Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2123

Verbesserung eines EAI Pattern Editors

Christian Strempfer

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Thorsten Scheibler
begonnen am:	12. November 2007
beendet am:	13. Mai 2008
CR-Klassifikation:	D.2.6, D.2.11, H.3.5

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Anwendungsprogramme von verschiedenen Anbietern arbeiten selten zusammen, deshalb müssen Daten über mehrere Anwendungssysteme konsistent gehalten werden. Das ist aufwändig und fehleranfällig. Techniken die dem versuchen entgegen zu wirken, werden unter dem Begriff Enterprise Application Integration (kurz: EAI) zusammengefasst.

Um die Software-Integration zu vereinfachen können Entwurfsmuster (engl.: Patterns) dokumentiert werden, mit deren Hilfe man leicht die wichtigen Komponenten einer Systemarchitektur beschreiben kann. Diese Entwurfsmuster sind kombinierbar, so dass ihre Implementierungen wiederverwendbar sind. Sie werden Enterprise Integration Patterns (oder auch: EAI Patterns) genannt.

Aus den Enterprise Integration Patterns kann ein Systemmodell erstellt werden, das in ausführbaren Programmcode übersetzt werden kann. Dies entspricht dem Prinzip der Modellgetriebenen Softwareentwicklung.

Diese Arbeit beschäftigt sich mit einem bereits vorhandenen EAI Pattern Editor. Mit ihm ist es möglich ein Systemmodell zu erstellen und dieses auch in verschiedene Sprachen zu transformieren. Der praktische Teil der Arbeit bestand in der Verbesserung der Benutzerfreundlichkeit des Editors, während in einem theoretischen Teil die Zusammensetzung von Messaging Systemen untersucht und beschrieben werden sollte.

1.2 Aufgabenstellung

In einer früheren Diplomarbeit von Bettina Druckenmüller an der Universität Stuttgart wurde der Eclipse-basierte EAI Pattern Editor entwickelt (siehe [Dru07]). Dieser Editor dient dazu, Messaging System mit Hilfe von Entwurfsmustern graphisch darzustellen. Es ist auch möglich Parameter für die Transformation in die Business Process Execution Language (kurz: BPEL) anzugeben und diese Transformation durchzuführen. Parallel

zu der vorliegenden Arbeit wurde an der Unterstützung der Apache Camel Plattform gearbeitet (siehe [Kolo8]).

Nun sollte die Benutzerfreundlichkeit verbessert werden. Unter Anderem sollte die Verknüpfung verschiedener EAI Patterns erleichtert werden (Kapitel 4). Zuvor musste in Abhängigkeit der ausgewählten EAI Patterns spezielle Verknüpfungen ausgewählt werden. Diese Auswahl, die vom Benutzer Wissen über das zugrundeliegende Modell verlangte, sollte zum größten Teil von dem Editor übernommen werden. Eine weitere Aufgabe bestand in der Einbindung von externen Web Service Description Language (kurz: WSDL) Beschreibungen. Diese sollten in den EAI Pattern Editor importiert werden können, um in den Dialogen eine leichtere Auswahl der gewünschten Operationen zu ermöglichen (Kapitel 5).

Da der EAI Editor selbst auf einem Unified Modeling Language Modell (kurz: UML Modell) basiert, spielten Änderungen an diesem Modell eine größere Rolle. Im Laufe der Arbeit wurde klar, dass dieses UML Modell nicht immer konform mit der Standardliteratur war. So dass die Betrachtung vom Aufbau der Messaging Systeme zu einer Hauptaufgabe wurde. Es mussten die Fragen geklärt werden, welche Entwurfsmuster zusammenarbeiten können und welche Muster sich gegenseitig ausschließen. Im theoretischen Teil dieser Arbeit wurde deshalb ein Modell für Messaging Systeme entwickelt (Kapitel 3).

Kapitel 2

Grundlagen

In diesem Kapitel werden die theoretischen und die technischen Voraussetzungen für die vorliegende Arbeit erläutert.

2.1 Enterprise Application Integration

Enterprise Application Integration bezeichnet die Integration großer Softwaresysteme, in denen viele separate Anwendungen zur Zusammenarbeit bewegt werden sollen, die dies nicht von Haus aus unterstützen. Ziel ist es ein effektiveres und effizienteres Gesamtsystem zu erhalten, dass von dem Benutzer bedient werden kann als handele es sich um eine einzige große Anwendung.

Zu den Techniken der Enterprise Application Integration gehören File Transfer, Shared Database, Remote Procedure Invocation und Messaging, von denen jede ihre Vor- und Nachteile hat. Das wesentliche Entscheidungskriterium für die Integration ist allerdings die Stärke der Kopplung und diese nimmt in der genannten Reihe ab. Die Stärke der Kopplung ist ein Maß für die Menge der Annahmen, die die Systemkomponenten übereinander machen. Daraus lässt sich ableiten, dass lose gekoppelte Systeme flexibler sind, da sie weniger zu beachten haben. Diesem Vorteil steht allerdings oft ein komplexeres Anwendungsdesign gegenüber. So dass Entscheidungen von Fall zu Fall getroffen werden müssen (siehe [HW03]).

In dieser Studienarbeit wird Messaging als Mittel zur Integration betrachtet. Bei Messaging werden Daten zwischen den Anwendungen in Form von unabhängigen Nachrichten ausgetauscht. Die Kommunikation erfolgt also asynchron, was eine räumliche und zeitliche Entkopplung zur Folge hat. Die Verwaltung und Zustellung der Nachrichten übernimmt in der Regel eine Middleware, so dass sich die Entwickler kaum noch um die Infrastruktur kümmern müssen.

2.2 Enterprise Integration Patterns

Entwurfsmuster beschreiben bewährte Lösungsansätze für alltägliche Probleme in einer strukturierten Form. Die Muster erläutern den Kontext, in dem ein Problem auftauchen kann, und auch einen Lösungsansatz mit allen beteiligten Komponenten und ihrer Beziehung untereinander (siehe auch B96). Einige Muster-Sammlungen machen auch Querverweise, so kann man Schritt für Schritt zu einer idealen Lösung gelangen.

Im Rahmen der Software-Integration werden diese Entwurfsmuster Enterprise Integration Patterns genannt. Bei Messaging Systemen bestehen die Hauptaufgaben in der Transformation von Nachrichten und in deren Weiterleitung. Dementsprechend gibt es für diese Aufgaben eine umfangreich Sammlung von Entwurfsmustern. Daneben gibt es noch weitere Muster für die Nachrichtenstruktur, Message Channels, System Management und für Messaging Endpoints (siehe Kapitel 3).

Gregor Hohpe hat ein populäres Buch über Enterprise Integration Patterns geschrieben, das auch als Basis für diese Arbeit dient. [HW03]

2.3 XML Schema

XML Schema dient zur abstrakten Beschreibung eines gemeinsamen Vokabulars. Mit den Schemas können Regeln für die Anordnung und Häufigkeit von Datenelementen in einem Dokument festgelegt werden. Dadurch wird die Kommunikation zwischen unabhängigen Partnern leichter, weil bekannt ist, welche Art von Nachricht möglich ist.

Die XML Schema Spezifikationen können losgelöst voneinander eingesetzt werden. Ein Teil bietet eine schnelle Einführung in das Thema, während die anderen beiden Teile sehr technisch angelegt sind und sich mit der Beschreibung von Strukturen und Datentypen beschäftigen (siehe [Min02]).

2.4 Web Service Description Language

Die Web Service Description Language (kurz: WSDL) ist ein XML-Format, das vom World Wide Web Consortium entwickelt wurde und dient zur Beschreibung von Nachrichtenbasierten Netzwerkservices. Die Nachrichten können als Dokumente oder als Prozeduraufrufe strukturiert sein. Die WSDL ist auf eine sehr abstrakte Beschreibung der Services ausgelegt, kann aber für bestimmte Formate und Protokolle erweitert werden.

WSDL-Beschreibungen enthalten Informationen über die Daten und die Nachrichten die mit dem Service ausgetauscht werden können. Port-Types sind die Schnittstellen, die abstrakt den Ablauf der Kommunikation beschreiben. Bindings legen die konkreten Formate und Protokolle fest. Und Services sind die Endpunkte, die die beschriebene Funktionalität anbieten (siehe [CCMW01]).

2.5 Eclipse Modeling Framework und Graphical Editing Framework

Das Eclipse Modeling Framework (kurz: EMF) wurde zur Unterstützung der Modellgetriebenen Softwareentwicklung in der Eclipse Entwicklungsumgebung entworfen. Es wandelt ein Modell in lauffähigen Code um, der alle Objekte aus dem Modell implementiert und sogar ein standardmäßiges Serialisationsformat anbietet. Der Entwickler kann sich also ganz auf die Kernkomponenten der Software konzentrieren.

Das Graphical Editing Framework (kurz: GEF) bietet die Möglichkeit mit einem vorhandenen Modell zu arbeiten. Das heisst, es wird ein graphischer Editor erzeugt mit dem neue Instanzen des Modells erstellt werden können, vorhandene Instanzen manipuliert und gesichert werden können.

Die Kombination von EMF und GEF ermöglicht es sehr schnell Eclipse-basierte Editoren zu entwickeln. Eine sehr gute Einführung in die Entwicklung mit diesen Frameworks stellt IBM kostenlos zur Verfügung gestellt ([MDG+04]).

Kapitel 3

Messaging System Modell

In diesem Teil der Arbeit wird ein Modell zur Beschreibung von Messaging Systemen entwickelt. Dieses Modell basiert auf der Kombination von Enterprise Integration Patterns.

Der vorgegebene EAI Pattern Editor ([Dru07]) wurde aus einem UML Modell erzeugt und es würde sich anbieten dieses Modell wiederzuverwenden, allerdings entspricht es nicht allen Anforderungen die hier gestellt werden. Das UML Modell ist unvollständig, das heißt es enthält nicht alle gängigen Entwurfsmuster. Es wurde mit Bezug auf die BPEL Transformation erstellt, weshalb Teile des Modells nicht mit den abstrakteren Mustern übereinstimmen. Außerdem sind die Kombinationsmöglichkeiten der einzelnen Entwurfsmuster nur unzureichend abgebildet.

Die folgende Definition des Modells für Messaging Systeme wurde mit XML Schema formalisiert (Kapitel 3.1). Demnach werden die Beschreibungen der Messaging Systeme in einem Extensible Markup Language (kurz: XML) Format gespeichert. Die Beschreibungen können anhand des XML Schemas leicht validiert werden.

Bei der Modellierung der Messaging Systeme gibt es ein paar Einschränkungen. Die Abfolge der Entwurfsmuster im System kann nur schwer mit XML dargestellt werden, weil Verzweigungen und Zyklen enthalten sein könnten. Eine weitere Folge daraus ist, dass die Kombination verschiedener Entwurfsmuster teilweise nur durch Kennungen möglich ist. Dadurch verschlechtert sich die Lesbarkeit der Beschreibungen. Zur Betrachtung und Erstellung von Systembeschreibungen sollte deshalb ein graphischer Editor verwendet werden, vorzugsweise der EAI Pattern Editor.

