

# Apache ActiveMQ





## Gesamtinhaltsverzeichnis

<b>1</b>	<b>Messaging .....</b>	<b>7</b>
1.1	Übersicht .....	7
1.2	Prinzip des Messaging .....	8
1.3	Anschauliches Beispiel.....	9
1.3.1	Strenge Kopplung durch ein Telefongespräch:.....	9
1.3.2	Lose Kopplung durch Versenden eines Briefes .....	9
1.4	Begriffe .....	10
1.5	Arten der Nachrichtenverteilung .....	11
1.5.1	Publish-Subscribe .....	11
1.5.2	Point-to-Point .....	11
<b>2</b>	<b>Message oriented middleware .....</b>	<b>12</b>
2.1	Überblick .....	12
2.2	Von Pipes zum Bus .....	13
2.2.1	Inter-Prozess-Kommunikation.....	13
2.2.2	Standardisierte Kommunikationsprotokolle.....	14
2.2.3	Services .....	14
2.2.4	Quality of Service.....	17
2.2.5	Entkopplung.....	19
2.2.6	Workflows .....	21
2.3	Messaging Pattern.....	22
2.3.1	Message Channel .....	22
2.3.2	Message .....	22
2.3.3	Pipes.....	22
2.3.4	Message Router .....	23
2.3.5	Message Translator .....	24
2.3.6	Message Endpoint .....	25
2.3.7	Message Channels Typen .....	25
2.3.8	Datatype Channel .....	25
2.3.9	Invalid Message Channel und Dead Letter Channel.....	26
2.3.10	Guaranteed Delivery .....	27
2.3.11	Channel Adapter .....	27
2.3.12	Message Bridge .....	27
2.3.13	Message Bus .....	28

---

2.3.14	Document Message .....	28
2.3.15	Command Message .....	29
2.3.16	Event Message .....	30
2.3.17	Return Address .....	30
2.3.18	Request-Reply .....	30
2.3.19	Correlation Identifier .....	30
2.3.20	Message Sequence .....	31
2.3.21	Message Expiration .....	31
2.3.22	Splitter .....	31
2.3.23	Aggregator .....	31
2.3.24	Resequencer .....	32
2.3.25	Composed Message Processor .....	32
2.3.26	Splitter-Gather .....	32
2.3.27	Message Transformation .....	33
2.3.28	Messaging Endpoints .....	34
2.3.29	Patterns für das System Management .....	35
<b>3</b>	<b>Java Messaging Service .....</b>	<b>37</b>
3.1	Aufbau der Verbindung .....	37
3.1.1	Message Consumer .....	38
3.1.2	Message Listener .....	40
3.1.3	Message Producer .....	40
3.1.4	Messages .....	41
3.2	Beispiele .....	43
3.2.1	Senden und Empfangen durch eine Queue .....	45
3.2.2	Publizieren und Empfangen über ein Topic .....	46
3.3	Message-Selektoren .....	48
3.3.1	Einsatz .....	48
3.3.2	Literale und Bezeichner .....	49
3.3.3	Ausdrücke .....	49
3.3.4	Beispiele .....	51
3.4	Weitergehende Themen .....	52
3.4.1	Bestätigung des Empfangs .....	52
3.4.2	Persistenz von Messages .....	52
3.4.3	Message-Prioritäten .....	53
3.4.4	Ablaufdatum einer Message .....	53
3.4.5	Antwort-Destinations .....	53
3.4.6	Temporäre Destinations .....	53
3.4.7	Beständige Subscriptions .....	53
3.4.8	Transaktionen .....	54

<b>4</b>	<b>Einführung in ActiveMQ .....</b>	<b>55</b>
4.1	Übersicht .....	55
4.2	Die Dokumentation .....	55
4.3	Literatur .....	57

# 1 Messaging

## 1.1 Übersicht

Neben der interfacebasierten Kopplung von Programmen hat sich besonders auf Großrechnerseite schon seit Langem eine lose Kopplung über sogenannte Messages etabliert. Statt ein entferntes Objekt durch einen Lookup über einen Naming-Services direkt zu referenzieren, wird ein Versenden von Messages über einen Message-Provider durchgeführt. Die Java 2 Enterprise Edition enthält bereits seit langem die Version 1.0 des Java Messaging System (JMS) durch das Paket `javax.jms`. Ab J2EE 1.3 ist jeder J2EE-konforme Applikationsserver notwendigerweise auch ein Message-Provider. Dass die Versionsnummer immer noch 1.0 ist, spricht im Übrigen klar für das Design.

Messaging ist eine zentrale Technologien für die Enterprise Application Integration: Messaging Systeme sind im Unternehmen schon seit mindestens Anfang der Neunziger Jahre erfolgreich im Einsatz. Durch die lose Kopplung ist auch sofort eine Sprachunabhängigkeit gegeben: Eine Messaging System transportiert Nachrichten an heterogene Applikationen.

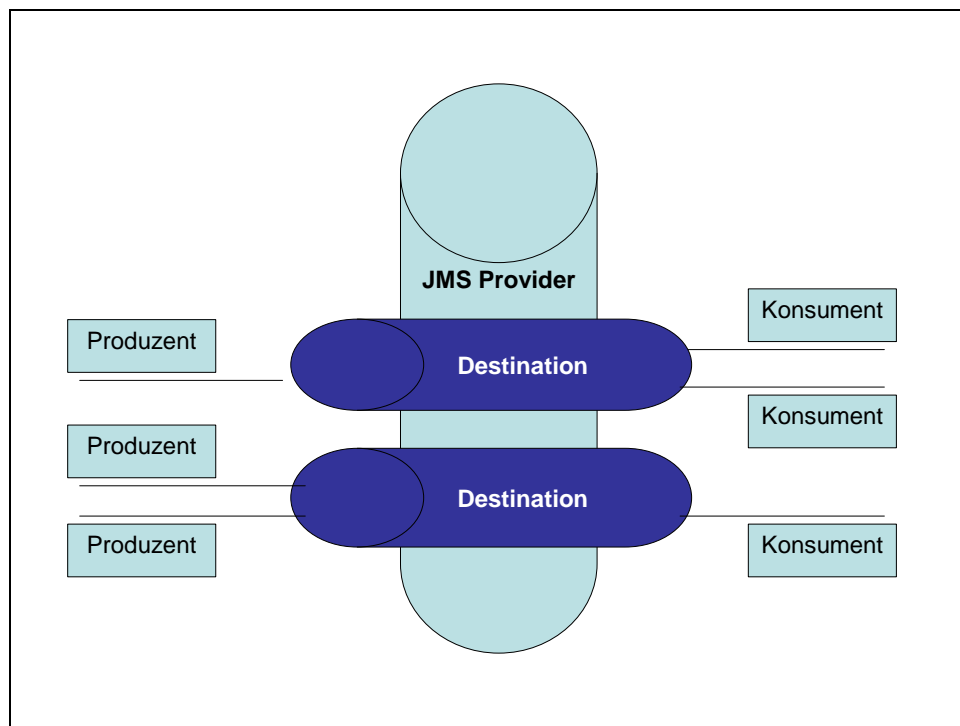
## 1.2 Prinzip des Messaging

Wie bereits angemerkt entkoppelt Messaging Anwendungen. In traditionellen Client-Server-Systemen kommuniziert ein Client direkt mit dem ihm zugeordneten Server Objekt, in dem eine entfernte Referenz darauf benutzt wird. Die Aufrufe sind synchron, d.h., der aufrufende Thread des Clients blockiert, bis die Antwort vom Server komplett eingetroffen ist. Beim Messaging werden jedoch Nachrichten an einen JMS Provider gesendet. Dieser ist dafür verantwortlich:

- Den Empfang der Nachrichten möglichst schnell durchzuführen. Damit erfolgt der Messaging-Aufruf für den Client asynchron.
- Die Nachrichten müssen natürlich auch weitergeleitet werden. Der JMS Provider übernimmt im Gegensatz zum Applikationsserver nicht die Ausführung von Business Logik, sondern sendet die Nachrichten an eine verarbeitende Instanz weiter.

Aus dem Client wird damit ein Produzent von Nachrichten, aus dem Server ein Konsument.

Der JMS-Provider tritt für Produzenten und Konsumenten jedoch nicht als einzelner Block auf, sondern feiner granuliert in einzelne Message Destinations. Konsument und Produzent registrieren sich unabhängig voneinander bei einer Destination, den Transport der Nachrichten übernimmt der Provider.





### 1.3 Anschauliches Beispiel

Um die Unterschiede beider Varianten herausarbeiten zu können, sei ein einfaches Beispiel aus dem Alltagsleben angeführt: Sie benötigen eine Information von einer Behörde:

#### 1.3.1 Strenge Kopplung durch ein Telefongespräch:

Sie tätigen einen Anruf (Netzwerk: Telefonleitung) und verlangen, einen Sachbearbeiter zu sprechen (Lookup im NamingService). Sie warten (blockierender Methodenaufwurf), bis Sie einen Sachbearbeiter vermittelt bekommen und sprechen mit Ihm die benötigten Dinge durch (Aufruf entfernter Methoden). Während des Telefongesprächs können Sie keine anderen Dinge erledigen, das Gleiche gilt für den Sachbearbeiter.

#### 1.3.2 Lose Kopplung durch Versenden eines Briefes

Sie füllen ein Formular aus und senden dies an die Behörde (Netzwerk: Post). Sie werfen den Brief ein und machen Ihre normalen Tätigkeiten aus, ohne auf das Antwortschreiben zu warten (Asynchrone Verarbeitung). Ihr Brief wird von der Post an den Adressaten versendet (Message-Provider: Post), der den Brief entweder direkt entgegen nimmt oder turnusmäßig seinen Posteingang prüft. Ob und wie der Brief bearbeitet wird, bleibt für den Sender verborgen. Ob ein Sachbearbeiter den Brief liest, ein Scanner per Texterkennungssoftware die Bearbeitung automatisiert oder eine Kombination aus mehreren Vorgängen durch Weiterleiten des Briefes erfolgt, bleibt für den Sender verborgen. Die gewünschten Informationen können dann wiederum an die Absenderadresse zurückgesendet werden.

Die Vor- und Nachteile der gewählten Variante sind evident: Während im ersten Fall die benötigte Information dringend sofort benötigt wird und alle anderen Aktionen warten müssen, ist im zweiten Falle eine asynchrone Verarbeitung möglich.

## 1.4 Begriffe

Messaging führt eine ganze Reihe neuer Begriffe ein, die in der folgenden Übersicht kurz vorgestellt werden:

Message	Eine Message besteht aus einem Header, einem Body mit Daten und einer Möglichkeit, den Empfang zu bestätigen. Der Header enthält beispielsweise das Ziel der Nachricht und eventuell eine Weiterleitungs- oder Antwortadresse.
MessageConsumer	Ein Konsument von Messages. Der Message-Provider ruft entweder den MessageConsumer direkt auf oder einen registrierten <code>MessageListener</code> .
MessageProducer	Ein Produzent von Messages
Session	Im Rahmen einer Session werden Consumer und Producer generiert
MessageListener	Das <code>MessageListener</code> -Interface deklariert die Methode: <code>onMessage(Message)</code> . Eine eintreffende Nachricht wird vom Java Messaging System an diese Methode delegiert, falls der Listener registriert wurde.
Topic	Ein Thema, zu dem der Produzent Nachrichten erzeugt, jeder interessierte Konsument wird vom Eintreffen einer Nachricht zu diesem Thema informiert. Im Rahmen der JMS-Terminologie spricht man dann statt Produzent und Konsument von <code>TopicPublisher</code> und <code>TopicSubscriber</code>
Queue	Eine Queue definiert eine Kette von Nachrichten, die an jeweils einen einzigen Empfänger geleitet, der die Nachricht entweder bestätigt oder zurück auf die Queue stellen kann. Eine Message innerhalb einer Queue wird also von exakt einem Empfänger verarbeitet. JMS spricht von <code>QueueSender</code> und <code>QueueReceiver</code>
MessageSelektor	Ein Message-Selektor liefert einen logischen Ausdruck, der bei der Anmeldung eines Konsumenten beim Message-Provider hinterlegt werden kann. Nur falls dieser Ausdruck erfüllt ist, wird die Nachricht an den Konsumenten geliefert.

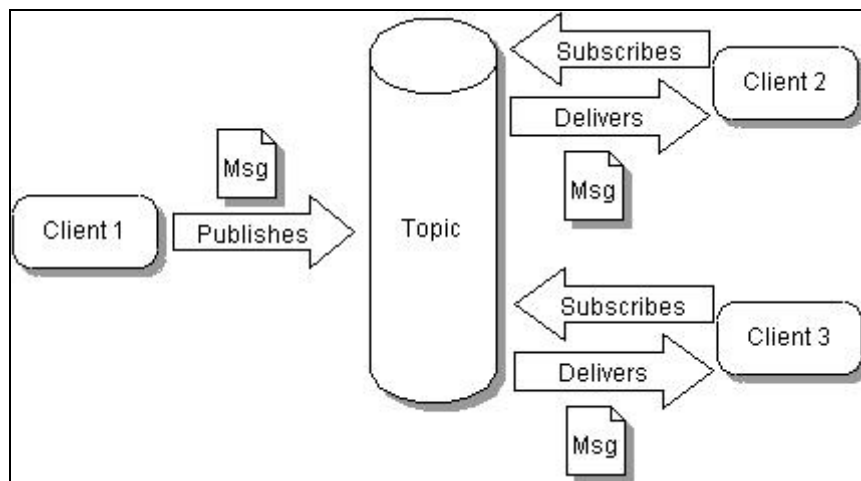
Der Message-Provider muss das gesamte Message-System implementieren. Hierzu muss ein Mechanismus zum Transport von Messages bereitgestellt werden. Interessenten an einer Message müssen sicher beim Eintreffen von Nachricht informiert werden.

## 1.5 Arten der Nachrichtenverteilung

Es gibt nun zwei unterschiedliche Arten, wie der JMS-Provider Nachrichten an die registrierten Konsumenten übermitteln: Bei Publish-Subscribe bekommen alle Konsumenten eine Kopie der Nachricht übermittelt, bei Point-to-Point bekommt exakt ein Konsument die Nachricht übermittelt.

