KONZEPTION UND REALISIERUNG VON DATEN- UND FUNKTIONSINTEGRATIONSSZENARIEN MIT HILFE DES EIP-FRAMEWORKS APACHE CAMEL

SVEN EISENHAUER
Matrikel-Nr.: 707173

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Daten- und Systemintegration Wintersemester 2009/2010

Fachbereich Informatik

Hochschule Darmstadt

Januar 2010

ABSTRAKT

Die Integration bestehender Systeme bzw. Daten stellt immer eine neue und eigenständige Aufgabe dar, da sich der Bestand an zu integrierenden Daten oder Systemen von Szenario zu Szenario unterscheidet. Bei der Integration vieler Systeme sind somit viele eigenständige Aufgaben zu lösen. Dies führt zu einer unstrukturierten Sammlung unterschiedlicher, eigenständiger Integrationslösungen. Diese Arbeit untersucht, wie sich Integrationslösungen besser strukturieren lassen. Der hierbei verfolgte Ansatz basiert auf der Tatsache, daß trotz der Eigenständigkeit jeder Integrationslösung, es oftmals wiederkehrende Gemeinsamkeiten zwischen ihnen gibt. Für eine Vielzahl dieser Gemeinsamkeiten existieren Muster, sog. Enterprise Integration Patterns (EIPs), mit denen sich wiederkehrende Integrationsaufgaben beschreiben lassen. Zur Veranschaulichung soll ein Integrationsszenario mittels eines EIP-Diagramms abstrakt beschrieben werden. Weiterhin soll aufgezeigt werden, wie die Implementierung einer so entworfenen Integrationslösung auf Basis von Apache Camel aussehen kann. Diese Implementierung soll schließlich Erkenntnisse über die Tauglichkeit von Apache Camel als Plattform für Funktions- und Datenintegrationslösungen liefern.

INHALTSVERZEICHNIS

1 EINLEITUNG

Nahezu jedes Unternehmen in der heutigen Zeit unterstützt seine Wertschöpfung durch IT-Systeme in vielfältigen Bereichen. Allerdings existiert kein IT-System, das alle IT-Aufgaben in Unternehmen allein abdeckt. So existieren meistens unterschiedliche IT-Systeme in Unternehmen für unterschiedliche Aufgaben. Oftmals ergänzen sich verschiedene IT-Systeme in Unternehmen zur Erfüllung der Aufgaben.

Im Zuge der Optimierung oder Umgestaltung von Geschäftsprozessen ergibt sich der Bedarf, Daten oder Funktionalität eines IT-Systems in einem anderen IT-System zu verwenden, was als Integration bezeichnet wird. Da es sich dabei um die Integration von Unternehmensanwendungen bzw. -daten handelt, spricht man im Englischen auch von enterprise integration.

Integrationsarten auf unterschiedlichen Ebenen

Diese Integration kann technisch auf unterschiedlichen Ebenen stattfinden. Auf welcher Ebene eine Integration stattfindet hängt im Einzelfall von vielen Faktoren ab, es gibt keine „beste" Integrationsebene. Vielmehr verlangt jedes Integrationsszenario eine Abwägung von Argumente auf deren Basis eine Entscheidung für die zu wählende Integrationsebene getroffen wird. Dabei spielen sowohl technische als auch wirtschaftliche Aspekte eine Rolle. Oftmals lassen technische Einschränkungen der bestehenden Systeme nur die eine oder andere Ebene der Integration zu. Die verschiedenen Integrationsarten unterscheiden sich unter anderem durch die Komplexität der Realisierung oder die nötigen Eingriffe in bestehende Anwendungen. Diese unterschiedliche Komplexität spiegelt sich auch in unterschiedlichen Aufwänden für die Implementierung wider. Auf folgenden Ebenen kann die Integration von Daten oder Funktionen erfolgen:

DATENINTEGRATION auf dieser Ebene nutzt eine Anwendung die Daten einer anderen Anwendung. Diese Integrationsart erfolgt oftmals durch eine Replikation der Daten. Auch ETL-Werkzeuge (Extract Transform Load) kommen hier oft zum Einsatz, periodisch Daten aus einem Quellsystem zu exportieren, ggf. in das nötige Zielformat zu transformieren und schließlich in das Zielsystem zu laden.

FUNKTIONSINTEGRATION dabei verwendet eine Anwendung eine bereits vorhandene Funktion einer anderen Anwendung. Hierfür sind Eingriffe in bestehende Systeme notwendig, weshalb die Komplexität als vergleichsweise hoch einzuschätzen ist.

OBERFLÄCHENINTEGRATION hierbei werden Benutzungsschnittstellen bestehender Anwendungen in einer neuen Benutzungsschnittstelle zusammengefasst.

PROZESSINTEGRATION bei dieser Art der Integration verfolgt der Integrator das Ziel, die Funktionen unterschiedlicher Anwendungssystem entlang der Geschäftsprozesse zu integrieren, so dass sie

die Geschäftsprozesse möglichst gut unterstützen. Gemeinhin wird diese Art der Integration als die komplexeste angesehen.

Entwurfsmuster

In der Software-Entwicklung haben sich über die Zeit Entwurfsmuster (Design Pattern) für ähnliche Problemstellungen etabliert. Diese beschreiben ein Problem in seinem Kontext und skizzieren eine bewährte Lösung für ähnliche Probleme.

Da auch bei der Integration von Unternehmensanwendungen oft ähnliche Probleme auftreten, haben sich auch hier mit der Zeit entsprechende Lösungsmuster, Enterprise Integration Patterns(kurz EIPs), etabliert.

2 ENTERPRISE INTEGRATION PATTERNS

2.1 ZIELSETZUNG VON ENTERPRISE INTEGRATION PATTERNS

Enterprise Integration Patterns (EIPs) stellen eine intuitive Beschreibung von Integrationsszenarien dar. Sie dienen der möglichst einfachen Kommunikation von Integrationskonzepten auf einer symbolischen und logischen Ebene. Sie sollen so die Kommunikation mit anderen Projektbeteiligen, wie Mitarbeitern von Fachabteilungen, Managern und anderen Entwicklern vereinheitlichen. EIPs liefern eine hersteller- und technologieneutrale Darstellung und Nomenklatur. Im Gegensatz zu UML-EAI OMG [10] liegt hierbei der Schwerpunkt auf einer intuitiven Darstellung anstatt architektonischer Vollständigkeit bei reduzierter Komplexität und Detailgenauigkeit. Deshalb existieren für die meisten EIP standardisierte Symbole zur Verwendung in Diagramme, welche die zu konstruierende Integrationslösung beschreiben.

2.2 ÜBERSICHT ÜBER DEFINIERTE EIPS

Bereits definierte Enterprise Integration Patterns entstanden aus praktischen Erfahrungen vieler verschiedener Integratoren. Dieser Definitionsprozeß wird stetig weitergeführt, weshalb die Menge verfügbarer EIPs nicht abgeschlossen ist.

Die Abbildung 1 zeigt eine Übersicht über einige der zum Zeitpunkt der Erstellung dieser Arbeit verfügbaren EIP. Hohpe and Woolf [9] stellt eine Vielzahl von EIPs detailliert vor.

2.3 AUFBAU EINES EIP

Neben dem Symbol enthält die Beschreibung eines EIPs weitere wichtige Elemente:

NAME: Der eindeutige Bezeichner des EIPs.

KONTEXT: Er beschreibt das Umfeld der Problemstellung. für die das EIP geeignet ist.

PROBLEM: Hierbei handelt es sich um eine textuelle Beschreibung des Problems, zu dessen Lösung das EIP dient.