Eine weitere Anforderung an die Beschreibungen war, dass die Darstellungsinformationen von dem eigentlichen Modell getrennt werden. Deshalb werden Modelle bei dem EAI Pattern Editor in zwei Dateien getrennt. Die erste enthält alle Systeminformationen, währen die zweite Datei für die Darstellung zuständig ist (Kapitel 3.2).

Die vorgestellten Entwurfsmuster wurden zum größten Teil aus dem Buch von Gregor Hohpe und Bobby Wolf entnommen ([HW03]) und die englischen Begriffe beibehalten. Da diese Arbeit auf der Parametrisierung von EAI Patterns aufbaut (siehe [Dru07] und

[Yuao8]), beinhaltet das Modell der Messaging Systeme auch Parameter die speziell für die Verwendung von Web Services und die Transformation in BPEL benötigt werden.

3.1 Formalisierung

Hier folgt die formale Beschreibung der Messaging Systeme. Die Entwurfsmuster sind nach der Systemkomponente, die sie betreffen aufgegliedert. Filter wurden zusätzlich in Message Routing, Message Transformation und System Management unterschieden.

3.1.1 Messaging System

Die Beschreibung des Messaging Systems enthält selbst bereits einige Informationen für die Unterstützung von BPEL und WSDL. Unter Anderem können Namespaces und Correlation Sets angegeben werden. Außerdem kann hier entschieden werden, ob Nachrichten ein Verfallsdatum haben können (siehe Message Expiration, Kapitel 3.1.2).

Das Element „messageSystem“ umschließt alle weiteren Entwurfsmuster. Da die Abfolge der Filter und Messaging Channels mit Verzweigungen und Zyklen nicht direkt abgebildet werden kann, wurden keine Einschränkungen bei der Anordnung der Muster gemacht. Dadurch gibt es mehr Freiheiten bei der Erstellung der Systembeschreibungen (Listing 3.1).

3.1.2 Message Construction

Die Entwurfsmuster zur Message Construction beziehen sich zum größten Teil auf den Inhalt der Nachrichten. Da die Nachrichten selbst aber im Messaging System Modell nicht abgebildet werden, wird hier nur auf ein paar ausgewählte Muster eingegangen.

Request-Reply

Die Pipes-And-Filter Architektur (Kapitel 3.1.6) von Messaging Systemen lässt Nachrichten immer nur in eine bestimmte Richtung fließen, weil die Messaging Channels (Kapitel 3.1.5) unidirektional sind. Trotzdem kann es vorkommen, dass Nachrichten von einem Filter losgeschickt werden, auf die er eine Antwort erwartet. Deshalb werden Anfrage und Antwort mit dem Request-Reply Muster gekennzeichnet, um sie auch erkennen zu können, wenn die Antwort einen anderen Pfad als die Anfrage durchläuft.

Zur Modellierung dieses Sachverhaltes wurde die Definition aus dem EAI Pattern Editor übernommen. Dort können zwei Filter als Receive-Reply Partners gekennzeichnet werden. Im Modell müssen dazu die Kennungen der beiden Filter angegeben werden (Listing 3.2).

Listing 3.1 XML Schema für Messaging System

```
<xsd:complexType name="messageSystem">
	<xsd:sequence>
		<xsd:element name="namespaceBPEL" minOccurs="0" maxOccurs="unbounded">
			<xsd:complexType>
				<xsd:attribute name="prefix" type="xsd:string" use="required"/>
				<xsd:attribute name="name" type="xsd:string" use="required"/>
			</xsd:complexType>
		</xsd:element>
		<xsd:element name="namespaceWSDL" minOccurs="0" maxOccurs="unbounded">
			<xsd:complexType>
				<xsd:attribute name="prefix" type="xsd:string" use="required"/>
				<xsd:attribute name="name" type="xsd:string" use="required"/>
			</xsd:complexType>
		</xsd:element>
		<xsd:element name="correlationSet" minOccurs="0" maxOccurs="unbounded">
			<xsd:complexType>
				<xsd:attribute name="prefix" type="xsd:string"/>
				<xsd:attribute name="name" type="xsd:string"/>
				<xsd:attribute name="property" type="xsd:string"/>
			</xsd:complexType>
		</xsd:element>
		... hier wird Message Expiration eingesetzt ...
		<xsd:choice minOccurs="0" maxOccurs="unbounded">
			... hier werden die weiteren Entwurfsmuster eingesetzt ...
		</xsd:choice>
	</xsd:sequence>
	<xsd:attribute name="name" type="xsd:string" use="required"/>
	<xsd:attribute name="targetNamespaceBPEL" type="xsd:string" use="required"/>
	<xsd:attribute name="targetNamespaceWSDL" type="xsd:string" use="required"/>
</xsd:complexType>
```

Listing 3.2 XML Schema für Request-Reply

```
<xsd:complexType name="receiveReplyPartners">
	<xsd:attribute name="receiveFilter" type="xsd:IDREF" use="required"/>
	<xsd:attribute name="replyFilter" type="xsd:IDREF" use="required"/>
</xsd:complexType>
```

Dependency Channel

Der Dependency Channel beschreibt die zeitliche Abhängigkeit zweier Filter. Der erste Filter muss vor dem zweiten Filter ausgeführt werden. Dieses Muster wird angewendet, wenn die Filter auf parallelen Pfaden liegen. Denn dann kann aufgrund der asynchronen Kommunikation die Reihenfolge nicht ohne Weiteres garantiert werden.

Listing 3.3 XML Schema für Dependency Channel

```
<xsd:complexType name="dependencyChannel">
	<xsd:attribute name="inputFilter" type="xsd:IDREF" use="required"/>
	<xsd:attribute name="outputFilter" type="xsd:IDREF" use="required"/>
</xsd:complexType>
```

Message Expiration

Message Expiration stellt das Verfallsdatum einer Nachricht dar. Wenn eine bestimmte Zeitspanne oder ein Zeitpunkt überschritten wurde, darf die Nachricht nicht weiterverarbeitet werden. Das Messaging System ist dafür verantwortlich, dass sie aussortiert wird.

Im EAI Pattern Editor wurde Message Expiration falsch implementiert. Dort kann jedem Messaging Channel einzeln das Muster zugeordnet werden. Bei genauer Betrachtung müsste es allerdings von allen Channels umgesetzt werden. Eine Nachricht könnte gerade auf einem Messaging Channel liegen, der Message Expiration nicht unterstützt, wenn das Verfallsdatum überschritten wird. Der nachfolgende Filter würde die Nachricht dann trotzdem verarbeiten.

In dieser Arbeit wird Message Expiration als Teil des Messaging Systems modelliert (Listing 3.4). Weitere Angaben sind nicht nötig. Es soll nur darauf hingewiesen werden, dass das System die Nachrichten auf ein Verfallsdatum überprüfen muss.

Listing 3.4 XML Schema für Message Expiration

```
<xsd:element name="messageExpiration" minOccurs="0">
       <xsd:complexType>
              <xsd:attribute name="id" type="xsd:ID"/>
       </xsd:complexType>
</xsd:element>
```

3.1.3 Filter

Filter sind Datenverarbeitungsschritte. Sie erhalten Daten, die sie einlesen, verändern und weitersenden können. Filter können je nach Kontext Methodenaufrufe innerhalb eines Systems sein, oder komplette Anwendungen die über ein Netzwerk kommunizieren (siehe auch Pipes And Filters, Kapitel 3.1.6).

In Messaging Systemen stellen Filter Anwendungen dar, die Nachrichten an das Messaging System senden und von ihm empfangen. Es kann auch vorkommen, dass dieselbe Anwendung mehrmals als EAI Pattern in der Beschreibung eines Systems abgebildet wird.

Filter

Der „filter“-Typ stellt die Basis aller weiteren Filter dar und definiert deren grundlegenden Attribute. Jeder Filter besitzt einen Endpoint (Kapitel 3.1.4), der den Filter mit dem Messaging System verbindet. Und jeder Filter kann einen Invalid Message Channel (Kapitel 3.1.5) besitzen, an den Nachrichten weitergeleitet werden, die der Filter nicht verarbeiten kann (Listing 3.5).

Für die Darstellung im EAI Pattern Editor wird eine Kennung festgelegt, über die die Position im Schaubild zugeordnet werden kann (siehe Präsentation, Kapitel 3.2).

Listing 3.5 XML Schema für Filter

```
<xsd:complexType name="filter">
       <xsd:sequence>
              <xsd:element name="endpoint" type="msgsys:endpoint" minOccurs="0"/>
              ... hier wird der Invalid Message Channel eingesetzt ...
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:ID" use="required"/>
</xsd:complexType>
```

Connections

Die Verknüpfungen der Filter mit den Messaging Channels (Kapitel 3.1.5) werden durch Erweiterung des „filter"-Typs hinzugefügt. Die Filter müssen dann ein Element „connections" enthalten, das dann wiederum „inputChannel" und „outputChannel" als Elemente enthält. Die häufigsten Kombinationen von Ein- und Ausgängen wurden in Elementgruppen zusammengefasst. So kann für jeden Filter leicht die richtige Anzahl Ein- und Ausgänge hinzugefügt werden. Auf ein Listing wird hier verzichtet.

Anwendung

Als Anwendung werden hier Filter bezeichnet über die keine weiteren Informationen bekannt sind. Insbesondere ist nicht bekannt, ob sie über einen Web Service aufgerufen werden können oder ob ihre Semantik einem spezielleren Entwurfsmuster entspricht.

Deshalb besitzt das Muster „application" keine weiteren Attribute. Optional kann ein Name zur Identifikation angegeben werden, er hat allerdings keine technische Bedeutung. Außerdem kann jeweils eine eingehende und eine ausgehende Verbindunge zu einem Messaging Channel festgelegt werden (Listing 3.6).

Filter mit Web Service

Speziell für die Kommunikation mit Web Services wurde der Typ „filterWithWS" eingeführt, der sich von „filter" ableitet. Er stellt die Oberklasse für viele speziellere Filter dar. Für diese Arbeit sind die genauen Parameter aber unrelevant, Einzelheiten finden Sie in [Dru07].

Listing 3.6 XML Schema für Anwendung

```
<xsd:complexType name="application">
       <xsd:complexContent>
              <xsd:extension base="msgsys:filter">
                     <xsd:sequence>
<xsd:element name="connections">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="inputChannel" type="msgsys:connection"
                         minOccurs="0"/>
                     <xsd:element name="outputChannel" type="msgsys:connection"
                         minOccurs="0"/>
              </xsd:sequence>
       </xsd:complexType>
</xsd:element>
                     </xsd:sequence>
                     <xsd:attribute name="name" type="xsd:string" use="optional"/>
              </xsd:extension>
       </xsd:complexContent>
</xsd:complexType>
```

Externer Service

Ein Externer Service ist das einfachste parametrisierte Entwurfsmuster für Filter, weil keine Annahmen über seine Funktion gemacht werden. Oft stellen Externe Services den Anfang und das Ende einer Nachrichtenkette in einem Messaging System dar.

