BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Camel Basics

- CamelContext object represents the Camel runtime system.
- Registry: Holds all the bean necessary for routings.
- Exchange: contains message related headers properties.
- TypeConverter: Type conversion utility for converting types.
- Component is a factory for creating Endpoint instances.
  - create a CamelContext
  - optionally configure components or endpoints
  - add routing rules
  - start the context

Producer Template

```
CamelContext camelContext = getContext();
ProducerTemplate producer = camelContext.createProducerTemplate();

byte[] data = getData();
String fileName = "mydata.csv";
String url = "ftp:someserver:2121/somedir?username=me&password=secret";

producer.sendBodyAndHeader(url, data, Exchange.FILE_NAME, fileName);
```

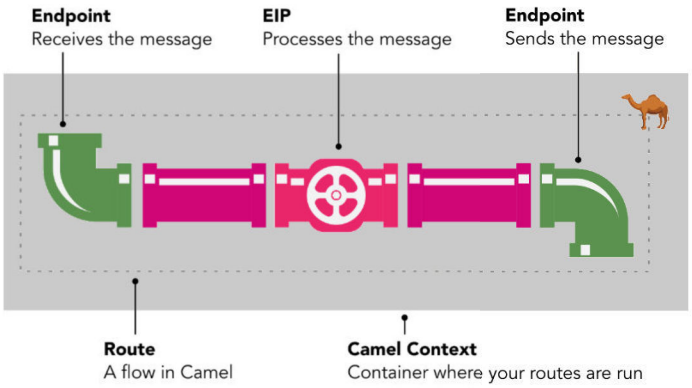

Camel facts

- It's built in Java
- The entire code is completely open source
- It's not just for web services – it's a general integration framework
- It comes with a huge library of components
- It's mature – Apache Camel is at the foundation of some commercial integration products, like Red Hat Fuse and Talend ESB.

What is Camel used for?

- Almost any time you need to move data from A to B, you can probably use Camel.
- Any of the following scenarios could be implemented using Camel:
- Picking up invoices from an FTP server and emailing them to your Accounts department
- Taking files from a folder and pushing them into Google Drive
- Taking messages from a JMS queue and using them to invoke a web service
- Making a web service that allows users to retrieve customer details from a database

Apache CAMEL



innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Apache CAMEL

innovate achieve lead

- **Route**

- The basic concept in Camel is the route.
- Routes are objects which you configure in Camel, which move your data from A to B.

- To use the plumbing example from earlier, a route is a pipe that moves data from one place to another.
- It moves data between things called endpoints.

- You can create routes in Camel either using a Java syntax, or using an XML syntax.
- Here's a very simple Java route:

```
// This is a complete Camel route definition!
from("file:home/customers/new")
    .to("file:home/customers/old")
```


Apache CAMEL

innovate achieve lead

- **Endpoints**

- In Camel, an **endpoint** represents any other external system to Camel.
- For an example, at the start of a route, Camel **receives** a message from an endpoint.
- Then, the message might be processed in some way – perhaps by an EIP – before being sent to another destination **endpoint**.

- **Components**

- To allow Camel to connect to an endpoint, it comes with a library of **components**.
- A component is simply like a plug that allows you to connect to an external system (such as a file on disk, a mailbox, or an app like Dropbox, Twitter, etc).
- You can also think of it like an adaptor.

Apache CAMEL

innovate achieve lead

- **Camel components** are reusable, open source, and you can even contribute your own.
- Here are some of the most common components, and how you might reference them in an endpoint.

Component	Purpose	Endpoint URI
HTTP	for creating or consuming web sites	http:
File	for reading and writing files	file:
JMS	for reading and writing to messaging queues	jms:
Direct	for joining your Camel routes together	direct:
Salesforce	for getting data in and out of Salesforce	salesforce:

- As you can see, each component can usually read and write:
- A component that is configured to **write** something is called a **producer** – for example, writing to a file on disk, or writing to a message queue.
- A component that is configured to **read** something is called a **consumer** – for example, reading a file from disk, or receiving a REST request.
- Between each endpoint, the data can also be transformed or modified, either by passing the data through another endpoint, or by using an EIP.

Apache CAMEL - Enterprise Integration Patterns (EIPs)

innovate achieve lead

- **EIPs** are another important part of Camel. They do special processing on messages.
- When you want to perform some common activities on a message, such as **transformation, splitting and logging**, you'll use an EIP.

EIP name	What it does	Java syntax
Splitter	Splits a message into multiple parts	.split()
Aggregator	Combines several messages into one message	.aggregate()
Log	Writes a simple log message	.log()
Marshal	Converts an object into a text or binary format	.marshal()
From*	Receives a message from an endpoint	.from()
To*	Sends a message to an endpoint	.to()

innovate achieve lead

Apache CAMEL - Camel Context

- Finally, to run and manage your routes,
- Camel has a container called the Camel Context.

- Your routes run inside this engine.
- You could think of it almost like a mini application server.

- When Camel **starts**, it reads your route definitions (in Java or XML), creates the routes, adds them to a Camel Context, and starts the Camel Context.

- When Camel **terminates**, it shuts down your routes, and closes the Camel Context.

Apache CAMEL - Route

- **What does a Route look like?**
- Camel create **routes** that move data between **endpoints**, using **components**.
- Although Camel is a library for Java, it can be configured using one of two languages - either **Java** or **XML**.
- In Camel-speak, these are known as **DSLs (Domain Specific Languages).**
- Each route starts with a from, configured with a URI, that defines the endpoint where the data is coming from.
- A route can consist of multiple steps – such as transforming the data, or logging it.
- But a route usually ends with a to step, which describes where the data will be delivered to.

- A really simple route in Camel's Java DSL could look something like this:

```
// This is a complete Camel route definition!
from("file:home/customers/new")
  .to("file:home/customers/old")
```


Apache CAMEL - Route

- That same route above could be expressed in Camel's XML DSL like this:

```
<route>
  <from uri="file:home/customers/new"/>
  <to uri="file:home/customers/old"/>
</route>
```

- But… what if we wanted to add another step in our route?
- Let's say we want to log a message when we've received a file.
- Then we simply need to add our new step in between the existing steps. Like this:

```
<route>
  <from uri="file:home/customers/new"/>
  <log message="Received a new customer!"/>
  <to uri="file:home/customers/old"/>
</route>
```

- In the code above, a message is moved from the new folder to the old folder. In the middle, we use the Enterprise Integration Pattern (EIP) called log, which writes a simple Log message to the console

Apache CAMEL - What does data look like in Camel?

- Camel treats data as individual messages – like letters flowing through a post office.

- Each message is an individual object.
- A message can be huge, or it can be very small. Camel has an object to represent messages, and helpfully it's called Message.

- A Message has a body, where the message content lives.
- It also has headers, which can be used to hold values associated with the message.
- The Message object is then passed along a route.

- A Message is part of a Camel object called an Exchange.
- You'll often see the term Exchange mentioned in Camel documentation.
- An Exchange is simply a message or interaction currently taking place inside your Camel route.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Apache CAMEL - What does data look like in Camel?

- The important thing to understand about Camel's message model is that the message body can contain almost any kind of Java object, such as a List, Map or String
- The body doesn't have to be a String, like JSON or XML.

- The real **power** of Camel becomes clear when you start using these different types of objects.
- Camel has good built-in support for converting between different object types.
- In fact, for many common file types, you might barely even have to write any conversion code.

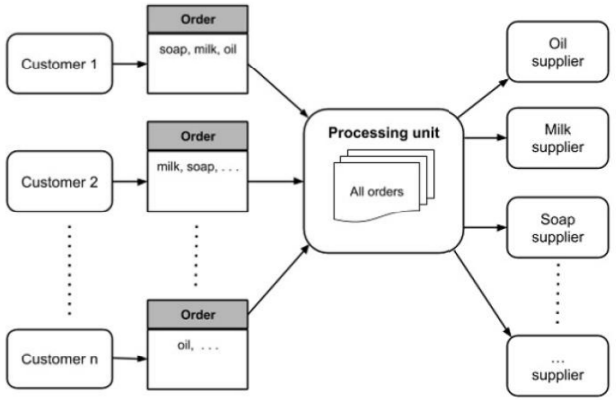
Apache CAMEL – Case Study - Assignment

- Consider a situation where a large online grocery store in your town such as the
- Bigbasket in India invites you to design an IT solution for them.
- The stable and scalable solution will help them overcome the software maintenance problems they are facing today.
- This online store has been running its business for the last decade.
- The store accepts online orders for different categories of products from their customers and distributes those to the respective suppliers.
- For example, suppose you order some soaps, oil and milk; these three items will be distributed to the three respective suppliers.
- The three suppliers will then send their supplies to a common distribution point from where the entire order will be fulfilled by the delivery center.
- Now, let us look at the problem they are facing today.

Apache CAMEL – Case Study - Assignment

- When this store started its business, it was accepting orders in a comma-separated plain
- text file.
- Over a period of time, the store switched to message-driven order placement.
- Later, some software developer suggested an XML based order placement.
- Eventually, the store even adapted a web service interface.
- Now, here comes the real problem. The orders now come in different formats.
- At the same time, as the business kept on growing, the store periodically added new suppliers to its repertoire.
- Each such supplier had its own protocol for accepting orders.
- Once again, we face the integration issue; our application architecture must be scalable to accommodate new suppliers with their unique order placement mechanism

Assignment – Design an Integration Solution



innovate achieve lead

innovate achieve lead

Thank You

innovate achieve lead

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Middleware Technologies-CSIZG524
Contact Session-5 19th April 2025
Messaging Systems

Faculty Name:Prof.(Dr.)Prasanna Balaji N

BITS Pilani
Pilani Campus

innovate achieve lead

IMP Note to Self

Start Recording

innovate achieve lead

Important Note to Students

- It is important to know that just login to the session does not guarantee the attendance.
- Once you join the session, continue till the end to consider your attendance.
- IMPORTANTLY, you need to make the class more interactive by responding to Professors queries in the session.
- Answering to Polls / Quiz questions during/after the class is mandatory to get attendance and the score will count for IA marks.
- **Whenever Professor calls your number / name, you need to respond, otherwise it will be considered as ABSENT**

innovate achieve lead

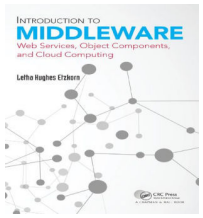
Disclaimer

- *The slides presented here are obtained from the authors of the books and from various other contributors.*
- *I hereby acknowledge all the contributors for their material and inputs.*
- *I have added and modified a few slides to suit the requirements of the course.*

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Text Books



T1: Letha Hughes Etzkorn - Introduction to middleware _ web services, object components, and cloud computing-Chapman and Hall_CRC (2017).

T2: William Grosso - Java RMI (Designing & Building Distributed Applications)

R1: Gregor Hohpe, Bobby Woolf - Enterprise Integration Patterns_ Designing, Building, and Deploying Messaging Solutions -Addison-Wesley Professional (2003)

R2: MongoDB in Action

Note: In order to broaden understanding of concepts as applied to Indian IT industry, students are advised to refer books of their choice and case-studies in their own organizations

Modular Structure

No	Title of the Module
M1	Introduction and Evolution
M2	Enterprise Middleware
M3	Middleware Design and Patterns
M4	Middleware for Web-based Application and Cloud-based Applications
M5	Specialized Middleware

CS5: Message Channels and Construction

Agenda

- Integration Styles
- Messaging Systems
- Messaging Channels
- Message Construction
- Message Routing
- Message Transformation
- Message Endpoints
- Message System Management

Click to edit Master title style

innovate achieve lead

- Messaging system has different Message Channels for different types of information the applications want to communicate.
- When two applications wish to exchange data, they do so by sending the data through a channel that connects the two.
- The application sending the data may not know which application will receive the data
- But by selecting a particular channel to send the data on, the sender knows that the receiver will be one that is looking for that sort of data by looking for it on that channel.
- In this way, the applications that produce shared data have a way to communicate with those that wish to consume it.

Click to edit Master title style

innovate achieve lead

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

- **One-to-one or one-to-many**
- **What type of data**
- **Invalid and dead messages**
- **Crash proof**
- **Non-messaging clients**
- **Communications backbone**

Click to edit Master title style

innovate achieve lead

- An application is using Messaging to make remote procedure calls (RPC's) or transfer documents. How can the caller be sure that exactly one receiver will receive the document or perform the call?
- RPC is invoked on a single remote process, so the receiver performs the procedure.
- And since the receiver was only called once, it only performs the procedure once.
- But with messaging, once a call is packaged as a Message and placed on a Message Channel, potentially many receivers could see it on the channel and decide to perform the procedure.
- The messaging system could prevent more than one receiver from monitoring a single channel, but this would unnecessarily limit callers that wish to transmit data to multiple receivers.