LÖSUNG: Eine textuelle Beschreibung des Lösungsansatzes, der durch das EIP abgedeckt wird.

SYMBOL: Eine grafische Darstellung des EIP zur Verwendung in Diagrammen.

BEISPIEL: Für viele EIP existieren kleine Beispielimplementierungen, aus denen sich die geeignete Verwendung des EIP erschließen läßt.

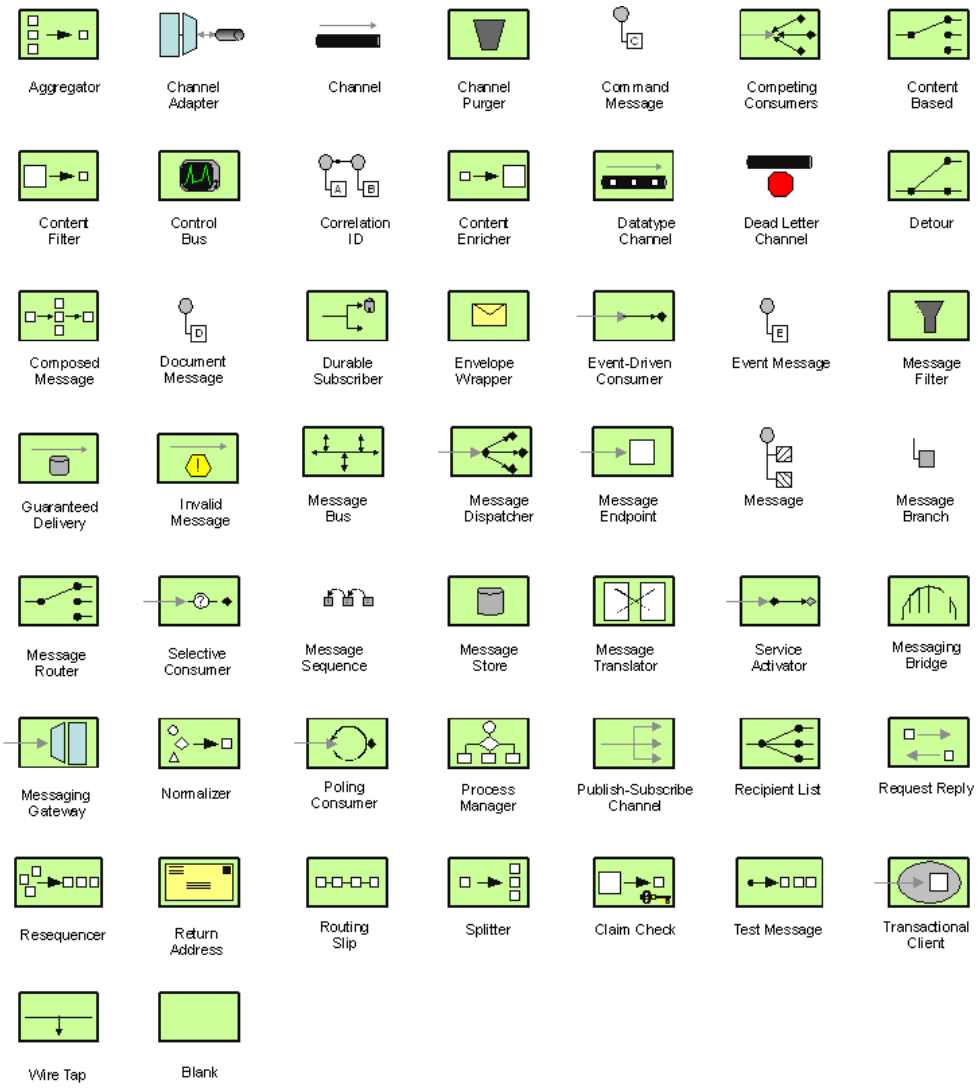

Abbildung 1: Übersicht EIP

3 APACHE CAMEL

3.1 BESCHREIBUNG

Die Internetseite zum Projekt [1] beschreibt Apache Camel als Opensource Integrationsframework für EIPs. Die Implementierung verwendet Java als Programmiersprache und legt besonderes Augenmerk auf eine starke Modularisierung des Frameworks. Dies führt zu einem Plugin-Konzept, das es dem Entwickler erlaubt, nur die Camel-Kompenenten seiner Lösung hinzuzufügen, die er für sie tatsächlich benötigt. In der Termininolgie des Camel Projekts tauchen sehr oft die Begriffe Component, im Deutschen Komponente, und Route auf. Zur Implementierung einer Integrationslösung auf Basis von Camel verbinden eine oder mehrere Routen mehrere Komponenten als eine Abfolge von Routing- und Mediationsregeln. Innerhalb einer Route lassen sich Komponenten über URIs referenzieren. Eine Route besteht aus dem Start- und einem Endpunkt, die jeweils durch Komponenten repräsentiert werden. Die meisten Kompontentypen stehen in eigenen Plugins zur Verfügung, wodurch der Entwickler nur die Plugins benötigt, deren Komponenten er verwendet. Dieses Konzept erleichert die Erweiterung von Camel um weitere Komponenten.

Da Apache Camel auf Maven [2] als Build-System basiert, lassen sich zusätzlich Plugins sehr einfach über die zentrale Steuerungsdatei vom Maven (pom.xml) per dependency injection zum eigenen Projekt hinzufügen. Maven versucht neue Abhängigkeiten aufzulösen und die entsprechenden Artefakte, meistens jar-Dateien, aus zentralen Repositories im Internet herunterzuladen. Maven ist kein Bestandteil von Camel, weshalb es separat von [2] heruntergeladen und installiert werden muss.

Der Kern von Apache Camel und die Plugins stehen auf [1] zum Herunterladen bereit. Dort finden sich auch die Einträge für die Datei pom.xml, um ein Plugin als Abhängigkeit zu definieren und es so dem eigenen Projekt hinzuzufügen. Maven übernimmt dann automatisch das Herunterladen der benötigten Dateien. Leider würde eine tiefergehende Betrachtung von Maven den Rahmen dieser Arbeit überschreiten.

Das zentrale Element einer Integrationslösung auf Basis von Apache Camel ist ein Java-Objekt vom Typ *CamelContext*. Diesem Objekt fügt der Integrator Routen hinzu. Routen lassen sich in einer Java-Klasse definieren, die von der Klasse *RouteBuilder* erbt und die Methode *configure* überschreibt. Nachdem der *CamelContext* über alle Routen verfügt läßt er sich durch den Aufruf der Methode *start* starten bzw. durch die Methode *stop* beenden. Nach dem Starten des CamelContextes lassen sich Nachrichten entlang der definierten Routen verschicken und verarbeiten.

3.2 GRUNDKONZEPT ZUR IMPLEMENTIERUNG VON EIPS

Message Channel: Apache Camel stellt das EIP Message Channel als interne Implementierung des Interfaces Endpoint bereit.

Komponente	URI	Beschreibung
Direct	„direct:destination"	Synchroner Aufruf eines internen Endpunkts im gleichen Kontext
JMS	„jms:[topic:]destination"	Anbindung an ein Java Messaging System wie z. B. ApacheActiveMQ
JBI	„jbi:serviceName"	Anbindung an ein JBI-System wie z. B. Apache ServiceMix
JDBC	„jdbc:dataSourceName?options "	Zugriff auf JDBC Ressourcen
SEDA	„seda:destination"	Asynchroner Aufruf eines internen Endpunkts im gleichen Kontext
Bean	„bean:beanRef"	Übergabe an einfache Java Objekte
CXF	„cxf:address[?serviceClass=...] "	Integration von CXF basierten Webservices
Velocity	„velocity:someTemplateResource"	Verarbeitung einer Nachrichts mittels Velocity Vorlagen
SMTP	„smtp://user-info@host[:port]"	Versenden von E-Mails mittels SMTP und JavaMail

Tabelle 1: Endpunkte und ihre URIs

Message Endpoint: Dieses EIP läßt sich durch Implementieren des Interfaces Endpoint realisieren. Im Normalfall verwendet eine Camel-Lösung dieses EIP über URIs in Routen.

