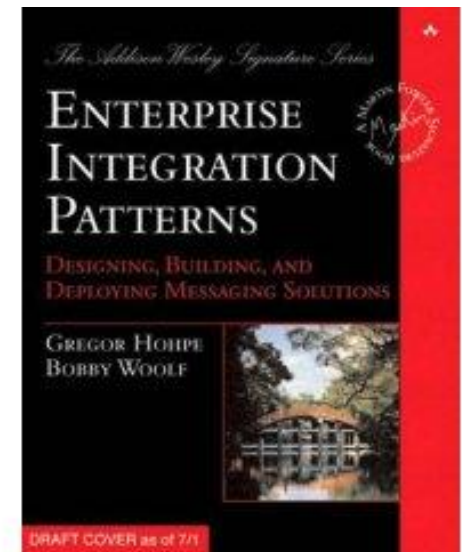




Verteilte Verarbeitung

Kapitel 10

Anwendungsfall Messaging: System-Integration



Systemintegration (EAI)

IT ist teilweise
> 50 Jahre alt

Systeme stark vernetzt

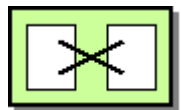
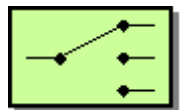
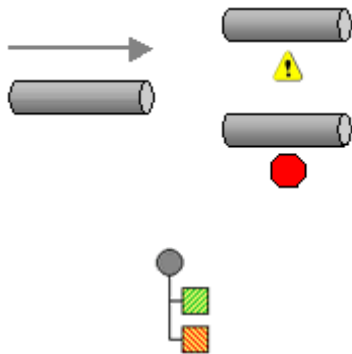
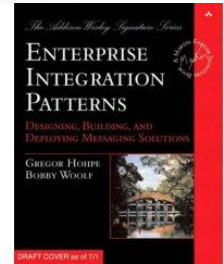
- viele Schnittstellen
- Abläufe über mehrere Systeme

Technisch sehr
heterogen

Quelle: Engels, Humm, ...:
Quasar Enterprise Buch,
dpunkt, 2008

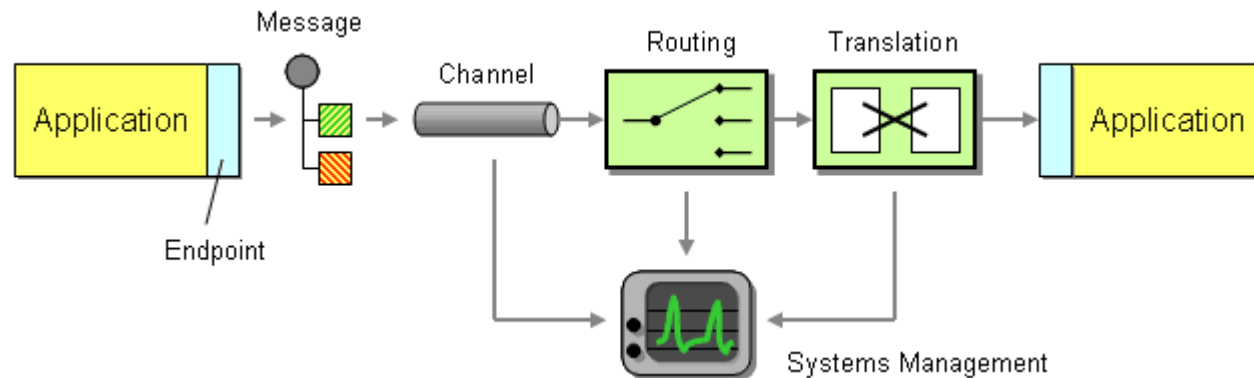
Symbole für diesen Foliensatz

(geklaut von Hohpe und Woolf)



- Queues / Channels
 - Invalid Message Queue,
 - Dead Letter Queue
- Message
- Pipe and Filter
- Message Router
- Message Translator

Beispiel für Verwendung der Symbole



- Einfache Architekturbeschreibung über die Symbole von Hohpe und Woolf (www.eaipatterns.com)
 - Intuitiver als UML oder vergleichbare Notationen
 - Jedes Element = Piktogramm
- Dargestellt: Flussdiagramm für Nachrichten
 - Pfeil = Nachrichtentransport