Click to edit Master title style

innovate achieve lead

- All of the receivers on a channel could coordinate to ensure that only one of them actually performs the procedure, but that would be complex, create a lot of communications overhead.
- Multiple receivers on a single channel may be desirable so that multiple messages can be consumed concurrently, but any one receiver should consume any single message.
- Send the message on a Point-to-Point Channel, which ensures that only one receiver will receive a particular message.

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

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

innovate achieve lead

Click to edit Master title style

- An application is using Messaging to announce events.
- How can the sender broadcast an event to all interested receivers?
- There are well-established patterns for implementing broadcasting.
- The Observer pattern provides event notification to all interested observers no matter how many observers there are (even none).
- The Publisher-Subscriber pattern [POSA] expands upon Observer by adding the notion of an event channel for communicating event notifications.
- That's the theory, but how does it work with messaging?

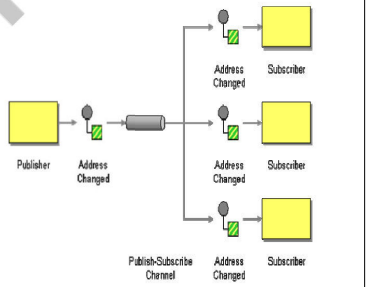
- Next Slide answers this ……..

innovate achieve lead

Click to edit Master title style

- The event can be packaged as a Message so that messaging will reliably communicate the event to the observers (subscribers). Then the event channel is a Message Channel.
- Each subscriber needs to be notified of a particular event once, but should not be notified repeatedly of the same event. The event cannot be considered consumed until all of the subscribers have been notified.
- But once all of the subscribers have been notified, the event can be considered consumed and should disappear from the channel.
- Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each receiver.
- Pictorial representation follows …..

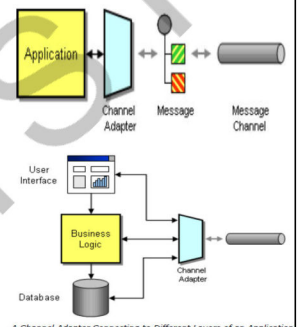
Send the event on a *Publish-Subscribe Channel*, which delivers a copy of a particular event to each receiver.



innovate achieve lead

Click to edit Master title style

- Many enterprises use Messaging to integrate multiple, disparate applications.
- How can you connect an application to the messaging system so that it can send and receive messages?
- Use a Channel Adapter that can access the application's API or data and publish messages on a channel based on this data, and
- Likewise can receive messages and invoke functionality inside the application.
- The adapter acts as a messaging client to the messaging system and invokes applications functions via an application-supplied interface.
- This way, any application can connect to the messaging system and be integrated with other applications as long as it has a proper Channel Adapter.
- The Channel Adapter can connect to different layers of the application's architecture, depending on that architecture and the data the messaging system needs to access.



A Channel Adapter Connecting to Different Layers of an Application

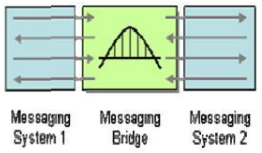
innovate achieve lead

Click to edit Master title style

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

- How can multiple messaging systems be connected so that messages available on one are also available on the others?
- What is needed is a way for messages on one messaging system that are of interest to applications on another messaging system to be made available on the second messaging system as well.

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



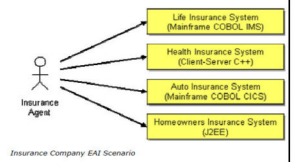
Use a message bridge, a connection between messaging systems to replicate messages between systems

innovate achieve lead

innovate achieve lead

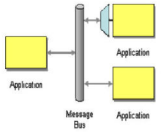
Click to edit Master title style

- An enterprise contains several existing systems that must be able to share data and operate in a unified manner in response to a set of common business requests.
- What is an architecture that enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others?
- An enterprise often contains a variety of applications that operate independently, yet need to work together in a unified manner.
- Enterprise Application Integration (EAI) describes a solution to this problem but doesn't describe how to accomplish it.
- For example, consider an insurance company that sells different kinds of insurance products (life, health, auto, home, etc.).
- As a result of corporate mergers, and of the varying winds of change in IT development, the enterprise consists of a number of separate applications for managing the company's various products.



Insurance Company EAI Scenario

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

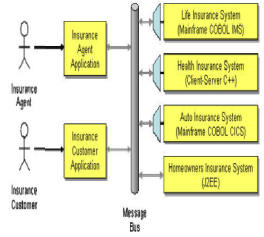


innovate achieve lead

Click to edit Master title style

- An insurance agent trying to sell a customer several different types of policies must log into a separate system for each policy, wasting effort and increasing the opportunity for mistakes.
- The agent needs a single, unified application for selling customers a portfolio of policies
- What is needed is an integration architecture that enables the product applications to coordinate in a loosely coupled way, and for user applications to be able to integrate with them.
- Structure the connecting middleware between these applications as a Message Bus that enables them to work together using messaging.
- A Message Bus is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces.

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



Insurance Company Message Bus

innovate achieve lead

Click to edit Master title style

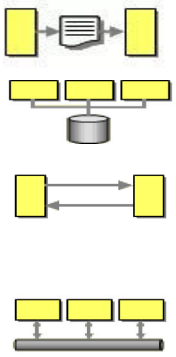
- Yet the message contents may be time-sensitive, such that if the message isn't received by a deadline, it should just be ignored and discarded.
- In this situation, the sender can use "Message Expiration" to specify an expiration date.
- Now let us go into the details of different types Message construction :

1. Command Message
2. Document Message
3. Event Message
4. Request-Reply
5. Return Address
6. Correlation Identifier
7. Message Sequence
8. Format Identifier
9. Message Expiration

innovate achieve lead

Integration Styles

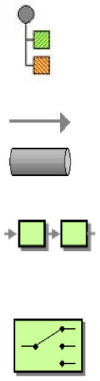
- File transfer mode applications exchange information using files, files are shared at some common location.
- Shared Database applications use a common database schema.
- RPC based an application exposes its functionality using interfaces, the caller needs to be aware of those and invokes them using stubs. Except RPC all the three mechanisms above are asynchronous in nature.
- Messaging : An entity mediates between applications that want to exchange data. It does the job of accepting messages from producers and then delivering to the consumers. Messaging helps to achieve loose coupling while integrating applications. It isolates the connecting applications from API changes/upgrades that happens over time.



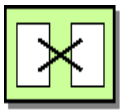
innovate achieve lead

Messaging Systems

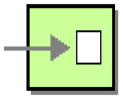
- **Message** Structure is well defined by the messaging system used. It usually contains header and body sections.
- **Channels** are the mediums where messages are produced. They are the usual queue and topics
- **Pipe and Filters** pattern is useful when one needs to process messages before they are delivered to consumer applications.
- **Message Router:** When the sender application does not know on which channel the receiver is subscribed to. A router is used in between for delivering messages on the correct channel. It has routing rules that decides where the messages shall be delivered.



- **Translators** are used to change the format of message. The sender application might send CSV data while the receiver application understands XML, in that case we need to have a translator before the receiving application that does the job of CSV to XML conversion.



- **Endpoint** is a component that helps an application interact with messaging systems. They have the protocol built in for communication with the messaging systems. They are the message producers and the consumers.



Messaging Channels

Peer 2 Peer

Channels that deliver a message to a single consumer. Example is a queue

Publish Subscribe

Channels that broadcast a message to all subscribing consumers. Topics are of pub-sub nature.

Dead Letter Channel

Channels used for moving messages which cannot be processed. Cases when the consumer can't understand or messages get expired. This is important from the point of monitoring & management.

Messaging Bridge

These are the channel adapters that bridges different messaging systems. Consider a case when there are two enterprise systems, one uses Mircosoft's MQ while the other uses IBM's MQ server. There you need a bridge that can connect these.

Guaranteed delivery

Persistent channels are used to guarantee message delivery. In case the message system crashes, it would lose all messages present in memory. So usually channels are backed up a persistent store where all messages in the channel are stored.

innovate achieve lead

Message Construction

Command Message

These specify a method or a function that the receiver should invoke. Consider the case when XML is being used, the root node may specify the method name while the child elements specify the arguments for method invocation.

Document Message

When the sender passes data but it does not know what the receiver should do with it.

Event Message

Sender sends notification messages about changes happening on its end.
Here the receiver may choose to ignore or react to it.

Request Reply

In this the sender expects a reply back. Message might be composed of two parts, one contains the request and other is populated by the receiver i.e. the response.

Message Routing

Content Based Routing

Messages are inspected for determining the correct channel. Where XML is used, rules are written in XPath.

Message Filter

When a receiver is only interested in messages having certain properties, then it needs to apply a filter. This capability is generally comes in built with messaging systems.

Splitter

In case when messages arrive in a batch. A splitter is required to break the message into parts that can be processed individually.

Aggregator

An aggregator does the opposite job of a splitter. It correlates and combines similar messages.

Message Transformation

Content Enricher

An enricher does the job of adding extra information to a message.
This is required in case all the data to process the message is not present.

Content Filter

The Content Filter does the opposite it removes unwanted data from a message.

Normalizer

A normalizer does the job of converting messages arriving in different formats to a common format. Your application needs the ability to accept data in JSON, XML, CSV etc but the processing logic only understands XML, in that case you need a normalizer.

Message Endpoints

Transaction Client

A transaction client helps in coordinating with external transaction services.

Message Dispatcher

Message dispatcher is a pattern where the receiver dispatches the arriving messages to workers. The workers have the job of processing the message.

Event Driven Consumer

In this the receiver registers an action on the messaging system;
on receiving a message the messaging systems calls that action.

innovate achieve lead

Message System mgmt

innovate achieve lead

Detour

As the name specifies, the path of the message is changed for doing activities such as validation, logging etc. This extra processing is control based, which can be turned off for performance reasons.

Wire Tap

Here the message is copied to a channel and is later retrieved for inspection or analysis.

Message Store

As the message is passed on from the receiver to the processing unit, the whole message or parts of it (some properties from header or message body) are stored in a central location.

Some Pattern examples

innovate achieve lead

Publish-Subscribe Channel	Google Cloud Pub/sub
Dead Letter Channel	Amazon SQS
Return Address	GoLang
Content-based Router	Apache Camel
Message Filter	RabbitMQ
Event-driven Consumer	RabbitMQ
Competing Consumers	Apache Kafka
Channel Purger	Amazon SQS

Integration Framework – Apache Camel

innovate achieve lead

- Powerful open source integration framework based on EIP
- java framework for integration and mediation
- rules in multiple DSLs:
  - Fluent Java
  - Spring XML
  - Blueprint XML
  - Scala
- over > 150 out-of-box components
- bean binding and integration with popular frameworks

Camel Architecture

innovate achieve lead

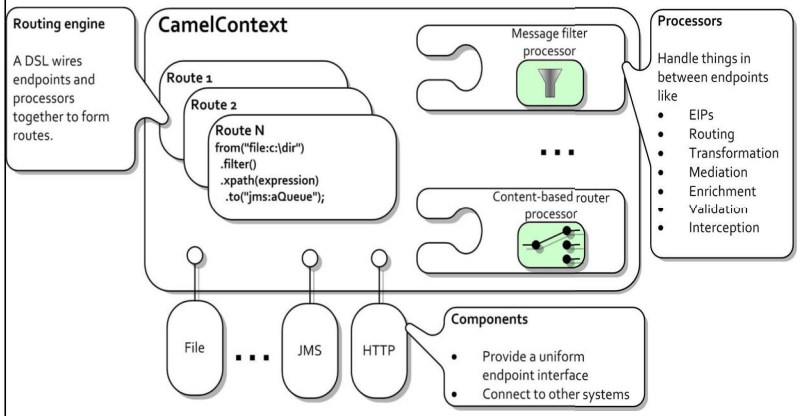


Image from: (Camel in Action)

Basics

innovate achieve lead

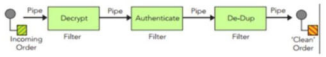
- CamelContext object represents the Camel runtime system.
- Registry: Holds all the bean necessary for routings.
- Exchange: contains message related headers properties.
- TypeConverter: Type conversion utility for converting types.
- Component is a factory for creating Endpoint instances.
  - create a **CamelContext**
  - optionally configure **components** or **endpoints**
  - add **routing rules**
  - start the **context**

ProducerTemplate

```
CamelContext camelContext = getContext();
ProducerTemplate producer = camelContext.createProducerTemplate();

byte[] data = getData();
String fileName = "mydata.csv";
String url = "ftp:someserver:2121/somedir?username=me&password=secret";

producer.sendBodyAndHeader(url, data, Exchange.FILE_NAME, fileName);
```