Der „externalService“-Typ leitet sich von „filterWithWS“ ab, denn dieser Typ enthält alle für den Web Service Aufruf wichtigen Parameter. Einem Externen Service kann jeweils ein eingehender und ein ausgehender Messaging Channel zugeordnet werden. Zusätzlich kann wie bei Anwendungen ein Namen angegeben werden (Listing 3.7).

3.1.4 Messaging Endpoint

Messaging Endpoints sind die Schnittstelle, die Filter (Kapitel 3.1.3) verwenden um mit dem Messaging System zu arbeiten. Entwurfsmuster dieser Art beziehen sich unter Anderem auf die Art oder Qualität der Nachrichtenzustellung oder auf die Lastenverteilung.

Ein Messaging Endpoint kann höchstens mit einem Messaging Channel verbunden werden (siehe [HW03], Kapitel 3: Message Endpoint). Für diese Arbeit hat das keine Bedeutung, weil eine Anwendung mehrfach abgebildet werden kann. Dadurch kann ihr unterschiedliche Messaging Endpoints zugeordnet werden.

Listing 3.7 XML Schema für Externen Service

```
<xsd:complexType name="externalService">
       <xsd:complexContent>
              <xsd:extension base="msgsys:filterWithWS">
                     <xsd:sequence>
<xsd:element name="connections">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="inputChannel" type="msgsys:connection"
                         minOccurs="0"/>
              <xsd:element name="outputChannel" type="msgsys:connection" minOccurs="0"/>
       </xsd:sequence>
</xsd:complexType>
                            </xsd:element>
                     </xsd:sequence>
                     <xsd:attribute name="name" type="xsd:string"/>
              </xsd:extension>
       </xsd:complexContent>
</xsd:complexType>
```

Message Endpoint

Das Entwurfsmuster Message Endpoint entspricht der grundlegenden Idee der Messaging Endpoints, es werden keine weiteren Aussagen über die Art der Kommunikation mit dem Messaging System gemacht.

Im Modell ist „endpoint“ das umschließende Element für alle weiteren Messaging Endpoint Entwurfsmuster (Listing 3.8).

Listing 3.8 XML Schema für Message Endpoint

```
<xsd:complexType name="endpoint">
       <xsd:sequence>
              ... hier werden die weiteren Entwurfsmuster eingefügt ...
       </xsd:sequence>
</xsd:complexType>
```

Messaging Gateway und Messaging Mapper

Messaging Gateway und Messaging Mapper kapseln die Messaging Schnittstelle, sodass bei einem Austausch des Messaging Systems nur der Messaging Endpoint geändert werden muss. Das Messaging Gateway macht die Methoden der Anwendung für das Messaging System sichtbar, während der Messaging Mapper direkt auf Datenobjekte zugreift und sie im Messaging System verfügbar macht.

Diese beiden Entwurfsmuster sind nicht kombinierbar, da sie sich in der Art des Zugriffes auf die Anwendung unterscheiden. Ein Endpoint kann ohne diese Muster definiert werden, weil auch Mischformen denkbar sind. Beide Entwurfsmuster können optional eine Kennung

erhalten, dies ist im Typ „endpointPattern“ definiert. Auf ein Listing des „endpointPattern“ wird hier verzichtet.

Listing 3.9 XML Schema für Messaging Gateway und Messaging Mapper

```
<xsd:choice minOccurs="0">
       <xsd:element name="messagingGateway" type="msgsys:endpointPattern"/>
       <xsd:element name="messageingMapper" type="msgsys:endpointPattern"/>
</xsd:choice>
```

Event-Driven Consumer und Polling Consumer

Ein Event-Driven Consumer beginnt mit der Verarbeitung von Nachrichten sofort wenn er sie bekommt. Ein Polling Consumer dagegen bearbeitet Nachrichten in seiner eigenen Geschwindigkeit, indem er sich die Nachrichten selbst abholt, wenn er bereit ist.

Diese beiden Entwurfsmuster sind nicht kombinierbar, da sie eingehende Nachrichten unterschiedlich behandeln. Wenn nicht bekannt ist, wie sich die Anwendung verhält, muss keines der beiden Muster gewählt werden.

Listing 3.10 XML Schema für Event-Driven Consumer und Polling Consumer

```
<xsd:choice minOccurs="0">
       <xsd:element name="eventDrivenConsumer" type="msgsys:endpointPattern"/>
       <xsd:element name="pollingConsumer" type="msgsys:endpointPattern"/>
</xsd:choice>
```

Service Activator

Der Service Activator wird eingesetzt, wenn ein Service sowohl synchron als auch asynchron aufgerufen werden kann. Bei synchroner Kommunikation wird direkt auf den Service zugegriffen. Das Messaging System unterstützt allerdings nur asynchrone Kommunikation, weshalb der Service Activator bei einer eingehenden Nachricht den Service synchron aufruft und die Antwort wieder in eine (asynchrone) Nachricht umwandelt.

Der Service Activator ist mit den Mustern Event-Driven Consumer und Polling Consumer kombinierbar, da sich diese dann darauf beziehen wie schnell der Service Activator die Nachrichten verarbeitet und nicht auf den Service selbst, der immer Event-driven arbeitet.

Listing 3.11 XML Schema für Service Activator

```
<xsd:element name="serviceActivator" type="msgsys:endpointPattern" minOccurs="0"/>
```

Competing Consumers und Message Dispatcher

Competing Consumers und Message Dispatcher beschreiben, wie mehrere Instanzen eines Filters eingesetzt werden können. Die Instanzen arbeiten parallel um einen schnelleren Datendurchsatz zu erreichen. Die Muster unterscheiden sich nur bei der Nachrichtenverteilung.

Filter stehen als Competing Consumers in Konkurrenz zueinander. Sie holen sich die Nachrichten vom Messaging Channel (Kapitel 3.1.5) sobald sie bereit sind diese zu verarbeiten. Ein Message Dispatcher dagegen verteilt die Nachrichten nach eigenem Ermessen auf die Instanzen, die sie dann verarbeiten müssen.

Diese beiden Entwurfsmuster sind nicht kombinierbar, da sie sich in der Art der Nachrichtenverteilung auf die Filter unterscheiden. Im Modell werden die Muster, unabhängig von der Anzahl der Instanzen, nur einmal abgebildet.

Listing 3.12 XML Schema für Competing Consumers und Message Dispatcher

```
<xsd:choice minOccurs="0">
       <xsd:element name="competingConsumers" type="msgsys:endpointPattern"/>
       <xsd:element name="messageDispatcher" type="msgsys:endpointPattern"/>
</xsd:choice>
```

Durable Subscriber

Subscriber sind die Empfänger von Nachrichten über einen Publish-Subscribe Channel. Um garantieren zu können, dass Nachrichten auch zugestellt werden, wenn der Empfänger zeitweise nicht erreichbar ist, macht man ihn zu einem Durable Subscriber. Die Nachrichten werden dann solange gespeichert, bis sie zugestellt werden konnten.

Das Entwurfsmuster Durable Subscriber kann nur in Verbindung mit einem Publish-Subscribe Channel verwendet werden. In XML Schema konnte diese Einschränkung nicht definiert werden.

Listing 3.13 XML Schema für Durable Subscriber

```
<xsd:element name="durableSubscriber" type="msgsys:endpointPattern" minOccurs="0"/>
```

Idempotent Receiver

Idempotent Receiver sollen sicherstellen, dass keine Nachrichten mehrfach zugestellt werden. Dazu müssen Nachrichten eindeutig indentifizierbar sein. Wird eine Nachricht mehrfach empfangen, werden die Duplikate verworfen.

Listing 3.14 XML Schema für Idempotent Receiver

```
<xsd:element name="idempotentReceiver" type="msgsys:endpointPattern" minOccurs="0"/>
```

Selective Consumer

Selective Consumer verarbeiten nicht alle Nachrichten, die auf dem Messaging Channel liegen, sondern wählen sie sich anhand bestimmter Parameter im Nachrichten-Header aus. In der Regel werden mehrere Selective Consumers eingesetzt, sodass alle Nachrichten von mindestens einem Filter abgeholt werden.

Dieses Muster wurde bereits parametrisiert (Listing 3.15). Es enthält einen List mit Attributnamen und -werten, die mit dem Nachrichten-Header verglichen werden. Bei Übereinstimmung werden sie verarbeitet, ansonsten bleiben sie auf dem Messaging Channel.

Listing 3.15 XML Schema für Selective Consumer

```
<xsd:element name="selectiveConsumer" minOccurs="0">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="attribute" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:sequence>
						<xsd:group ref="msgsys:attribute"/>
					</xsd:sequence>
				</xsd:complexType>
			</xsd:element>
		</xsd:sequence>
		<xsd:attribute name="id" type="xsd:ID" use="optional" />
	</xsd:complexType>
</xsd:element>
```

Transactional Client

Bei einem Transactional Client wird der Nachrichtenversand bzw. -empfang innerhalb einer Transaktion ausgeführt. Nachrichten werden erst an den Messaging Channel weitergegeben bzw. von ihm gelöscht, wenn die Transaktion erfolgreich beendet wurde.

Listing 3.16 XML Schema für Transactional Client

```
<xsd:element name="transactionalClient" type="msgsys:endpointPattern" minOccurs="0"/>
```

3.1.5 Messaging Channel

Messaging Channels stellen die Verbindung zwischen den einzelnen Filtern her. Die Filter senden und empfangen ihre Nachrichten von Messaging Channels. Die Channels selbst sind nur Vermittler, sie verarbeiten oder verändern Nachrichten nicht (siehe auch Pipes And Filters, Kapitel 3.1.6).

Message Channel

Das Entwurfsmuster Message Channel entspricht der grundlegenden Idee der Messaging Channels, es werden keine weiteren Aussagen über die Nachrichtenübermittlung gemacht.

Im Modell ist „messageChannel“ das umschließende Element für alle weiteren Entwurfsmuster im Bereich Messaging Channel. Ihm muss lediglich ein Kennung zugeordnet werden (Listing 3.17).

Im UML Modell des EAI Pattern Editor stimmen die Entwurfsmuster Channel, Point-To-Point Channel und Datatype Channel nicht mit der Literatur überein. Im UML Modell ist jeder Channel ein Datatype Channel und der Point-To-Point Channel wird als „Datatype Channel“ bezeichnet. Die Informationen zur Darstellung werden deshalb über die Kennung des Datatype Channels (entspricht der UML Klasse „Channel“) und nicht des Message Channels (entspricht der UML Klasse „Pipe“) geladen.