Agenda

- Typen von Nachrichten
- Pipes and Filter
- Integrationsprogrammierung: Apache Camel
- Enterprise Integration Patterns
 - Message Router
 - Message Filter
 - Message Multiplexer
 - Message Transformer
- Besondere Queues

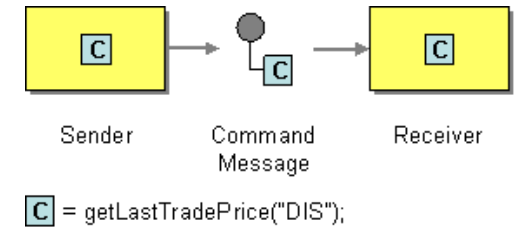
Typen von Nachrichten

Nachricht mit verschiedenen Bedeutungen

- Kommandos
 - Beispiel: Request (anlegenBestellung(...)) , Reply (= OK)
- Dokumente / Daten
 - Beispiel: Bestellung 4711
- Ereignisse
 - Beispiel: Bestellung ist eingegangen
- Kombinationen der Typen möglich, z.B. Ereignis + Dokument

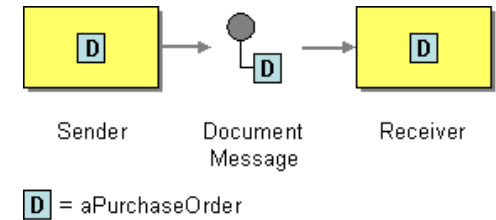
Wichtig: Früh entscheiden, welchen Typ die Nachrichten haben sollen

Nachricht == Kommando



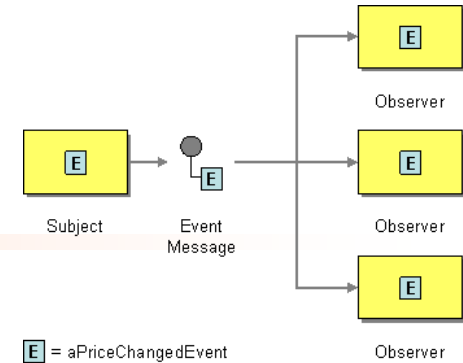
- = „Command-Pattern“ (aus Entwurfsmuster-Buch)
- Beispiele
 - RPC Kommandos, Requests wie (z.B. anlegenBestellung)
 - Steuernde Kommandos (Start, Shutdown, ...)
 - Query Kommando (z.B. findeBestellungen(...))
- Typisch: Zwei Kommunikationspartner
 - Z.B. mit Rollen: Client und Server
- Architekturen
 - RPC basierte Architektur, Client / Server-Architektur

Nachricht == Dokument / Datensatz



- = „Dokument“ z.B. stellt eine Bestellung dar
 - Hat eindeutige Identität (keine Kopien)
 - Unterwegs erweiterbar / transformierbar
 - Technische Darstellung abhängig von Sendern/Empfängern
- Wichtig: *Inhalt der Nachricht*
- Architekturen
 - Workflow zur Daten / Dokument Verarbeitung
 - Pipes and Filter (= Verarbeitungskette)

Nachricht == Ereignis



■ Beispiele

- Hinweis auf Anlegen, Ändern, Löschen bestimmter Daten (Preise, Produkte, Bestellungen, ...)

■ Nützlich für lose Integration (jeder hat eigene Daten)

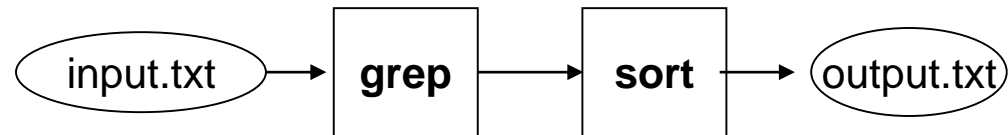
- Sender muss die Empfänger nicht kennen
- Empfänger können sich für ein Ereignis registrieren

■ Wichtig: Typ des Ereignisses, Inhalt weniger wichtig

■ Architekturen

- Observer-Pattern nachbaubar (Publish/Subscribe)
- Varianten: Push und Pull-Modell (siehe unten)