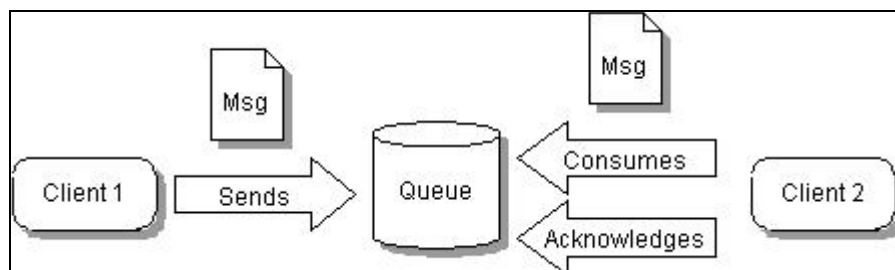
### 1.5.1 Publish-Subscribe

Mit Hilfe von Publish-Subscribe ist es möglich, Nachrichten an mehrere Konsumenten gleichzeitig zu senden. Die Kopplung von Publisher und Subscriber erfolgt durch ein Topic:



### 1.5.2 Point-to-Point

Bei Point-To-Point-Messaging sind Sender und Empfänger dadurch gekoppelt, dass jede Message nur zu exakt einem Empfänger gesendet wird. Die Messages sind in einer Queue vorhanden:



## 2 Message oriented middleware

### 2.1 Überblick

Eine Message oriented middleware (MOM) benutzt die im vorherigen Kapitel eingeführte Messaging-Technologie zur komfortablen Umsetzung komplexer Fachvorgaben.

Sehr wichtig hierbei sind:

- Ausfallsicherheit:
  - Steht ein Server nicht zur Verfügung, so soll die MOM diesen Fehlersituation kompensieren oder zumindest detailliert protokollieren.

Überwachung:

- Anzahl der Requests, Antwortzeiten usw. sollen einfach überwacht werden können.

Workflows:

Dies ist ein Bestandteil der Anwendungs-Programmierung. Der Fokus liegt hierauf der Realisierung einer flexiblen Sequenz von Verarbeitungsschritten mit (teilweiser) Parallelverarbeitung mit Hilfe lose gekoppelter Systeme.

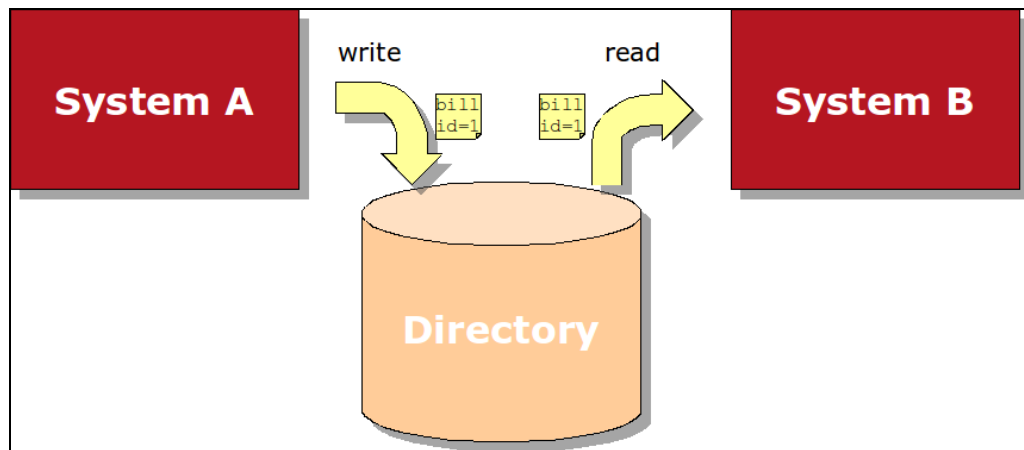
## 2.2 Von Pipes zum Bus

### 2.2.1 Inter-Prozess-Kommunikation

Sobald Prozesse auf gemeinsame Ressourcen zugreifen können, ist eine Kommunikation der Systeme möglich.

- Pipes
- Dateisystem
- Zwischenablage des Betriebssystems
- Socket-Verbindungen

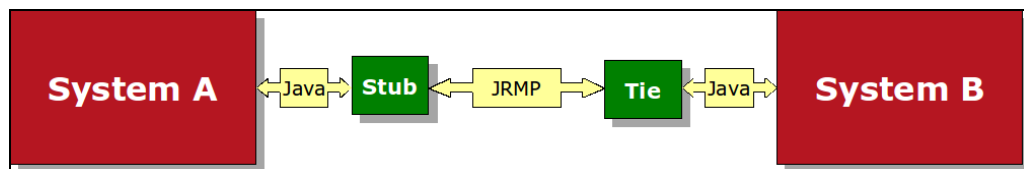
Notwendig ist die Definition eines gemeinsamen Kommunikationsprotokolls.



### 2.2.2 Standardisierte Kommunikationsprotokolle

Durch Standards wird die Anwendung um die Details der Kommunikation entlastet.

- IIOP (CORBA)
- Java-RMI
- DCOM
- Messaging
- SOAP
- ...



### 2.2.3 Services

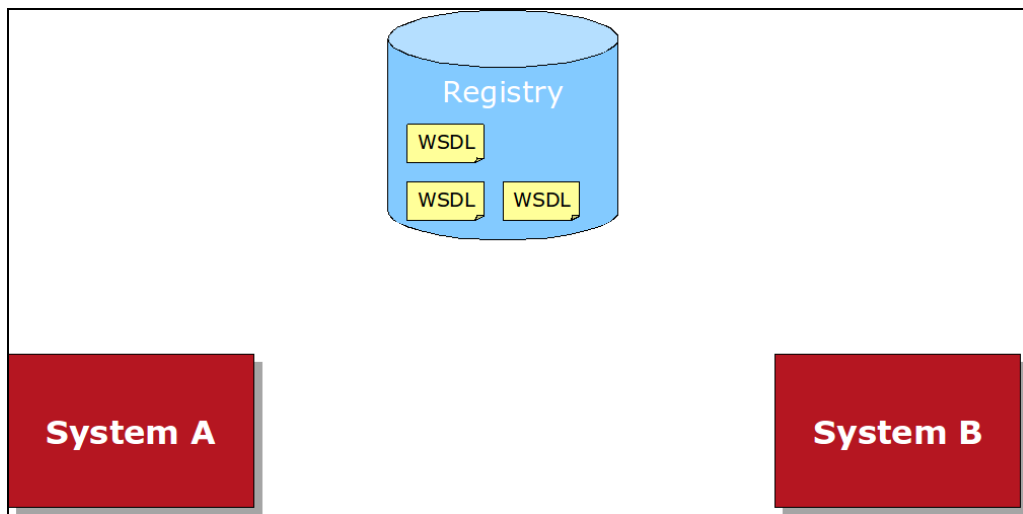
Der Zugriff auf das zu integrierende System erfolgt durch eine wohl-definierte Service-Beschreibung. Die eigentliche Implementierung wird darüber gekapselt.

Eine Service Registry hält alle vorhandenen Service-Beschreibungen. Zum Zugriff auf den Service genügt ein logischer Alias-Name.

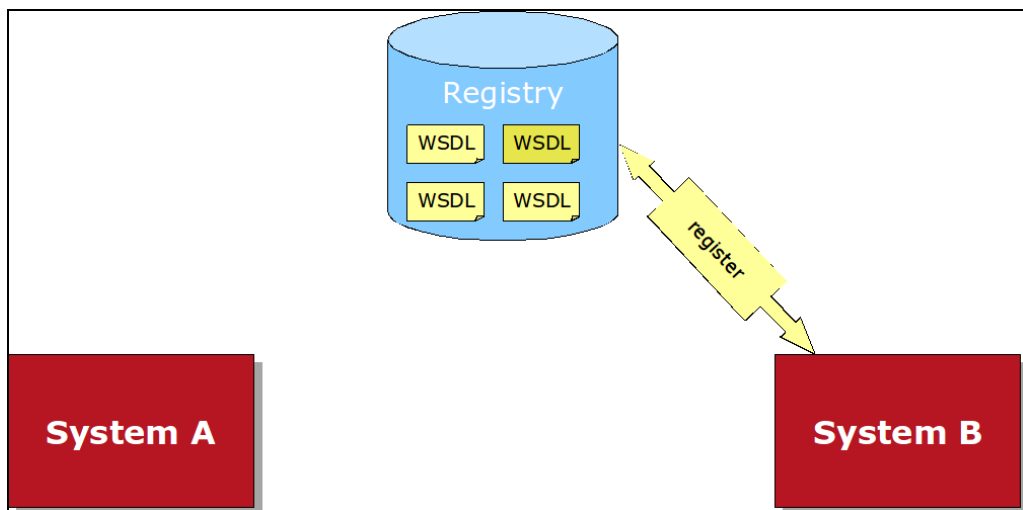
Ausgangspunkt: Zwei entkoppelte Systeme:



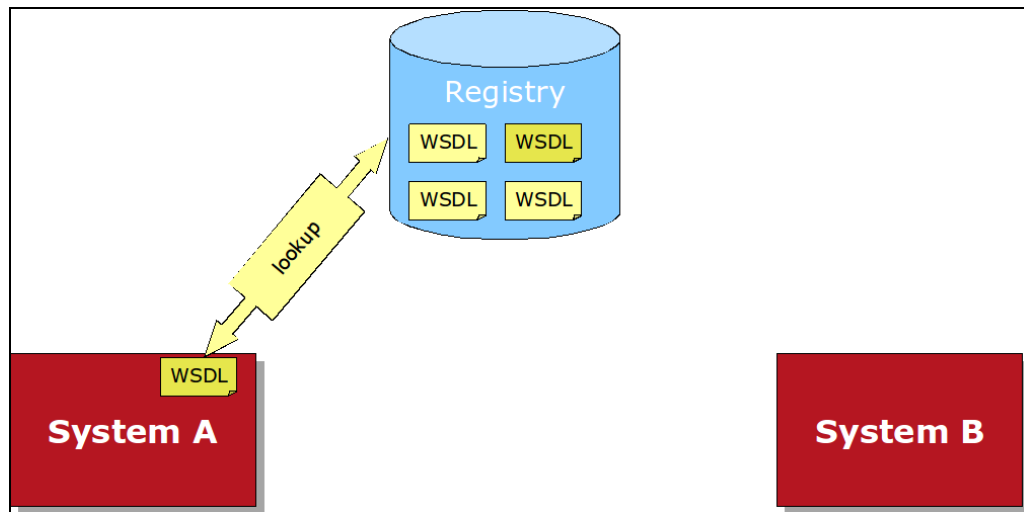
Eine Registry enthält Service-Beschreibungen, hier in Form von WSDL-Dokumenten:



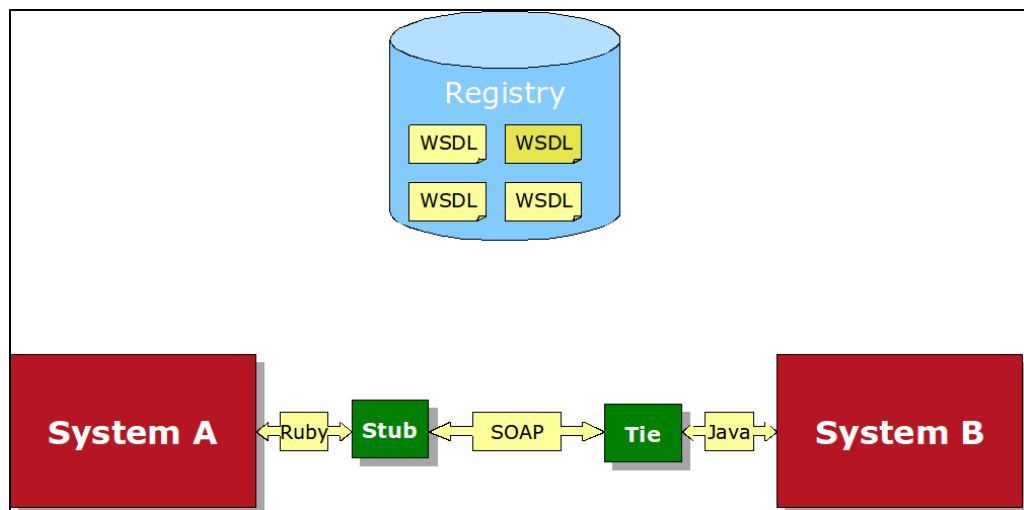
System B registriert seine Services bei der Registry:



System A holt sich die benötigte Service-Beschreibung:



Nun können in System A und B eine Reihe von Hilfs-Klassen erzeugt werden, die die Kommunikation übernehmen:





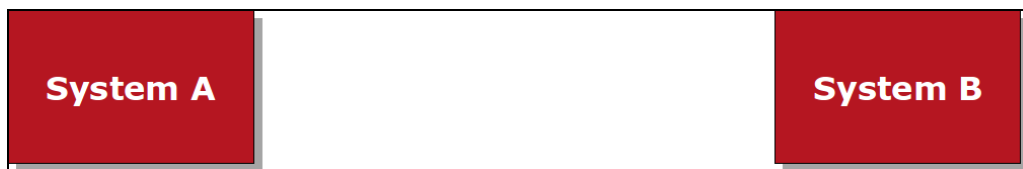
### 2.2.4 Quality of Service

- Garantierte Nachrichten-Übermittlung.
- Nachvollziehbarkeit.
- Failover.
- Skalierbarkeit durch Clustering.
- Authentifizierung.
- Transaktionen.

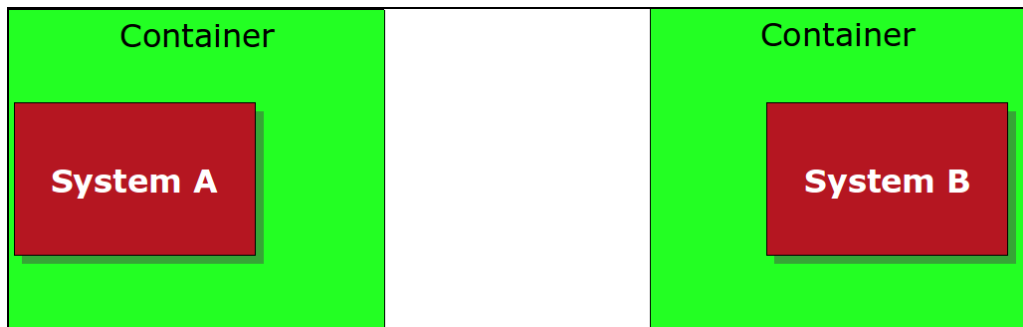
QoS erfordert die Einführung eines Containers, der die notwendigen Qualitäten realisiert.