EIP with Camel

Pipes And Filters EIP

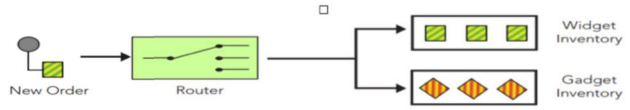


- Camel supports Pipes and Filters using the pipeline node.

```
from("jms:queue:order:in")
  .pipeline("direct:transformOrder","direct:validateOrder","jms:queue:order:process");

from("jms:queue:order:in")
    .to("direct:transformOrder","direct:validateOrder", "jms:queue:order:process");

from("jms:queue:order:in")
    .to("direct:transformOrder")
    .to("direct:validateOrder")
    .to("jms:queue:order:process");
```


innovate achieve lead

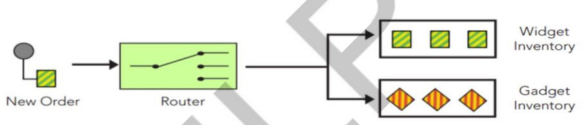
Content Based Router



- Camel supports content based routing based on choice, filter, or any other expression.

```
from("jms:queue:order")
    .choice()
    .when(header("type").in("widget","wiggy"))
    .to("jms:queue:order:widget")
    .when(header("type").isEqualTo("gadget"))
    .to("jms:queue:order:gadget")
    .otherwise().to("jms:queue:order:misc")
    .end();
```

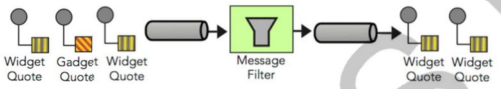
Message Translater



• Camel supports the **message translator** using the **processor**, **bean** or **transform** nodes.

```
class OrderTransformProcessor
 implements Processor {
 public void process(Exchange exchange)
 throws Exception {
 // do message translation here
 }
}
from("direct:transformOrder")
  .process(new OrderTransformProcessor());
```

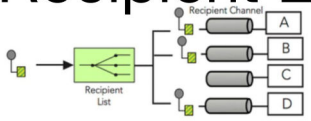
Message Filter EIP



- Camel has support for Message Filter using the filter node

```
from("jms:queue:inbox")
.filter(header("test").isNotEqualTo("true"))
.to("jms:queue:order");
```

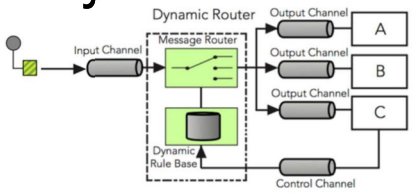
Recipient List



- How to route messages based on a **static** or **dynamic list** of destinations
- Camel has support for Message Filter using the filter node

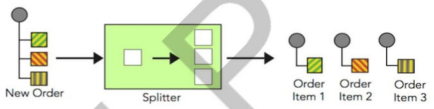
```
from("jms:queue:inbox")
  .multicast().to("file://backup","seda:inbox");

from("seda:confirmMails").beanRef(processMails)
.recipientList("destinations")
```

Dynamic Router




```
from("jms:queue:order")
.dynamicRouter(bean(new MyRouter()));
```

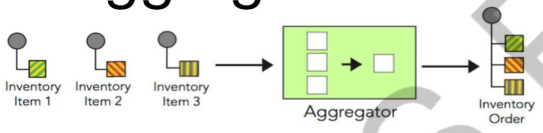
Splitter EIP



- Camel has support for Splitter using the split node

```
from("file://inbox")
.split(body().tokenize("\n"))
.to("seda:orderLines");
```

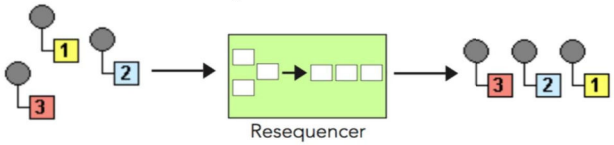
Aggregator EIP



- Camel has support for Aggregator using the aggregator node

```
from("jms:topic:stock:quote")
.aggregate()
  .xpath("/quote/@symbol")
  .batchTimeout(5 * 60 *1000)
  .to("seda:quotes");
```

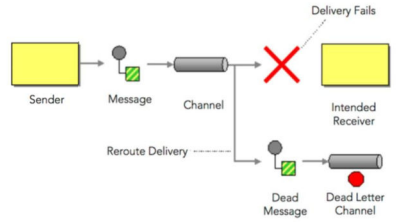
Resequencer EIP



- Camel has support for Resequencer using the resequence node

```
from("jms:topic:stock:quote")
.resequence().xpath("/quote/@symbol")
            .timeout(60 * 1000)
            .to("seda:quotes");
```

Dead Letter Channel EIP

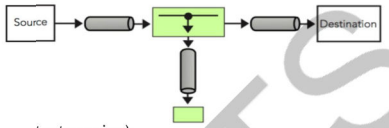



```
errorHandler(
  deadLetterChannel("activemq:error")
    .maximumRedeliveries(5)
    .redeliveryDelay(5000)
);

from("activemq:incoming")
  .marshal().jaxb()
  .to("mq:QueueWithXmlMessages");
```

try .. catch style

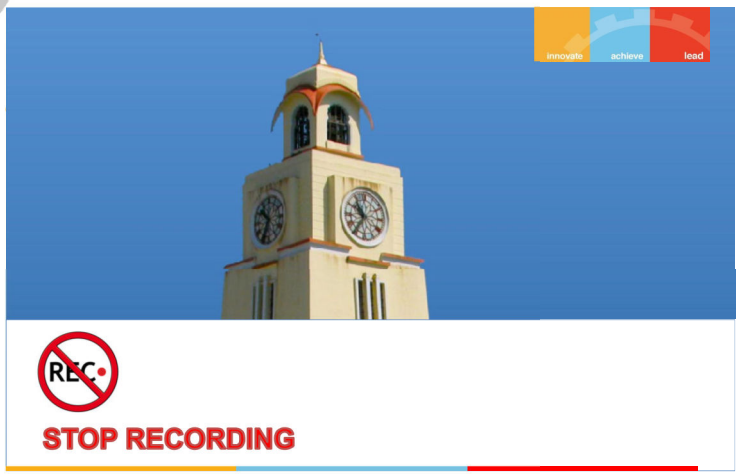
```
from("activemq:incoming")
  .doTry()
    .marshal().jaxb()
    .to("mq:QueueWithXmlMessages")
  .doCatch(Exception.class)
    .to("activemq:error")
  .end();
```

Wiretap



contact session)

```
from("jms:queue:order")
        .wireTap("seda:tappedOrder")
        .to("bean:processOrder");
```



BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

innovate achieve lead

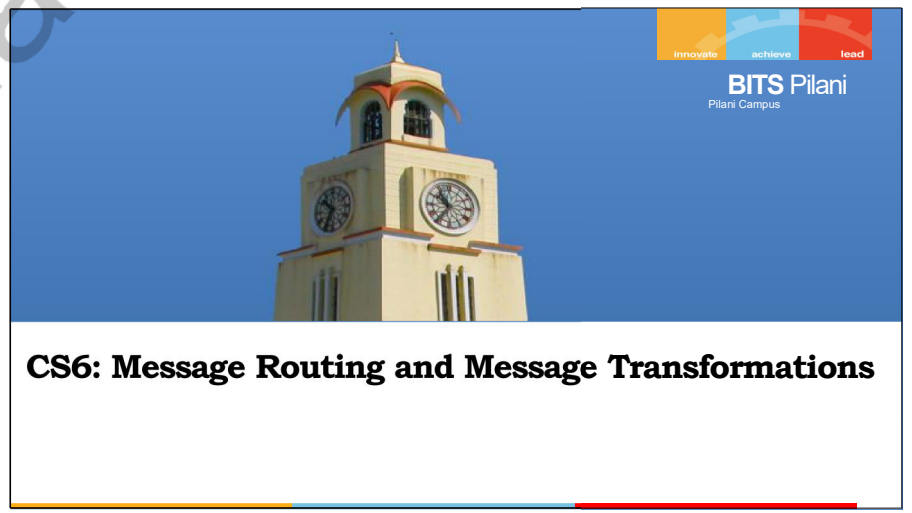
Thank You

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

IMP Note to Students

- It is important to know that just login to the session does not guarantee the attendance.
- Once you join the session, continue till the end to consider you as present in the class.
- IMPORTANTLY, you need to make the class more interactive by responding to Professors queries in the session.
- **Whenever Professor calls your number / name ,you need to respond, otherwise it will be considered as ABSENT**



CS6: Message Routing and Message Transformations

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Agenda

- Message Routers
- Message Transformations

Message Routers

Below are list of routers types available

- Content-Based Router
- Message Filter
- Dynamic Router
- Recipient List
- Splitter
- Aggregator
- Re sequencer
- Composed Message Processor
- Scatter-Gather
- Routing Slip
- Process Manager
- Message Broker

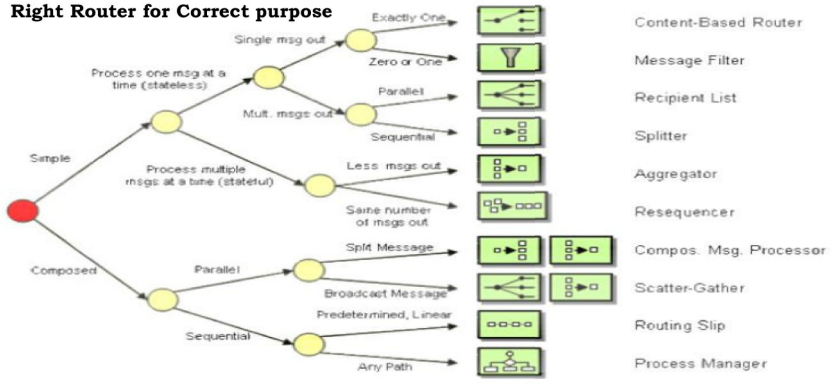
Message Routers

- **Content and Dynamic Router**
- Content Routers inspects the content of the message and passes on the message to successive stage if the content of the message is correct. A message filter filters out messages with wrong content.
- Usage of Dynamic routers with Content Routers improves the flexibility of transmission Dynamic Routers allows Routing logic to be varied during transmission A recipient list is a Content Router which can send the information to more then one destination.

- **Composed Router**
- Combines multiple Message Router Variants to obtain more comprehensive Solutions.
- Routing Slip specifies the path the message should take.
- Message Broker Architectural Pattern can be used to obtain solution to architect Message Routers.

Message Routers

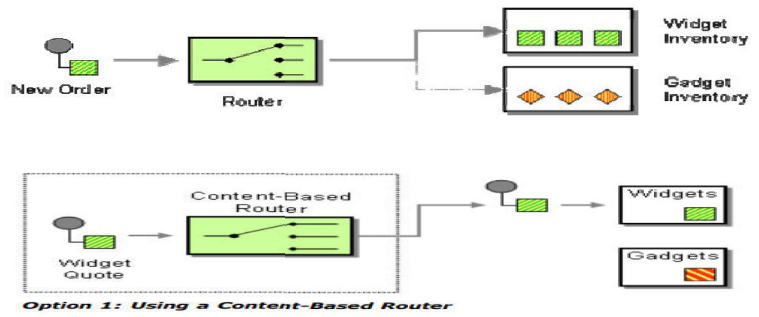
- **Right Router for Correct purpose**



innovate achieve lead

Message Routers

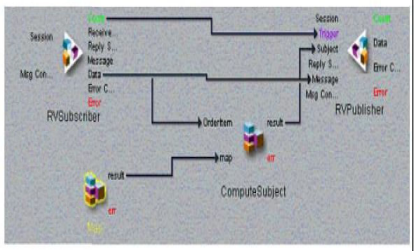
- **Content Based Router**



Option 1: Using a Content-Based Router

Message Routers - Publishing and Subscribing to TIBCO Rendezvous Messages

- Rendezvous programs use messages as the common currency to exchange data. The Rendezvous palette includes activities that allow you to easily setup subjects to send and receive messages. This sample shows how to use Rendezvous activities
- Prerequisites
- TIBCO Rendezvous must be running on the machine
- To run this sample you can either use TIBCO ActiveMatrix BusinessWorks 6.x client or TIBCO ActiveMatrix BusinessWorks 5.x as a client
- Refer to Configure the Client in TIBCO Designer section to configure client in TIBCO ActiveMatrix BusinessWorks 5.x