Listing 3.17 XML Schema für Message Channel

```
<xsd:complexType name="messageChannel">
	<xsd:sequence>
		... hier werden die weiteren Entwurfsmuster eingesetzt ...
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:ID" use="required"/>
</xsd:complexType>
```

Point-To-Point Channel und Publish-Subscribe Channel

Die Messaging Channels können grundsätzlich in zwei Klassen eingeteilt werden: Point-To-Point Channels und Publish-Subscribe Channels.

Point-To-Point Channels besitzen genau einen Ein- und einen Ausgang über die sie Nachrichten empfangen und versenden können. Publish-Subscribe Channels dagegen können Nachrichten an eine beliebige Anzahl von Empfängern weiterleiten. Die Empfänger eines Publish-Subscribe Channels werden nicht von ihm selbst bestimmt, stattdessen können sich Filter beliebig an- und abmelden.

Diese beiden Entwurfsmuster sind nicht kombinierbar, da sie sich in der Anzahl und der Auswahl der Nachrichtenempfänger unterscheiden. Aber jeder Message Channel muss eines der Muster umsetzen (Listing 3.18). Beide können optional eine Kennung erhalten, dies ist im Typ „channelTypePattern“ definiert. Auf ein Listing des „channelTypePattern“ wird hier verzichtet.

Der EAI Pattern Editor bietet den Publish-Subscribe Channel an, aber er wurde noch nicht vollständig implementiert.

Listing 3.18 XML Schema für Point-to-Point Channel und Publish-Subscribe Channel

```
<xsd:choice>
       <xsd:element name="pointToPointChannel" type="msgsys:channelTypePattern"/>
       <xsd:element name="publishSubscribeChannel" type="msgsys:channelTypePattern"/>
</xsd:choice>
```

Datatype Channel

Datatype Channels schränken die Nachrichten, die an einen Message Channel gesendet werden können, auf ein bestimmtes Datenformate ein. Der Vorteil besteht darin, dass die empfangenden Filter nur Nachrichten erhalten, die sich auch verstehen können. Der Datatype Channel kann sowohl bei einem Point-To-Point Channel als auch bei einem Publish-Subscribe Channel verwendet werden.

Der EAI Pattern Editor erzwingt den Einsatz für beide Channel Klassen. Hier ist es - wegen der besonderen Unterstützung von Web Services - möglich ein Prefix und einen Message Type anzugeben. Es kann auch eine Kennung festgelegt werden.

Listing 3.19 XML Schema für Datatype Channel

```
<xsd:element name="datatypeChannel" minOccurs="0">
       <xsd:complexType>
              <xsd:attribute name="id" type="xsd:ID" use="optional" />
              <xsd:attribute name="prefix" type="xsd:string"/>
              <xsd:attribute name="messageType" type="xsd:string"/>
       </xsd:complexType>
</xsd:element>
```

Invalid Message Channel

Der Invalid Message Channel ist ein Message Channel an den Nachrichten gesendet werden, die der Filter nicht verarbeiten konnte.

Der Invalid Message Channel kann nur als Teil eines Filters definiert werden, innerhalb des Message Channel Elements kann er nicht definiert werden (siehe auch Kapitel 3.1.3).

Für die Modellierung eines Invalid Message Channels gibt es zwei Möglichkeiten. Die erste Möglichkeit besteht darin einen Web Service aufzurufen, der die ungültigen Nachrichten speichert und eventuell weiterverarbeitet. Dann müssen die Parameter für den Web Service Aufruf angegeben werden. Außerdem kann man zusätzlich das Muster Guaranteed Delivery einsetzen. Die zweite Möglichkeit ist, dass ein normaler Message Channel wie ein Invalid Message Channel behandelt wird. Dazu verweist man einfach auf einen vorhandenen Message Channel durch die Angabe seiner Kennung.

Der EAI Pattern Editor beherrscht nur die Möglichkeit eines Web Service Aufrufs.

Listing 3.20 XML Schema für Invalid Message Channel

```
<xsd:choice>
	<xsd:element name="invalidMessageChannel" minOccurs="0">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element name="guaranteedDelivery"
				    type="msgsys:guaranteedDelivery" minOccurs="0"/>
			</xsd:sequence>
			<xsd:attribute name="id" type="xsd:ID" use="required"/>
			<xsd:attributeGroup ref="msgsys:externalCall"/>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="invalidMessageChannelRef" type="xsd:IDREF" minOccurs="0"/>
</xsd:choice>
```

Dead Letter Channel

Damit Nachrichten die nicht zugestellt werden können, nicht den Message Channel belegen, kann man einen Dead Letter Channel bestimmen. Auf den Dead Letter Channel werden Nachrichten geleitet, wenn eine bestimmte Anzahl an Zustellungsversuchen fehlgeschlagen oder eine bestimmte Zeitspanne abgelaufen ist.

Wie bei Invalid Message Channels gibt es auch hier wieder die zwei Möglichkeiten eines Web Service Aufrufes oder eines Verweises auf einen vorhandenen Message Channel. Genauso kann Guaranteed Delivery eingesetzt und eine Kennung bestimmt werden.

Der EAI Pattern Editor beherrscht nur die Möglichkeit eines Web Service Aufrufs.

Listing 3.21 XML Schema für Dead Letter Channel

```
<xsd:choice>
	<xsd:element name="deadLetterChannel" minOccurs="0">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element name="guaranteedDelivery"
				    type="msgsys:guaranteedDelivery" minOccurs="0"/>
			</xsd:sequence>
			<xsd:attribute name="id" type="xsd:ID" use="optional" />
			<xsd:attributeGroup ref="msgsys:externalCall"/>
		</xsd:complexType>
	</xsd:element>
	<xsd:element name="deadLetterChannelRef" type="xsd:IDREF" minOccurs="0"/>
</xsd:choice>
```

Guaranteed Delivery

Guaranteed Delivery soll sicherstellen, dass beim Transport keine Nachrichten verloren gehen. Sie werden erst gelöscht, wenn sie erfolgreich zugestellt wurden.

Dieses Entwurfsmuster sollte nur bei Point-To-Point-Channels eingesetzt werden. Für Publish-Subscribe-Channels sollte man einen Durable Subscriber (Kapitel 3.1.4) verwenden. Er ermöglicht es für jeden Filter separat festzulegen, ob die Zustellung garantiert werden soll. In XML Schema konnte diese Einschränkung nicht definiert werden.

Für die Sicherung der Nachricht ist ein Web Service zuständig, die entsprechenden Parameter müssen angegeben werden. Optional kann eine Kennung zugeordnet werden.

Listing 3.22 XML Schema für Guaranteed Delivery

```
<xsd:complexType name="guaranteedDelivery">
       <xsd:attribute name="id" type="xsd:ID" use="optional" />
       <xsd:attributeGroup ref="msgsys:externalCall"/>
</xsd:complexType>
```

Messaging Bridge

Die Messaging Bridge soll den Austausch von Nachrichten zwischen mehreren Messaging Systemen erlauben. Dabei ist es wichtig, nicht nur die Nachricht zu übersetzen, sondern auch die Anforderungen an den Nachrichtentransport zu übernehmen. Wenn ein Messaging System beispielsweise Message Expiration einsetzt, müssen die anderen Systeme ebenfalls damit umgehen können.

Dieses Entwurfsmuster wurde nicht modelliert. Über die Anforderungen an einen Austausch zwischen mehreren Messaging Systemen ist wenig bekannt. Die reine Datenübertragung könnte durch einen Web Service realisiert werden, mit dem EAI Pattern Editor würde man dann jeweils nur die Struktur innerhalb eines Messaging Systems abbilden.

Message Bus

Der Message Bus wird eingesetzt um Middleware so zu organisieren, dass Anwendungen leicht hinzugefügt und entfernt werden können. Er ist dann auch dafür verantwortlich, dass Nachrichten an die richtigen Empfänger geleitet werden.

Ein Ziel des EAI Pattern Editors ist es die Modellgetriebene Softwareentwicklung zu unterstützen. Der Message Bus vergröbert aber die Darstellung des Messaging Systems zu sehr, um daraus noch einen Nutzen ziehen zu können. Deshalb wurde der Message Bus nicht modelliert.

3.1.6 Message Routing

Message Routing Entwurfsmuster beschreiben die Verzweigungen und Zusammenführungen von Nachrichtenpfäden. Sie sind notwendig, weil die einzelnen Bearbeitungsschritte einer Nachricht nicht immer gleich sind. Das kann daran liegen, dass mehrere Filter

dieselben Funktionen für eine andere Datenbasis anbieten oder dass Nachrichten mit unterschiedlicher Absicht über dieselben Messaging Channels verschickt werden.

Pipes And Filters

Das Pipes-And-Filters Muster vereinfacht das Message Routing. Es besagt, dass sich Filter (Kapitel 3.1.3) und Pipes (siehe Messaging Channel, Kapitel 3.1.5) immer abwechseln müssen. Dadurch dass die Filter nicht direkt kommunizieren, wird die Kopplung zwischen ihnen gelöst. Die Verbindungen zwischen den Filtern können leicht geändert werden, wenn ein Filter ersetzt werden soll, muss nur der Nachrichtenpfad geändert werden. Die Filter selbst müssen nicht angepasst werden.

Die Filter und die Messaging Channels können im Gegensatz zum ursprünglichen Pipes-And-Filters Muster allerdings mehrere Ein- und Ausgänge besitzen. Diese Variante wird auch Tee-and-Join-Pipeline-System genannt (siehe [BMRS96]).

Im Modell für Messaging Systeme kann das Pipes-And-Filters Muster nur schwer eingebaut werden. Die Filter bestimmen jeweils die Messaging Channels mit denen sie verbunden werden, weil dies aber nur über eine allgemeine Kennung möglich ist, kann fälschlicherweise eine Filterkennung eingefügt werden (siehe auch 3.1.3).

Message Router

Das Entwurfsmuster Message Router entspricht der grundlegenden Idee des Message Routings, es werden keine weiteren Aussagen über die Kriterien für die Weiterleitung gemacht. Es kann immer eingesetzt werden, wenn die Weiterleitung komplett dem Message Router überlassen wird oder nicht bekannt ist, um welche Art es sich handelt.

Der Message Router wird von „filter“ (Kapitel 3.1.3) abgeleitet und hat keine weiteren Attribute, deshalb wird er durch den Typ „filter1ToMany“ beschrieben. Er legt lediglich fest, dass ein Eingang und beliebig viele Ausgänge bestimmt werden können. Auf ein Listing des „filter1ToMany“ wird hier verzichtet.

Dynamic Router

Der Dynamic Router leitet Nachrichten immer an denselben Messaging Channel weiter. Um welchen Messaging Channel es sich dabei handelt, kann aber dynamisch über den Control Bus (Kapitel 3.1.8) bestimmt werden.

