Point: How can the caller be sure that exactly one receiver will receive the document or perform the call?



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

Messaging Channels

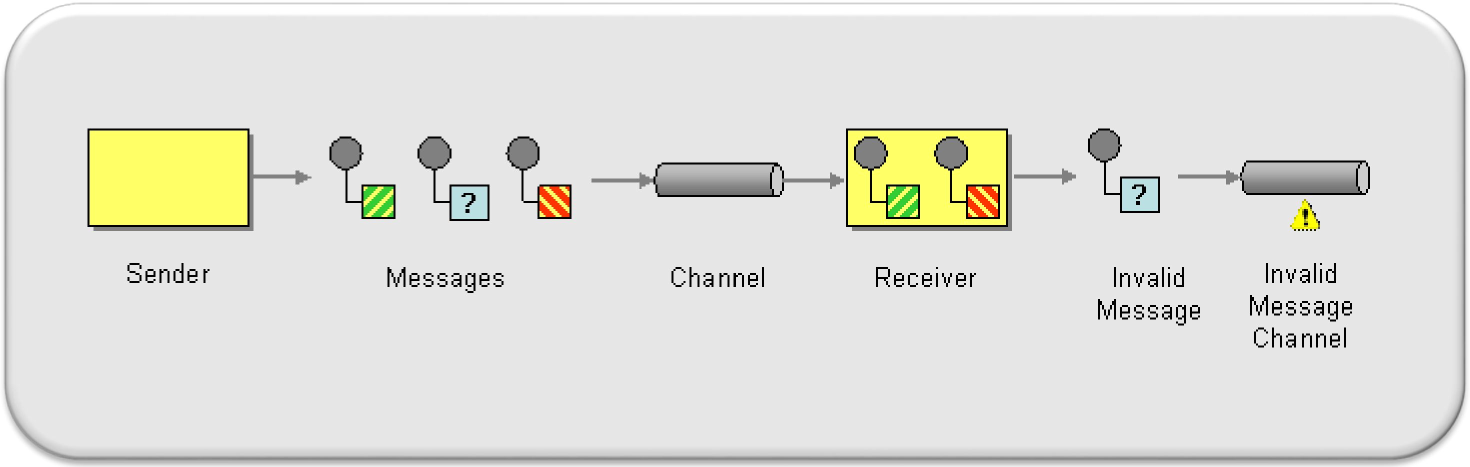
Publish-Subscribe Channel: How can the sender broadcast an event to all interested receivers?



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

Messaging Channels

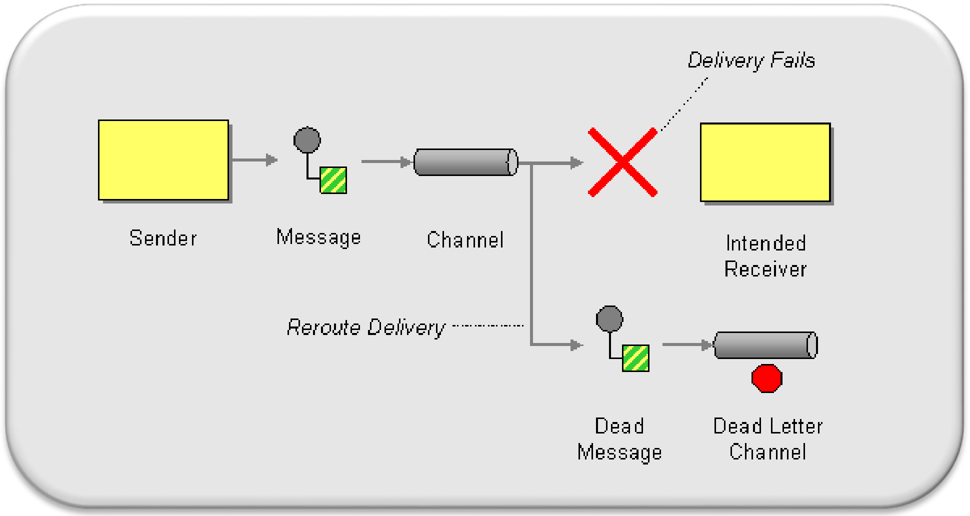
Invalid Message Channel: How can a messaging receiver gracefully handle receiving a message that makes no sense?



The receiver should move the improper message to an *Invalid Message Channel*, a special channel for messages that could not be processed by their receivers.
When designing a messaging system for applications to use, the administrator will need to define one or more *Invalid Message Channels* for the applications to use.

Messaging Channels

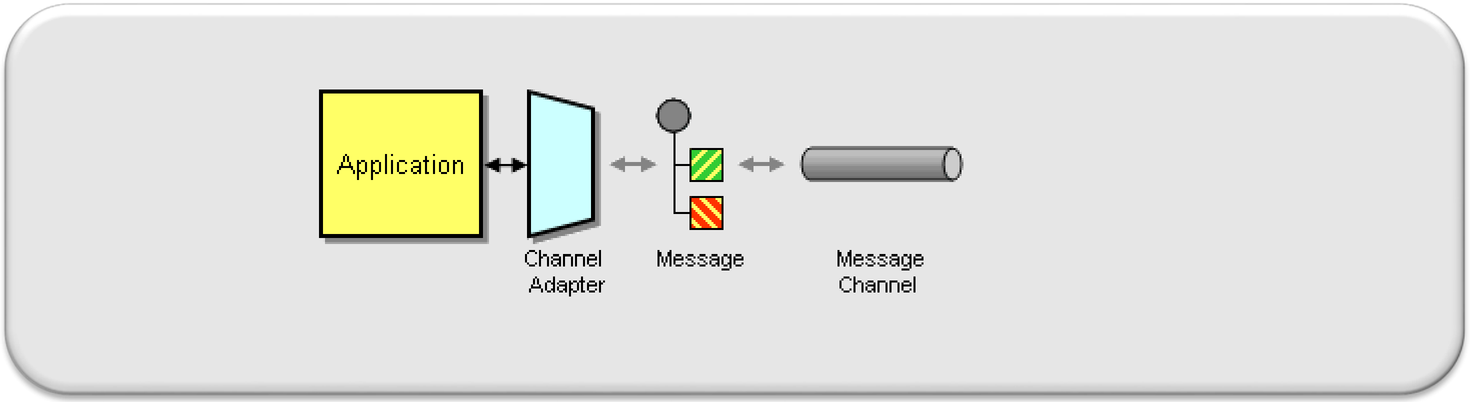
Dead Letter Channel: What will the messaging system do with a message it cannot deliver?