- Applikationsserver
- Spring dm

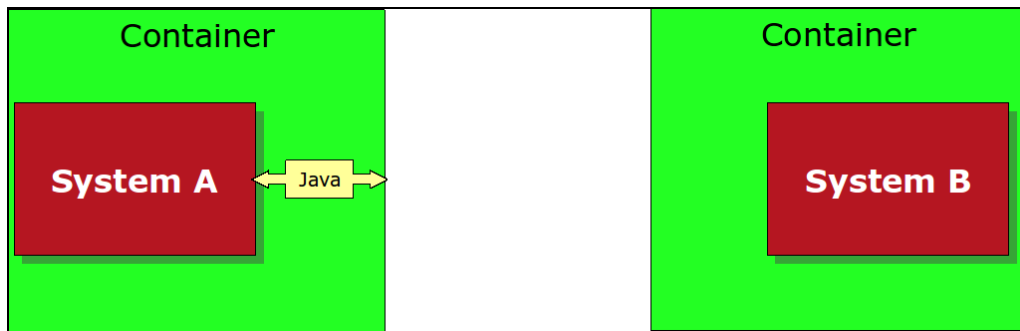
Ausgangspunkt sind auch hier zwei Systeme, die Anwendungslogik enthalten:



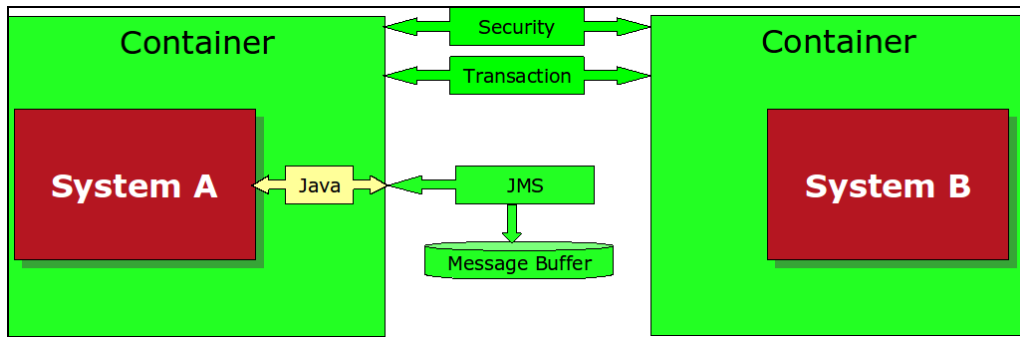
Die Systeme werden jeweils in einen Container installiert:



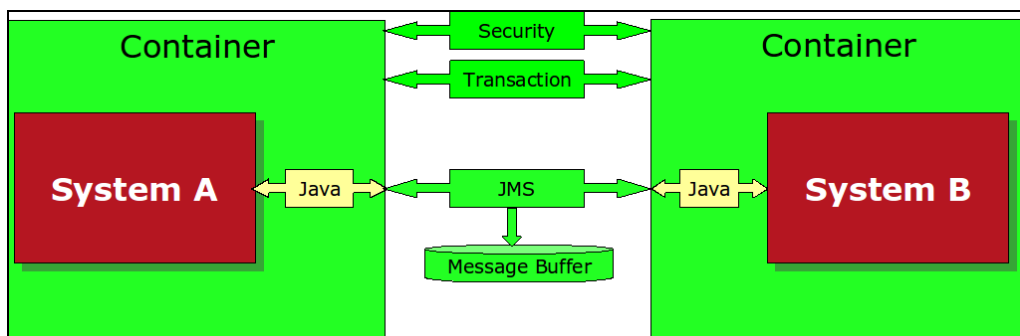
Der Container fängt sämtliche Aufrufe ab:



Die Container kommunizieren miteinander und benutzen dabei zusätzliche Dienste:



Der Ziel-Container delegiert die Aufrufe anschließend an das Zielsystem B weiter:



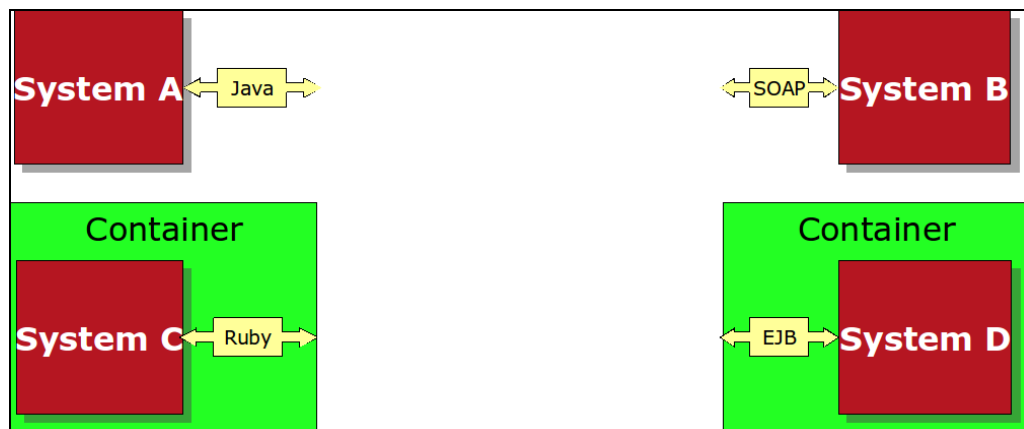
### 2.2.5 Entkopplung

Die direkte Kommunikation zwischen den beteiligten Systemen wird aufgegeben. Die Aufrufe werden stattdessen von einem Bus orchestriert.

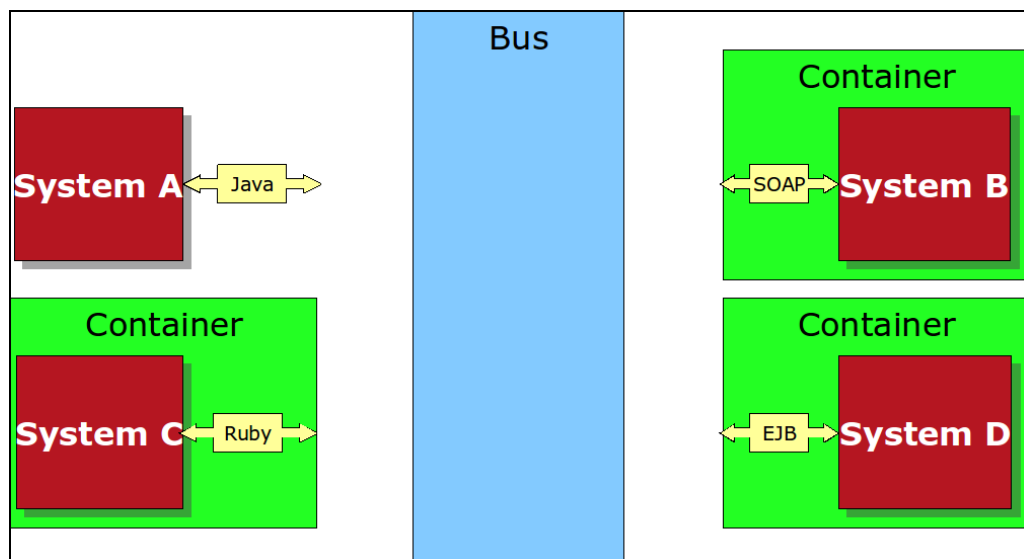
Funktionen der Bus-Implementierung sind:

- Routing/Filtering
- Datentransformation
- Fehlerbehandlung

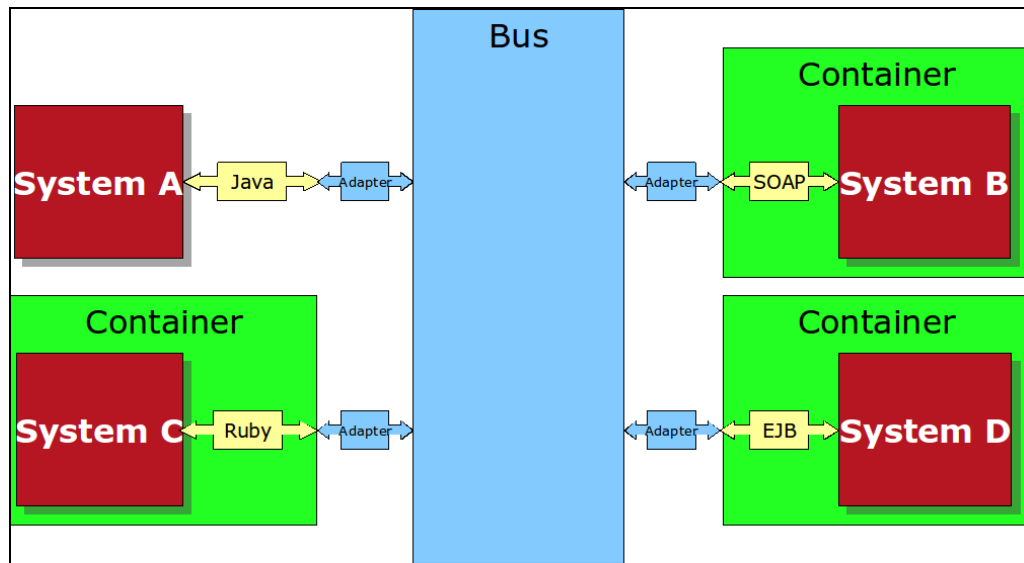
Ausgangspunkt hier sind verschiedene Systeme, die in unterschiedlichen Umgebungen laufen:



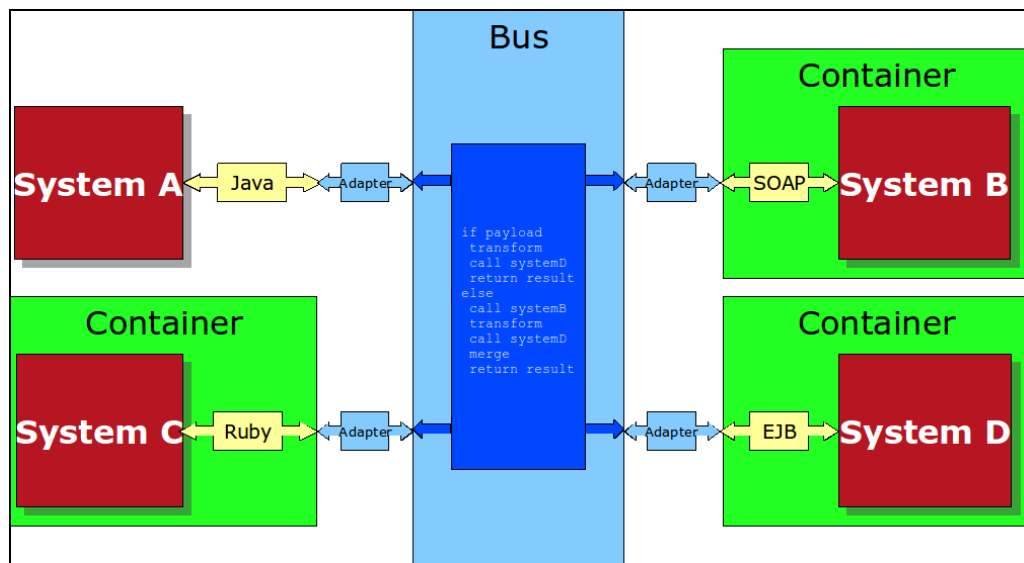
Ein weiteres System, der zentrale Bus, wird eingeführt:



Die Systeme werden über Adapter angekoppelt:



Innerhalb des Busses werden die Systeme nun verdrahtet:



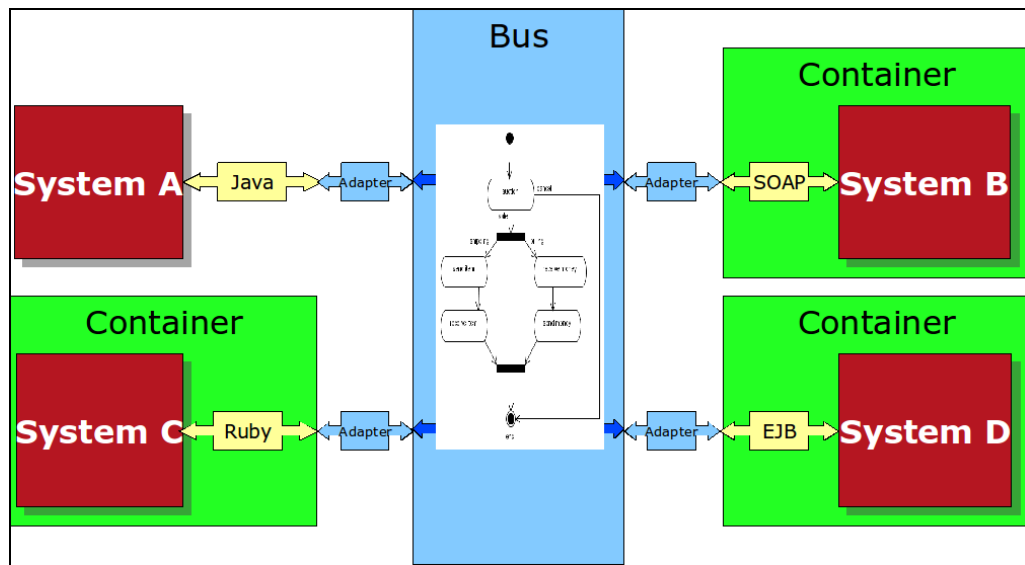
### 2.2.6 Workflows

Einführung von Prozess-Variablen. Diese werden automatisch zwischen den einzelnen Prozessschritten persistiert.

Workflows werden durch einen Graphen repräsentiert. Damit ist Java als Programmiersprache wenig geeignet.

Besser sind XML-basierte Skript-Sprachen:

- BPEL
- jBPM

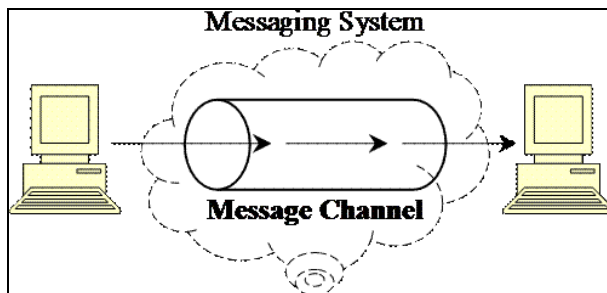


## 2.3 Messaging Pattern

Die Messaging Technologie und die darauf beruhende Message Oriented Middleware (MOM) ist unabhängig vom Java Messaging Service etabliert. Es sind eine ganze Reihe von EAI Patterns definiert worden, die auf diese Technologie zugeschnitten sind (<http://www.eaipatterns.com>). Einige dieser Patterns sind bereits in die JMS Spezifikation mit aufgenommen worden. Die folgende Übersicht präsentiert eine Reihe dieser EAI Pattern und ihre Einordnung im JMS Umfeld. Nachdem diese Pattern auch beim Web Services Messaging eingesetzt werden können, erfolgt hier eine ausführliche Aufzählung.