Der Dynamic Router wird durch „filter1ToManyWithWS“ beschrieben. Die Einstellung des ausgehenden Messaging Channels wird über einen Web Service geregelt. Die Web Service Daten sind im Element „configInfoRetrieval“ enthalten. Auf Listings von „filter1ToMany“ und „configInfoRetrieval“ wird hier verzichtet, weil diese keine weitere Bedeutung für das Messaging System Modell haben.

Content-Based Router

Der Content-Based Router hat mehrere Ausgänge, an die er Nachrichten weiterleiten kann. Anhand von Header-Informationen in einer Nachricht entscheidet er sich für genau einen Ausgang.

Dieses Entwurfsmuster wurde bereits parametrisiert. Für die Auswahl des ausgehenden Message Channels wird eine Liste verwendet, die jedem Channel Attribute und Vergleichswerte zuordnet, die mit dem Nachrichten-Header verglichen werden. Sollte es keine Übereinstimmung geben, kann man die Nachricht optional an einen eindeutig festgelegten „Else Channel“ weitergeben.

Der „contentBasedRouter“-Typ leitet sich vom „filterWithWS“ ab (siehe Kapitel 3.1.3). Neben der Attributliste muss mit „elseEmpty“ explizit festgelegt werden, ob es einen Messaging Channel für nicht zuordenbare Nachrichten gibt.

Listing 3.23 XML Schema für Content-Based Router

```
<xsd:complexType name="contentBasedRouter">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
<xsd:group ref="msgsys:filter1ToManyConnections"/>
<xsd:element name="contentBasedList">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="selector" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:sequence>
						<xsd:group ref="msgsys:attributeForChannel"/>
					</xsd:sequence>
				</xsd:complexType>
			</xsd:element>
		</xsd:sequence>
	</xsd:complexType>
</xsd:element>
<xsd:element name="elseEmpty" type="xsd:boolean"/>
<xsd:element name="elseChannel" type="xsd:string"/>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Message Filter

Der Message Filter wertet wie der Content-Based Router die Nachrichten-Header aus, er entscheidet, ob eine Nachricht aussortiert oder weitergeleitet wird. Falls die Nachricht weitergeleitet wird, wird sie immer an denselben Messaging Channel gesendet.

Im Modell wird er durch den Typ „filter1To1WithWS“ beschrieben. Da durch ihn lediglich die verbundenen Messaging Channels festgelegt werden, wird hier auf ein Listing verzichtet.

Join Router

Der Join Router ist ein Filter, der mehrere Eingänge und einen Ausgang hat. Er verändert Nachrichten nicht, sondern leitet sie nur an einen gemeinsamen Messaging Channel weiter (siehe auch [Yua08]).

Der Join Router wird durch den Typ „filterManyTo1WithWS“ modelliert.

Recipient List

Die Recipient List leitet Nachrichten an mehrere Empfänger weiter. Im Gegensatz zu einem Publish-Subscribe Channel (Kapitel 3.1.5) bestimmt hier die Recipient List, wer die Nachrichten bekommt und nicht die Empfänger selbst. Im Modell ist auch die Dynamic Recipient List enthalten, die die Empfänger anhand von Header-Informationen in den Nachrichten auswählt.

Da die Recipient List für die Verwendung mit Web Services parametrisiert wurde, wird sie von „filterWithWS“ abgeleitet. Sie hat einen Eingang und mehrere Ausgänge. Falls eine Dynamic Recipient List gewünscht ist, kann eine „contentBasedList“ mit den Parametern für die Auswahl der Empfänger angegeben werden, dann wird nur ein Teil der Empfänger angesprochen.

Splitter

Der Splitter teilt Nachrichten in mehrere Bestandteile auf, die er an verschiedene Messaging Channels weiterleitet. Zur späteren Identifikation werden den neuen kleineren Nachrichten Correlation Identifiers mitgegeben.

Hier handelt es sich um einen 1-To-Many Router, weshalb der Splitter auch durch den Typ „filter1ToManyWithWS“ modelliert wird. Die Aufsplittung der Nachrichten wird von einem Web Service übernommen. Auf ein Listing wird hier verzichtet.

Aggregator

Der Aggregator empfängt mehrere Nachrichten und vereint sie zu einer Nachricht, die er weiterleitet. Dabei besteht die Möglichkeit, dass die Nachrichten zu einer neuen Nachricht verschmolzen werden oder dass nur die Nachricht mit den besten Werten weitergeleitet wird. Dazu ist es nötig, dass die Nachrichten einen Correlation Identifier enthalten.

Listing 3.24 XML Schema für Recipient List

```
<xsd:complexType name="recipientList">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
<xsd:group ref="msgsys:filter1ToManyConnections"/>
<xsd:element name="prefixForMessageType" type="xsd:string"/>
<xsd:element name="messageTypeOfReturnedData" type="xsd:string"/>
<xsd:element name="contentBasedList" minOccurs="0">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="selector" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:sequence>
						<xsd:group ref="msgsys:attributeForChannel"/>
					</xsd:sequence>
				</xsd:complexType>
			</xsd:element>
		</xsd:sequence>
	</xsd:complexType>
</xsd:element>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Der Aggregator wurde bereits parametrisiert. Er enthält verschiedene Parameter, die bestimmen wie die Nachrichten zusammengesetzt werden. Unter Anderem besteht die Möglichkeit ein Header-Attribut anzugeben, anhand dessen entschieden wird welche Nachricht weitergeleitet wird (Listing 3.25).

Listing 3.25 XML Schema für Aggregator

```
<xsd:complexType name="aggregator">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
				<xsd:group ref="msgsys:filterManyTo1Connections"/>
				<xsd:element name="channelnameForExternalEvent"
				    type="xsd:string"/>
				<xsd:element name="callbackOperation" type="xsd:string"/>
				<xsd:element name="aggregationStrategy" type="xsd:string"/>
				<xsd:element name="completenessCondition" type="xsd:string"/>
				<xsd:element name="duration" type="xsd:string"/>
				<xsd:element name="nameOfBestProperty" type="xsd:string"/>
				<xsd:element name="bestPropertyMin" type="xsd:boolean"/>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Resequencer

Der Resequencer wird eingesetzt um Nachrichtensequenzen wieder in die richtige Reihenfolge zu bringen. Da die Kommunikation im Messaging System asynchron abläuft und Nachrichten aus derselben Sequenz verschiedene Pfade benutzen können, kann es vorkommen, dass die Reihenfolge durcheinander gerät.

Hierbei handelt es sich um einen 1-To-1 Router, der Nachrichten sammelt und weiterleitet, sobald die Reihenfolge wiederhergestellt wurde. Er wird durch den Typ „filter1To1WithWS“ dargestellt, weil er leicht durch einen Web Service umgesetzt werden kann.

Composed Message Processor

Der Composed Message Processor beschreibt das Vorgehen Nachrichten zu zerteilen, um sie auf verschiedenen Pfaden durch das Messaging System zu leiten und anschließend wieder zusammenzuführen. Dies kann leicht durch die beiden Entwurfsmuster Splitter und Aggregator erreicht werden.

Da es sich hier um ein übergeordnetes Muster handelt, dass leicht durch andere Entwurfsmuster ersetzt werden kann, wurde es nicht in das Modell für Messaging Systeme eingebaut.

Scatter-Gather

Scatter-Gather beschreibt das Vorgehen Nachrichten an mehrere Filter mit derselben Funktion weiterzuleiten, um anschließend die beste oder schnellste Antwort weiterzuverarbeiten. Dies kann leicht durch die beiden Entwurfsmuster Recipient List und Aggregator erreicht werden. Alternativ zur Recipient List kann auch ein Publish-Subscribe Channel verwendet werden.

Da es sich hier um ein übergeordnetes Muster handelt, dass leicht durch andere Entwurfsmuster ersetzt werden kann, wurde es nicht in das Modell für Messaging Systeme eingebaut.

Routing Slip

Ein Routing Slip wird eingesetzt, wenn die Reihenfolge in der Verarbeitungsschritte auf eine Nachricht angewendet werden, von der Nachricht abhängig ist. Der Routing Slip ist dabei eine geordnete Liste mit Filteradressen, die der Nachricht angehängt wird und die bestimmt welcher Filter wann besucht werden muss.

Da der Inhalt von Nachrichten in diesem Modell nicht abgebildet wird, wurde dieses Muster nicht aufgenommen.

Process Manager

Der Process Manager übernimmt die Aufgabe des Routing Slips in Form eines zentralen Filters. Die Nachrichten werden vom Process Manager an den Filter weitergeleitet, der den Verarbeitungsschritt durchführt, der gerade benötigt wird. Der Vorteil gegenüber dem Routing Slip ist, dass nicht jeder Filter sondern nur der Process Manager mit der dynamischen Reihenfolge umgehen können muss.

Der Process Manager vergröbert die Darstellung des Messaging Systems zu sehr, um daraus noch einen Nutzen ziehen zu können. Deshalb wurde der Process Manager nicht modelliert.

Message Broker

Der Message Broker vereinfacht die Systemarchitektur, in dem er das komplette Message Routing übernimmt. Andere Filter sind nur noch mit ihm verbunden und er entscheidet welche Nachricht an welchen Filter weitergeleitet wird.

Der Message Broker vergröbert die Darstellung des Messaging Systems zu sehr, um daraus noch einen Nutzen ziehen zu können. Deshalb wurde der Message Broker nicht modelliert.

3.1.7 Message Transformation

Nachrichten in einem Messaging System können aus sehr verschiedenen Quellen stammen. Nicht jede Anwendung kann das Datenformat einer anderen Anwendung verstehen, deshalb wird Message Transformation eingesetzt um ein Datenformat in ein anderes zu übersetzen. Außerdem muss beachtet werden, dass es neben rein formalen auch semantische Unterschieden zwischen den Datenformaten geben kann.

Die Transformation Entwurfsmuster übernehmen keine Routing Funktionen, deshalb hat jeder Translator genau einen Eingang und einen Ausgang.

Message Translator

Message Translators werden eingesetzt um ein Datenformat in ein anderes zu übersetzen. Wie dies geschieht bleibt dem Translator überlassen.

Der EAI Pattern Editor verwendet ein Style Sheet zur Beschreibung der Übersetzung. Dazu muss eine URI angegeben werden, unter der das Style Sheet hinterlegt ist.

Listing 3.26 XML Schema für Message Translator

```
<xsd:complexType name="messageTranslator">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
				<xsd:group ref="msgsys:filter1To1Connections"/>
				<xsd:element name="stylesheetURI" type="xsd:string"/>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Envelope Wrapper

Der Envelope Wrapper packt Rohdaten aus einer Anwendung in ein Nachrichtenformat, mit der das Messaging System und die dazugehörigen Filter arbeiten können.

Der Envelope Wrapper wurde noch nicht parametrisiert und wird demnach durch den Typ „filter1To1“ beschrieben.