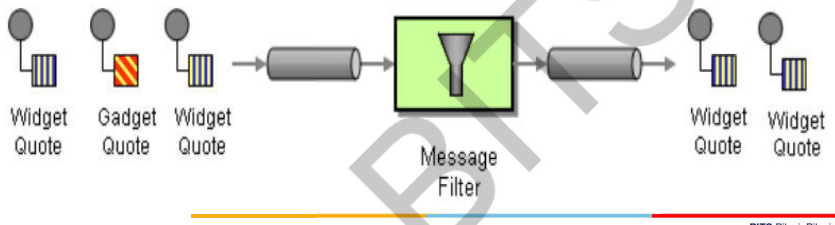


TIBCO Message Broker

Message Routers

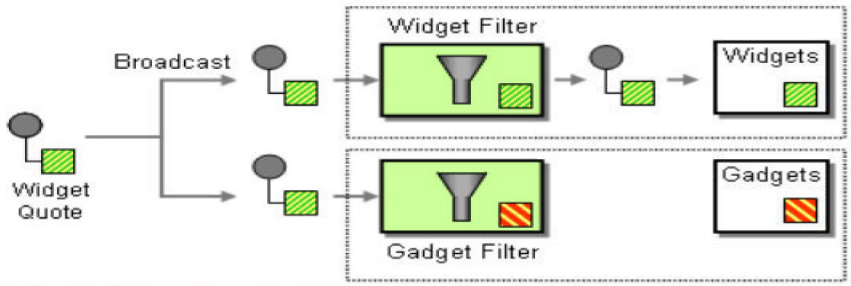
- **Message Filter**

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



Message Routers

- **Message Filter with Broadcast Channel**

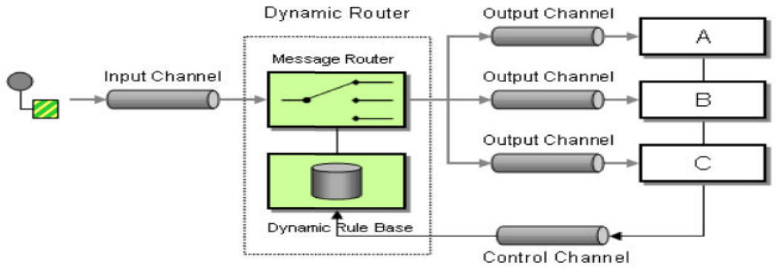


Option 2: Using a broadcast channel and a set of Message Filters

Message Routers

- **Dynamic Router**

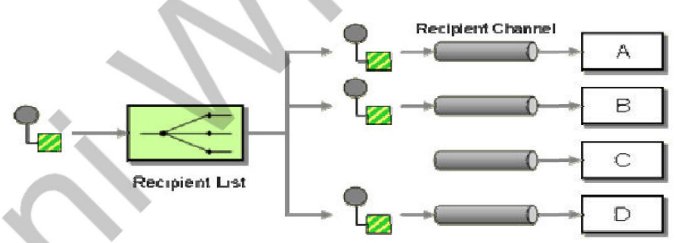
Use a *Dynamic Router*, a Router that can self-configure based on special configuration messages from participating destinations.



Message Routers

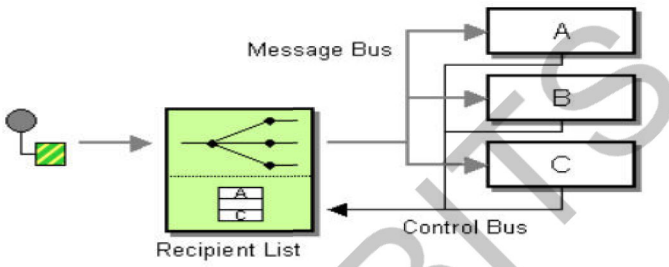
- **Recipient List**

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.



Message Routers

- **Recipient List with Bus**

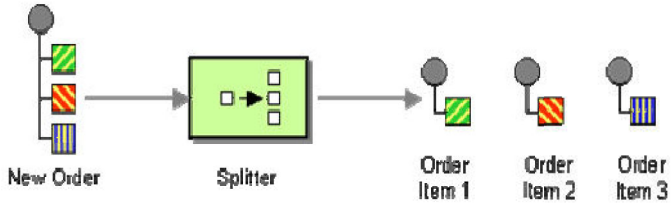


A Dynamic Recipient List Is Configured by the Recipients via a Control Channel

Message Routers

- **Splitter**

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

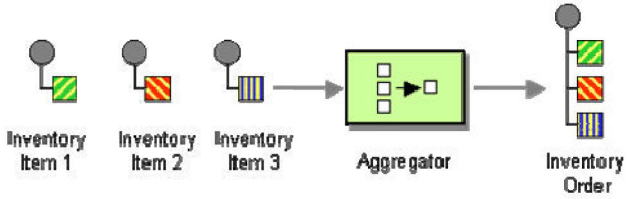


innovate achieve lead

Message Routers

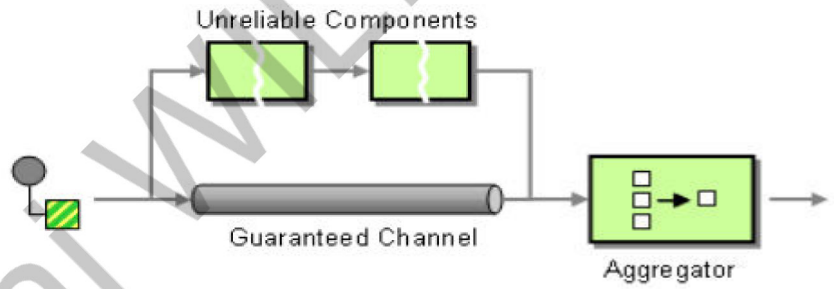
- **Aggregator**

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.



Message Routers

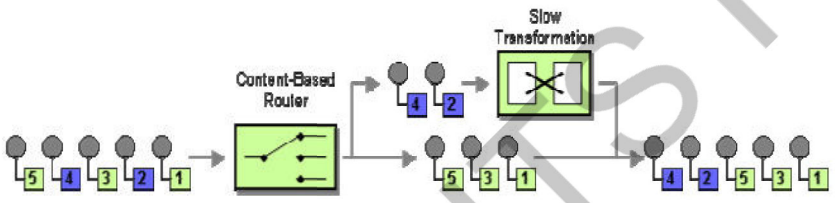
- **Aggregator with channels**



An Aggregator with Time-out Detects Missing Messages

Message Routers

- **Aggregator other options**

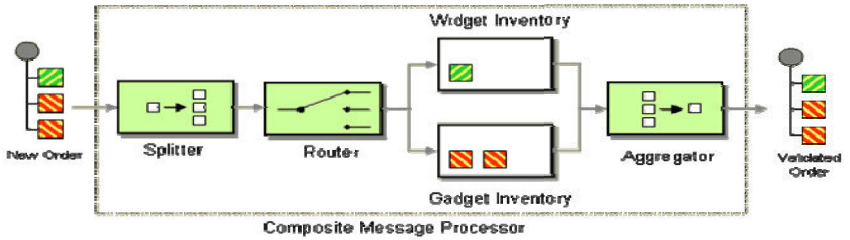


To avoid getting the messages out of order, we could introduce a loop-back (acknowledgment) mechanism that makes sure that only one message at a time passes through the system. The next

Message Routers

- **Composed Message Processor**

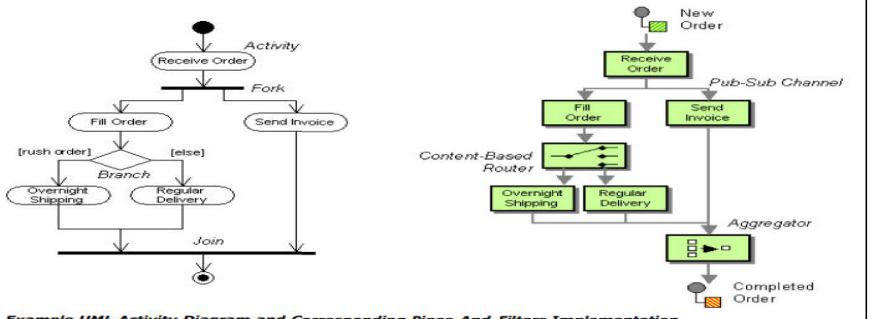
Use *Composed Message Processor* to process a composite message. The *Composed Message Processor* splits the message up, routes the sub-messages to the appropriate destinations and re-aggregates the responses back into a single message.



innovate achieve lead

Message Routers

- **Composed Message Processor**

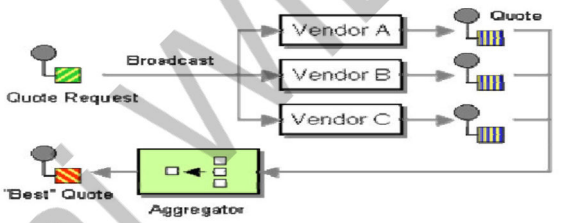


Example UML Activity Diagram and Corresponding Pipes-And-Filters Implementation

Message Routers

- **Scatter – Gather Router**

Use a *Scatter-Gather* that broadcasts a message to multiple recipients and re-aggregates the responses back into a single message.

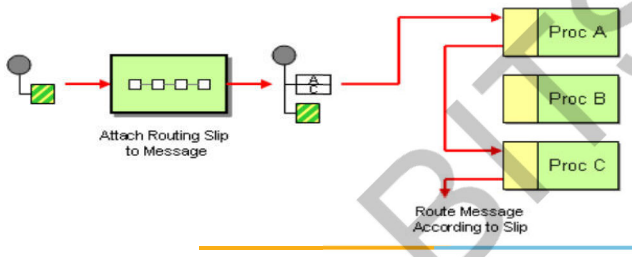


The *Scatter-Gather* routes a request message to the a number of recipients. It then uses an *Aggregator* to collect the responses and distill them into a single response message.

Message Routers

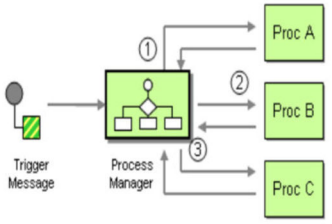
- **Routing Slip**

Attach a *Routing Slip* to each message, specifying the sequence of processing steps. Wrap each component with a special message router that reads the *Routing Slip* and routes the message to the next component in the list.



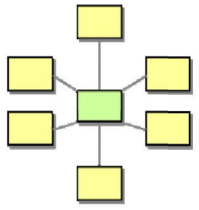
Message Routers

- **Central Process Unit,** a process manager to maintain the state of the sequence and determine the next processing step based on intermediate results



- **Message Broker**

Use a central *Message Broker* that can receive messages from multiple destinations, determine the correct destination and route the message to the correct channel. Implement the internals of the *Message Broker* using the design patterns presented in this chapter.



Using a central *Message Broker* is sometimes referred to as *hub-and-spoke* architectural style, which appears to be a descriptive name when looking at the diagram above.

innovate achieve lead

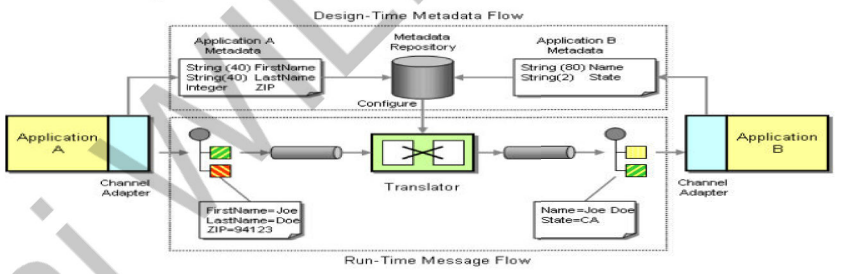
Message Transformations

Below are list of transformations types available

- Envelope Wrapper
- Content Enricher
- Content Filter
- Claim Check
- Normalizer
- Canonical Data Model

Message Transformations

- **Metadata Integration**

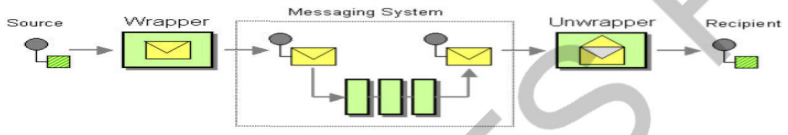


Metadata Integration

For example, the picture above depicts the integration between two applications that need to exchange customer information. Each system has a slightly different definition of customer data. Application A stores first and last name

Message Transformations

- **Wrapping and unwrapping**

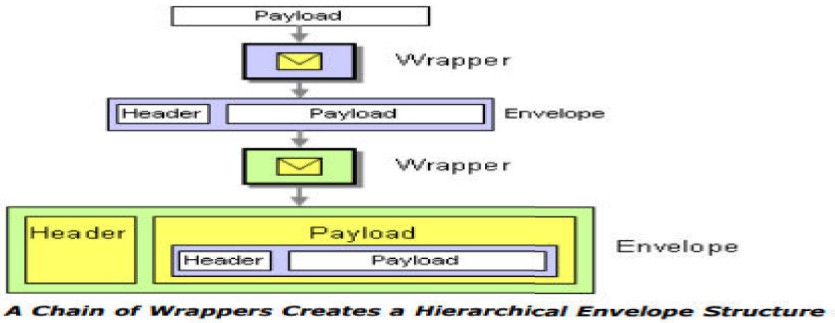


The process of wrapping and unwrapping a message consists of five steps:

- The Message Source publishes a message in a raw format. This format is typically determined by the nature of the application and does not comply with the requirements of the messaging infrastructure.
- The Wrapper takes the raw message and transforms it into a message format that complies with the messaging system. This may include adding message header fields, encrypting the message, adding security credentials etc.
- The Messaging System processes the compliant messages.
- A resulting message is delivered to the Unwrapper. The unwrapper reverses any modifications the wrapper made. This may include removing header fields, decrypting the message or verifying security credentials.
- The Message Recipient receives a 'clear text' message.