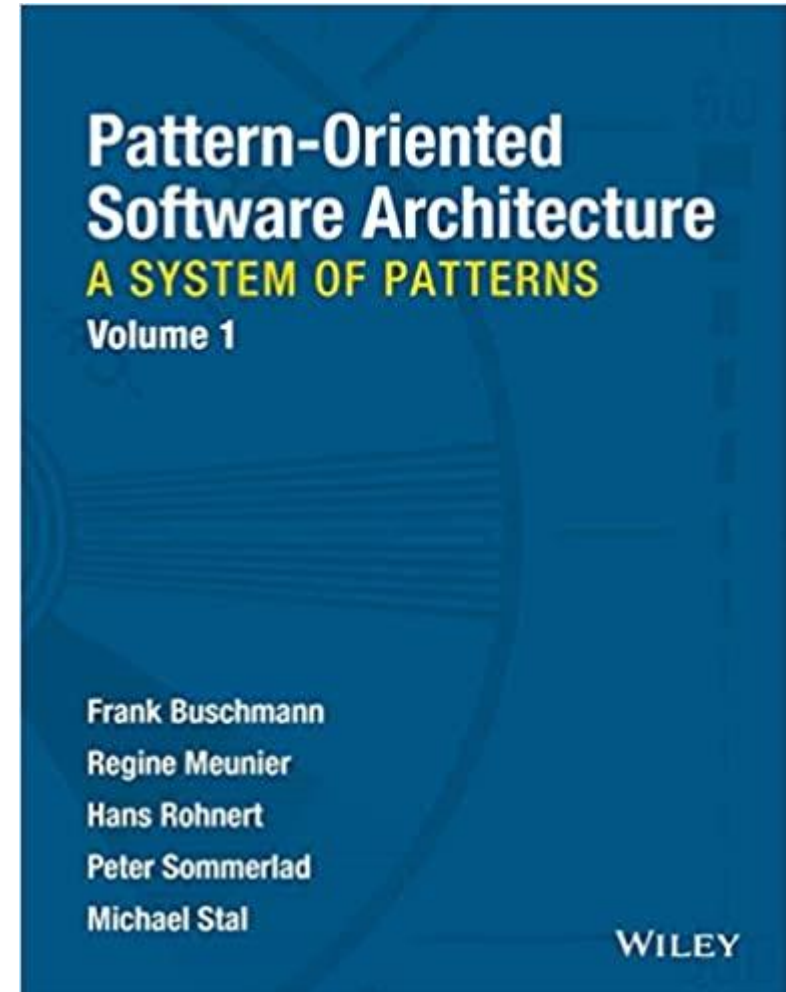
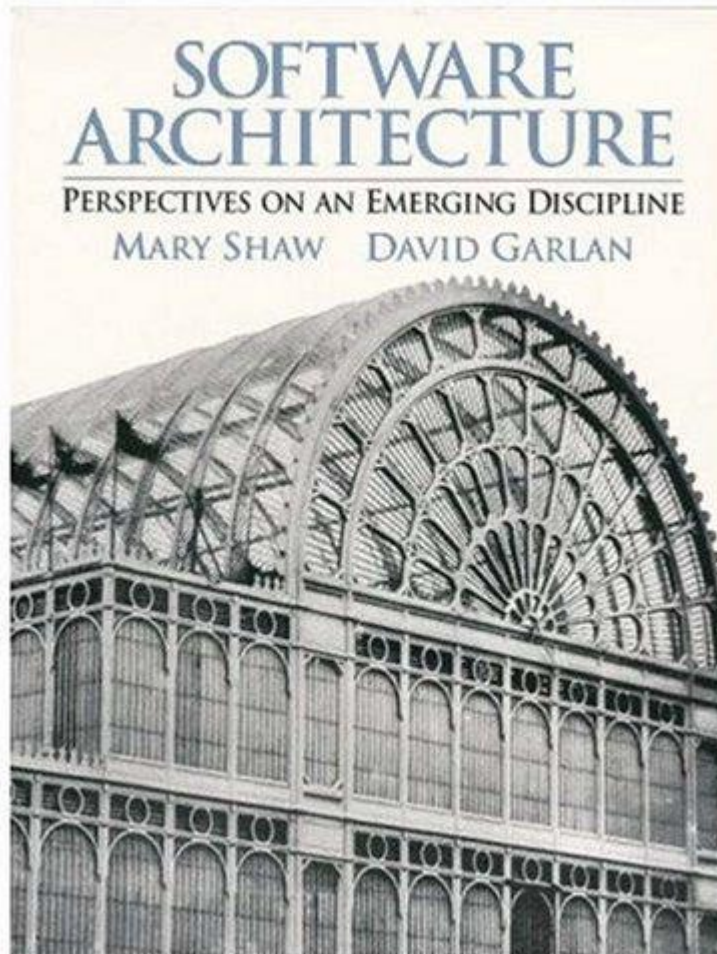
Pipes and Filter