Content Enricher

Der Content Enricher fügt Nachrichten weitere Daten hinzu. Dies können sowohl statische Daten als auch dynamische Daten aus einer Datenbank sein. Der Content Enricher kann auch Header-Informationen als Parameter verwenden.

Im Modell werden Web Services eingesetzt, um die zusätzlichen Daten einzufügen.

Listing 3.27 XML Schema für Content Enricher

```
<xsd:complexType name="contentEnricher">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
				<xsd:group ref="msgsys:filter1To1Connections"/>
				<xsd:element name="webServiceReturnsData" type="xsd:boolean"/>
				<xsd:element name="stylesheetURI" type="xsd:string"/>
				<xsd:element name="prefixForMessageType" type="xsd:string"/>
				<xsd:element name="messageTypeOfReturnedData"
				    type="xsd:string"/>
				<xsd:element name="keyDeterminesData" type="xsd:boolean"/>
				<xsd:element name="key" type="xsd:string"/>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Content Filter

Mit dem Content Filter können unrelevante Informationen aus einer Nachricht entfernt werden. Es können auch hierarchische Strukturen in Nachrichten abgeflacht werden, das erleichtert die Auswertung der Inhalte. Durch mehrere Content Filter kann man ein Static Splitter simulieren.

Modelliert wird der Content Filter mit Hilfe des Typs „filter1To1WithWs“. Er ist auch im EAI Pattern Editor verfügbar.

Claim Check

Claim Check funktioniert wie ein Content Filter, er entfernt Informationen aus den Nachrichten. Allerdings speichert ein Claim Check diese Informationen in einer Datenbank, damit sie später durch einen Content Enricher wieder eingefügt werden können.

Dieses Entwurfsmuster wird durch den Typ „filter1To1WithWS“ modelliert.

Normalizer

Ein Normalizer nimmt im Gegensatz zu einem Message Translator Nachrichten in verschiedenen Datenformaten an. Diese Nachrichten übersetzt er dann alle in dasselbe Format.

Der Normalizer wurde noch nicht parametrisiert, deshalb wird er durch den Typ „filter1To1“ beschrieben.

3.1.8 System Management

Zusätzlich zu den Entwurfsmustern für Filter in den Bereichen Message Routing (Kapitel 3.1.6) und Message Transformation (Kapitel 3.1.7) gibt es Muster, die sich mit administrativen Aufgaben beschäftigen. Diese Muster werden hier vorgestellt.

Control Bus

Der Control Bus stellt eine zentrale Administration des Messaging Systems dar. Unter Anderem sollen dynamische Filter wie Dynamic Routers gesteuert werden können. Testläufe sollen durch ihn verwaltet werden und er kann auch Statistiken über das Gesamtsystem sammeln. Wie der Control Bus dabei vorgeht, ist aber nicht weiter bestimmt.

In den Modellen von Messaging Systemen dient er als optischer Hinweis auf eine mögliche Einflussnahme von außen. Die Richtung des Nachrichtenflusses spielt hier nur eine untergeordnete Rolle, deshalb wird im Modell nicht zwischen Ein- und Ausgang unterschieden (siehe Listing 3.28).

Listing 3.28 XML Schema für Control Bus

```
<xsd:complexType name="controlBus">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filter">
			<xsd:sequence>
<xsd:element name="connections">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="channel" type="msgsys:routingConnection"
			    maxOccurs="unbounded"/>
		</xsd:sequence>
	</xsd:complexType>
</xsd:element>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Detour

Ein Detour wird eingesetzt um in bestimmten Situationen zusätzliche Arbeitsschritte in den Nachrichtenpfad einzufügen. Beispielsweise könnten zusätzlich Statistiken erfasst werden, die nur im Testbetrieb relevant sind. Im Grunde handelt es sich um einen Dynamic Router mit zwei Ausgängen (siehe Listing 3.29).

Detour wird bereits im EAI Pattern Editor eingesetzt. Zur Konfiguration wird ein Web Service verwendet, dessen Daten in „configInfoRetrieval“ festgelegt werden.

Listing 3.29 XML Schema für Detour

```
<xsd:complexType name="detour">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
<xsd:element name="connections">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="inputChannel" type="msgsys:connection"/>
			<xsd:element name="outputChannel" type="msgsys:routingConnection"
			    minOccurs="2" maxOccurs="2"/>
		</xsd:sequence>
	</xsd:complexType>
</xsd:element>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Wire Tap

Ein Wire Tap ermöglicht es Nachrichten auf einem Nachrichtenpfad abzuhören. Dazu wird der Wire Tap in den Pfad eingefügt. Er kann dann über Nachrichten über den „Command Channel“ aktiviert und deaktiviert werden. Das ist insbesondere für die Analyse des Messaging Systems hilfreich. So könnten temporär Statistiken erstellt werden.

Wire Tap hat einen Ein- und zwei Ausgänge. Zusätzlich wird ein Messaging Channel für den Empfang von Konfigurationsnachrichten benötigt (siehe Listing 3.30).

Listing 3.30 XML Schema für Wire Tap

```
<xsd:complexType name="wireTap">
       <xsd:complexContent>
              <xsd:extension base="msgsys:filter">
                     <xsd:sequence>
<xsd:element name="connections">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="inputChannel" type="msgsys:connection"/>
                     <xsd:element name="outputChannel" type="msgsys:connection"/>
                     <xsd:element name="secondaryOutputChannel" type="msgsys:connection"/>
                     <xsd:element name="commandChannel" type="msgsys:connection"
                         minOccurs="0"/>
              </xsd:sequence>
       </xsd:complexType>
</xsd:element>
                     </xsd:sequence>
              </xsd:extension>
       </xsd:complexContent>
</xsd:complexType>
```

Test Data Generator

Der Test Data Generator erzeugt wie der Name sagt Testnachrichten. Diese werden dann in den Nachrichtenpfad eingeschleust. Die Auswertung des Tests erfolgt durch den Test Data Verifier. So kann das System auch im Produktionsbetrieb überprüft werden.

Dieses Entwurfsmuster wurde bereits parametrisiert (siehe [Yuao8]). Neben den verbunden Messaging Channels muss auch noch angegeben werden, ob der Test Data Generator eine Kopie der Testnachricht an den Test Data Verifier sendet.

Test Data Verifier

Der Test Data Verifier ist dafür zuständig die durch den Test Data Generator erzeugten Testnachrichten auszuwerten. Er muss überprüfen, ob die Nachricht das geforderte Resultat verursacht.

Listing 3.31 XML Schema für Test Data Generator

```
<xsd:complexType name="testDataGenerator">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
				<xsd:group ref="msgsys:filter1ToManyConnections"/>
				<xsd:element name="messageDuplication" type="xsd:boolean"/>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Auch dieses Muster wurde parametrisiert (siehe [Yuao8]). Die Verifikation der Nachrichten kann von einem Web Service durchgeführt werden, das wird durch den Parameter „webServiceForVerification“ bestimmt (Listing 3.1.8).

Listing 3.32 XML Schema für Test Data Verifier

```
<xsd:complexType name="testDataVerifier">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filterWithWS">
			<xsd:sequence>
<xsd:group ref="msgsys:filterManyTo1Connections"/>
<xsd:element name="channelForDuplicatedMessages" type="xsd:string"/>
<xsd:element name="messageDuplication" type="xsd:boolean"/>
<xsd:element name="portTypeForVerification" type="xsd:string"/>
<xsd:element name="prefixPortTypeForVerification" type="xsd:string"/>
<xsd:element name="testMessageOperation" type="xsd:string"/>
<xsd:element name="callbackOperation" type="xsd:string"/>
<xsd:element name="webServiceForVerification" type="xsd:boolean"/>
<xsd:element name="resultMessageOP" type="xsd:string"/>
<xsd:element name="verificationOP" type="xsd:string"/>
<xsd:element name="attributes">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="attribute" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:sequence>
						<xsd:group ref="msgsys:attribute"/>
					</xsd:sequence>
				</xsd:complexType>
			</xsd:element>
		</xsd:sequence>
	</xsd:complexType>
</xsd:element>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

Channel Purger

Der Channel Purger entfernt alle Nachrichten von einem Message Channel. Beispielsweise kann es im Testbetrieb vorkommen, dass Nachrichten auf den Messaging Channels liegenbleiben, die nun entfernt werden müssen. Da der Channel Purger nicht selbst erkennen kann, wann er aktiv werden muss, kann er über einen „Command Channel“ aktiviert werden (Listing 3.1.8). Es sind auch andere Möglichkeiten zur Aktivierung denkbar.

Listing 3.33 XML Schema für Channel Purger

```
<xsd:complexType name="channelPurger">
       <xsd:complexContent>
              <xsd:extension base="msgsys:filter">
                     <xsd:sequence>
<xsd:element name="connections">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="inputChannel" type="msgsys:routingConnection"
                         maxOccurs="unbounded"/>
                     <xsd:element name="outputChannel" type="msgsys:connection"
                         minOccurs="0"/>
                     <xsd:element name="commandChannel" type="msgsys:connection"
                         minOccurs="0"/>
              </xsd:sequence>
       </xsd:complexType>
</xsd:element>
                     </xsd:sequence>
              </xsd:extension>
       </xsd:complexContent>
</xsd:complexType>
```

Message History

Die Message History ist der Pfad den eine Nachricht durch das Messaging System nimmt, er wird im Nachrichten-Header gespeichert. Durch sie wird die Analyse der Nachrichtenpfade möglich.

Da Nachrichteninhalte nicht im Modell für Messaging System dargestellt werden, wurde die Message History nicht modelliert.

Message Store

Das Messaging System ist lose gekoppelt, um trotzdem Informationen über die im Umlauf befindlichen Nachrichten zentral abrufen zu können, wird der Message Store eingesetzt. Dabei handelt es sich um eine Datenbank, die von Filtern mit Informationen über die Nachrichten versorgt wird (Listing 3.34). Die Verbindung zum Message Store übernimmt jeweils der Messaging Endpoint und nicht der Filter selbst.

Listing 3.34 XML Schema für Message Store

```
<xsd:complexType name="messageStore">
	<xsd:sequence>
		<xsd:element name="partOf" maxOccurs="unbounded">
			<xsd:complexType>
				<xsd:attribute name="filterId" type="xsd:IDREF"/>
			</xsd:complexType>
		</xsd:element>
	</xsd:sequence>
	<xsd:attribute name="id" type="xsd:ID" use="required"/>
</xsd:complexType>
```

Smart Proxy

Smart Proxies erlauben es die Rücksendeadresse vor Filtern zu verbergen. Dazu wird eine Nachricht bevor sie zu ihrem Ziel gelangt durch den Smart Proxy geleitet. Dieser ersetzt dann die Rücksendeadresse durch seine eigene und speichert die Informationen ab, um die Nachricht bei der Rückkehr wieder an den richtigen Messaging Channel weiterzuleiten. Außerdem kann der Smart Proxy Statistiken an den Control Bus schicken (Listing 3.35).

Ein Grund für den Einsatz eines Smart Proxies kann es sein, dass man verhindern möchte, dass Testnachrichten anders behandelt werden als normale Nachrichten. Beispielsweise wenn man die Geschwindigkeit eines externen Anbieters messen möchte.

Listing 3.35 XML Schema für Smart Proxy