Message Transformations

- **Envelope structure**

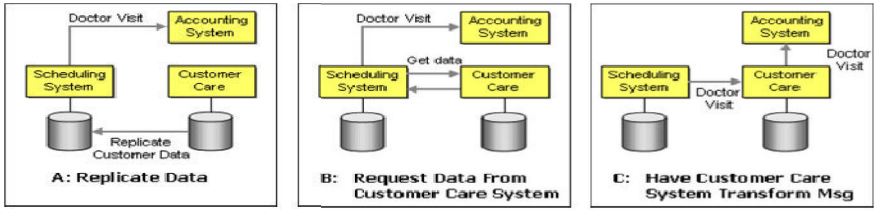


A Chain of Wrappers Creates a Hierarchical Envelope Structure

innovate achieve lead

Message Transformations - Example

Let's consider the following example (see picture). A hospital scheduling system publishes a message announcing that the patient has completed a doctor's visit. The message contain's the patient's first name, his or her patient ID and the date of the visit. In order for the accounting system to log this visit and inform the insurance company, it requires the full patient name, the insurance carrier and the patient's social security number. However, the scheduling system does not store this information, it is contained in the customer care system. What are our options?



Message Transformations - Example

Option A: We could modify the scheduling system so it can store the additional information. When the customer's information changes in the customer care system (e.g. because the patient switches insurance carriers) the changes need to be replicated to the scheduling system. The scheduling system can now send a message that includes all required information. Unfortunately, this approach has two significant drawbacks. First, it requires a modification to the scheduling system's internal structure. In most cases, the scheduling system is going to be a packaged application and may not allow this type of modification. Second, even if the scheduling system is customizable, we need to consider that we are making a change to the system based on a the specific needs of another system. For example, if we also want to send a letter to the patient confirming the visit we would have to change the scheduling system again to accommodate the customer's mailing address. The integration solution would be much more maintainable if we decouple the scheduling system from the specifics of the applications that consume the "Doctor Visit" message.

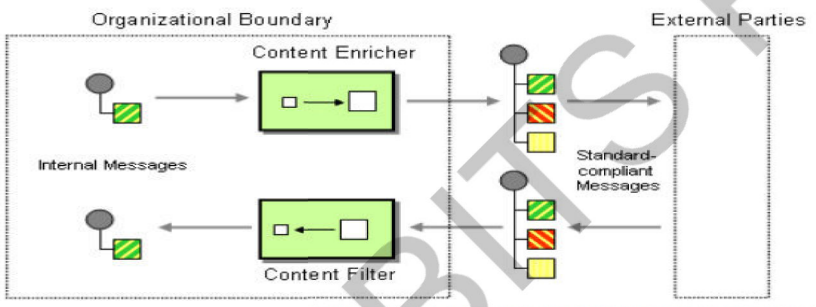
Option B: Instead of storing the customer's information inside the scheduling system, the scheduling system could request the SSN and carrier data from the customer care system just 298 before it is sending the 'Doctor Visit' message. This solves the first problem -- we no longer have to modify the storage of the scheduling system. However, the second problem remains: the scheduling system needs to know that the SSN and carrier information is required in order to notify the accounting system. Therefore, the semantics of the message is more similar to 'Notify Insurance' than 'Doctor Visit'. In a loosely coupled system we do not want one system to instruct the next one on what to do. We rather send an Event Message and let the other systems decide what to do. In addition, this solution couples the scheduling system more tightly to the customer care system because the scheduling system now needs to know where to get the missing data. This ties the scheduling system to both the accounting system and the customer care system. This type of coupling is undesirable because it leads to brittle integration solutions.

Option C: We can avoid some of these dependencies if we send the message to the customer care system first instead of the accounting system. The customer care system can then fetch all the required information and send a message with all required data to the accounting system. This decouples the scheduling system nicely from the subsequent flow of the message. However, now we implement the business rule that a the insurance company receives a bill after the patient visits the doctor inside the customer care system. This requires us to modify the logic inside the customer care system. If the customer care system is a packaged application, this modification may be difficult or impossible. Even if we can make this modification, we now make the customer care system indirectly responsible for sending bills messages. This may not be a problem if all the data items required by the accounting system are available inside the customer care system. If some of the fields have to be retrieved from other systems we are in a similar situation to where we started.

Option b (not shown): We could also modify the accounting system to only require the customer ID and retrieve the SSN and carrier information from the customer care system. This approach has two disadvantages. First, we now couple the accounting system to the customer care system. Second, this option again assumes that we have control over the accounting system. In most cases, the accounting system is going to be a packaged application with limited options for customization.

Message Transformations

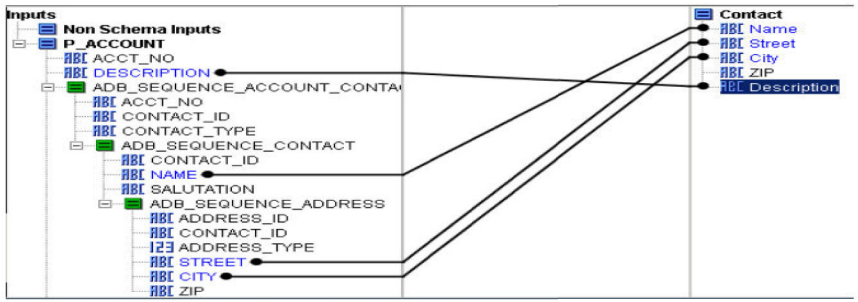
- **Content Enricher**



Using a Content Enricher / Content Filter Pair When Communicating with External Parties

Message Transformations

- **Adapter**

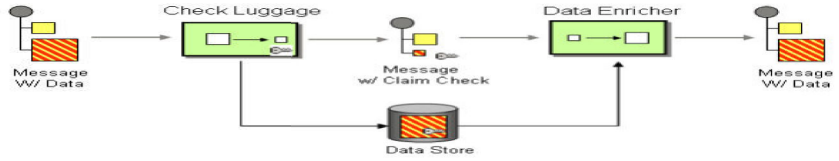


Simplifying a Message Published by a Database Adapter

innovate achieve lead

Message Transformations

- **Claim Check**

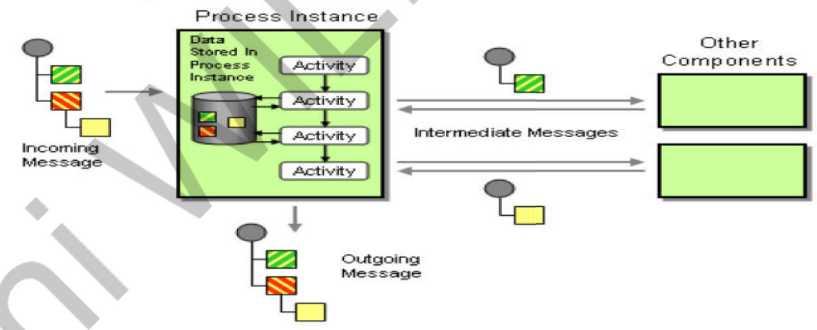


The *Claim Check* pattern consists of the following steps:

1. A message with data arrives.
2. The "Check Luggage" component generates a unique key for the information. This key will be used later as the *Claim Check*
3. The Check Luggage component extracts the data from the message and stores it in a persistent store, e.g. a file or a database. It associates the stored data with the key .
4. It removes the persisted data from the message and adds the *Claim Check*.
5. Another component can use a *Content Enricher* to retrieve the data based on the *Claim Check*.

Message Transformations

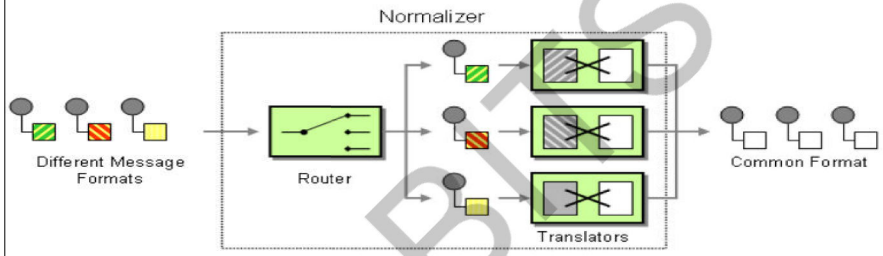
- **Process Manager with Claim Check**



Message Transformations

- **Normalizer with Message Translator**

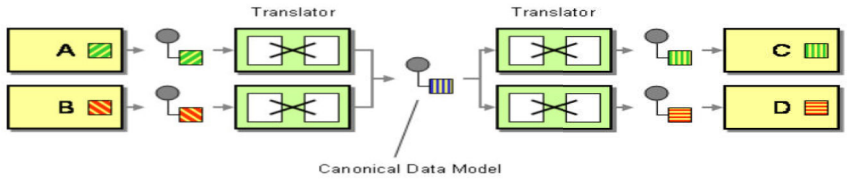
Use a *Normalizer* to route each message type through a custom *Message Translator* so that the resulting messages match a common format.



Message Transformations

- **Canonical Data Model**

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.

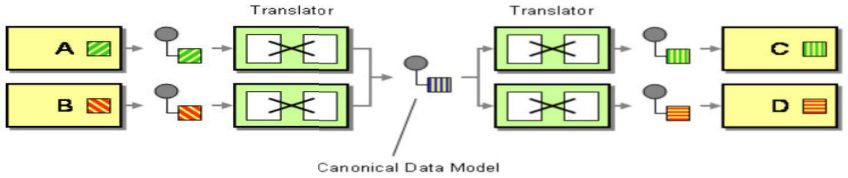
innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Message Transformations

- **Canonical Data Model**

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.

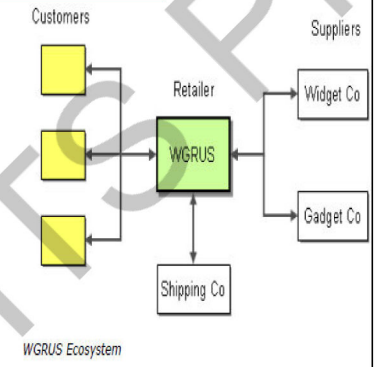


The *Canonical Data Model* provides an additional level of indirection between application's individual data formats. If a new application is added to the integration solution only transformation between the *Canonical Data Model* has to created, independent from the number of applications that already participate.

Case Study

Widget-Gadget Corp -- An Example

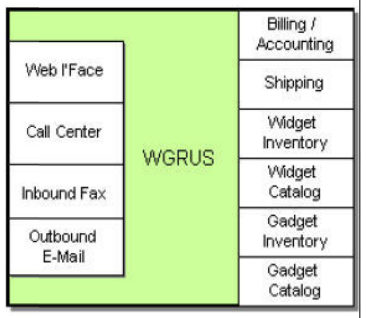
- The best way to understand message-based integration solutions is by walking through a concrete example.
- Let's consider Widgets & Gadgets 'R Us (WGRUS), an on-line retailer that buys widgets and gadgets from manufacturers and resells them to customers.
- The solution needs to support the following requirements.
- **Take Orders:** Customers can place orders via Web, phone or fax
- **Process Orders:** Processing an order involves multiple steps, including verifying inventory, shipping the goods and invoicing the customer
- **Check Status:** Customers can check the order status
- **Change Address:** Customers can use a Web front-end to change their billing and shipping address
- **New Catalog:** The suppliers update their catalog periodically. WGRUS needs to update its pricing and availability based in the new catalogs.