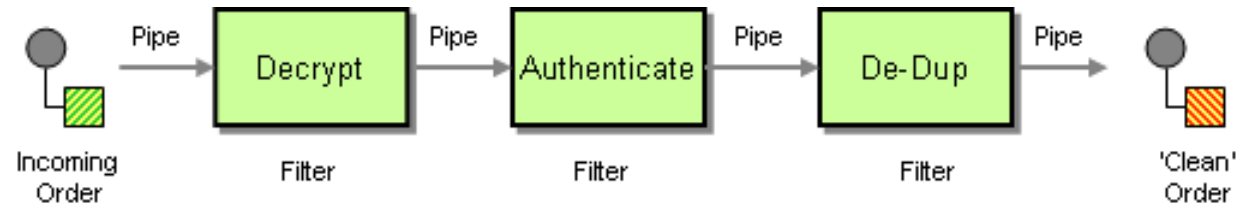
- Altes Architekturmuster (Shaw/Garlan, POSA1)
- Pipe
 - = unendlicher Datenstrom
 - Überträgt Daten zwischen zwei Filtern, Datenquellen, Datensenken
 - Synchronisiert ggf. Quelle und Senke
 - Puffert ggf. Daten
 - Beispiele: Unix-Pipe, Socket-Verbindung, Message Queue
- Filter
 - Kapselt Verarbeitungsschritt: ändert Daten/ filtert Daten
- Beispiele Pipes & Filter
 - Unix Shell: `cat input.txt | grep "text" | sort > output.txt`
 - Streams, Stream Hierarchie in Java:
z.B. `BufferedReader` = „Filter“

Architekturpatterns

Pipe & Filter, Layered Arch., Blackboard Arch.



Pipes and Filters



- Anwendung: Verknüpfung von Verarbeitungsschritten bei ankommenden / gesendeten Nachrichten
 - Pro Verarbeitungsschritt ein Filter
 - z.B. Entschlüsselung, Authentisierung, Duplikate finden
 - z.B. Komprimierte Kommunikation: zip / unzip Filter
- Architekturvarianten
 - Koppelung der Filter über Queues (Integration von Applikationen)
 - Koppelung der Filter über Methodenaufruf (Decorator Pattern)
- Leistung
 - **Entkoppelung** der Filter-Implementierungen
 - **Flexible Gestaltung des Protokolls**
 - Verschlüsselt?, Komprimiert?, Zuverlässig?
 - Einstellbar über verwendete Filter
 - Flexible Gestaltung des Nachrichtenformats
- Beispiel für Umsetzung: WCF- Channel Stack

Agenda

- Grundlegende Patterns
 - Typen von Nachrichten
 - Pipes and Filter
- Integrationsprogrammierung: Apache Camel
- Enterprise Integration Patterns
 - Message Router
 - Message Filter
 - Message Multiplexer
 - Message Transformer
- Besondere Queues