### 2.3.1 Message Channel

Ein Message Channel definiert eine Zieladresse einer Nachricht:



Im Java Messaging Service übernehmen die Destination-Typen Queue und Topic die Rolle von Message Channels.

### 2.3.2 Message

Eine Message dient zum Austausch von Informationen. Im Java-Umfeld sind dies natürlich wiederum die Message-Schnittstellen aus `javax.jms`. Zusätzlich werden aber bei den Web Services noch die XML-basierten SOAP-Messages eingeführt.

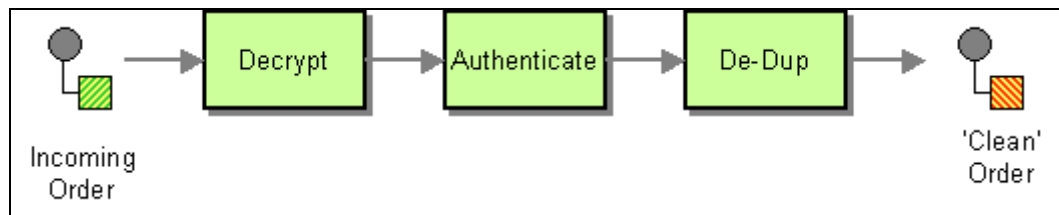
### 2.3.3 Pipes

Erfordert die Verarbeitung einer Nachricht mehrere logisch voneinander unabhängige Verarbeitungsschritte, so werden diese Schritte sicherlich in mehreren Komponenten abgearbeitet.

Beispiel:

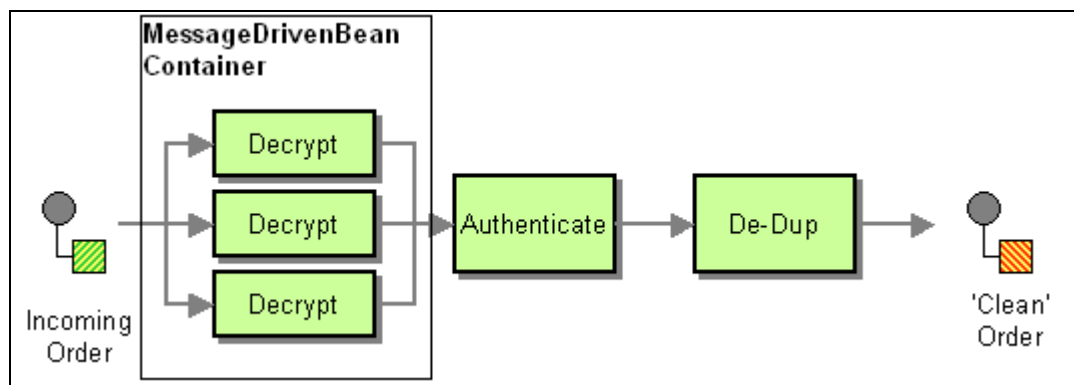
- Entschlüsselung
- Authentifizierung
- Erkennen und Aussortieren doppelt gesendeter Nachrichten

Soll die Verarbeitung innerhalb der Message Oriented Middleware erfolgen, muss die Message eine Pipe durchlaufen:



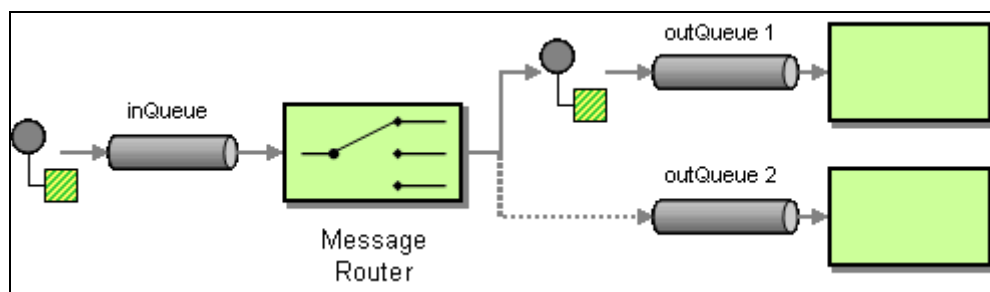
Die Kopplung der einzelnen Message Listener erfolgt über eigens eingerichtete Queues. Für jeden zusätzlichen Schritt wird eine eigene Queue benötigt.

Sind die einzelnen Verarbeitungsschritte zustandslos, können hierfür zur Lastverteilung auch sehr schön MessageDrivenBeans eingesetzt werden:



### 2.3.4 Message Router

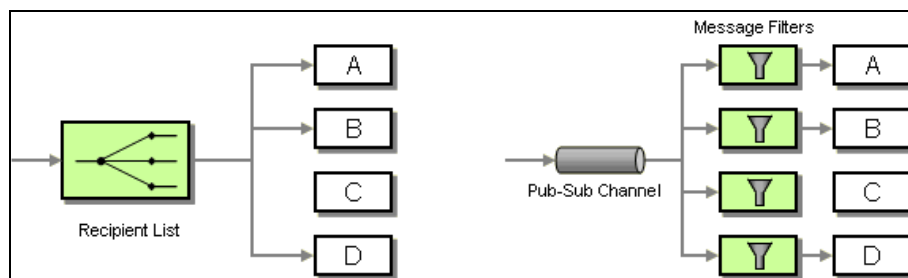
Die eben eingeführte Message Pipe ist in dieser Form nur für einfache Sequenzen der Verarbeitung möglich. Muss der nächste Schritt auf Grund von komplexen Entscheidungen getroffen werden, sollte diese Anwendungslogik in eine spezielle Router-Klasse ausgelagert werden:



**Hinweis:** Diese Argumentation führt bei den Request-Response orientierten Web Anwendungen zur Einführung des J2EE Patterns Service to Worker.

Beispiele für den Einsatz eines Message Router sind:

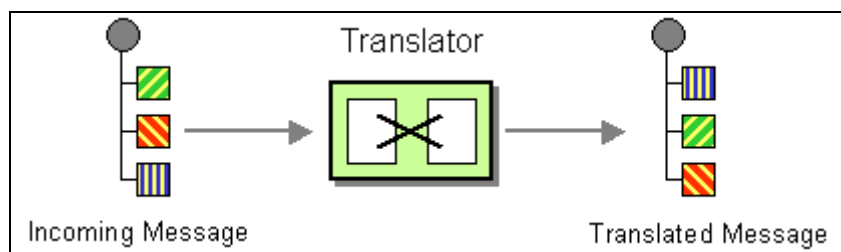
- „Content Based Router“: Der Router liest den Inhalt oder die JMS Metadaten und bestimmt daraus den nächsten Empfänger.
- „Message Filter“: Hier tritt der JMS Provider als Router auf, der die Nachrichten nur an die Empfänger übermittelt, für die der Message Selector passt.
- „Recipient List“: Hier übernimmt ein entweder ein Message Listener die Registrierung und Weiterleitung der Nachrichten, oder aber ein Topic wird in Kombination mit Message Filtern eingesetzt:



- „Routing Slip“: Hier kann der Router aktiv den nächsten Empfänger bestimmen. Zu diesem Zweck kann jede Message eine Liste der Empfänger in der Reihenfolge ihrer Abarbeitung enthalten.
- „Process Manager“: Die Liste kann bei Bedarf modifiziert werden. Der Empfänger kann diese Aufgabe an eine Hilfsklasse, den Process Manager delegieren.

### 2.3.5 Message Translator

Produzent und Konsument sind innerhalb einer MOM insbesondere beim Fokus auf Enterprise Application Integration vollkommen voneinander entkoppelt. Damit ist auch nicht in allen Fällen garantiert, dass die beiden Systeme die gleichen Message Formate verstehen. Benötigt beispielsweise ein Service Activator ein Value Object, kann dies eine C-Anwendung nicht liefern. In solchen Fällen hilft der Message Translator:



Die Implementierungen für den Übersetzer fallen unterschiedlich komplex aus. Unterstützen Produzent und Konsument XML, so wird der Übersetzer mit einer einfachen XSL-Transformation auskommen.



### 2.3.6 Message Endpoint

Ein Message Endpoint kapselt den Client vor dem angesprochenen Messaging System. Der Grad der notwendigen Entkopplung entscheidet über die Strategie der Realisierung:

- Für einen einfachen Alias-Mechanismus zwischen dem vom Client verwendeten und dem administrativ im Provider eingerichteten Namen einer Destination genügt eine konfigurierbare Variante des Service Locators. Hier kennt der Client aber noch die verwendete Technologie, beispielsweise JMS.
- Soll die verwendete Message-Technologie für den Client transparent getauscht werden können, muss eine eigene Zwischenschicht eingeführt werden. Dies kann beispielsweise der Business Delegate übernehmen.
- Übernimmt ein ganzes Framework für Applikationen die Übermittlung der Nachrichten, so spricht man von einem „Message Gateway“.

### 2.3.7 Message Channels Typen

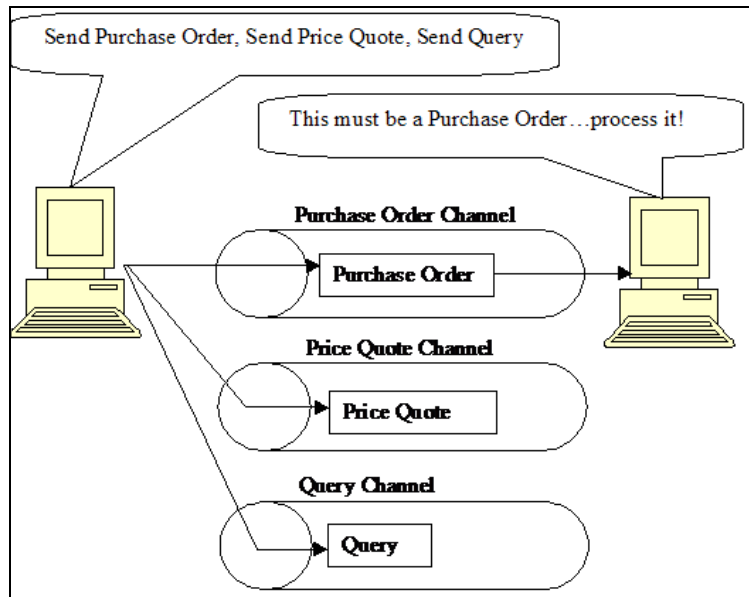
Die Typen

- Point-to-Point
- Publish-Sucscribe
- Durable Subscriber

werden bereits vom JMS-Provider umgesetzt.

### 2.3.8 Datatype Channel

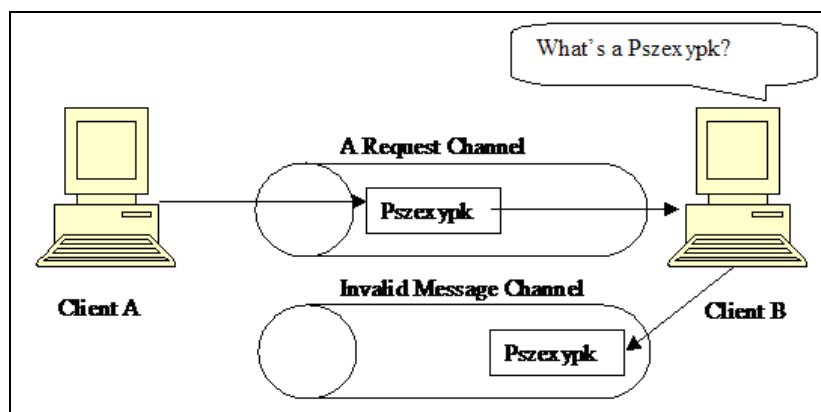
Der Empfänger einer Message bekommt häufig nur Daten gesendet. Was mit diesen Daten ausgeführt werden soll, wird häufig dadurch definiert, dass die Message an eine speziell dafür eingerichtete Destination gesendet wird.



Hier wird der Name der Destination als Aktionskommando verwendet.

### 2.3.9 Invalid Message Channel und Dead Letter Channel

Dieses Pattern beschreibt einen Mechanismus zum Exception Handling: Wie soll ein Empfänger signalisieren, dass er eine Nachricht nicht auswerten kann? Natürlich braucht der Empfänger die Nachricht nicht bestätigen oder kann eine Transaktion zurücksetzen. Dann muss der Provider die Nachricht zurücknehmen und versucht sie, einem anderen Konsumenten zu übermitteln. Ist dies aber auf Grund der Anwendungslogik nicht sinnvoll, wird die Nachricht an eine bestimmte Adresse für nicht zu verarbeitende Nachrichten weitergeleitet:



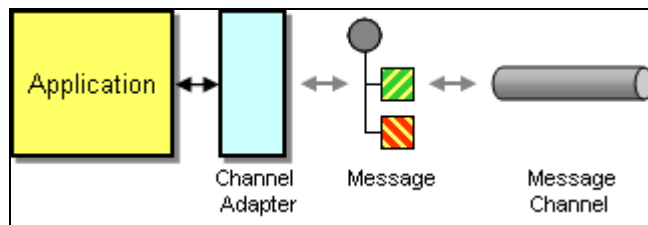
**Hinweis:** Messaging Provider benutzen auch intern solche speziellen Destinations für nicht-zustellbare Nachrichten. Dies ist dann der Dead Letter Channel.

### 2.3.10 Guaranteed Delivery

Fällt der JMS-Provider aus, so sind die Nachrichten, die sich nur im Hauptspeicher der Anwendung befunden hatten, verloren. Soll dies verhindert werden, müssen die Nachrichten persistent abgelegt werden. Dies ist bei JMS durch den `javax.jms.DeliveryMode.PERSISTENT` möglich, der beim Senden einer Nachricht als Parameter gesetzt werden kann.

### 2.3.11 Channel Adapter

Ein Channel Adapter übernimmt die Rolle des klassischen Adapters und passt die Signaturen von Anwendung und Message Provider an. Dazu kann der Channel Adapter die Anwendungsdaten Auslesen und zum Versenden von Nachrichten benutzen. Empfängt der Adapter eine Nachricht, kann eine Callback-Methode der Anwendung aufgerufen werden.