WGRUS Ecosystem

Widget-Gadget Corp -- An Example

- **Announcements:** Customers can subscribe to selective announcements from WGRUS.
- **Testing and Monitoring:** The operations staff needs to be able to monitor all individual components and the message flow between them.
- We will tackle each of these requirements separately and describe the solution alternatives and trade-offs using the pattern language
- Like in most integration scenarios, WGRUS is not a so-called "green field" implementation, but rather the integration of an existing IT infrastructure comprised of a variety of packaged and custom applications.
- The fact that we have to work with existing applications often makes integration work challenging. In our example WGRUS runs the following systems



WGRUS IT Infrastructure

innovate achieve lead

Widget-Gadget Corp – Taking order

- The first function we want to implement is taking orders.
- The first step to streamlining order processing is to unify taking orders.

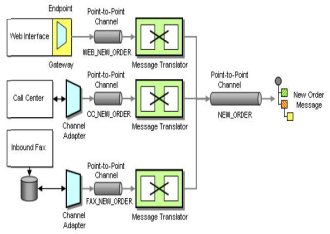
- A customer can place orders over one of three channels:
- Web site, call center or fax.
- Each system is based on a different technology and stores incoming orders in a different data format.
- The call center system is a packaged application while the Web site is a custom J2EE application.
- The inbound fax system requires manual data entry into a small Microsoft Access application.
- We want to treat all orders equally, regardless of their source

- The packaged call center application and the inbound fax application was not developed with integration in mind, we connect it to the messaging system using a Channel Adapter.
- A Channel Adapter is a component that can attach to an application and publish messages to a Message Channel whenever an event occurs inside the application
- Because the Web application is custom-built we implement the Endpoint code inside the application.

Widget-Gadget Corp – Taking order

- Because each system uses a different data format for the incoming orders we use three Message
- Translators to convert the different data formats into a common New Order message that follows a Canonical Data Model.
- A Canonical Data Model defines message formats that are independent from any specific application so that all applications can communicate with each other in this common format

- The NEW_ORDER Message Channel is a Datatype Channel because it carries messages of only one type, i.e. new orders. This makes it easy for message consumers to know what type of message to expect.
- We name application-specific Message Channels starting with the name of the application
- e.g. WEB_NEW_ORDER.
- Channels carrying canonical messages are named after the intent of the message without any prefix,
- e.g. NEW_ORDER.
- We connect each Channel Adapter to the Message Translator via a Point-to-Point Channel because we want to be sure that each order message is consumed only once

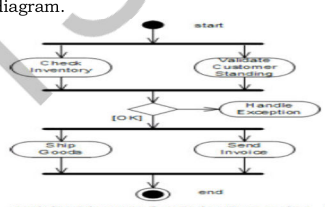


Taking Orders From Three Different Channels

Widget-Gadget Corp – Processing orders

- Now that we have a consistent order message that is independent from the message source we need to process orders.
- In order to fulfill an order we need to complete the following steps:
- Verify the customer's credit standing. If the customer has outstanding bills, we want to reject the new order.
- Verify inventory. We can't fulfill orders for items that are not in stock.
- If the customer is in good standing and we have inventory we want to shop the goods and bill the customer.
- We can express this sequence of events using a UML activity diagram.

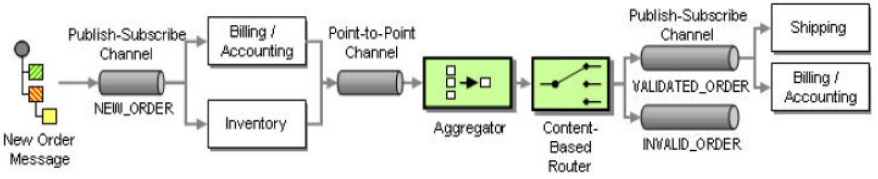
- The activity diagram executes the "Check Inventory" task and the "Verify Customer Standing" task in parallel.
- The join bar waits until both activities are completed before it allows the next activity to start.
- The next activity verifies the results of both steps – do we have inventory and is the customer in good standing?
- If both conditions are fulfilled, the process goes on to fulfill the order.
- Otherwise, we transition to an exception handling activity. For example, we may call the customer to remind them to pay the last invoice or send an e-mail letting him or her know that the order will be delayed.



Activity Diagram for Order Processing

Widget-Gadget Corp – Processing orders

- To convert the logical activity diagram into an integration design, we use a
- Publish-Subscribe Channel to implement the fork action and an Aggregator to implement the join action.
- A Publish-Subscribe Channel sends a message to all active consumers while an Aggregator receives multiple incoming messages and combines them into a single outgoing message

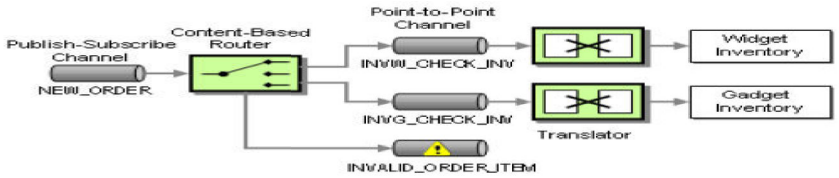


The Aggregator combines the results from both messages and passes the resulting message to a Content-Based Router.

innovate achieve lead

Widget-Gadget Corp – Processing orders

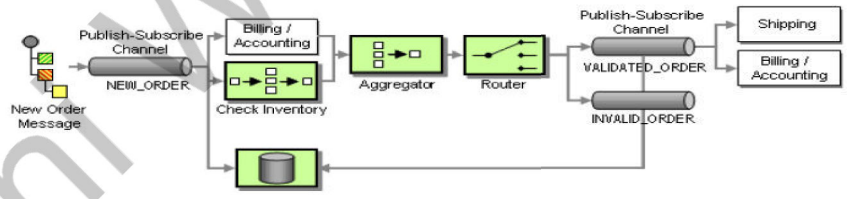
- WGRUS has two inventory systems, one for widgets and one for gadgets.
- As a result, we have to route the request for inventory to the correct system.
- As we want to hide the peculiarities of the inventory systems we insert a Content-Based Router that routes the message to the correct system based on the type of item ordered
- For example, all incoming messages with an item number starting with 'W' are routed to the widget inventory system and all orders with an item number starting with 'G' are routed to the gadget inventory system.



Routing the Inventory Request

Widget-Gadget Corp – Checking Status

- In order to ascertain the status of the order in the sequence of steps we would have to know the "last" message related to this order.
- One of the advantages of a Publish-Subscribe Channel is that we can add additional subscribers without disturbing the flow of messages.
- We can use this property to listen in to new and validated orders and store them into a Message Store.
- We could then query the Message Store database for the status



Adding a Message Store To Track Order Status

Widget-Gadget Corp – Change Address

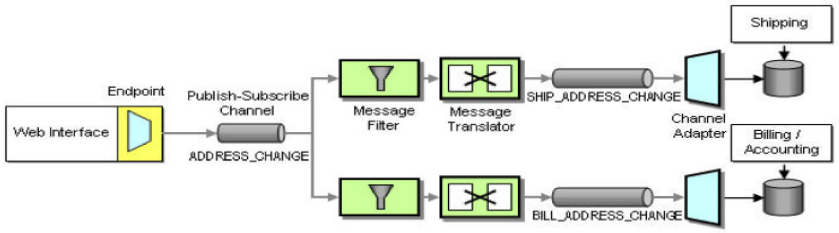
- WGRUS needs to deal with a number of addresses.
- For example, the invoice has to be sent to the customer's billing address while the goods are shipped to the shipping address.
- We want to allow the customer to maintain all these addresses through the Web Interface to eliminate unnecessary manual steps.

- We can choose between two basic approaches to get the correct billing and shipping addresses to the billing and shipping systems:
  - Include address data with the New Order message
  - Replicate address data to other systems

- We update the address directly into the system database using a database Channel Adapter.
- Sending the goods or producing an invoice has to invoke the applications' business logic.
- Because we are dealing with multiple types of addresses (shipping and billing addresses) we need to make sure that only the right type of address is stored in each system.
- We accomplish this by using Message Filters that only pass messages matching certain criteria

Widget-Gadget Corp – Change Address

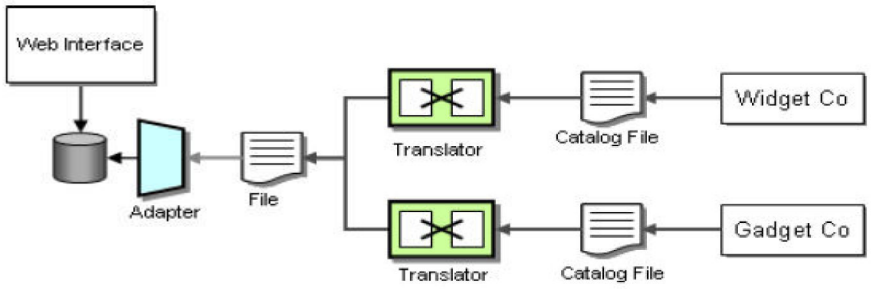
- We use Message Translators to translate the generic Address Change message into the specific message format used by the applications.
- Both the shipping and the billing system store addresses in a relational database so that we use a database Channel Adapter to update the data in each system.



Propagating Address Changes via a Separate Publish-Subscribe Channel

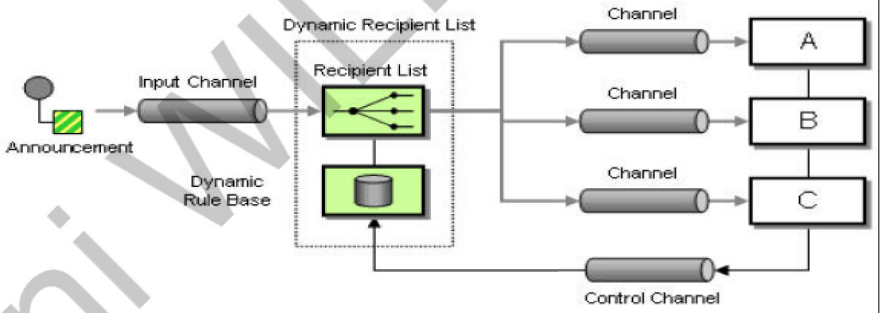
innovate achieve lead

Widget-Gadget Corp – Updating Catalog



Updating Catalog Data via File Transfer

Widget-Gadget Corp – Sending Announcements



Sending Announcements With a Dynamic Recipient List

Thank You

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

BITS Pilani
Pilani Campus

Middleware Technologies- CSIZG524
Contact Session-7 03rd May 2025
EAI: Simple and Composed Messaging

IMP Note to Self

Start Recording

IMP Note to Students

- It is important to know that just login to the session does not guarantee the attendance.
- Once you join the session, continue till the end to consider you as present in the class.
- IMPORTANTLY, you need to make the class more interactive by responding to Professors queries in the session.
- **Whenever Professor calls your number / name ,you need to respond, otherwise it will be considered as ABSENT**

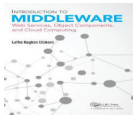
Disclaimer

- *The slides presented here are obtained from the authors of the books and from various other contributors.*
- *I hereby acknowledge all the contributors for their material and inputs.*
- *I have added and modified a few slides to suit the requirements of the course.*

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Textbooks



T1: Letha Hughes Etzkorn - Introduction to middleware _ web services, object components, and cloud computing-Chapman and Hall_CRC (2017).

T2: William Grosso - Java RMI (Designing & Building Distributed Applications)

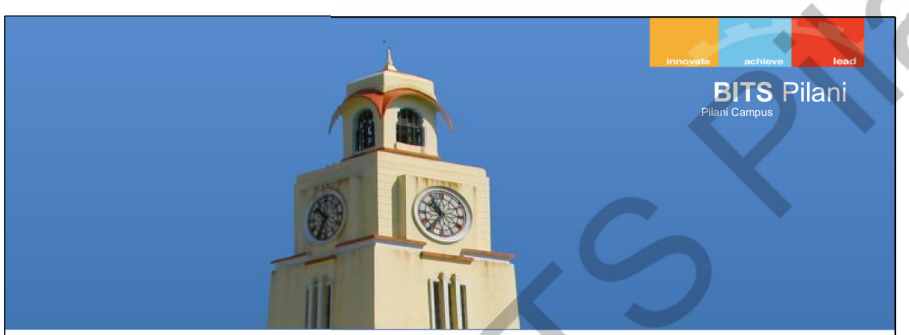
R1: Gregor Hohpe, Bobby Woolf - Enterprise Integration Patterns_ Designing, Building, and Deploying Messaging Solutions -Addison-Wesley Professional (2003)

R2: MongoDB in Action

Note: In order to broaden understanding of concepts as applied to Indian IT industry, students are advised to refer books of their choice and case-studies in their own organizations

Modular Structure

No	Title of the Module
M1	Introduction and Evolution
M2	Enterprise Middleware
M3	Middleware Design and Patterns
M4	Middleware for Web-based Application and Cloud-based Applications
M5	Specialized Middleware



CS7: EAI: Simple and Composed Messaging

Agenda

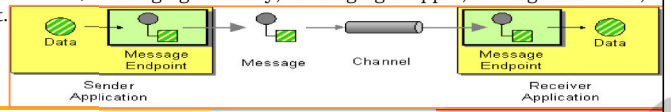
- Messaging Endpoints.
- Simple Messaging.
- Composed Messaging.
- How to compose routing and transformation patterns into a larger solution.
- Overview of Messaging Patterns.