Programmierung mit Messaging Patterns

z.B. Apache Camel

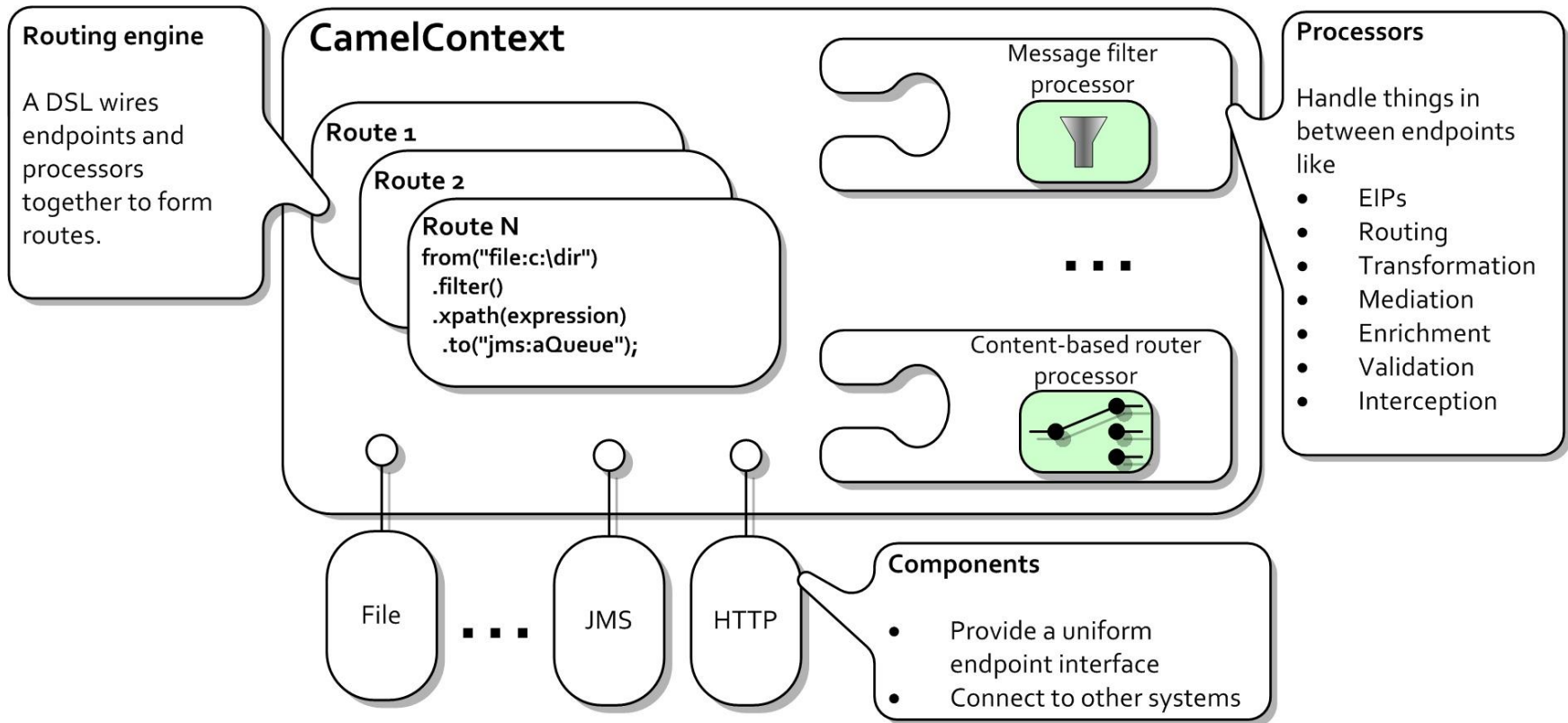
- implementiert die Messaging Patterns unabhängig von der Middleware
- Bietet Java Integrations DSL („fluent Interface“)
- Gut integriert mit anderen Apache Projekten



APACHE®
Camel

Architektur von Apache Camel

Zentrale Konzepte



Camel Context

= Laufzeitumgebung

```
CamelContext context
    = new DefaultCamelContext();
// ...

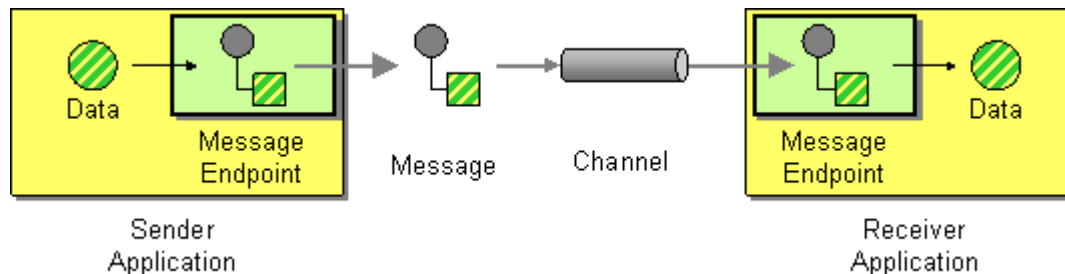
context.start();
Thread.currentThread().sleep(10000);
context.stop();
```


Routen: Nachrichten verschieben

Camel pollt das Verzeichnis data/inbox und verschiebt die Dateien nach data/outbox