When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a *Dead Letter Channel*.
The specific way a *Dead Letter Channel* works depends on the specific messaging system's implementation, if it provides one at all. The channel may be called a "dead message queue" [Monson-Haefel, p.125] or "dead letter queue." [MQSeries], [Dickman, pp.28-29] Typically, each machine the messaging system is installed on has its own local *Dead Letter Channel* so that whatever machine a message dies on, it can be moved from one local queue to another without any networking uncertainties. This also records what machine the message died on. When the messaging system moves the message, it may also record the original channel the message was supposed to be delivered on.

Messaging Channels

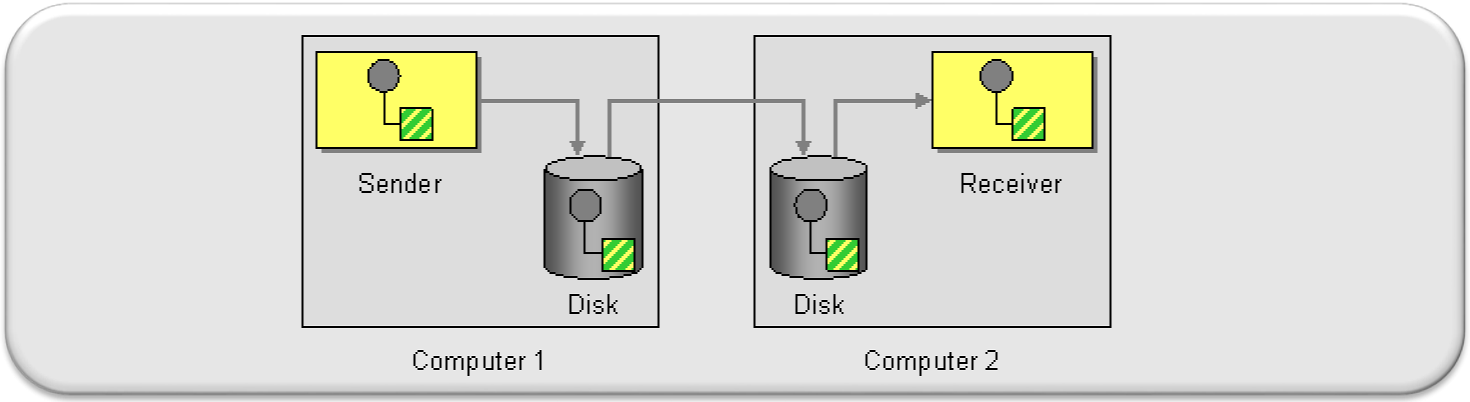
Channel Adapter**:** How can you connect an application to the messaging system so that it can send and receive messages?



Use a *Channel Adapter* that can access the application's API or data and publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application.
The adapter acts as a messaging client to the messaging system and invokes applications functions via an application-supplied interface. This way, any application can connect to the messaging system and be integrated with other applications as long as it has a proper *Channel Adapter*.

Bausteine der Mediation Layer – Messaging Channels

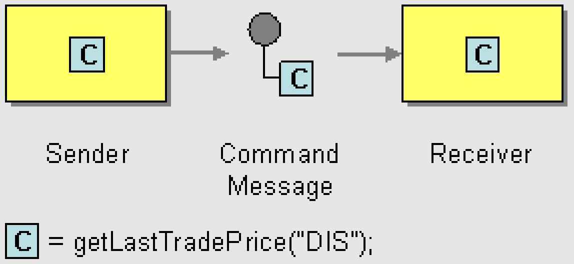
***Guaranteed Delivery*:** How can the sender make sure that a message will be delivered, even if the messaging system fails?



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

Message Construction

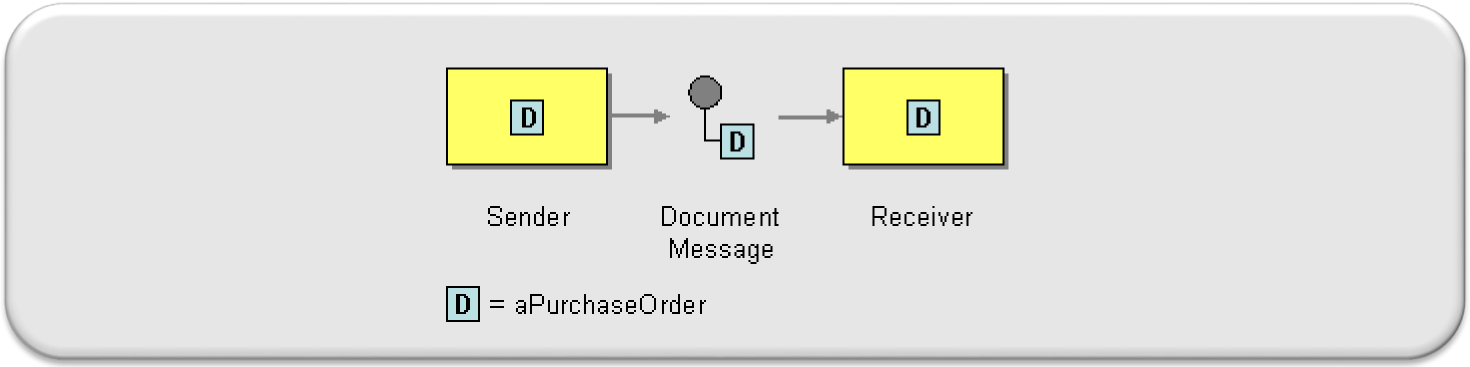
***Command Message*:** How can messaging be used to invoke a procedure in another application?



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

Message Construction

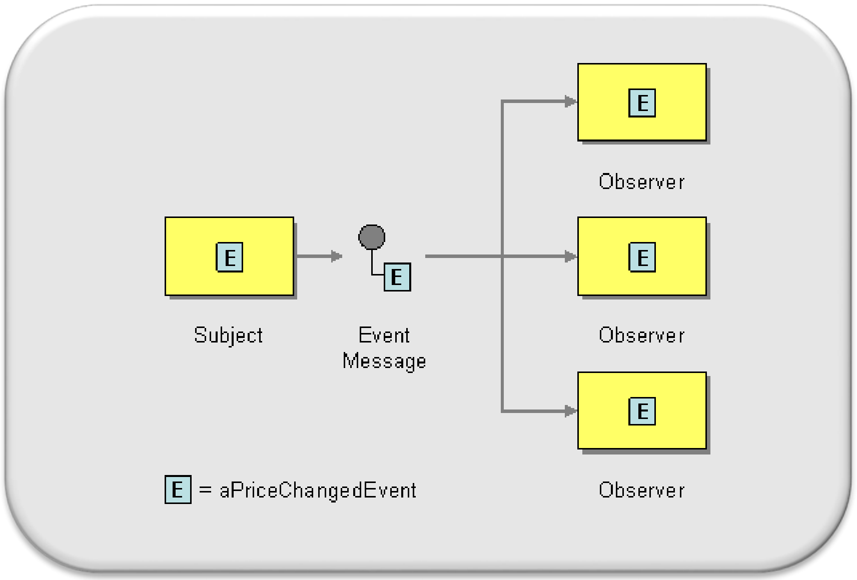
***Document Message*: How can messaging be used to transfer data between applications?**