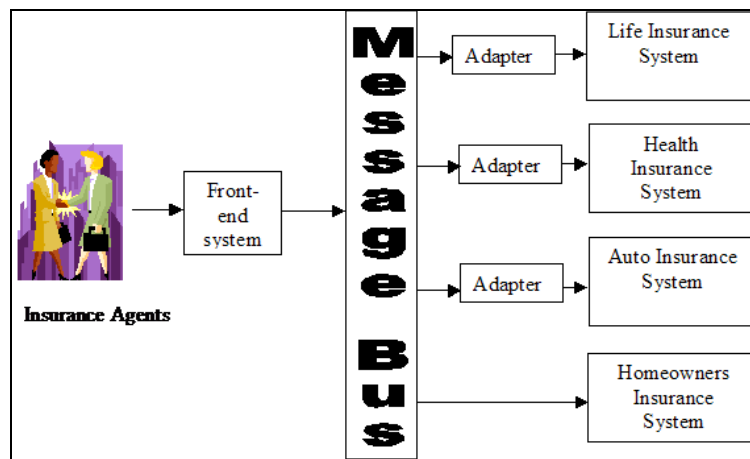
### 2.3.12 Message Bridge

Die Message Bridge ist ein sehr einfaches Pattern: Ein Message Provider muss in der Lage sein, Nachrichten an einen anderen Provider zu senden. Ist dies gewährleistet, fungiert dieser Provider als Message Bridge.

### 2.3.13 Message Bus

Diese Pattern beschreibt, wie verschiedene Anwendungen unabhängig voneinander dynamisch zu einem funktionierenden Gesamtsystem verknüpft werden können. Die Implementierung dieses Patterns erfordert:

- Einen Messaging Provider
- Eventuell Pipes für die einzelnen Backend-Systeme. Die Pipe enthält beispielsweise einen Translator und einen Adapter.



### 2.3.14 Document Message

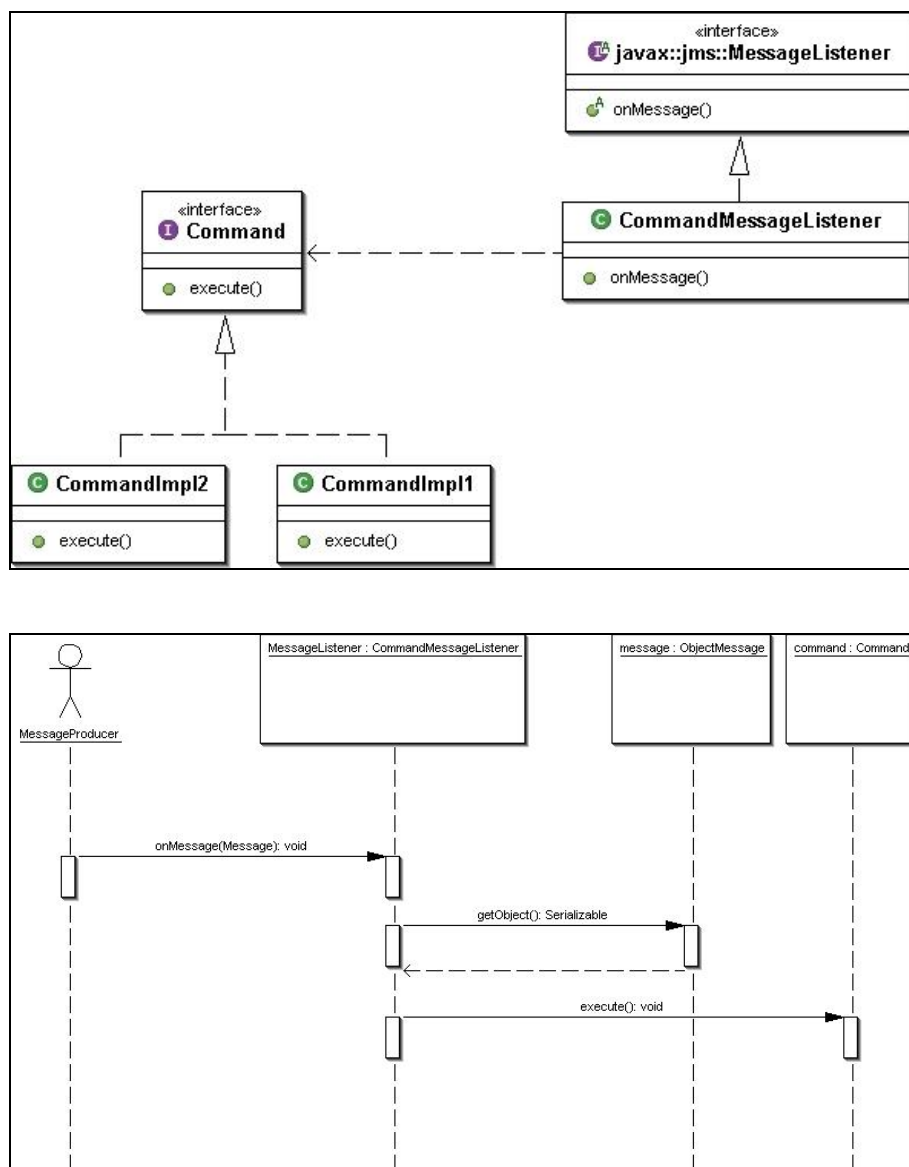
Eine Document Message transportiert ausschließlich Daten. Im JMS-API werden dafür die Subklassen der `Message`-Schnittstelle verwendet.

Die vom Empfänger auszuführende Aktion wird dann durch einen Data-type Channel definiert.

### 2.3.15 Command Message

Eine Command Message enthält neben den reinen Informationen auch die auszuführende Aktion. Dies kann in Java auf zwei Arten realisiert werden:

- Die Message transportiert ein Aktionskommando, das der Empfänger interpretieren kann.
- Die Message enthält eine Instanz einer funktionellen Java-Klasse, die gemäß des Command-Patterns erstellt wurde. Dies ist bei JMS durch die Übertragung einer `javax.jms.ObjectMessage` möglich. Der Empfänger liest das übertragene Objekt und ruft die zugehörige Aktionsmethode auf:



### 2.3.16 Event Message

In Java ist eine Event Message eine beliebige Message. Alleine der Kontext des Versendens der Nachricht legt fest, dass es sich hierbei um das Signalisieren eines Ereignisses handeln wird.

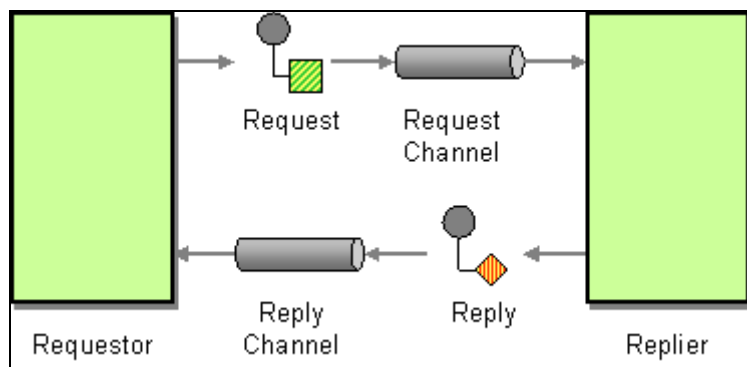
### 2.3.17 Return Address

Der Empfänger einer Nachricht muss eine Nachricht an den Sender zurücksenden können. Dafür muss er dessen Adresse kennen. Dies ist bei einem JMS System sehr einfach: Jede Message kann eine Destination als Reply-To-Eigenschaft setzen:

```
message.setJMSReplyTo(replyDestination);
```

### 2.3.18 Request-Reply

Ein Nachrichtensender erwartet von seinem Empfänger eine Antwort. Dazu müssen die Rollen vertauscht werden, d.h.: der Sender (Requestor) tritt als Empfänger auf, der Empfänger (Replier) als Sender:



Die Realisierung eines Request-Replies kann innerhalb eines JMS-Systems wiederum auf mehrere Arten erfolgen:

- Setzen einer eindeutigen Client ID beim Erzeugen der Nachricht. Der Replier setzt diese ID als Property seiner Antwort-Message. Der Requestor kann beispielsweise einen Message Selector verwenden, um nur Nachrichten mit seiner ID zu empfangen.
- Verwenden der temporären Destinations. Diese definieren ja gerade private Kommunikationskanäle und können einfach als Message Property „JMSReplyTo“ gesetzt werden.

### 2.3.19 Correlation Identifier

Durch die Asynchronizität der MOM kann ein Requestor nicht mehr auf Grund der Reihenfolge des Eintreffens von Antworten erkennen, auf welche seiner Nachrichten sich die Antwort bezieht. Um dies zu erkennen, wird die JMS Message-Property „CorrelationID“ verwendet.

### 2.3.20 Message Sequence

Große Mengen an Informationen können häufig auf Grund der Beschränkung der Größe nicht in einer einzigen Message übertragen werden. Aus diesem Grund muss die Nachricht in mehrere Einzelpakete übertragen werden. Der Empfänger muss dann den vollständigen Datenbestand Portionsweise rekonstruieren können.

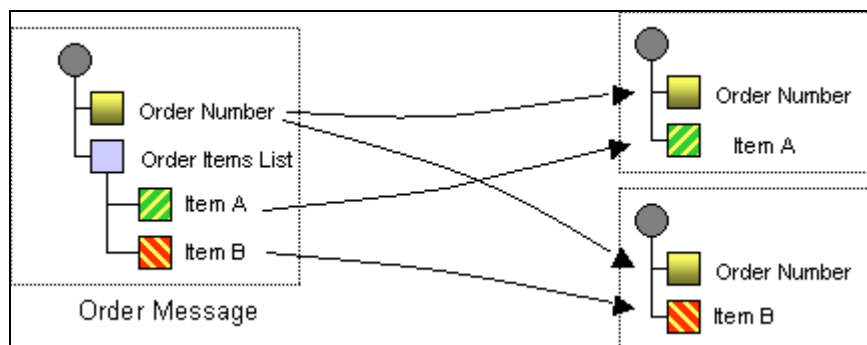
Zur Realisierung können natürlich Properties der Message verwendet werden. Dies muss aber die Anwendungslogik selbst übernehmen: JMS bietet keine Möglichkeit, Message Sequenzen automatisch erzeugen zu lassen.

### 2.3.21 Message Expiration

Die Umsetzung erfolgt einfach durch Setzen der JMS Property „JMSExpiration“.

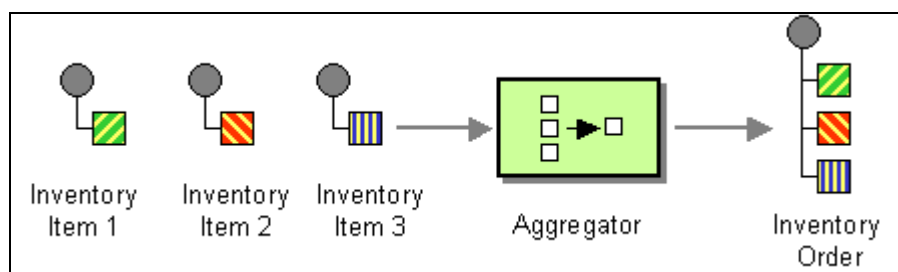
### 2.3.22 Splitter

Ein Splitter liest eine Nachricht und zerlegt sie in Einzelnachrichten, die an unabhängige Subsysteme gesendet werden können:



### 2.3.23 Aggregator

Die Umkehrung des Splitters:

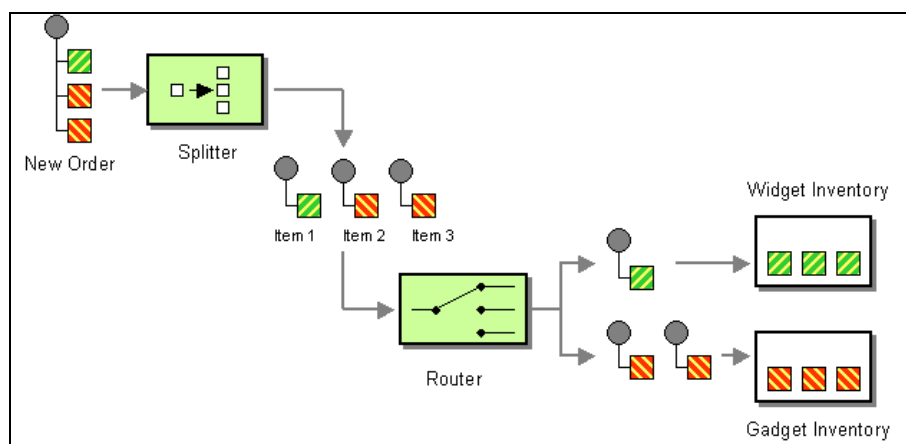


### 2.3.24 Resequencer

Der Resequencer muss Nachrichten, die in nicht-korrekt Reihenfolge übertragen wurden, für die Verarbeitung in die korrekte Reihenfolge bringen. Der Resequencer ist eine spezielle Abart des Aggregators: Aggregiert werden die Nachrichten einer Message Sequence.