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
    public void configure() throws Exception {
        from("file:data/inbox")
        .to("file:data/outbox");
    }
});
```

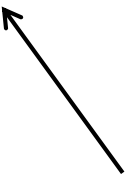
URI eines Endpunkts



Prozessoren: Nachrichten verarbeiten

Camel pollt das Verzeichnis data/inbox und gibt die Dateinamen und –inhalt aus

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
    public void configure() throws Exception {
        from("file://data/inbox?noop=true")
            .process(new Processor() {
                public void process(Exchange exc)
                    throws Exception {
                    System.out.println("Processing: " +
                        exc.getIn().getHeader("CamelFileName"))
                        + exc.getIn().getBody(String.class));
                }
            });
    }
});
```

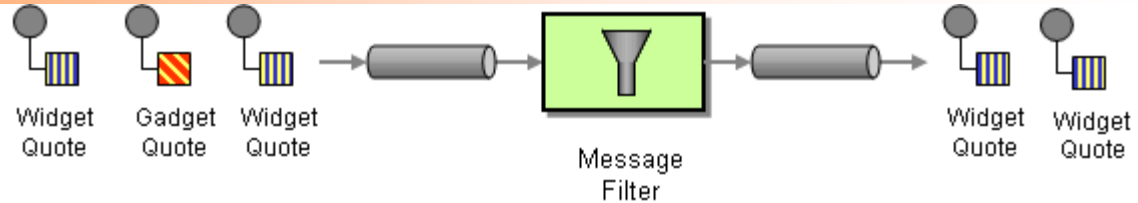


Inhalt der Datei / Nachricht
Analysieren / Ändern / Loggen

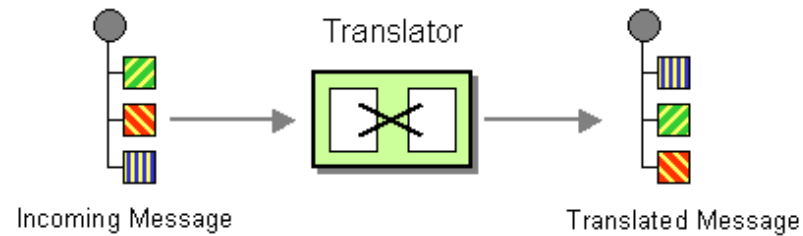
EAI Patterns

EAI-Patterns

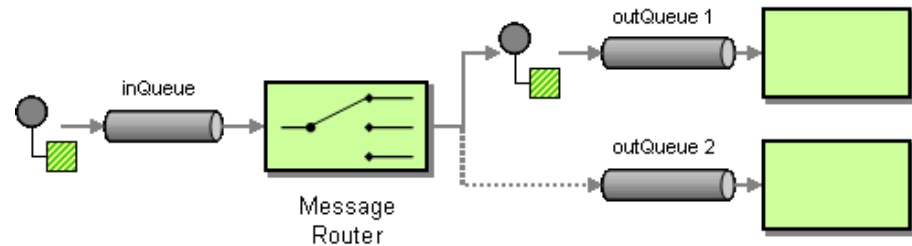
Filter



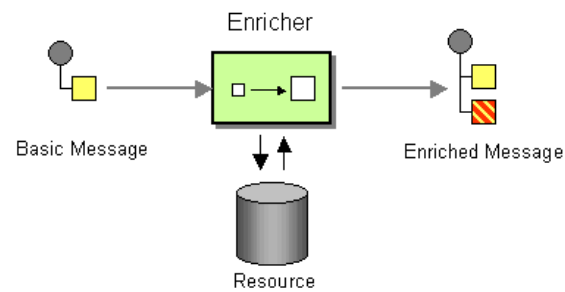
Translator



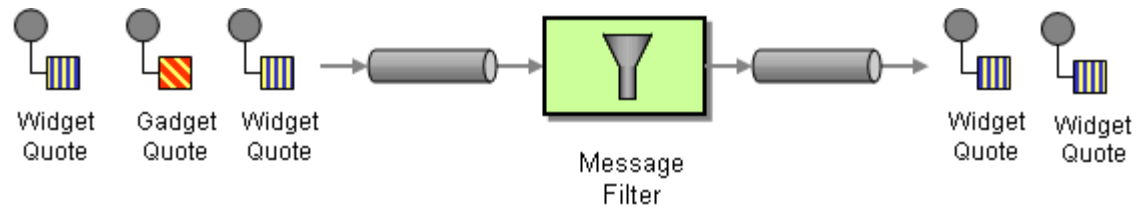
Router



Enricher



Message Filter



- Integrationsproblem: Empfänger will nicht alle Nachrichten erhalten, sondern nur bestimmte
- Lösung: Filter entfernt bestimmte Nachrichten
- Anwendungen:
 - z.B. Entfernen von Test-Nachrichten
 - z.B. Entfernen von erkannten DoS-Angriffen
 - z.B. Doppelte Nachrichten entfernen

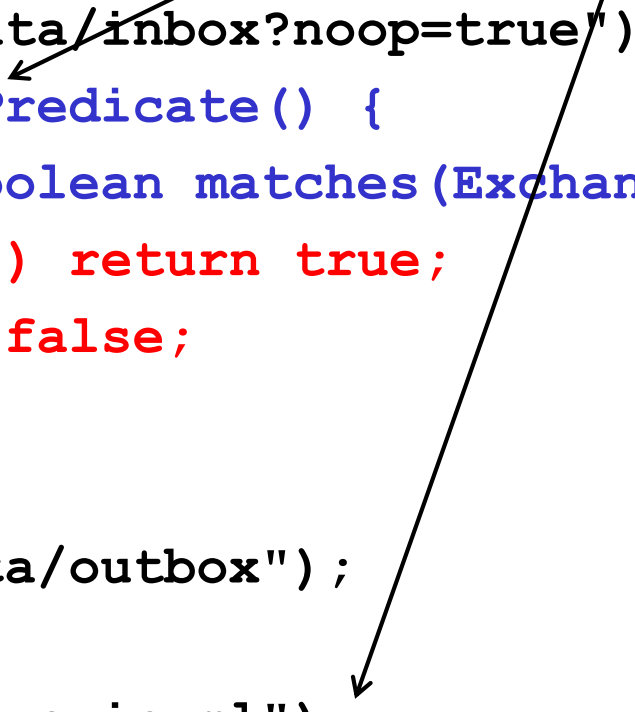
Message Filter mit Camel

Zwei Beispiele:

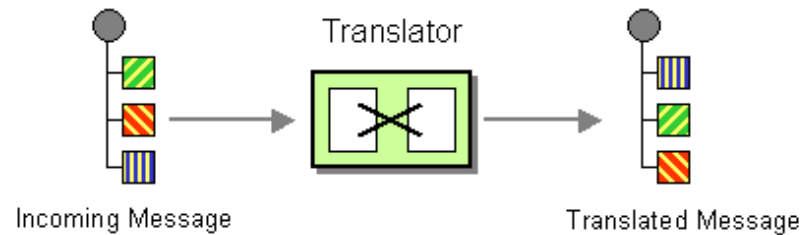
1. Filtern über Prädikat
2. Filtern über jsonpath/xpath

```
from("file:data/inbox?noop=true")
  .filter(new Predicate() {
    public boolean matches(Exchange exc) {
      if ( ... ) return true;
      return false;
    }
  })
  .to("file:data/outbox");

from("jms:queue:inxml")
  .filter(jsonPath("$.positionen[?(@.lieferbar)]"))
  .to("amqp:/bestellungen?queue=bestellungqueue");
```

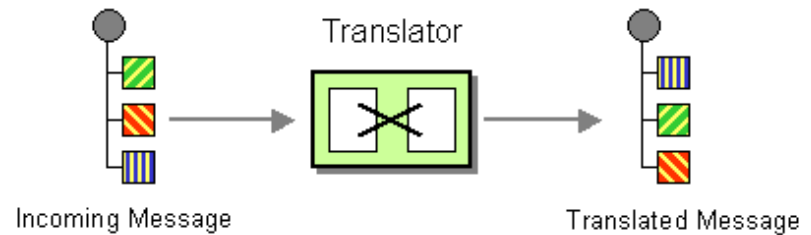


Message Translator



- Typisches Integrationsproblem: System A sendet Daten in einem Format, das System B nicht versteht
- Lösung: Translator/Adapter übersetzt Nachrichten bei Übertragung
- Ebenen der Transformation
 - Datenstrukturen = Strukturelle Änderungen
(z.B. Reduktion der Vererbungstiefe, Verschiebung Attribut, ...)
 - Datentypen = Transformation von Attributnamen, Datentypen, Wertebereichen, Aufzählungstypen, Kürzeln
(z.B. Postleitzahl von int -> string)
 - Datenrepräsentationen
= Datenformat Änderungen, Kompression, Verschlüsselung
(z.B. XML <-> Name/Wert-Paare, XML<->CDL, ASCII <-> Ebcdic, ...)
 - Transportprotokolle
(z.B. TCP-Socket -> HTTP, EDI -> ...)

Message Translator /2



■ Vorteile

- Sender und Empfänger müssen sich nicht über Nachrichtenformat einig sein
- Quelltexte von Sender und Empfänger müssen ggf. nicht geändert werden (solange die Nachrichten die richtigen Infos enthalten)

■ Implementierung

- z.B. XML -> XML / XML -> anders Textformat über XSLT
- z.B. über vorgefertigte Adapter und Transformatoren
- z.B. manuell ausprogrammiert

Message Translator in Camel

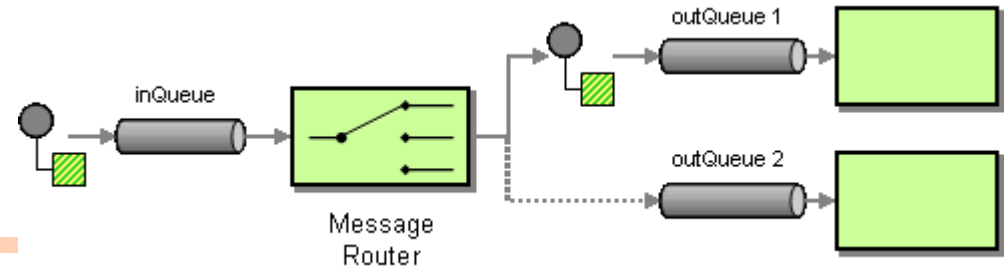
Zwei Beispiele:

1. Transformieren von XML über Style Sheet (XSLT)
2. Transf. über Processor

```
from("file://data/inbox?noop=true")
.to("xslt://test.xsl")
.to("file://data/outbox");
```

```
from("file://data/inbox")
.process(new Processor() {
    public void process(Exchange exc)
        throws Exception {
        String body = exc.getIn().getBody(String.class);
        body = body.replaceAll("KundeRoot", "Kunde");
        exc.getIn().setBody(body);
    }
})
.to("file://data/outbox");
```

Message Router



- = (Zwischen-)Komponente entscheidet über Nachrichtenfluss
 - Kann „gesteuertes“ Publish/Subscribe sein
 - Entkoppelt Sender von Empfänger-Queues
- Kriterien für das Zustellen von Nachrichten:
 - Properties der Nachrichten (Filtern)
 - Typ/Body der Nachrichten
 - Infos zur aktuellen Systemlast oder zu Ausfällen
 - (Absender der Nachricht)
- Anwendungen
 - Lastverteilung auf verschiedene (identische) Server
 - Behandlung von Ausfällen/Wartung von (identische) Servern
- Nachteil
 - Performanceverlust durch zusätzliche Verarbeitung
 - Ggf. Koppelung des Routers an die Empfänger (wg. Interpretation der Nachrichten)
- Ausblick: Router = Workflow Engine

(Content based) Router mit Camel

Camel fragt das Property „CamelFileName“ im Header ab und entscheidet dann, in welche Queue die Nachricht weitergeleitet wird.

```
from("file://data/inbox")
    .choice()
    .when(header("CamelFileName").endsWith(".json"))
        .to("amqp://json?queue=jsonqueue")
    .when(header("CamelFileName").endsWith(".csv"))
        .to("amqp://csv?queue=csvqueue")
    .otherwise()
        .to("file://data/outbox/other")
    .end();
```

Routing auf der Grundlage des Inhalts

```
from("file://data/inbox?noop=true")
  .filter((exc) -> { return exc.getIn()
    .getHeader("CamelFileName", String.class)
    .endsWith(".json"); })

  .choice()
  .when().jsonpath("[?($.kunde=='TH-Rosenheim')]")
    .to("amqp://rosenheim?queue=rosenheimqueue")
  .when().jsonpath("[?($.kunde=='TH-Deggendorf')]")
    .to("amqp://deggendorf?queue=deggendorfqueue")
  .otherwise()
    .to("file://data/outbox/other")
  .end();
```

Literatur