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

Message Construction

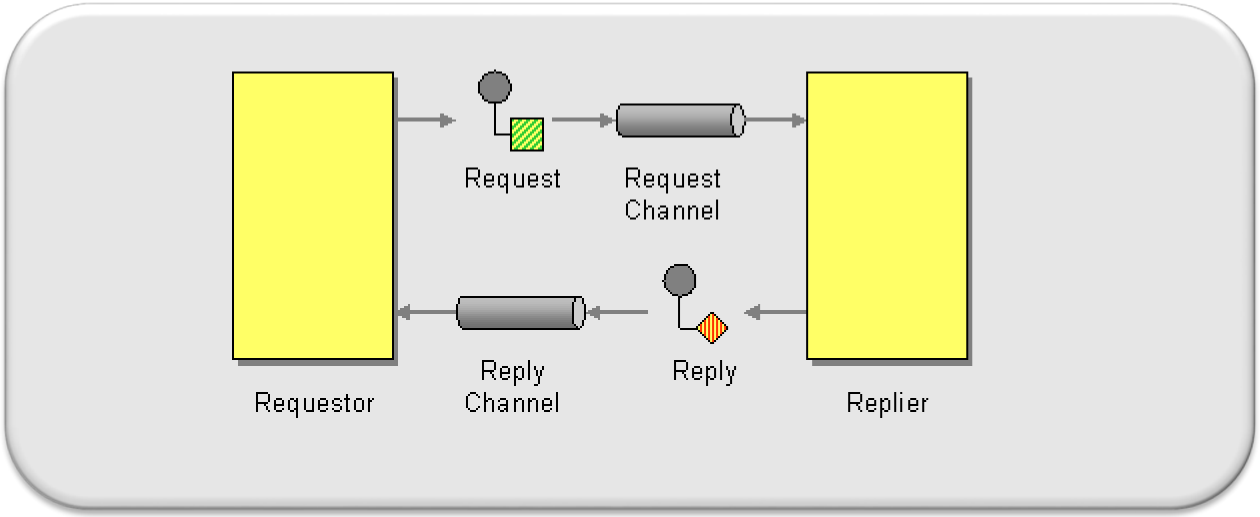
Event Message: How can messaging be used to transmit events from one application to another?



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

Message Construction

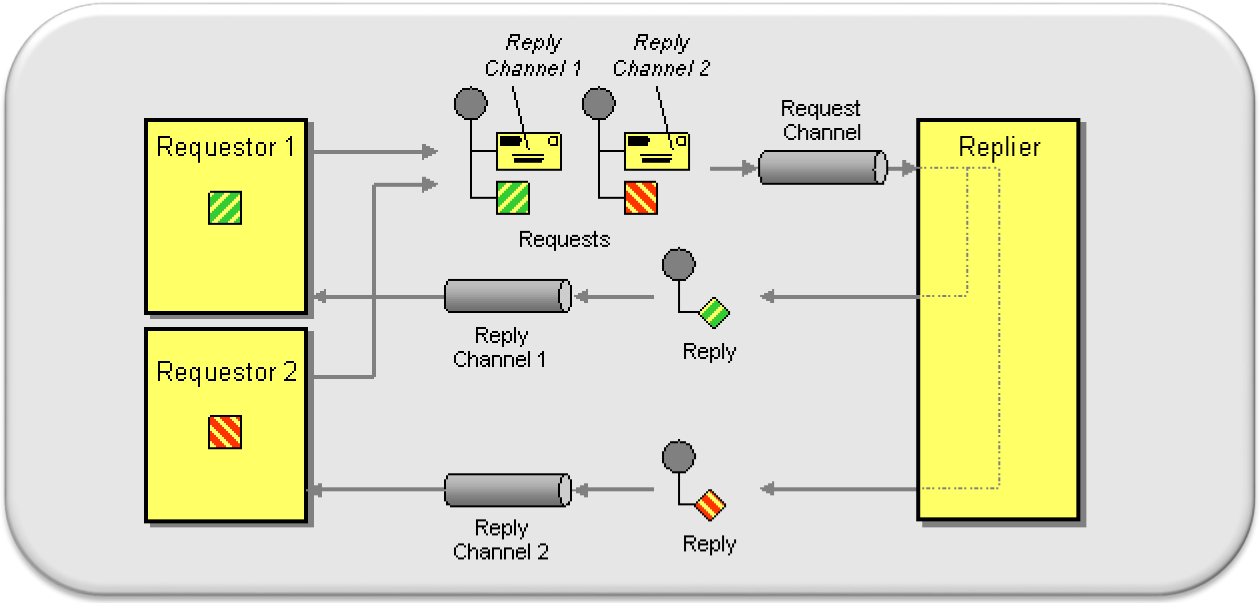
Request-Reply**:** When an application sends a message, how can it get a response from the receiver?



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

Message Construction

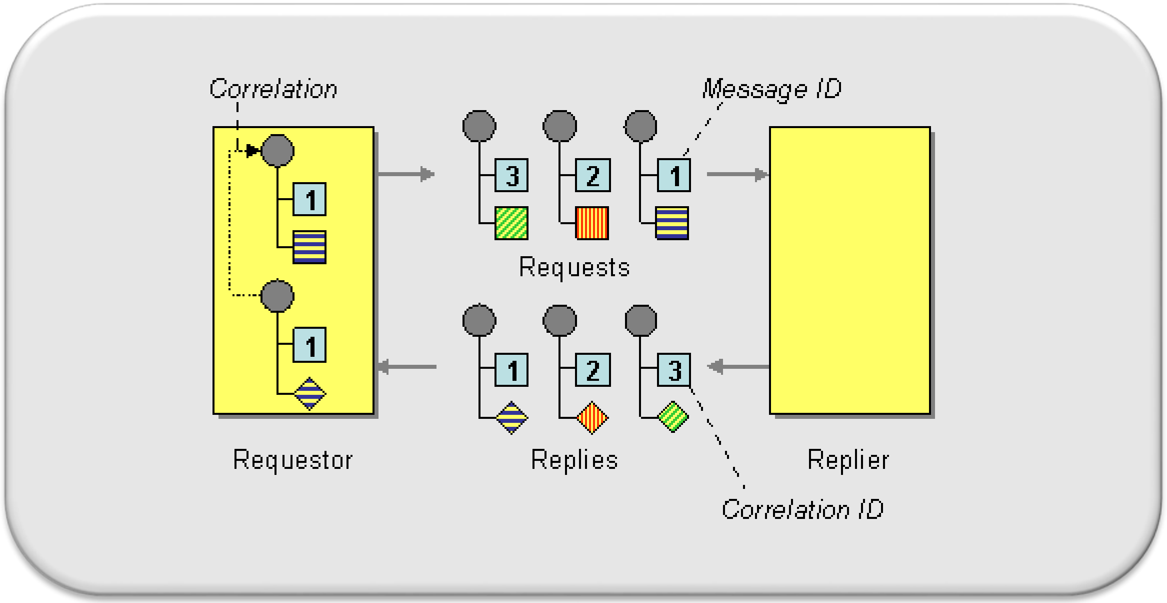
***Return Address*: How does a replier know where to send the reply?**