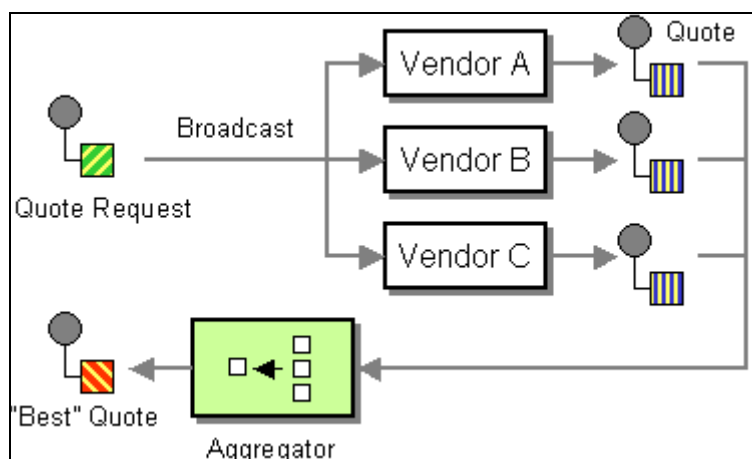
### 2.3.25 Composed Message Processor

Dieses Pattern kann durch eine Kombination bereits bekannter Muster erzeugt werden. Die Aufgabe des Composed Message Processors ist es, eine Nachricht zu zerlegen und an getrennte Subsysteme zu versenden. Dies ist ein Splitter in Kombination mit einem Router:



### 2.3.26 Splitter-Gather

Eine Nachricht wird von mehreren Subsystemen behandelt, der Sender erwartet eine Bestätigung:

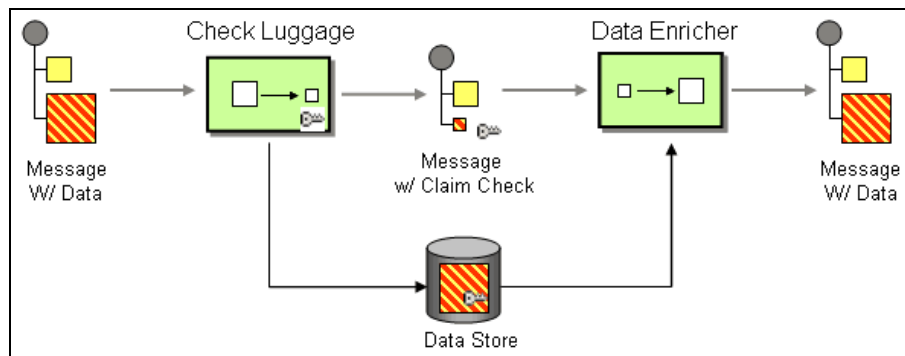




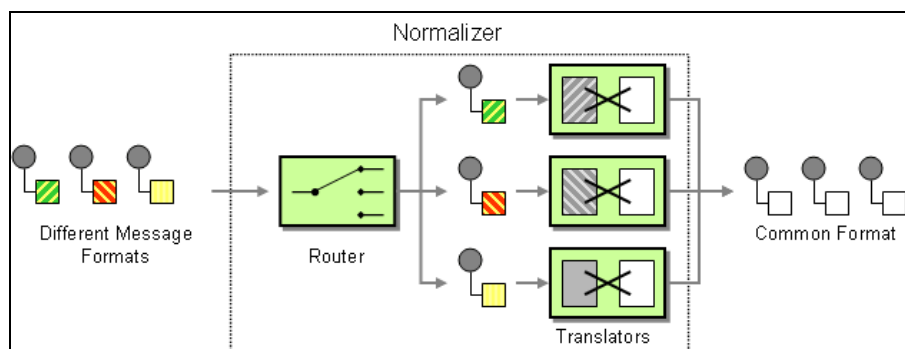
### 2.3.27 Message Transformation

Zur Transformation des Inhalts von Messages wurde bereits der Translator vorgestellt. Andere Typen von Transformationen sind:

- „Content Enricher“: Der Empfänger fügt der Nachricht weitere Inhalte hinzu.
- „Content Filter“: Dieser Filter aus einer Nachricht mit einer großen Datenmenge nur die interessierenden Teile heraus und sendet diese weiter.
- „Claim Check“: Dieses Pattern wird eingesetzt, um die Menge der zu übertragenden Informationen zu minimieren. Übertragen wird nicht mehr der eigentliche Inhalt, sondern beispielsweise ein Verweis auf einen Eintrag in einer Datenbank:



- „Normalizer“: Der Normalizer empfängt Nachrichten in verschiedenen Formaten und konvertiert diese in ein Nachrichtenformat, das die empfangende Anwendung versteht.



- „Canonical Data Model“: In obiger Abbildung muss der Normalizer zwei unterschiedliche Formate kennen und folglich bei Änderungen sowohl auf der sendenden als auch auf der empfangenden Seite geändert werden. Um dies zu vermeiden, kann ein gemeinsames, applikationsunabhängiges Datenmodell eingeführt werden.

### 2.3.28 Messaging Endpoints

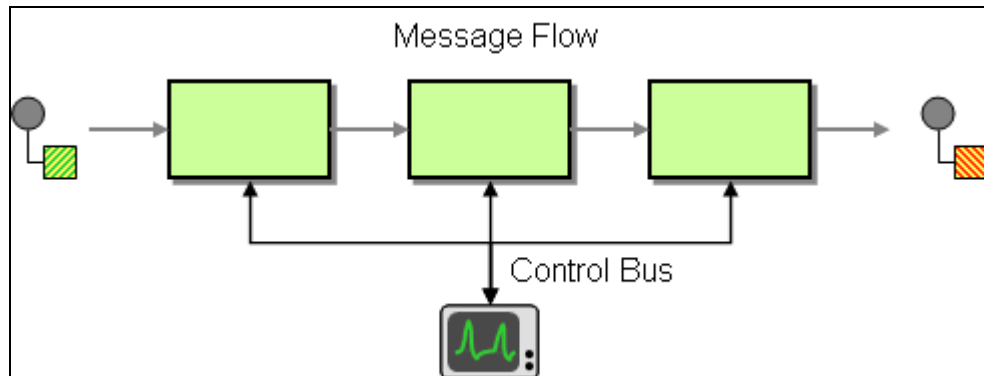
Das Verhalten von Message Empfängern lässt sich wiederum an Hand einiger Pattern gruppieren:

- „Polling Consumer“: Der Empfänger wartet auf das Eintreffen einer Nachricht. Dies ist nichts anderes als das blockierende `receive()` des Empfängers.
- „Event-Driven-Consumer“: Der `javax.jms.MessageListener`.
- „Competing Consumers“: Hier sollen mehrere Messages von einem Empfänger quasi gleichzeitig ausgeführt werden. Hier bieten sich zur Realisierung die `MessageDrivenBeans` der J2EE an.
- „Message Dispatcher“, um eine Nachricht von verschiedenen Prozessen gleichzeitig ausführen lassen zu können. Auch hier bieten sich zur Umsetzung `Enterprise JavaBeans` an.
- „Transactional Client“: Bei JMS ist eine Session in der Lage, das Empfangen und Senden von Nachrichten innerhalb einer Transaktion zu gruppieren.
- „Message Adapter“: Geschäftsprozesse sollen synchron und asynchron aufgerufen werden können. Dieses Pattern entspricht im J2EE-Umfeld einer Kombination einer Session Fassade für synchrone Zugriffe und einer Message Fassade für asynchrone.
- „Selective Consumer“: Nur bestimmte Nachrichten sollen den Empfänger erreichen. Bei JMS übernehmen die `Message Selectors` diese Aufgabe.
- „Idempotent Receiver“: Falls das JMS System bei nicht-quittierten Nachrichten diese nochmals an den Empfänger verschickt, muss dieser damit umgehen können. In einem JMS System kann der Acknowledge-Mode auf „DUPS\_OK“ gesetzt sein. In diesem Falle sollte der Empfänger „idempotent“ sein, d.h.: Das Ergebnis des Methodenaufrufs ist unabhängig von der Anzahl der Aufrufe.

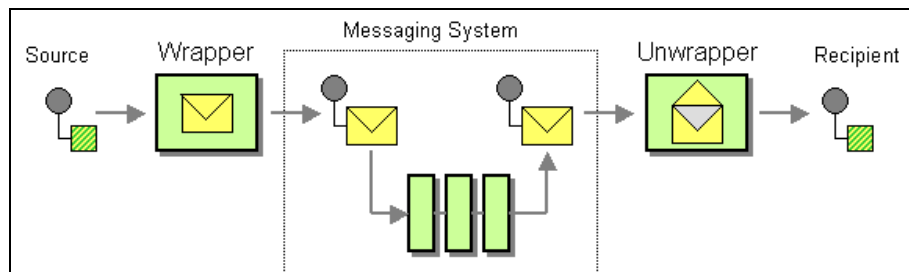
### 2.3.29 Patterns für das System Management

Die oben vorgestellten Messaging Patterns sind aus den Erfordernissen der Anwendungsentwickler heraus entstanden. Die folgende Liste enthält noch einige Patterns für das Management des Messaging Systems.

- „Control Bus“: Dieser empfängt spezielle Kontroll-Nachrichten (Logging, Auditing...) auf einer eigenen Destination.

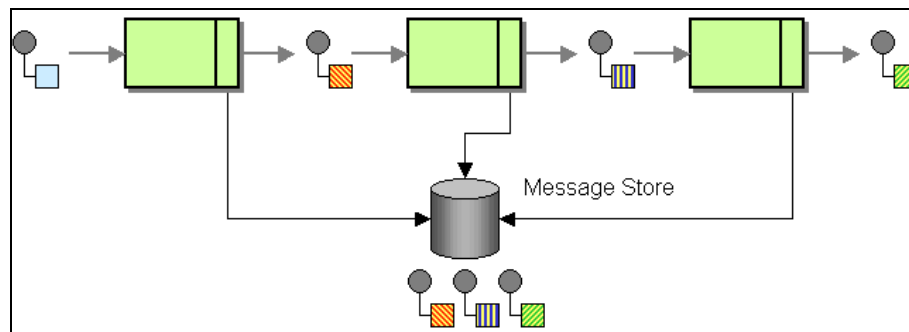


- „Envelope“: Ein spezielles Messaging System kann bestimmte Metadaten erwarten. Um diese zur Verfügung zu stellen, umhüllt ein Envelope die originale Nachricht.

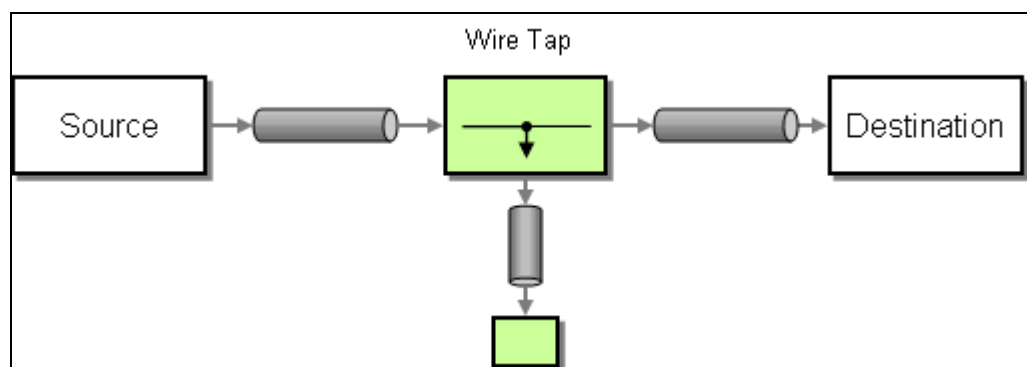


- „Message History“: Für Debugging-Zwecke ist es sehr hilfreich, wenn jeder an der Verarbeitung der Nachricht beteiligte Empfänger nachvollzogen werden kann. Dazu definiert die Nachricht eine Liste als Message History.

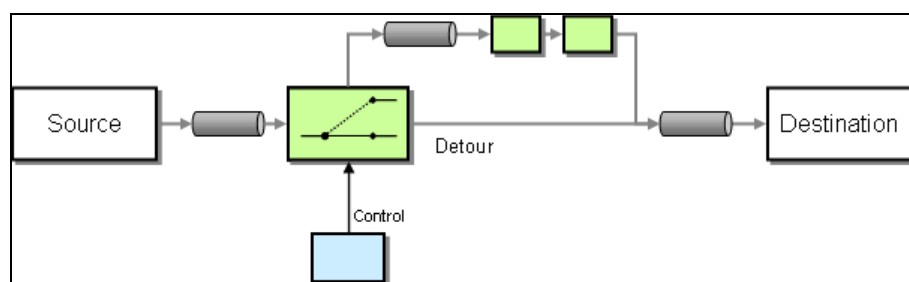
- „Message Store“: Kopien von Nachrichten (oder zumindest die relevanten Teile) werden in einem persistenten Store abgelegt:



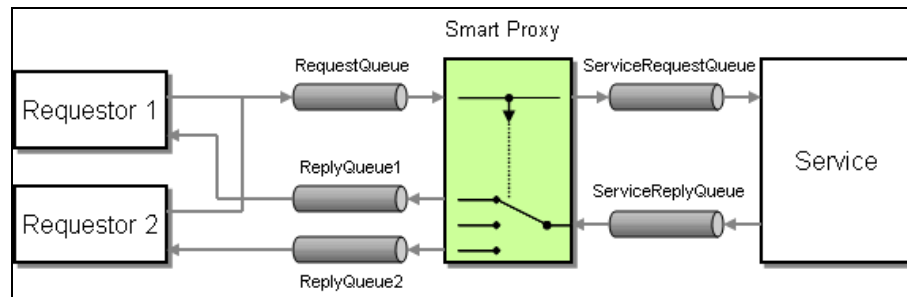
- „Channel Purger“: Nicht ausgelieferte Nachrichten eines persistenten Messaging Systems können beim Neustart eines Empfängers zu ungewollten Reaktionen führen. Deshalb sollten vorher die Destinations geleert werden. Dies übernimmt ein einfacher Empfänger, der vorab an der Destination registriert wird und alle darin vorhandenen Meldungen entfernt.
- „Wire Tap“: Wie können nachrichten an eine Queue debugged werden? Dazu dient eine kleine zwischengeschaltete Recipient List mit dem originalen Empfänger und einem debuggenden:



- „Detour“: Es sollen dynamisch weitere Verarbeitungsschritte (Validierung, Authentifizierung etc.) ausgeführt werden können. Dies übernimmt der Detour, der beispielsweise aus einer Konfigurationsdatei die notwendigen Informationen zum Aufbau der Liste entnimmt:



- „Smart Proxy“: Hier besteht die Problemstellung darin, dass in einem Request-Response-System wiederum zusätzliche Funktionen eingeführt werden müssen. Dazu muss nun aber die Antwortadresse ebenfalls berücksichtigt werden, eine Aufgabe für einen Proxy:



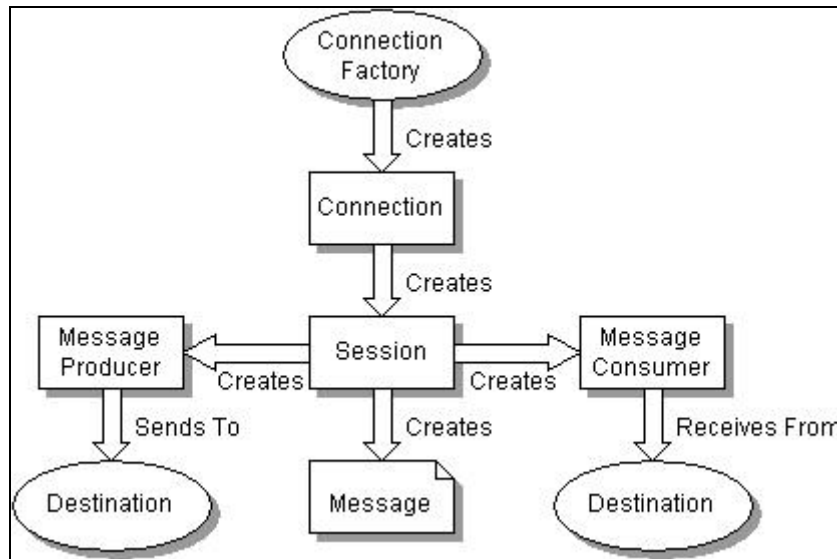
### 3 Java Messaging Service

Wie schon üblich spezifiziert Java ein plattformunabhängiges Messaging-System. Das Paket `javax.jms` besteht folglich aus einer Menge von Interfaces, die Implementierung der Schnittstellen bleibt die Aufgabe des Message-Providers.