```
<xsd:complexType name="smartProxy">
	<xsd:complexContent>
		<xsd:extension base="msgsys:filter">
			<xsd:sequence>
<xsd:element name="connections">
	<xsd:complexType>
		<xsd:sequence>
			<xsd:element name="requestChannel" type="msgsys:connection"/>
			<xsd:element name="serviceRequestChannel" type="msgsys:connection"/>
			<xsd:element name="serviceReplyChannel" type="msgsys:connection"/>
			<xsd:element name="replyChannel" type="msgsys:routingConnection"
			    maxOccurs="unbounded"/>
			<xsd:element name="commandChannel" type="msgsys:connection"
			    minOccurs="0"/>
		</xsd:sequence>
	</xsd:complexType>
</xsd:element>
			</xsd:sequence>
		</xsd:extension>
	</xsd:complexContent>
</xsd:complexType>
```

3.2 Präsentation

Das vorgestellte formale Modell für Messaging Systeme kann durch Informationen für die Präsentation erweitert werden. Im Falle des EAI Pattern Editor sind das die Bildschirmpositionen der Entwurfsmuster und die Pfade zu importierten Dateien. Zum Speichern der Datei sollte die Endung „.eaip“ verwendet werden.

Der EAI Pattern Editor unterstützt bisher nur WSDL-Dateien. Sie werden eingelesen und die enthaltenen Message Types, Port Types und Operationen in den Dialogen als Auswahl bereitgestellt (siehe Kapitel 5). Die Bildschirmpositionen werden den Entwurfsmuster über ihre Kennung zugeordnet. Mit XML Schema kann nicht sichergestellt werden, dass für jedes Muster genau eine Position enthalten ist. Dies muss durch den Editor geregelt werden.

Listing 3.36 XML Schema für Messaging System Presentation

```
<xsd:element name="messageSystemPresentation" type="msgsysp:messageSystemPresentation"/>