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

Message Endpoints

- 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.
- One must know what request or piece of data to send to another application or is expecting those from another application.
- Messaging endpoint code takes the 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.
- Related patterns: Channel Adapter, Competing Consumers, Durable Subscriber, Event-Driven Consumer, Idempotent Receiver, Message Channel, Message Dispatcher, Selective Consumer, Service Activator, Messaging Gateway, Messaging Mapper, Polling Consumer, Transactional Client.



Integrate Applications using Messaging

- So far, we've introduced a lot of patterns. We've seen the basic Messaging Components, such as Message Channel, Message, and Message Endpoint. We've also seen detailed patterns for Messaging Channels and for Message Construction.
- So how do all of these patterns fit together? How does a developer integrate applications using these patterns? What does the code look like, and how does it work?

- **Simple Messaging with illustrative Patterns**
- This is a simple but powerful example, transmitting a request and transmitting back a reply. It consists of two main classes:
- Requestor — The object that sends the request message and expects to receive the reply message.
- Replier — The object that receives the request message and sends a reply message in response.
- Message Channel and Point-to-Point Channel — One channel for transmitting the requests, another for transmitting the replies.

Simple Messaging with illustrative Patterns

- Document Message — The default type of message, used as both the request and the reply.
- Request-Reply — A pair of messages sent over a pair of channels, allowing the two applications to have a two-way conversation.
- Return Address — The channel to send the response on.
- Correlation Identifier — The ID of the request that caused this response.
- Datatype Channel — All of the messages on each channel should be of the same type.
- Invalid Message Channel — What happens to messages that aren't of the right type.
- Publish/Subscribe — Explores how to use a JMS Topic to implement the Observer pattern.

- **JMS Request-Reply Example**
- jms/RequestQueue — The Queue the Requestor uses to send the request message to the Replier.
- jms/ReplyQueue — The Queue the Replier uses to send the reply message to the Requestor.
- jms/InvalidMessages — The Queue that the Requestor and Replier move a message to when they receive a message that they cannot interpret.
- Here's how the example works. When the Requestor is started in a command-line window, it starts and prints output like this:

```
Sent request
    Time:       1048261736520 ms
    Message ID: ID:_XYZ123_1048261766139_6.2.1.1
    Correl. ID: null
    Reply to:   com.sun.jms.Queue: jms/ReplyQueue
    Contents:   Hello world.
```


JMS Request Reply Example

- When the Replier is started in another command-line window, it starts and prints output like this:

```
Received request
        Time:       1048261766790 ms
        Message ID: ID:_XYZ123_1048261766139_6.2.1.1
        Correl. ID: null
        Reply to:   com.sun.jms.Queue: jms/ReplyQueue
        Contents:   Hello world.
Sent reply
        Time:       1048261766850 ms
        Message ID: ID:_XYZ123_1048261758148_5.2.1.1
        Correl. ID: ID:_XYZ123_1048261766139_6.2.1.1
        Reply to:   null
        Contents:   Hello world.
```

- This shows that the Replier received the request message and sent a reply message.
- Observe the following:
  - MessageID same hence same message.
  - TimeStamp in Receiver is 30270 later.
  - Contents Same.(Hello world.)

innovate achieve lead

JMS Request Reply Java Code.(

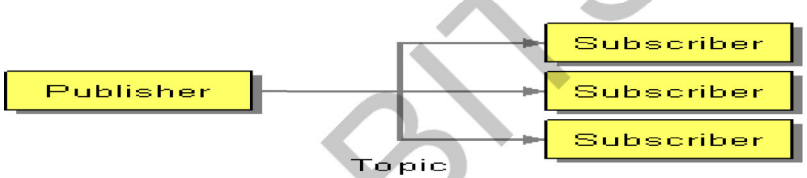
```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class Requestor {

        private Session session;
        private Destination replyQueue;
        private MessageProducer requestProducer;
        private MessageConsumer replyConsumer;
        private MessageProducer invalidProducer;

        protected Requestor() {
                super();
        }

        public static Requestor newRequestor(Connection connection, String requestQueueName,
                String replyQueueName, String invalidQueueName)
                throws JMSException, NamingException {

                Requestor requestor = new Requestor();
                requestor.initialize(connection, requestQueueName, replyQueueName, invalidQueueName);
                return requestor;
        }

        protected void initialize(Connection connection, String requestQueueName,
                String replyQueueName, String invalidQueueName)
                throws NamingException, JMSException {

                session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

                Destination requestQueue = JndiUtil.getDestination(requestQueueName);
                replyQueue = JndiUtil.getDestination(replyQueueName);
                Destination invalidQueue = JndiUtil.getDestination(invalidQueueName);

                requestProducer = session.createProducer(requestQueue);
                replyConsumer = session.createConsumer(replyQueue);
                invalidProducer = session.createProducer(invalidQueue);
        }
```


JMS Request Reply Java

```
    public void send() throws JMSException {
            TextMessage requestMessage = session.createTextMessage();
            requestMessage.setText("Hello world.");
            requestMessage.setJMSReplyTo(replyQueue);
            requestProducer.send(requestMessage);
            System.out.println("Sent request");
            System.out.println("\tTime:          " + System.currentTimeMillis() + " ms");
            System.out.println("\tMessage ID: " + requestMessage.getJMSMessageID());
            System.out.println("\tCorrel. ID: " + requestMessage.getJMSCorrelationID());
            System.out.println("\tReply to:   " + requestMessage.getJMSReplyTo());
            System.out.println("\tContents:   " + requestMessage.getText());
    }

    public void receiveSync() throws JMSException {
            Message msg = replyConsumer.receive();
            if (msg instanceof TextMessage) {
                    TextMessage replyMessage = (TextMessage) msg;
                    System.out.println("Received reply ");
                    System.out.println("\tTime:          " + System.currentTimeMillis() + " ms");
                    System.out.println("\tMessage ID: " + replyMessage.getJMSMessageID());
                    System.out.println("\tCorrel. ID: " + replyMessage.getJMSCorrelationID());
                    System.out.println("\tReply to:   " + replyMessage.getJMSReplyTo());
                    System.out.println("\tContents:   " + replyMessage.getText());
            } else {
                    System.out.println("Invalid message detected");
                    System.out.println("\tType:          " + msg.getClass().getName());
                    System.out.println("\tTime:          " + System.currentTimeMillis() + " ms");
                    System.out.println("\tMessage ID: " + msg.getJMSMessageID());
                    System.out.println("\tCorrel. ID: " + msg.getJMSCorrelationID());
                    System.out.println("\tReply to:   " + msg.getJMSReplyTo());

                    msg.setJMSCorrelationID(msg.getJMSMessageID());
                    invalidProducer.send(msg);

                    System.out.println("Sent to invalid message queue");
                    System.out.println("\tType:          " + msg.getClass().getName());
                    System.out.println("\tTime:          " + System.currentTimeMillis() + " ms");
                    System.out.println("\tMessage ID: " + msg.getJMSMessageID());
                    System.out.println("\tCorrel. ID: " + msg.getJMSCorrelationID());
                    System.out.println("\tReply to:   " + msg.getJMSReplyTo());
            }
    }
```


JMS Publish/Subscriber Example

- This is a simple example that shows the power of publish/subscribe messaging and explores the alternative designs available. It shows how multiple subscriber applications can all be informed of a single event by publishing the event just once and considers alternative strategies for how to communicate details of that event to the subscribers.
- To understand how helpful a simple Publish-Subscribe Channel really is, we first need to consider what it is like to implement the Observer pattern in a distributed fashion, amongst multiple applications. Before we get to that, let's review the basics of Observer.



JMS Publish/Subscriber Observer Pattern

- Observer provides two ways to get the new state from the subject to the observer:
  - the push model and the pull model.
- With the push model, the Update call to each observer contains the new state as a parameter. Thus, interested observers can avoid having to call GetState(), but effort is wasted passing data to uninterested observers.
- The opposite approach is the pull model, where the subject sends basic notification, and each observer requests the new state from the subject. Thus, each observer can request the exact details it wants, even none, but the subject often has to serve multiple requests for the same data.
- The push model requires a single, one-way communication—the Subject pushes the data to an Observer as part of the update.
- The pull model requires three one-way communications—the Subject notifies an Observer, the Observer requests the current state from the Subject, and the Subject sends the current state to the Observer. As we'll see, the number of one-way communications affects both the design-time complexity and the runtime performance of the notification.

innovate achieve lead

JMS Publish/Subscriber Java Code

- A Publish-Subscribe Channel implements the Observer pattern, making the pattern much easier to use amongst distributed applications. The pattern is implemented in three steps:
- The messaging system administrator creates a Publish-Subscribe Channel. (This will be represented in Java applications as a JMS Topic.)
- The application acting as the Subject creates a TopicPublisher (a type of MessageProducer) to send messages on the channel.
- Each of the applications acting as an Observer (e.g., a dependent) creates a TopicSubscriber (a type of MessageConsumer) to receive messages on the channel. (This is analogous to calling the Attach(Observer) method in the Observer pattern.)
- This establishes a connection between the subject and the observers through the channel. Now, whenever the subject has a change to announce, it does so by sending a message. The channel will ensure that each of the observers receives a copy of this message.

JMS Publish/Subscriber Java Code. (Init)

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class SubjectGateway {

        public static final String UPDATE_TOPIC_NAME = "jms/Update";
        private Connection connection;
        private Session session;
        private MessageProducer updateProducer;

        protected SubjectGateway() {
                super();
        }

        public static SubjectGateway newGateway() throws JMSException, NamingException {
                SubjectGateway gateway = new SubjectGateway();
                gateway.initialize();
                return gateway;
        }

        protected void initialize() throws JMSException, NamingException {
                ConnectionFactory connectionFactory = JndiUtil.getQueueConnectionFactory();
                connection = connectionFactory.createConnection();
                session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
                Destination updateTopic = JndiUtil.getDestination(UPDATE_TOPIC_NAME);
                updateProducer = session.createProducer(updateTopic);

                connection.start();
        }

        public void notify(String state) throws JMSException {
                TextMessage message = session.createTextMessage(state);
                updateProducer.send(message);
        }

        public void release() throws JMSException {
                if (connection != null) {
                        connection.stop();
                        connection.close();
                }
        }
}
```


JMS Publish/Subscriber (Receive the change notificaiton)

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class ObserverGateway implements MessageListener {

        public static final String UPDATE_TOPIC_NAME = "jms/Update";
        private Observer observer;
        private Connection connection;
        private MessageConsumer updateConsumer;

        protected ObserverGateway() {
                super();
        }

        public static ObserverGateway newGateway(Observer observer)
                throws JMSException, NamingException {
                ObserverGateway gateway = new ObserverGateway();
                gateway.initialize(observer);
                return gateway;
        }

        protected void initialize(Observer observer) throws JMSException, NamingException {
                this.observer = observer;

                ConnectionFactory connectionFactory = JndiUtil.getQueueConnectionFactory();
                connection = connectionFactory.createConnection();
                Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
                Destination updateTopic = JndiUtil.getDestination(UPDATE_TOPIC_NAME);
                updateConsumer = session.createConsumer(updateTopic);
                updateConsumer.setMessageListener(this);
        }
```


JMS Publish/Subscriber Java Code

- SubjectGateway is a Messaging Gateway between the Subject (not shown) and the messaging system. The subject creates the gateway and then uses it to broadcast notifications.
- Essentially, the subject's Notify() method is implemented to call SubjectGateway.notify(String). The gateway then announces the change by sending a message on the update channel.
- **Observer Gateway**
- Observer Gateway is another Messaging Gateway, this time between the Observer (not shown) and the messaging system. The observer creates the gateway, then uses attach() to start the Connection (which is analogous to calling the Attach(Observer) method in the Observer pattern).
- These two classes implement the push model version of Observer. With the notification message sent by SubjectGateway.notify(String), the existance of the message tells the Observer that a change has occurred, but it is the contents of the message that tell the Observer what the Subject's new state is. The new state is being pushed from the Subject to the Observer. As we'll see later, there's another way to implement all this using the pull model.