#### 3.1 Aufbau der Verbindung

Weiterhin stellt der Message-Provider Implementierungen des `javax.jms`-APIs zur Verfügung.

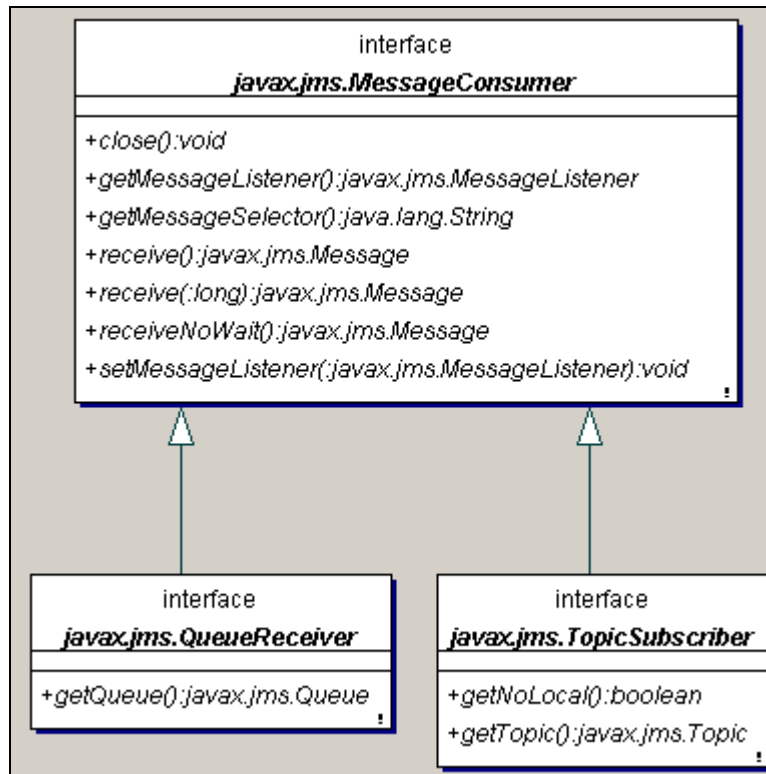
Das Zusammenspiel der Komponenten ist in der folgenden Grafik dargestellt:



- Zentraler Einstiegspunkt ist die Connection Factory. Diese wird in der Regel durch einen JNDI-Lookup erhalten.
- Mit der Connection Factory wird eine Connection zum JMS Provider aufgebaut.
- Die Connection erzeugt für eine beliebige Menge unterschiedlicher Sessions.
- Die Session wieder dient als Factory für
  - Messages. Der Produzent erzeugt als im strengen Sinne gar keine Nachrichten, sondern holt aus sich der Session leere und füllt diese aus.
  - Produzenten und Konsumenten.
- Diese registrieren sich anschließend bei einer Destination. Destinations werden auch per JNDI-Lookup in den Namensraum des JMS-Providers erhalten.

### 3.1.1 Message Consumer

Wie in den obigen Bildern dargestellt, werden Messages von einer Quelle an den Message Provider übermittelt. Dieser wird dann die Nachricht an einen (Kopplung über Queue) oder mehrere (Kopplung über Topic) Empfänger oder „Konsumenten“ übermitteln. Dies sind die `javax.jms.MessageConsumer`:



Ein Konsument kann entweder auf das Eintreffen einer Nachricht warten (Methode `receive()`), oder ein Objekt vom Typ `javax.jms.MessageListener` als Listener anmelden. Im ersten Fall erfolgt die Übermittlung synchron, d.h. die `receive` blockiert den ausführenden Thread. Das setzen eines Listeners erfolgt asynchron.

### 3.1.2 Message Listener

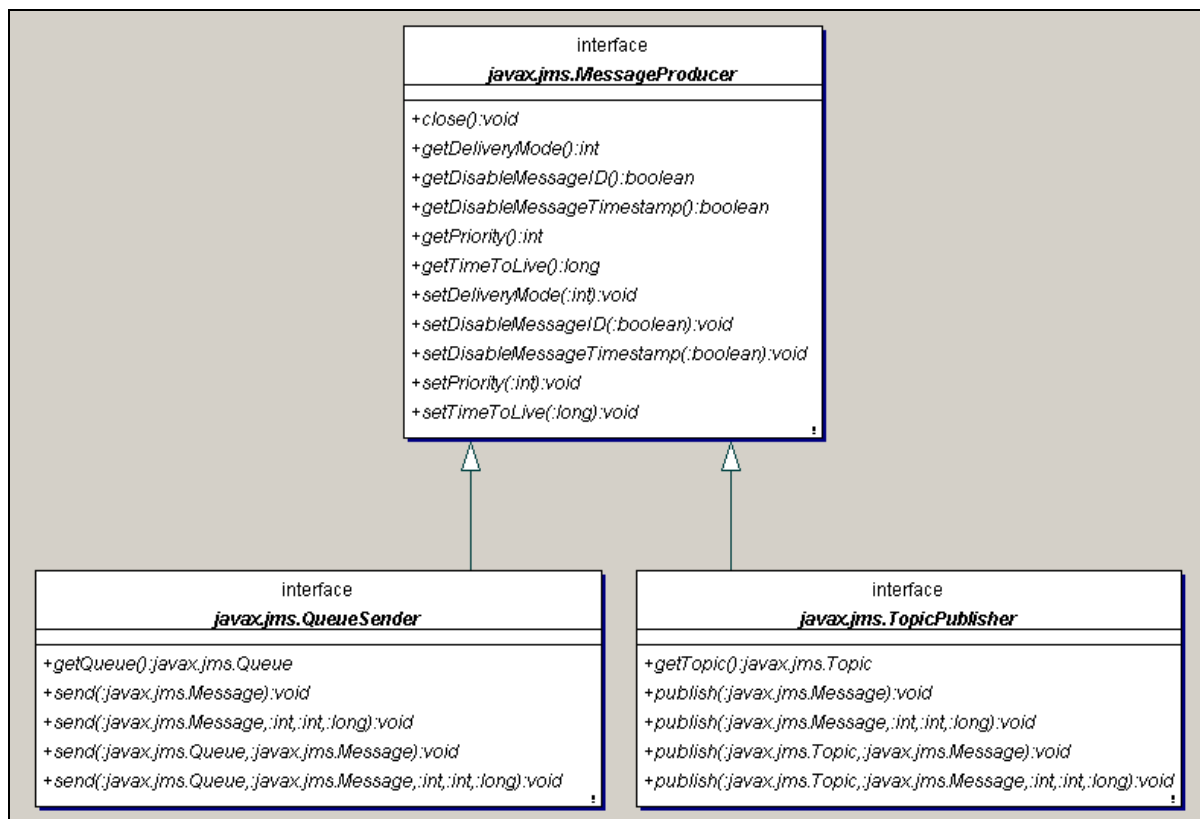
Der eben erwähnte Message Listener ist klarerweise ein Interface:



Die Methode `onMessage` erhält als Parameter die übermittelte `javax.jms.Message`, die dann weiter verarbeitet werden kann.

### 3.1.3 Message Producer

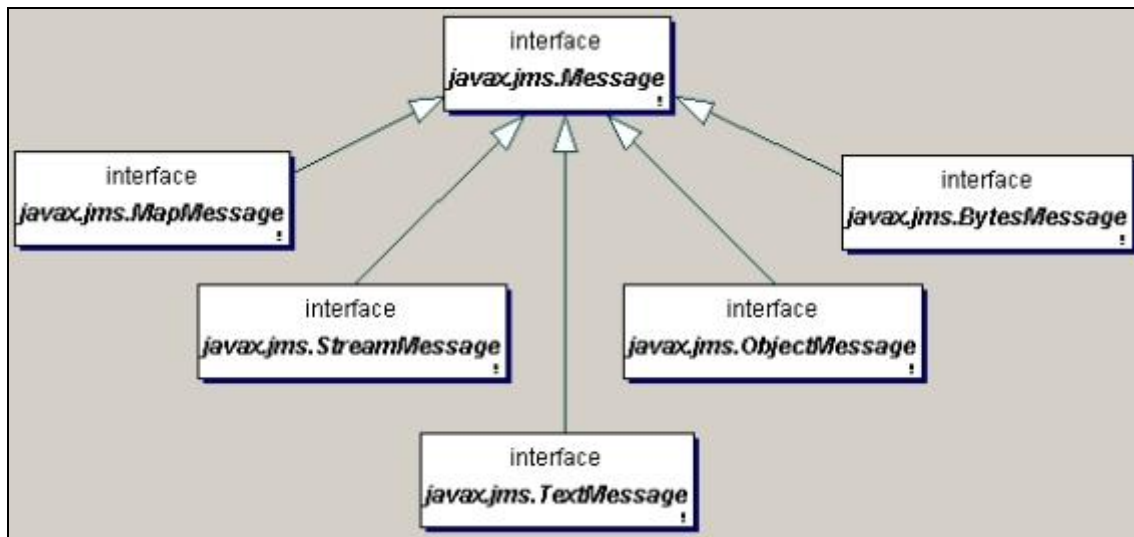
Die Produzenten von Messages sind die `javax.jms.MessageProducer`





### 3.1.4 Messages

Innerhalb des Java Messaging Systems sind insgesamt 6 Typen von Message-Interfaces definiert:



#### 3.1.4.1 Die javax.jms.Message

Eine `javax.jms.Message` besteht aus einem Header, einem Satz von Properties und einem Body. Der Message-Header enthält eine Reihe von Attributen:

Header Field	Beschreibung	Gesetzt von
JMSDestination	Ziel der Nachricht	send oder publish
JMSDeliveryMode	Modus der Nachrichten-Übermittlung, , persistent' oder , non-persistent'	send oder publish
JMSExpiration	Bis zu diesem Zeitpunkt darf die Nachricht übermittelt werden	send oder publish
JMSPriority	Priorität	send oder publish
JMSMessageID	Eindeutige ID, die bei der Übermittlung der Message vom Provider gesetzt wird	send oder publish
JMSTimestamp	Zeitmarke, zu der die Message an den JMS Provider übermittelt wurde	send oder publish
JMSCorrelationID	Einige Provider unterstützen die native Correlation ID, um Nachrichten zu verknüpfen. Das Byte-Array muss mit ,ID:'	Client

Header Field	Beschreibung	Gesetzt von
	starten	
JMSReplyTo	Name einer Queue oder eines Topic, die als Antwortadresse verwendete werden soll	Client
JMSType	Name einer Message-Definition, die von einem JMS-Provider definiert wird	Client
JMSRedelivered	Zeitmarke, zu dem eine Nachricht erneut ausgeliefert wurde	JMS Provider

Properties können durch einen Satz von Methoden komfortabel gesetzt werden:

- BooleanProperty
- ByteProperty
- DoubleProperty
- FloatProperty
- IntProperty
- LongProperty
- ObjectProperty
- ShortProperty
- StringProperty

### 3.1.4.2 Sub-Interfaces von javax.jms.Message

Die einfache javax.jms.Message und die darin gesetzten Eigenschaften sind als Metadaten für die eigentliche zu transportierende Information aufzufassen. Die weiteren Sub-Schnittstellen dienen als Container für jeweils einen bestimmten Typ von Informationen.

Message Typ	Message-Körper enthält
TextMessage	Ein <code>java.lang.String</code> beliebiger Länge
MapMessage	Schlüssel-Werte-Paare
BytesMessage	Strom von Bytes
StreamMessage	Strom einfacher Java-Datentypen
ObjectMessage	Serialisiertes Objekt

### 3.2 Beispiele

Die folgenden Beispiele zeigen exemplarisch die verschiedenen Varianten zum Senden und Empfangen von Messages. Selbstverständlich können im realen Projekt bestimmte Aufgaben auch in Hilfsklassen ausgelagert werden. So bietet es sich natürlich an, den Service Locator zum Suchen der Factories und Destinations zu erweitern:

```
public class ServiceLocator {
    private Context jndiContext;
    private ServiceLocator() {
        try{
            jndiContext = new InitialContext();
        } catch (Exception e) {
            e.printStackTrace();
            IllegalStateException ie =
                new IllegalStateException("ServiceLocator failed!");
            ie.initCause(e);
            throw ie;
        }
    }
    private static ServiceLocator instance;
    public static ServiceLocator getInstance() {
        if (instance == null) {
            instance = new ServiceLocator();
        }
        return instance;
    }
    public QueueConnectionFactory findQueueConnectionFactory(String name) {
        try {
            return (QueueConnectionFactory)
jndiContext.lookup(name);
        } catch (Exception e) {
            e.printStackTrace();
            IllegalStateException ie =
                new IllegalStateException("ServiceLocator failed!");
            ie.initCause(e);
            throw ie;
        }
    }
}
```

```
}

public TopicConnectionFactory findTopicConnectionFactory(String
name) {
    try {
        return (TopicConnectionFactory)
jndiContext.lookup(name);
    } catch (Exception e) {
        e.printStackTrace();
        IllegalStateException ie =
            new IllegalStateException("ServiceLocator failed!");
        ie.initCause(e);
        throw ie;
    }
}

public Queue findQueue(String name) {
    try {
        return (Queue) jndiContext.lookup(name);
    } catch (Exception e) {
        e.printStackTrace();
        IllegalStateException ie =
            new IllegalStateException("ServiceLocator failed!");
        ie.initCause(e);
        throw ie;
    }
}

public Topic findTopic(String name) {
    try {
        return (Topic) jndiContext.lookup(name);
    } catch (Exception e) {
        e.printStackTrace();
        IllegalStateException ie =
            new IllegalStateException("ServiceLocator failed!");
        ie.initCause(e);
        throw ie;
    }
}
}
```

In den gedruckten Beispielen ist jedoch zur Darstellung des Ablaufs auf den Einsatz des Service Locators verzichtet worden.