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

Message Construction

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

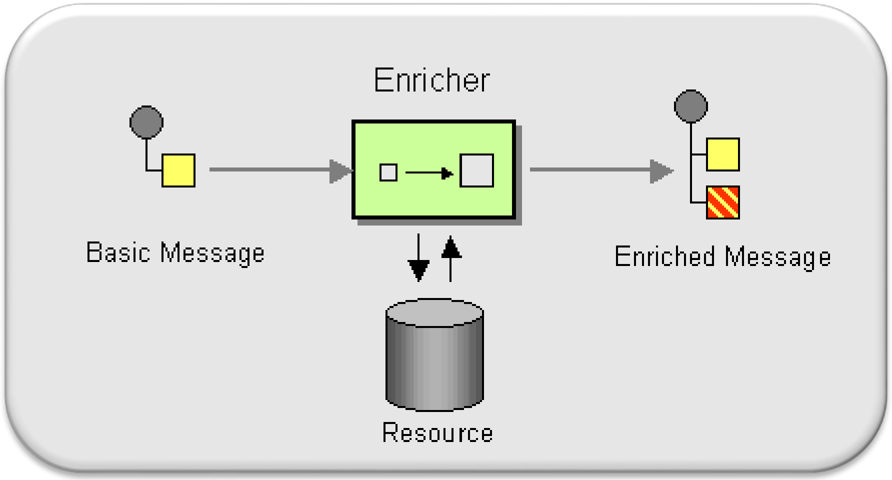


Each reply message should contain a *Correlation Identifier*, a unique identifier that indicates which request message this reply is for.

There are six parts to *Correlation Identifier*:

Message Transformation

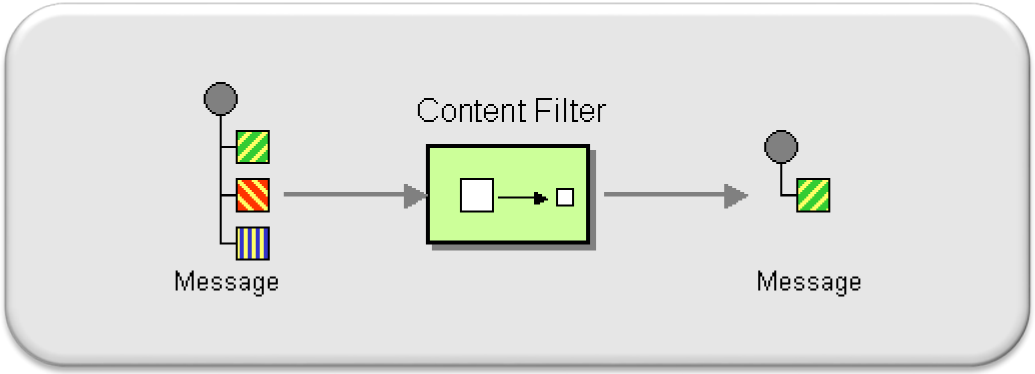
Content Enricher: How do we communicate with another system if the message originator does not have all the required data items available?



Use a specialized transformer, a *Content Enricher*, to access an external data source in order to augment a message with missing information.
The *Content Enricher* uses information inside the incoming message (e.g. key fields) to retrieve data from an external source. After the *Content Enricher* retrieves the required data from the resource, it appends the data to the message. The original information from the incoming message may be carried over into the resulting message or may no longer be needed, depending on the specific needs of the receiving application.

Message Transformation

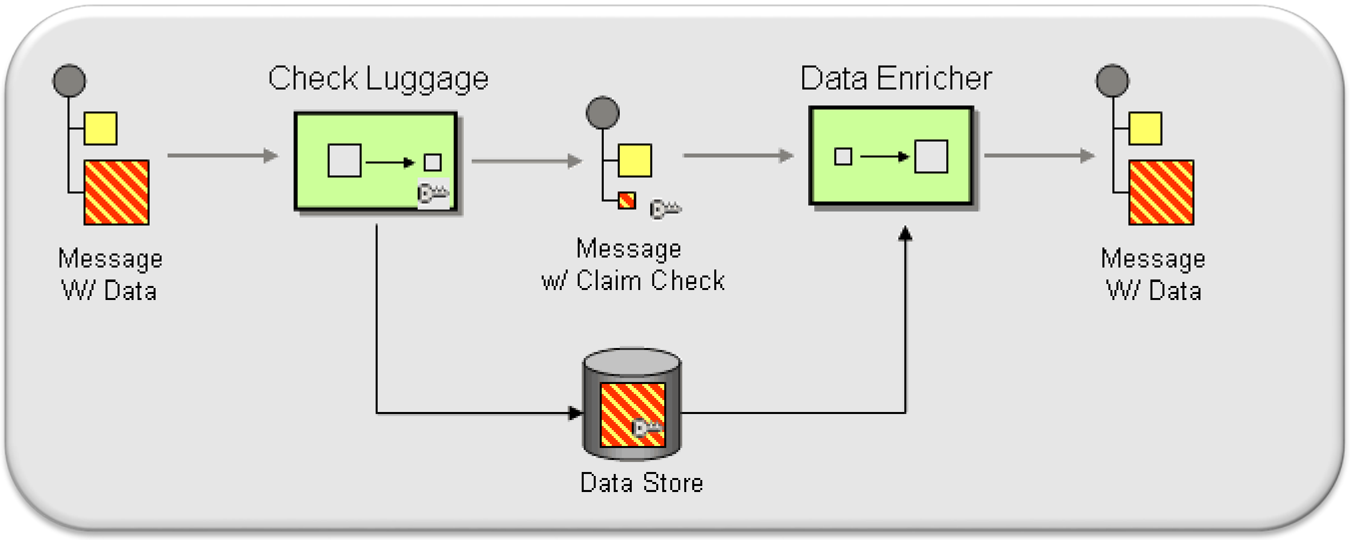
Content Filter: How do you simplify dealing with a large message, when you are interested only in a few data items?



Use a *Content Filter* to remove unimportant data items from a message leaving only important items.
The *Content Filter* does not necessarily just remove data elements. A *Content Filter* is also useful to simplify the structure of the message. Often times, messages are represented as tree structures. Many messages originating from external systems or packaged applications contain many levels of nested, repeating groups because they are modeled after generic, normalized database structures. Frequently, known constraints and assumptions make this level of nesting superfluous and a *Content Filter* can be used to 'flatten' the hierarchy into a simple list of elements than can be more easily understood and processed by other systems.

Message Transformation

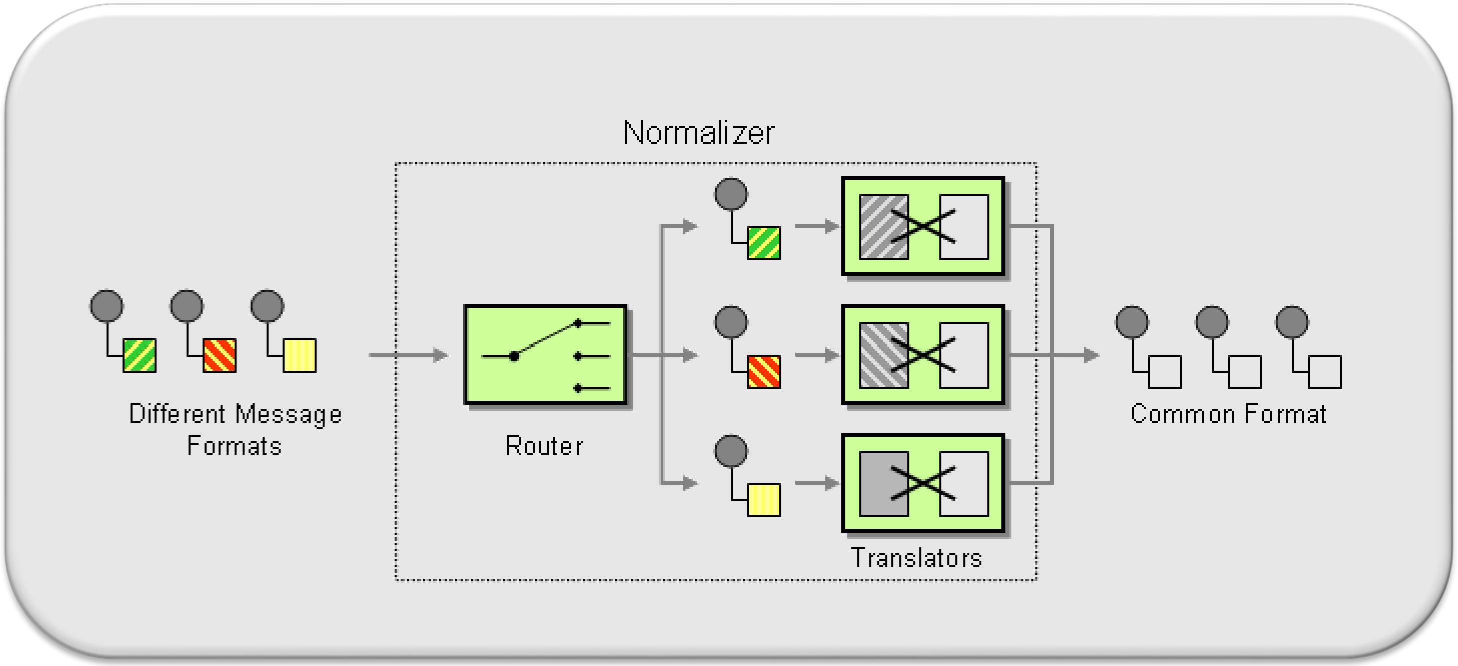
Claim Check: How can we reduce the data volume of message sent across the system without sacrificing information content?



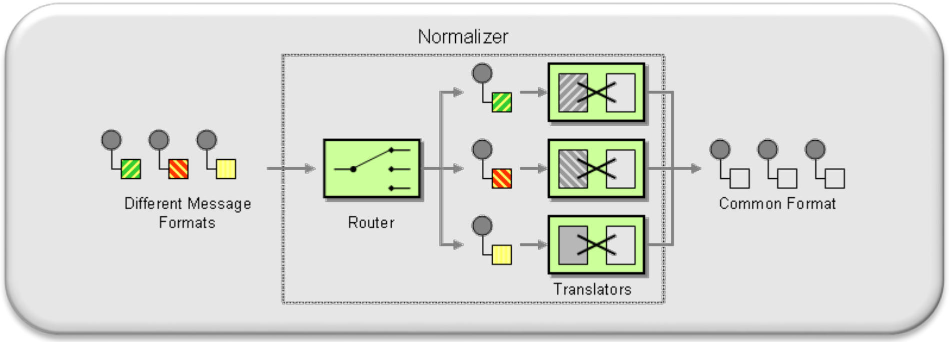
Store message data in a persistent store and pass a *Claim Check* to subsequent components. These components can use the *Claim Check* to retrieve the stored information.

Message Transformation

Normalizer: How do you process messages that are semantically equivalent, but arrive in a different format?



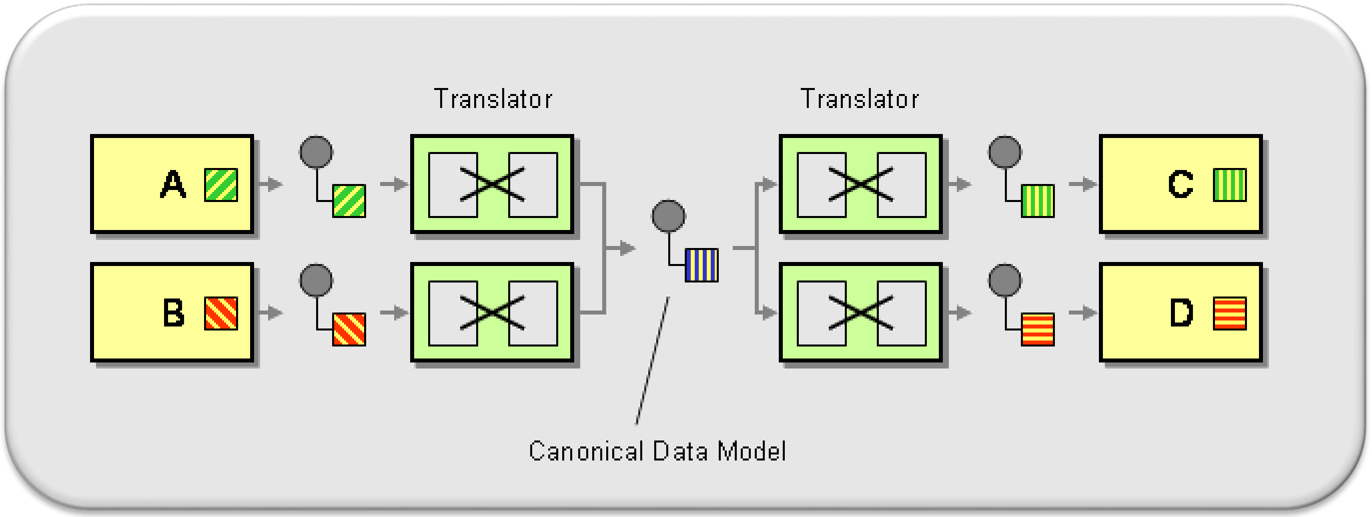
Message Transformation



The *Normalizer* uses a *Message Router* to route the incoming message to the correct *Message Translator*. This requires the *Message Router* to detect the type of the incoming message. Many messaging systems equip each message with a type specifier field in the Message Header to make this type of task simple. However, in many B2B scenarios messages do not arrive as messages compliant with the enterprise's internal messaging system, but in diverse formats such as comma separated files or XML document without associated schema. While it is certainly best practice to equip any incoming data format with a type specifier we know all to well that the world is far from perfect. As a result, we need to think of more general ways to identify the format of the incoming message. One common way for schema-less XML documents is to use the name of the root element to assume the correct type. If multiple data formats use the same root element, you can use XPATH expressions to determine the existence of specific sub-nodes. Comma-separated files can require a little more creativity. Sometimes you can determine the type based on the number of fields and the type of the data (e.g. numeric vs. string). If the data arrives as files, the easiest way may be to use the file name or the file folder structure as a surrogate *Datatype Channel*. Each business partner can name the file with a unique naming convention. The *Message Router* can then use the file name to route the message to the appropriate *Message Translator*.

Message Transformation

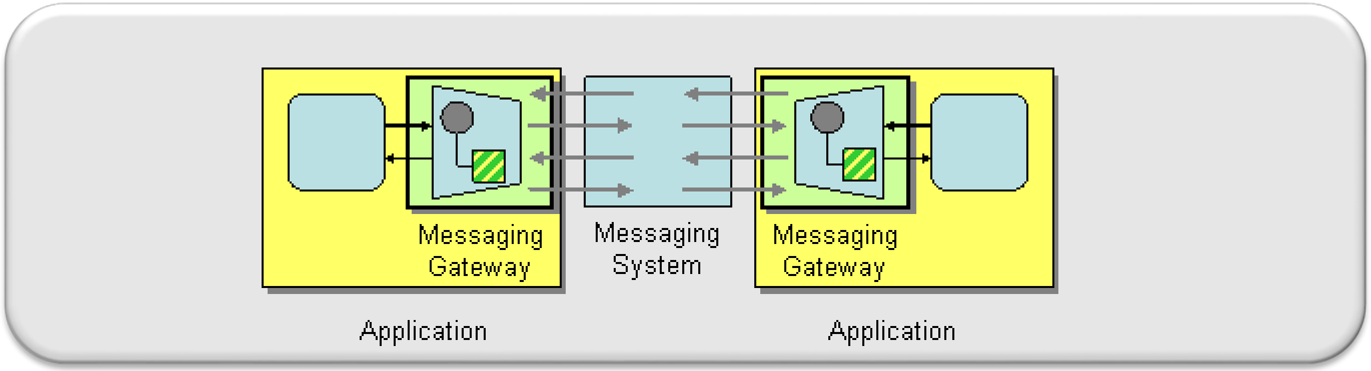
Canonical Data Model: How can you minimize dependencies when integrating applications that use different data formats?



Therefore, design a *Canonical Data Model* that is independent from any specific application. Require each application to produce and consume messages in this common format.
The *Canonical Data Model* provides an additional level of indirection between application's individual data formats. If a new application is added to the integration solution only transformation between the *Canonical Data Model* has to created, independent from the number of applications that already participate.

Messaging Endpoints

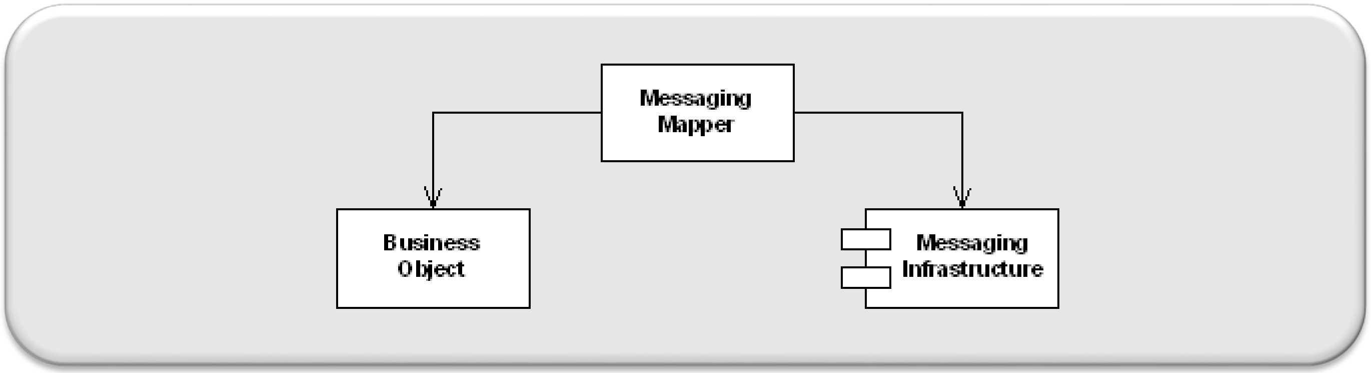
Messaging Gateway: How do you encapsulate access to the messaging system from the rest of the application?



Use a *Messaging Gateway*, a class than wraps messaging-specific method calls and exposes domain-specific methods to the application.
The *Messaging Gateway* encapsulates messaging-specific code (e.g., the code required to send or receive a message) and separates it from the rest of the application code. This way, only the *Messaging Gateway* code knows about the messaging system; the rest of the application code does not. The *Messaging Gateway* exposes a business function to the rest of the application so that instead of requiring the application to set properties like Message.MessageReadPropertyFilter.AppSpecific, a *Messaging Gateway* exposes methods such as GetCreditScore that accept strongly typed parameters just like any other method. A *Messaging Gateway* is a messaging-specific version of the more general *Gateway* pattern [EAA].

Messaging Endpoints

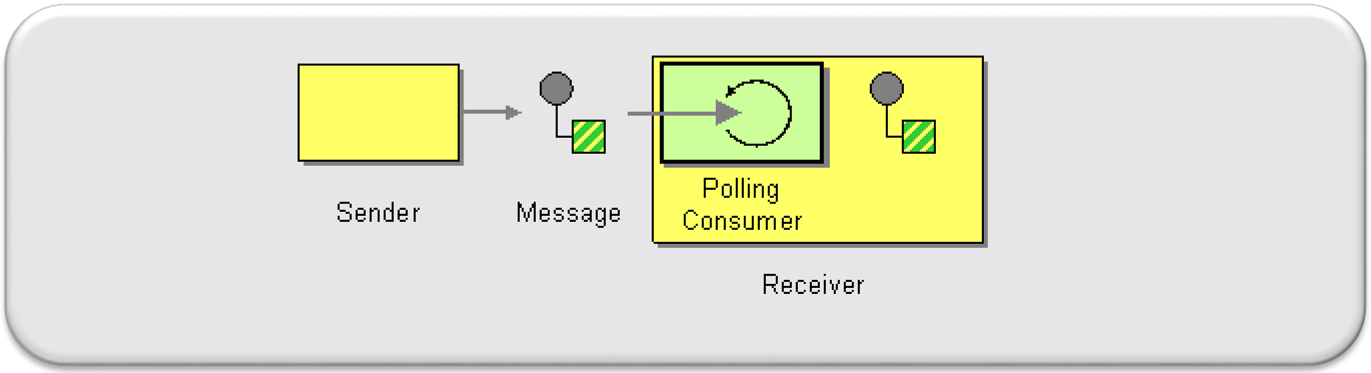
Messaging Mapper**:** How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?



Create a separate *Messaging Mapper* that contains the mapping logic between the messaging infrastructure and the domain objects. Neither the objects nor the infrastructure have knowledge of the *Messaging Mapper*'s existence.
The *Messaging Mapper* accesses one or more domain objects and converts them into a message as required by the messaging channel. It also performs the opposite function, creating or updating domain objects based on incoming messages. Since the *Messaging Mapper* is implemented as a separate class that references the domain object(s) and the messaging layer, neither layer is aware of the other. The layers don't even know about the *Messaging Mapper*.

Messaging Endpoints

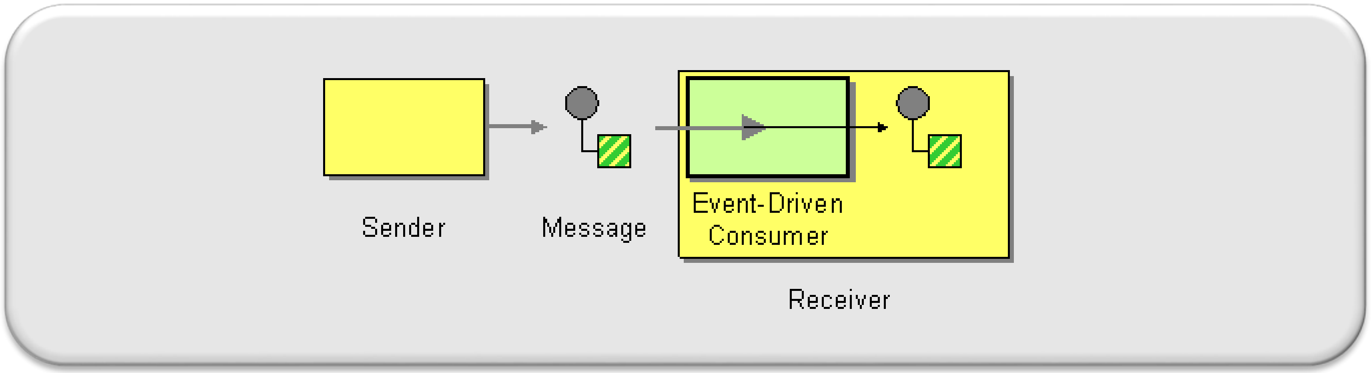
Polling Consumer: How can an application consume a message when the application is ready?



The aplication should use a *Polling Consumer*, one that explicitly makes a call when it wants to receive a message.
This is also known as a synchronous receiver, because the receiver thread blocks until a message is received. We call it a *Polling Consumer* because the receiver polls for a message, processes it, then polls for another. As a convenience, messaging API's usually provide a receive method that blocks until a message is delivered, in addition to methods like receiveNoWait() and Receive(0) that return immediately if no message is available. This difference is only apparent when the receiver is polling faster than messages are arriving.

Messaging Endpoints

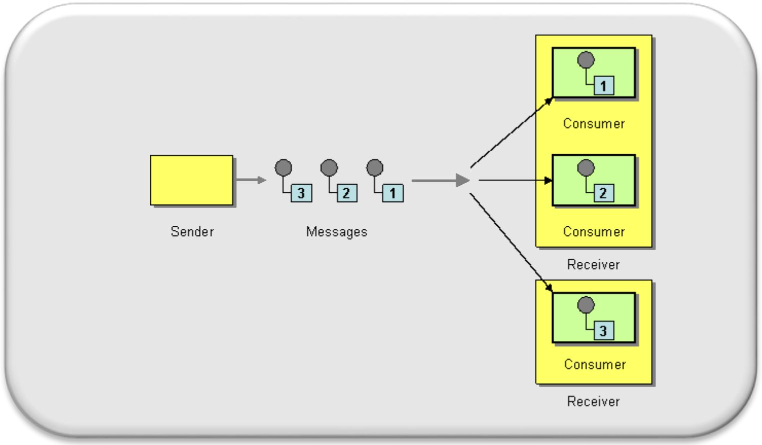
Event-Driven Consumer: How can an application automatically consume messages as they become available?



The application should use an *Event-Driven Consumer*, one that is automatically handed messages as they're delivered on the channel.
This is also known as an asynchronous receiver, because the receiver does not have a running thread until a callback thread delivers a message. We call it an *Event-Driven Consumer* because the receiver acts like the message delivery is an event that triggers the receiver into action.

Messaging Endpoints

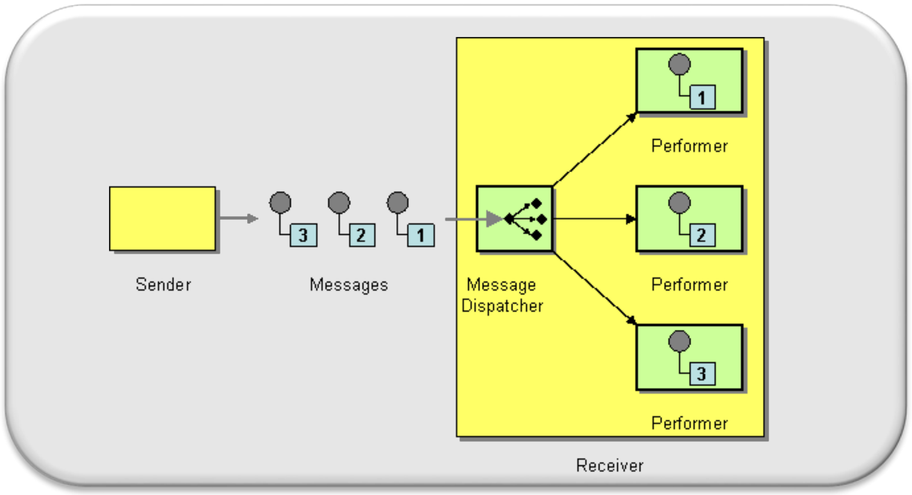
***Competing Consumers*:** How can a messaging client process multiple messages concurrently?



Create multiple *Competing Consumers* on a single channel so that the consumers can process multiple messages concurrently.
Competing Consumers are multiple consumers that are all created to receive messages from a single *Point-to-Point Channel*. When the channel delivers a message, any of the consumers could potentially receive it. The messaging system's implementation determines which consumer actually receives the message, but in effect the consumers compete with each other to be the receiver. Once a consumer receives a message, it can delegate to the rest of its application to help process the message. (This solution only works with *Point-to-Point Channel*s; multiple consumers on a *Publish-Subscribe Channel* just create more copies of each message.)

Messaging Endpoints

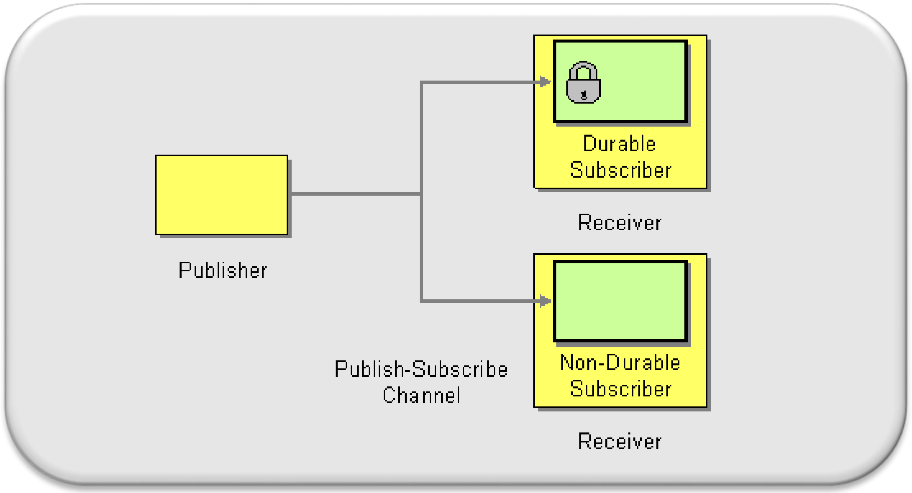
***Message Dispatcher* :** How can multiple consumers on a single channel coordinate their message processing?



Create a *Message Dispatcher* on a channel that will consume messages from a channel and distribute them to performers.

Messaging Endpoints

***Durable Subscriber*: How can a subscriber avoid missing messages while it’s not listening for them?**



Use a *Durable Subscriber* to make the messaging system save messages published while the subscriber is disconnected.
A durable subscription saves messages for an inactive subscriber and deliveres these saved messages when the subscriber reconnects. In this way, a subscriber will not loose any messages even though it disconnected. A durable subscription has no effect on the behavior of the subscriber or the messaging system while the subscriber is *active* (e.g., connected). A connected subscriber acts the same whether its subscription is durable or non-durable. The difference is in how the messaging system behaves when the subscriber is disconnected.

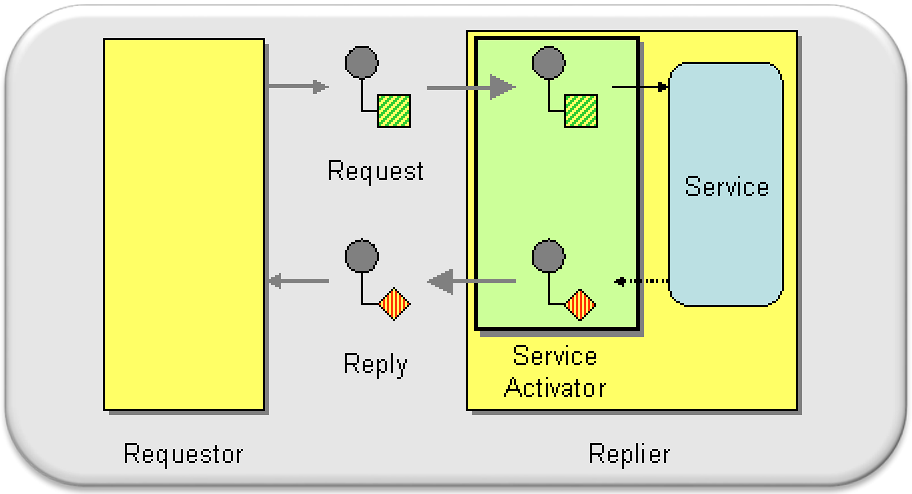
Messaging Endpoints

***Idempotent Receiver*:** How can a message receiver deal with duplicate messages?

Design a receiver to be an *Idempotent Receiver*--one that can safely receive the same message multiple times. The term *idempotent* is used in mathematics to describe a function that produces the same result if it is applied to itself, i.e. f(x) = f(f(x)). In *Messaging* this concepts translates into the a message that has the same effect whether it is received once or multiple times. This means that a message can safely be resent without causing any problems even if the receiver receives duplicates of the same message.

Messaging Endpoints

***Service Activator*:** How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?



Design a *Service Activator* that connects the messages on the channel to the service being accessed.
A *Service Activator* can be one-way (request only) or two-way (*Request-Reply*). The service can be as simple as a method call—synchronous and non-remote—perhaps part of a Service Layer [EAA]. The activator can be hard-coded to always invoke the same service, or can use reflection to invoke the service indicated by the message. The activator handles all of the messaging details and invokes the service like any other client, such that the service doesn't even know it's being invoked through messaging.