<xsd:complexType name="messageSystemPresentation">
       <xsd:sequence>
              <xsd:element name="import" type="msgsysp:import" minOccurs="0"
                  maxOccurs="unbounded"/>
              <xsd:element name="pattern" type="msgsysp:pattern" minOccurs="0"
                  maxOccurs="unbounded"/>
       </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="import">
       <xsd:attribute name="type" type="xsd:string" use="required"/>
       <xsd:attribute name="location" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pattern">
       <xsd:attribute name="id" type="xsd:ID" use="required"/>
       <xsd:attribute name="x" type="xsd:int" use="required"/>
       <xsd:attribute name="y" type="xsd:int" use="required"/>
</xsd:complexType>
```

3.3 Implementierung

Um Dateien mit Hilfe des Eclipse Modeling Frameworks zu öffnen, muss zu einer Dateiendung eine ResourceFactory registriert werden. Die Registrierung findet im MessageSystemModelManager statt. Diese ResourceFactory wird dann zur Erzeugung von spezialisierten Resourcen verwendet. Für die Verwendung des in dieser Arbeit formalisierten Modells, wird die EAIResourceFactoryImpl für die Dateiendung „.eai“registriert. Die EAIResourceFactoryImpl erzeugt dann Objekte vom Typ EAIResourceImpl. EAIResourceFactoryImpl und EAIResourceImpl sind im Paket de.unistuttgart.iaas.eaiparam.editors.resource abgelegt.

Um eine eigene Resource Implementierung verwenden zu können, muss diese das Interface org.eclipse.emf.ecore.resource.Resource umsetzen. Das kann vereinfacht werden, indem man die eigene Klasse von ResourceImpl ableitet, sodass nur noch die Methoden doLoad() und doSave() ergänzt werden müssen. Für den Zugriff auf die Dateien, wird doLoad() ein InputStream und doSave() ein OutputStream übergeben.

Es werden Java-Bibliotheken eingesetzt, die den Umgang mit XML Dateien erleichtern. Der javax.xml.parsers.DocumentBuilder erzeugt ein document-Objekt, das das org.w3c.dom.Document Interface des World Wide Web Consortiums implementiert (siehe [HHW+00], Anhang D). Mit Hilfe dieses document-Objekts können leicht XML Elemente, Attribute und Texte erzeugt und abgefragt werden.

Auf das im Editor geöffnete Modell kann durch Resource.getContents() zugegriffen werden. Dabei handelt es sich um eine Liste, der die geladenen Inhalte durch ihre die add()-Methode hinzugefügt werden können.

EAIResourceImpl beinhaltet für fast jede Klasse des EMF-Modells eine Methode zum Laden und Speichern der Daten. Bei Unterschieden zwischen dem EMF-Modell und dem Modell aus dieser Arbeit, kann das Laden und Speichern aber auch durch andere oder zusätzliche Methoden übernommen werden. Beispielsweise werden die Web Service Informationen für Filter durch die zusätzlichen Methoden loadInterfaceToExternalWebService() und saveInterfaceToExternalWebService() behandelt.

Die Verknüpfungen zwischen den Entwurfsmustern sowie die Positionen der Muster müssen beim Laden zwischengespeichert werden, weil die Objektzeiger für das XML Format durch Kennungen ersetzt werden müssen.

Kapitel 4

Verknüpfungen zwischen EAI Patterns

In diesem Kapitel wird beschrieben wie die Verknüpfung verschiedener EAI Patterns erleichtert wurde.

4.1 Pipes-And-Filters Connection

Zuvor musste in Abhängigkeit der ausgewählten EAI Patterns spezielle Verknüpfungen ausgewählt werden. Um dem Benutzer die Wahl der Konnektoren abzunehmen, wurde ein neues Werkzeug in den EAI Pattern Editor eingebaut, das dort Pipes-And-Filters Connection genannt wird. Es erzeugt eine gerichtete Verbindung zwischen zwei EAI Patterns.

Datenmodell

Zunächst muss das zugrunde liegende EMF-Modell des EAI Pattern Editors erläutert werden. Es basiert auf dem Arichtekturmuster „Pipes And Filters“ (siehe Kapitel 3.1.6) und verwendet Konnektoren um Pipes mit Filtern graphisch zu verbinden.

Diese Konnektoren waren zu Beginn dieser Arbeit an einen Nachrichtentyp gebunden, wofür es keine entsprechenden Entwurfsmuster gab. Da Messaging Channels aber laut Literatur an einen Nachrichtentyp gebunden sein können, wurde vom Anwender gefordert, dass der eingehende und ausgehende Konnektor eines Messaging Channels an denselben Typ gebunden sein sollten. Diese zusätzliche Aufwand für den Benutzer wurde ihm durch eine Änderung am Modell abgenommen. Konform zur Literatur ist jetzt der Messaging Channel selbst typgebunden und nicht mehr dessen Konnektoren.

Im EMF Modell leiten sich aus der Oberklasse „Pipe“ verschiedene Messaging Channel Patterns ab. Unter anderem die Klasse „Channel“, die nun dem Datatype Channel (Kapitel

3.1.5) entspricht. Von „Channel“ werden dann die Klassen „PubSubChannel“ und „DatatypeChannel“ abgeleitet, wobei Letzteres eine Kombination aus Point-To-Point-Channel (Kapitel 3.1.5) und Datatype Channel ist. Andere Entwurfsmuster leiten sich direkt von „Pipe“ ab oder sind nur mit dieser verknüpft.

Das EMF-Modell leiten sich alle Filter Muster von der abstrakten Klasse „Filter“ ab. Die Filter werden in vier Gruppen unterteilt:

- Transformation: Message Translator, Content Filter, ...
- Routing 1 to 1: Splitter, Resequencer, ...
- Routing 1 to many: Contentbased Router, Detour, ...
- Routing many to 1: Aggregator, Test Data Verifier, ...

Um nun Pipes und Filter zu verbinden, benötigt man Konnektoren die sich in „Sender“ und „Recipient“ unterscheiden lassen. „Senders“ führen Nachrichten von Filtern zu Pipes, „Recipients“ führen die Nachrichten von Pipes zu Filtern. Zusätzlich gibt es die Klassen „Routing Sender“ und „Routing Recipient“, die verwendet werden, wenn ein Filter mehrere Ausgänge bzw. mehrere Eingänge besitzt.

Der Benutzer musste also wissen, in welche Kategorie der Filter einzuordnen ist. Aus der Benutzeroberfläche kann man nicht immer auf die richtige Kategorie schließen. Erschwerend kam hinzu, dass Deadletter Channels und Invalid Message Channels nicht mit Hilfe von Konnektoren einem Channel oder einem Filter zugewiesen wurde, sondern hier die „Part-of“-Verbindung gewählt werden musste.

Modelländerungen

Das Graphical Editing Framework unterscheidet zwischen Knoten und Kanten, was in diesem Fall EAI Patterns und Konnektoren entspricht. Werkzeuge für Kanten können der Palette leicht hinzugefügt werden, indem man einen „ConnectionCreationToolEntry“ erstellt. Bei der Erstellung des „ConnectionCreationToolEntry“ muss die Klasse des Konnektors übergeben werden, damit eindeutig bestimmt ist, welches Werkzeug verwendet wurde und wie der Konnektor gespeichert wird.

Die Modell-Klassen „Sender“ und „Recipient“ kamen nicht in Frage, da bei ihnen die Richtung zwischen Filter und Pipe festgelegt ist. Der Benutzer soll die Richtung aber selbst bestimmen können, indem er zuerst den Start und anschließend das Ziel auswählt. Deshalb wurde im EMF-Modell die Klasse „Connection“ eingeführt, die als Oberklasse von „Sender“ und „Recipient“ bestimmt wurde. Sie selbst hat keine Attribute.

Create Connection Command

Die Bearbeitung von Elementen in einem GEF Editor wird mit „Commands“ gesteuert. Diese Java-Klassen sind dafür zuständig die Befehle auf Durchführbarkeit zu überprüfen und auch durchzuführen. Außerdem können sie Funktionen zur Rückgängigmachung und Wiederherstellung von Befehlen bereitstellen.

Im Fall der Pipes-And-Filters Connection musste nur „CreateConnectionCommand“ erstellt werden, da die weitere Bearbeitung über „Commands“ des ausgewählten Konnektors gesteuert werden kann. „CreateConnectionCommand“ ist auch für die Speicherung der Konnektorinformationen zuständig, es kann jeweils einen Quell- und Ziel-Filter und ein Quell- und Ziel-Pipe aufnehmen. Insgesamt werden nur zwei der Daten benötigt, aufgrund der Zusammenstellung dieser Informationen kann man dann auf den benötigten Konnektor schließen.

Node Edit Policy

Ausgelöst werden die „Commands“ über sogenannte „Policies“. Sie bestimmen welche Befehle erlaubt sind und welche nicht ausgeführt werden können.

Für die Pipes-And-Filters Connection wurde die „NodeEditPolicy“ angepasst. Dort gibt es mehrere Funktionen, von denen eine für den Start einer Verbindung - mit entsprechender Darstellung auf der Benutzeroberfläche - und eine zweite für die Fertigstellung der Verbindung zuständig ist. Die dabei gesammelten Daten werden an das „CreateConnectionCommand“ weitergegeben und anschließend wird die Ausführung des Befehls durch das GEF ausgelöst.

Mit Hilfe der „NodeEditPolicy“ wurde es auch möglich teilweise die Funktionen des „Part-of“-Werkzeuges zu übernehmen. So können die Muster Message Expiration und Guaranteed Delivery mit Pipes verbunden werden. Die Verknüpfung der Pattern wird dabei nicht vom „CreateConnectionCommand“ übernommen, sondern einfach an das dafür zuständige „CreateLinkCommand“ weitergereicht.

4.2 Datatype Connection

Die Datatype Connection verbindet zwei Filter und setzt einen Point-To-Point-Channel (inklusive Datatype Channel) dazwischen. Der Benutzer spart sich so das vorherige Einfügen des Messaging Channels, wodurch die Modellierung beschleunigt wird. Insbesondere da der Publish-Subscribe Channel zu diesem Zeitpunkt noch nicht vollständig implementiert war und dem Benutzer somit gar keine andere Möglichkeit offenstand.

Modelländerungen

Bei der Datatype Connection handelt es sich nur um eine Zusammensetzung von vorhandenen Element, deshalb wurde das EMF-Modell nicht direkt verändert. Allerdings musste für den „ConnectionCreationToolEntry“ auch hier wieder eine Klasse angegeben werde (siehe Kapitel 4.1). Deshalb wurde nachträglich das Java-Interface „DatatypeConnection“ und die Java-Klasse „DatatypeConnectionImpl“ eingefügt, die wie die anderen Elemente des EMF-Model aufgebaut sind. Die Datatype Connection selbst hat keine Attribute.

Create Datatype Connection Command

Die Hauptarbeit steckt bei diesem Werkzeug in der Java-Klasse „CreateDatatypeConnectionCommand“. Neben der Verwaltung der Daten, werden bei der Datatype Connection auch die Funktionen zur Rückgängigmachung und Wiederherstellung von Befehlen bereitgestellt.

Bei einer Datatype Connection wird zuerst der Point-To-Point Channel (inklusive Datatype Channel) erstellt, dann wird dieser durch einen „Sender“ (bzw. „RoutingSender“) und einen „Recipient“ (bzw. „RoutingRecipient“) mit den Filtern verbunden. Eine Verwendung der dazugehörigen „Commands“ war nicht ohne Weiteres möglich, weshalb die Element direkt durch die „ModelCreationFactory“ erstellt wurden.

Node Edit Policy

Um die Befehle verwenden zu können, mussten diese auch in die „NodeEditPolicy“ eingebaut werden. Dies war durch die saubere Implementierung des „CreateDatatypeConnectionCommand“s ohne Probleme möglich.

Kapitel 5

Einbindung der Web Service Description Language

Für den leichteren Umgang mit der Web Service Description Language (kurz: WSDL) soll der EAI Pattern Editor WSDL-Dateien importieren können und die darin enthaltenen Informationen dem Benutzer bereitgestellt werden. Zu diesem Zweck wurden die Klassen „Import“ und „ImportList“ im Paket „de.unistuttgart.iaas.eaiparam.editors.resource“ erstellt.

Dateien importieren

Die Klasse „Import“ enthält den Typ und den Speicherort der importierten Datei. Bisher werden aber lediglich WSDL-Dateien unterstützt. „ImportList“ verwaltet eine Liste der importierten Dateien und stellt Message Types, Port Types und Operations aus den WSDL-Dateien bereit. Damit die geladenen Daten nicht verlorengehen, wird das Singleton Entwurfsmuster angewendet. Es kann also immer nur genau eine Instanz der „ImportList“ pro Messaging System erzeugt werden, diese kann über die statische Methode getInstance() abgerufen werden.

Um Dateien zu importieren, öffnet man den Configuration Dialog des Messaging Systems. Dort wird eine Tabelle „Import files“ und der dazugehörige Button „Import“ angezeigt. Durch einen Klick auf den Button kann das Dateisystem nach WSDL-Dateien durchsucht werden. Wenn beide Dialoge bestätigt wurden, wird die importierte Datei geladen und die Informationen können abgerufen werden. Die Verweise auf die WSDL-Dateien werden auch mit dem Messaging System abgespeichert und geladen (siehe Kapitel 3.2).

Zu beachten ist, dass dies nur für WSDL-Dateien innerhalb des Projektpfads problemlos funktioniert. Bei Dateien außerhalb des Projektpfades ist es nicht möglich, diese auch auf anderen Rechnern zu verwenden. Beispielsweise bei einem Team-Projekt bei dem mit Subversion gearbeitet wird.

Dialoge anpassen

Die Dialoge befinden sich im Paket „de.unistuttgart.iaas.eaiparam.editors.dialogs“. Beispielhaft werden hier die Änderungen am „ExternalServicePropertiesDialog“ erklärt.

Ein Externer Service (Kapitel 3.1.3) verlangt die Angabe eines Port Types und einer Operation. Um die Daten aus den WSDL-Dateien anzeigen zu können, müssen die Textfelder im Dialog durch „Combo“-Felder ersetzt werden. „Combo“-Felder sind Auswahllisten, die gleichzeitig die Eingabe von eigenen Werten erlauben. Die Auswahl der Operationen ist vom Port Type abhängig, deswegen müssen zwei Event Listener (SelectionAdapter und KeyListener) registriert werden, die bei Änderungen des Port Types die Methode „PorttypeEdited“ aufrufen. Sie sorgt dafür, dass die Auswahl der Operationen aktualisiert wird.

Damit die Auswahllisten von Beginn an benutzt werden können, werden sie in der Methode „postInitGUI“ gefüllt. Das hat außerdem den Vorteil, dass die Werte immer aktuell sind wenn der Dialog geöffnet wird.

Die Anpassung der anderen Dialoge erfolgt genauso. Die Einbindung der Message Types stellt auch kein Problem dar. Sie können ebenfalls über die „ImportList“ abgefragt werden.

Kapitel 6

Zusammenfassung und Ausblick

Ziel dieser Studienarbeit war es zunächst ein Modell für Messaging Systeme zu erarbeiten. Dabei wurde unter Anderem die Kombinierbarkeit der EAI Patterns und ihr Wert für die Veranschaulichung in einem Systemmodell betrachtet. Bei der Formalisierung wurden die in früheren Arbeiten (siehe [Dru07] und [Yua08]) ermittelten Parameter in eine strukturierte Form gebracht.

Im weiteren Verlauf der Arbeit konnten auch ein paar Mängel in der Implementierung des EAI Pattern Editors aufgedeckt werden. So wurde die fälschliche Datentyp-Bindung der Konnektoren auf die Messaging Channels übertragen.

Aber nicht alle Mängel wurden korrigiert:

1. Es müssen noch Fehler im UML Modell des EAI Pattern Editors behoben werden. Message Expiration muss neu implementiert werden (siehe Kapitel 3.1.2).
2. Unterschiede zwischen der Literatur und dem UML Modell sollten beseitigt werden, um zukünftige Weiterentwicklungen zu erleichtern. Die Datentyp-Bindung des Point-To-Point Channels und des Publish-Subscribe Channels sollte optional gemacht werden.
3. Die falschen Bezeichnungen der Channel Patterns sollten angepasst werden (siehe Message Channel, Kapitel 3.1.5).
4. Es sollte dokumentiert werden, welche EAI Patterns jeweils von den Transformationen in ausführbaren Code unterstützt werden. Eventuell fehlende Unterstützung muss eingebaut oder für den Modellierer kenntlich gemacht werden.

Ein weiteres Ziel war die Verbesserung der Benutzerfreundlichkeit des EAI Pattern Editors. Dazu wurden zwei neue Werkzeuge eingeführt, die dem Modellierer ein wenig Arbeit bei der Verknüpfung der einzelnen EAI Patterns ersparen (Kapitel 4). Und die Web Service Description Language wurde integriert, um eine schnelle und sichere Auswahl der gewünschten Web Service Funktionen zu ermöglichen. Trotzdem gibt es bei den Dialogen noch Verbesserungsbedarf, denn sie sind noch sehr technikspezifisch formuliert.

Verzeichnis der Listings

Literaturverzeichnis

[BMRS96] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter: *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996. – 456 S. – ISBN: 975284359X

[CCMW01] Christensen, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva ; World Wide Web Consortium (Hrsg.): *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium, März 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[Dru07] Druckenmüller, Bettina: *Parametrisierung von EAI Patterns*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Diplomarbeit, Februar 2007. http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2583&engl=0. – 137 S.

[HHW+00] Hors, Arnaud L. ; Hégaret, Philippe L. ; Wood, Lauren ; Nicol, Gavin ; Robie, Jonathan ; Champion, Mike ; Byrne, Steve ; World Wide Web Consortium (Hrsg.): *Document Object Model (DOM) Level 2 Core Specification*. World Wide Web Consortium, November 2000. http://www.w3.org/TR/DOM-Level-2-Core/java-binding.html

[HW03] Hohpe, Gregor ; Woolf, Bobby: *Enterprise Integration Patterns: Designing, Building, And Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. – 736 S. – ISBN: 0321200683

[Kolo8] Kolb, Pascal: *Realization of EAI Patterns with Apache Camel*. Studienarbeit: Universität Stuttgart, Institut für Architektur von Anwendungssystemen. Version: April 2008. http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2127&engl=0

[MDG+04] Moore, William ; Dean, David ; Gerber, Anna ; Wagenknecht, Gunnar ; Vanderheyden, Philippe: *Eclipse Development Using the Graphical Editing Framework And the Eclipse Modeling Framework*. IBM, 2004. – 250 S. http://www.redbooks.ibm.com/abstracts/sg246302.html. – ISBN: 0738453161

[Mino2] Mintert, Stefan: *XML & Co. Die W3C-Spezifikationen für Dokumenten- und Datenarchitektur*. Addison-Wesley Professional, 2002. – 760 S. http://www.w3.org/XML/Schema#dev. – ISBN: 3827318440

[Yuao8] Yuan, Xin: *Prototype for Executable EAI Patterns*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Diplomarbeit, Januar 2008. http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2634&engl=0. – 102 S.

Alle URLs wurden zuletzt am 08.05.2008 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Christian Strempfer)