innovate achieve lead

JMS Publish/Subscriber Observer Gateway

- The gateway is an Event-Driven Consumer, so it implements the MessageListener interface, which requires the onMessage method. In this way, when an update is received, the gateway processes the message to get the new state, and calls is own update(String) method which calls the corresponding message in the observer.

```
        public void onMessage(Message message) {
                try {
                        TextMessage textMsg = (TextMessage) message; // assume cast always works
                        String newState = textMsg.getText();
                        update(newState);
                } catch (JMSException e) {
                        e.printStackTrace();
                }
        }

        public void attach() throws JMSException {
                connection.start();
        }

        public void detach() throws JMSException {
                if (connection != null) {
                        connection.stop();
                        connection.close();
                }
        }

        private void update(String newState) throws JMSException {
                observer.update(newState);
        }
}
```

Comparisons

- For distributed notification between applications, the Publish-Subscribe (e.g., messaging) approach has several advantages over the traditional, synchronous (e.g., RPC) approach of implementing Observer:
- Simplifies Notification — The Subject's implementation of Notify() becomes incredibly simple; the code just has to send a message on a channel. Likewise, Observer.Update() just has to receive a message.
- Simplifies Attach/Detach — Rather than attach to and detach from the Subject, an Observer needs to subscribe to and unsubscribe from the channel. The Subject does not need to implement Attach(Observer) or Detach(Observer) (although the Observer may implement these methods to encapsulate the subscribe and unsubscribe behavior).
- Simplifies Concurrent Threading — The Subject only needs one thread to update all Observers concurrently—the channel delivers the notification message to the Observers concurrently—and each Observer handles the update in its own thread. This simplifies the Subject's implementation, and because each Observer uses its own thread, what one does in its update thread does not affect the others.

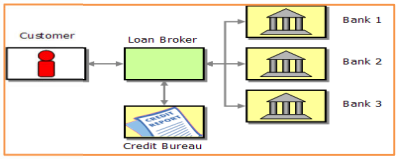
Pub/Sub Advantages

- Publish-Subscribe (e.g., messaging) approach has several advantages over the traditional, synchronous (e.g., RPC) approach of implementing Observer:
- Simplifies Notification — The Subject's implementation of Notify() becomes incredibly simple; the code just has to send a message on a channel. Likewise, Observer.Update() just has to receive a message.
- Simplifies Attach/Detach — Rather than attach to and detach from the Subject, an Observer needs to subscribe to and unsubscribe from the channel. The Subject does not need to implement Attach(Observer) or Detach(Observer) (although the Observer may implement these methods to encapsulate the subscribe and unsubscribe behavior).
- Simplifies Concurrent Threading — The Subject only needs one thread to update all Observers concurrently—the channel delivers the notification message to the Observers concurrently—and each Observer handles the update in its own thread. This simplifies the Subject's implementation, and because each Observer uses its own thread, what one does in its update thread does not affect the others.
- Increases Reliability — Because the channel uses messaging, notifications will be queued until the Observer can process them, which also enables the Observer to throttle the notifications. If an Observer wants to receive notifications that are sent while that Observer is disconnected, it should make itself a Durable Subscriber.

Composed Messaging Example

- Design a loan broker system using integration patterns studied.
- List the individual tasks that the loan broker needs to perform.

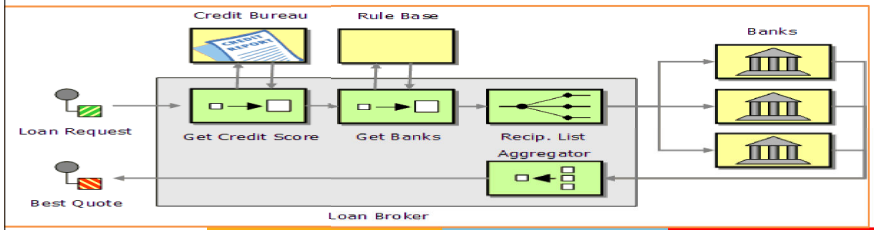
- **Patterns used to implement Loan Broker**
- How Broker receives the incoming request.
- We cover this topic in much more detail in the following chapter on endpoint patterns. So for now, let's skip over this step and assume the message is somehow received by the broker.
- Now, the broker has to retrieve some additional information, i.e. the customer's credit score. A Content Enricher sounds like the ideal choice for this task.
- Once the broker has the complete information, the broker has to determine the appropriate banks to route the request message to. We can accomplish this with another Content Enricher that computes the list of recipients for the request.



innovate achieve lead

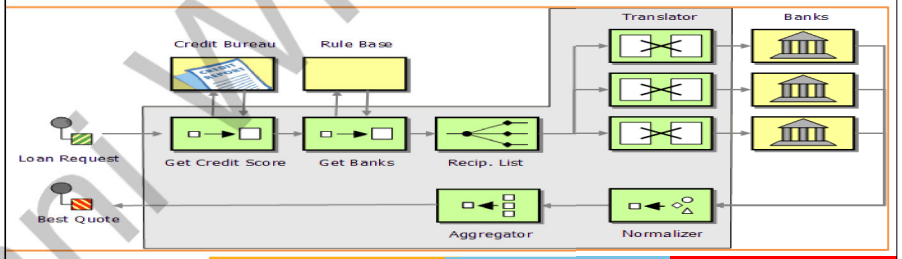
Simple Loan Broker Design

- Sending a request message to multiple recipients and re-combining the responses back into a single message is the specialty of the Scatter-Gather.
- The Scatter-Gather can use a Publish-Subscribe Channel or a Recipient List to send the request to the banks. Once the banks reply with their rate quotes, the Scatter-Gather aggregates the individual rate quotes into a single quote for the consumer using an Aggregator.



Complete Loan Broker Design

- Each bank may use a slightly different message format for the loan request and response. Since we want to separate the routing and aggregation logic from the banks' proprietary formats, we need to insert Message Translators into the communication lines between the broker and the banks. We can use a Normalizer to translate the individual responses into a common format:

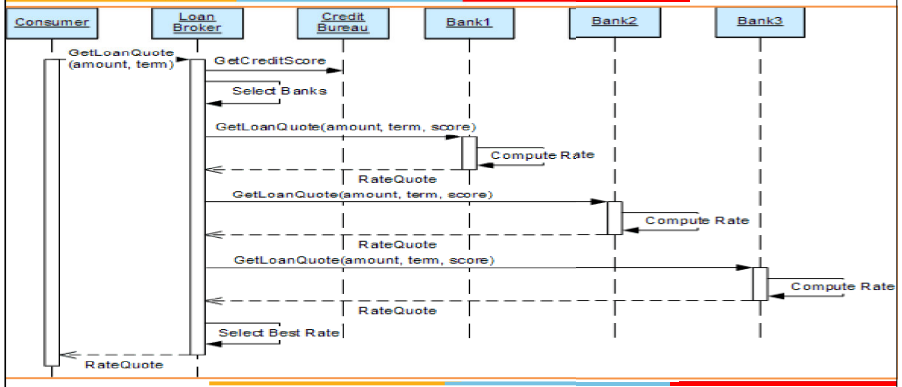


Synchronous vs Asynchronous

- So far, we have described the flow of the messages and the routing and transformation patterns we can use to describe the design of the loan broker component. We have not yet discussed the timing of the broker operation. We have two primary choices:
- Synchronous (Sequential): The broker asks one bank for the quote and waits for a response.
- Asynchronous (Parallel): The broker sends all quote requests at once and waits for the answers to come back.

- **Synchronous, Sequential Processing of Loan Requests**
- UML sequence diagrams to illustrate the two options.
- The synchronous option implies a sequential processing of all loan requests (see picture).
- This solution has the advantage of being simpler to manage because we do not have to deal with any concurrency issues or threads.
- However, it is an inefficient solution because we do not take advantage of the fact that each bank possesses independent processing power and could be executing requests simultaneously. As a result, the consumer might have to wait a long time for an answer..

Synchronous, Sequential Processing of Loan Requests



innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad | Mumbai
WORK INTEGRATED LEARNING PROGRAMMES

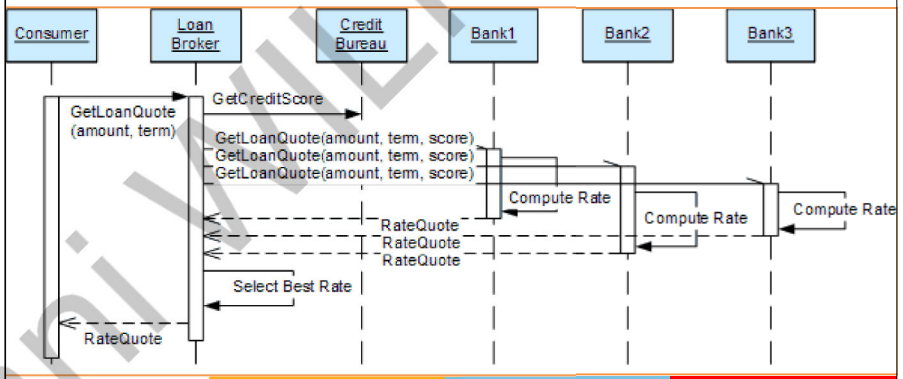
Asynchronous, Parallel Processing of Loan Requests

- Another significant advantage of using asynchronous invocation over a message queue is the ability to create more than one instance of a service.
- What are the observations from the above Sequence Diagram ?
- How can the design be made more performant ?

- For example, if it turns out that the credit bureau is a bottleneck, we could decide to run two instances of the credit bureau component. Because the loan broker sends the request message to a queue instead of directly to the credit bureau component it does not matter which component instance processes the message as long as the response is put back onto the response channel..

- However, it is an inefficient solution because we do not take advantage of the fact that each bank possesses independent processing power and could be executing requests simultaneously. As a result, the consumer might have to wait a long time for an answer..

Asynchronous, Parallel Processing of Loan Requests



Addressing: Distribution vs Auction

- Using a Scatter-Gather to obtain the best quote allows us to choose from two addressing mechanisms, a Recipient List or a Publish-Subscribe Channel. The decision primarily depends on how much control do we want to exert over the banks who are allowed to participate in a specific loan request?

- Again, we have a number of choices:

- Fixed: The list of banks is hard-coded. Each loan request goes to the same set of banks.

- Distribution: The broker maintains criteria on which banks are a good match for a specific request. For example, it would not send a quote request for a customer with a poor credit history to a bank that specializes in premier customers.

Products with Enterprise Integration Patterns

- The patterns are not tied to a specific implementation. They help you design better solutions, whether you use any of the following platforms:

- EAI and SOA platforms, such as IBM WebSphere MQ, TIBCO, Vitria, Oracle Service Bus, WebMethods (now Software AG), Microsoft BizTalk, or Fiorano.

- Open source ESB's like Mule ESB, JBoss Fuse, Open ESB, WSo2, Spring Integration, or Talend ESB

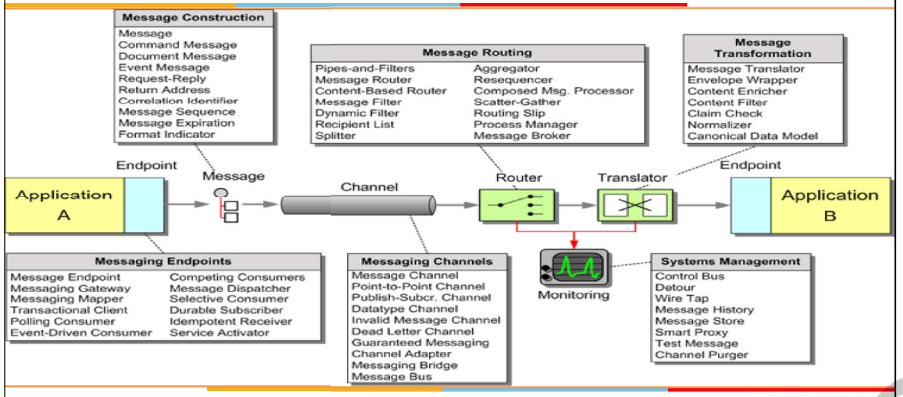
- Message Brokers like ActiveMQ, Apache Kafka, or RabbitMQ

- Web service- or REST-based integration, including Amazon Simple Queue Service (SQS) or Google Cloud Pub/Sub

- JMS-based messaging systems
- Microsoft technologies like MSMQ or Windows Communication Foundation (WCF)

innovate achieve lead

Messaging Overview



Thank You

innovate achieve lead