Message: Das Interface Message dient der Bereitstellung des EIP Message. Dabei lassen sich grundsätzlich zwei Typ von Nachrichten unterscheiden. Eingehende Nachrichten vom Typ „InOnly" wie z. B. ereignisgesteuerte Nachrichten und Nachrichten vom Typ „InOut", deren Nachrichtenaustausch eine Antwortnachricht enthält.

3.3 AUGEWÄHLTE ENDPUNKTE UND IHRE URIS

Die Tabelle 1 zeigt eine Auflistung interessanter Endpunkte, die Apache Camel bereitstellt. Eine vollständige Liste findet sich auf [1]. Weiterhin erlaubt es Camel eigene Endpunkte zu implementieren.

3.4 IMPLEMENTIERUNGSALTERNATIVEN

Apache Camel bietet dem Entwickler zwei Möglichkeiten, Routen zu definieren.

JAVADSL: Mit dieser Variante der Routendefinition lassen sich in eine sog. domain specific language (DSL) formulieren. Diese Art der Formulierung ähnelt herkömmlichen Methodenaufrufen in Java und läßt sich in gängigen Entwicklungsumgebungen verwenden. Routen lassen sich somit programmieren. Eine weitere bedeutende Eigenschaft der JavaDSL ist die Typen-Sicherheit, da die implementierende Java-Klasse vor dem Start kompiliert werden muss. Somit kann der Javacompiler die formale Korrektheit der Routendefinition überprüfen.

XML: Bei dieser Methode werden die Routen in einer XML-Datei beschrieben und zur Laufzeit beim Start des Systems vom Spring-Framework [7] umgesetzt. Hierbei steht eine enge Verzahnung mit dem Spring Framework, das u. a. die Lebenszyklen der verwendeten Objekte steuert. Eine ausführliche Betrachtung des Spring Frameworks würde leider den Rahmen dieser Arbeit überschreiten.

3.5 GENERIERUNG EINES APACHE CAMEL ARTEFAKTS

In der Windows-Konsole in einem leeren Verzeichnis erzeugt Maven mit dem Kommando

```
mvn archetype:generate -DarchetypeGroupId=org.apache.camel.
    archetypes -DarchetypeArtifactId=camel-archetype-java -
    DarchetypeVersion=2.0.0 -DgroupId=de.h_da.fbi.dsi.ws0910
    -Dpackage=de.h_da.fbi.dsi.ws0910.camelprototype -
    DartifactId=camelprototype
```

ein neues Artefakt, das als Ausgangspunkt für die zu entwickelnde Apache Camel Integrationslösung dienen kann. Dabei sind die Parameter artifactId und package entsprechend anzupassen. Das Artefakt enthält eine vordefinierte Beispielroute und läßt sich direkt mit dem Kommando `mvn camel:run` ausführen.

4 PROTOTYPISCHE IMPLEMENTIERUNG

Anhand des hier beschriebenen fiktiven Geschäftsprozesses veranschaulichen die nachfolgenden Abschnitte verschiedene Implementierungen von Funktions- und Datenintegration, deren Konzepte in Form von EIPs dargestellt sind, mittels Apache Camel.

4.1 BESCHREIBUNG DES INTEGRATIONSSZENARIOS

In diesem Abschnitt soll ein fiktives Szenario zur Integration mehrerer Anwendung auf Funktions- und Datenebene beschrieben werden.

4.1.1 *Der Beispielgeschäftsprozeß*

Dem Integrationskonzept und der prototypischen Implementierung liegt ein einfacher fiktiver Geschäftsprozeß zugrunde, der einen Bestellvorgang beschreibt.

Die Bestellungen durch Kunden erreichen das System auf zwei Wegen. Erstens sollen Dateien im CSV-Format in einem bestimmten Verzeichnis beispielsweise per ftp abgelegt werden. Dabei folgen die Dateinamen der Konvention <Kundennummer>.csv. Jede Zeile in der CSV-Datei stellt eine Bestellposition im Format <Artikel-Nummer>,<Artikel-Bezeichnung>,<Bestellmenge> dar. Das Verzeichnis soll regelmäßig auf neue Dateien überprüft werden. Neue Dateien sollen automatisch verarbeitet werden.

Desweiteren soll eine Webservice-Schnittstelle bereit gestellt werden, über die Kunden Bestellungen senden können. Bei Bestellungen auf diesem Weg soll dem Kunde die Bestellnummer der neuen Bestellung zurückgegeben werden.

Diese Bestellnummer wird von einem auf Funktionsebene zu integrierenden Legacy-System erzeugt.

Nach Verarbeitung einer Bestellung durch das Legacy-System soll eine E-Mail zur Benachrichtigung mit den Bestelldaten an eine bestimmte Adresse versendet werden. Jede verschickte E-Mail soll in einem bestimmten Verzeichnis als Sicherungsdatei aufbewahrt werden.

4.1.2 *Darstellung als EIP Diagramm*

Die Abbildung 2 zeigt das Integrationsszenario zur Realisierung des Prototyps unter Verwendung der EIP-Symbolik. Die nachfolgende Beschreibung der Implementierung orientiert sich an diesem Diagramm, ebenso wie die Implementirung selbst. Zwar existiert kein Verfahren zur direkten Ableitung einer Camel-Implementierung aus einem EIP-Diagramm, allerdings lassen sich Teile der Implementierung anhand eines Diagramms manuell identifizieren. Ebenso erhält der Integrator einen „roten Faden" zur Implementierung der Routing- und Mediationsregeln.

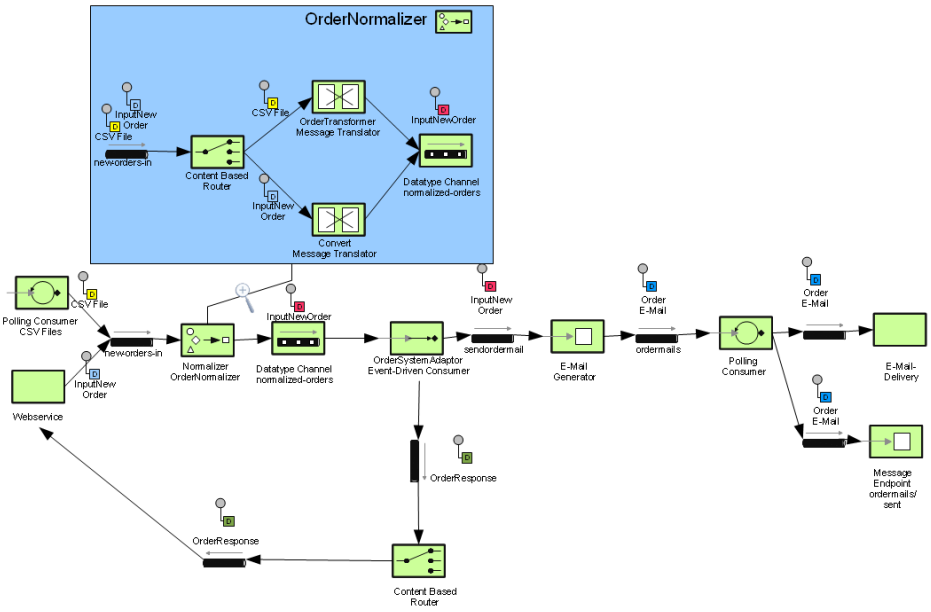

Abbildung 2: Integrationsszenario

4.2 APACHE CAMEL IMPLEMENTIERUNG

4.2.1 *Konzept der Implementierung*

Wie bereits in Abschnitt 3.4 angemerkt, bestehen zwei Möglichkeiten, wie sich Integrationslösungs mit Apache Camel implementieren lassen. Die hier vorgestellte Implementierung verwendet größtenteils die JavaDSL, da die Verwendung der XML-Konfiguration eine Betrachtung des Spring 2 Frameworks, die den Rahmen dieser Arbeit überschreiten würde, erfordern würde. Durch Verwendung der JavaDSL ist es möglich eine Integrationslösung in einer Entwicklungsumgebung wie z. B. Eclipse zu schreiben bzw. zu programmieren. Dabei stehen dem Integrator alle Hilfsmittel der Entwicklungsumgebung wie z. B. Auto-Vervollständigung zur Verfügung. Eine Datenintegration erfolgt innerhalb dieses Prototyps an allen Stellen, an denen das Format von Daten verändert wird. So z. B. innerhalb der Transformer im Order-Normalizer 4.2.4. Der Prototyp verzichtet dabei bewußt auf jede Form der Ausnahme- und Fehlerbehandlung zugunsten der Einfachheit und Übersichtlichkeit.

4.2.2 *Einbindung der Webservice-Schnittstelle*

Apache Camel bietet mittels entsprechender Plugins die Möglichkeit, eine Webservice-Schnittstelle direkt als Endpunkt zu definieren.

Listing 4.1: OrderService.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    " xmlns:tns="http://camelprototype.ws0910.dsi.fbi.h_da.de"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http:
```

```
  //www.w3.org/2001/XMLSchema" name="OrderService"
  targetNamespace="http://camelprototype.ws0910.dsi.fbi.h_da.de">
<wsdl:types>
  <xsd:schema targetNamespace="http://camelprototype.ws0910.dsi.
      fbi.h_da.de">
    <xsd:element name="inputNewOrder">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="customerNo" type="xsd:string" />
          <xsd:element name="orderpositions"
            type="tns:orderPosition" maxOccurs="unbounded"
            minOccurs="1">
          </xsd:element>

        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="outputNewOrder">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="status" type="xsd:string"></
              xsd:element>
          <xsd:element name="message" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>


    <xsd:complexType name="orderPosition">
      <xsd:sequence>
        <xsd:element name="article" type="xsd:string"></
            xsd:element>

        <xsd:element name="amount" type="xsd:int"></xsd:element>
        <xsd:element name="articleNo" type="xsd:string"></
            xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
</wsdl:types>


<wsdl:message name="inputNewOrder">
  <wsdl:part name="in" element="tns:inputNewOrder"></wsdl:part>
</wsdl:message>
<wsdl:message name="outputNewOrder">
  <wsdl:part name="out" element="tns:outputNewOrder"></wsdl:part>
</wsdl:message>
<wsdl:portType name="OrderEndpoint">
  <wsdl:operation name="CreateOrder">
    <wsdl:input message="tns:inputNewOrder" name="input"/>
    <wsdl:output message="tns:outputNewOrder" name="output"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="OrderServiceBinding" type="tns:OrderEndpoint"
    >

  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
```

```
    <wsdl:operation name="CreateOrder">

      <soap:operation
        soapAction="http://camelprototype.ws0910.dsi.fbi.h_da.de/
            PlaceNewOrder" />
      <wsdl:input name="input">

        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="output">

        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="OrderEndpointService">
    <wsdl:port binding="tns:OrderServiceBinding" name="OrderService
        ">
      <soap:address location="http://localhost:8080/camelprototype/
          webservices/order"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Dazu verwendet der Prototyp den Servletcontainer Jetty [6] und Apache CXF [4] als Webservice-Framework. Beide Komponenten stehen als Apache Camel Plugins zur Verfügung. Dieser Ansatz erlaubt es, die Webservice-Schnittstelle nach deren Definition in einer WSDL-Datei 4.1 ohne implementierende Java-Klasse bereitzustellen. Dazu sind die Konfigurationsdateien pom.xml 4.2, camel-config.xml 4.3 und web.xml 4.4 entsprechend anzupassen.

Listing 4.2: pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http:
    //www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>de.h_da.fbi.dsi.ws0910</groupId>
  <artifactId>camelprototype</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>H_DA/FBI/DSI Camel Prototype WS2009/2010</name>

  <properties>
    <cxf-version>2.2.5</cxf-version>
    <jetty-version>6.1.9</jetty-version>
    <camel-version>2.1.0</camel-version>
  </properties>
  <repositories>
  </repositories>

  <dependencies>
    <!-- camel -->
    <dependency>
      <groupId>org.apache.camel</groupId>
```

```
    <artifactId>camel-core</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-javaconfig</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-csv</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cxf</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jetty</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-velocity</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mail</artifactId>
    <version>${camel-version}</version>
  </dependency>
  <dependency>
    <groupId>xerces</groupId>
    <artifactId>xercesImpl</artifactId>
    <version>2.8.1</version>
  </dependency>

  <!-- HSQLDB embedded db for OrderSystem stub -->
  <dependency>
    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>1.8.0.10</version>
  </dependency>
  <!-- Apache Derby embedded db for OrderSystem stub -->
  <dependency>
    <groupId>org.apache.derby</groupId>
    <artifactId>derby</artifactId>
    <version>10.4.2.0</version>
  </dependency>

  <!-- cxf -->
  <dependency>
```

```
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-core</artifactId>
    <version>${cxf-version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-jaxws</artifactId>
    <version>${cxf-version}</version>
  </dependency>

  <!-- Special Jetty http transport, so we can run mvn jetty:run
      -->
  <dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-jetty</artifactId>
    <version>${cxf-version}</version>
  </dependency>

  <!-- logging -->
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>

  <!-- cxf web container for unit testing -->
  <!--<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-jetty</artifactId>
    <version>${cxf-version}</version>
    <scope>test</scope>
  </dependency>-->

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <!--<version>3.8.2</version>-->
    <version>4.7</version>
    <scope>test</scope>
  </dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>2.5.6</version>
</dependency>
  <!-- unit testing mail using mock -->
  <dependency>
    <groupId>org.jvnet.mock-javamail</groupId>
    <artifactId>mock-javamail</artifactId>
    <version>1.7</version>
    <scope>test</scope>
  </dependency>

  <!-- Explicitly depend on javax.mail-1.4.1, because 1.4 is
      corruct-->
  <dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
    <version>1.4.1</version>
```

```
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>${camel-version}</version>
    <scope>test</scope>
    <type>test-jar</type>
  </dependency>
</dependencies>

<build>
  <plugins>
    <!-- to compile with 1.5 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
    <!-- CXF wsdl2java generator, will plugin to the compile goal
         -->
    <plugin>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-codegen-plugin</artifactId>
      <version>${cxf-version}</version>
      <executions>
        <execution>
          <id>generate-sources</id>
          <phase>generate-sources</phase>
          <configuration>
            <sourceRoot>${basedir}/target/generated/src/main/java
                </sourceRoot>
            <wsdlOptions>
              <wsdlOption>
                <wsdl>${basedir}/src/main/resources/OrderService.
                    wsdl</wsdl>
              </wsdlOption>
            </wsdlOptions>
          </configuration>
          <goals>
            <goal>wsdl2java</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!-- so we can run mvn jetty:run -->
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <version>${jetty-version}</version>
    </plugin>
    <!-- allows the route to be ran via 'mvn camel:run' -->
    <plugin>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-maven-plugin</artifactId>
```

```
        <version>${camel-version}</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 4.3: camel-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.
           springframework.org/schema/beans/spring-beans-2.5.xsd
       http://camel.apache.org/schema/spring http://camel.apache.
           org/schema/spring/camel-spring.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/
      >
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

  <!-- create a camel context as to start Camel -->
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <package>de.h_da.fbi.dsi.ws0910.camelprototype</package>
  </camelContext>
  <bean id="ordersystem"
    class="de.h_da.fbi.dsi.ws0910.camelprototype.OrderSystem"
    destroy-method="shutdown"/>
  <bean id="ordersystemadaptor"
    class="de.h_da.fbi.dsi.ws0910.camelprototype.OrderSystemAdaptor
        ">
    <constructor-arg ref="ordersystem" />
  </bean>
</beans>
```

Listing 4.4: web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java
        .sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>DSI CamelPrototype Web Application</display-name>

  <!-- location of spring xml files -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
      <param-value>classpath:camel-config.xml</param-value>
  </context-param>

  <!-- the listener that kick-starts Spring -->
  <listener>
    <listener-class>org.springframework.web.context.
      ContextLoaderListener</listener-class>
  </listener>

  <!-- CXF servlet -->
```

```
  <servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</
        servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- all our webservices are mapped under this URI pattern -->
  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/webservices/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Die Generierung der Java-Klassen aus der WSDL-Datei 4.1 erfolgt innerhalb des Maven Build-Vorgangs. Das Starten von Jetty erfolgt im Zuge des Starts von Camel durch Maven. In der Spring-Konfigurationsdatei 4.3 werden zwei Beans definiert, die die Einbindung des Bestellsystems realisieren. Das Spring-Framework übernimmt dabei die Verwaltung der Lebenszyklen der Bean-Objekte.

4.2.3 *Zentrale Route*

Den zentralen Verarbeitungsablauf der Integrationslösung realisiert die Java-Klasse *OrderRoutes*. Sie erbt wie zuvor beschrieben von der Camel-Klasse *RouteBuilder* und überschreibt die Methode *configure*. Diese Methode enthält eine Abfolge von Routing- und Mediationsregeln entsprechend der Anforderungen.

Im ersten Schritt erfolgt die Definition der Endpunkte für den Webservice4.1 und das Verzeichnis für die CSV-Dateien. Sobald eine Bestellung an einem dieser beiden Endpunkt eintrifft, wird sie in eine entsprechende Nachricht umgewandelt und im Kopf der Nachricht der Ursprung der Bestellung (*cxf* bwz. *csvinput*) vermerkt. Danach erfolgt die Weiterleitung in den Kanal *new-orders-in*. Die Nachrichten in diesem Kanal werden vom OrderNormalizer 4.2.4 konsumiert. Dieser stellt die Nachrichten in einem einheitlichen Datenformat im Kanal *normalized-orders* bereit. Kanäle mit einheitlichen Nachrichten stellen das EIP *Datatype Channel* dar.

Listing 4.5: OrderRoutes.java

```
package de.h_da.fbi.dsi.ws0910.camelprototype;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.builder.RouteBuilder;

public class OrderRoutes extends RouteBuilder {
  //OrderSystemAdaptor orderSysAdaptor = new OrderSystemAdaptor();
  private String email_authuser = null;
  private String email_authpw = null;
  private String email_recipient = null;

  public OrderRoutes() {
    this("neworders@eisy.net.eu.org","crasher@smtp.eisy.net.eu.org"
        ,"crashm3");
  }
```

```
public OrderRoutes(String email_recipient,String email_authuser,
    String email_authpw) {
  super();
  this.email_recipient = email_recipient;
  this.email_authuser = email_authuser;
  this.email_authpw = email_authpw;
}

  public void configure() throws Exception {
      // endpoint to our CXF webservice
      String cxfEndpoint = "cxf://http://localhost:8080/
          camelprototype/webservices/order"
              + "?serviceClass=de.h_da.fbi.dsi.ws0910.
                  camelprototype.OrderEndpoint"
              + "&wsdlURL=OrderService.wsdl";

      from(cxfEndpoint)
        // set header field to keep track of requester
        .setHeader("from",constant("cxf"))
        .to("direct:new_orders_in");
      from("file://target/csvinput?recursive=true&delete=true")
      .setHeader("from",constant("csvinput"))
      .to("direct:new_orders_in");

      from("direct:normalized_orders")
        // pass the new order to the order management system
            gateway
        // function integration
        .processRef("ordersystemadaptor")
        // send the notification email
        .to("direct:sendordermail")
        // CBR: if the order was sent by a webservice request
        .choice()
          .when(header("from").isEqualTo("cxf"))
            // send a response back to webservice client
            // return OK as response
              .process(new Processor() {
            public void process(Exchange exchange) throws Exception
                {
              OutputNewOrder res = new OutputNewOrder();
              res.setStatus("0");
              res.setMessage((String) exchange.getIn().getHeader("
                  orderno"));
              exchange.getOut().setBody(res,OutputNewOrder.class);
            }
        })
      .otherwise().end();

  // generate order mail file
      from("direct:sendordermail")
          // set the file name
        .setHeader("CamelFileName", bean(FilenameGenerator.class,
            "getFilename"))
          // and create the mail body using velocity templating
          .to("velocity:OrderMailBody.vm")
          // and store the file
          .to("file://target/ordermails/");

      // from the file -> send email
```

```
        from("file://target/ordermails/?move=sent")
            // set the subject of the email
            .setHeader("subject", constant("New Order notification"
                ))
            // send the email
            .to("smtp://"+email_authuser+"?password="+email_authpw+
                "&to="+email_recipient);
    }
}
```

4.2.4 *OrderNormalizer*

Die Implementierung des EIP *Normalizer* zur Realisierung des Order-Normalizers erfolgt in einer separaten Java-Klasse, die ebenfalls von *RouteBuilder* erbt. Dadurch bleiben die Routen übersichtlich und lassen sich ggf. leichter wiederverwenden. Ebenso folgt die Auslagerung des OrderNormalizers in eigene Klasse dem in Diagramm 2 dargestellten Implementierungskonzept.

Listing 4.6: OrderNormalizer.java

```
package de.h_da.fbi.dsi.ws0910.camelprototype;

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.model.dataformat.CsvDataFormat;

public class OrderNormalizer extends RouteBuilder{
  @Override
  public void configure() throws Exception {
    from("direct:new_orders_in")
        .choice()
          .when(header("from").isEqualTo(constant("csvinput")))
          //csv order transformer
            // convert CSV-File to InputNewOrder that further
                processing expects
            .unmarshal(new CsvDataFormat())
            .bean(OrderTransformer.class, "transformFromCsv")
            // put the InputNewOrder into the channel
            .to("direct:normalized_orders")

          .when(header("from").isEqualTo(constant("cxf")))
          // webservice order transformer
              // we need to convert the CXF payload to
                  InputNewOrder that further processing expects
              .convertBodyTo(InputNewOrder.class)
              // put the InputNewOrder into the channel
              .to("direct:normalized_orders");
  }
}
```

Der OrderNormalizer konsumiert die Nachrichten in unterschiedlichen Formaten aus dem Kanal *new-orders-in*. Mithilfe eines Content Based Router, der das zuvor eingefügte Kopfzeilenfeld *from* auswertet. Im Falle von *cxf* erfolgt die Konvertierung ohne weiteren Aufwand durch Verwendung der methode *convertTo*.

Lautet das *from*-Feld *csvinput*, so wird die Nutzlast der Nachricht nach Konvertierung in das Camel-CSV-Format an die Methode *trans-*

formFromCsv einer Bean-Instanz der Klasse *OrderTransformer* übergeben. Dort stehen die Daten als zweidimensionale Liste zur Verfügung. Aus diesen Daten wird ein Objekt vom Typ *InputNewOrder* generiert und als Nutzlast der Nachricht gesetzt.

4.2.5 *Normalized-Orders*

Der Kanal *normalized-orders* enthält somit ausschließlich Nachrichten, deren Nutzlast im Format *InputNewOrder* vorliegt. Der Einfachheit halber benutzt der Prototyp dieses Format als normalisiertes Datenformat. Alle nachfolgenden Verarbeitungsschritte, die diesen Kanal verwenden, finden die Nutzlast der Nachrichten in einem festen Format vor.

4.2.6 *Funktionsintegration des Legacy-Systems*

Der hier gezeigte Prototyp repräsentiert ein fiktives Legacy-System, das den Bestellprozeß abwickelt, als einfache Java-Klasse *OrderSystem*, die keinerlei Bestandteile von Camel verwendet und somit sehr lose gekoppelt ist. Die Integration erfolgt mithilfe der Java-Klasse *OrderSystemAdaptor* 4.7, die einen Adapter nach dem EIP *Event-Driven Consumer* zwischen Camel und dem Legacy-System darstellt. Der Einfachheit halber verwendet das Legacy-System hier direkt das normalisierte Datenformat, es könnte aber im Adapter durch einen integrierten Transformer in ein beliebiges Format konvertiert werden.

Listing 4.7: OrderSystemAdaptor.java

```
package de.h_da.fbi.dsi.ws0910.camelprototype;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class OrderSystemAdaptor implements Processor{
  private OrderSystem ordersystem;
  public OrderSystemAdaptor(OrderSystem ordersystem) {
    super();
    this.ordersystem = ordersystem;
  }
  public void process(Exchange exchange) throws Exception {
    InputNewOrder newOrder = exchange.getIn().getBody(InputNewOrder
        .class);
    // here happens functional integration of a legacy system
    String orderNum =ordersystem.handleNewOrder(newOrder);
    exchange.getIn().setHeader("orderno", orderNum);
  }
}
```

Das EIP *Event-Driven Consumer* läßt sich mit Camel durch Implementieren des Interfaces Processor realisieren. An dieser Stelle findet eine Integration eines Legacy-Systems auf Funktionsebene statt, indem der *OrderSystemAdaptor* die Methode *handleNewOrder* der *OrderSystem*-Instanz mit dem *InputNewOrder* Objekt als Parameter aufruft. Diese Methode liefert die generierte Bestellnummer als String zurück.

Die Funktion des Bestellsystem beschränkt sich auf das Speichern der Bestellung in einer relationalen SQL-Datenbank mit dem Zeitstempel des Bestelleingangs und die Rückgabe der Bestellnummer. Diese speichert der *OrderSystemAdaptor* 4.7 in der Kopfzeile *orderno* der Nachricht.

Der Aufruf des Bestellsystems erfolgt dabei über eine Bean-Referenz durch die Camel-Methode *processRef*. Der Parameter dieser Methode ist ein String, der mit der id einer Bean aus der Spring-Konfigurationsdatei 4.3 übereinstimmen muss.

4.2.7 *Generierung und Speicherung der Benachrichtigungs-E-Mails*

Die vom Bestellsystem generierte Bestellnummer dient der eindeutigen Speicherung der Sicherungskopie der Benachrichtigungs-E-Mail nach dem Muster ordermail-<Bestellnummer>.txt. Das Templating-System Velocity [8], das Camel als Endpunkt im Plugin *camel-velocity* bereitstellt, befüllt die Vorlage *OrderMailBody.vm* 4.8 mit den Daten der jeweils zu verarbeitenden Bestellung.

Listing 4.8: OrderMailBody.vm

```
Customer $body.customerNo has placed a new order via $headers.from

Order-No.:      $headers.orderno
Cust.No.:       $body.customerNo

Art.-No.        Article                 Amount
=================================================
#foreach($orderPos in $body.orderpositions)
$orderPos.articleNo             $orderPos.article               $
    orderPos.amount
#end

This is an auto generated email. You cannot reply.
```

Der nachgelagerte Routing-Eintrag speichert die E-Mail-Dateien im Verzeichnis „target/ordermails" mittels einer Dateiendpunkt-Komponente. Eine Polling-Consumer-Komponte überwacht dieses Verzeichnis und übergibt dort gefundene E-Mail-Dateien an eine SMTP-Komponente. Diese übernimmt das Versenden der E-Mail an die konfigurierte Zieladresse. Schließlich erfolgt das Speichern der E-Mail-Datei im Verzeichnis „target/ordermails/sent".

4.2.8 *Rückgabe der Bestellnummer an den Webservice-Konsumenten*

Ein zusätzlicher Content Based Router nach dem *OrderSystemAdaptor* 4.7 leitet die Nachricht an einen weiteren Prozessor weiter, falls die Kopfzeile *from* den Text *cxf* enthält. Dieser ist als Inline-Prozessor implementiert und setzt als Nutzlast der Ausgangsnachricht des Nachrichtenaustauschs eine neue Objektinstanz der Klasse *OutputNewOrder* mit entsprechendem Erfolgsstatus und der generierten Bestellnummer. Dieses Objekt stellt die Antwort der Webservice-Schnittstelle 4.1 an den Webservice-Konsumenten dar.

4.3 TESTFALL

Zum Testen des Prototypen dient ein JUnit-Test, der in der Java-Klasse CreateOrderRoutesTest 4.9 implementiert ist.

Listing 4.9: CreateOrderRoutesTest.java

```
package de.h_da.fbi.dsi.ws0910.camelprototype;
```

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.jvnet.mock_javamail.Mailbox;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
    AbstractJUnit4SpringContextTests;

@ContextConfiguration(locations="/CreateOrderRoutesTest-context.xml
    ")
public class CreateOrderRoutesTest extends
    AbstractJUnit4SpringContextTests{
  private static final long WAIT = 2000;
    private static Mailbox inbox;
    private static String EMAILRECIPIENT = "neworders@mycompany.com
        ";
    // should be the same address as we have in our route
    private static String ADDRESS = "http://localhost:8080/
        camelprototype/webservices/order";

    protected static OrderEndpoint createCXFClient() {
        // we use CXF to create a client for us as its easier than
            JAXWS and works
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean()
            ;
        factory.setServiceClass(OrderEndpoint.class);
        factory.setAddress(ADDRESS);
        return (OrderEndpoint) factory.create();
    }
    protected InputNewOrder createWSTestData() {
        // create input parameter
        InputNewOrder input = new InputNewOrder();
        input.setCustomerNo("456");
        OrderPosition orderPos1 = new OrderPosition();
        orderPos1.setAmount(2);
        orderPos1.setArticle("Super Gadget");
        orderPos1.setArticleNo("7890");
        input.getOrderpositions().add(orderPos1);

        OrderPosition orderPos2 = new OrderPosition();
        orderPos2.setAmount(8);
        orderPos2.setArticle("Mega Gadget");
        orderPos2.setArticleNo("6543");
        input.getOrderpositions().add(orderPos2);

        OrderPosition orderPos3 = new OrderPosition();
        orderPos3.setAmount(3);
        orderPos3.setArticle("Great Nonsens");
        orderPos3.setArticleNo("N6543");
        input.getOrderpositions().add(orderPos3);
        return input;
```

```
}
private void createCsvTestData() throws IOException {
  BufferedWriter bw = new BufferedWriter(new FileWriter("target/
      csvinput/34524.csv"));
  bw.write("574,One Article,3\n");
  bw.write("575,Second Article,4\n");
  bw.write("852,Another Article,5");
  bw.close();
}
@Test
  public void testWebserviceNewOrder() throws Exception {
    try {
        // assert mailbox is empty before starting
        assertEquals("Should not have mails", 0, inbox.size());
        // create the webservice client
        OrderEndpoint client = createCXFClient();
        // create Test Order for webservice testclient
        InputNewOrder testOrder = createWSTestData();
        for (int count=1;count<=5;count++) {
          // send the request
          OutputNewOrder out = client.createOrder(testOrder);
          // assert we got a OK back
          assertEquals("0", out.getStatus());
          //assert we got an order number back
          assertFalse("Should have an order number", out.
              getMessage().isEmpty());
          System.out.println("Received orderno: "+out.getMessage
              ());
          // let some time pass to allow Camel to pickup the file
               and send it as an email
          Thread.sleep(WAIT);
          // assert mail box
          assertEquals("Should have got "+count+" mails", count,
              inbox.size());
        }
    } catch (Exception ex) {
        throw new Exception("Error executing Webservice Test", ex
            );
    }
  }

@Test
public void testCSVNewOrder() throws Exception {
      // assert mailbox is empty before starting
      assertEquals("Should not have mails", 0, inbox.size());
      // create csv test data
      createCsvTestData();
      // let some time pass to allow Camel to pickup the file and
           send it as an email
      Thread.sleep(WAIT);
      // assert mail box
      assertEquals("Should have got 1 mail", 1, inbox.size());
}
@Before
public void cleanInbox() {
  inbox.clear();
}
@BeforeClass
public static void setUp() throws Exception {
```

```
    inbox = Mailbox.get(EMAILRECIPIENT);
  }
  @AfterClass
  public static void tearDown() throws Exception {
      OrderSystem.getInstance().dumpOrders();
  }
}
```

Listing 4.10: CreateOrderRoutesTest-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.
            springframework.org/schema/beans/spring-beans-2.5.xsd
       http://camel.apache.org/schema/spring http://camel.apache.
            org/schema/spring/camel-spring.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/
      >
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

  <!-- create a camel context as to start Camel -->
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="orderroutes" />
    <routeBuilder ref="ordernormalizer" />
  </camelContext>
  <bean id="ordernormalizer" class="de.h_da.fbi.dsi.ws0910.
      camelprototype.OrderNormalizer"/>
    <bean id="ordersystem"
      class="de.h_da.fbi.dsi.ws0910.camelprototype.OrderSystem"
      destroy-method="shutdown"/>
  <bean id="ordersystemadaptor"
    class="de.h_da.fbi.dsi.ws0910.camelprototype.OrderSystemAdaptor
        ">
    <constructor-arg ref="ordersystem" />
  </bean>
  <bean id="orderroutes"
    class="de.h_da.fbi.dsi.ws0910.camelprototype.OrderRoutes">
    <constructor-arg index="0" type="java.lang.String" value="
        neworders@mycompany.com" />
    <constructor-arg index="1" type="java.lang.String" value="
        someone@localhost" />
    <constructor-arg index="2" type="java.lang.String" value="" />
  </bean>
</beans>
```

Diese Test-Klasse enthält zwei Tests, sowie die Definition der Testumgebung. Die Testumgebung beinhaltet den in der Datei 4.10 definierten Camel-Context und einen Mailbox-Mock, der die generierten E-Mails empfängt. Dabei wird die zentrale Route *OrderRoutes* durch Konstruktor-Parameter so konfiguriert, dass sie anstatt an eine reale E-Mail-Adresse an den Mailbox-Mock sendet. Der erste Test generiert fünf Bestellungen und überträgt sie nacheinander über die Webservice-Schnittstelle. Nach jedem Senden überprüft sie die Test-Mailbox-Mock,

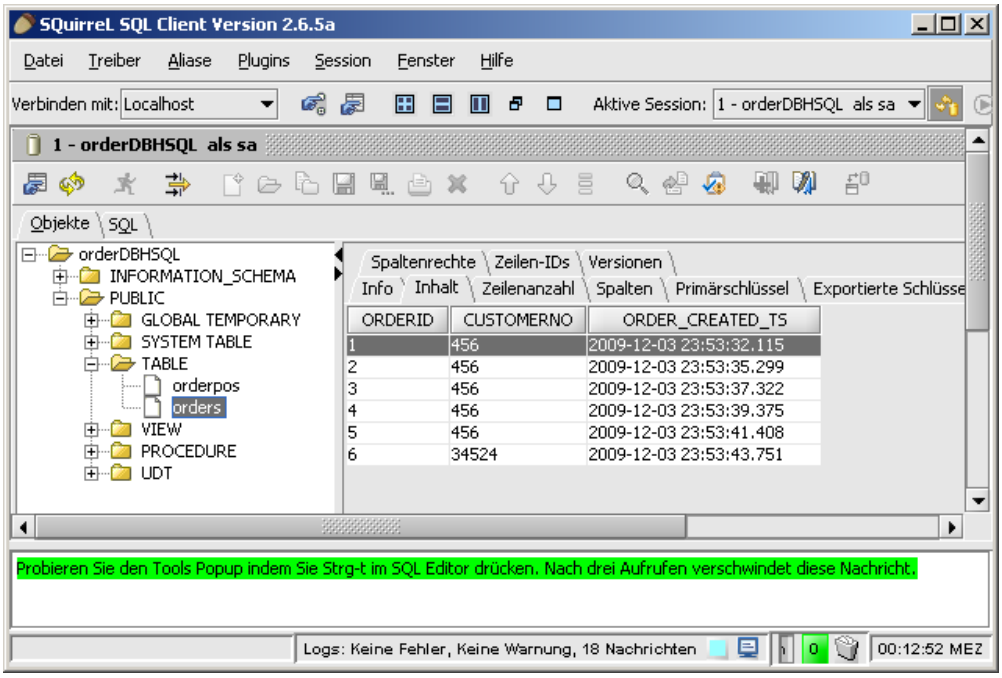

Abbildung 3: Inhalt der Tabelle *orders*

ob sie jeweils eine Nachricht mehr als im vorherigen Testdurchlauf enthält und ob die Antwort des Webservice eine Bestellnummer enthält.

Zum Test der CSV-Funktionalität generiert der Testfall eine CSV-Datei, legt sie im Zielverzeichnis ab und überprüft Mailbox-Mock auf eine neue Nachricht. Jede Testbestellung enthält drei Positionen.

Die *tearDown* Methode gibt den Inhalt der Bestelldatenbank zur Überprüfung auf der Konsole aus.

Der Start des Testfalls erfolgt durch das Kommando *mvn test* auf der Konsole im Wurzelverzeichnis des Projekts.

Mithilfe eines geeigneten Datenbank-Betrachters wie z. B. Squirrel [3] lassen sich die Einträge der Bestell-Datenbank ansehen. Die Abbildungen 3 und 4 zeigen die beiden Tabellen mit den Testdaten.

Die Abbildung 5 zeigt den Inhalt einer vom System generierten Benachrichtigungs-E-Mail aus dem Ordner „target/ordermails/sent".

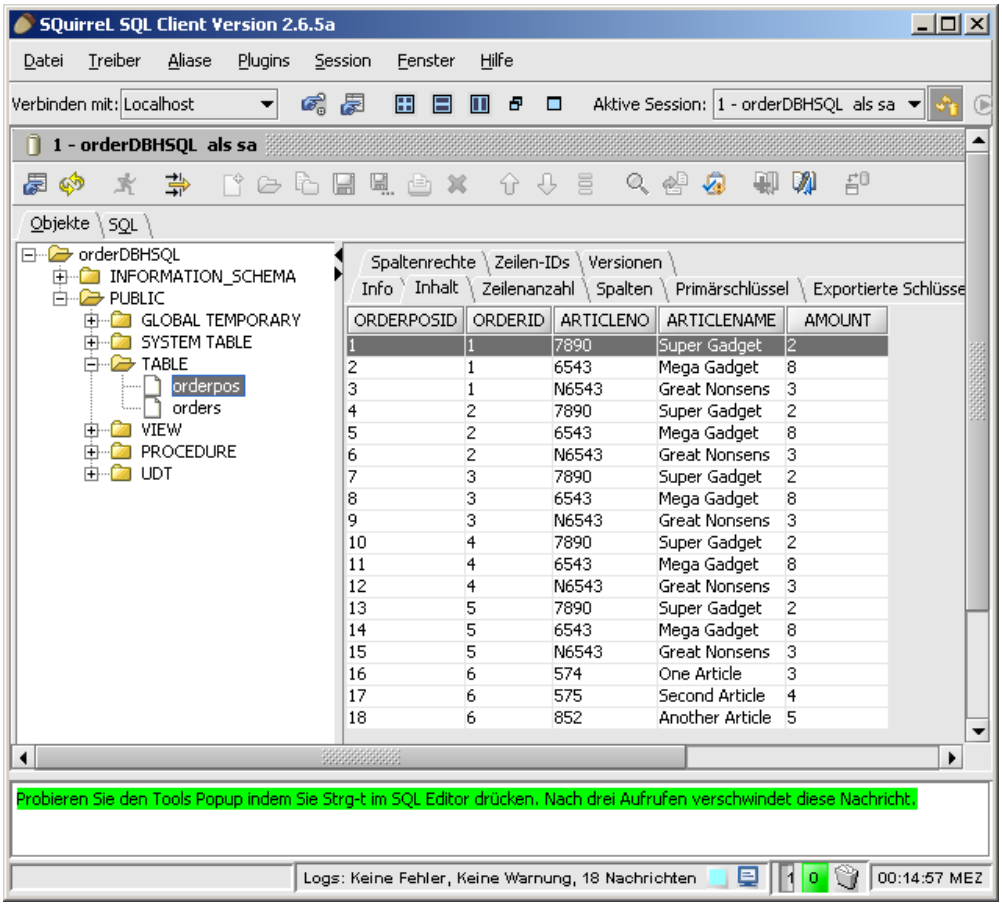

Abbildung 4: Inhalt der Tabelle *orderpos*

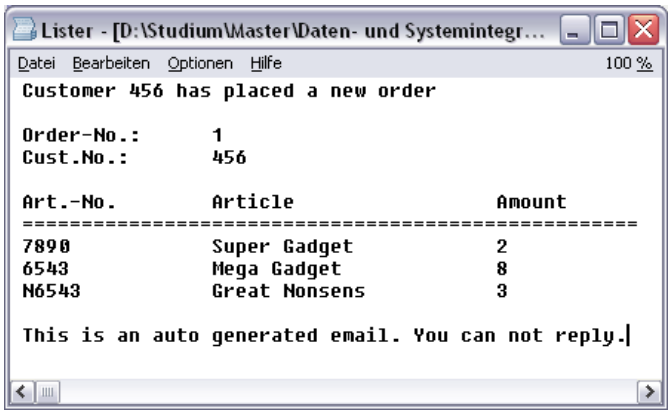

Abbildung 5: Sicherungsdatei einer Bestellbenachrichtigung

5 ZUSAMMENFASSUNG, FAZIT UND AUSBLICK

In der Praxis tauchen oftmals ähnliche Integrationsprobleme oft, weshalb sich auch deren Lösungen ähneln. Diese Ähnlichkeit führt zu bestimmten Entwurfsmustern, den Enterprise Integration Patterns (kurz EIPs). Diese Arbeit zeigt, wie sich Integrationsszenarien mittels EIPs darstellen lassen. Sie dienen hauptsächlich der grafischen Veranschaulichung mit einheitlichen Symbolen in Diagrammen.

Neben der grafischen Darstellung lassen sich anhand von EIP-Diagrammen Integrationslösungen auf Basis von Apache Camel realisieren.

Anhand eines fiktiven Geschäftsprozesses, der die Integration verschiedener Systeme auf Daten- und Funktionsebene erfordert, wird ein Integrationskonzept erarbeitet und als EIP-Diagramm dargestellt. Die Implementierung der dabei identifizierten EIPs auf Basis von Apache Camel orientiert sich stark an dem EIP-Konzept.

Der Prototyp gibt einen Einblick in die Funktionsweise und Konzept von Apache Camel, kann aber nur einen kleinen Teil der Mächtigkeit veranschaulichen. Der Prototyp stellt die Routing- und Mediationsregeln zur Erfüllung der Anforderungen des fiktiven Geschäftsprozesses dar. Weiterhin gibt er einen Einblick in den Lebenszyklus einer Integrationslösung auf Basis von Apache Camel. Die hier vorgestellten Schritte sind Entwurf, Realisierung und Test. Der Prototyp kann als Ausgangspunkt weiterer Betrachtungen dieses Lebenszyklus wie z. B. Deployment oder Betrieb dienen. Im Betrieb stellen besonders die Themen Business Activity Monitoring (BAM) und System Monitoring einer Apache Camel Integrationslösung eine interessante Fragestellung dar.

Weiterhin stellt sich die Frage, inwieweit Apache Camel als Integrationsserver betrachtet werden kann. Diese Arbeit stellt unter dieser Fragestellung die Aspekte Datenadapter, Middleware und Transformation vor. Bleibt die Frage offen, wie sich Prozeß-Management auf Basis von Apache Camel realisieren läßt.

Diese Arbeit zeigt eindeutig, dass Apache Camel als Plattform für Daten- und Funktionsintegration geeignet ist und einen wichtigen Baustein einer Service orientierten Architektur (SOA) darstellen kann. Neben dem Open-source Projekt Apache Camel zielt das kommerziell vertriebene Produkt FUSE Mediation Router [5] auf den Einsatz im Unternehmensumfeld ab. Es basiert auf Apache Camel und der Hersteller bietet u. a. den für Unternehmen wichtigen garantierten Support.

LITERATURVERZEICHNIS

[1] Diverse. Apache camel website, . URL http://camel.apache.org. (Cited on pages 5 and 6.)

[2] Diverse. Maven project website, . URL http://maven.apache.org. (Cited on page 5.)

[3] Diverse. Squirrel website, . URL http://squirrel-sql.sourceforge.net/. (Cited on page 24.)

[4] Diverse. Apache cxf project website, . URL http://cxf.apache.org/. (Cited on page 11.)

[5] Diverse. Fuse mediation router, . URL http://fusesource.com/products/enterprise-camel/. (Cited on page 26.)

[6] Diverse. Jetty project website, . URL http://www.mortbay.org/jetty/. (Cited on page 11.)

[7] Diverse. Spring project website, . URL http://www.springsource.org/. (Cited on page 7.)

[8] Diverse. Apache velocity project website, . URL http://velocity.apache.org/. (Cited on page 20.)

[9] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley, Boston, MA, USA, 1st edition, 2004. URL http://www.enterpriseintegrationpatterns.com/. (Cited on page 3.)

[10] OMG. Uml for enterprise application integration, v1.0. URL http://www.omg.org/cgi-bin/doc?formal/2004-03-26. (Cited on page 3.)