### 3.2.1 Senden und Empfangen durch eine Queue

#### 3.2.1.1 Empfangen der Message durch den QueueReceiver

Der `QueueReceiver` kann auf das Eintreffen einer Nachricht warten:

```
try {
    InitialContext context = new InitialContext();
    QueueConnectionFactory fac =
        (QueueConnectionFactory) context.lookup("QueueConnectionFactory");
    Queue queue = (Queue) context.lookup("jms/Queue");
    QueueConnection con = fac.createQueueConnection();
    QueueSession session = con.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    QueueReceiver receiver = session.createReceiver(queue);
    Message message = receiver.receive();
    System.out.println("Received Message");
} catch (Exception e) {
    e.printStackTrace();
}
```

#### 3.2.1.2 Empfangen der Message durch einen registrierten MessageListener

```
try {
    InitialContext context = new InitialContext();
    QueueConnectionFactory fac =
        (QueueConnectionFactory) context.lookup("QueueConnectionFactory");
    Queue queue = (Queue) context.lookup("jms/Queue");
    QueueConnection con = fac.createQueueConnection();
    QueueSession session = con.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    QueueReceiver receiver = session.createReceiver(queue);
    receiver.setMessageListener(new
        DemoMessageListener());
    System.out.println("Received Message");
    Object sync = new Object();
    synchronized(sync) {
        sync.wait();
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

### 3.2.1.3 Senden einer Message an eine Queue

```
try {
    InitialContext context = new InitialContext();
    QueueConnectionFactory fac =
    (QueueConnectionFactory) context.lookup("QueueConnectionFactory");
    Queue queue = (Queue) context.lookup("jms/Queue");
    QueueConnection con = fac.createQueueConnection();
    QueueSession session = con.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
    QueueSender sender = session.createSender(queue);
    Message sendMessage = session.createMessage();
    sendMessage.setStringProperty("key", "value");
    sender.send(sendMessage);
    con.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

## 3.2.2 Publizieren und Empfangen über ein Topic

### 3.2.2.1 Empfangen der Message durch den TopicSubscriber

```
try {
    InitialContext context = new InitialContext();
    TopicConnectionFactory fac =
    (TopicConnectionFactory) context.lookup("TopicConnectionFactory");
    Topic topic = (Topic) context.lookup("jms/Topic");
    TopicConnection con = fac.createTopicConnection();
    TopicSession session = con.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
    TopicSubscriber subscriber =
    session.createSubscriber(topic);
    con.start();
    Message receivedMessage = subscriber.receive();
    System.out.println(receivedMessage);
    System.out.println(receivedMessage.getStringProperty("key"));
    con.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

### 3.2.2.2 Publizieren einer Message

```
try {
    InitialContext context = new InitialContext();
    TopicConnectionFactory fac =
(TopicConnectionFactory) context.lookup("TopicConnectionFactory");
    Topic topic = (Topic) context.lookup("jms/Topic");
    TopicConnection con = fac.createTopicConnection();
    TopicSession session = con.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
    TopicPublisher sender = session.createPublisher(topic);
    MapMessage sendMessage = session.createMapMessage();
    sender.publish(sendMessage);
    con.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

### 3.3 Message-Selektoren

#### 3.3.1 Einsatz

Mit Message-Selektoren kann der Empfänger einer Nachricht die Botschaften, die ihm übermittelt werden, filtern. Die Abfrage-Syntax ist eine Untermenge des SQL92-Spezifikation. Als Kriterium können die Properties der Message verwendet werden.

Mit Message Selektoren kann die Menge der administrativ zu erzeugenden Destinations sowie die Netzbelastung verringert werden.

Betrachten wir das Beispiel eines News-Systems mit den drei Kategorien „Sport“, „EDV“ und „Politik“. Es könnten nun drei Topics eingerichtet werden, an die sich jeweils der interessierte Subscriber getrennt registriert. Es ist nun aber klar, dass diese Realisierung etwas künstlich wirkt und, noch wichtiger: Bei der Einführung einer neuen Nachrichten-kategorie muss eine neue Topic eingerichtet werden.

Erhält jeder Subscriber jede Nachricht und prüft dann für sich, ob er die Nachricht auswertet oder nicht, wird viel überflüssiger Netzwerkverkehr initiiert.

Werden stattdessen Message Selektoren verwendet, so genügt ein einziges Topic. Neue Kategorien können dynamisch eingeführt werden.

Es ist jedoch zu bedenken, dass die Message Selektoren auch dazu benutzt werden können, fast beliebig komplexe Kriterien vom JMS-Provider ausführen zu lassen. Das hat zwei Nebeneffekte:

- Der JMS-Provider wird durch die Ausführung der Selektionsausdrücke zusätzlich belastet.
- Innerhalb der Message Selektoren kann Geschäftslogik versteckt sein, die eventuell besser in den funktionellen Klassen der verarbeitenden Empfänger aufgehoben sein kann.

Richtig eingesetzt sind die Message Selektoren damit ein äußerst praktisches Hilfsmittel innerhalb eines Messaging Systems.



### 3.3.2 Literale und Bezeichner

Zeichen-Literale werden in einfache Hochkommas eingeschlossen:

```
'Hello'
```

Exakte Zahlen (18, 60) und Kommazahlen (18.60) werden unterstützt. Als logische Literale sind die Wörter `TRUE`, `true`, `FALSE`, `false` reserviert.

Als Bezeichner sind alle Zeichenketten beliebiger Länge gültig, die

- keinem Selector Schlüsselwort entsprechen (s.u.),
- mit einem Java-Sonderzeichen (',' oder '\$') beginnen,
- nicht mit der reservierten Zeichenkette „JMS“ beginnen.

Groß- und Kleinbuchstaben werden unterschieden, Leerzeichen, Tabulatoren und Zeilenumbrüche ignoriert.

Bezeichner entsprechen den Namen der Properties einer Message.

### 3.3.3 Ausdrücke

Ausdrücke werden in ihrer Reihenfolge wie in Java ausgewertet, Klammerung ist möglich.

#### 3.3.3.1 Logische Operatoren

`NOT`

`AND`

`OR`

### 3.3.3.2 Vergleichsoperatoren

=  
>  
>=  
<  
<=  
<>

Für Zeichenketten und logische Bezeichner sind nur die Operatoren = und <> zulässig.

### 3.3.3.3 Numerische Operationen

+  
-  
\*  
/

Weiterhin gültig ist der Bereichsoperator:

BETWEEN

### 3.3.3.4 Zeichenkettenoperationen

IN

*Ausdruck IN (Liste)*

Beispiel:

Name IN (John, Paul, George, Ringo)

LIKE

*Ausdruck LIKE Muster*

als Platzhalter sind gültig:

, \_ ' Einzelzeichen

, % ' Folge von Zeichen beliebiger Länge

### 3.3.3.5 Unbekannte Werte

Wie in SQL92 wird die NULL als unbekannter Wert verwendet. Operationen mit der NULL liefern als Ergebnis stets wiederum einen unbekannten Wert.

### 3.3.4 Beispiele

category is 'Sport'	Die Eigenschaft „category“ hat den Wert 'Sport'
level > 3	Die Eigenschaft „level“ ist größer als 3
author is not null	Die Eigenschaft „author“ ist gesetzt
category is 'Sport' and author='Oskar Klose'	Eine Sport-News stammt von Oskar Klose

## 3.4 Weitergehende Themen

### 3.4.1 Bestätigung des Empfangs

Eine `Message` kann vom Empfänger bestätigt werden. Dies erfolgt entweder automatisch oder muss vom Konsumenten der `Message` durch den Aufruf der Methode `acknowledge()` durchgeführt werden. Der Modus der Empfangsbestätigung wird beim Erzeugen der `Session` als Argument der entsprechenden `create<Queue|Topic>Session-`Methode übergeben. Definiert sind:

#### 3.4.1.1 `Session.AUTO_ACKNOWLEDGE`

Die `Session` bestätigt die `Message` automatisch dann, wenn entweder die `receive()` - oder `onMessage(...)` -Methode normal beendet wird.

#### 3.4.1.2 `Session.CLIENT_ACKNOWLEDGE`

Bestätigung durch den Empfänger. Dieser ruft die `acknowledge`-Methode auf. Alle vorher empfangenen `Messages` werden automatisch mit bestätigt.

#### 3.4.1.3 `Session.DUPS_OK_ACKNOWLEDGE`

Solange eine Bestätigung einer `Message` nicht erfolgt ist, kann der Provider ein Duplikat der `Message` nochmals senden. Dies kann dann eintreten, wenn der JMS-Provider neu gestartet werden musste. Dieser Modus vereinfacht die Arbeit der `Session` etwas, kann aber zum Eintreffen von Duplikaten führen. Dies muss bei der Programmierung des Empfängers bedacht sein.

### 3.4.2 Persistenz von Messages

Ein `Message Provider` kann Botschaften persistent halten und so bei einem potenziellen Ausfall restaurieren. Für unkritische `Messages` kann dieses Standard-Verhalten geändert werden, um so die Performance des Systems zu verbessern. Zu diesem Zweck kann die Methode `setDeliveryMode(int pMode)` des `MessageProducer`-Interfaces verwendet werden. Alternativ hierzu kann den `send/publish`-Methoden der Modus als Parameter übergeben werden. Zulässig sind die beiden Werte:

- `javax.jms.DeliveryMode.PERSISTENT`
- `javax.jms.DeliveryMode.NON_PERSISTENT`

### 3.4.3 Message-Prioritäten

Die Priorität einer Message kann den `send/publish`-Methoden als Parameter übergeben werden. Der Wertebereich ist zwischen 0 und 10, Standard ist 4.

### 3.4.4 Ablaufdatum einer Message

Als letztes Argument kann den `send/publish`-Methoden ein Ablaufdatum in Form einer Zahl übergeben werden. Diese wird als Zeit in Millisekunden interpretiert. Die Null bedeutet „bleibt immer gültig“.

### 3.4.5 Antwort-Destinations

Jede Message kann in ihrer Eigenschaft `JMSReplyTo` eine Destination transportieren. Der Empfänger kann dann entweder sich mit der gleichen oder einer unabhängigen Session als Sender registrieren und eine Nachricht an diese Antwort-Destination senden.

### 3.4.6 Temporäre Destinations

Die JMS-Session-Objekte können temporäre Destinations generieren.

- `TemporaryQueue createTemporaryQueue()`
- `TemporaryTopic createTemporaryTopic()`

Diese sind nur während der Lebenszeit der zu Grunde liegenden `Connection` gültig und werden häufig für eine private Anfrage-/Antwort-Szenarien genutzt. So kann ein Sender in der Message Property „replyTo“ eine temporäre Destination angeben, die der Empfänger dann als Antwort-Destination nutzen kann.

### 3.4.7 Beständige Subscriptions

Der JMS-Provider übermittelt alle Messages an die Empfänger, die sich bei einem Topic angemeldet haben. Diese Anmeldungen sind nicht beständig: Wird die Subskription beendet (Methode `close()` des `MessageConsumers`), so werden keine weiteren Messages mehr an diesen Konsumenten übertragen:



Damit beständige Subscriptions benutzt werden können, muss die Client-ID der Connection gesetzt werden:

```
void setClientID(String id);
```

Damit können Messages an diesen Subscriber vom JMS-Provider gespeichert und bei einer erneuten Erzeugung des Konsumenten ausgeliefert werden.



### 3.4.8 Transaktionen

Die `Session` ist in der Lage, flache Transaktionen zu definieren. Dazu muss bei der Erzeugung der `Session` aus der `Connection` heraus der `transactional`-Parameter mit `true` übergeben werden:

```
TopicSession topicSession =
con.createTopicSession(isTransactional, Sessi-
on.AUTO_ACKNOWLEDGE);
```

```
QueueSession queueSession =
con.createQueueSession(isTransactional, Sessi-
on.AUTO_ACKNOWLEDGE);
```

Übermittelt ein Sender an den JMS-Provider Nachrichten innerhalb einer Transaktion, so wartet der Provider auf das Bestätigen der Transaktion und übermittelt erst dann die Nachrichten. Bei einem Rollback werden die Nachrichten nicht übermittelt:

```
topicSession.commit();
queueSession.rollback();
topicSession.commit();
queueSession.rollback();
```

Genau so kann ein Empfänger, der mehrere Nachrichten erhält, den Empfang in einer Transaktion gruppieren.

Was aber natürlich nicht funktionieren kann ist eine Transaktion über einen Sende-Empfangs-Zyklus hinweg: Sendet der Sender eine Nachricht innerhalb einer Transaktion und wartet vor dem `commit()` auf eine Bestätigung des Empfängers, so ist dies eine nicht sonderlich effiziente Art und Weise, den aufrufenden Thread auf immer und ewig zu blockieren: Der Empfänger bekommt die Nachricht ja nie übermittelt.

## **4 Einführung in ActiveMQ**

### **4.1 Übersicht**

ActiveMQ ist ein Top-Level Open-Source-Produkt der Apache Software Group. Die Installation erfordert ein Betriebssystem mit einer Java-Laufzeitumgebung. Die ApacheMQ-Distribution besteht nur aus einer gezippten Datei. Darin enthalten ist eine embedded Datenbank (Apache Kaha) und eine Administrations- und Überwachungs-Weboberfläche.

Kommerziellen Support sowie eine ausführliche Dokumentation im HTML- und PDF-Format bietet die ProgeSS Software mit dem Produkt Fuse.

### **4.2 Die Dokumentation**

Um diese Broschüre mit der jeweils aktuellen Version von ActiveMQ synchron zu halten wird darauf verzichtet, die Dokumentation zu replizieren. Stattdessen wird hier auf die Apache Online-Dokumentation sowie die downloadbare Fuse-Dokumentation verwiesen.

Progress | FUSE

PROGRESS  
SOFTWARE


## ActiveMQ 5.3 Documentation

Welcome to the ActiveMQ 5.3 documentation page. This page contains the documentation for ActiveMQ in HTML and Portable Document Format (PDF). To view PDF files, use Adobe Acrobat Reader, available from the [Adobe web site](#).

### Installation

The following books will help you install ActiveMQ.


[Release Notes](#)  
[Installation Guide](#)



### Getting Started

The following books familiarize you with ActiveMQ, the documentation, and how to approach a few basic use cases.




[Exploring JMS](#)



### Configuring the Message Broker

The following documents describe how to configure the ActiveMQ message broker.


[Configuration Guide](#)  
[ActiveMQ Connectivity Guide](#)  
[Using Persistent Messaging](#)



### Managing the Message Broker

The following documents describe how to manage the ActiveMQ message broker using a variety of tools.


[Administration Guide](#)



### Using Enterprise Integration Patterns

The following books detail how to develop integration solutions using Apache Camel and the Enterprise Integration Patterns described by Gregor Hohpe and Bobby Woolf.



[Implementing Enterprise Integration Patterns](#)



### Client APIs

The following documents describe how to use the various client APIs provided for working with ActiveMQ.

[Using the C++ Client](#)  
[Using the .Net Client](#)





### 4.3 Literatur

Als Literatur kann uneingeschränkt das Buch ActiveMQ in Action aus dem Manning-Verlag empfohlen werden:

