

Universität Stuttgart

Diplomarbeit Nr. 2830

Dienstgüteeigenschaften für ausführbare EAI Patterns

Frank Schmid

| | |
|---------------------------|-------------------------------|
| Studiengang: | Softwaretechnik |
| Prüfer: | Prof. Dr. Frank Leymann |
| Betreuer: | Dipl.-Inf. Thorsten Scheibler |
| Beginn am: | 16. Oktober 2008 |
| Beendet am: | 16. April 2009 |
| CR-Klassifikation: | D.2.11, D.3, H.4.1 |

UNIVERSITÄT STUTTGART

INSTITUT FÜR ARCHITEKTUR VON ANWENDUNGSSYSTEMEN

Inhaltsverzeichnis

| | |
|--|----|
| 1 Einführung | 5 |
| 2 Grundlagen | 8 |
| 2.1 Enterprise Application Integration (EAI)..... | 8 |
| 2.2 Messaging..... | 10 |
| 2.3 Pipes and Filters Architektur | 12 |
| 2.4 Enterprise Integration Patterns (EIP) | 13 |
| 2.4.1 Der Control Bus..... | 16 |
| 2.4.2 Die Management Konsole | 17 |
| 2.5 Dienstgüteeigenschaften (Quality of Service)..... | 18 |
| 2.6 Parametrisierung der EAI Patterns..... | 20 |
| 3 Fehlerbehandlungskonzepte | 22 |
| 3.1 Quality of Service bei Web Services | 22 |
| 3.1.1 WS-ReliableMessaging | 23 |
| 3.1.2 WS-Security | 23 |
| 3.1.3 WS-Transaction | 24 |
| 3.2 WS-BPEL..... | 26 |
| 3.2.1 Scopes..... | 26 |
| 3.2.2 Fehlerbehandlung in WS-BPEL | 27 |
| 3.2.3 Compensation Handlers (CH) | 28 |
| 3.2.4 Fault Handlers (FH) | 30 |
| 3.3 Workflows und Transaktionen | 31 |
| 3.3.1 Atomic Spheres..... | 32 |
| 3.3.2 Compensation Spheres..... | 33 |
| 3.3.3 Phönix Verhalten | 34 |
| 4 Fehleranalyse und Strategien zur Fehlerbehandlung..... | 36 |
| 4.1 Fehlersituationen | 37 |
| 4.1.1 Unkritische Fehlersituationen | 38 |
| 4.1.2 Kritische Fehlersituationen..... | 39 |
| 4.2 Fehler beim Filter | 39 |
| 4.2.1 Fall 1..... | 39 |
| 4.2.2 Fall 2..... | 41 |
| 4.3 Fehler in der Message | 43 |

| | |
|---|----|
| 4.4 Fehler im Kanal | 44 |
| 4.5 Strategien zur Fehlerbehandlung | 45 |
| 4.5.1 Koordinierte Konversation (2-Phasen-Commit Protokoll) | 47 |
| 4.5.2 Kompensationsaktionen..... | 49 |
| 4.5.3 Die Compensation Sphere..... | 53 |
| 4.5.4 Retry und Retry-All | 53 |
| 4.5.5 Correction Loop (Repair & Retry) | 55 |
| 4.5.6 Fehler ignorieren (Write Off)..... | 57 |
| 4.5.7 Phönix Verhalten | 57 |
| 4.6 Fazit der Fehlerbehandlungsanalyse | 59 |
| 5 Änderungen am System | 60 |
| 5.1 Fehlerbehandlungsstrategien für EIP mit BPEL..... | 62 |
| 5.1.1 Koordinierte Konversation | 62 |
| 5.1.2 Kompensationsaktionen..... | 63 |
| 5.1.3 Die Compensation Sphere..... | 64 |
| 5.1.4 Retry und Retry-All | 65 |
| 5.1.4 Correction Loop | 65 |
| 5.1.6 Fehler ignorieren | 66 |
| 5.1.7 Phönix Verhalten | 66 |
| 5.2 Der Dead Letter Channel | 66 |
| 5.3 Der Invalid Message Channel | 67 |
| 5.4 Die Klasse Filter | 68 |
| 5.4.1 Der Fault Handler des Filters | 69 |
| 5.4.2 Das Catch Fault Pattern | 71 |
| 5.4.3 Der Compensation Handler des Filters..... | 72 |
| 5.5 Der External Service..... | 72 |
| 5.6 Fehlerbehandlung für Filtergruppen | 73 |
| 5.6.1 Das Global Sphere Pattern | 73 |
| 6 Implementierung und Szenario | 76 |
| 6.1 Implementierung des kompensierbaren External Service | 76 |
| 6.2 Änderungen am Modell..... | 80 |
| 6.2.1 Die Klasse Filter | 80 |
| 6.2.2 Die Klasse CatchFault..... | 82 |
| 6.2.3 Die Klasse GlobalSphere..... | 82 |
| 6.3 Der WSDL Writer | 83 |

| | |
|---|----|
| 6.4 Das Szenario | 84 |
| 6.4.1 Simulation des Szenarios | 85 |
| 6.4.2 Vorbereitungen für die Ausführung des Szenarios | 85 |
| 6.4.3 Ausführung des Szenarios | 88 |
| 6.5 Checkliste für Entwickler | 92 |
| 6.6 Ausblick..... | 93 |
| 7 Zusammenfassung..... | 94 |
| Abbildungsverzeichnis..... | 98 |
| Anhang A | 99 |

1 Einführung

Unternehmen besitzen typischerweise eine große Anzahl verschiedener Anwendungen, mit denen sie ihre Geschäftsprozesse realisieren. Dabei sind diese Geschäftsprozesse meist über verschiedene Applikationen auf unterschiedlichen Plattformen verteilt. Mit Hilfe von Enterprise Application Integration (EAI) können diese heterogenen Teilsysteme zu einem großen Anwendungssystem integriert werden. Integrierte Anwendungen sind unabhängige Programme, die sowohl alleine ausgeführt werden können, als auch mit anderen, unabhängigen Anwendungen in lose gekoppelter Weise kommunizieren und zusammenarbeiten, um den notwendigen Datenaustausch zwischen den betreffenden Komponenten zu realisieren, damit auch applikationsübergreifende Geschäftsprozesse unterstützt werden können.

Eine Möglichkeit, EAI durchzuführen, basiert auf asynchroner Kommunikation [3]. Hier werden die verschiedenen Applikationen durch eine Message-orientierte Middleware (MOM) miteinander verbunden. Diese Middleware stellt die nötigen Verbindungen und Methoden für den Nachrichtentransport, die Nachrichtentransformation und das Weiterleiten der Nachrichten bereit. Die einzelnen Applikationen sind hierbei von den anderen Applikationen entkoppelt, das heißt, sie müssen sich weder des Orts, noch der (oft proprietären) Datenrepräsentationen oder Datenstrukturen dieser Applikationen bewusst sein. Da oftmals viele Bearbeitungsschritte notwendig sind, um Format, Struktur, Datenrepräsentation oder Kontext und Inhalt der Nachrichten an den Empfänger anzupassen, bietet die Pipes and Filters (PaF) -Architektur [3] ein Konzept, um diese notwendigen Arbeitsschritte in lose gekoppelter Art und Weise flexibel anzuordnen.

Hierzu wird eine Prozesskette erstellt; aus nachrichtenverarbeitenden Teilen, den **Filtern**, und Kanälen, die diese Filter miteinander verbinden, den **Pipes**. Die Filter repräsentieren dabei die einzelnen Arbeitsschritte, die für die Integration der Applikationen erforderlich sind. Sie wechseln sich hierbei immer mit den Pipes ab und bilden einen Graphen, der die Prozesskette für die Integrationslösung darstellt. Dabei können die Filter unterschiedliche Aufgaben erfüllen, wie zum Beispiel die Transformation von Nachrichten in ein anderes Format oder das Routen von Nachrichten durch das Netzwerk. Ein großer Vorteil des PaF Konzepts ist hierbei, dass die einzelnen Filter, da zustandslos, beliebig angeordnet, neu strukturiert, umorganisiert oder ausgetauscht werden können, ohne das dabei die Logik der einzelnen Filter geändert werden muss.

Diese Integrationslösung kann mit Hilfe von EAI Patterns [3] und deren Kombinationen erstellt werden. EAI Patterns stellen hierbei Muster für immer wiederkehrende Probleme dar, die Expertenwissen und Best-Practices einfangen. Ein Pattern wirft hierbei ein spezielles Entwurfsproblem auf, diskutiert und erörtert die Randbedingungen des Problems und präsentiert eine elegante Lösung, die oft auf gesammelten Erfahrungen aus ähnlichen Situationen beruht. Die Patterns sind ein Mittel, um die Architektur eines Enterprise Application Systems auf einem höheren Abstraktionslevel festzuhalten und liefern eine standardisierte Ausdrucksform für Analysten und Systemarchitekten [3].

Wenn man die Struktur der EAI Patterns, die auf der PaF-Architektur basieren, [3] betrachtet, stellt man fest, dass sie sich mit der von Graphen vergleichen lässt. Die Filter repräsentieren hierbei die

Knoten und die Pipes die Kanten des Graphen. Eine solche, Graphen-orientierte, Notation wiederum ist vergleichbar mit Workflows. Hier werden die Knoten von Aktivitäten und die Kanten von *control links* repräsentiert [12]. Aufgrund dieser Übereinstimmungen kann eine PaF-Architektur als Workflow angesehen werden, bei dem der Kontroll- und Datenfluss identisch ist.

Trotz der oben erwähnten Übereinstimmungen haben diese beiden Konzepte auch grundlegende Unterschiede [16]. Der Hauptunterschied liegt hierbei in der Unterstützung von Instanzen. In Workflow Systemen startet jede hereinkommende Request-Nachricht eine neue Instanz des Geschäftsprozesses. Dadurch werden die Nachrichten mit den jeweiligen Instanzen dieser Geschäftsprozesse verbunden. Bei der PaF-Architektur gibt es keinen Instanz-Begriff. Ein Filter, der eine Nachricht hereinbekommt, verarbeitet diese, leitet sie weiter und ist sofort wieder bereit, weitere Nachrichten zu empfangen und zu verarbeiten. Die Nachrichten, die er verarbeitet sind also nicht an eine bestimmte Instanz gebunden. Gerade dieses Konzept ist jedoch besonders in Produktionsumgebungen äußerst wichtig, da sich hiermit Kontroll-, Auditing- und Fehlerbehandlungsmechanismen umsetzen lassen [5].

Da es sich bei WS-BPEL um eine Graphen-orientierte Sprache handelt, die Instanz-basiert arbeitet, liegt nun die Möglichkeit nahe, WS-BEL zur Ausführung einer auf EAI Patterns und PaF-Architektur basierenden Integrationsinfrastruktur zu verwenden [12]. WS-BPEL ermöglicht es, verschiedene Web Services oder Filter in einem Prozess zu verknüpfen und die ein- und ausgehenden Nachrichten weiterzuverarbeiten. Bildet man entsprechende EAI Patterns auf BPEL ab und verwendet man damit verbundene Web Service Technologien, so lässt sich hiermit die zwischen den Applikationen gespannte MOM nutzen, um verschiedene Applikationen zu integrieren und damit applikationsübergreifend Geschäftsprozesse umzusetzen.

In den dieser Diplomarbeit vorangegangenen Arbeiten ([1], [2]) wurden EAI Patterns verschiedener Arten analysiert und parametrisiert, damit sie eingesetzt werden können, um eine Integrationslösung zu erstellen. Hierzu wurde ein Eclipse-basiertes Werkzeug entwickelt, das die zuvor parametrisierten EAI Patterns grafisch darstellt und automatisch in BPEL und WSDL transformiert.

Bisher wurden jedoch wichtige Aspekte, wie die der Fehlerbehandlung, nicht weiter betrachtet. In einer Produktionsumgebung ist eine wohldefinierte Fehlerbehandlung jedoch von äußerster Wichtigkeit. Es müssen Mechanismen zur Verfügung stehen, mit denen ein System auch im Fehlerfall zu einem für alle Parteien akzeptablen Ergebnis kommen kann. Wichtig sind hierbei, außer der Fehlerbehandlung, die Zuverlässigkeit des Systems (Dienstgüte) und das transaktionale Verhalten.

Im zweiten Kapitel dieser Diplomarbeit werden zunächst die Grundlagen, die hierfür wichtig sind, aufgeführt. Es werden zuerst wichtige Begriffe wie EAI, Konzepte wie Messaging, PaF-Architektur und das der EAI Patterns abgegrenzt und näher betrachtet. Weiter wird hier die Notation der parametrisierten Patterns aufgezeigt und der Begriff Dienstgüteeigenschaften (*englisch: Quality of Service*) im Umfeld der EAI Patterns in einer PaF-Architektur definiert.

Da später eine Abbildung der EAI Patterns auf WS-BPEL erfolgt, wird im dritten Kapitel das Fehlerbehandlungs- und Transaktionskonzept von BPEL selbst herausgestellt. Da BPEL sowohl die Prozesse in Workflow Systemen beschreibt und zusammenstellt und auch selbst ein Web Service Standard ist und sich somit theoretisch auch mit anderen WS-Standards kombinieren lässt, werden im dritten Kapitel außerdem die fundamentalen Konzepte für Fehlerbehandlung, Dienstgüte und

transaktionales Verhalten aus dem Bereich der Web Services und der Workflow Management Systeme aufgezeigt.

Im vierten Kapitel findet eine eingehende Analyse der Fehlersituationen und Fehlerorte statt. Es werden Überlegungen angestrebt, in welchen Situationen es zu Fehlern kommen kann und wie diesen Fehlern entgegengewirkt werden kann. Anschließend werden verschiedene Fehlerbehandlungsstrategien zusammengetragen und erörtert, wie diese in einer PaF-Architektur umgesetzt werden können und welche Probleme sich hier stellen. Den Abschluss des vierten Kapitels bildet das Fazit dieser Fehlerbehandlungsanalyse.

Das Ziel dieser Diplomarbeit ist es zu untersuchen, inwieweit Dienstgüteeigenschaften bei der Parametrisierung und der anschließenden Generierung eingebunden werden können. Deshalb wird im fünften Kapitel das Tool „EAltoBPEL“ analysiert und aufgezeigt, welche Dienstgüteeigenschaften durch die Parametrisierung selbst umgesetzt werden können und welche Dienstgüteeigenschaften Änderungen an der Struktur des Tools bedingen. In diesem Kapitel werden auch die im vierten Kapitel herausgestellten Fehlerbehandlungsstrategien auf ihre Umsetzung mit WS-BPEL geprüft.

Im sechsten Kapitel werden die Änderungen des Systems „EAltoBPEL“, die implementiert wurden, beschrieben und dokumentiert. Mit dem erweiterten Editor wurde ein Szenario entwickelt, das einen Prozess mit einem kompensationsbasierten Transaktionskonzept realisiert. Eine praktische Ausführung dieses Szenario wird an dieser Stelle beschrieben und das Vorgehen und die Ergebnisse werden bildhaft dargestellt.

Den Abschluss dieser Diplomarbeit bilden ein Ausblick, indem eine Auflistung möglicher Arbeiten, die am System in Zukunft vorgenommen werden können erfolgt und eine kurze Zusammenfassung.

2 Grundlagen

In diesem Kapitel werden zunächst die fundamentalen Grundlagen, die für diese Diplomarbeit wichtig sind, näher betrachtet. Hierfür wird zunächst auf wichtige Begriffe und Konzepte eingegangen.

In **Abschnitt 2.1** wird zuerst der Begriff *Enterprise Application Integration* (EAI) erklärt und definiert. Wichtige Integrationstypen und –Stile werden genannt.

Das Konzept des *Messaging* wird in **Abschnitt 2.2** erläutert.

Eine genauere Beschreibung der *Pipes and Filters Architektur* findet sich im **Abschnitt 2.3** dieses Kapitels.

Das Konzept der *Enterprise Integration Patterns* (EIP) [3], das von Gregor Hohpe und Bobby Woolf zusammengetragen und ausgeführt wurde, wird im **Abschnitt 2.4** zusammengefasst und erläutert. Für die Umsetzung von System Management- und Verwaltungsaufgaben wurden hierfür der *Control Bus* und die *Management Konsole* eingeführt, diese werden in **Abschnitt 2.4.1** und **2.4.2** etwas genauer betrachtet werden.

Anschließend wird im **Abschnitt 2.5** der Begriff *Dienstgüteeigenschaften* (Quality of Service) definiert und festgestellt, welche dieser Eigenschaften durch die Enterprise Integration Pattern [3] bereits umsetzbar sind oder wie für die Gewährleistung dieser Eigenschaften vorgegangen werden muss.

In **Abschnitt 2.6** wird kurz die Notation der *Parametrisierung der EAI Patterns* anhand einer Abbildung erläutert.

Wichtige Konzepte, wie das der Fehlerbehandlung von Workflow Management Systemen, Web Services und WS-BPEL selbst, werden im nachfolgenden Kapitel untersucht.

2.1 Enterprise Application Integration (EAI)

Die Softwarelandschaft eines Unternehmens besteht typischerweise aus hunderten bis tausenden von verschiedenen Anwendungen, den Enterprise Applikationen (EA) [10]. Bei den EA handelt es sich meistens um eine Mischung aus selbst hergestellten, von verschiedenen Herstellern zugekauften und übernommenen oder ererbten, älteren Anwendungen (Legacy Systems). Sie alle arbeiten auf verschiedenen Schichten und in verschiedenen Umgebungen.

Der Grund für diese heterogene Softwarelandschaft ist, dass die Schaffung einer einzigen, riesigen Applikation, die alle Funktionen des Unternehmens reflektiert nahezu unmöglich ist [10]. Selbst sehr große Standardapplikationen wie SAP oder Oracle liefern nur Bruchteile der Funktionen, die ein typisches Unternehmen benötigt.

Ein Grund für den Erwerb der Applikationen von verschiedenen Herstellern ist, dass dies den Unternehmen erlaubt, jeweils den besten Anbieter der benötigten Funktionen auf dem Markt zu wählen. Diese Applikationen wurden jedoch nicht gebaut, um zusammenzuarbeiten. Eine Funktion,

die in einer Applikation ausgeführt wurde, impliziert jedoch oft die Ausführung von Funktionen in anderen Applikationen. Dies zu erreichen ist jedoch keine einfache Aufgabe. Hauptprobleme sind hier die Unterschiede in den Datenformaten, Datenrepräsentationen, Aufrufmechanismen und den Systemumgebungen, in denen die verschiedenen Applikationen laufen. Das hieraus resultierende Integrationsproblem ist in der Praxis unvermeidbar.

Durch die Integration der heterogenen EA wird eine vereinheitlichte Menge von Funktionen aus unabhängigen Anwendungen gewonnen. Das heißt, jede Anwendung kann auch immer noch für sich alleine betrieben werden. Ein wichtiger Aspekt ist dabei die lose Kopplung zwischen diesen EA. Sie erlaubt es den einzelnen EA, sich weiterzuentwickeln, ohne dabei die anderen EA zu beeinflussen. Für die Nutzer des Systems (Kunden, Mitarbeiter, Geschäftspartner) sind die Abgrenzungen zwischen den Applikationen dabei nicht sichtbar. Die Geschäftsfunktionen, die sie ausführen, sind dabei oft intern über verschiedenen Applikationen, manchmal sogar extern über verschiedene Unternehmen verteilt. Trotzdem erscheinen sie dem Nutzer des Systems dabei wie eine einzelne Transaktion.

Es gibt sechs verschiedene Integrationstypen, die auch in Kombinationen vorkommen ([10], [3]):

1. **Portale** – Portale aggregieren an den Nutzer angepasste Informationen von verschiedenen Quellen in einer einzelnen Benutzerschnittstelle. Dies macht den Zugang zu verschiedenen Systemen unnötig, der Nutzer muss sich nur im Portal anmelden.
2. **Datenreplikation (Data Replication)** - Viele Systeme müssen auf dieselben Daten zugreifen, haben dabei aber ihre eigenen Datenspeicher. Um die Konsistenz dieser Daten zu gewährleisten, müssen sie mittels Replikationsfunktionen repliziert werden.
3. **Geteilte Geschäftsfunktionen (Shared Business Functions)** - Viele Anwendungen müssen nicht nur auf dieselben Daten zugreifen, sie nutzen auch dieselben Funktionen. Diese redundanten Funktionen können als geteilte Geschäftsfunktionen herausgestellt werden, sie können dann von allen anderen Anwendungen im System genutzt werden.
4. **Serviceorientierung (Service-oriented Architecture - SOA)** - Geteilte Geschäftsfunktionen können in Services umgewandelt werden. Ein Service ist eine wohldefinierte Funktion, die Benutzern des Services universell zur Verfügung gestellt wird. Der Service stellt ein Interface zur Verfügung, das den Benutzern ermöglicht, mit dem Service zu kommunizieren (*negotiation*). Weiter wird ein zentrales Serviceregister (*service discovery*) unterhalten, in dem sich jeder Service registriert. Negotiation und service discovery sind die Schlüsselemente, die SOA ausmachen [3].
5. **Verteilte Geschäftsprozesse** - Eine einzelne Geschäftstransaktion umspannt oft viele verschiedene Systeme. Die für die Geschäftstransaktion relevanten Funktionen sind in den meisten Fällen in Applikationen eingebunden. Was fehlt, ist die Koordination zwischen diesen Applikationen. Hier kann eine Geschäftsprozess-Management Komponente (z.B. Workflow Management System, WfMS [5]) eingebunden werden, die die Ausführung dieser Funktionen über die verschiedenen Systeme überwacht [10].
6. **Business-to-Business Integration (B2B Integration)** - Integration ist nicht nur innerhalb eines Unternehmens wichtig. Sie taucht auch zwischen Geschäftspartnern auf. Diese stellen ihre Funktionen auch außerhalb ihres Unternehmens zur Verfügung. Standardisierte Datenformate und Dienstgüteeigenschaften sind hier von großer Wichtigkeit.

EAI lässt sich auf verschiedene Arten durchführen. Hierbei haben sich vier Hauptstilrichtungen herauskristallisiert [3]. Diese sind:

1. **File Transfer**
2. **Shared Database**
3. **Remote Procedure Invocation (RMI)**
4. **Messaging**

Jede dieser Lösungen hat ihre Vor- und Nachteile. Anwendungsintegration kann auch durchgeführt werden, indem mehrere Stilrichtungen eingeschlagen werden. Die flexibelste Lösung ist die des Messaging. Asynchrone Kommunikation entkoppelt den Versender der Nachricht vom Empfänger und die Message-orientierte Middleware (MOM) kümmert sich um den zuverlässigen Versand der Nachrichten (siehe Abschnitt 2.2 Messaging). Eine Applikation stellt hierbei eine Nachricht in einen gemeinsamen Kanal und andere Applikationen können diese Nachricht vom Kanal holen und zu einem späteren Zeitpunkt lesen. Die Applikationen müssen sich sowohl auf den Kanal als auch auf das Nachrichtenformat einigen [3].

2.2 Messaging

Die Messaging-Fähigkeiten werden typischerweise von einem separaten Software System bereitgestellt. Diese Software wird Messaging System oder Message-orientierte Middleware (MOM) genannt [3]. Die Hauptaufgabe der MOM liegt dabei in der zuverlässigen Bewegung der Nachrichten vom Rechner des Versenders zu dem des Empfängers. Da Netzwerkverbindungen zwischen den Applikationen von Natur aus unzuverlässig sind und beim Versenden einer Nachricht noch überhaupt nicht klar ist, ob die Zielapplikation der Nachricht auch bereit ist, diese zu empfangen, wird für eine zuverlässige Nachrichtenübertragung eine MOM benötigt. Die MOM überwindet diese Einschränkungen, indem sie die Übertragung der Nachricht so oft vornimmt, bis sie erfolgreich ist. Die Übertragung der Nachricht erfolgt in 5 Schritten [3]:

1. *Create* – Der Versender erstellt die Nachricht und füllt sie mit den für ihre Bearbeitung relevanten Daten.
2. *Send* – Der Versender gibt die Nachricht an den Kanal weiter.
3. *Deliver* – Die MOM bewegt die Nachricht vom Rechner des Versenders auf den Rechner des Empfängers und macht sie für den Empfänger verfügbar.
4. *Receive* – Der Empfänger liest die Nachricht vom Kanal.
5. *Process* – Der Empfänger extrahiert die Daten aus der Nachricht und verarbeitet sie.

Beim Messaging werden zwei fundamentale Konzepte angewendet [3]:

1. Send and Forget
2. Store and Forward

Sobald die Nachricht versendet ist, kann sich der Versender wieder anderen Aufgaben zuwenden. Er kann sich sicher sein, dass die Nachricht zuverlässig zugestellt wird und muss nicht auf dieses Ereignis warten.

Die MOM nimmt die Nachricht vom Kanal und speichert sie auf dem Rechner des Versenders. Dies erfolgt entweder im Hauptspeicher oder auf persistentem Speicher (z.B. Festplatte). Nun leitet die MOM die Nachricht weiter in Richtung des Empfängers und speichert sie dabei erneut auf jedem Rechner, der einen Zwischenschritt in dem Übertragungsprozess darstellt. Dieser Prozess wiederholt sich so oft, bis die Nachricht schließlich auf dem Rechner des Empfängers angekommen ist. Die Nachricht wird auf der Seite des Versenders erst dann gelöscht, wenn sie sicher beim Empfänger abgespeichert wurde. Auf diese Weise kann dem Versender eine zuverlässige Auslieferung der Nachricht garantiert werden.

Diese Art der Kommunikation ist zwar weniger effizient, da das Versenden einer Nachricht immer auch einen Overhead beinhaltet, dafür ist sie aber sehr robust. Die Auslieferung kann so oft versucht werden, bis sie letztendlich erfolgreich abgelaufen ist.

Messaging als Integrationsstil kann viele Probleme der Anwendungsintegration lösen [3]. Die asynchrone Kommunikation entkoppelt die einzelnen Applikationen voneinander, Sender und Empfänger müssen nicht gleichzeitig erreichbar sein. Neu hinzugefügten Anwendungen beeinflussen die Integrationslösung der bereits integrierten Anwendungen nicht und Prozessschritte, die notwendig sind, um Anwendungen zu integrieren (Transformation, Routing), können der Integrationslösung hinzugefügt werden, ohne dass sich der Sender oder der Empfänger dessen bewusst sein müssen. Auf diese Weise stellt Messaging eine elegante Art der Anwendungsintegration dar [3].

Dies geschieht jedoch nicht, ohne neue Herausforderungen zu stellen. So bedingt die asynchrone Kommunikation ein komplexeres Programmiermodell, das schwieriger zu entwickeln und schwerer zu testen ist. Das Äquivalent zu einem simplen Methodenaufruf ist beim Messaging eine Prozedur, die eine *Request*-Nachricht, einen *Request*-Kanal, eine *Reply*-Nachricht, einen *Reply*-Kanal, eine *Correlation ID* und einen *Invalid Message Channel* benötigt (vergleiche Pattern *Request-Reply* [3]).

Durch Messaging können Nachrichten aus der Reihenfolge geraten. In manchen Situationen, in denen die ursprüngliche Reihenfolge der Nachrichten wichtig ist, muss diese Reihenfolge also wieder hergestellt werden. Da nicht alle Applikationen im Send and Forget Modus arbeiten können und auch synchrone Szenarien unterstützt werden müssen, muss Messaging die Kluft zwischen synchronen und asynchronen Lösungen überbrücken [3]. Durch den Overhead, den die Versendung von Nachrichten einführt und die damit verbundenen Leistungseinbußen, ist Messaging nicht für alle Integrations Szenarien geeignet. So ist z.B. Datenreplikation zur Synchronisierung zweier Systeme ein für Messaging ungeeignetes Szenario. Hinzu kommt, dass viele Messaging Systeme nicht für alle Plattformen erhältlich sind und auf ganz eigenen Protokollen begründet sind. Ein Resultat dessen ist, das Messaging System verschiedener Hersteller nicht ohne Weiteres zusammenarbeiten. Dies führt ein ganz neues Integrationsproblem ein, die Integration von Integrationslösungen [3].

Gregor Hohpe stellt in seinem Buch „*Enterprise Integration Patterns*“ [3] verschiedene Entwurfsmuster dar, die Lösungen für immer wieder auftretende Probleme bereitstellen und den Umgang mit Messaging erleichtern sollen. Dabei halten diese Pattern Methoden und Vorgehensweisen fest, die sich bereits in vielen Projekten bewährt haben. Der Abschnitt 2.4 dieser Diplomarbeit gibt eine kurze Einführung in das Thema Enterprise Integration Pattern.

2.3 Pipes and Filters Architektur

Im einfachsten Fall liefert ein Messaging System eine Nachricht direkt vom Sender zum Empfänger. Oft müssen jedoch zwischen dem Versenden und dem Empfangen der Nachricht gewisse Aktionen auf der Nachricht ausgeführt werden, weil der Empfänger z.B. ein ganz anderes Nachrichtenformat erwartet, als das in der ursprünglichen Nachricht versendet wurde. Die Pipes and Filters (PaF) - Architektur beschreibt, wie multiple Prozessschritte (Filters) verkettet werden können, indem man sie mit Kanälen (Pipes) verknüpft [3].

Diese Ausführungsform erlaubt es, größere Bearbeitungsaufgaben in eine Sequenz von kleinen, unabhängigen Bearbeitungsschritten (Filters) aufzuteilen, die mit Kanälen (Pipes) verbunden sind. Jeder Filter stellt hier eine einfache Schnittstelle zur Verfügung: er empfängt Nachrichten auf der eingehenden Pipe, bearbeitet die Nachrichten und versendet sie auf der ausgehenden Pipe. Eine Pipe verbindet zwei Filter, sie versendet die ausgegebenen Nachrichten des Filters an den nächsten. Zu beachten ist hier, dass die Pipe unidirektional ist; der Nachrichtenfluss geht nur in eine Richtung. Weil alle Filter dieselbe Schnittstelle haben, können sie nach Belieben ausgetauscht, erweitert oder neu angeordnet werden. Viele Enterprise Integration Patterns (EIP, siehe Abschnitt 2.4) basieren auf dieser PaF Ausführungsform. Hierdurch ist es möglich, aus Kombinationen einzelner dieser EIP eine größere Lösung bereitzustellen.

Ein möglicher Nachteil der PaF-Architektur ist die große Anzahl an benötigten Kanälen [3]. Diese sind stehen nicht unlimitiert zur Verfügung, sie kosten Berechnungszeit und konsumieren Hauptspeicher (Buffering). Außerdem stellt eine Nachricht zu versenden immer einen gewissen Overhead dar. Der Preis für die erhöhte Flexibilität äußert sich in möglichen Leistungseinbußen des Systems.

Die ursprüngliche Definition von PaF sieht für jeden Filter genau einen Ein- und Ausgang vor. Im Umfeld des Messaging wird diese Definition jedoch etwas aufgeweicht. Ein *Message Router* [3] hat mehr als einen Ausgang und mehrere Komponenten können an demselben Eingang Nachrichten entgegennehmen.

Pipeline Verarbeitung

Wenn die Filter untereinander mit asynchronen Nachrichtenkanälen verbunden werden, ist es jeder Einheit erlaubt, in ihrem eigenen Prozess oder Thread zu arbeiten. Sobald ein Filter eine Nachricht verarbeitet und weitergegeben hat, kann er bereits die nächste Nachricht verarbeiten. Er muss nicht darauf warten, dass nachfolgende Filter die Nachricht erhalten oder verarbeiten. Dies erlaubt es mehreren Nachrichten gleichzeitig verarbeitet zu werden, während sie durch die individuellen Filter gereicht werden. Dieser Vorgang wird *Pipeline Processing* [3] genannt, weil die Nachrichten durch die Filter fließen, wie das Wasser durch die Leitung. Verglichen mit einem sequentiellen Prozess kann eine Pipeline den Durchsatz des Systems signifikant erhöhen [3]. Die folgende Abbildung veranschaulicht die Verarbeitung in einer Pipeline im Vergleich zu sequentieller Ausführung.

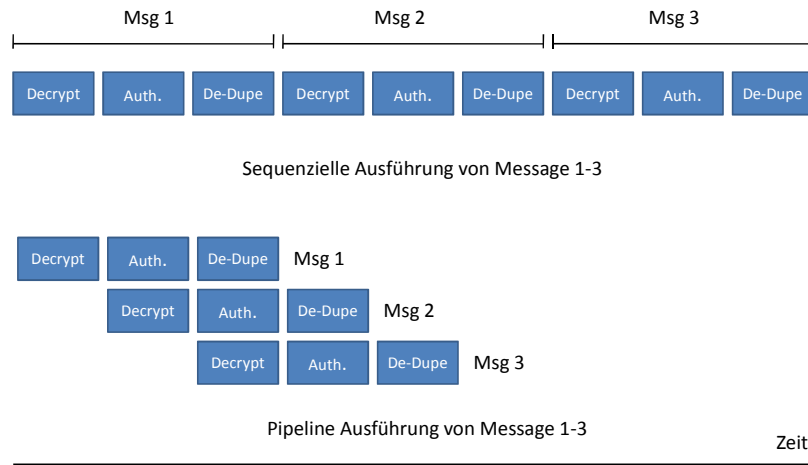


Abbildung 1: Pipeline Verarbeitung mit Pipes und Filters (Quelle [3])

Parallele Verarbeitung

Da die Geschwindigkeit der Ausführung von der des langsamsten Filters in der Kette abhängt, können durch einen Point-to-Point Kanal, an dem mehrere *Competing Consumers* [3] lauschen, mehrere parallele Instanzen dieses Filters aufgestellt werden. Auf diese Weise kann der Durchsatz der Filterkette erhöht werden. Zu beachten gilt es hier jedoch, dass durch die *Competing Consumers* die ursprüngliche Reihenfolge der Nachrichten durcheinander geraten kann. Wenn die Reihenfolge der Nachrichten wichtig ist, kann nur eine Instanz von jedem Filter aufgestellt werden. Eine andere Möglichkeit, die parallele Verarbeitung auch in diesem Fall erlaubt, ist die Benutzung eines *Resequencer* [3].

Eine wichtige Eigenschaft der PaF-Architektur ist, dass es keine globale Sicht auf ihre Prozesse gibt. Dies ist zum Einen bedingt durch die lose Kopplung der Komponenten und zum Anderen durch das Fehlen eines Instanz Begriffs. Nachrichten traversieren hier die Filter der Prozesskette und sind dabei nicht Teil einer bestimmten Instanz dieser Filter. Dies hat auch Vorteile, so ist das System sehr flexibel und tolerant gegenüber Veränderungen, wie z.B. dem Austausch von Filterkomponenten. Aber speziell im Fehlerfall ist dies sehr problematisch, da z.B. bereits getätigte Verarbeitungsschritte im Nachrichtenfluss eventuell rückgängig gemacht werden müssen. Dies ohne globale Statusinformation durchzuführen, macht eine Fehlerbehandlung jedoch sehr schwierig und mühevoll.

2.4 Enterprise Integration Patterns (EIP)

EIP sind schon von einigen Leuten erforscht worden. Die wohl bekannteste Beiträge zu diesem Thema sind die Bücher „*Enterprise Integration Patterns*“ [3] von Gregor Hohpe und Bobby Woolf und

„*Patterns of Enterprise Application Architecture*“ von Martin Fowler. Des Weiteren haben Microsoft und IBM ihre veröffentlichten Patterns zu diesem Themengebiet im Internet zur Verfügung gestellt. Dieser Abschnitt bezieht sich jedoch nur auf das oben erwähnte Buch von Gregor Hohpe, da hier Messaging als Integrationsstil benutzt wird und dieses Buchs außerdem die Basis der vorangegangenen Diplomarbeiten ([1], [2]) bildet. Viele EIP aus diesem Buch wurden dort parametrisiert und lassen sich nun mittels eines grafischen Editors in BPEL und WSDL transformieren.

EAI ist keine einfache Aufgabe. EIP helfen dabei, diese Aufgabe zu bewältigen. Patterns sind dabei nicht etwa Codefragmente, die einfach kopiert und eingefügt werden können oder zusammengeschrumpfte Komponenten, die alles unter Dach und Fach bringen, sondern eher eine Sammlung von Ratschlägen, die Lösungen zu immer wiederkehrenden Problemen beschreiben. Wenn sie richtig angewendet werden, können sie die Kluft zwischen der Vision vom integrierten System und der tatsächlichen Implementierung überbrücken [3].

Die von G. Hohpe zusammengetragenen und erforschten, insgesamt 65 Patterns, gliedern sich in 6 verschiedene Kategorien. Diese sind:

Message Construction

Die Patterns dieser Kategorie beschreiben alle möglichen Arten von Nachrichten. Nachrichten sind Datenpakete, die die Informationen, die zwischen den Applikationen ausgetauscht werden müssen repräsentieren. Message Construction Patterns beschreiben vor allem die verschiedenen Typen dieser Nachrichten und ihre Struktur.

Message Routing

Message Routing kann verwendet werden, um den Sender einer Nachricht vom Empfänger zu entkoppeln. Die Message Router lassen sich in 3 weitere Kategorien unterteilen [3]:

- **Einfache Router** – Varianten von Message Routern, die Nachrichten vom eingehenden Kanal an einen oder mehrere ausgehende Kanäle weiterleiten. Als Beispiele in dieser Kategorie kann der *Content-Based Router* [3] genannt werden, der die Nachricht anhand von ihrem Inhalt weiterleitet.
- **Zusammengesetzte Router** – kombinieren mehrere einfache Router um komplexere Nachrichtenflüsse darzustellen. Als Beispiel kann die Kombination von *Recipient List* und *Aggregator* genannt werden. Der hier entstandene, zusammengesetzte Router, ist der *Scatter-Gather* [3]. Er kann u. A. genutzt werden, um ein Auktionsszenario zu implementieren, bei dem das beste Angebot den Zuschlag erhält.
- **Architektonische Patterns** – Message Router geben die Möglichkeit, eine Integrationslösung zu erstellen, die auf einem zentralen *Message Broker* [3] basiert. Mit diesem Pattern kann eine *Hub-and-Spoke* [3] Architektur beschreiben werden.

Message Transformation

Da sich die entkoppelten Applikationen nicht auf ein gemeinsames Format einigen müssen, übernehmen die *Message Transformation* Patterns diese Aufgabe. Ihr Aufgabenfeld besteht darin, Nachrichten in ein anderes Format umzuwandeln, den Nachrichteninhalte zu erweitern (*Content*

Enricher [3]) oder anpassen oder Nachrichten nach dem Need-to-Know Prinzip umzugestalten, um sensitive Daten vor den Augen anderer zu verbergen (*Claim-Check* [3]).

Messaging Endpoints

Die *Messaging Endpoint* Patterns dienen als eine Art Brücke zwischen dem Messaging System und den Applikationen, die Nachrichten senden und empfangen müssen. Die Patterns dieser Kategorie beschreiben hierbei verschiedene Möglichkeiten, wie die Applikationen Nachrichten senden oder empfangen können.

Messaging Channels

Bei den Patterns dieser Kategorie handelt es sich um die Beschreibung der verschiedenen Typen und Kommunikationsmechanismen der Kanäle. In ihr werden auch Patterns beschrieben, die mit Fehlersituationen im Messaging System selbst (*Dead Letter Channel* [3]) oder mit der Nachricht an sich (*Invalid Message Channel* [3]) umgehen.

System Management

Da in einem nachrichtenbasierten Integrationsnetz eine riesige Anzahl an Nachrichten generiert, versendet, weitergeleitet, transformiert und konsumiert werden, muss es auch eine Möglichkeit geben, dieses System zu verwalten, zu kontrollieren, zu analysieren und zu testen. Die Patterns dieser Kategorie geben hierbei eine Hilfestellung. Die drei Kategorien, in die sie sich weiter aufteilen, sind [3]:

- **Überwachung und Kontrolle** – Ein *Control Bus* [3] liefert einen zentralen Punkt der Kontrolle. Er verbindet mehrere Komponenten mit einer zentralen *Management Konsole*. Hierauf wird in Abschnitt 2.4.1 näher eingegangen.
- **Beobachtung und Analyse** – Patterns aus dieser Kategorie helfen beispielsweise dabei, den Inhalt einer Nachricht zu inspizieren (*Wire Tap* [3]), oder den gesamten Nachrichtenverkehr festzuhalten (*Message Store* [3]), ohne dabei den regulären Nachrichtenfluss zu beeinflussen.
- **Test und Fehlerbeseitigung** – Um die korrekte Funktion von Komponenten zur Laufzeit zu überprüfen, gibt es das Pattern *Test Message* [3]. Durch periodisches Injizieren, sowie die Verifizierung der Resultate, kann so das einwandfreie Verhalten einer Komponente sichergestellt werden.

Die auf der nächsten Seite aufgeführte Abbildung gibt einen Überblick über die Patterns und ihre jeweilige Kategorie und Einordnung im integrativen Nachrichtenfluss der Applikationen.

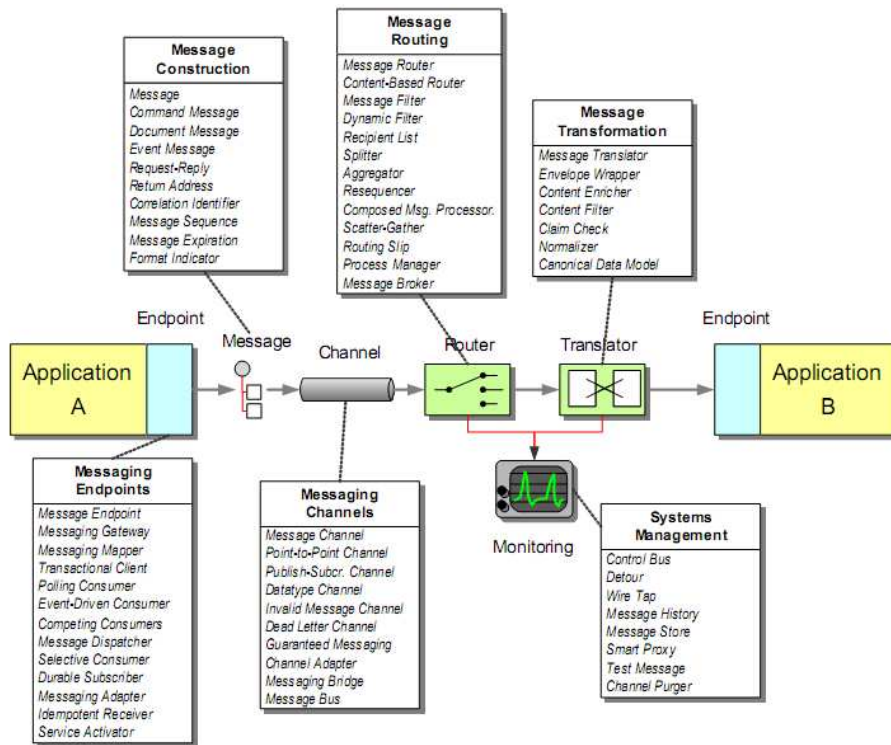


Abbildung 2: EIP bei der Integration von Applikationen (Quelle [11])

2.4.1 Der Control Bus

Der *Control Bus* verbindet alle Komponenten der Integrationslösung mit zusätzlichen Kanälen für den Kontrollnachrichtenfluss. Jede Komponente im System ist nun verbunden mit zwei Messaging Subsystemen [3]:

1. Dem des anwendungsspezifischen Nachrichtenflusses
2. Dem des Control Bus

Die unten stehende Abbildung zeigt eine Filterkette, die mit den beiden oben erwähnten Messaging Subsystemen verbunden ist.

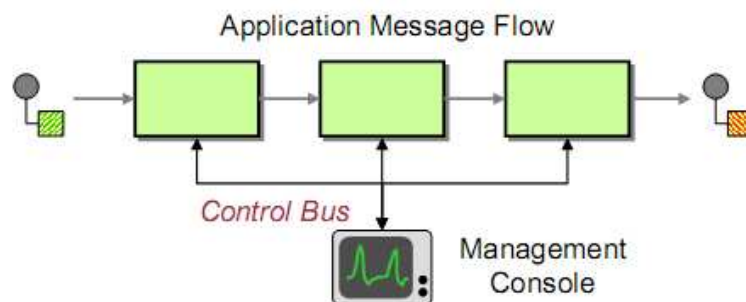


Abbildung 3: Das Pattern Control Bus und die Management Konsole (Quelle [13])

Während der anwendungsspezifische Nachrichtenfluss alle Nachrichten zwischen den zu integrierenden Anwendungen, also in der Integrationslösung transportiert, existieren weitere Kanäle, von denen die Komponenten Nachrichten erhalten und an die die Komponenten Nachrichten versenden können. Diese weiteren Kanäle bilden den Control Bus und schließen sich in einer zentralen Verwaltungskomponente zusammen, der *Management Konsole*. Der Control Bus selbst ist geeignet um folgende Nachrichtentypen zu transportieren [3]:

- **Konfigurationsnachrichten** – Jede Komponente, die in dem applikationsspezifischen Nachrichtenfluss tätig ist, sollte konfigurierbare Parameter haben, die nach Bedarf geändert werden können. Zum Beispiel kann auf diese Weise die Routing-Tabelle eines *Content-Based Router* [3] dynamisch verändert werden, um Komponentenfehlern oder Überladung des Systems entgegen zu wirken.
- **Heartbeat** – Jede Komponente kann periodisch eine Heartbeat Message auf dem Control Bus versenden, die angibt, dass die Komponente funktioniert und die metrische Daten über sie enthalten kann.
- **Test Messages** – Heartbeat Messages zeigen an, dass eine Komponente erreichbar ist. Die Korrektheit selbst kann jedoch nur mit *Test Messages* überprüft werden.
- **Exceptions** – Jede Komponente kann über den Control Bus Ausnahmesituationen anzeigen, die z.B. in der Management Konsole ausgewertet werden können. Auch ein Anzeigen von Bestätigungen (Acks) bei erfolgreicher Verarbeitung wäre hier durchaus denkbar.
- **Statistiken** – Jede Komponente kann Statistiken sammeln über die Anzahl der verarbeiteten Nachrichten, die durchschnittliche Verarbeitungszeit, und so weiter. Mit diesen Daten kann der Nachrichtenfluss analysiert und gegebenenfalls verbessert werden.
- **Live Console** – Die meisten dieser Funktionen können aggregiert werden und in einer zentralen Management Konsole angezeigt werden. Von hier aus kann man die Lauffähigkeit des Gesamtsystems betrachten und, falls nötig, korrektiv eingreifen.

Viele dieser genannten Funktionen, die ein Control Bus unterstützt, ähneln den traditionellen Netzwerkmanagement Funktionen, die genutzt werden, um ein Netzwerk zu überwachen und zu verwalten [3].

2.4.2 Die Management Konsole

Während der Control Bus die Infrastruktur für den Kontrollnachrichtenfluss darstellt, ist die Management Konsole die zentrale Verwaltungskomponente. In ihr werden alle Heartbeat Messages ausgewertet und die verschiedenen Kontrollnachrichten für die jeweiligen Komponenten (Filter) mit einer internen Logik generiert und über den Control Bus versendet.

Dies kann entweder durch eine direkte Adressierung des Empfängers in der Kontrollnachricht, oder über einen Broadcast (*Publish-Subscribe Channel* [3]) erfolgen. Ersteres setzt eine engere Kopplung der Management Konsole an die Komponenten voraus und führt zu einem unflexibleren System. Wird die Kontrollnachricht einfach durch einem Broadcast versendet, so sind die Komponenten selbst dafür verantwortlich, die (un)passenden Kontrollnachrichten anhand gewisser Filterkriterien

(Prozess-ID, Nachrichtentyp), abzugleichen (oder zu ignorieren). Dies führt zwar zu einem erhöhten Nachrichtenfluss auf dem Control Bus, behält aber die lose Kopplung der Komponenten bei. Da der Control Bus ein Subsystem des Messaging Systems ist, wird der anwendungsspezifische Nachrichtenfluss jedoch hiervon nicht beeinträchtigt.

Die Einführung des Control Bus kann ein Mittel sein, um Dienstgüteeigenschaften für die PaF-Architektur umzusetzen. Er bietet die Möglichkeit, eine zentrale Kontrollkomponente einzuführen, die es erlaubt, Management Funktionen umzusetzen. Da es in einer PaF-Architektur keine globale Sicht auf den Prozess gibt und die einzelnen Filter zustandslos sind, kann die Management Konsole auch verwendet werden, um Statusinformationen des Prozesses festzuhalten. Hierfür wäre jedoch eine Kommunikation aller Filter mit der Management Konsole über den Control Bus notwendig.

2.5 Dienstgüteeigenschaften (Quality of Service)

Unter dem Begriff Dienstgüteeigenschaften (*englisch: Quality of Service (QoS)*) werden Eigenschaften zusammengefasst, die die Qualität eines Dienstes ausmachen. Hierzu gehören:

- **Verfügbarkeit**
- **Zuverlässigkeit**
- **Sicherheit**
- **Robustheit (Fehlersicherheit, transaktionales Verhalten)**
- **Latenz und Durchsatz**

Unter der **Verfügbarkeit** eines Dienstes wird die Erreichbarkeit desselben verstanden. Ein Dienst, der zur gegebenen Zeit für den Klienten nicht erreichbar ist, kann seinen Service auch nicht anbieten. Gründe hierfür können z.B. ein Systemausfall oder eine zu volle Queue auf der Seite des Dienstes sein, die keine weiteren Requests mehr gestattet. Eine geeignete Maßnahme, um die Verfügbarkeit eines Dienstes zu erhöhen, ist z.B. mehrere Instanzen dieses Dienstes redundant zur Verfügung zu stellen und an einem *Point-to-Point Channel* [3] mehrere *Competing Consumers* [3] zu haben, die die einkommenden Nachrichten an ihre jeweilige Instanz des Dienstes weiterleiten. Ein Ausfall eines einzelnen Consumers hätte somit keine Auswirkungen auf die Verfügbarkeit des Dienstes [3].

Die **Zuverlässigkeit** des Dienstes spiegelt außer seiner Verfügbarkeit noch die semantische Korrektheit des Dienstes wider. Das bedeutet, nur weil ein Dienst verfügbar ist, ist noch nicht gewährleistet, dass er auch korrekte Ergebnisse liefert. Die Korrektheit eines Dienstes kann in einer auf EAI Patterns basierenden Integrationslösung mit dem Pattern *Test Message* [3] überprüft werden. Ein weiterer Punkt ist die Zuverlässigkeit des Nachrichtenaustausches. Diese Dienstgüteeigenschaft wird vom z.B. vom unterliegenden Messaging System mit dem Store-and-Forward Mechanismus (vergleiche Abschnitt 2.2) unterstützt. Ein Pattern, das diesen Vorgang beschreibt, ist *Guaranteed Delivery* [3].

Die **Sicherheit** eines Dienstes wird z.B. durch die Verschlüsselung sensibler Daten gewährleistet. Sicherheit kann auch durch ein geeignetes Transportprotokoll gewährleistet werden. Da Geschäftsprozesse typischerweise nicht nur über verschiedene Plattformen mit verschiedenen Betriebssystemen verteilt sind, sondern auch über verschiedene Netzwerke verteilt sein können,

kann die Wahl eines sicheren Transportprotokolls auf der Seite des Versenders jedoch nicht garantieren, dass beim Verlassen des Netzwerks des Versenders dieses Transportprotokoll auch beibehalten wird. Verlässt eine Nachricht das lokale Netzwerk, dann ist die Sicherheit Sache des Versenders und des Empfängers. Ist dies gewünscht, so muss es ein explizites Verschlüsseln und Entschlüsseln der Nachricht z.B. durch einen jeweiligen Filter in der Prozesskette der Integrationslösung geben.

Robustheit bedeutet im weitesten Sinne Fehlertoleranz. Ein Filter darf durch einen fehlerhaften Request nicht gleich zusammenbrechen und seinen Dienst verweigern. Dies ist jedoch Sache der Implementierung dieses Filters. Eine Integrationslösung verhält sich robust, wenn sie auch im Fehlerfall zu einem konsistenten und zufriedenstellenden Ergebnis führt. Dies kann im besten Fall z.B. durch eine fehlertolerante Modellierung der Filter der Prozesskette erreicht werden. Hierbei kann versucht werden, einen Fehlerfall durch vorher modellierte, alternative Pfade abzufangen. Eine weitere Möglichkeit, dem Ausfall eines Filters zu begegnen, ist die Kontaktherstellung so oft zu wiederholen, bis diese schließlich erfolgreich ist.

Da dies in Geschäftsprozessen jedoch nicht immer ausreichend ist und sich das System nach einem Fehler, der nicht abgefangen werden kann (z.B. nach dem Erreichen eines vordefinierten Retry-Schwellenwerts, das zum *abort* einer Transaktion führt) in einem inkonsistenten Zustand befinden kann, muss es eine Möglichkeit zur Kompensation von vorangegangenen Arbeitsschritten der Prozesskette der Integrationslösung geben. Das bedeutet, ein robustes System muss für diesen Fall ein kompensationsbasiertes Transaktionskonzept bieten. Kompensationsaktionen müssen in einer PaF-Architektur mit EIP explizit modelliert werden.

Ein traditionelles Konzept für kurzlaufende Prozesse ist eine koordinierte Konversation mit einem 2-Phasen-Commit Protokoll. Dies entbindet jedoch nicht von der Notwendigkeit eines Kompensationsmechanismus, da auch erfolgreich ausgeführte Transaktionen zu einem späteren Zeitpunkt noch scheitern können (Produktionsprobleme, Änderungen von Bestimmungen, Produktmängel).

Das Ziel dieser Diplomarbeit ist es, Möglichkeiten zur Einbindung und Umsetzung dieser Konzepte in das bestehende System zu untersuchen. Eine genauere Betrachtung hierzu wird in Kapitel 4 und 5 erfolgen.

Die **Latenz** eines Systems stellt einen weiteren wichtigen Punkt der Dienstgüte dar. Ist ein Filter der Prozesskette mit Anfragen überlastet, so kann sich die Laufzeit einer Nachricht durch die Integrationslösung erheblich verzögern und zu nicht wünschenswert langen Wartezeiten führen. Hierdurch wird auch der **Durchsatz** von Nachrichten durch das System limitiert. Ist der Klient, der die Anfrage stellt, ein Mensch, so wird er eine Wartezeit auf das Ergebnis von mehreren Minuten oder gar Stunden nicht tolerieren. Um diesem Punkt entgegenzuwirken, muss zunächst der Filter identifiziert werden, durch den dieses Problem entsteht. Ist dies geschehen, können durch Maßnahmen wie Load-Balancing oder Hot-Pooling neue Instanzen des überladenen Filters gestartet werden. Eine Möglichkeit, dies in einem EIP-basierten System umzusetzen, wurde bereits unter dem Punkt Verfügbarkeit aufgezeigt. Eine weitere Möglichkeit wäre hier, einen *Message Dispatcher* [3] zu verwenden, der gezielt Load-Balancing durchführen könnte, indem er bei einer kritischen Menge von Anfragen eine neue Instanz des Filters aktiviert und die Nachrichten nun auf diese Instanzen verteilt.

2.6 Parametrisierung der EAI Patterns

In den vorangegangenen Arbeiten [1] und [2] wurden die EIP parametrisiert. Hierbei wurde für jedes Pattern untersucht, welche Eingaben das Pattern benötigt, welche Ausgaben es liefert und welche Einstellungen vom Nutzer festgelegt werden müssen, damit das Pattern seine Funktion erfüllen kann.

Damit dies unabhängig von der späteren Art der Umsetzung geschieht, wurden die Patterns hierzu zunächst auf abstrakter Ebene untersucht. Die Ein- und Ausgaben und Parameter der Patterns wurden unabhängig davon untersucht, dass sie später auf BPEL abgebildet werden sollen [1].

Zur besseren Übersicht wurde dieser Vorgang grafisch dargestellt. Die folgende Abbildung zeigt eine beispielhafte Darstellung eines parametrisierten Patterns.

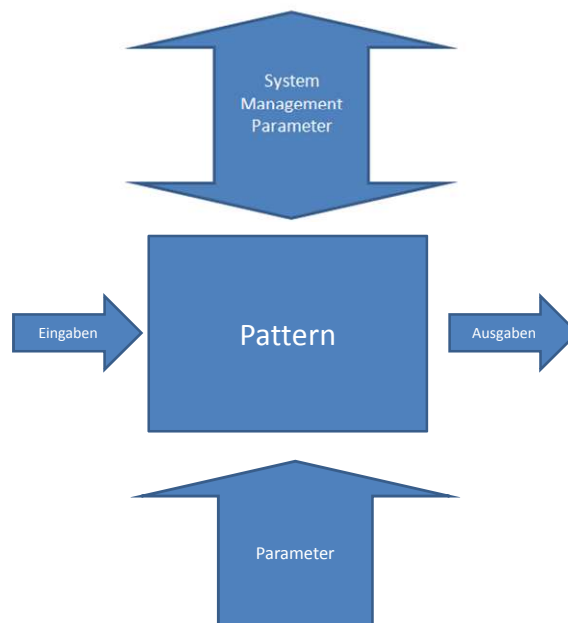


Abbildung 4 : Grafische Notation eines Patterns

Die Ein- und Ausgaben des Patterns stellen die Nachrichten dar, die durch das Pattern geleitet werden. Die System Management Parameter sind die Verbindung des Patterns zum Control Bus und zur Management Konsole. Hier können Konfigurationsnachrichten eingehen oder Fehlermeldungen versendet werden. Im unteren Pfeil werden die verschiedenen Parameter schriftlich zusammengefasst, die das Pattern besitzen muss, dass es seine Funktion durchführen kann.

Durch die Parametrisierung der Patterns wurden Überlegungen angestellt, welche Einstellungen vom Benutzer vorgenommen werden müssen, damit das Pattern seine Funktion erfüllt und implementiert werden kann. Eine Parametrisierung der einzelnen EIP kann in [1] und [2] nachgelesen werden. Da die Parametrisierung der Patterns neutral durchgeführt wurde, also ohne speziellen Hinblick auf eine spätere Abbildung auf BPEL, wurden hierbei Parameter, die zur Nutzung des Fehlerbehandlungskonzepts von BPEL notwendig sind, noch nicht berücksichtigt. Änderungen diesbezüglich werden im fünften Kapitel dieser Diplomarbeit aufgezeigt.

3 Fehlerbehandlungskonzepte

In diesem Kapitel werden die Fehlerbehandlungskonzepte von Web Services, WS-BPEL und Workflow Systemen näher untersucht. Da die parametrisierten EIP vom System „EAItoBPEL“ in BPEL transformiert werden, ist eine Untersuchung dieser Konzepte an dieser Stelle sehr sinnvoll, weil BPEL ein Teil dieser Technologien ist und einen Zusammenhang zwischen ihnen herstellt.

BPEL selbst ist ein Web Service Standard und lässt sich also auch mit den Web Service Standards für die Dienstgüteeigenschaften, die für Web Services in der Quality of Service Schicht definiert sind, komponieren. Eine Betrachtung der Standards dieser Schicht erfolgt in **Abschnitt 3.1** dieses Kapitels.

Im **Abschnitt 3.2** wird dann das Fehlerbehandlungs- und Kompensationskonzept von BPEL selbst näher betrachtet und zusammengefasst.

Da eine PaF-Architektur wie bereits erwähnt auch Gemeinsamkeiten mit einem Workflow aufweist und diese Konzepte, da beide Graphen-orientiert, durchaus vergleichbar sind ([16]) und BPEL in Workflow Systemen genutzt wird, um Geschäftsprozesse als und aus verschiedenen Web Services zusammenzustellen und ihr Verhalten zu definieren, wird in **Abschnitt 3.3** auf die Fehlerbehandlungs- und transaktionalen Konzepte von Workflow Management Systemen (WfMS) eingegangen.

3.1 Quality of Service bei Web Services

Dieser Abschnitt behandelt kurz die verschiedenen Spezifikationen der Quality of Service Schicht des Architekturmodells des Service Bus für serviceorientierte Architekturen (SOA). Die Web Service Technologie ist eine Basis um SOA zu implementieren und der Service Bus ist das Herzstück dieser Implementierung [4]. Die folgende Abbildung zeigt das Architekturmodell des Service Bus.

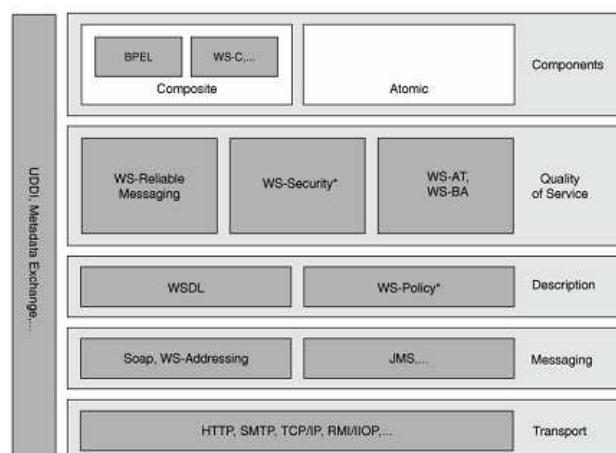


Abbildung 5: Der SOA Stack

Zu den Komponenten der Quality of Service Schicht gehören u. A. die Spezifikationen WS-Reliable Messaging, WS-Security und WS-Transactions mit ihren Spezifikationen WS-Coordination, WS-Atomic Transaction und WS-BusinessActivity [4]. In den nun folgenden Abschnitten werden diese Spezifikationen etwas genauer betrachtet.

3.1.1 WS-ReliableMessaging

Die Web Service Spezifikation WS-ReliableMessaging (WS-RM) [4] definiert Protokolle, die Web Services befähigen, zuverlässigen und interoperablen Nachrichtenaustausch zu gewährleisten, indem sie gewisse Auslieferungszusicherungen bieten. Bei diesen Auslieferungszusicherungen handelt es sich um:

- *In-order delivery* - Auslieferung der Nachrichten in derselben Reihenfolge, in der sie versendet wurden.
- *At least once delivery* - Jede Nachricht wird mindestens einmal ausgeliefert.
- *At most once delivery* - Es werden keine Duplikate der Nachricht ausgeliefert.

Diese Zusicherungen können auch kombiniert werden, wodurch neue Auslieferungszusicherungen entstehen. Beispielsweise kann man den 2. und den 3. Punkt kombinieren, um exakt eine Nachricht auszuliefern (*Exactly once delivery*). Das WS-RM Protokoll ermöglicht es, zwischen verschiedenen Betriebs- und Middleware-Systemen zuverlässigen Nachrichtenaustausch vorzunehmen.

3.1.2 WS-Security

Sicherheit ist für Unternehmen von fundamentaler Wichtigkeit. WS-Security (WS-S) [4] ist der Grundbaustein für sichere Web Services. Heutzutage verlassen sich die meisten verteilten Web Services bei ihren Sicherheitsfunktionen auf Unterstützung durch die Transportschicht (HTTPS-Protokoll, BASIC-Auth). Diese Ansätze liefern zwar eine minimale Basis für eine sichere Kommunikation, stehen in den Funktionen, die heutige, verteilte Systemumgebungen oder Middleware-Produkte haben, aber weit zurück [4].

WS-S nutzt existierende Sicherheitsmodelle wie Kerberos oder X509. Die Spezifikation definiert außerdem konkret, wie man existierende Modelle in einer interoperablen Weise nutzen kann. Sicherheit in Web Services ist auf vordefinierten Vertrauensbeziehungen begründet. Im Falle von Key-basierter Verschlüsselung verlassen sich die Teilnehmer auf die Verteilungszentren für die Schlüssel (Kerberos Key Distribution Center) oder auf digitale Zertifikate, die die Echtheit eines Schlüssels bestätigen (Public Key Infrastructure, PKI).

Die Spezifikation WS-Trust [4] definiert ein erweiterbares Modell um solche Vertrauensbeziehungen aufzustellen und zu überprüfen, indem sie einen Security Token Service (STS) bereitstellt. Dieser STS ist ein Web Service, der Security Tokens ausstellt, sie vermittelt und validiert. WS-Trust erlaubt es Web Services, sich auf Security Server zu einigen, denen sie vertrauen und sich dann an diese zu binden.

Eine weitere Spezifikation von WS-S ist WS-SecureConversation [4] (WS-SC). Da WS-S nur ein simples Modell für sichere Nachrichtenübertragung darstellt, das sich vor allem für Szenarien eignet, in denen nur einige wenige Nachrichten zwischen den Web Services ausgetauscht werden, stellt WS-SC eine Erweiterung der WS-S Spezifikation her, bei der auch Szenarien abgedeckt werden, in denen Web Services ausgedehnten und multiplen Nachrichtenaustausch vornehmen müssen.

Würde man sich hier auf PKI für Signaturen und Verschlüsselung verlassen, hätte dies ineffizientere Berechnungen zur Folge. Außerdem stellt die Menge an Information, die verschlüsselt werden muss, eine Bedrohung des Sicherheitskonzepts an sich dar. Je mehr Information mit einem Schlüssel verschlüsselt wird, desto einfacher ist es auch, diesen Schlüssel zu knacken.

WS-SC löst dieses Problem, indem, nachdem die Teilnehmer mit WS-S und PKI eine Konversation gestartet haben, ein Session-spezifischer Satz von Schlüsseln übertragen wird, auf den sich die Teilnehmer über WS-SC einigen und mit denen der nun folgende Nachrichtenfluss verschlüsselt wird. Dies erlaubt eine effizientere Verschlüsselung und verbessert die Sicherheit der Schlüssel.

Eine weitere Spezifikation, die hier erwähnt werden sollte, ist WS-Federation [4] (WS-F). WS-F ist eine Erweiterung von WS-Trust, die es mehreren Organisationen erlaubt, sich zu einer einzelnen Sicherheitsdomäne zusammenzuschließen. Wenn sich nun ein Endnutzer bei einem Mitglied dieser Sicherheitsdomäne einloggt, so ist er auch bei den restlichen Mitgliedern dieser Domäne eingeloggt. Dies vereinfacht die Kommunikation mit den Mitgliedern dieser Domäne, da sich der Endnutzer nur ein einziges Mal anmeldet und sich dann nicht erneut authentifizieren muss. Dies reduziert Kosten, die für Identitätsmanagement entstehen, mildert Sicherheitsrisiken ab und verbessert die Interaktion des Nutzers mit dem System.

Weitere Information über Spezifikationen von WS-S und WS-F können in [4] nachgelesen werden.

3.1.3 WS-Transaction

In heutigen Geschäftsszenarien sind häufig Applikationen beteiligt, die aus vielen Web Services bestehen, die wiederum eine Vielzahl von Nachrichten miteinander austauschen müssen [4]. Solche Applikationen können sehr komplex sein, ihre Ausführung erstreckt sich oft auf viele heterogene, verteilte und lose gekoppelten Systeme. Dadurch ist ihre Ausführung sehr fehleranfällig und es entstehen signifikante Zuverlässigkeitsprobleme. Diese Applikationen müssen mit Fehlern einer jeden beteiligten Web Service Komponente im Kontext der gesamten Applikation umgehen können. Ein koordiniertes Zusammenspiel der Ergebnisse der teilnehmenden Dienste, die die Geschäftsfunktion ausmachen, ist hierfür unerlässlich. Dadurch wird erreicht, dass ein einheitliches Ergebnis aller beteiligten Applikationen garantiert werden kann.

Um dies zu gewährleisten, müssen involvierte Web Services befähigt werden, Aktivitäten auszuführen, die mit anderen Aktivitäten koordiniert werden müssen und sich dann auf ein einheitliches Gesamtergebnis dieser Aktivitäten zu einigen. WS-Coordination (WS-C) [4] ist ein globaler Mechanismus, um Web Service Anwendungen verschiedener Parteien zu initiieren, zu koordinieren und auf ein einheitliches Resultat festzulegen. WS-C ist ein Standard, der zusammen

von IBM, BEA und Microsoft erarbeitet wurde und in seiner Version 1.0 erstmals im September 2003 vorgelegt wurde. Die derzeit aktuelle Version WS-Coordination 1.1 wurde im Juli 2007 veröffentlicht.

Das WS-C Framework hat hierbei folgende Bestandteile [4]:

- Ein **Aktivierungsdienst**, der die Aktivität erzeugt und einen Koordinationskontext herstellt – Die WS-Addressing [4] Endpoint Referenz eines Koordinationsdienstes und zusätzliche Information, die die spezielle Aufgabe identifiziert, die koordiniert werden muss (u. A. einen eindeutigen Identifier).
- Ein **Registrierungsdienst**, der die Möglichkeit bietet, sich als Teilnehmer in dieser Aktivität zu registrieren und die Protokollauswahl für die Konversation koordiniert.
- Ein **Koordinationsdienst** – Dieser liefert einen Dienst, eine koordinierte Aktivität zu starten, sie korrekt zu beenden, und stellt den Koordinationskontext her, der Teil aller Nachrichten ist, die innerhalb der Teilnehmergruppe versendet werden.
- Eine **Schnittstelle** – Die Teilnehmer können diese Schnittstelle nutzen, um über das Resultat, auf das sich geeinigt wurde zu informieren.

Um das WS-C Framework zu ergänzen und es an die unterschiedlichen Bedingungen, die an die Geschäftsprozesse und an die Resultate der Teilnehmer gestellt werden, anzupassen, gibt es die Protokolle **WS-AtomicTransaction** (WS-AT) [4] und **WS-BusinessActivity** (WS-BA) [4]. Sie gliedern sich in das WS-C Framework ein.

WS-AT definiert einen Satz von Protokollen, die sich in das WS-C Framework einfügen, um das traditionelle 2-Phasen-Commit (2PC) Transaktionsprotokoll zu implementieren. Dieses Protokoll gewährleistet die **ACID-Eigenschaften** [4] von Transaktionen.

Diese besagen:

- Eine Transaktion wird entweder ganz oder gar nicht ausgeführt (**Atomicity**).
- Eine Transaktion hinterlässt einen konsistenten Zustand (falls der Zustand vorher auch konsistent war) (**Consistency**).
- In der Ausführung befindliche Transaktionen beeinflussen sich nicht gegenseitig, d.h. Zwischenergebnisse der Transaktion sind für andere Transaktionen unsichtbar. Dies wird durch Sperrprotokolle forciert, die den Zugang zu den beteiligten Ressourcen für die Dauer der Transaktion blockieren (**Isolation**).
- Nachdem eine Transaktion erfolgreich beendet wurde, sind ihre Ergebnisse dauerhaft (**Durability**).

ACID Transaktionen oder WS-AT sind jedoch nicht immer die geeignete Wahl für alle Arten von Business Transaktionen. Transaktionsprotokolle für Business Transaktionen müssen auch mit langen Laufzeiten der Aktivitäten umgehen können. Solche langlaufenden Transaktionen (Long Running Transactions, LRT) erfordern es, die oben genannten ACID Eigenschaften in einigen Punkten aufzuweichen.

Die einzelnen Transaktionen können nicht für die Gesamtdauer der LRT isoliert werden, da es sonst zu signifikanten Leistungseinbußen des Systems kommen kann. Auch kann es möglich sein, dass gewisse, in die Transaktion verwickelte, Ressourcen nicht auf eine externe Koordination vertrauen und es nicht zulassen, dass ein solcher Koordinator ihre Ressourcen für eine unbestimmte Zeit sperrt,

da hierdurch z.B. eine Denial-of-Service (DoS) Attacke gegen sie gefahren werden könnte. Da auf Grund der Sperrprotokolle und dem daraus resultierenden Blockieren der Ressourcen der Durchsatz des Systems verringert wird, kann sich die Latenz des Systems signifikant erhöhen. Möglicherweise sind in LRTs auch Aktivitäten beinhaltet, die ein menschliches Eingreifen erfordern. Ein Blockieren aller beteiligten Ressourcen während der gesamten Laufzeit der LRT ist also oft nicht möglich und auch nicht praktikabel. LRTs müssen einen anderen Weg als die traditionellen ACID Transaktionen einschlagen, um diesen Problemen zu begegnen.

Da die Ressourcen nicht für eine längere Zeitdauer gesperrt werden können, muss zunächst die Isolation der an der LRT beteiligten Transaktionen aufgegeben werden. Die Zwischenzustände der LRT werden also sichtbar gemacht und andere Aktivitäten können auf diese Ergebnisse zugreifen, bevor das endgültige Resultat der LRT feststeht. Auch die Atomizität der LRT muss aufgeweicht werden. Es ist durchaus denkbar, dass die LRT fehlschlägt, nachdem schon viele einzelne ACID Transaktionen erfolgreich beendet haben. In diesem Fall müssen Kompensationsmechanismen angewendet werden, um das System wieder in einen konsistenten Zustand zu überführen.

WS-BusinessActivity (WS-BA) [4] definiert einen Satz von Protokollen, die sich in das WS-C Framework einfügen, um solche langlaufenden, kompensationsbasierten Transaktionsprotokolle zu liefern. Eine genauere Beschreibung dieser Protokolle kann in [4] nachgelesen werden.

3.2 WS-BPEL

BPEL4WS ist eine Kombination aus zwei vorangegangenen, konkurrierenden XML Dialekten zur Komposition von Geschäftsprozessen: der IBM Web Service Flow Language (WSFL) und Microsoft XLANG. In BPEL wird ein Prozess kreiert, indem man eine Kombination des Graphen-orientierten Stils von WSFL und des algebraischen Stils von XLANG anwendet. BPEL erlaubt es, Web Services rekursiv zusammenzustellen und in neue Web Services zu aggregieren [4]. Die erste Version der BPEL4WS Spezifikation wurde im Jahr 2002 herausgegeben, die Version 1.1, die 2003 erschien, wurde OASIS (*Organization for the Advancement of Structured Information Standards*) zur Standardisierung übergeben. Die Version 2.0 von BPEL wurde umbenannt in WS-BPEL, um der Schreibweise der anderen Web Service Spezifikationen gerecht zu werden. Die Web Service Spezifikation WS-BPEL wurde am 11. April 2007 veröffentlicht [6].

Dieser Abschnitt beschreibt Ausschnitte der Fehlerbehandlungs- und Kompensationsmechanismen von WS-BPEL. Es wird nur soweit auf diese Mechanismen eingegangen, wie für diese Arbeit sinnvoll und notwendig ist. Für eine genaue Betrachtung der Spezifikation und der oben erwähnten Mechanismen wird auf [6] verwiesen.

3.2.1 Scopes

Das `<scope>` Konstrukt in WS-BPEL schafft den Kontext, der das Ausführungsverhalten seiner eingeschlossenen Aktivitäten beeinflusst [6]. Es beinhaltet Variablen (in BPEL werden Nachrichten in Variablen kopiert), Partner Links, Message Exchanges, Correlation Sets und die Event- (EH), die

Termination- (TH), die Fault- (FH) und die Compensation Handlers (CH). Scopes können hierarchisch verschachtelt werden, sie dürfen sich jedoch nicht überschneiden. Den Wurzelkontext stellt das `<process>` Konstrukt her.

Im Gegensatz zum `<scope>` Konstrukt können am `<process>` Konstrukt keine CH oder TH angefügt werden. Weiter können keine WS-BPEL Standardelemente oder Standardattribute ans `<process>` Konstrukt angefügt werden, da es sich im Gegensatz zum `<scope>` Konstrukt nicht um eine Aktivität handelt. Das *isolated* Attribut ist nicht auf das `<process>` Konstrukt anwendbar.

Jeder Scope hat eine benötigte *primary activity*. Diese reflektiert das normale Verhalten. Alle anderen Aktivitäten sind optional, einige haben Default-Semantik. Der gelieferte Kontext vom Scope wird von den verschachtelten Aktivitäten geteilt.

Die **Initialisierung** beginnt, sobald in das `<process>` oder `<scope>` Konstrukt eingetreten wird. Die FH und die TH werden installiert. Die *primary activity* wird gestartet, parallel dazu werden die EH installiert. Wenn bei der Initialisierung ein Fehler entsteht, wird ein *bpel:scopeInitializationFailure* an den übergeordneten Scope, den **Vaterscope**, weitergegeben. Bei einem Initialisierungsfehler auf Prozessebene (innerhalb des `<process>` Konstrukts) schlägt der gesamte Prozess fehl.

Vaterscope:

Ein Scope S ist Vaterscope von X, wenn X in S ist. Dabei ist X Kindscope von S.

Partnerscope:

Ein Partnerscope P1 und P2 sind Scopes, die sich nicht überschneiden und die nicht in einem Vater- oder Kind Verhältnis zueinander stehen. Sie haben einen gemeinsamen Vaterscope. Dieser kann entweder selbst in einem Vater- oder Kind Verhältnis mit einem anderen Scope stehen oder Kindscope vom Prozess oder der Prozess selbst sein.

3.2.2 Fehlerbehandlung in WS-BPEL

Geschäftsprozesse sind oft von langer Dauer, die Fehlerbehandlung ist schwierig aber unerlässlich. ACID Semantik (vergleiche Abschnitt 3.1.3) limitiert sich auf lokale Updates, Sperrmechanismen können nicht für die gesamte Dauer des Prozesses gehalten werden und manche externen Partner lassen keine Kontrolle über die Sperrung ihrer Ressourcen von außen zu [6] (vergleiche auch 3.1.3 WS Transaction – WS-BA).

Das Resultat, das hieraus hervorgeht ist, dass die gesamte Transaktion fehlschlagen kann, nachdem bereits viele einzelnen, atomaren Transaktionen erfolgreich abgeschlossen haben. In diesem Fall muss diese Teilverarbeitung bestmöglich wieder rückgängig gemacht werden. Die Fehlerbehandlung in WS-BPEL sieht deshalb hierfür das Konzept der Kompensation vor [6]. Dies beinhaltet applikationsspezifische Aktivitäten, die versuchen Effekte umzukehren, die eine Aktivität, die Teil einer größeren Arbeitseinheit war, die abgebrochen wurde, bereits vorgenommen hat. Dieses Konzept findet seinen Ursprung in *Sagas* und *Open Nested Transactions* [5].

WS-BPEL liefert eine Variante eines solchen Kompensationsmechanismus, indem es die Möglichkeit zur flexiblen Kontrolle von Stornierungen von Aktivitäten gibt. WS-BPEL bietet durch die Möglichkeit FH und CH zu definieren ein Mittel um solche LRTs zu unterstützen.

Der Begriff LRT, der hier verwendet wird bezieht sich allerdings auf eine lokale, nur einen einzelnen Prozess betreffende Instanz. Wenn Koordination zwischen mehreren Teilnehmern mit verteilten Prozessen und Web Services notwendig ist, handelt es sich um ein orthogonales Problem, das außerhalb der Spezifikation von WS-BPEL liegt [6]. Hier könnten die Spezifikationen WS-C mit WS-BA hilfreich sein (vergleiche Abschnitt 3.1.3 WS-Transaction). Eine bildhafte Darstellung des Zusammenhangs von WS-BPEL, Web Services und WS-Transaction kann der folgenden Abbildung entnommen werden.

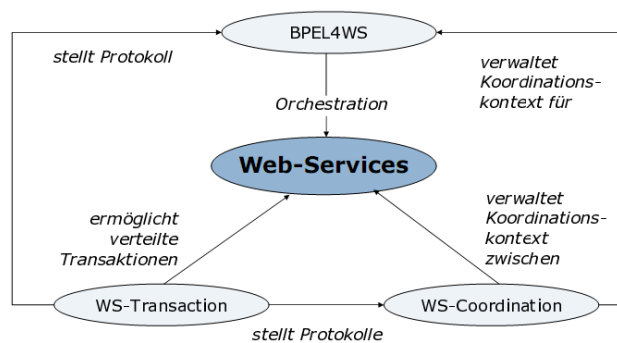


Abbildung 6: WS-BPEL und WS-Transaction (Quelle [7])

Kompensation muss nicht immer in der umgekehrten Reihenfolge der Transaktionen vorgenommen werden. Es gibt auch Situationen, die es verlangen, dass Kompensation in der ursprünglichen Reihenfolge vorgenommen werden muss. WS-BPEL begegnet dieser Anforderung mit einem flexiblen Kompensationsmechanismus, der durch die Ausführung von `<compensateScope target="NCName"/>` einen Scope mit dem Namen *NCName* kompensiert. *NCName* repräsentiert hierbei einen "non-colonized" XML-Namen, d.h. einen XML-Namen, der keinen Doppelpunkt enthält [8]. In diesem Fall ist der *NCName* der Name des Scopes, der kompensiert werden soll.

3.2.3 Compensation Handlers (CH)

Die Fähigkeit, Kompensationslogik Seite an Seite mit der vorwärtslaufenden Logik zu verknüpfen, ist die Untermauerung des applikationskontrollierten Fehlerbehandlungsgerüsts von WS-BPEL [6]. Durch Spezifizierung eines CH wird in WS-BPEL Scopes erlaubt, das Verhalten zu schildern, wie in Applikationen Effekte rückgängig gemacht werden können. Eine Ausnahme bildet die `<invoke>` Aktivität. Innerhalb dieser Aktivität kann direkt ein `<compensationHandler>` Konstrukt mit einer korrespondierenden Kompensationsaktivität angegeben werden, ohne explizit einen `<scope>` zu deklarieren.

Wenn kein CH für einen Scope definiert wurde, wird **Default Kompensation** geliefert (siehe weiter unten im Abschnitt). CH werden im Gegensatz zu FH, EH oder TH erst installiert, wenn der Scope **erfolgreich** abschließt. Der Zustand dieses Scopes wird zur Abschlusszeit beibehalten. Solche Scopes laufen zwar nicht mehr, sind durch ihre CH aber immer noch erreichbar, indem diese von den FH oder CH ihres Vaterscopes aufgerufen werden. Ist der Vaterscope eines solchen, erfolgreich abgeschlossenen und somit kompensierbaren Scopes der Prozess selbst, so kann der CH des Scopes nur vom FH des Prozesses aufgerufen werden, da der Prozess selbst nicht mit einem CH verknüpft werden kann.

Der gesicherte Zustand eines erfolgreich abgeschlossenen, unkompensierten Scopes ist der **Scope-Snapshot**.

CH werden ausgelöst durch die *compensation activities*. Diese sind `<compensateScope>` oder `<compensate>`. Zu beachten gilt es:

- Ein CH ist nur erreichbar, wenn der gesamte Scope, mit dem der CH verbunden ist, erfolgreich abschließt.
- Erneutes Ausführen eines CH oder der Versuch einen Scope mit einem nicht installierten CH zu kompensieren, resultiert in einer `<empty>` Aktivität. Hierdurch sind die Handlers nicht auf den Zustand der Scopes angewiesen.
- Ein CH kann wiederum einen eigenen Scope definieren. Diesem Scope kann ein FH aber kein CH angefügt werden, da dieser zu keiner Zeit erreichbar wäre. Der FH des Scopes kann genutzt werden, um einen weiteren Kindscope des Scopes dieses CHs zu kompensieren. Ein FH oder CH kann diesem Kindscope also angefügt werden.
- Die *compensation activities* dürfen nur in CH, TH oder in einem `<catch>` oder `<catchAll>` Konstrukt eines FH ausgeführt werden.
- `<compensate/>` kompensiert alle eingekapselten Scopes in umgekehrter Reihenfolge der ursprünglichen Ausführung (siehe Default Kompensation).
- `<compensateScope target="ScopeName"/>` kompensiert gezielt den Kindscope, dessen Name mit dem des target Attributs übereinstimmt. Hierdurch können Reihenfolge und Notwendigkeit der zu kompensierenden Scopes selbst bestimmt werden.

Default Kompensation

Wird vom Prozessdesigner explizit kein CH definiert, so liefert WS-BPEL Default Kompensation (Default Compensation). Ein Default CH sieht folgendermaßen aus:

- ```
<compensationHandler>
 <compensate/>
</compensationHandler>
```

### 3.2.4 Fault Handlers (FH)

Fehlerbehandlung in Geschäftsprozessen kann als ein Modus-Wechsel vom normalen Verhalten eines Scopes in die Fehlerbehandlung des Scopes angesehen werden. Die Fehlerbehandlung eines Scopes wird primär genutzt, um einen Fehler abzufangen und alternative Pfade außerhalb des Scopes zu gehen. Hierbei sind ausgehende Links des FH erlaubt. Sie dürfen jedoch nicht zurück in den fehlgeschlagenen Scope leiten [9]. Ein Scope, der einen assoziierten FH aufgerufen hat, wird als fehlgeschlagen behandelt, auch wenn der FH erfolgreich abschließt. Kompensation für diesen Scope ist nicht möglich, da der CH für einen Scope, der einen FH aufruft, nicht installiert wird.

FH können auch genutzt werden, um Kompensation in den Kindscores des Scopes, mit dem er assoziiert ist, auszulösen. Typischerweise versucht ein FH aber erst einmal, die Fehlersituation durch alternative Pfade abzufangen. Folgendes gilt es zu beachten:

- Es muss entweder ein `<catch>` oder ein `<catchAll>` Element in FH geben. Diese werden von Aktivitäten gefolgt.
- Nur FH können einen Fehler erneut werfen. Er wird dann an den FH des Vaterscopes weitergegeben. Dies geschieht explizit mit dem Element `<rethrow>`. Es werden immer die originalen Fehlerdaten geworfen und der Fehlertyp wird gesichert. Dies kann auch vorgenommen werden, wenn der FH den Fehler abfangen konnte.
- Ein FH für Scope S terminiert alle Aktivitäten in S und alle Aktivitäten der in S eingekapselten Kindscores. Diese Terminierung erfolgt **vor** dem spezifischen Verhalten des FH.
- Ein CH wird für einen Scope, der einen FH aufgerufen hat, **nie** installiert.
- Wird ein FH nicht explizit angegeben, so liefert WS-BPEL **Default Fehlerbehandlung** (siehe weiter unten im Abschnitt).
- Kann ein Fehler von einem FH nicht abgefangen werden, so wird er an den FH des Vaterscopes weitergegeben. Auch hier stoppt nun die normale Bearbeitung, d.h. auch dieser Scope wird nun als fehlgeschlagen betrachtet. Handelt es sich beim Vaterscope um den Prozess selbst und kann der FH der Prozessebene den Fehler nicht abfangen, so wird der gesamte Prozess terminiert.
- Nutzt eine FH für seine Aktivität einen eigenen Scope, so kann diesem Scope ein weiterer FH angefügt werden. Ein CH für den Scope des FH ist jedoch **nicht** anwendbar, da dieser CH zu keiner Zeit erreichbar wäre, da CH für fehlgeschlagene Scopes nicht installiert werden. Wird jedoch ein weiterer Kindscope eingeschachtelt, so kann in diesem wieder jeweils ein FH und ein CH angefügt werden.

Wie auch bei CH ist bei FH innerhalb eines `<invoke>` Elements die Definition von FH ohne explizites `<scope>` Element möglich. Fehlernamen können innerhalb eines `<catch>` Elements abgefangen werden und als Routing Entscheidungen benutzt werden.

Ein Fehler, der in einer Aktivität auftritt und in derselben Aktivität als Routing Entscheidung für einen alternativen Pfad genutzt wird, nennt man „*caught exception*“. Die folgende Abbildung verdeutlicht diesen Mechanismus. Fehler, die nicht in dieser Art und Weise aufgefangen werden können, werden an den umschließenden Kontext weitergegeben (siehe oben).

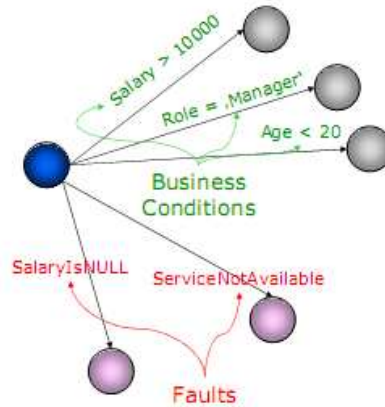


Abbildung 7: "caught exceptions" - Fehlernamen für Routing Entscheidungen (Quelle [9])

### Default Fehlerbehandlung

Wird vom Prozessdesigner explizit kein FH definiert, so liefert WS-BPEL Default Fehlerbehandlung (Default FH). Ein Default FH sieht folgendermaßen aus:

- `<catchAll>`
  - `<sequence>`
    - `<compensate/>`
    - `<rethrow/>`
  - `</sequence>`
- `</catchAll>`

### 3.3 Workflows und Transaktionen

Ein Workflow Management Systemen (WfMS) [5] koordiniert die Ausführung der einzelnen Aktivitäten, die einen Geschäftsprozess ausmachen. Das Resultat ist, dass die Aktivitäten eines gegebenen Workflows ein gemeinsames Schicksal haben, sie repräsentieren eine Arbeitseinheit. Der Ausfall einer Aktivität kann andere Aktivitäten beeinflussen, die einzelnen Aktivitäten sind nicht mehr unabhängig voneinander. Das Ergebnis dieser Aktivitäten beeinflusst also den gesamten Workflow. Um dem zu begegnen, gibt es in WfMS zwei grundlegende Konzepte.

- Das WfMS unterstützt eine alles-oder-nichts Semantik innerhalb der Workflows. Dies bedeutet, dass Kombinationen von Aktivitäten, jede einzelne davon als Transaktion implementiert, innerhalb des Workflows als eine einzelne Transaktion auftreten sollten. Das WfMS verbindet diese Transaktionen in eine einzelne, verteilte Transaktion. Dies ist das Konzept der *Atomic Spheres* [5]. Es wird in **Abschnitt 3.3.1** erläutert.

- Das WfMS unterstützt ein geschäftsorientiertes Arbeitseinheitsmodell. Dies bedeutet, dass vom WfMS Mechanismen unterstützt werden, die bereits getätigte Arbeitseinheiten rückgängig machen können. Diese müssen sowohl nicht-transaktionale Implementierungen von Aktivitäten als auch möglicherweise langandauernde Workflows und Aktivitäten widerspiegeln. Dies ist das Konzept der *Compensation Spheres* [5]. Es wird in **Abschnitt 3.3.2** erläutert.
- Für langlaufende Prozesse die fehlschlagen, unterstützt das WfMS ein Konzept, das es erlaubt, den Kontext der Prozesse in einer Datenbank zu persistieren. Somit kann *Phoenix Verhalten* [5] des WfMS erreicht werden. Es wird in **Abschnitt 3.3.3** erläutert.

### 3.3.1 Atomic Spheres

Modularisierung fördert die Wiederverwendung von Applikationsfunktionen. Ist eine Applikationsfunktion eine Transaktion, so müssen die Transaktionsgrenzen mit Vorsicht behandelt werden [5].

Atomic Spheres (AS) [5] erlauben es dem Prozessmodellierer, Aktivitäten mit transaktionalen Implementierungen in einer neuen Arbeitseinheit, einer globalen Transaktion, zu gruppieren. Die Transaktionen in AS nehmen an dieser globalen Transaktion teil, sie sind transaktionale Funktionen, Komponenten oder Programme. Sie grenzen ihre eigenen Transaktionen nicht ab und geben nicht selbstständig ein *commit* oder *abort* Statement ab. Wäre dies nicht der Fall, so wäre eine Wiederverwendung dieser Komponenten stark eingeschränkt.

Es werden Mechanismen benötigt, die Transaktionsgrenzen von transaktionalen Funktionen (TF) von außen anzulegen. Die TF werden dann zu Subtransaktionen der globalen Transaktion. Ein Transaktionsmanager koordiniert die Statements der einzelnen TF und entscheidet über das Ergebnis der globalen Transaktion. Das WfMS repräsentiert die globale Transaktion für den Transaktionsmanager (TM).

Das WfMS liefert hierbei die Fähigkeit, die globale Transaktion abzugrenzen, die Subtransaktionen aufzurufen und auf Ausnahmesituationen, wie ein negatives Acknowledgement von einer durch den TM initiierten *commit* Anfrage, zu reagieren.

#### Der Mechanismus von AS

Die AS ist eine Sammlung von Aktivitäten, die die folgenden drei Bedingungen erfüllen muss:

1. Alle Aktivitäten innerhalb der AS haben transaktionale Implementierungen und müssen Transaktionsgrenzen, die von außen angelegt sind, akzeptieren.
  2. Entweder haben alle Aktivitäten der AS, die einen Vorgänger außerhalb der AS haben denselben oder keinen.
  3. Entweder alle Aktivitäten der AS schließen mit einem *commit* ab oder alle mit einem *abort*.
- Die Bedingung 1 sichert die Implementierung mit globalen Transaktionstechniken.

- Die Bedingung 2 versichert, dass die AS schnell läuft und die Ressourcen Manager der beteiligten Transaktionen, die die Sperrprotokolle für die Zeit der globalen Transaktion aufrecht erhalten müssen, diese nicht zu lange blockieren, um den Durchlauf der Umgebung nicht negativ zu beeinflussen.
- Die Bedingung 3 sichert die alles-oder-nichts Semantik. Ein 2PC Transaktionsprotokoll versichert ein uniformes Gesamtergebnis der AS.

Bei einem *abort* kann die AS bis zu einem vorher definierten maximalen Schwellenwert wiederversucht werden. Das Konzept der AS spiegelt das Konzept von WS-C mit dem Protokoll WS-AT wieder (vergleiche Kapitel 2.6.2).

### 3.3.2 Compensation Spheres

Das Konzept der Compensation Spheres (CS) [5] nimmt sich den Situationen an, in denen es sich um langandauernde Transaktionen (LRT) in Umgebungen mit hohem Durchlauf handelt. Hier stellt das 2PC Protokoll einen Overhead dar und die Sperrmechanismen limitieren den Durchlauf des Systems erheblich. Wenn der Fall, dass eine AS fehlschlagen kann, in der Praxis nur selten vorkommt, so kann ein kompensationsbasiertes Wiederherstellen des konsistenten Systemzustands toleriert werden.

Ein weiterer Punkt ist, dass es erlaubt sein sollte, nicht nur Aktivitäten mit transaktionsbasierten Implementierungen in Arbeitseinheiten zu gruppieren, sondern auch solche, deren Implementierung nicht auf transaktionaler Basis begründet ist, oder eine beliebige Mischung von beidem. Ein angemessenes Arbeitseinheitskonzept muss in diesem Fall kompensationsbasiert sein.

Oft wird ein unkorrekt ausgeführtes Arbeitsstück erst lange nachdem die korrespondierende Aktivität beendet wurde entdeckt. Auch wenn eine solche Aktivität innerhalb der AS durchgeführt wurde, kann nur noch Kompensation die resultierende Fehlersituation wiedergutmachen. Die Komplettierung einer Aktivität bedeutet nicht automatisch, dass diese Aktivität auch korrekt durchgeführt wurde. Zum Beispiel könnte ein falscher Preis, oder ein falsches Lieferdatum dazu führen, dass eine erfolgreich abgeschlossene Aktivität kompensiert werden muss. Auch bei syntaktischem Erfolg muss es also ein Mittel geben, um semantische Fehler anzuzeigen [5].

Unkorrekt durchgeführte Arbeit muss repariert oder korrigiert werden. Dies ist oft kein simples Problem. Man stelle sich z.B. einen bereits versendeten Brief oder ein fehlerhaft zurechtgeschnittenes Bauteil vor. Eine Aktivität kann auch fehlschlagen, nachdem die implementierten Aktivitäten korrekt und erfolgreich abgeschlossen haben und der Kontrollfluss bereits weitergegangen ist. Als Beispiel hierfür könnte ein Produktionsproblem oder ein Brand im Lager der Firma, der wesentliche Teile der Ware vernichtet hat, genannt werden.

## Das Konzept der CS

Die CS ist eine beliebig zusammengestellte Sammlung von Aktivitäten mit transaktionalen und nicht-transaktionalen Implementierungen. Arbeit, die in einer CS durchgeführt wird, wird durch Kompensation repariert. Zu diesem Zweck kann jede Aktivität in der CS definiert werden als ein Paar von der eigentlichen Aktivität und der dazugehörenden Kompensationsaktivität. Darüber hinaus kann die CS selbst mit einer Kompensationsaktivität verbunden werden. Diese Kompensationsaktivität kann hierbei durch ein Programm, eine gewöhnliche Aktivität oder auch ein Prozessmodell repräsentiert werden [5].

Im Fehlerfall bricht das WfMS die CS ab (*abort*). Im Anschluss daran ermittelt das WfMS alle Aktivitäten der CS, die bereits durchgeführt wurden und erstellt einen Kompensationsgraphen. Dieser Kompensationsgraph kehrt den Prozessfluss um und ignoriert dabei alle Übergangsbedingungen. Er bildet die umgekehrte Reihenfolge der ursprünglichen Ausführung der Aktivitäten ab. Durch Instanzieren des Kompensationsgraphen werden die Kompensationsaktivitäten ausgeführt. Wenn das WfMS keine Kompensation unterstützt, so muss dieser Kompensationsgraph explizit modelliert werden.

Es gibt zwei verschiedenen Granularitäten von Kompensation [5]:

- **Diskrete Kompensation** – Jede Aktivität wird separat kompensiert.
- **Globale Kompensation** – kommt zur Anwendung, wenn ein simples Umkehren des Prozessflusses und die Ausführung der Kompensationsaktionen nicht ausreichen. An diese Stelle tritt dann z.B. ein Prozessmodell, das eine Kompensationsaktivität darstellt. Als Beispiel hierfür könnte ein Prozessmodell dienen, das die Probleme behandelt, wenn Ware bereits ausgeliefert wurde, sich der Kunde aber im nach hinein als insolvent herausgestellt hat.

Die CS ist also eine nicht leere Menge von Aktivitäten, von der verlangt werden kann, dass sie ihre Arbeit in umgekehrter Reihenfolge wieder rückgängig macht, indem man ihre Kompensationsaktivitäten aktiviert. Ist dies in speziellen Fällen nicht ausreichend, so kann die Kompensationsaktivität, die mit der CS selbst verbunden ist, aktiviert werden.

### 3.3.3 Phönix Verhalten

Unter Phönix Verhalten wird die Fähigkeit eines Systems verstanden, sich nach einem Totalausfall selbstständig wieder herzustellen und zu erholen. Der Name resultiert aus der ägyptischen und griechischen Mythologie, in der Phönix, ein mythischer Vogel, verbrennt, um aus seiner Asche neu zu entstehen. Phönix Verhalten steht also sinnbildlich dafür, dass sich das WfMS aus den resultierenden Fehlersituationen eines Totalausfalls wieder selbst herstellen kann. Dies ist vor allem bei langdauernden Workflows sehr sinnvoll, da sonst zunächst alle bisher ausgeführten Arbeitseinheiten kompensiert werden müssten und dies zur Folge hätte, dass manche Arbeiten unnötig mehrfach ausgeführt werden müssten. Dies wäre in erster Linie ein Zeit- und Kostenproblem.

Phönix Verhalten eines Workflows wird erreicht durch Persistieren dessen Kontexts in der Kontext Datenbank des WfMS [5]. Die Kontext Datenbank wird transaktional aktualisiert und diese Transaktionen müssen die entsprechenden Events und ihre zugehörigen Kontextaktualisierungen

innerhalb derselben Transaktionsgrenzen vornehmen. Diese Technik stellt sicher, dass jedes für das WfMS relevante Ereignis letztlich auch entsprechend in der Datenbank reflektiert wird.

Der Kontext repräsentiert die relevanten Daten des Workflows. Diese sind der Zustand einer jeden Aktivität (*completed, active, waiting for execution*) und die *Output Container* [5] der abgeschlossenen Aktivitäten. Diese dienen als *Input Container* [5] für zukünftig auszuführende Aktivitäten.

Im Falle eines Totalausfalls ist also der Kontext von jedem Workflow nach dem Neustart des WfMS in seiner Datenbank vorhanden. Genauer gesagt: keine der bereits komplettierten Aktivitäten muss erneut durchgeführt werden und keine der Aktivitäten, die noch auf Ausführung gewartet haben sind verloren. Für Aktivitäten, die während des Ausfalls aktiv waren, muss eine Fallunterscheidung vorgenommen werden [5].

1. Die Implementierung der Aktivität ist von dem Ausfall auch betroffen.

- Wenn die Aktivität keine Transaktion war, müssen die Effekte, die sie eventuell verursacht hat, untersucht werden. Aufgrund dieser Analyse muss dann entschieden werden, ob die Aktivität kompensiert und neu ausgeführt werden muss, wiederaufgenommen werden kann oder andere Aktionen unternommen werden müssen.
- Wenn die Aktivität eine Transaktion war, wird sie durch den Neustart ihres zu Grunde liegenden Resource Managers automatisch abgebrochen und der letzte konsistente Zustand der Ressource wird wieder hergestellt. Diese Aktion kann also einfach neu ausgeführt werden.

2. Die Implementierung der Aktivität ist von dem Ausfall nicht betroffen.

- Wenn die Aktivität keine Transaktion war muss wie oben vorgegangen werden
- Wenn die Aktivität eine Transaktion war, so muss festgestellt werden, ob sie mit einem *commit* oder einem *abort* abgeschlossen hat, während die ausführende Komponente ausgefallen war. Wenn sie abgebrochen hat, so kann sie erneut ausgeführt werden. Hat sie erfolgreich abgeschlossen, muss nichts weiter unternommen werden. Ist sie immer noch aktiv, so sollte sie abgebrochen werden, da das Umfeld fragwürdig ist. Danach kann sie erneut ausgeführt werden.

Das WfMS muss also in jedem Fall aktiv in den Prozess eingreifen, um Phönix Verhalten zu ermöglichen. Es muss die Änderung eines Zustands einer Aktivität erzwingen können, da sonst Kompensation, Neustart oder Wiederaufnahme der selbigen nicht möglich wäre. Des Weiteren muss das WfMS fähig sein, nach einem Neustart eventuell unbekannte Zustände einer Aktivität in Erfahrung zu bringen, um den Prozessfluss angemessen fortzuführen.

## 4 Fehleranalyse und Strategien zur Fehlerbehandlung

In diesem Kapitel erfolgt eine Analyse der Fehlermöglichkeiten und Fehlersituationen, die in einer PaF-Architektur auftreten können. Nach dieser Analyse werden verschiedene Fehlerbehandlungsstrategien aufgeführt und ihre Umsetzungsmöglichkeiten in einer, auf EIP und PaF-Architektur begründeten Integrationslösung geprüft und diskutiert. Eine Abbildung der EIP auf BPEL und die damit verbundene Möglichkeit, ein globales Fehlerbehandlungskonzept umzusetzen, wird an dieser Stelle noch nicht berücksichtigt. Es werden hier explizit Möglichkeiten geprüft, Fehlersituationen mit den Mitteln von EIP und der PaF-Architektur umzusetzen.

Hierzu werden zuerst die möglichen Auswirkungen der Fehler in unkritische und kritische Fehlersituationen eingeteilt (**Abschnitt 4.1**), um zwischen den Möglichkeiten des **Forward Recovery** oder einem zwingend notwendigen **Backward Recovery** zu differenzieren. In den darauf folgenden Abschnitten beschäftigt sich dieses Kapitel mit den Fehlermöglichkeiten und den daraus resultierenden Fehlersituationen, die in der PaF-Architektur durch den Ausfall eines Filters (**Abschnitt 4.2**), eine ungültige Message (**Abschnitt 4.3**) oder einen Übertragungsfehler im Kanal (**Abschnitt 4.4**) entstehen können. Hier werden auch Prozesse betrachtet, bei denen einzelne Filter atomare Transaktionen vornehmen, die Fehler die hierbei entstehen können und die Strategien diskutiert, wie bei diesen Prozessen vorgegangen werden muss.

Die verschiedenen Strategien, mit denen man Fehlernbehandlung durchführen kann, werden in **Abschnitt 4.5** dieses Kapitels zusammengetragen.

Fehlerbehandlung in einer PaF-Architektur ist sehr mühselig und schwierig. Durch die Entkopplung der Filter und das Fehlen der Zuordnung eines Prozesses zu einer gewissen Instanz gibt es kein globales Wissen über den Zustand der Pipes and Filters Kette. Eine Fehlerbehandlung ist deshalb nur sehr schwer umzusetzen. Es muss ein Mittel geben, um Fehler anzuzeigen, um diesen mit geeigneten Maßnahmen zu begegnen. Vor allem die einzelnen, atomaren Transaktionen, die im Prozessverlauf bereits ausgeführt worden sind, müssen bei einem globalen Scheitern des Prozesses identifiziert und geeignete Maßnahmen müssen getroffen werden, um den Prozess zu reparieren (Forward Recovery) oder das System, das durch einen irreparablen Prozess, der auf Grund eines nicht abgefangenen Fehlers nur teilweise ausgeführt werden konnte, wieder in einen konsistenten Zustand zu überführen (Backward Recovery).

Wenn es sich bei einer PaF Kette um einen Prozess handelt, der nur Filter beinhaltet, die transaktionale Implementierungen (*Transactional Client* [3]) haben, so handelt es sich um eine Situation, bei der in vergleichbaren Architekturen und anderen Integrationskonzepten (Workflow, SOA) koordinierte Transaktionsprotokolle (2PC) zum Einsatz kommen, um Dienstgüteeigenschaften zu gewährleisten. Ob dies auch in einer PaF-Architektur umgesetzt werden kann wird in **Abschnitt 4.5.1** diskutiert.

Wenn es sich bei einer PaF Kette um einen Prozess handelt, der zum Teil aus Filtern mit transaktionalen und nicht-transaktionalen Implementierungen besteht, muss hier ein kompensationsbasiertes Transaktionskonzept (Backward Recovery) zum Einsatz kommen (vergleiche 3.3.2 Compensation Spheres). Bei solchen vermischten Prozessen, bei denen auch Aktivitäten

denkbar sind, die von Hand ausgeführt werden müssen, ist die Gesamtdauer des Prozesses vorher oft nicht abzusehen, sie werden als langlaufende Prozesse definiert. Bei langlaufenden Prozessen muss Fehlersituationen mit Kompensation entgegen getreten werden, für eine unabsehbar lange Zeit können keine Sperrprotokolle über die beteiligten Ressourcen aufrecht erhalten werden. Kompensationsaktionen müssen bereits beim Design der PaF Kette bedacht werden und Teil des modellierten Prozesses sein. Dieser Aspekt wird im **Abschnitt 4.5.2** genauer erörtert.

Anders als bei einem WfMS, dass die Kontrolle über die CS übernimmt und beim Scheitern der CS ermittelt, welche Aktivitäten erfolgreich abgeschlossen haben (und somit kompensiert werden müssen) und welche Aktivitäten noch nicht ausgeführt wurden (vergleiche 3.3.2), fehlt bei der PaF-Architektur dieses globale Kontrollorgan jedoch, was eine Kompensation der CS sehr schwierig gestaltet. Eine Umsetzung einer CS mit EIP wird in **Abschnitt 4.5.3** geprüft.

In den **Abschnitten 4.5.4, 4.5.5, 4.5.6** und **4.5.7** werden die Fehlerbehandlungsstrategien Retry und Retry All, Correction Loop, Fehler ignorieren und das Konzept des Phönix Verhaltens in einer PaF-Architektur diskutiert.

Den Abschluss dieses Kapitels bildet in **Abschnitt 4.6** ein Fazit dieser Fehlerbehandlungsanalyse.

## 4.1 Fehlersituationen

Die Aufgaben, die ein auf PaF und EIP basierendes Integrationsnetz erledigt, können sehr unterschiedlich und vielseitig sein. Nicht immer müssen sie direkten Einfluss auf andere Komponenten haben oder Veränderungen des Zustands des Systems herbeiführen. Manche Aufgaben sind auch eher zeitunkritisch, wie zum Beispiel die Anfrage nach dem Bestellstatus oder eine Versandbestätigung eines bestellten Artikels an einen Kunden, der keinen bestimmten Status hat. Hier ist eine komplexe und dadurch oft sehr teure Fehlerbehandlung nicht immer zwingend notwendig. Oft stehen auch die Kosten in keinem Verhältnis zu dem Nutzen, den die Fehlerbehandlung bringt [14] (Vergleiche **Abschnitt 4.5.6**). Meist lässt sich einem Fehler dieser Kategorie mit einer Neu- oder Wiederausführung des fehlgeschlagenen Arbeitsschrittes oder Prozesses begegnen. Unkritische Fehlersituationen werden in **Abschnitt 4.1.1** näher betrachtet

Weitaus komplexer sind die Fehlersituationen, die entstehen können, wenn in der Prozesskette von einer auf PaF und EIP basierenden Integrationslösung einzelne Arbeitsschritte Transaktionen vornehmen. Fehlersituationen die sich hieraus ergeben können, werden als kritisch betrachtet, da sie Einfluss auf das gesamte System haben. Als Beispiel kann hier die Abbuchung und Überweisung eines Betrages von einem Konto auf ein anderes genannt werden. Wird im Verlauf der Integrationslösung, die sich dieser Aufgabe annimmt, nun einem Kunden ein gewisser Betrag vom Konto abgebucht, um diesen Betrag auf dem Konto des Verkäufers gut zu schreiben. Schlägt aber die Umbuchung fehl, da der Kontierungsservice des Verkäufers wegen einer falschen Struktur oder fehlenden Information in der Nachricht fehlschlägt oder nicht erreichbar ist, so ist ein Forward-Recovery der fehlgeschlagenen Prozesskette nicht sinnvoll, es hätte unter Umständen sogar weitreichende Konsequenzen für das System.

Ein simpler Neustart dieser Prozesskette hätte hier eine doppelte Abbuchung des Betrags beim Kunden zur Folge und für den Fall, dass die Nachricht für den Kontierungsservice falsch strukturiert ist, wäre eine wiederholte Kontaktaufnahme mit diesem somit erneut zum Scheitern verurteilt.

Selbst wenn der Fehler nicht bei der Nachricht selbst liegt, sondern auf einem netzwerkbedingten Ausfall des Kontierungsservice begründet ist, ist ein periodisches Wiederholen der Kontaktaufnahme nicht immer sinnvoll, da es sich eventuell um einen zeitkritischen Vorgang handelt und nicht bekannt ist, wann der Netzwerkfehler wieder behoben sein wird. Ist das Geld beim Kunden abgebucht, erwartet dieser auch eine Leistung dafür. Diese wird aber erst erbracht, wenn der Dienstleister einen Zahlungseingang verbucht hat. Also muss es für solche Situationen einen Backward Recovery Mechanismus geben, mit dem die Effekte wieder rückgängig gemacht werden können, die Teil eines nur teilweise ausgeführten Prozesses sind, der irgendwann in seinem weiteren Verlauf fehlgeschlagen ist. Kritische Fehlersituationen werden in **Abschnitt 4.1.2** behandelt.

#### 4.1.1 Unkritische Fehlersituationen

Zu den fehlerunkritischen Aufgaben, die in einem Integrationsnetz ausgeführt werden, gehören auch PaF Prozessketten, in denen keine geschäftskritischen Aktionen vorgenommen werden, die Auswirkungen auf dahinterliegende Systeme haben. Hier kann das System durch einen möglichen Fehler nicht in einen inkonsistenten Gesamtzustand geraten. Tritt ein Fehler in dieser genannten Situation auf, so ist meist eine Neuausführung des fehlgeschlagenen Prozesses möglich, da eventuell mehrfach getätigte Verarbeitungsschritte in der Prozesskette der Integrationslösung keine kritischen Auswirkungen hätten. Als Beispiel hierfür kann eine doppelt versendete Versandbestätigung an den Kunden genannt werden. Gestaltet man die Filter jedoch als *Idempotent Receiver* [3], so könnte auch dieser Situation entgegen gewirkt werden.

Eine weitere Möglichkeit, einem Fehler, der in diese Kategorie fällt, zu begegnen, ist statt einer Neuausführung die Kontaktaufnahme mit der ausgefallenen Komponente so oft zu wiederholen, bis sie erfolgreich ist (vergleiche **Abschnitt 4.5.4**). Dies ist in zeitunkritischen Situationen ein einfaches Konzept, das zumindest die Zuverlässigkeit der Kommunikation gewährleistet. Erreicht werden kann dies auch zum Beispiel mit dem Store-and-Forward Mechanismus des unterliegenden Messaging Systems. Das Pattern *Guaranteed Delivery* [3] beschreibt diesen Mechanismus für eine PaF-Architektur. Allerdings wird hier nur eine Auslieferung der Nachricht an sich garantiert. Ist der Empfänger nicht bereit dafür, dann wird sie an den *Dead Letter Channel* [3] ausgeliefert. Hier endet nun die Weiterverarbeitung der Nachricht. Ein Mittel, dass dem entgegengewirkt, wird in **Abschnitt 4.5.4** beschrieben.

In WfMS gibt es das Konzept des Phönix Verhaltens (vergleiche Abschnitt 3.3.3). Eine Möglichkeit, dies in der PaF-Architektur umzusetzen wird in **Abschnitt 4.5.7** aufgezeigt.

Fehlersituationen, die aus oben genannten Gründen unkritisch sind, können mit Forward Recovery Mechanismen aufgefangen werden. Eine genauere Betrachtung der Fehlerbehandlungsstrategien erfolgt im **Abschnitt 4.5** dieses Kapitels.

## 4.1.2 Kritische Fehlersituationen

Es gibt zwei grundlegende transaktionale Konzepte, um einer kritischen Fehlersituation entgegen zu wirken:

- Das eine basiert auf einer koordinierten Konversation mit einem Transaktionsmanager, der ein uniformes Ergebnis der beteiligten Transaktionen garantiert (vergleiche WS-AT, Abschnitt 3.1.3 und Atomic Spheres, Abschnitt 3.3.1). Hierzu ist es jedoch notwendig, dass die beteiligten Transaktionen nicht selbstständig ein *commit* oder *abort* durchführen. Die beteiligten Transaktionen müssen von außen angelegte Transaktionsgrenzen akzeptieren. Ob dieses Konzepts mit EIP und PaF umsetzbar ist, wird ausführlich in **Abschnitt 4.5.1** diskutiert.
- Das andere Transaktionskonzept ist kompensationsbasiert. Es erscheint für den Einsatz in einer PaF-Architektur als sinnvoller, da es nicht voraussetzt, dass in der Gruppe nur transaktionale Aktivitäten durchgeführt werden. Dieses Konzept (vergleiche WS-BA, Abschnitt 3.1.3 und Compensation Spheres, Abschnitt 3.3.2) ist besonders für langlaufende Prozesse (LRTs) geeignet, um dem Overhead und der Sperrung der beteiligten Ressourcen, auf die die Transaktionen zugreifen, entgegen zu steuern.

Ein weiterer Grund, der für dieses Konzept spricht ist, dass Kompensation in einer Produktionsumgebung in jedem Fall unterstützt werden muss. Auch eine koordinierte, erfolgreich abgeschlossene Transaktion kann aus verschiedenen Gründen letztlich scheitern. In diesem Fall kann nur Kompensation aus der resultierenden Fehlersituation wieder einen konsistenten Systemzustand herstellen. Eine Umsetzung eines kompensationsbasierten Transaktionskonzepts wird in **Abschnitt 4.5.2** diskutiert.

Im Abschnitt 4.2 werden nun die möglichen Fehlerfälle beim Filter selbst näher betrachtet. Im Anschluss daran wird in Abschnitt 4.3 der Fehlerfall, der durch die Nachricht selbst erzeugt werden kann, beleuchtet und im Abschnitt 4.4 werden die Auswirkungen von Übertragungsfehlern auf dem Kanal aufgezeigt.

## 4.2 Fehler beim Filter

Der Filter stürzt ab. Dies kann theoretisch jederzeit im Nachrichtenfluss geschehen und hat je nach Zeitpunkt des Ausfalls zwei verschiedene Wirkungen:

### 4.2.1 Fall 1

Der weiterverarbeitende Filter ist bereits abgestürzt, als der vorangehende Filter der PaF Kette erfolgreich abschließt und versucht die Nachricht über seinen ausgehenden Kanal an den abgestürzten Filter weiterzugeben. Dies führt dazu, dass der Filter nicht erreicht werden kann und in

diesem Fall wird von dem Kanal die Nachricht an den *Dead Letter Channel* [3](DLC) weitergegeben. Dies geschieht natürlich nur dann, wenn der Kanal auch mit einem DLC verbunden ist.

#### **Problem:**

Die eventuell bereits teilverarbeitete Nachricht ist nun vom weiterführenden Verarbeitungsprozess ausgeschlossen. Dies kann gravierende Folgen haben. In jedem Falle kommt die Nachricht nicht bei ihrem ultimativen Empfänger an und das System befindet sich im schlimmsten Fall nun in einem inkonsistenten Zustand. Es findet keine automatische Benachrichtigung durch das System statt, der Fehler wird erst erkannt, wenn die Nachricht auf dem DLC vom Administrator des Systems entdeckt und analysiert wird. Dies führt in vielen Fällen zu einem unbefriedigenden Ergebnis, da der Zeitpunkt der weiteren Bearbeitung der Nachricht oder das Erkennen des Fehlerfalls für den Initiator des Prozesses vorerst im Unklaren bleibt.

#### **Idee:**

Wir brauchen ein Mittel, um eine automatische Weiterverarbeitung der Nachricht im Falle des oben genannten Fehlerfalles zu erreichen. Hierfür gibt es verschiedene Möglichkeiten:

- Dies kann z.B. dadurch erreicht werden, indem bei fehlerkritischen Situationen redundante Implementierungen ausfallgefährdeter Filter vorgesehen werden. So kann der Ausfall eines einzelnen Filters (und somit die Weiterleitung der Nachricht auf den DLC) abgefangen werden, indem an einem *Point-to-Point Channel* [3] mehrere *Competing Consumers* [3] lauschen.
- Eine weitere Möglichkeit wäre ein *Message Dispatcher* [3] oder ein spezieller Router, der an jedem seiner Ausgänge eine redundante Implementierung desselben Filters hat. Dieser Router wird in Abschnitt 4.5.4 vorgestellt. Somit könnte bei einem fehlgeschlagenen Kontaktversuch von diesem Router oder Message Dispatcher automatisch eine erreichbare, redundante Implementierung aufgerufen werden. Dies ist in einer PaF-Architektur jedoch problematisch, da die Definition vorsieht, dass nach jedem Filter ein Kanal kommen muss. Das Fehlschlagen des Kontakts wird also nicht vom Router oder vom Message Dispatcher selbst festgestellt, sondern vom Kanal. Da in einem Kanal für einen solchen Fall aber nicht eine Rücksendung der Nachricht an den Versender, sondern eine Weiterleitung an den DLC vorgesehen ist, müsste der DLC dahingegen verändert werden, dass er per Konfigurationseinstellung auch befähigt ist, eingehende Nachrichten wieder an einen ausgehenden Kanal weiter zu versenden. Dies wird ausführlich in **Abschnitt 4.5.4** diskutiert.
- In kritischen Fehlersituationen, in denen die oben genannten Mechanismen des Forward Recovery nicht erfolgreich ablaufen, muss Kompensation von bereits getätigten Arbeitsschritten erfolgen. Diese müsste hier dann direkt vom DLC angestoßen werden. Auch hier muss der DLC zuerst befähigt werden, eingehende Nachrichten an einen Ausgang weiterzuleiten, der der PaF Kette entspricht, die die Kompensationsaktionen durchführt. Diese müssen vorher modelliert werden. Ein genaueres Vorgehen, wie dies zu erreichen ist und was für Probleme sich hier stellen, wird in **Abschnitt 4.5.2** diskutiert.
- In zeitunkritischen Fehlersituationen könnte der DLC die Nachricht auch einfach immer wieder an den Kanal, der die Nachricht an den DLC weitergegeben hat, zurückgeben, bis die Übertragung letztendlich erfolgreich verläuft.

Wenn all diese Möglichkeiten nicht erfolgreich verlaufen, so muss zumindest der Systemadministrator über das Scheitern des Prozesses informiert werden. Der DLC könnte hierfür über seinen Ausgang eine Fehlerbehandlung anstoßen, die Daten aus der originalen Nachricht in eine Exception kopieren. Diese kann dann über den Control Bus an die Management Konsole versendet werden (siehe Abschnitt 2.4.1). Anbieten würde sich hier, falls vorhanden, die *Message History* [3], die *Return Address* [3] und zusätzliche Daten über den fehlerhaften Prozess wie seine ID und einen Timestamp. In der Management Konsole (siehe Abschnitt 2.4.2) kann diese Exception dann ausgewertet werden und der ursprüngliche Versender der Nachricht kann nun mittels der Return Address über das Fehlschlagen des Prozesses informiert werden. Mit den Daten der Message History könnte auch eine manuelle Kompensation der entsprechenden Komponenten durchgeführt werden.

#### 4.2.2 Fall 2

Die Nachricht wird an den weiterverarbeitenden Filter der Integrationslösung geleitet, erfolgreich entgegengenommen und der Filter stürzt ab.

##### **Problem:**

Selbst wenn die Nachricht mit *Guaranteed Delivery* [3] oder einem Store-and-Forward Mechanismus des Messaging Systems ausgeliefert wurde, ist sie nun verloren und die Weiterverarbeitung der Nachricht ist beendet. Es findet keine Benachrichtigung des Initiators über das Systemversagen statt und das System befindet sich im schlimmsten Fall in einem inkonsistenten Zustand. Die Nachricht ist in diesem Fall nun auch nicht mehr erhältlich, da sie sich zum Zeitpunkt des Systemabsturzes im flüchtigen Speicher der Filterkomponente befand und nicht mehr persistent war. Da die einzelnen Filter außerdem zustandslos sind, ist es sehr komplex, nach einem Systemausfall und dem Neustart der einzelnen Komponenten die Arbeit an teilverarbeiteten Messages wieder aufzunehmen.

##### **Idee:**

Um den oben genannten Problemen entgegen zu wirken muss ein Mittel geschaffen werden, um Statusinformationen in einem Prozess zu erhalten, ohne dabei das Prinzip der losen Kopplung zu verletzen. Dies kann durch das Persistieren der Nachricht erreicht werden.

Die Nachricht selbst ist eine diskrete Einheit von Daten. Sie enthält einen *Header*, in dem relevante Information für den Transport, zur Struktur und dem Inhalt der Nachricht enthalten sind. Diese Informationen werden vom Messaging System genutzt. Im *Body* der Nachricht befinden sich die zu übertragenden Daten an sich. Der Body wird im Allgemeinen vom Messaging System ignoriert [3].

Hierdurch ist gewährleistet, dass die Nachricht selbst alle für den Transport durch das System und die Verarbeitung durch die Filter relevanten und benötigten Informationen enthält. Sie trägt somit also implizit den Status des Prozesses in sich, da sie selbst das Produkt von vorher vorgenommenen Arbeitsschritten ist.

Da die Nachricht selbst den Status ihrer Verarbeitung trägt, sind die Filter davon befreit, sich diese Statusinformationen in irgendeiner Form zu merken. Diese können mit dem Pattern *Message History* [3] ergänzt werden. Bevor eine erfolgreich bearbeitete Nachricht an den nächsten Filter übergeben

wird, wird die Message History um den Eintrag des aktuellen Filters in der PaF Kette erweitert. Auf diese Weise kann später auch der Weg nachvollzogen werden, den die Nachricht durch das System gegangen ist.

Damit die Nachricht nun auch im oben beschriebenen Fall nicht verloren gehen kann, muss sie zwischen jedem Arbeitsschritt persistiert werden. Dies kann mit einem *Message Store* [3] erfolgen. Hierzu könnte ein *Wire Tap* [3] zwischen jedem Filter die Nachricht an den Message Store weiterleiten. Für die Zuordnung der Nachrichten im Message Store sollten diese mit einer über den Prozessfluss gleichbleibenden Correlation ID versehen sein. Ein *Timestamp*, vorgenommen durch den Wire Tap oder den Message Store, würde außerdem eine Möglichkeit bieten, die Verarbeitungszeiten der jeweiligen Filter abzuschätzen und die Nachricht zeitlich im PaF Fluss einzuordnen. Die Message History lässt eine örtliche Einordnung der Nachricht im PaF Fluss zu. Sollte nun der oben erwähnte Fall eintreten, dass der Filter nach Entgegennehmen der Nachricht abstürzt, so wäre zumindest die Nachricht und die Information über die Kette der Filter die ihre Teilverarbeitung vorgenommen haben nicht verloren. Da jedoch keine globale Sicht auf den Prozess vorhanden ist, gestaltet es sich trotzdem als schwierig, diese Nachricht wieder automatisiert ins System einzuspielen. Eine Möglichkeit, dies zu realisieren, könnte durch das Messaging System selbst gegeben werden. Dies wird in **Abschnitt 4.5.7** diskutiert.

Da es in einer PaF-Architektur durch ihre Struktur auch denkbar ist, dass eine Nachricht von einem Filter erfolgreich verarbeitet wurde und dann im Kanal zum Wire Tap (oder im Wire Tap selbst) verloren geht (also bevor sie durch den Message Store persistiert wurde, aber nachdem sie erfolgreich bearbeitet wurde), muss hier beachtet werden, dass der Fall eintreten kann, dass ein Wiedereinspielen der Nachricht ins System Duplikate dieser zur Folge haben kann. Es muss sich bei den Filtern also um *Idempotent Receiver* [3] handeln.

Handelt es sich um eine kritische Fehlersituation, so ist es notwendig, das System durch Kompensation der vorangegangenen Arbeitsschritte wieder in einen konsistenten Zustand zu bringen. In einem WfMS wird dem in diesem Abschnitt behandelten Fehlerfall mit dem Konzept des Phönix Verhaltens (vergleiche Abschnitt 3.3.3) entgegen getreten. Die oben beschriebene Möglichkeit, das Persistierens der Nachricht durch den Message Store entspricht in etwa diesem Vorgang. Der grundlegende Unterschied ist jedoch, dass das WfMS in der Lage ist, selbstständig den Zustand der Aktivitäten im Prozessfluss abzufragen und zu beurteilen und auf dieser Basis entscheiden kann, wie mit dem fehlgeschlagenen Prozess nun weiter umgegangen werden muss. Müssen einzelne Aktivitäten kompensiert werden oder können sie einfach neu ausgeführt werden, dass sind einige der Fragen, die sich hier stellen.

In einer PaF Architektur kann dies so nicht erfolgen, da die Nachrichten keiner Instanz der Filter zugeordnet werden können und es keine globalen Statusinformationen über die einzelnen Filter gibt. Das oben beschriebene Mittel des Persistierens der Nachrichten im Message Store lässt es lediglich zu, zu kompensierende Komponenten anhand der Message History zu identifizieren und die notwendigen Kompensationsaktionen anhand der Correlation ID und des Inhalts dieser persistenten Nachricht zuzuordnen.

Da bei einem Absturz durch einen Filter, wie in Fall 2 beschrieben, der Prozessfluss der PaF Kette jedoch unterbricht, ohne dass dies von einem Kanal oder einem Filter bemerkt wird (im Gegensatz zu

Fall 1), handelt es sich hier um ein Problem, für das es keine einfache Lösung gibt. Ansätze hierfür werden in **Abschnitt 4.5.7** behandelt.

### 4.3 Fehler in der Message

Eine weitere, mögliche Fehlersituation, die entstehen kann, ist, dass eine Nachricht durch eine falsche Struktur oder syntaktische Fehler (z.B. Feld(er): *Name* versus *Vorname* und *Nachname*) oder Unvollständigkeit nicht bearbeitet werden kann. In diesem Fall leitet der Filter selbst die Nachricht an den *Invalid Message Channel* [3] (IMC) weiter oder löscht diese, falls kein IMC vorgesehen ist.

#### **Problem:**

Die eventuell bereits teilverarbeitete Nachricht ist nun vom weiterführenden Verarbeitungsprozess ausgeschlossen. Die Nachricht kommt nie bei ihrem ultimativen Empfänger an und das System befindet sich im schlimmsten Fall in einem inkonsistenten Zustand. Es findet keine automatische Benachrichtigung durch das System statt, der Fehler wird erst erkannt, wenn die Nachricht auf dem Invalid Message Channel vom Administrator des Systems entdeckt und analysiert wird. Dies führt in vielen Fällen zu einem unbefriedigenden Ergebnis, da der Zeitpunkt der weiteren Bearbeitung der Nachricht oder das Erkennen des Fehlerfalls für den Initiator des Prozesses vorerst im Unklaren bleibt.

#### **Idee:**

Hier handelt es sich um eine Fehlersituation die ähnlich der in Abschnitt 4.2.1 ist. Der grundlegende Unterschied hier ist jedoch, dass ein Wiederversuchen einer Kontaktaufnahme oder eine Neuausführung des Prozesses immer wieder zu demselben Problem führen würde.

Für diese Fehlersituation gibt es verschiedene Strategien:

- Wenn es sich um eine unkritische Fehlersituation handelt, kann versucht werden, mit einem *Correction Loop* [17] die Nachricht zu reparieren und wieder ins System einzuspeisen. Dieser müsste ein bereits vorher modellierter Prozess sein, der über einen Ausgang des IMC von diesem aus erreicht werden kann. Hierzu muss zunächst der IMC befähigt werden, eingehende Nachrichten weiterzuleiten, d.h. er müsste auch einen Ausgang besitzen. Dies ist momentan in der Definition des Patterns nicht vorgesehen. Der Correction Loop kann versuchen, die Struktur und den Inhalt der Nachricht so anzupassen, dass sie vom Empfänger verarbeitet werden kann. Wie dies erreicht werden könnte und was für Probleme sich dabei stellen, wird ausführlicher in **Abschnitt 4.5.5** diskutiert.
- Wenn es sich um eine kritische Fehlersituation handelt, muss zwischen den beiden folgenden Punkten abgewogen werden:
  - (a) Es handelt sich um eine LRT, die schon weit fortgeschritten ist. Die Wahrscheinlichkeit, den Fehler durch einen Correction Loop zu beheben und die Zeit, die dies benötigen wird

rechtfertigt einen Forward Recovery (Repair-and-Retry, **Abschnitt 4.5.5**) Versuch des fehlgeschlagenen Prozesses.

- (b) Ist die LRT noch nicht weit fortgeschritten oder ist die Dauer oder Erfolgsaussicht der Korrekturmaßnahmen nicht abzusehen, sollte statt einem Correction Loop eine Kompensationsaktion durch den IMC angestoßen werden. Die Kompensationsaktion muss bereits vorher modelliert worden sein und durch die Weitergabe der ungültigen Nachricht an den Ausgang des IMC angestoßen werden. Eine genauere Betrachtung dieses Vorgangs wird in **Abschnitt 4.5.2** gegeben.

Da es sich bei dem Correction Loop um ein sehr optimistisches Szenario handelt, sollten auch die Möglichkeiten, dass es sich um eine sogenannte „vergiftete Nachricht“, eine „*Poisoned Message*“ handelt in Betracht gezogen werden. Sie könnte in einer böswilligen Absicht versendet worden sein, um den Filter absichtlich in eine Fehlersituation zu bringen. Die könnte Teil einer Attacke sein, die gegen das PaF Integrationsnetz gefahren wird. In einem solchen Fall kann ein Correction Loop diese Nachricht nicht automatisch korrigieren. Was dies für Konsequenzen hat und wie dieser Situation begegnet werden kann, wird in **Abschnitt 4.5.5** erörtert.

Eine unkomfortablere Methode, diesem Fehler zu begegnen, ist die manuelle Fehlerbehandlung dieser Situation. Diese wird hier aufgeführt, da es Situationen geben kann, in denen eine automatisierte Fehlerbehandlung nicht erfolgreich ist. In diesem Fall ist es wieder wichtig, dass die Nachricht mit einer Message History und einem Correlation Identifier versehen wurde. Die Message History liefert mit ihren Einträgen die Gruppe der Filter, bei denen der Filter, der die ungültige Nachricht produziert hat, zu finden ist. Es müssen korrektive Maßnahmen an diesem Filter vorgenommen werden. Eine Analyse und ein korrekatives Eingreifen in die Logik oder das Schema der Datenrepräsentation dieses Filters muss „von Hand“ erfolgen.

Dies ist sinnvoller als die fehlerhafte Implementierung, die zu der ungültigen Nachricht geführt hat mit einem Correction Loop zu korrigieren, der seinerseits wieder fehleranfällig sein kann. Da dies sehr zeitaufwendig ist, müssen vorher ausgeführte Arbeitsschritte in kritischen Fehlersituationen zunächst kompensiert werden. Kompensationsaktionen müssten dann manuell durch die Analyse der Einträge in der Message History der Nachricht erfolgen. Hilfreich wäre hier auch, wenn die Nachrichten aller Zwischenschritte in einem Message Store persistiert wurden.

Eine *Return Address* [3] kann, falls vorhanden, genutzt werden, um den Initiator des Prozesses über das Scheitern desselben zu informieren. Eine ausführliche Diskussion über den Correction Loop und die Kompensationsaktionen sind in **Abschnitt 4.5** zu finden.

## 4.4 Fehler im Kanal

Bei einem Fehler im Kanal geht die Nachricht bei einer Übertragung zwischen zwei Filtern verloren. Dies kann vor allem bei Kanälen geschehen, die nicht *Guaranteed-Delivery* [3] gewährleisten. Geschieht dies, so wird in lose gekoppelten Systemen auch niemand diese Nachricht vermissen. Eine PaF Kette wird durch einen Kanalfehler bei dem die Nachricht verloren geht nicht nur unterbrochen, sondern die Informationen, die eventuell über den Status des Prozesses in der Nachricht gespeichert

worden sind, sind nun unwiederbringlich verloren. Dies schafft eine äquivalente Situation zu der in Fall 2 des Abschnitts 4.2.2.

In unkritischen Fehlersituationen, wo der Verlust einer Nachricht hingenommen werden kann, kann bewusst auf eine Fehlerbehandlung verzichtet werden, wenn die Verlustwahrscheinlichkeit sehr gering ist und in keinem Verhältnis zu dem finanziellen Aufwand stehen würde, der für eine sichere Übertragung notwendig wäre (vergleiche **Abschnitt 4.5.6**).

Wenn beim Prozessdesign jedoch Geschäftsprozesse umgesetzt werden sollen, die Dienstgüteeigenschaften gewährleisten müssen, so liegt es im Ermessen des Prozessdesigners, sichere Kanäle für die PaF Kette auszuwählen. Verlustfreie Nachrichtenübertragung im Kanal kann durch den Store-and Forward Mechanismus des unterliegenden Messaging Systems forciert werden. Ein EI-Pattern, das diesen Vorgang in PaF-Architekturen beschreibt ist Guaranteed Delivery.

In einer PaF-Architektur geht eine Nachricht, die mit Guaranteed Delivery ausgeliefert wird nicht verloren. Garantiert ankommen tut sie jedoch im schlimmsten Fall nur auf dem DLC, sollte der eigentliche Empfänger der Nachricht durch einen Ausfall hierzu nicht im Stande sein. Dieser Fall wurde jedoch bereits in Abschnitt 4.2.1 behandelt.

## 4.5 Strategien zur Fehlerbehandlung

Dieser Abschnitt fasst die verschiedenen Strategien zusammen, die zur Fehlerbehandlung in vergleichbaren Systemen herangezogen werden können. Es werden außerdem Überlegungen angestellt, in welchen Situationen diese Strategien zum Einsatz kommen, ob und wie sie mit EIP in einer PaF-Architektur umgesetzt werden können und welche Probleme sich hier stellen. Die Abbildung der EIP auf BPEL und die damit einhergehende Einführung eines globalen Fehlerbehandlungskonzepts werden an dieser Stelle nicht berücksichtigt, die folgenden Fehlerbehandlungsstrategien werden hier zuerst auf eine mögliche Umsetzung mit EIP in einer PaF-Architektur geprüft. Eine Umsetzung dieser Strategien mit WS-BPEL wird in Kapitel 5 dieser Arbeit beschrieben.

Als erstes Konzept wird in **Abschnitt 4.5.1** die *Koordinierte Konversation* mit einem 2PC Protokoll betrachtet. Dieses pessimistische Konzept findet seine Anwendung sowohl in SOA (vergleiche WS-AT, Abschnitt 3.1.3) als auch in WfMS (vergleiche Atomic Spheres, Abschnitt 3.3.1). Mittels eines zentralen Transaktionsmanagers oder Koordinators wird hier ein uniformes Resultat der globalen Transaktion garantiert. Teilnehmende Transaktionen sind hier Subtransaktionen der globalen Transaktion und die Transaktionsgrenzen werden vom Koordinator von außen um die Gruppe der Subtransaktionen gelegt.

Da bei LRTs in Produktionsumgebungen die Ressourcen nicht für eine lange Dauer gesperrt werden können, Aktivitäten Teil dieser LRT sein können, die nicht transaktionaler Art sind oder gar menschliches Eingreifen erfordern, manche (externe) Geschäftspartner keine Kontrolle über die Sperrung ihrer Ressourcen zulassen, die koordinierte Konversation einen Overhead durch Vervielfachung des Nachrichtenaustauschs darstellt und durch Sperrprotokolle der Gesamtdurchsatz des Systems signifikant verringert wird, wird bei SOA (vergleiche WS-BA, Abschnitt 3.1.3) und WfMS

(vergleiche *Compensation Spheres*, Abschnitt 3.3.2) für LRTs ein kompensationsbasiertes Transaktionskonzept herangezogen. Eine Umsetzung von *Kompensationsaktionen* in einer PaF-Architektur wird im **Abschnitt 4.5.2** diskutiert.

Die Umsetzung einer *Compensation Sphere* in einer PaF-Architektur wird in **Abschnitt 4.5.3** untersucht.

Ein sehr einfaches, optimistisches, traditionelles Konzept der Fehlerbehandlung ist *Retry* oder *Retry-All*. Welchen Fehlersituationen hiermit begegnet werden kann und für welche Fehlersituationen dies nicht geeignet ist, wird in **Abschnitt 4.5.4** betrachtet.

Ein etwas komplizierteres, optimistisches Konzept der Fehlerbehandlung ist *Repair-and-Retry*. Hierbei wird versucht, eine Fehlersituation durch eine Root Cause Analyse der Nachricht zu ermitteln und mittels (hier: modellierten) Korrekturmaßnahmen diese Nachricht zu reparieren und wieder ins System einzuspeisen. Eine Möglichkeit, dies in einer PaF-Architektur umzusetzen und die Probleme die sich hier stellen, werden in **Abschnitt 4.5.5** erörtert.

Eine sehr triviale Möglichkeit Fehlern zu begegnen, ist sie einfach zu *ignorieren* [14] (*Write-Off*). Situationen oder Gründe, die für ein solches Vorgehen sprechen können, werden im **Abschnitt 4.5.6** diskutiert.

Abschließend wird das Konzept des *Phönix Verhaltens* aufgezeigt. Es findet seine Anwendung in WfMS (vergleiche Abschnitt 3.3.3). Hier wird durch geeignete Maßnahmen versucht, einem System, das einen Totalausfall erlitten hat, die notwendigen Informationen zurückzugeben, mit denen es sich selbstständig aus dieser Fehlersituation erholen und die Arbeit an den unterbrochenen Prozessen weiter fortführen kann. Dieses Konzept kommt dann zum Einsatz, wenn es sich um sehr langlaufende Prozesse handelt und eine Kompensation von allen bisher erfolgreich durchgeführten Arbeitsschritten zu viel Zeit und Geld kostet (Mehrfachausführung von Arbeitsschritten). Es ist dem *Repair-and-Retry* Konzept ähnlich, das in Abschnitt 4.5.5 besprochen wird. Der Unterschied ist jedoch, das beim *Repair-and-Retry* Konzept versucht wird, eine einzelne, ungültige Nachricht zu reparieren während beim Phönix Verhalten der gesamte Prozess, der von einem Systemausfall betroffen wurde repariert und weiter ausgeführt wird. Dies kann auch ein Wiederversuchen einzelner Aktivitäten des Prozesses beinhalten. Eine Idee, wie dieses Konzept auf PaF Architekturen angewendet werden könnte, erfolgt in **Abschnitt 4.5.7**.

In **Abschnitt 4.6** wird abschließend ein Fazit aus den Ergebnissen dieser Fehlerbehandlungsstrategien gezogen.

### 4.5.1 Koordinierte Konversation (2-Phasen-Commit Protokoll)

Koordinierte Konversation basiert auf dem pessimistischen 2PC Protokoll. Es ist auf einer getrennten Vorbereitungs- und Ausführungsphase der beteiligten Subtransaktionen (Teilnehmer) begründet. Ein Transaktionsmanager (Kordinator) sammelt in der Vorbereitungsphase auf einen *prepare* Request die Antworten (*Votes*) der Subtransaktionen ein. Aufgrund dieser Auswahl trifft der Koordinator dann eine Entscheidung, wie die globale Transaktion weiter verläuft.

Wenn alle Subtransaktionen mit einem *commit* gestimmt haben, dann sendet der Koordinator ein *commit* Request an alle Teilnehmer. Wenn mindestens einer der Teilnehmer auf den *prepare* Request ein *abort* sendet, dann wird die globale Transaktion vom Koordinator mit einem *abort* Request an alle Teilnehmer abgebrochen. Auf diese Weise wird ein uniformes Ergebnis aller Teilnehmer (Subtransaktionen) garantiert, das System befindet sich immer in einem konsistenten Zustand. Die folgende Abbildung zeigt den Mechanismus einer koordinierten Transaktion mit 3 Teilnehmern:

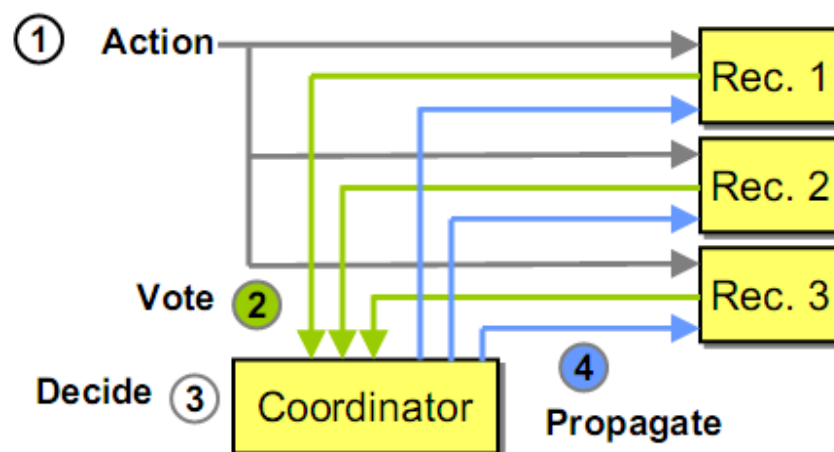


Abbildung 8: Koordinierte Transaktion mit 3 Teilnehmern (Quelle [15])

Die Nachteile dieses Protokolls sind, dass zum Einen ein erhöhter Overhead entsteht. Es müssen mehrere Nachrichten zwischen dem Koordinator und den Teilnehmern versendet werden. Zum Anderen wird die Leistung des Systems ausgebremst, da für den Zeitraum der koordinierten Transaktion durch Sperrprotokolle keine weiteren Zugriffe auf die beteiligten Ressourcen möglich gemacht werden.

Koordinierte Konversationen sind nur in kurzlaufenden Prozessen sinnvoll, da der Gesamtdurchsatz des Systems sonst signifikant beeinträchtigt werden würde. In langlaufenden Prozessen ist ein kompensationsbasiertes Transaktionskonzept die geeignetere Wahl (siehe Abschnitt 4.5.2).

### Umsetzung mit EIP

Bei einer Umsetzung dieses Konzepts mit EIP treten mehrere Probleme auf.

Einerseits werden die 2 Phasen der Kommunikation in einer PaF-Architektur von einem *Transactional Client* [3] nicht unterstützt, eine Einbindung eines Transaktionsmanagers ist also momentan nicht möglich, der Transactional Client lässt keine von außen angelegte Transaktionsgrenzen zu.

Andererseits stellt sich das Problem, dass dieser Transaktionsmanager ein globales Wissen über die Teilnehmer dieser Transaktion haben muss. Er muss bereits vorher wissen, wie viele Teilnehmer sich an dieser Transaktion beteiligen und wie sie zu erreichen sind. Da ein globales Wissen über den Prozess in eine PaF Architektur nicht existiert, ist dem Transaktionsmanager nicht von vorneherein bekannt, wie viele Teilnehmer die globale Transaktion haben wird. Er weiß also nicht, ab welchem Ereignis er die Ausführungsentscheidung für die globale Transaktion treffen kann. Ein Übergang von Phase 1 zu Phase 2 kann also nicht stattfinden.

Ein weiteres Problem, das sich hier stellt ist, dass eine koordinierte Transaktion nur transaktionale Implementierungen gruppieren kann. Ist in der PaF Kette jedoch z.B. eine Nachrichtentransformation durch einen *Message Translator* [3] enthalten, so müsste auch dieser Filter sich an der globalen Transaktion beteiligen. Eine transaktionale Implementierung dieses Filters erscheint jedoch wenig sinnvoll.

Durch diese Vorgabe ergeben sich für eine koordinierte Konversation zwei möglich Anwendungsfälle:

1. Es handelt sich um einen PaF Prozess, der nur transaktionale Filter beinhaltet.
2. Sind eine Mischung von nicht-transaktionalen und transaktionalen Filtern Teil der PaF Kette, so müssten die transaktionalen Filter so gruppiert werden, dass sie alle denselben, nicht-transaktionalen Filter als Vorgänger haben oder überhaupt keinen. Hiermit wird sichergestellt, dass es sich um einen kurzlaufenden Prozess handelt. Innerhalb der Gruppierung für die koordinierte Konversation dürfen dann nur transaktionale Filter enthalten sein.

Bei beiden Anwendungsfällen stellen sich die Probleme, die bereits oben aufgezeigt wurden. Zusätzlich ergibt sich durch den 2. Anwendungsfall ein weiteres Problem, dessen Auswirkungen Fragen aufwerfen, die die Eignung des Konzepts der koordinierten Konversation für nicht rein transaktionale Prozesse in PaF-Architekturen in Frage stellt.

Durch eine koordinierte Konversation kann ein einheitliches Resultat aller beteiligten Transaktionen erzwungen werden. Dies allein ist aber noch kein ausreichendes Mittel um Dienstgüte für den PaF Prozess zu erlangen. Beim 2. Anwendungsfall könnte, nach einer erfolgreichen Ausführung der koordinierten Transaktion im weiteren Verlauf der PaF Kette, die nicht Teil der koordinierten Transaktion ist, ein Fehler auftreten, der z.B. den Verlust einer Nachricht zur Folge hat. Nun sind die Transaktionen zwar bereits erfolgreich vorgenommen, der Prozess befindet sich aber trotzdem in einem Fehlerzustand. Als Beispiel kann ein Prozess genannt werden, bei dem bereits eine Umbuchung eines Betrags vom Konto des Käufers auf das des Verkäufers erfolgt ist, der

Versendeauftrag an die Spedition jedoch verloren gegangen ist oder auf Grund eines falschen Formats auf dem IMC gelandet ist. Da dieser Prozess nicht Teil der globalen Transaktion war, wird dies nicht bemerkt. Das System befindet sich nun zwar nicht in einem inkonsistenten Zustand, der Prozess ist aber dennoch fehlgeschlagen und der Kunde wird mit dem Ausgang der Bestellung nicht zufrieden sein.

Eine kritische Situation kann aber auch entstehen, wenn nach einer erfolgreichen Abwicklung des Falls 1 ein Ereignis eintritt, dass diesen Bestellvorgang, lange nach dem dieser erfolgreich und korrekt durchgeführt wurde, annullieren muss. Als Beispiel hierfür kann eine Situation dienen, in der sich z.B. gesetzliche Rahmenbedingungen für diese Bestellung geändert haben oder in der ein Brand das Warenlager des Verkäufers vernichtet hat.

Hier muss ein kompensationsbasiertes Transaktionskonzept zum Einsatz kommen, um die Effekte wieder rückgängig zu machen, die die koordinierte Transaktion bereits vorgenommen hat.

Eine koordinierte Konversation entbindet also nicht von einem kompensationsbasierten Transaktionskonzept. Wie dies in einer PaF Architektur umsetzbar ist, wird im nächsten Abschnitt beschrieben.

#### **4.5.2 Kompensationsaktionen**

Kompensation ist ein Konzept, das Effekte, die Transaktionen vorgenommen haben die Teil eines Prozesses sind der fehlschläft wieder rückgängig macht, um das System wieder in einen konsistenten Zustand zurück zu bringen.

In einem System, das keine Kompensation unterstützt, müssen Kompensationsaktionen explizit modelliert werden. Sie müssen Teil des Prozessflusses sein und die Ausnahmebehandlung von Fehlersituationen modellieren. Hierbei muss jeder mögliche Fehlerfall, der entstehen kann, separat modelliert werden.

Da eine PaF-Architektur im Gegensatz zu SOA oder Workflow keine Kompensation unterstützt, müssen alle Kompensationsaktionen vorher modelliert werden. Sie sind dann Teil der PaF Integrationslösung. In einer Produktionsumgebung, in der Dienstgüteeigenschaften eine große Rolle spielen, ist die Modellierung von Kompensationsaktionen ein sehr komplexes und äußerst aufwendiges Unterfangen, das jedoch von allergrößter Wichtigkeit ist.

#### **Umsetzung mit EIP**

Die folgende Abbildung zeigt eine einfache Kompensationsaktion in einer PaF-Architektur.

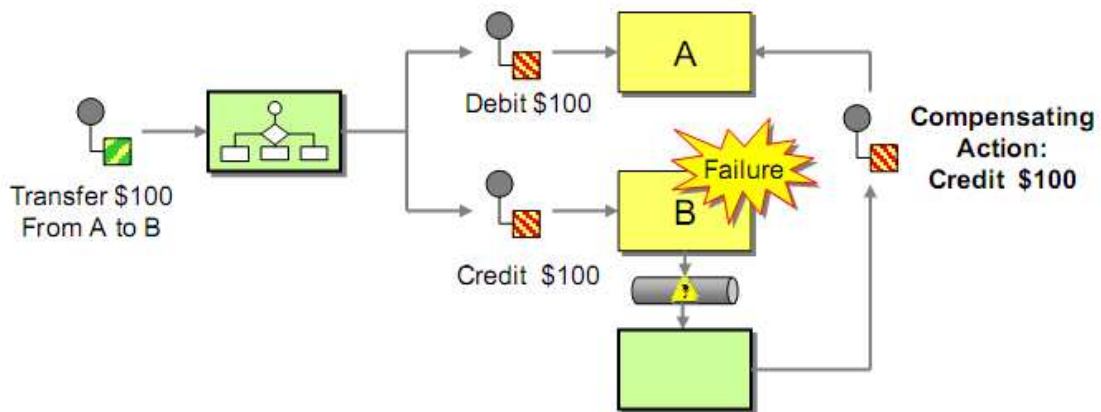


Abbildung 9: Kompensationsaktion (Quelle [15])

Bei dieser Abbildung handelt es sich um ein sehr optimistisches Szenario. Da kein globales Wissen über den Prozess existiert, wird in der Abbildung vorausgesetzt, dass alle Informationen, die zur Kompensation der Transaktion des Filters A notwendig sind in der „Credit \$ 100“ Nachricht an den Filter B bereits enthalten sind. Notwendige Informationen sind:

- Das benötigte Format und die Struktur der Nachricht für die Kompensationsaktion
- Der Methodename für die Ausführung der Kompensationsaktion
- Der Inhalt der Nachricht, der für die Kompensation benötigt wird

Weitere wichtige Fragen, die hier außer Acht gelassen wurden, sind:

- Waren andere Transaktionen beteiligt, die nun kompensiert werden müssen?
- Welche Daten, welche Nachrichten, welche Strukturen, welche Formate, welche Informationen werden für die Kompensationsaktionen benötigt?
- Wo können diese Informationen eingeholt werden?
- Wie können diese Informationen eingeholt werden?

Diese Fragen müssen bereits zum Zeitpunkt des Prozessdesigns bedacht und geklärt werden. Kompensationsaktionen müssen entsprechend so modelliert werden, dass sie Zugang zu diesen Informationen haben:

- Metainformation über die jeweiligen Nachrichten (Struktur, Format)
- Metainformationen über die jeweiligen Filter (Datenrepräsentation, Aufrufmethoden für die Kompensationsaktionen)
- Lokation der notwendigen Nachrichten (Message Store, Correlation)

Weitere Punkte sind hier denkbar. Bei einer Umsetzung von Kompensation mit EIP stellt sich also eine Vielzahl an Problemen.

Da nicht automatisch ermittelt werden kann, welche Aktionen kompensiert werden müssen und welche nicht, muss jeder transaktionale Filter eine entsprechende Kompensationsaktion wie oben in Abbildung 9 bei Filter B anstoßen können und diese Modellierung muss alle in der bisherigen PaF Kette vorgenommenen Transaktionen in dieser Weise kompensieren. Zusätzlich dazu muss es auch

eine Fehlerbehandlung für die Kompensationsaktionen selbst geben. Dies wirft die Frage auf, wie sicher eine Fehlerbehandlung, die im vorneherein modelliert werden muss, überhaupt sein kann. Schlägt eine Kompensationsaktion fehl, befindet sich das System ja immer noch in einem inkonsistenten Zustand. Da auch diese Fehlerbehandlung wiederum modelliert werden muss und somit erneut Fehlerquellen beinhalten kann, die behandelt werden müssen, resultiert das Vorhaben, die Dienstgüteeigenschaften umzusetzen, die für eine Produktionsumgebung notwendig sind, in einem kaum zu überblickenden, schwer zu wartenden Geflecht aus PaF Ketten, das eine hinreichende Fehlerbehandlung so teuer werden lässt, das eine Realisation durch Modellierung nicht praktikabel erscheint. Die Struktur eines solchen „sicheren“ Prozesses wäre äußerst chaotisch und könnte fraktale Ausmaße annehmen. Es ist zu befürchten, dass eine zu modellierende Fehlerbehandlung die Komplexität und den Aufwand ihrer modellierten *primary activity* um ein Vielfaches übertrifft.

Ein weiterer Nachteil ist, dass durch die Einführung von Kompensationsaktionen die lose Kopplung der Filter gebrochen wird. Zumindest die Filter, die kompensiert werden müssen, können nun nicht mehr beliebig umgeordnet werden, ohne dass auch deren Kompensationsaktionen nachgeführt werden müssen.

Ein Szenario, das einen Ausschnitt einer Kompensationsaktion eines einfachen Prozesses darstellt, bei dem mehrere Transaktionen beteiligt sind, wird in der folgenden Abbildung bildhaft vorgestellt. Die aus den zusätzlichen Kanälen und Komponenten für die Kompensationsaktion resultierende Unübersichtlichkeit des Gesamtprozesses ist hierbei noch stark abgemildert, da nur eine mögliche Kompensationsaktion ohne eine zugehörige Fehlermodellierung erstellt wurde.

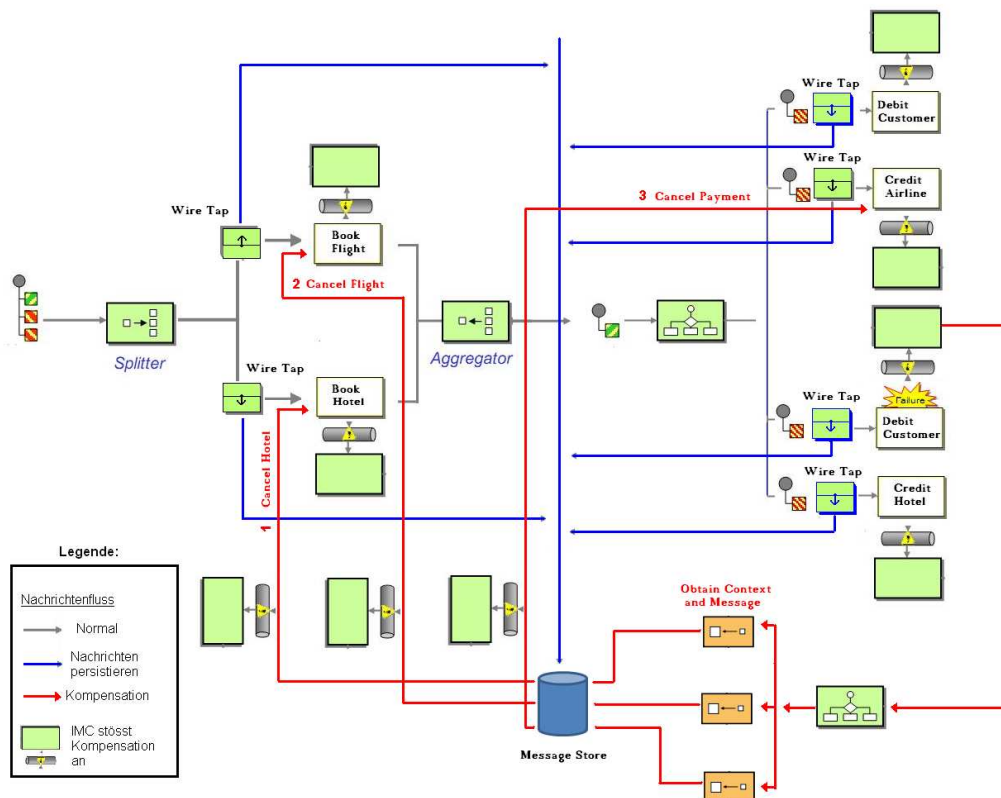


Abbildung 10: Ausschnitt einer Kompensationsaktion für einen einfachen Prozess

Betrachtet man nun diesen Prozess, so werden die Probleme sichtbar, mit denen eine zu modellierende Kompensationsaktion konfrontiert wird. Stellt sich der Kunde nach der Abbuchung des Flugs als insolvent heraus, so kann das Hotel nicht bezahlt werden und der IMC müsste analog zu Abbildung 9 eine Kompensationsaktion vornehmen. Ist dies in dem Szenario von Abbildung 9 noch recht offensichtlich und einfach zu bewerkstelligen, so stellen sich in dem Szenario von Abbildung 10 schon schwerwiegendere Probleme:

- Der Hotelaufenthalt muss storniert werden, hierfür wird Information benötigt die in der „Debit Customer“ Nachricht wahrscheinlich nicht mehr enthalten ist, die „Book Hotel“ Nachricht und der Name der Stornierungsmethode werden für die Kompensation benötigt.
- Der bereits bezahlte Flug muss storniert werden. Hierfür benötigt die Kompensationsaktion die „Book Flight“ Nachricht und wiederum den Namen der Stornierungsmethode.
- Der Zahlungsauftrag an die Airline muss storniert werden. Hierfür wird die „Credit Airline“ Nachricht und der Methodename des Services für die Stornierung benötigt.
- Der Kunde sollte über den fehlgeschlagenen Buchungsvorgang informiert werden.

Es müssen also zuvor alle Nachrichten in einem Message Store mit einer eindeutigen Correlation ID, die von allen Nachrichten dieses Prozesses geteilt wird, gespeichert worden sein. Dies wurde in Abbildung 10 beispielhaft mit einem *Wire Tap* [3] umgesetzt.

Ist dies geschehen, könnte eine Kompensationsaktion so modelliert werden, dass sie für jeden der oben aufgeführten Schritte die benötigten Informationen aus dem Message Store einholt und mit diesen die Transaktionen kompensiert.

Die Modellierung wurde bewusst einfach gehalten, es wurde nur ein möglicher Fehlerfall der Pipes and Filters Kette behandelt. Auch auf die Fehlerbehandlung der hier modellierten Kompensationsaktion wurde verzichtet. Zur besseren Übersicht wurden die Kanäle der Kompensationsaktion rot eingefärbt, die zur Speicherung der relevanten Nachrichten im Message Store blau. Alleine für die modellierte Kompensationsaktion wurden (mindestens) drei neue Fehlerquellen identifiziert, dargestellt durch den IMC und das grüne Kästchen. An jedem dieser Kästchen müsste eine weitere entsprechende Kompensationsaktion oder Fehlerbehandlung modelliert werden.

Eine Kompensationsaktion wurde modelliert, mindestens drei neue (zu behandelnde) Fehlerquellen sind hinzugekommen. Dies zeigt, dass der Versuch, Kompensation in einer Pipes and Filters Architektur mit Kompensationsaktionen durchzuführen, sehr teuer und aufwendig ist und wirft Zweifel an diesem Konzept auf. Es stellt sich die Frage, ob eine statische Lösung für eine Produktionsumgebung, in der jeder mögliche Fehlerfall durch vorher modellierte Pipes and Filters Ketten abgefangen werden muss, überhaupt für eine zufriedenstellende Umsetzung von Dienstgüteeigenschaften tauglich ist.

Durch die enge Kopplung, die die Kompensationsaktionen nötig machen, gehen zusätzlich die Vorteile der Flexibilität des Systems verloren.

### 4.5.3 Die Compensation Sphere

Die Compensation Sphere ist ein Konzept aus dem Bereich Workflow (vergleiche Abschnitt 3.3.2).

In einem Workflow übernimmt beim Scheitern einer Compensation Sphere (CS) das WfMS die Ermittlung der zu kompensierenden Arbeitseinheiten. Es stellt fest, welche Aktivitäten bereits ausgeführt wurden und erstellt einen Kompensationsgraphen, der in umgekehrter Reihenfolge der ursprünglichen Ausführung instanziiert wird. Dieses Unterfangen setzt voraus, dass für alle zu kompensierenden Arbeitsschritte der CS vorher Kompensationsaktionen definiert worden sind. Sie definieren das Verhalten, das im Falle eines Scheiterns des Prozesses notwendig ist, um das System wieder in einen konsistenten Zustand zu überführen. Dies wird diskrete Kompensation genannt.

Eine weitere Möglichkeit, eine CS zu kompensieren, ist die globale Kompensation. Hierbei wird eine einzelne Kompensationsaktion mit der CS zu assoziiert. Dies reflektiert Situationen, bei denen beim Scheitern eines Prozesses ein simples Umkehren des Prozessflusses und die Ausführung der Kompensationsaktionen nicht ausreichend sind. Diese globale Kompensationsaktion kann aus einer oder mehreren Aktivitäten oder auch einem eigenständigen Prozessmodell bestehen.

#### Umsetzung mit EIP

Eine CS mit EIP umzusetzen bringt eine Vielzahl an Problemen mit sich. Da die EIP in einer PaF-Architektur lose gekoppelt miteinander verbunden sind und kein globales Wissen über den Prozess und die einzelnen Aktivitäten vorhanden ist, gibt es nach dem Scheitern des Prozesses keine Möglichkeit festzustellen, welche Aktivitäten erfolgreich abgeschlossen haben und somit kompensiert werden müssen und welche Aktivitäten der CS beim Scheitern des Prozesses noch nicht ausgeführt wurden.

Bei einem Workflow übernimmt eben diese, für die weitere Behandlung der gescheiterten CS essentielle Analyse das WfMS selbst, das über globales Wissen über den Prozess verfügt.

Eine CS mit EIP in einer PaF-Architektur umzusetzen ist ohne globales Prozesswissen nicht möglich.

### 4.5.4 Retry und Retry-All

Wenn die Ausführung einer Aktivität in einem Prozess misslingt, gibt es zwei Möglichkeiten:

1. Die gescheiterte Aktivität wird gezielt erneut ausgeführt, bis dies gelingt (Retry). Ein vorher definierter Schwellenwert kann die maximale Anzahl der Wiederausführungsversuche festlegen.
2. Der ganze Prozess wird komplett neu ausgeführt (Retry-All). Allerdings muss es sich dann bei transaktionalen Aktivitäten um *Idempotent Receiver* [3] handeln, die eine erneute Ausführung derselben Nachricht ignorieren.

Die erste Möglichkeit kann in Situationen, in denen es sich um einen temporären, netzwerkbedingten Ausfall einer Aktivität handelt, sinnvoll sein. Ist der Schwellenwert für die Wiederausführungsversuche erreicht, müssen in Prozessen, in denen Transaktionen vorgenommen



- Kommt nun eine Nachricht herein, muss der Router einen Abgleich ihrer Correlation ID mit seiner Tabelle vornehmen.
  - (a) Die Correlation ID ist noch nicht vorhanden: Der Router speichert die Correlation ID der Nachricht mit der ID seines Standardausgangs und versendet die Nachricht auf diesem.
  - (b) Die Correlation ID ist bereits vorhanden: Der Router versendet die Nachricht an den nächsten Ausgang und speichert dessen ID mit der Correlation ID der Nachricht in seiner Tabelle ab.
- Der in (b) beschriebene Vorgang wiederholt sich solange, bis er erfolgreich ist oder der Zustellversuch an den letzten redundanten Filter gescheitert ist. Ist dies der Fall, so gibt dieser DLC die Nachricht nicht mehr an den Router zurück. Das Retry ist gescheitert. Nun kann durch den DLC eine beim Prozessdesign modellierte Kompensationsaktion oder Fehlerbehandlung angestoßen werden.
- Eine weitere Möglichkeit wäre, bei (b) die Ausgänge nicht sofort zu wechseln, sondern bis zum Erreichen eines Schwellenwertes den Kontakt erneut zu versuchen und erst dann den Ausgang zu wechseln.

Der oben beschriebene Router könnte in einer Pipes und Filters Architektur vielseitig eingesetzt werden. Er kann eine Maßnahme sein um den Durchsatz des Systems zu erhöhen, indem er die langsamsten Filter der Pipes and Filters Kette redundant zur Verfügung stellt.

Mit einer einfachen Änderung der Verteilungslogik könnte er auch Load Balancing vornehmen.

Zusätzlich bietet er ein Forward Recovery des Prozesses, das auch in kritischen Fehlersituationen verwendet werden kann, da nach einem Scheitern dessen, wie oben gezeigt, eine Kompensationsaktion angestoßen werden kann.

Ein weiterer Anwendungsfall wird im nun folgenden Abschnitt gezeigt.

#### **4.5.5 Correction Loop (Repair & Retry)**

Kann eine Nachricht auf Grund eines falschen Formats, einer falschen Struktur oder einer fehlenden Information von einer Komponente nicht bearbeitet werden, so legt diese sie in einer Pipes and Filters Architektur auf dem IMC ab.

Ein Weiterleiten der Nachricht auf den IMC verlagert das Problem jedoch nur. Dieser muss kontrolliert werden, um Anomalien des Systems zu entdecken. Mit einem Correction Loop könnte eine Weiterverarbeitung der Message durch den IMC angestoßen werden. Dazu müsste zunächst eine Analyse der Nachricht erfolgen. Hierfür ist jedoch Metawissen über die Komponente, die die Nachricht nicht verarbeiten konnte erforderlich.

- Welche Struktur, welches Format erwartet die Komponente?
- Welche Information muss die Message beinhalten, damit eine Verarbeitung für die Komponente möglich ist?

Kann dies in der Analyse geklärt werden, so könnte die erforderliche Information aus einem Message Store erworben werden. Eine anschließende Restrukturierung der Nachricht oder eine Anreicherung der Nachricht mit der fehlenden Information könnte somit eine wohlgeformte Nachricht erzeugen, die nun wieder ins System eingeschleust werden kann.

### Umsetzung mit EIP

Um dies mit EIP umzusetzen, müsste man den IMC mit einem Ausgang für die Weiterverarbeitung versehen. Soll die Analyse der Nachricht automatisch erfolgen, so muss an jedem Filter ein separater IMC angeschlossen sein. Nun müsste die Information über die Struktur der Nachricht, die dieser Filter erwartet, eingeholt werden. Dies könnte über die Übertragung eines XML-Schemas für die Nachricht erfolgen. Anhand dieses Schemas könnte die ungültige Nachricht mit einem *Message Translator* [3] ins korrekte Format übersetzt oder auch richtig strukturiert werden. Wenn Information fehlt, die für eine Bearbeitung notwendig gewesen wäre, so könnte diese mit einem *Content Enricher* [3] hinzugefügt werden. Dabei müssten allerdings alle relevanten Informationen und Nachrichten in einem *Message Store* [3] abrufbar sein. Diese Schritte müssen angemessen modelliert worden sein und sind Teil des Prozesses. Alle benötigten (Meta-)Informationen müssen zur Laufzeit abrufbar sein. Die folgende Abbildung zeigt eine mögliche Modellierung dieses Correction Loops:

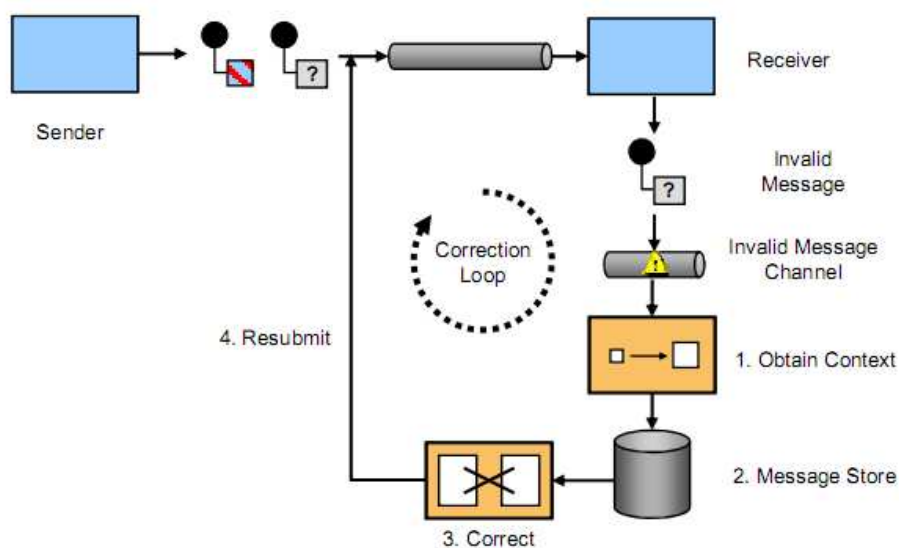


Abbildung 12: Correction Loop (Quelle [17])

Problematisch ist hier jedoch, dass der Correction Loop, wie oben dargestellt, eine gravierende Sicherheitslücke enthält. Wird eine Attacke gegen das Pipes and Filters Integrationsnetzwerk mit einer „vergifteten Nachricht“ (poisoned message) gefahren, so wird eine Korrektur dieser Nachricht nicht möglich sein. Wird dies im Correction Loop nicht festgestellt, dann befindet sich dieser in einer Endlosschleife. Dies würde die Situation, die eine Attacke herbeiführen will, also noch verschlimmern. Die „vergiftete Nachricht“ würde endlos ins System eingespeist werden.

Um dem entgegen zu wirken, kann erneut der Router, der in Abschnitt 4.5.4 vorgestellt wurde, eingesetzt werden. Der Correction Loop müsste dazu folgendermaßen verändert werden:

- Der Filter, der die Nachricht nicht verarbeiten kann, gibt sie direkt an den Router.
- Sein Standardausgang wird hierzu mit dem IMC verbunden.
- Sein Alternativausgang stößt eine Fehlerbehandlung an, die dem Prozess und den Anforderungen an dessen Dienstgüte entsprechen modelliert ist.

Wird nun die ungültige Nachricht erneut ins System eingespielt, stellt dies der Router auf Grund der Einträge in seiner Tabelle fest und unterbricht den Correction Loop, in dem er einen alternativen Pfad auswählt. In diesem Sinne könnte der Router auch als Detektor für „vergiftete Nachrichten“ fungieren, indem die Fehlerbehandlung seines alternativen Ausgangs eine Benachrichtigung des Systemadministrators modelliert, die ihn über die „vergiftete Nachricht“ informiert.

Statt dem, in diesem Abschnitt behandelten, Problem jedoch mit einer komplizierten Fehlerbehandlung zu begegnen, erscheint es sinnvoller, bei wiederkehrenden Problemen mit ungültigen Nachrichten den Filter zu identifizieren, der hierfür verantwortlich ist. Die für dieses Problem in Frage kommenden Filter können durch eine Analyse der Message History eingegrenzt werden. Der nicht korrekt arbeitende Filter muss manuell korrigiert werden, damit er gültige Nachrichten produziert.

#### **4.5.6 Fehler ignorieren (Write Off)**

Eine weitere Strategie, Fehlern zu begegnen, ist sie zu ignorieren. Die macht nur Sinn in Situationen, wo durch Fehler keine Kundenunzufriedenheit entsteht. Write Offs werden vor allem in Systemen eingesetzt, in denen eine teure Fehlerbehandlung in keinem effizienten Verhältnis dazu steht, dass ein in Anspruch genommener Dienst nicht berechnet wird [14]. Da diese Strategie für diese Arbeit nicht relevant ist, wird nun nicht näher darauf eingegangen.

#### **4.5.7 Phönix Verhalten**

In Workflow Systemen gibt es das Konzept des Phönix Verhaltens (vergleiche Abschnitt 3.3.3). Da in einer Pipes and Filters Architektur der Prozessstatus mit dem Persistieren der Nachrichten festgehalten werden kann, besteht die Möglichkeit, auch hier Phönix Verhalten umzusetzen.

Allerdings kann dies nicht mit EIP selbst erreicht werden. Zum Einen wird das 2PC Protokoll nicht unterstützt und zum Anderen ist eine Speicherung aller Nachrichten im Message Store ohne globales Wissen über den Prozess für eine Umsetzung des Phönix Verhaltens in einer PaF-Architektur nicht zuordnungsfähig.

Das der Pipes and Filters Architektur unterliegende Messaging System selbst könnte jedoch hierfür genutzt werden. Wird JMS für die Messaging Infrastruktur verwendet, gibt es eine Möglichkeit, den Fehler der in Abschnitt 4.2.2 beschrieben ist, abzufangen.

Bei JMS heißt die zuverlässige Nachrichtenübertragung Persistent Message [10]. Dies entspricht dem Store-and-Forward Mechanismus, der in Abschnitt 2.2 beschrieben wurde. JMS bietet die Möglichkeit, den Zeitpunkt für das Acknowledge der empfangenen Nachricht selbst zu wählen. Dies kann mit der Methode *acknowledge()* im *CLIENT\_ACKNOWLEDGE* Modus vorgenommen werden. Auf diese Weise kann erreicht werden, dass die Nachricht für den Filter A erst dann aus der Eingangsqueue gelöscht wird, wenn diese auch erfolgreich bearbeitet wurde. Die folgende Abbildung verdeutlicht diesen Vorgang:

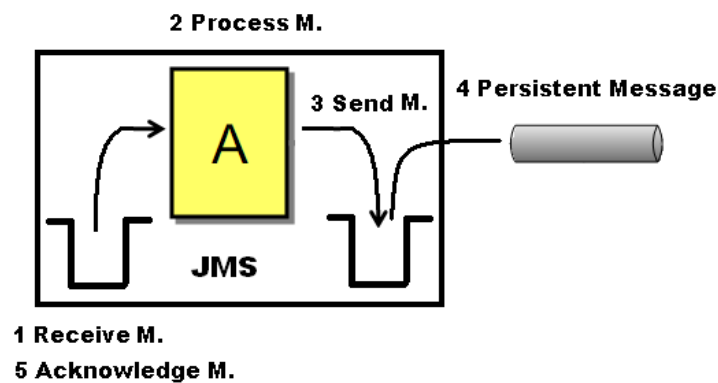


Abbildung 13: CLIENT\_ACKNOWLEDGE Modus in JMS

Stürzt das System oder der Filter, nachdem er die Nachricht angenommen hat ab, ist diese (im Gegensatz zu Guaranteed Delivery) noch immer in der Queue vorhanden. Ein Phoenix Verhalten des Pipes and Filters Prozesses wäre hierdurch möglich.

Nach einem Neustart des Systems wird JMS die Nachrichten der Queues an die entsprechenden Filter weiterleiten, der Prozessfluss geht also automatisch an den Stellen weiter, an denen er ausfallbedingt unterbrochen wurde.

Für den Fall, dass das System abstürzt, nachdem Punkt 3 oder Punkt 4 in der Abbildung abgeschlossen war, aber bevor Punkt 5 vorgenommen wurde, muss es sich um Idempotent Receiver handeln.

Die Realisierung dieses Konzepts setzt allerdings die Umsetzung des CLIENT\_ACKNOWLEDGE Modi durch die *acknowledge()* Methode durch den Filter voraus.

## 4.6 Fazit der Fehlerbehandlungsanalyse

Die Fehlerbehandlungsanalyse von EIP, die auf einer PaF-Architektur basieren, hat gezeigt, dass es sehr problematisch ist, Fehlerbehandlung ohne globale Mechanismen durchzuführen. Das Modellieren von Fehlerbehandlungen und Kompensationsaktionen ist für eine Produktionsumgebung kein ausreichendes Mittel, um Dienstgüteeigenschaften umzusetzen. Konzepte für Dienstgüte von vergleichbaren Architekturen, wie koordinierte Konversationen mit 2PC oder Compensation Spheres, sind in einer PaF-Architektur nicht umsetzbar. Mit der Modellierung von Fehlerbehandlung verlagert sich nur das Problem, Dienstgüte umzusetzen auf die Fehlerbehandlung selbst. Da hiermit, wie in Abschnitt 4.5.2 und 4.5.5 gezeigt, jedoch wiederum neue Fehlerquellen eingeführt werden, ist die Möglichkeit, Fehlerbehandlung durch Modellierung und ohne globales Wissen über den Prozess umzusetzen, nicht nur sehr aufwendig, teuer und komplex, ein Erfolg dieser in Produktionsumgebungen steht außerdem in Frage.

Wie bereits festgestellt wurde, kann man EIP auf BPEL abbilden [12]. Da BPEL selbst Teil von WfMS und der Web Service Spezifikationen ist und ein eigenes Fehlerbehandlungs- und kompensationsbasiertes Transaktionskonzept besitzt, stellt sich nun die Frage, wie man diese Konzepte auf EIP abbilden kann. Die Analyse hat gezeigt, dass ein kompensationsbasiertes Transaktionskonzept für eine Modellierung von EIP am besten geeignet ist.

BPEL gibt die Möglichkeit, Kompensation in mit EIP modellierte Szenarien einzuführen. Dies kann sowohl über die Parametrisierung der EIP als auch über die Modellierung von komplexeren Kompensationsaktionen oder Fehlerbehandlungen durch EIP im Editor selbst erfolgen. Die Kontrolle über die Ausführung von Kompensationsaktionen würde dann jedoch nicht mehr dem mit EIP modellierten PaF Szenario überlassen werden, sondern BPEL selbst würde sie übernehmen. Dies ist besonders deshalb von Vorteil, weil nun Modellierungen, wie in Abbildung 10 gezeigt, nicht mehr notwendig sind.

Der Snapshot des Scopes (siehe 3.2.3) der zu kompensierenden Aktivität, hält alle für die Kompensation notwendigen Nachrichten und Operationen bereit. Eine, für die Kompensation notwendige, Eingabe dieser Operationen kann auch durch eine entsprechende Parametrisierung der EIP im Vorfeld erreicht werden. Eine genauere Beschreibung dieses Vorgangs wird im nächsten Kapitel gegeben.

Ein weiterer Vorteil ist die Komponierbarkeit von BPEL mit anderen Web Service Standards. Hierdurch lassen sich weitere, von BPEL nicht direkt unterstützte, Konzepte wie über verschiedene Prozesse verteilte, koordinierte Transaktionen und WS-Security Modelle umsetzen. Da WS-BPEL selbst ein Web Service Standard ist, lässt er sich theoretisch auch mit allen anderen, in Abschnitt 3.1 aufgeführten Spezifikationen zur Umsetzung von Dienstgüte in SOA kombinieren.

Durch Abbildung der EIP auf BPEL wird ein für eine sinnvolle und effiziente Umsetzung von Dienstgüteeigenschaften notwendiges globales Fehlerbehandlungskonzept eingeführt. Was dies für Änderungen am bisherigen System bedingt, wird nun im nächsten Kapitel beschrieben.

## 5 Änderungen am System

Dieses Kapitel der Diplomarbeit beschreibt die Änderungen, die am bestehenden System „EAltoBPEL“ vorgenommen werden müssen, um das Fehlerbehandlungs- und das transaktionale Konzept von BPEL umzusetzen.

Die Analyse des vorangegangenen Kapitels hat gezeigt, dass eine Fehlerbehandlung ohne ein globales Konzept für eine Produktionsumgebung nicht ausreichend ist. Zwar können Kompensationsaktionen oder Fehlerbehandlungen mit EIP in einer PaF-Architektur modelliert werden, ohne einen Instanz - Begriff und globales Wissen über die (Fehler-)Zustände der einzelnen Aktivitäten des Prozesses, ist es jedoch sehr mühevoll, dies umzusetzen. Einzelne Kompensationsaktionen lassen sich nur zuordnen, wenn bekannt ist, welche dieser Aktivitäten erfolgreich abgeschlossen haben und welche sich in einem Fehlerzustand befinden.

Ohne eine Instanziierung der Filter, verbunden mit einer eindeutigen Prozessinstanz, lassen sich im Falle des Scheiterns eines Prozesses, vor allem in Produktionsumgebungen, in denen laufend Nachrichten durch die Filter und Pipes der PaF-Architektur fließen, die Arbeitsschritte, die von den verschiedenen Prozessen erfolgreich vorgenommen wurden, nicht unterscheiden von denen der Prozesse, die fehlgeschlagen sind. Auf diese Weise entsteht in einer Architektur, die kein globales Fehlerbehandlungskonzept unterstützt, bei einem Fehlerfall in einer Produktionsumgebung mit hohem Nachrichtendurchsatz ein Fehlerzustand, der nicht einfach zu beheben ist.

Gegenmaßnahmen verlagern die Fehleranfälligkeit nur auf die Fehlerbehandlung selbst und resultieren gezwungenermaßen in einem schwer zu wartenden und teuren Geflecht von Fehlerbehandlungs- und Kompensationsmodellierungen, die die ihrer primären Aktivität um ein Vielfaches übertreffen können. Auf diese Weise wird nicht nur die lose Kopplung der PaF-Architektur gebrochen, auch die Vorteile der Flexibilität des Systems gehen verloren. Eine hinreichende Fehlerbehandlung ist kaum durchführbar, das System bläht sich auf, wird teuer und schwer zu warten, ist weiterhin fehleranfällig und bleibt unzuverlässig.

Eine Abbildung der EIP auf BPEL scheint diese Probleme lösen zu können. Hierdurch wird der eigentliche Charakter der PaF-Architektur zwar verfälscht, dafür wird aber ein globales Fehlerbehandlungs- und ein kompensationsbasiertes Transaktionskonzept eingeführt, das essentiell für den Einsatz in Produktionsumgebungen ist.

In **Abschnitt 5.1** dieses Kapitels erfolgt nun zunächst eine Abbildung der im letzten Kapitel behandelten Fehlerbehandlungsstrategien auf die in BPEL transformierten EIP. Dabei wird geprüft, welche dieser Strategien sich durch die Parametrisierung der EIP umsetzen lassen und in welchen Situationen eine Parametrisierung der EIP alleine nicht ausreichend ist. Zur besseren Übersicht wurden die Fehlerbehandlungsstrategien gleich gegliedert wie im letzten Kapitel.

Die Notwendigkeit und Möglichkeit einer Umsetzung des DLCs und IMCs wird in **Abschnitt 5.2** und **5.3** diskutiert.

Da sowohl FH als auch CH in BPEL eine beliebig lange und verschachtelte Aktivität übergeben werden kann, muss für eine Umsetzung dessen im System „EAltoBPEL“ eine Änderung erfolgen, die es möglich macht, Kompensations- und Fehlerbehandlungsausgänge an beliebigen Filtern anzulegen. Dargestellt werden könnte dies mit einer FH bzw. CH Verbindung analog zu den PaF- oder Part-of Verbindungen, die es bereits jetzt im Editor gibt. Ist nun eine solche Verbindung gesetzt, so ist diese der Anfang einer, im Editor beliebig mit EIP zu modellierenden, Sequenz, die je nach der gewählten Verbindung eine Kompensationsaktion oder eine Fehlerbehandlung darstellt. Beim Durchlaufen der EIP durch die Klasse *BPELWriter* muss diese den Filter, der eine FH oder CH Verbindung besitzt, in einen Scope setzen und die Fehlermodellierung der Verbindung entsprechend dem FH oder CH dieses Scopes übergeben. Auf diese Weise lassen sich auch komplexere Fehlerbehandlungs- und Kompensationsszenarien modellieren. Eine Generierung dieser Szenarien mittels vorher eingegebener Parameter erscheint als wenig sinnvoll und ist nicht flexibel genug. Zu umfangreich müssten die Parameter und die Konfigurationsdialoge gestaltet werden. Änderungen des Modells der Klasse Filter und die Neueinführung eines Patterns mit dem Namen „**Catch Fault**“, das hierfür erforderlich ist, werden in **Abschnitt 5.4** genannt.

Eine Ausnahme bildet hier die <invoke> Kompensationsabkürzung von BPEL (vergleiche Abschnitt 3.2.3). Bei einem <invoke> Aufruf kann sowohl ein FH als auch ein CH direkt in den Aufruf eingekapselt werden, ohne dass explizit ein Scope definiert werden muss. Es kann also z.B. direkt ein Paar aus einer regulären Aktivität und ihrer Kompensationsaktivität gebildet werden. Im Falle eines FH kann eine Fehlermeldung direkt als Routing-Entscheidung für einen alternativen Pfad gewählt werden (vergleiche Abbildung 7, Abschnitt 3.2.4). Da für die Kompensation von <invoke> lediglich eine Kompensationsoperation benötigt wird, kann dies mittels der Eingabe dieses Parameters erfolgen. Es muss hierzu die Möglichkeit geschaffen werden, im Konfigurationsdialogs des External Service Patterns des Systems „EAltoBPEL“ eine Entscheidung zu treffen, ob der External Service kompensiert werden soll und wenn dies der Fall ist, wie der Name der Kompensationsoperation ist. Mit diesen Parametern kann diese Kompensationsaktion nun durch das System in BPEL transformiert werden. Diese Änderungen werden in **Abschnitt 5.5** dieses Kapitels aufgezeigt.

Eine weitere Möglichkeit, die sich durch eine Abbildung des Fehlerbehandlungskonzepts von BPEL auf EIP bietet, ist die Zusammenfassung mehrerer Filterpatterns in eine Einheit, die ein gemeinsames Schicksal hat. Dies wäre ähnlich einer CS bei WfMS (vergleiche 3.3.2). Hierzu müssten die gruppierten Filter von den Komponenten der BPEL-Codegenerierung in einen gemeinsamen Scope gepackt werden. Diesem Scope kann nun eine globale Fehlerbehandlung oder Kompensationsaktion übergeben werden, die im Editor mit EIP modelliert werden muss. Auf diese Weise lässt sich sowohl eine gemeinsame Fehlerbehandlung für eine Gruppe von Filtern spezifizieren, als auch eine globale Kompensationsaktion oder eine Compensation Sphere realisieren. Hierfür muss ein neues Pattern, die „**Global Sphere**“, eingeführt werden. Dies wird in **Abschnitt 5.6** aufgezeigt.

## 5.1 Fehlerbehandlungsstrategien für EIP mit BPEL

Dieser Abschnitt greift die im vorangegangenen Kapitel aufgeführten Fehlerbehandlungsstrategien auf und prüft sie nun auf eine Abbildung der EIP in BPEL. Es wird untersucht, inwieweit eine Umsetzung durch eine Parametrisierung der Patterns erfolgen kann, wann dies alleine nicht ausreichend ist und welche dieser Strategien in BPEL nicht umgesetzt werden können. Die Struktur dieses Abschnitts richtet sich zur besseren Übersicht an der aus Kapitel 4 Abschnitt 5.

### 5.1.1 Koordinierte Konversation

Bei koordinierten Konversationen, die sich über mehrere Prozesse und verteilte Ressourcen erstrecken, handelt es sich um ein orthogonales Problem, das außerhalb der Spezifikation von BPEL liegt [6]. Da WS-BPEL jedoch eine Web Service Spezifikation ist, sollte sie sich mit den anderen Web Service Spezifikationen aus dem SOA-Stack kombinieren lassen.

Im Speziellen wäre dies die Web Service Spezifikation WS-Transaction mit ihren Spezifikationen WS-AtomicTransaction, WS-BusinessActivity und WS-Coordination (vergleiche Abschnitt 3.1.3).

Um Business Transaction Management (BTM) in WS-BPEL zu integrieren, müssen vier Hauptfragen berücksichtigt werden [20]:

- Wie können Transaktionen zwischen Web Services und Geschäftsprozessen propagiert werden?
- Wie können Transaktionen zwischen Geschäftsprozessen und verschachtelten Scopes propagiert werden?
- Wie wird eine neue Transaktion innerhalb eines Geschäftsprozesses initiiert und terminiert?
- Wie wird das Verhalten von Prozessen und Subprozessen als Teilnehmer in der Transaktion definiert?

Wenn ein Client eine Web Service Operation aktivieren möchte, die von einem BPEL Prozess angeboten wird, so sollte dieser befähigt sein, den Kontext der Transaktion in die Aktivierungsnachricht einzubinden. Dies impliziert jedoch zusätzliche Syntax in dem <receive> Element [21].

Eine Ausarbeitung, die sich mit den oben genannten Fragen beschäftigt und die zusätzlich notwendige Syntax spezifiziert, wurde dem OASIS WS-BPEL technischen Komitee bereits im September 2003 vorgelegt [19]. Der Titel dieser Ausarbeitung lautet: *“BPEL and Business Transaction Management: Choreology Submission to OASIS WS-BPEL Technical Committee.”*

Das Dokument macht Vorschläge zur Erreichung von Kompatibilität mit den Web Service Spezifikationen von WS-Transaction. Ein Ausschnitt der Vorschläge zur Erweiterung der Syntax von WS-BPEL wird im Folgenden aufgeführt [19].

1. Das Hauptkonstrukt, das nötig ist, um mit einer Business Transaktion (BT) umzugehen, ist der Transaktionskontext. Dieser kann in einer Variablen gespeichert werden und könnte wie folgt aussehen:

- **<variable name="receivedContext" type="wscoor:CoordinationContext" />**

2. Weitere Konstrukte werden benötigt, um zu spezifizieren, wie dieser Kontext mit den Applikationsnachrichten übertragen und empfangen werden kann, die über <invoke>, <receive>, <reply>, usw. versendet und empfangen werden können. Bei einem <invoke> Aufruf eines Web Services, der sich auch in dieser Transaktion registrieren soll, muss zusätzlich zum *portType*, dem *partnerLink*, der *operation* und der *inputVariable* ein Transaktionskontext beigefügt werden und zusätzlich zur *outputVariable* muss die ID des teilnehmenden Web Services in der Transaktion beigefügt werden können. Die vorgeschlagenen Erweiterung der Syntax für einen <invoke> Aufruf ist:

- **inputBusinessTransactionContext="someContext"**
- **outputBusinessTransactionParticipant="WebServiceID"**

3. Weitere Konstrukte werden benötigt, um zu spezifizieren, wie eine BT initiiert wird. Vorgeschlagen wurde hier:

- **<businessTransaction action="new" context="someContext" />**

4. Um eine BT zu terminieren, vorzubereiten oder auch den Status der Transaktion zu erfragen, sind, unter anderen, folgende Erweiterungen notwendig:

- **<businessTransaction action="prepare" .../>**
- **<businessTransaction action="confirm" .../>**
- **<businessTransaction action="cancel" .../>**
- **<businessTransaction action="query" .../>**

5. Zusätzlich müssten ein **confirmHandler** und ein **cancelHandler** eingeführt werden, die über das Business Transaktions- Protokoll ausgelöst werden (falls kein FH ausgelöst wurde), um das jeweilige Ergebnis der BT fest zu legen.

Da diese Änderungen sinnvoll erscheinen, in WS-BPEL letztlich aber nicht eingeflossen sind, ist die Kompatibilität von WS-BPEL mit WS-T jedoch fragwürdig und bedarf weiterer Analysen.

Eine weitere Möglichkeit, koordinierte Transaktionen umzusetzen, die hier nicht außer Acht gelassen werden sollte, ist den generierten BPEL Prozess in eine Workflow Systemumgebung einzubinden, die Atomic Spheres (vergleiche Abschnitt 3.3.1) unterstützt.

## 5.1.2 Kompensationsaktionen

Um Kompensationsaktionen mit BPEL umzusetzen, gibt es mehrere Möglichkeiten. Zum Einen gibt es die Möglichkeit, einen <invoke> Aufruf eines Web Services direkt zu kompensieren. Hierfür ist kein eigener Scope notwendig, der CH, der die Kompensationsoperation spezifiziert wird direkt in den

<invoke> Aufruf der primären Aktivität eingekapselt. Diese Kompensationsaktion ist parametrisierbar, da sich nur der Name der Operation ändert. Eine Umsetzung dieser Möglichkeit wird in **Abschnitt 5.5** gegeben.

Die andere Möglichkeit besteht darin, eine Kompensationsaktion zu definieren, die aus mehreren Aktivitäten oder einem eigenständigen Prozessmodell besteht. Sie kommt zum Einsatz, wenn ein simples Stornieren der primären Aktivität nicht ausreichend ist oder nicht mehr durchgeführt werden kann.

Eine solche Kompensationsaktion kann nicht parametrisiert werden. Sie ist beliebig komplex und muss mit dem Editor des Systems „EAltoBPEL“ modelliert werden. Handelt es sich um eine Kompensationsaktion mit mehreren Aktivitäten, so muss die Modellierung, ausgehend von dem Filter, der diese Kompensationsaktion dem CH seines Scopes zuordnet, vom BPELWriter des Systems so interpretiert werden, dass die Modellierung beim Durchlaufen der Knoten des EIP-Netzes dem CH dieses Scopes übergeben wird. Dies wird in **Abschnitt 5.6** näher behandelt.

Handelt es sich bei der Kompensationsoperation um einen eigenständigen BPEL Prozess, so muss die <receive> -Schnittstelle des Kompensationsprozesses durch einen <invoke> Aufruf im CH des fehlgeschlagenen Scopes aufgerufen werden. Eine <exit> Anweisung muss den fehlgeschlagenen Prozess beenden.

### 5.1.3 Die Compensation Sphere

Um mit BPEL eine CS für das System „EAltoBPEL“ zu erstellen, bedarf es der Einführung eines neuen Patterns, der Global Sphere (vergleiche **Abschnitt 5.6.1**). Die Global Sphere reflektiert die Möglichkeit, in BPEL mehrere Aktivitäten (Filter) zusammenzufassen und einen Scope um sie zu legen. Die auf diese Weise zusammengefassten Aktivitäten teilen nun ein gemeinsames Schicksal.

Die Global Sphere wird im „EAltoBPEL“ Editor um ein oder mehrere Patterns gelegt und muss von der BPEL Code-generierenden Komponente so interpretiert werden, dass sie einen Scope um die beinhalteten Patterns legt und einen CH und/oder einen FH zur Verfügung stellt. Der FH kann diskrete und benutzerdefinierte Kompensation innerhalb der Global Sphere auslösen, im CH kann eine globale Kompensationsaktion für die Global Sphere definiert werden. Die Voraussetzungen hierfür sind:

- Sind die einzelnen Patterns, die in diesem Scope beinhaltet sind, bereits mit Kompensationsaktionen versehen (z.B. direkte Kompensation von <invoke> Aufrufen), so lässt sich im Falle eines Scheiterns der Global Sphere über ihren FH benutzerdefinierte- oder Standardkompensation durchführen. Dies entspricht der diskreten Kompensation von CS bei Workflowsystemen.
- Sind die einzelnen Patterns des Scopes nicht mit Kompensationsaktionen versehen, so kann für die Global Sphere eine globale Kompensationsaktion erstellt werden, die dann im Editor modelliert werden muss und dem CH der Global Sphere übergeben wird. Sie ist dann vom FH eines umschließenden Scopes oder des Prozesses selbst erreichbar. Zu beachten gilt es hier,

dass der CH mit der globalen Kompensationsaktion der Global Sphere nur dann erreichbar ist, wenn alle Aktivitäten in der Global Sphere (GS) erfolgreich abgeschlossen haben.

Auch kann hier ein FH definiert und eine Fehlerbehandlung modelliert werden. Dies macht Sinn in Situationen, in denen ein Fehler in der GS immer die gleichen Auswirkungen hätte, egal bei welcher Aktivität er vorkommt. Eine Möglichkeit, dies in BPEL umzusetzen, wird in **Abschnitt 5.6** gegeben.

#### **5.1.4 Retry und Retry-All**

Die Wiederausführung eines fehlgeschlagenen Prozesses ist in BPEL unproblematisch. Wenn Vorkehrungen zur Kompensation (z.B. direkte Kompensation eines <invoke> Aufrufs) getroffen wurden, hinterlässt der gescheiterte BPEL-Prozess keinen inkonsistenten Systemzustand und kann beliebig oft neu versucht werden, bis er letztlich erfolgreich abschließt. Ein Retry-All ist durch eine Neuausführung des fehlgeschlagenen Prozesses zu erreichen.

Wenn einzelne Aktivitäten des BPEL-Prozesses wieder versucht werden sollen, wie z.B. eine <invoke> Aktivität, so kann der fehlerhafte Kontaktversuch mit einem FH abgefangen werden, der in seiner Aktivität dieselbe <invoke> Aktion durchführen, die zuvor gescheitert war. Eine <wait> Aktivität kann den Zeitraum spezifizieren, der abgewartet werden soll, bis die neue Kontaktaufnahme versucht wird. Dies erfordert Änderungen am bisherigen Modell. Näheres hierzu kann in **Abschnitt 5.4.1** nachgelesen werden.

#### **5.1.4 Correction Loop**

Eine Umsetzung des Correction Loop Patterns ist in BPEL nicht möglich, da Links in BPEL keine zyklischen Strukturen haben dürfen. Außerdem ist sie aus mehreren Gründen wenig sinnvoll.

Zum Einen müsste ein großer Aufwand betrieben werden, um dies umzusetzen und zum Anderen kann ein Correction Loop sogenannte „poisoned Messages“ nicht automatisch korrigieren und würde die Systemressourcen daher nur unnötig belasten (vergleiche Abschnitt 4.5.5).

Wenn eine Nachricht auf Grund einer fehlerhaften Struktur von einem Filter nicht bearbeitet werden kann, so kann durch die Modellierung des FHs dieses Filters die Nachricht mit einem Web Service abgespeichert werden.

Da sich solche Probleme solange wiederholen, bis eine Root-Cause Analyse durchgeführt wurde und das Problem der falschen Struktur (oder des fehlenden Inhalts) identifiziert und behoben wurde, ist es sinnvoller, diesen Aufwand in die Behebung des eigentlichen Problems zu investieren, als einen Prozess zu entwerfen, der genau mit diesem (und nur diesem) Problem der Nachricht umgehen kann und versucht, die Inkompatibilität der Nachricht seinerseits durch eine fehleranfällige und aufwendige Modellierung aufzulösen.

### **5.1.6 Fehler ignorieren**

Fehler können in BPEL ignoriert werden, wenn der Fehlername z.B. als Routing Condition für den folgenden Prozessfluss verwendet wird oder innerhalb eines FHs in einem <catchAll> Konstrukt abgefangen wird, das nur eine <empty> Aktivität enthält. Da dies für diese Arbeit jedoch nicht wichtig ist, wird nun nicht näher darauf eingegangen.

### **5.1.7 Phönix Verhalten**

Phönix Verhalten für BPEL Prozesse lässt sich nicht umsetzen, da einerseits BPEL selbst keine Daten persistiert und andererseits ein fehlgeschlagener BPEL Prozess nicht einfach an beliebiger Stelle neu instanziiert werden kann. Es besteht zwar grundsätzlich die Möglichkeit, die Variablen eines BPEL Prozesses durch Web Services, die z.B. mit einer Datenbank verbunden sind abzuspeichern, jedoch können die für ein Phönix Verhalten viel wichtigeren Scope Snapshots der zu kompensierenden Aktivitäten nicht persistiert werden.

Eine Möglichkeit, Phönix Verhalten umzusetzen, die hier nicht außer Acht gelassen werden sollte, besteht darin, den generierten BPEL Prozess in eine Workflow Systemumgebung einzubinden, die Phönix Verhalten unterstützt (vergleiche Abschnitt 3.3.3).

## **5.2 Der Dead Letter Channel**

Wie bereits in [1] bei der Parametrisierung des DLCs festgestellt wurde, muss für eine Umsetzung des DLCs in BPEL dieser durch einen External Service modelliert werden, der die Schnittstelle zu einem Web Service darstellt, mit dem die „toten Nachrichten“ in einer Datei oder einer Datenbank abgelegt werden können, falls eine Speicherung dieser Nachrichten erwünscht ist.

Bei einer Umsetzung des DLCs in BPEL, gilt es verschiedene Dinge zu beachten. Wenn der DLC außerdem genutzt werden soll, um einen Forward Recovery Prozess wie in 4.5.4 anzustoßen, so ist es wichtig, dass durch die „tote Nachricht“ in dem Gesamtprozess keine Fehlersituation entstanden ist, die nicht speziell von einem FH abgefangen worden ist. Dies ist wichtig, da ein Forward Recovery in BPEL nur dann erfolgen kann, wenn sich der Prozessfluss nicht bereits in einer Fehlersituation befindet und das Forward Recovery in diesen fehlgeschlagenen Prozessbereich zurück navigiert. Weiter muss darauf geachtet werden, dass hierbei keine zyklischen Strukturen entstehen, diese sind in BPEL nicht erlaubt.

Durch die Abbildung der EIP auf BPEL stellt sich jedoch die Frage, ob das Pattern des DLCs in dieser Form überhaupt benötigt wird. Ist der DLC in der PaF -Architektur ein Ansatz, um Dienstgüteeigenschaften umzusetzen oder zumindest den Erhalt einer Nachricht nach einem fehlerhaften Zustellversuch zu gewährleisten, so ist der Ansatz, der in Abschnitt 4.5.4 gezeigt wurde, bei dem ein DLC ein Forward-Recovery oder eine Kompensationsaktion anstößt, für eine Produktionsumgebung doch eher ungeeignet, da dies zu aufwendig und fehleranfällig ist.

Kompensationsaktionen müssen im Editor modelliert werden und dem CH des Filters übergeben werden, über den diese Kompensationsaktion (bei erfolgreichem abschließen des Scopes) dann im Falle eines Scheiterns des Prozesses im weiteren Verlauf ausgelöst werden kann.

Scheidet also die Funktion des DLCs aus, Kompensationsaktionen anzustoßen oder ein Forward-Recovery vorzunehmen, so sind die einzigen Funktionen, die er gewährleisten muss, Nachrichten zu persistieren und Fehler zu signalisieren.

Die Funktion des DLCs, abgelaufene Nachrichten aus dem System zu nehmen, kann in BPEL durch einen EH auf Prozessebene realisiert werden. Wird eine Speicherung dieser Nachricht gewünscht, so kann dies im EH durch den Aufruf eines Web Services erfolgen.

Bei einem fehlerhaften Zustellversuch durch die Unerreichbarkeit eines Filters, kann ein DLC in BPEL umgesetzt werden, indem der gescheiterte Zustellversuch vom assoziierten FH des Scopes des Versenders abgefangen wird. Dieser FH stellt die Verbindung mit dem Web Service her und übergibt diesem die Nachricht zur Speicherung. Wird der Fehler an den umschließenden Scope weitergegeben, so wird die Kompensation von vorangegangenen Aktivitäten angestoßen und der Prozess terminiert.

### **5.3 Der Invalid Message Channel**

In einer PaF –Architektur gibt es die Möglichkeit einer Weiterverarbeitung, Korrektur und Wiedereinspielung einer ungültigen Nachricht zurück ins System durch den IMC (vergleiche Abschnitt 4.5.5). Ein Correction Loop kann in BPEL allerdings so nicht umgesetzt werden, da hier zyklische Strukturen nicht erlaubt sind.

Außer der Umsetzung eines Correction Loops muss der IMC die Möglichkeiten bieten, die Nachricht durch einen Web Service zu speichern und die Fähigkeit besitzen, einen Fehler anzuzeigen, um Kompensation auszulösen.

Eine, durch den IMC angestoßene und von diesem ausgehende Kompensationsaktion (vergleiche 4.5.2) ist bei einer Ausführung der EIP mit BPEL nicht mehr sinnvoll. Eben diese Kompensationsaktionen können durch das transaktionale Konzept von BPEL wesentlich besser und effizienter umgesetzt werden. Sie müssen im Editor modelliert werden und diese Modellierung muss in den CH derjenigen Filter abgebildet werden, die in die Fehlersituationen geraten können.

Wenn sich die Funktionalität des IMCs also auf das Abspeichern einer ungültigen Nachricht und die Weiterleitung des Fehlers beschränkt, so ist dies in BPEL einfacher zu lösen, als mit einem expliziten IMC. Durch eine ungültige Nachricht würde ohnehin ein BPEL Fehler ausgelöst werden, im dafür zuständigen FH kann diese Nachricht dann abgespeichert und der Fehler erneut geworfen werden (siehe 5.2), um Kompensation auszulösen.

## 5.4 Die Klasse Filter

Um das Fehlerbehandlungskonzept von BPEL auf EIP abzubilden, muss die Klasse Filter des Modells erweitert werden. Jedem Filter kann in BPEL theoretisch ein FH oder CH zugewiesen werden. Auf diese Weise kann nun für einen beliebigen Filter eine Kompensationsaktion oder eine Fehlerbehandlung eingeführt werden.

Als Beispiel für eine mögliche Fehlersituation kann ein Filter genommen werden, der eine Nachrichtentransformation vornimmt. Sollte dies auf Grund fehlender Information oder fehlerhafter Struktur scheitern, so würde in einer PaF-Architektur mit EIP dieser Filter die ungültige Nachricht auf dem IMC ablegen (vergleiche 4.3).

Eine weitere Fehlermöglichkeit ist, dass der Filter die Nachricht korrekt transformiert, sie an den nächsten Filter weitergeben will und dieser Kontaktversuch scheitert. Hier ist in einer PaF-Architektur mit EIP der DLC als Empfänger dieser Nachricht vorgesehen (vergleiche 4.2.1).

Wird nun das Fehlerbehandlungskonzept von BPEL auf eine PaF-Architektur mit EIP abgebildet, so eröffnet sich die Möglichkeit, die oben genannten Fehlersituationen mit einem FH abzufangen. Hierzu muss der Filter zunächst mit einem Scope versehen werden. Nun kann eine beliebige Modellierung mit EIP erfolgen, die das Verhalten umsetzt, dass in dieser Fehlersituation benötigt wird. Diese Modellierung muss dann von den Klassen, die für die BPEL Codegenerierung zuständig sind, so interpretiert werden, dass die Modellierung als Aktivität direkt in den FH des Scopes des Filters eingekapselt wird.

Auf diese Weise kann z.B. die Funktion des DLCs oder des IMCs emuliert werden, indem der FH des Filters die Nachricht mit der Hilfe eines Web Services speichert (vergleiche Abschnitt 5.2 und 5.3).

Auch andere Modellierungen sind denkbar, wie z.B. das Versenden einer Nachricht auf dem Control Bus an die Management Konsole.

Um dies umzusetzen, muss zunächst das Modell geändert werden. Die Klasse Filter muss um die Attribute *isCompensationHandled* und *isFaultHandled* vom Typ EBoolean erweitert werden. Nun kann angegeben werden, ob eine Fehlerbehandlung bzw. Kompensationsaktion für den Filter gewünscht ist oder nicht. Eine Auswahl dieser Möglichkeiten kann durch eine Checkbox im Konfigurationsdialog der Filter realisiert werden.

Dem Filter, der einen FH oder einen CH hat, der hierzu in einem Scope eingeschlossen werden muss, sollte die Möglichkeit gegeben werden, diesen Scope zu benennen, damit vom CH oder FH des übergeordneten Scopes oder vom FH des Prozesses benutzerdefinierte Kompensation mit der BPEL compensation activity `<compensateScope target="ScopeName"/>` durchgeführt werden kann. Hierzu muss die Klasse Filter um das Attribut *scopeName* vom Typ EString erweitert werden. Eine Angabe des Namens des Scopes kann durch eine Checkbox und die daraufhin aktivierte Texteingabe realisiert werden und ist optional.

### 5.4.1 Der Fault Handler des Filters

Um einen Filter mit FH zu versehen, sind weitere Änderungen an der Klasse Filter im Modell notwendig. In BPEL gibt es für einen FH zwei verschiedene Möglichkeiten, einen Fehler abzufangen. Zum Einen mit dem <catch> und zum Anderen mit dem <catchAll> Element.

Im Gegensatz zum <catch> Element muss für die Logik des <catchAll> Elements keine weitere Parametrisierung erfolgen. Die Aktivität selbst, die den Elementen folgt, ist meist nicht parametrisierbar. Ist <catchAll> angewählt, so muss der Editor einen zusätzlichen Ausgang für die Fehlerbehandlung am Filter bereitstellen. Hier kann nun im Editor eine Modellierung der Aktivität, die später dem <catchAll> Element des FHs zugewiesen wird, stattfinden. Der BPELWriter muss diese Modellierung dem <catchAll> Element des FHs des Filters übergeben. Eine Ausnahme bildet hier die Aktivität <empty>, bei der nichts unternommen wird.

Soll ein Fehler mit dem <catch> Element abgefangen werden, so ist zusätzlich der Fehlername, optional die Fehler Variable, der Nachrichtentyp des Fehlers und ein Fehler Element anzugeben. Dies kann jedoch nicht im Konfigurationsdialog des Filters selbst parametrisiert werden, da eventuell mehrere verschiedene Fehlersituationen des Filters mit verschiedenen <catch> Elementen abgefangen werden müssen. Hierfür muss ein neues Pattern mit dem Namen „**Catch Fault**“ eingeführt werden.

Ist im Konfigurationsdialog des Filters die <catch> Checkbox aktiviert, so lässt sich an diesem Filter eine beliebige Anzahl von **Catch Fault Patterns** anfügen, die jeweils einen, im Konfigurationsdialog des Catch Fault Patterns spezifizierten, Fehler abfangen. Die Modellierung der Fehlerbehandlung, die an das Catch Fault Pattern angelegt wird, muss von der Klasse BPELWriter des Systems in den FH des Filters in den jeweiligen <catch> Zweig, der durch das Catch Fault Pattern parametrisiert und spezifiziert wurde, abgebildet werden. Eine Parametrisierung des Catch Fault Patterns erfolgt in **Abschnitt 5.4.2.**

Folgende weitere Änderungen müssen also hierfür an der Klasse Filter vorgenommen werden:

- Das Attribut *catch* vom Typ EBoolean muss hinzugefügt werden.
- Das Attribut *catchAll* vom Typ EBoolean muss hinzugefügt werden.

Sind diese Parameter hinzugefügt, kann der Nutzer des Systems über den Konfigurationsdialog die Fehlerbehandlung spezifizieren. Da sowohl das <catch> als auch das <catchAll> Element in einem BPEL FH auftauchen können, sind mehrere Fehlerbehandlungsmodellierungen möglich oder unter Umständen sogar notwendig. Der Editor muss insofern angepasst werden, dass er mehrere verschiedene Ausgänge für eine zu modellierende Fehlerbehandlung anbietet und diese dann entsprechend den jeweiligen <catch> Elementen oder dem <catchAll> Element des FHs zuordnet. Die BPEL Codegenerierungskomponente des Systems muss hierfür die entsprechend modellierte Fehlerbehandlung in die zugehörigen <catch> oder <catchAll> Elemente des FHs abbilden.

In einem <catch> oder <catchAll> Element eines FHs sind weitere BPEL Basisaktivitäten denkbar, die nicht mit EIP modellierbar sind, aber für eine Fehlerbehandlung wichtig sein können. Diese Basisaktivitäten sind:

- <rethrow>
- <exit>
- <empty>
- <wait>

Bei einer <rethrow> Aktivität kann der Fehler an den übergeordneten Scope weitergegeben werden. Dies ist wichtig, falls z.B. trotz Abfangen des Fehlers Kompensation ausgelöst werden soll.

Bei einer <exit> Aktivität wird der Prozess auf Grund eines Fehlers beendet.

Eine <empty> Aktivität ist wichtig, falls bei einem Fehlerfall nichts unternommen werden soll.

Eine <wait> Aktivität spezifiziert entweder eine Duration-Expression oder eine Deadline-Expression. Hierdurch kann ein Zeitraum spezifiziert werden, den der Prozess abwartet, bevor er weitere Schritte unternimmt.

Die Syntax für eine Duration-Expression ist:

- `<wait for="P1Y1M1DT1H1M1S"/>` - Es wird 1 Jahr, 1 Monat, 1Tag, 1 Stunde, 1 Minute und 1 Sekunde gewartet.

Die Syntax für eine Deadline Expression ist:

- `<wait until="2009-12-24T12:00:00"/>` - Es wird bis Weihnachten, Punkt 12 Uhr mittags gewartet.

Um diese Aktivitäten in die Fehlerbehandlung des FHs aufzunehmen, muss erneut die Klasse Filter erweitert werden. Folgende Attribute müssen hinzugefügt werden:

- Das Attribut *rethrowActivity* vom Typ EBoolean muss hinzugefügt werden.
- Das Attribut *emptyActivity* vom Typ EBoolean muss hinzugefügt werden.
- Das Attribut *waitFor* vom Typ EString muss hinzugefügt werden.
- Das Attribut *waitUntil* vom Typ EString muss hinzugefügt werden.

Nun kann eine Abbildung dieser BPEL Basisaktivitäten in den FH des Filters über den Konfigurationsdialog ausgewählt werden. Hierbei sind diese Aktivitäten mit einer Checkbox umzusetzen. Die Aktivität <empty> schließt eine zu modellierende Fehlerbehandlung aus.

Zu beachten gilt es, dass bei einem Filter, bei dem *isCompensationHandled* oder *isFaultHandled* angewählt wurde und dem somit ein Ausgang für die Modellierung für einer Fehlerbehandlung oder Kompensationsaktion zur Verfügung gestellt wird, ein weiterer Filter modelliert und in einen eigenen Scope gesetzt werden kann, diesem Scope dann aber kein CH hinzugefügt werden darf, da dieser vom Prozess niemals erreicht werden kann. Die Auswahl des Attributes *isCompensationHandled* darf also in diesem Fall nicht möglich sein (vergleiche Abschnitt 3.2.3).

## 5.4.2 Das Catch Fault Pattern

In einem FH kann eine beliebige Anzahl von <catch> Zweigen verwendet werden, um spezielle Fehlersituation abzufangen, die ausgewiesen sind durch den Fehlernamen, den Nachrichtentyp des Fehlers, ein Fehler Element oder die Fehlervariable. Diese Zweige repräsentieren dann die verschiedene Logik, die einen speziellen Fehler abfängt. Theoretisch ist es möglich, eine große Anzahl von <catch> Elementen in einem FH anzugeben. Jedem dieser Zweige folgt dann eine separate, zu modellierende Fehlerbehandlung.

Da eine im Vorfeld unbekannte Anzahl an <catch> Elementen in einem FH nicht durch Parametrisierung erfolgen kann, muss hierfür ein neues Pattern eingeführt werden. Dieses Pattern beinhaltet dann die verschiedenen Parameter, die zum Abfangen eines Fehlers für die Logik innerhalb eines <catch> Zweigs im FH notwendig sind. Die Aktivität selbst, die nach dem Abfangen des Fehlers ausgeführt werden soll, kann nicht parametrisiert werden. Sie muss im Editor modelliert werden und dem FH des Filters, der das Catch Fault Pattern aktiviert hat, im jeweiligen, der Parametrisierung entsprechenden, <catch> Zweig des FHs abgebildet werden. Wenn der Fehler abgefangen, aber sonst nichts weiter unternommen werden soll, kann die <empty> Aktivität ausgewählt werden. Dann ist keine weitere Modellierung durch den Editor mehr möglich.

Weitere Aktivitäten, die durch Parameter angegeben werden können, sind <rethrow>, <exit> und <wait>. Wenn eine gewisse Zeit abgewartet werden soll, z.B. um eine erneute Kontaktaufnahme zu einer zuvor nicht erreichbaren Komponente wieder zu versuchen, so kann der Zeitraum in Form einer Duration- oder Deadline-Expression durch das <wait> Element angegeben werden. Eine bildhafte Darstellung des parametrisierten Catch Fault Patterns kann der unten stehenden Abbildung entnommen werden.

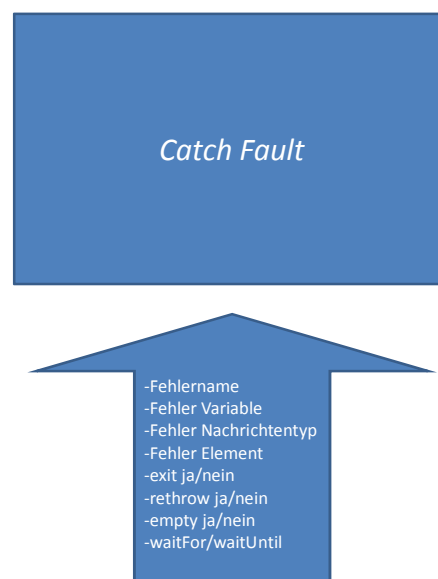


Abbildung 14: Parametrisierung des Catch Fault Patterns

Das Pattern selbst besitzt als Eingabe die Fehlerdaten, die durch die <catch> Logik abgefangen werden. Eine Ausgabe des Patterns ist entweder nicht gegeben (<empty> Aktivität) oder der modellierte alternative Pfad, der die Aktivität nach der <catch> Logik darstellt.

Ist die <catch> Checkbox im Konfigurationsdialog des Filters aktiviert, kann das Pattern mit diesem einzeln oder mehrfach kombiniert und in dieser Weise auch mit der Global Sphere verwendet werden.

### 5.4.3 Der Compensation Handler des Filters

Wird im Konfigurationsdialog des Filters Kompensation angewählt, so muss der Editor beim Filter einen zusätzlichen Ausgang für die Modellierung der Kompensationsaktion bereitstellen. An diesem Ausgang muss nun die gewünschte Aktivität mit EIP modelliert werden. Von der BPEL Codegenerierungskomponente des Systems muss diese Modellierung in den CH des Filters abgebildet werden. Wurde der Scope benannt, dann ist von einem übergeordneten Scope aus benutzerdefinierte Kompensation möglich (vergleiche Abschnitt 5.6.1).

Auch hier muss darauf geachtet werden, dass der modellierten Kompensationsaktion, sollte sie in einen Scope gekapselt werden, zwar ein FH, nicht aber ein CH hinzugefügt werden darf. Dies muss durch Deaktivierung der Checkbox mit dem Boolean *isCompensationHandled* forciert werden.

## 5.5 Der External Service

Der External Service stellt das Einzige der durch [1] und [2] parametrisierten Patterns dar, das mit der <invoke> - Abkürzung für Kompensation (vergleiche Abschnitt 3.2.3) direkt kompensiert werden kann. Diese Art der Kompensation ist die Einzige, die direkt durch die Parametrisierung eines Patterns realisiert werden kann.

Durch das External Service Pattern lässt sich ein Web Service aufrufen, dessen Arbeit bei einem Scheitern des Prozesses unter Umständen rückgängig gemacht werden muss. Mit ihm lässt sich zum Beispiel die Buchung eines Fluges oder die Speicherung einer Nachricht durch einen Web Service umsetzen. Das Beispiel zeigt, dass Kompensation für den ES zwar notwendig, nicht aber zwingend erforderlich sein muss. So sollte im Falle der Kompensation eines Prozesses ein Flug storniert, eine Nachricht aber nicht wieder aus dem „Speicher“ des Web Services gelöscht werden.

Ob ein ES im Fehlerfall also kompensiert werden muss oder nicht, kann nicht von vorne herein festgelegt werden, es muss dem Prozessmodellierer überlassen werden, welche Entscheidung diesbezüglich zu treffen ist. Hierzu muss ein neuer Parameter eingeführt werden, das Attribut *compensate* vom Typ EBoolean. Es beinhaltet einen booleschen Wert und kann im Konfigurationsdialog des ES-Patterns durch eine Checkbox dargestellt werden.

Für eine Kompensation wird als zusätzlicher Parameter hier nur noch der Name der Kompensationsoperation benötigt, das Attribut *compOperation* vom Typ EString. Der portType wird

als gleichbleibend angenommen. Die Eingabe für die Kompensationsoperation ist die Ausgabenachricht der primären Operation.

Der CH kann hierzu direkt in die *<invoke>* Aktivität eingekapselt werden, ohne explizit einen Scope zu definieren.

Bei einer erfolgreich abgeschlossenen *<invoke>* Aktivität, kann nun im Fehlerfall des weiteren Prozessverlaufs durch ihren CH die Kompensationsoperation der primären *<invoke>* Aktivität angestoßen werden.

## 5.6 Fehlerbehandlung für Filtergruppen

Die Fehlerbehandlung für Gruppierungen von Filtern reflektiert die Möglichkeit, in BPEL einen Scope um mehrere verschiedene Aktivitäten (Filter) zu legen. Diese Gruppe hat dann ein gemeinsames Schicksal. Tritt in einem der Filter dieser Gruppe ein Fehler auf, so kann dieser global durch einen FH, der mit dem Scope dieser Gruppe verbunden ist, abgefangen werden.

Soll Kompensation in der Filtergruppe realisiert werden, so gilt es folgendes zu beachten:

- Befinden sich die Filter der Gruppierung selbst in keinem Scope Konstrukt, so ist der CH der Filtergruppe nur erreichbar, wenn alle Filter der Gruppierung erfolgreich abschließen.
- Befinden sich die einzelnen Filter der Gruppierung selbst in einem Scope Konstrukt oder sinngemäß in einem *<invoke>* Aufruf mit eingekapseltem CH, so kann eine Compensation Sphere wie bei WfMS (vergleiche Abschnitt 3.3.2) umgesetzt werden. Im Falle des Scheiterns eines Filters der Gruppe wird nun Kompensation für die erfolgreich abgeschlossenen Scopes der Filtergruppe ausgelöst. Dies bedeutet, dass alle Filter die bereits abgeschlossen haben, über ihren eigenen CH erreichbar sind und vom FH der Filtergruppe kompensiert werden können. Der FH der Filtergruppe stoppt die Bearbeitung der nicht ausgeführten Scopes, die in ihn eingeschlossen sind und kann durch eine *<exit>* Aktivität den Prozess terminieren oder durch eine *<rethrow>* Aktivität den Fehler an den übergeordneten Scope oder an den Prozess selbst weitergeben, damit auch eventuell existierende Partnerscopes der Filtergruppe kompensiert werden können.

Um dies im „EAltBPEL“-Editor zu realisieren, muss ein neues Pattern eingeführt werden, die **Global Sphere**. Eine Parametrisierung dieses Patterns erfolgt im nächsten Abschnitt.

### 5.6.1 Das Global Sphere Pattern

Eine gemeinsame Fehlerbehandlung für eine ausgewählte Gruppe von Filtern erscheint dann sinnvoll, wenn alle Filter in der Gruppe fehlerfrei arbeiten müssen und der Ausfall oder das Fehlverhalten eines einzelnen Filters sich immer in derselben Fehlerbehandlung auswirkt.

Als Beispiel kann hier die Bearbeitung eines Werkstücks dienen. Egal welcher Arbeitsschritt eine fehlerhafte Bearbeitung vornimmt, es muss ein neues Werkstück angefordert werden, oder das fehlerhafte Werkstück muss zur Nacharbeit übergeben werden.

Soll die Global Sphere eine gemeinsame Fehlerbehandlung für die eingeschlossenen Filter realisieren, so muss die Fehlerbehandlung aktiviert werden. Wird nun das `<catchAll>` Element aktiviert, so muss vom System ein zusätzlicher Ausgang bereit gestellt werden, an dem die Fehlerbehandlung modelliert werden kann. Diese muss anschließend dem `<catchAll>` Element des FH der Global Sphere (GS) übergeben werden.

Wird das `<catch>` Element gewählt, kann zusätzlich eine beliebige Anzahl von Catch Fault (CF) Patterns an die GS angefügt werden. Jedem dieser CF Patterns muss vom System ein Ausgang für die Fehlermodellierung zur Verfügung gestellt werden, der die Aktivität der Fehlerbehandlung des jeweiligen CF Patterns repräsentiert (siehe Abschnitt 5.4.2).

Wird die Global Sphere als CS aufgesetzt und eine Standardkompensation der beinhalteten Filter soll erfolgen (diese müssen dann bereits jeweils Kompensationsaktionen definiert haben), so müssen für die Global Sphere keine Parameter angegeben werden. Die Global Sphere wird dann mit einem impliziten FH ausgestattet, der die beinhalteten Filter mit Standardkompensation kompensiert.

Wenn benutzerdefinierte Kompensation erfolgen soll, so müssen die jeweiligen Scope Namen der Filtergruppe in der Reihenfolge eingegeben werden, in der sie kompensiert werden sollen. Dies kann in Form einer Liste gespeichert werden, die dann vom `BPELWriter` entsprechend im FH des Scopes der Global Sphere abgebildet werden muss (`BPEL <compensateScope>` Aktivität).

Wenn eine globale Kompensationsaktion für die in der GS beinhalteten Filtern umgesetzt werden soll, muss im Konfigurationsdialog der GS der CH aktiviert werden. Nun muss vom System ein Ausgang bereitgestellt werden, an dem die Kompensationsaktion im Editor modelliert werden kann. Zu beachten gilt es, dass alle Aktivitäten der GS erfolgreich abschließen müssen, damit der CH der GS erreichbar ist.

Um dies umzusetzen, muss ein neues Pattern erstellt werden. Dieses Pattern hat weder eine Ein- noch Ausgabe und beinhaltet selbst eine beliebige Anzahl von Patterns. Wird die Global Sphere um eine Gruppe von Filtern gelegt, so kann die zugehörige Fehlerbehandlung und/oder Kompensationsaktion modelliert werden. Die Klasse `BPELWriter` muss das Pattern als Scope um die beinhalteten Filterpatterns interpretieren. Die mit dem Pattern verbundene, im Editor mit EIP zu modellierende, Fehlerbehandlung bzw. Kompensationsaktion muss von der Klasse `BPELWriter` direkt als BPEL Aktivität dem FH bzw. CH des Scopes übergeben werden. Eine Parametrisierung des Patterns Global Sphere kann der Abbildung, die auf der nächsten Seite folgt, entnommen werden.

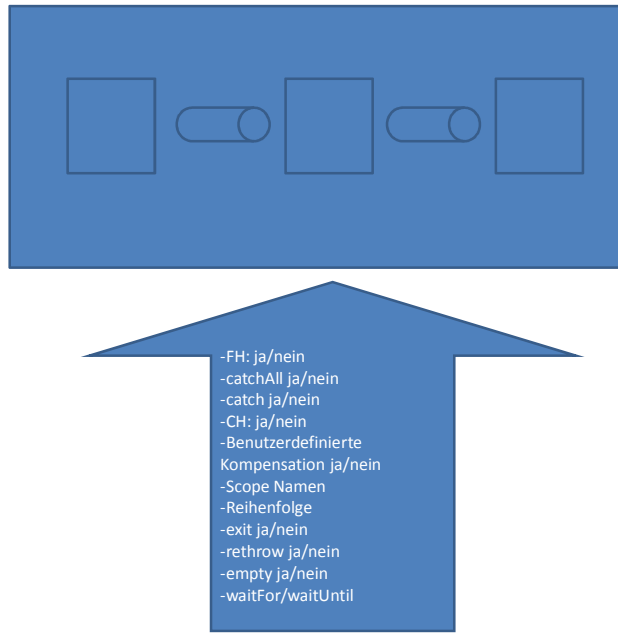


Abbildung 15: Parametrisierung der Global Sphere

## 6 Implementierung und Szenario

Dieses Kapitel der Diplomarbeit dokumentiert die Änderungen, die am System „EAltoBPEL“ vorgenommen wurden.

Zunächst wurde die direkte Kompensation für <invoke> Aufrufe umgesetzt. Die Änderungen am Modell und in der Implementierung des Systems „EAltoBPEL“, die hierfür erforderlich waren, werden in **Abschnitt 6.1** dokumentiert.

Die Änderungen am Modell der Klasse Filter, die notwendig sind, um das Fehlerbehandlungskonzept von BPEL auf EIP abzubilden, wurden durchgeführt. Die im letzten Kapitel neu eingeführten und parametrisierten Patterns „Global Sphere“ und „Catch Fault“ wurden dem System „EAltoBPEL“ hinzugefügt, sind aber noch nicht funktionstüchtig. Die Änderungen des Modells werden in **Abschnitt 6.2** dargestellt.

Ein syntaktischer Fehler in der Implementierung des WSDL-Writers des bestehenden Systems wurde behoben. Dies wird kurz in **Abschnitt 6.3** behandelt.

Ein Szenario, das das kompensationsbasierte Transaktionskonzept von BPEL in das External Service Pattern integriert, wurde mit dem „EAltoBPEL“ Editor erstellt und auf der BPEL Engine von ActiveVOS [18] aufgesetzt und ausgeführt. Das Vorgehen zur Ausführung und die Ergebnisse werden in **Abschnitt 6.4** beschrieben.

Eine Checkliste, die die Klassen beinhaltet, die von Entwicklern, die sich zukünftig mit dem System „EAltoBPEL“ befassen, geändert werden müssen, falls neue Pattern implementiert oder bereits angelegte Pattern umgesetzt werden sollen, befindet sich in **Abschnitt 6.5**.

Den Abschluss dieses Kapitels bildet in **Abschnitt 6.6** ein Ausblick, in dem zukünftige Arbeiten identifiziert werden, die am System „EAltoBPEL“ vorgenommen werden können.

### 6.1 Implementierung des kompensierbaren External Service

Dieser Abschnitt dokumentiert die Änderungen, die am bestehenden System „EAltoBPEL“ vorgenommen wurden, um die Kompensation eines ES zu ermöglichen. Die vorgenommenen Änderungen beziehen sich auf die in Abschnitt 5.5 genannte, direkte Kompensation einer <invoke> Aktivität. Es wird kein Scope für den ES angelegt, die CH werden bei der Generierung des BPEL Codes durch das System „EAltoBPEL“ in die <invoke> Aktivität des ES selbst eingeschlossen.

Bei einem Fehler im Prozess erfolgt Standardkompensation durch den impliziten FH des Prozesses.

## Änderungen am Modell

Um Kompensation für den ES zu ermöglichen, mussten zunächst Änderungen am Modell selbst vorgenommen werden. Der ES erbt alle Eigenschaften der Klasse Filter und besitzt selbst bisher keine eigenen Attribute.

Da direkte Kompensation für die Oberklasse Filter nicht relevant ist, weil sie nur für <invoke> Aufrufe selbst benötigt wird, wurden im Modell dem ES die Attribute „*compensate*“, vom Typ EBoolean und „*compOperationForInvokeCall*“ vom Typ EString hinzugefügt. Die folgende Abbildung zeigt die Änderungen am Modell des ES:

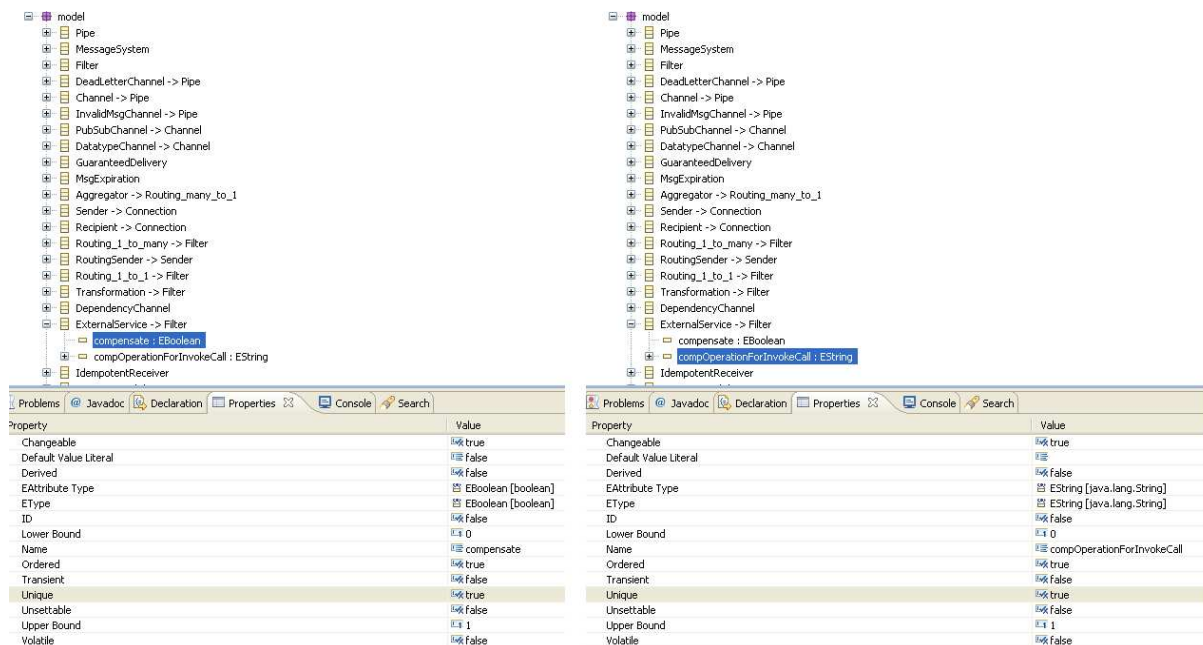


Abbildung 16: Änderungen am Modell des ES

Im Anschluss an diese Änderungen wurde der Modell Code neu generiert. Die Generierung vollzog automatische Änderungen in den Klassen „*ExternalService.java*“ des Pakets „*de.unistuttgart.iaas.eaiparam.model*“ und „*ExternalServiceImpl.java*“ des Pakets „*de.unistuttgart.iaas.eaiparam.model.impl*“.

## Änderungen am Dialogfenster des ES

Um die Möglichkeit zu geben, auszuwählen ob kompensiert werden soll und welche Kompensationsoperation dabei ausgeführt wird, mussten die Klassen „*ExternalServicePropertiesDialog.java*“ im Paket „*de.unistuttgart.iaas.eaiparam.editors.dialogs*“ und „*EditExternalServiceCommand.java*“ im Paket „*de.unistuttgart.iaas.eaiparam.editors.editParts.commands*“ manuell angepasst werden.

In der Klasse „*ExternalServicePropertiesDialog.java*“, die den Konfigurationsdialog des ES darstellt, wurde hierzu eine Checkbox mit dem Namen „Compensate“ und eine Combo-Box für den Namen der Kompensationsoperation eingefügt.

Damit diese Werte auch korrekt übergeben werden, mussten zusätzlich die Redo-, Undo- und Execute Methoden der Klasse „*EditExternalServiceCommand.java*“ erweitert werden.

Die folgende Abbildung zeigt das bei diesen Änderungen entstandene Dialogfenster des ES:

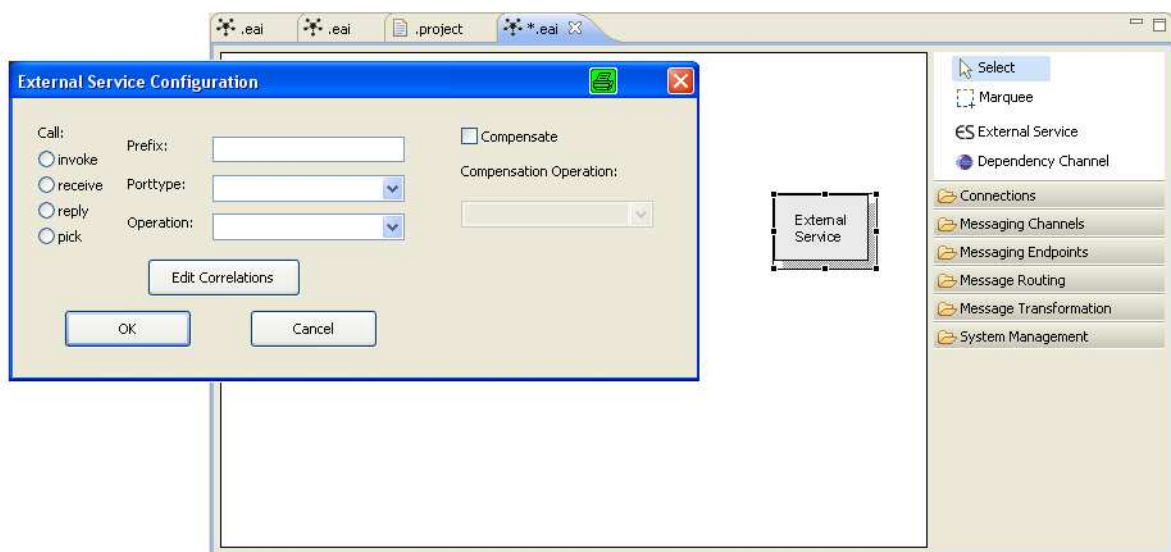


Abbildung 17 : ES Dialogfenster mit Kompensation

Die manuellen Änderungen, die am Code vorgenommen werden mussten, sind für eine bessere Übersichtlichkeit durch Kommentare kenntlich gemacht worden, die mit dem Namens Kürzel des Autors „@schmidfk“ beginnen. Das Ende des jeweiligen Blocks wurde mit “//” gekennzeichnet.

### Änderungen an der Implementierung für die Ressourcen des Editors

Damit das modellierte Szenario und die eingetragenen Werte der Konfigurationsdialoge korrekt gespeichert und später wieder eingeladen werden können, musste die Klasse „*EAIResourceImpl.java*“ aus dem Paket „*de.unistuttgart.iaas.eaiparam.editors.resource*“ erweitert werden.

### Änderungen an der Implementierung des Model Managers

Die Klasse „*MessageSystemModelManager.java*“ aus dem Paket „*de.unistuttgart.iaas.eaiparam.editors*“ ist zuständig für die Überprüfung des EAI Patterns Modells auf Fehler oder Probleme. Es

wurden Fehlermeldungen für die Kompensation des ES eingefügt, wie zum Beispiel eine Überprüfung, ob der Aktivierung von Kompensation im Dialog auch die Eingabe einer Kompensationsoperation gefolgt ist, oder ob es sich beim Call des ES um eine <invoke> Aktivität handelt. Entsprechende Fehlermeldungen werden nun im Problems-Tab des Editors ausgegeben.

Die folgende Abbildung zeigt die Ausgabe zu einer Fehlersituation, bei der vergessen wurde, einem kompensierten ES eine Kompensationsoperation zu übergeben.

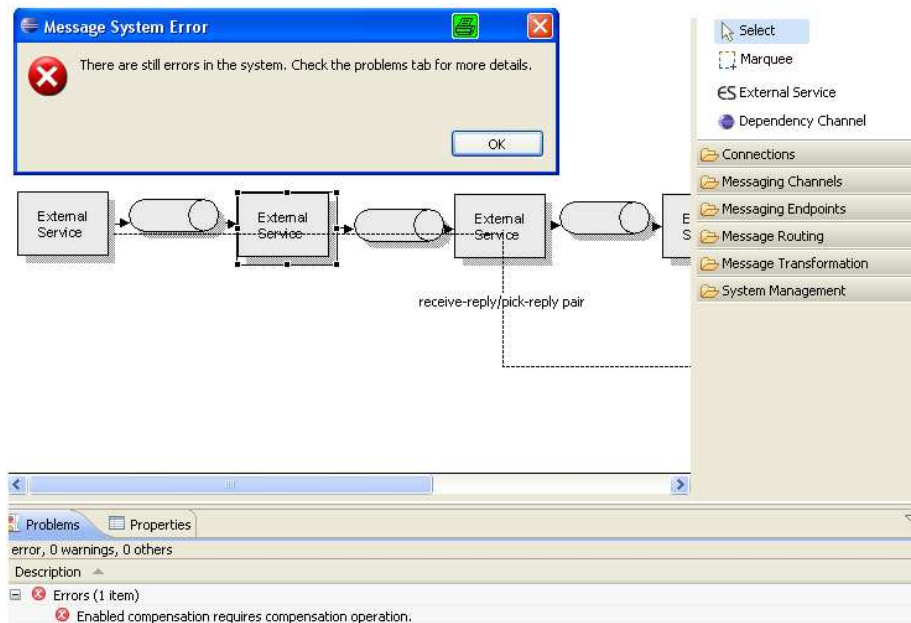


Abbildung 18 : Anpassung des Model Managers für Kompensation

## Änderungen für die Generierung von BPEL

Damit aus dem EAI Patterns Modell des kompensierten ES der BPEL Code generiert werden kann, mussten die Klasse „*ExternalServicePattern.java*“ aus dem Paket „*de.unistuttgart.iaas.framework.xform2bpel.patterns.other*“ und die Klasse „*BPELWriter.java*“ aus dem Paket „*de.unistuttgart.iaas.eaiparam.actions*“ angepasst werden.

In der Klasse „*ExternalServicePattern.java*“ wird hierzu im Fall eines zu kompensierenden ES eine entsprechende Kompensationsaktion erstellt und einem Compensation Handler übergeben. Dieser CH wird dann der ursprünglichen <invoke> Aktivität hinzugefügt. Auf diese Weise wird der <invoke> Aktivität direkt die Kompensationsaktion übergeben, ohne explizit einen Scope für den ES definieren zu müssen.

In der Klasse „*BPELWriter.java*“ werden, falls Kompensation gewählt wurde, der boolesche Wert „True“ und die eingegebene Kompensationsoperation für den Setup des ES zur BPEL Codegenerierung hinzugefügt.

## Änderungen an der Darstellung im Editor

Um die External Services, die kompensiert wurden, von den anderen External Services abzuheben, wurde ein neues Bild eingefügt. Es befindet sich im Pictures Verzeichnis des EAltoBPEL Projekts und heißt *externalservicecompensated.jpg*. Um dies im Editor anzuzeigen, wurde die Klasse „*ExternalServiceEditPart.java*“ des Pakets „*de.unistuttgart.iaas.eaiparam.editors.editParts*“ angepasst. Die folgende Abbildung zeigt einen kompensierten External Service. Durch die Kennzeichnung der kompensierten ES mit einem „C!“ in einer grünen Box wird die Bedienung des Editors und die Übersicht verbessert.



Abbildung 19: Kennzeichnung kompensierter ES

## 6.2 Änderungen am Modell

Dieser Abschnitt fasst die Änderungen zusammen, die am Modell vorgenommen wurden, um das Fehlerbehandlungs- und das kompensationsbasierte Transaktionskonzept von BPEL auf EIP abzubilden. Das Modell der Klasse Filter wurde erweitert und zwei neue Klassen wurden in das Modell aufgenommen, die die zwei neuen Patterns repräsentieren, die im letzten Kapitel parametrisiert wurden. Nach den Änderungen und Erweiterungen des Modells, wurde der Modell Code neu generiert. Die hierdurch neu entstandenen Klassen befinden sich in den Paketen „*de.unistuttgart.iaas.eaiparam.model*“ und „*de.unistuttgart.iaas.eaiparam.model.impl*“.

### 6.2.1 Die Klasse Filter

Dieser Abschnitt bezieht sich auf die Änderungen am Modell, die zur Erweiterung um die in Abschnitt 5.4 und Abschnitt 5.4.1 identifizierten Parameter für die Klasse Filter vorgenommen wurden. Es folgt eine Aufzählung der neu hinzugefügten Attribute und ihres Typs:

- isCompensationHandled vom Typ EBoolean
- isFaultHandled vom Typ EBoolean
- catch vom Typ EBoolean
- catchAll vom Typ EBoolean
- rethrowActivity vom Typ EBoolean
- emptyActivity vom Typ EBoolean
- exitActivity vom Typ EBoolean
- scopeName vom Typ EString
- waitFor vom Typ EString
- waitUntil vom Typ EString

Diese Attribute sind notwendig, wenn man das Fehlerbehandlungskonzept von BPEL auf EIP abbilden will. Sie wurden dem Modell in der Klasse Filter hinzugefügt. Anschließend wurde der Modell-Code neu generiert. Eine bildhafte Darstellung der Klasse Filter des Modells kann der untenstehenden Abbildung entnommen werden.

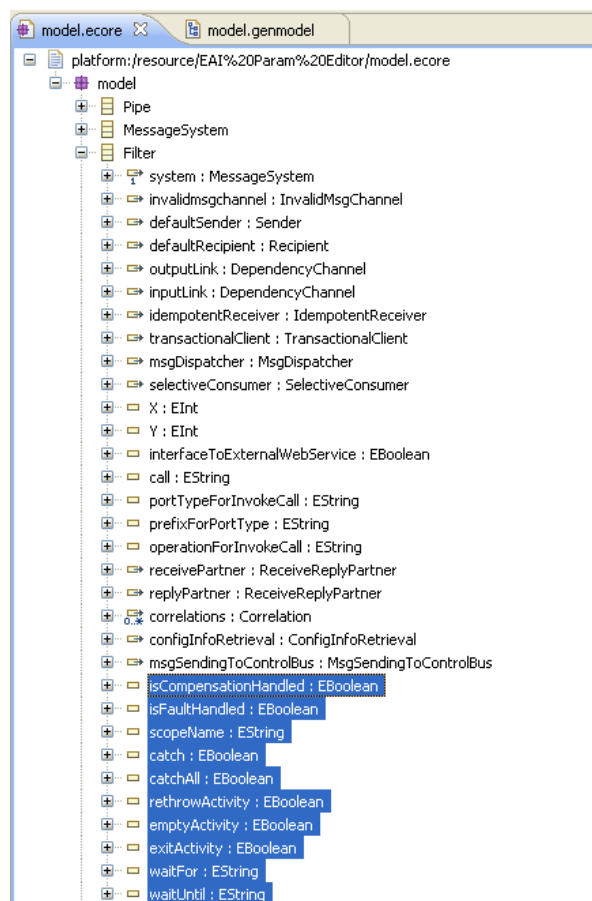


Abbildung 20: Änderungen an der Klasse Filter im Modell

## 6.2.2 Die Klasse CatchFault

In diesem Abschnitt wird beschrieben, wie das Modell um das Pattern Catch Fault, das in Abschnitt 5.2.4 parametrisiert wurde, erweitert wird. Zunächst folgt eine Auflistung der hinzugefügten Attribute und deren Typ:

- faultName vom Typ EString
- faultVariable vom Typ EString
- faultMessageType vom Typ EString
- faultElement vom Typ EString
- rethrowActivity vom Typ EBoolean
- emptyActivity vom Typ EBoolean
- exitActivity vom Typ EBoolean
- waitFor vom Typ EString
- waitUntil vom Typ EString
- X und Y vom Typ EInt für die Koordinaten des Patterns im Editor

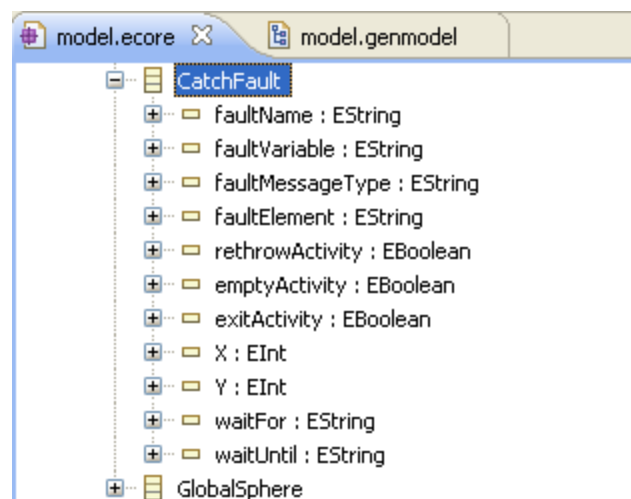


Abbildung 21: Das Modell der Klasse CatchFault

Nach der Erweiterung des Modells um die Klasse CatchFault und die oben genannten Attribute wurde das Modell gespeichert und der Modell-Code neu generiert.

## 6.2.3 Die Klasse GlobalSphere

Die Umsetzung des Patterns Global Sphere, das in Abschnitt 5.6.1 parametrisiert wurde, erforderte folgende Änderungen am bisherigen Modell: Hinzugefügt wurde die Klasse GlobalSphere mit den folgenden Attributen:

- isCompensationHandled vom Typ EBoolean
- isFaultHandled vom Typ EBoolean
- catch vom Typ EBoolean
- catchAll vom Typ EBoolean
- rethrowActivity vom Typ EBoolean
- emptyActivity vom Typ EBoolean
- exitActivity vom Typ EBoolean
- userDefinedCompensation vom Typ EBoolean
- scopeName vom Typ EString
- waitFor vom Typ EString
- waitUntil vom Typ EString
- X und Y vom Typ EInt für die Koordinaten des Patterns im Editor

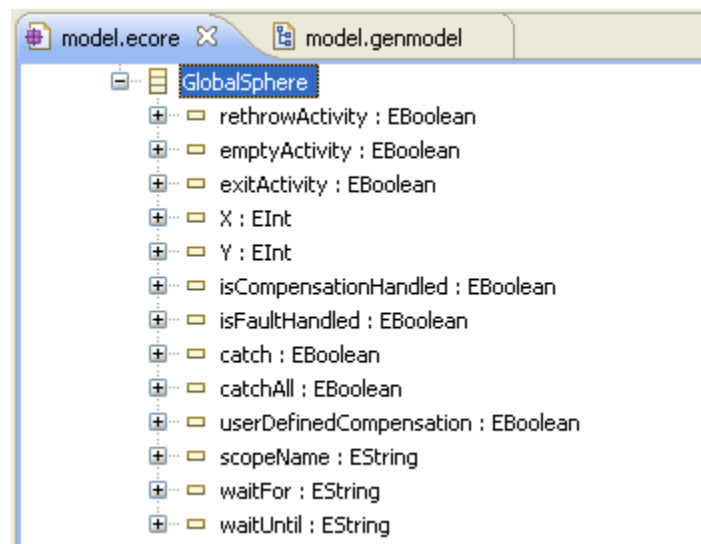


Abbildung 22: Das Modell der Klasse GlobalSphere

Nachdem das Modell erweitert und abgespeichert wurde, wurden die erforderlichen Klassen durch die Neugenerierung des Modell-Codes automatisch erstellt.

### 6.3 Der WSDL Writer

Im bisherigen System gab es syntaktische Fehler der Klasse *WSDLWriter.java* im Paket „*de.unistuttgart.iaas.eaiparam.actions*“, die dazu führten, dass die Operationen, die im WSDL-File spezifiziert wurden, von verschiedenen BPEL Engines (z.B. ActiveVOS [18]) nicht aufgelöst werden konnten.

Dies wurde korrigiert, die WSDL Ausgabe ist nun konsistent mit den Anforderungen an die Lesbarkeit der BPEL Engines. Die Eingriffe in den bisherigen Code wurden durch Kommentare sichtbar gemacht.

## 6.4 Das Szenario

Als Szenario für die Demonstration dient ein Prozess, bei dem eine Kundeninformation über die Reisedaten einer Person durch insgesamt vier External Services geschickt wird, die eine Schnittstelle zu Web Services für eine Hotel- und eine Flugbuchung darstellen und danach die Bezahlung des Hotels und der Airline simulieren. Dieser Prozess wurde mit dem „EAIttoBPEL“-Editor erstellt. Hierfür wurde eine Receive-Reply Schnittstelle für den Start des Prozesses erstellt und vier External Services, die jeweils das Hotel, die Airline, den Kontierungsservice des Hotels und den Kontierungsservice der Airline darstellen, miteinander verbunden. In den Konfigurationsmenüs der External Services wurde jeweils Kompensation ausgewählt und eine Kompensationsoperation angegeben. Eine bildhafte Darstellung dieses Szenarios kann der unten stehenden Abbildung 23 entnommen werden.

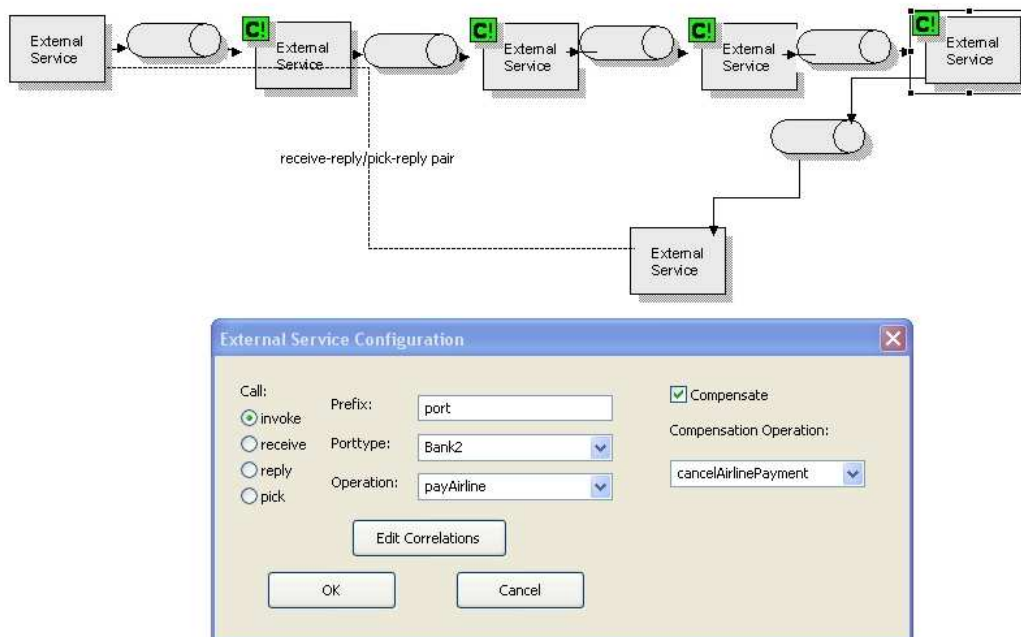


Abbildung 23: Das mit dem EAIttoBPEL-Editor erstellte Szenario

Nachdem alle notwendigen Einstellungen in den Konfigurationsdialogen der External Services und der Datatype-Channels, sowie im Messaging System selbst vorgenommen sind, lässt sich das Szenario abspeichern und der BPEL und WSDL Code generieren. Die generierte BPEL Datei befindet sich in Anhang A1. Die generierte WSDL Datei enthält den portType der Receive-Reply Schnittstelle und die PartnerLinkTypes der External Services und der Receive-Reply Schnittstelle. Eine Abbildung dieser

Datei befindet sich in Anhang A3. Die Messages und die portTypes der ES müssen der WSDL Datei von Hand hinzugefügt werden.

Nachdem dies geschehen ist kann die Ausführung des Szenarios simuliert werden. In den dieser Diplomarbeit vorangegangenen Arbeiten wurde die frei erhältliche Open-Source BPEL-Engine „ActiveBPEL“ von ActiveEndpoints verwendet. Leider ist diese BPEL-Engine heute nicht mehr frei erhältlich. Es handelt sich hierbei mittlerweile um ein kommerzielles Programm mit dem Namen „ActiveVOS“. Jedoch ist diese Version zu Testzwecken nach einer Registrierung auf der Internetseite von ActiveEndpoints [18] erhältlich und man bekommt eine einmonatige Lizenz zur Verfügung gestellt.

#### **6.4.1 Simulation des Szenarios**

Nach der Installation des Eclipse-basierten *Visual Orchestration Studios* „ActiveVOS“ [18], wurde die generierte BPEL Datei *itinerary.bpel* und die generierte WSDL Datei *itinerary.wsdl* (siehe Anhang) in ein zuvor mit „ActiveVOS“ erstelltes Orchestration Project importiert. Hierbei wurde die generierte BPEL Datei vom Originalzustand durch „ActiveVOS“ automatisch geringfügig verändert. Diese Änderungen sind wie folgt:

- Vor jedes Element wurde das Prefix „*bpel:*“ gesetzt.
- Die Reihenfolgen innerhalb der *<receive>*, *<reply>* und *<invoke>* Aktivitäten wurden geändert.
- Das *portType* Element innerhalb der oben genannten Aktivitäten wurde entfernt.

Manuelle Anpassungen an der generierten BPEL Datei wurden nicht vorgenommen. Eine Abbildung der durch „ActiveVOS“ geänderten Datei befindet sich in Anhang A2.

Um das Szenario zu simulieren, mussten von Hand die Message- und *portType* Elemente in die generierte WSDL Datei eingefügt werden. Hierbei wurde ein syntaktischer Fehler des WSDL Writers des Systems „EAltoBPEL“ entdeckt und behoben.

Die Simulation des Szenarios kann ohne die Binding-Informationen und die Web Services Endpoints erfolgen und simuliert nach Eingabe des erforderlichen Parameters eine Ausführung, um eventuelle Fehler in der BPEL Repräsentation des Prozesses aufzudecken. Aus diesen Gründen ist eine Simulation des Prozesses sinnvoll, bevor man das Binding unternimmt. Die Simulation des Prozesses ergab keine Fehler, der Prozess beendete, ohne ein Fehlverhalten anzuzeigen.

#### **6.4.2 Vorbereitungen für die Ausführung des Szenarios**

Nachdem die Korrektheit des BPEL-Prozesses durch die Simulation überprüft und sichergestellt wurde, kann mit dem Binding des Prozesses an konkrete Endpoints begonnen werden. Hierfür müssen zunächst vier Web Services implementiert werden, die die Schnittstellen des Prozesses mit dem Service des Hotels, der Airline und deren Kontierungsservices Bank und Bank2 darstellen.

Um die Web Services zu implementieren wurde, das Eclipse-Plugin „Web Tools Platform Project“ [22] verwendet. Es ist auch bereits in „ActiveVOS“ enthalten. Um die Web Services zu implementieren und aufzusetzen, muss zunächst ein dynamisches Web Projekt gestartet werden. Nun können die Klassen der Web Services implementiert werden. Eine Darstellung des Web Services „Bank2“, der den Kontierungsservice der Airline darstellt, kann Abbildung 24 entnommen werden. Da die anderen Web Services von ihrer Struktur her identisch sind, wird auf deren Abbildung hier verzichtet.

---

```
public class Bank2 {

 public String payAirline(String customerInfo) {
 System.out.println("Airline payment successfully ");
 customerInfo= customerInfo +" Airline payed";
 return customerInfo;
 }

 public void cancelAirlinePayment (String customerInfo) {
 System.out.println("Airline payment successfully cancelled");
 }

}
```

---

Abbildung 24 : Die Web Service Implementierung des Ports Bank2

Mit diesem und den anderen Web Services wird die Schnittstelle der Primären Aktivität (in Abbildung 24: *payAirline*) und der Kompensationsaktion (in Abbildung 24: *cancelAirlinePayment*) definiert. Durch die Ausgabe der jeweiligen Nachricht auf der Konsole, kann der Zustand und Fortschritt des Prozesses dort später abgelesen werden.

Nachdem die vier Web Service Klassen implementiert sind, können automatisch die für die Interaktion wichtigen weiteren Klassen und die WSDL Dateien generiert werden. Hierzu klickt man im Navigator mit der rechten Maustaste auf die jeweilige Web Service Klasse und wählt im Menü: New -> Other -> Web Services -> Web Service und bestätigt mit Next. Ein Menü erscheint zur Generierung eines „Bottom up Java bean Web Service“. Im unteren Teil des Menüs kann durch Verschieben eines Reglers auch gleich ein Test Client mit generiert werden, mit dem man die Funktionalität des Web Services testen kann. Als Web Server für die Web Services wurde Apache Tomcat 5.5.27 [23] verwendet.

Durch Bestätigen von Next kann nun der Binding-Style gewählt werden. Es wurde RPC/encoded ausgewählt. Durch Betätigung des Finish Buttons werden nun alle benötigten Klassen und die WSDL automatisch generiert und der Web Service wird auf dem Tomcat Web Server aufgesetzt. Mit Hilfe des Test Clients kann dieser nun auf seine korrekte Funktion überprüft werden. Abbildung 25 zeigt das Menü zur Generierung des „Bottom up Java bean Web Service“.

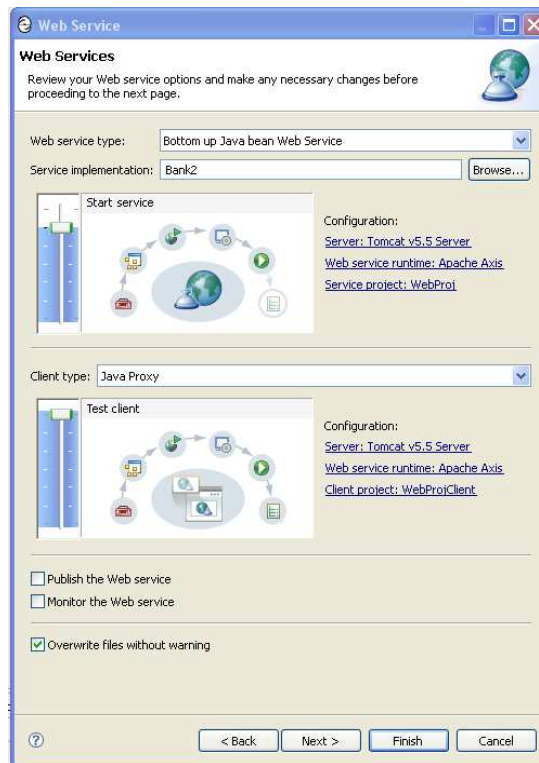


Abbildung 25: Das Menü zur Generierung der Web Services

Nachdem auf oben beschriebene Art und Weise alle 4 Web Services generiert und aufgesetzt wurden, müssen die Binding- und Service Informationen der WSDL Dateien dieser Web Services in die vom „EAltoBPEL“ Editor generierte „*itinerary.wsdl*“ Datei umkopiert werden. Die hierbei entstandene WSDL Datei kann in Anhang A4 nachgeschlagen werden. Ist dies geschehen, so kann der Process Deployment Descriptor für das Szenario erstellt werden.

Dies erfolgt durch einen Rechtsklick auf das Orchestration Projekt, in dem sich die BPEL und WSDL Datei befindet. Im Menü wird nun New -> Deployment Descriptor gewählt, die BPEL Datei ausgewählt und auf Finish geklickt.

Unter „General“ wurde Process Persistence -> Full ausgewählt, als Referenz wurde die WSDL Datei des Szenarios übergeben. Nun müssen unter „Partner Links“ die Endpoints für die Web Services eingegeben werden. Als *Invoke Handler* wurde „WSA Address“ und als *Endpoint Type* „static“ ausgewählt. Die Receive-Reply Schnittstelle wurde mit dem Binding „RPC Literal“ versehen. Der Source Code der fertigen *itinerary.pdd* Datei kann in Anhang A5 nachgeschlagen werden.

Nun sind alle Voraussetzungen erfüllt, um das Szenario auf der ActiveBPEL Engine zu deployen. Zuvor muss allerdings der ActiveVOS Server gestartet und konfiguriert werden. Dies kann unter „Servers“ New -> Server -> Active Endpoints, Inc. ->ActiveVOS Server erfolgen. Im Konfigurationsmenü kann auch die Adresse des Servers geändert werden. Für den Server des BPEL Prozesses des Szenarios, wurde die Adresse „localhost:8090“ ausgewählt.

Um den Prozess auf die BPEL Engine zu deployen, wird auf dem Orchestration Project ein Rechtsklick ausgeführt. Im erscheinenden Menü wird Export -> Orchestration -> Business Process Archive File gewählt, mit Next bestätigt und ein Menü erscheint, in dem die Adresse der BPEL Engine, der Deployment Typ und der Speicherort der „*itinerary.bpr*“ Datei angegeben werden müssen. Abbildung 26 zeigt das Menü für die Erstellung des BPR Files.

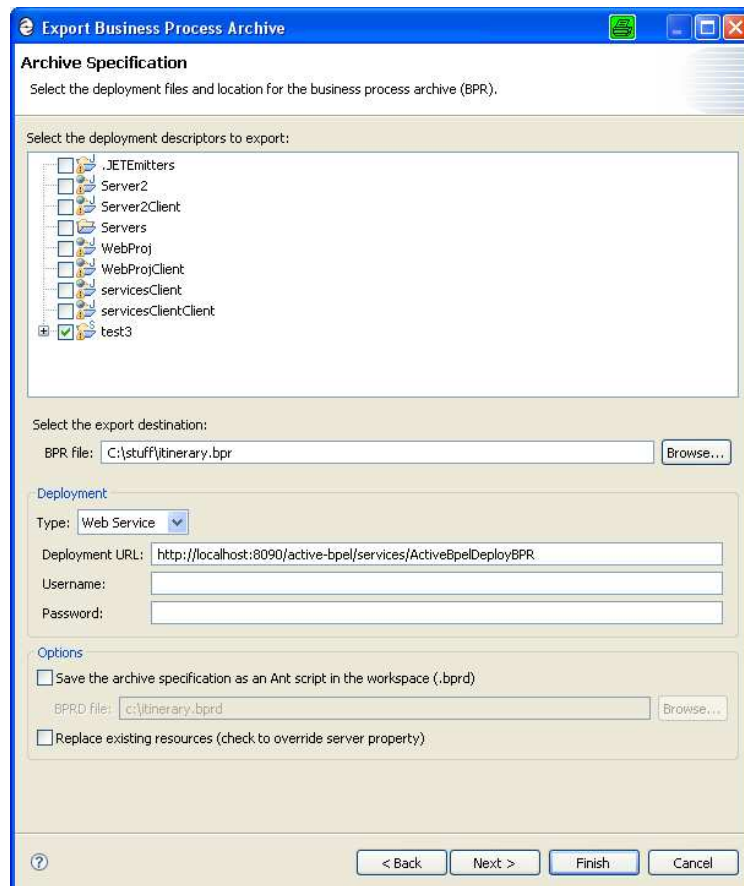


Abbildung 26: Das Menü für die Erstellung und das Deployment des BPR Files

### 6.4.3 Ausführung des Szenarios

Nun ist der BPEL Prozess des Szenarios deployed und kann ausgeführt werden. Wenn der ActiveVOS Server gestartet ist, kann im Webbrowser unter der Adresse:

„[http://localhost:8090/activevos/deployed\\_processes.action](http://localhost:8090/activevos/deployed_processes.action)“

der aufgesetzte Prozess und seine Details abgerufen werden. Abbildung 27 veranschaulicht diese Oberfläche.

## Deployed Process Version Detail

|                          |                         |
|--------------------------|-------------------------|
| <b>Name:</b>             | es-test                 |
| <b>Target Namespace:</b> | http://example.com/bpel |
| <b>Process Group:</b>    |                         |
| <b>Version:</b>          | 9,0                     |

[View Process Graph](#)

Version Detail
BPEL

**Migrated To:**

**Plan Id:** 20

**Deployed Date:** 2009/03/24 08:13 PM

**Effective Date:** 2009/03/24 08:13 PM

**Expiration Date:**  (yyyy/mm/dd hh:mm AM/PM)

**Running Process Disposition:** Maintain

**Status:** Current

**Logging Level:** system default

**Persistence Type:** full

**Exception Management Type:** engine

**Invoke Recovery Type:** engine

**Process Instance Retention Days:**

---

**My Role**

| Partner Link | Type             | Role    | Binding | Service             | Policy |
|--------------|------------------|---------|---------|---------------------|--------|
| PartnerLink5 | PartnerLinkType5 | MyRole5 | RPC-LIT | PartnerLink5Service |        |

**Partner Role**

| Partner Link | Type             | Role         | Linkage |
|--------------|------------------|--------------|---------|
| PartnerLink1 | PartnerLinkType1 | PartnerRole1 | static  |
| PartnerLink2 | PartnerLinkType2 | PartnerRole2 | static  |
| PartnerLink3 | PartnerLinkType3 | PartnerRole3 | static  |
| PartnerLink4 | PartnerLinkType4 | PartnerRole4 | static  |

**Indexed Properties**

(None)

---

**Resource Usage**

| Resource       | Target Namespace        |
|----------------|-------------------------|
| itinerary.wsdl | http://example.com/wsdl |

Abbildung 27: Oberfläche für Prozess Version und Details der ActiveBPEL Engine

Um den Prozess auszuführen, muss nun zunächst auf den Service „PartnerLink5Service“ geklickt werden. Die WSDL Datei dieser Schnittstelle erscheint im Browser und die Adresse muss kopiert werden.

Nun startet man über das Run Menü im „ActiveVOS“ den Web Service Explorer, wählt WSDL Main und fügt diese kopierte Adresse ein. Daraufhin wird eine Verbindung zum Prozess hergestellt und durch die Eingabe des Parameters *customerInfo* und die Bestätigung mit „Go“, wird der Prozess gestartet.

## Prozessausführung ohne Fehler

Zunächst wird der Prozess ausgeführt, ohne einen Fehlerfall herbeizuführen. Für die Ausführung des Prozesses wurden insgesamt drei Web Server verwendet. Der BPEL-Prozess selbst wurde auf dem ActiveVOS Server aufgesetzt, die Adresse ist „localhost:8090“. Die drei Web Services „PortHotel“, „Port Airline“ und „Bank“ wurden auf dem Apache Tomcat Web Server 5.5.27 deployed. Die Adresse

dieses Web Servers lautet „localhost:8080“. Die Web Service Schnittstelle „Bank2“ wurde auf einer zweiten Instanz des Tomcat Web Servers aufgesetzt, die Adresse dieser Instanz ist „localhost:8081“. Bevor der Prozess ausgeführt werden kann, müssen alle drei Web Server hochgefahren werden, nun kann der Prozess wie oben beschrieben gestartet werden.

Im Erfolgsfall des Prozesses sollen nun also auf der Konsole des Tomcat Servers mit der Adresse „localhost:8080“ die Meldungen „Hotel booked successfully“, „Flight booked successfully“ und „Hotel payment successfully“ ausgegeben werden. Auf der Konsole des zweiten Tomcat Servers, mit der Adresse „localhost:8081“, auf dem der Kontierungsservice für die Airline aufgesetzt ist, erscheint dann die Meldung „Airline payment successfully“.

Die SOAP Nachricht wird von den Web Services im Feld „customerInfo“ zur besseren Anschauung mit Einträgen der jeweiligen Web Services erweitert. Das Ergebnis der Ausführung und die SOAP Nachricht ist in der unten aufgeführten Abbildung 28 zu sehen.

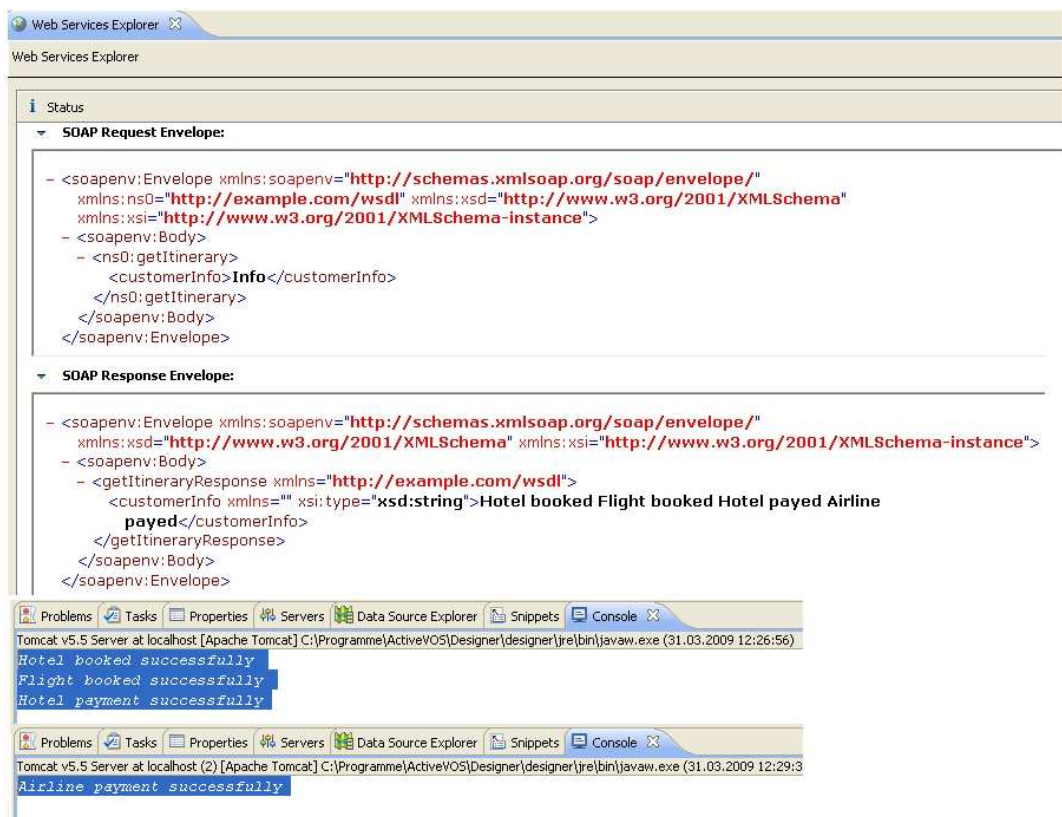


Abbildung 28: Ergebnisse der erfolgreichen Prozessausführung

## Prozessausführung mit Ausfall des Web Service Bank2

Um einen Fehlerfall im Prozess zu erzwingen, wird nun die zweite Instanz des Tomcat Servers mit dem Kontierungsservice der Airline „Bank2“ vor dem erneuten Starten des Prozesses heruntergefahren. Der Web Service „Bank2“ kann nun also vom Prozess nicht mehr erreicht werden.

Da der BPEL-Prozess „Itinerary“ durch den fehlerhaften Kontaktversuch mit dem Web Service „Bank2“ in eine Fehlersituation gerät, müssen die Buchungen und der Geldtransfer, der erfolgt ist, kompensiert werden. Hierzu muss der Prozess den Fehler mit seinem impliziten FH auffangen und

mit Standardkompensation die bisher erfolgreich abgeschlossenen Arbeitsschritte „bookHotel“, „bookFlight“ und „payHotel“ rückgängig machen.

Dies erfolgt durch die Ausführung der CH der jeweiligen <invoke> Aktivitäten des BPEL Prozesses und die damit verbundene Initiierung der Kompensationsoperationen „cancelHotel“, „cancelFlight“ und „cancelHotelPayment“ bei den jeweiligen Web Services. Entsprechende Mitteilungen werden auf der Konsole angezeigt, in der SOAP Nachricht wird eine Fehlermeldung ausgegeben. Die Ergebnisse der fehlerhaften Prozessausführung können der unten stehenden Abbildung 29 entnommen werden. Eine Abbildung der Logdatei des fehlgeschlagenen Prozesses und eine grafische Abbildung des Prozesses im Fehlerzustand können dem Anhang A6 und A7 entnommen werden.

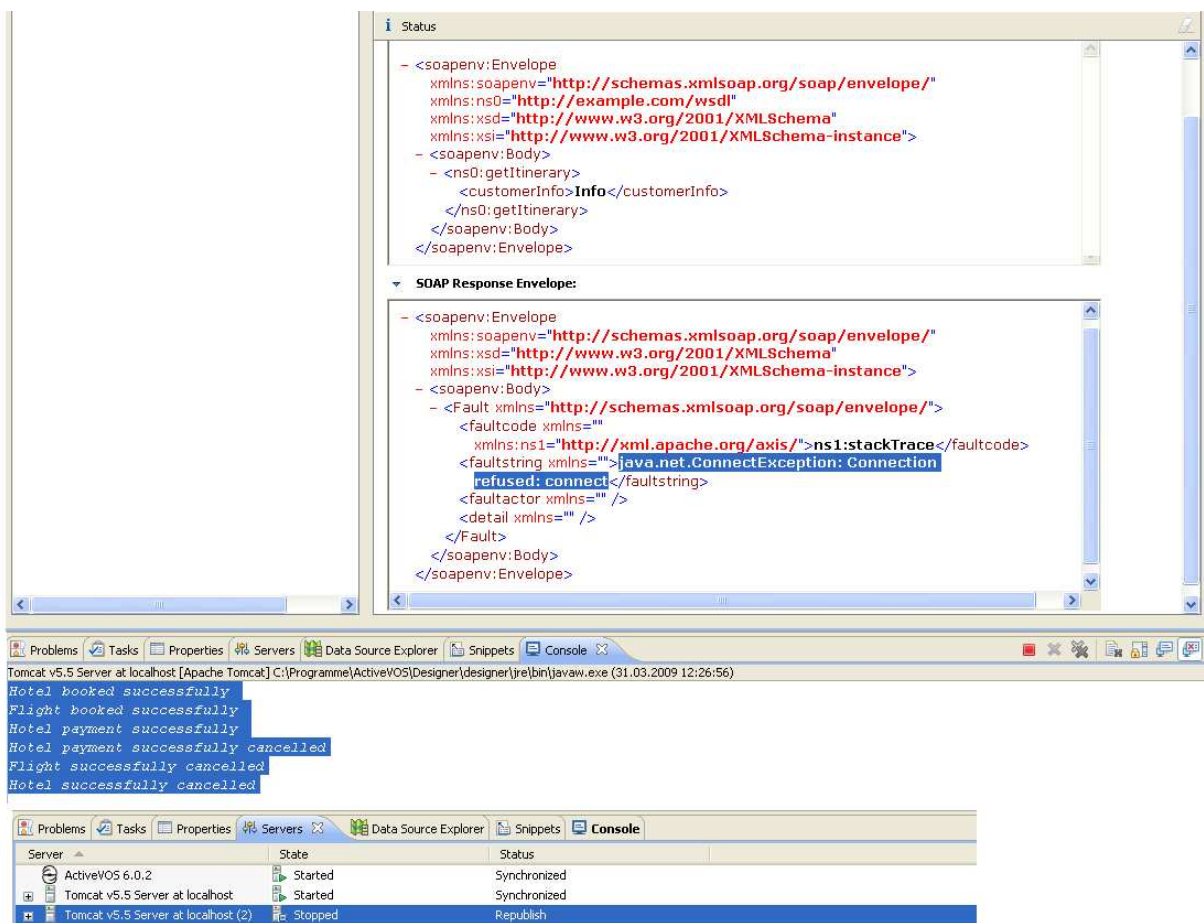


Abbildung 29: Kompensierter Prozess nach Serverausfall

## 6.5 Checkliste für Entwickler

Dieser Abschnitt richtet sich an Entwickler, die zukünftig mit dem System „EAltoBPEL“ arbeiten. Die folgenden Arbeitsschritte sind wichtig, wenn ein neues Pattern dem System hinzugefügt oder ein noch nicht implementiertes, im Modell vorhandenes, Pattern umgesetzt werden soll:

- Das Modell muss erweitert werden und der Modell-Code anschließend neu generiert werden (falls das Pattern noch nicht im Modell vorhanden ist).
- Das Icon muss der Klasse *EAIPaletteRoot* hinzugefügt werden, damit es im Menü des Editors ausgewählt werden kann.
- Die Klasse *ModelCreationFactory* muss erweitert werden.
- Die Klassen *EditAction* und *EAIEditPartFactory* müssen um das Pattern erweitert werden.
- Die zugehörige *EditPart*- Klasse muss in das Paket „*de.unistuttgart.iaas.eaiparam.-editors.editParts*“ aufgenommen werden.
- Ein Bild des Patterns kann angefertigt werden und in der oben genannten Klasse eingefügt werden.
- Der Konfigurationsdialog des Patterns muss in das Paket „*de.unistuttgart.iaas.eaiparam.-editors.dialogs*“ aufgenommen werden.
- Create, Delete und Edit Kommandos müssen in die zugehörigen Klassen in das Paket „*de.unistuttgart.iaas.eaiparam.editors.editParts.commands*“ aufgenommen werden.
- Die Klasse *MsgSysXYLayoutEditPolicy* muss erweitert werden.
- Die Klassen *NodeComponentEditPolicy* und *NodeEditPolicy* müssen eventuell modifiziert werden.
- Die Pattern Klasse muss in dem passenden Unterpaket des Pakets „*de.unistuttgart.iaas.framework.xform2bpel.patterns*“ eingefügt werden.
- Die Klasse *EAIResourceImpl* muss angepasst werden, da das Pattern korrekt gespeichert werden muss, bevor der BPEL Code generiert werden kann.
- Die Klasse *BPELWriter* muss modifiziert werden.
- Die Klasse *MessageSystemModelManager* sollte angepasst werden, hier findet die Überprüfung des Patterns auf Fehleingaben statt.

## 6.6 Ausblick

Der Grundbaustein für die Umsetzung von Dienstgüteeigenschaften für das System „EAltoBPEL“ ist gesetzt. Jedoch handelt es sich hierbei um einen Prototypen, bei dem es noch einige offene Baustellen gibt:

- Die, für die Umsetzung von Dienstgüteeigenschaften notwendigen, Parameter wurden identifiziert und dem Modell hinzugefügt.
- Direkte Kompensation des <invoke> Aufrufs des External Service Patterns wurde umgesetzt und ist lauffähig.
- Zwei neue Patterns wurden identifiziert, parametrisiert und dem Modell hinzugefügt. Diese Patterns wurden aus Zeitgründen leider noch nicht implementiert. Hier gibt es noch Arbeit zu erledigen.
- Der WSDL Writer des Systems wurde korrigiert. Allerdings kann der Import von Namespaces noch nicht ohne manuelles Eingreifen erfolgen.
- Die Klasse BPELWriter ist mit über 1200 Zeilen Code sehr unübersichtlich und dadurch nur sehr schwer zu warten. Erweiterungen an dieser Klasse sollten durch eine deutliche Kommentierung gekennzeichnet werden. Es sollten Überlegungen angestellt werden, diese Klasse neu zu strukturieren, um sie zu verschlanken.
- Um die neuen Patterns zu verwenden, muss die Struktur des Editors verändert werden. Es muss gestattet sein, mehrere ausgehende Kanäle an einzelne Patterns anzulegen, in denen die Fehlerbehandlungs- und die Kompensationsaktionen modelliert werden können.
- Es würde die Bedienung des Editors erleichtern, wenn es eine „Refresh“ Funktion der Anzeige gäbe. Dadurch würden kompensierte ES sofort angezeigt werden und Fehler im Problems-Tab würden nicht erst bei der Speicherung des Messaging Systems entdeckt werden.
- Die Fehler im Messaging System wären leichter zuzuordnen, wenn der Fehlerort im Editor zusätzlich grafisch angezeigt werden würde.
- Hilfreich für weitere Entwicklungsaufgaben am System „EAltoBPEL“ ist eine saubere Kommentierung der hinzugefügten Änderungen, da das System sonst droht, zu unübersichtlich zu werden.
- Viele der durch [1] und [2] parametrisierten Patterns wurden noch nicht implementiert.

Einige weitere offene Baustellen wurden in [1] aufgeführt und auch darüber hinaus sind noch viele weitere Punkte denkbar, an denen gearbeitet werden kann.

## 7 Zusammenfassung

Das Ziel dieser Diplomarbeit bestand darin, zu untersuchen, inwieweit Dienstgüteeigenschaften bei der Parametrisierung der EAI Patterns und der anschließenden Generierung eingebunden werden können. Hierfür wurden zunächst die Fehlerbehandlungs- und transaktionalen Konzepte von WS-BPEL und Workflow Systemen, sowie die Web Service Spezifikationen der Quality of Service Schicht des SOA Stacks zusammengefasst und beschrieben.

Im theoretischen Teil dieser Diplomarbeit erfolgte zunächst eine Analyse der möglichen Fehlersituationen in einer PaF-Architektur. Hierbei wurden verschiedene Fehlerklassen herausgearbeitet und Überlegungen angestrebt, wie in diesen Situationen vorgegangen werden muss, um den jeweiligen Fehlern entgegen zu wirken. Dabei wurden verschiedene Fehlerbehandlungsstrategien aufgeführt und es wurde überlegt, wie diese in einer PaF-Architektur umgesetzt werden können.

Die Ergebnisse dieser Fehleranalyse zeigten, dass in einer PaF-Architektur eine Fehlerbehandlung nicht ausreichend umgesetzt werden kann. Die Umsetzung von bestehenden, transaktionalen Konzepten ist, auf Grund des Fehlens eines Instanz-Begriffs, nicht möglich. Eine PaF-Architektur ist für die Anforderungen, die eine Produktionsumgebung stellt, daher nicht geeignet.

Die Abbildung der EIP auf BPEL führt ein globales Konzept für die Fehlerbehandlung und Transaktionen ein. Nach der theoretischen Analyse der möglichen Fehlersituationen in einer PaF-Architektur wurden die ermittelten Fehlerbehandlungsstrategien hinsichtlich einer Abbildung der EIP auf BPEL untersucht. Hierbei wurden neue Parameter herausgearbeitet, die für die Abbildung des Fehlerbehandlungs- und transaktionalen Konzepts von WS-BPEL auf die parametrisierten EIP notwendig sind. Zusätzlich wurden zwei neue Patterns identifiziert und parametrisiert, die für die Umsetzung dieser Konzepte benötigt werden. Das Modell des Editors „EAltoBPEL“ wurde geändert und die identifizierten Parameter und die neuen Pattern Klassen hinzugefügt.

Das System „EAltoBPEL“ ist ein Eclipse-basierter Editor, der von Florian Schebelle und Bettina Druckenmüller [1] implementiert wurde und genutzt werden kann, um EAI Messaging Systeme zu erstellen und automatisch BPEL Code daraus zu generieren. Das bestehende System wurde erweitert und mit dem Editor ein Szenario erstellt, dass das kompensationsbasierte Transaktionskonzept von WS-BPEL in das External Service Pattern integriert. Um die korrekte Funktionsfähigkeit des generierten BPEL Codes zu veranschaulichen, wurde der hierbei erstellte BPEL Prozess auf der BPEL Engine von ActiveVOS [18] aufgesetzt und ausgeführt. Der Prozess ist ohne manuelles Eingreifen in den generierten BPEL Code lauffähig.

Es sind jedoch weitere Arbeiten notwendig, um das System „EAltoBPEL“ fertig zu stellen. Diese Arbeiten können hierbei in verschiedene Richtungen gehen. Einige der Patterns von G. Hohpe [3] sind noch nicht parametrisiert worden und die generierte WSDL Datei ist noch nicht ohne manuelles Eingreifen lauffähig.

Das System „EAltoBPEL“ ist ein Prototyp und hat weiterhin viele offene Baustellen. Die meisten Patterns sind zwar parametrisiert, nur wenige von ihnen aber tatsächlich implementiert. Um das Fehlerbehandlungs- und das transaktionale Konzept von WS-BPEL vollständig auf die EIP abbilden zu können, muss die Struktur des Editors angepasst werden. Die, für eine Umsetzung der

Fehlerbehandlung notwendigen, neu identifizierten Patterns sind parametrisiert worden und wurden dem Modell des Systems hinzugefügt. Aus Zeitgründen ist eine komplette Implementierung dieser Patterns aber leider nicht erfolgt.

## Literaturverzeichnis

- [1] Druckenmueller, B. (Februar 2007). Diplomarbeit. *Parameterisierung von EAI Patterns*. Universität Stuttgart.
- [2] Yuan, X. (Januar 2008). Diplomarbeit. *Prototype for Executable EAI Patterns*. Universität Stuttgart.
- [3] Gregor Hohpe, B. W. (11th Printing Januar 2008). *Enterprise Integration Patterns*. Addison-Wesley.
- [4] F. Leymann, T. S., et al. (2005). *Web Services Platform Architecture*. U.S.A.: Pearson Education, Inc.
- [5] F. Leymann, D. Roller (2000). *Production Workflow*. USA: Prentice Hall, PTR.
- [6] A. Alves, et al. (2007, April 11). <http://docs.oasis-open.org/>. Retrieved 04 15, 2009, from OASIS WS-BPEL: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>
- [7] Thränert, M. (2004). *eBusiness I*. Abgerufen am 15. 04 2009 von Erweiterte WS-Standards: [www.bis.uni-leipzig.de/studium/vorlesungen/2004\\_ss/eb1/2004s\\_eb1\\_v\\_06a\\_1.pdf](http://www.bis.uni-leipzig.de/studium/vorlesungen/2004_ss/eb1/2004s_eb1_v_06a_1.pdf)
- [8] P. V. Biron, A. M. (2004, 10 28). *XML Schema Part 2: Datatypes Second Edition*. Retrieved 04 15, 2009, from W3C Recommendation 28 October 2004: <http://www.w3.org/TR/xmlschema-2/#NCName>
- [9] F. Leymann. (2008). *Workflow Management 2009, BPEL*. Universität Stuttgart, Baden-Württemberg, Deutschland.
- [10] Leymann, F. (2008). *Message-basierte Anwendungsintegration 08/09*. Universität Stuttgart, Baden-Württemberg, Deutschland.
- [11] Hohpe, G. (2004, 04 20). [www.eaipatterns.com](http://www.eaipatterns.com). Retrieved 04 15, 2009, from Enterprise Integration Patterns Presentation Downloads: [www.eaipatterns.com](http://www.eaipatterns.com)
- [12] T. Scheibler, F. L. (2008). *A Framework for Executable Enterprise Application Integration Patterns*. Universität Stuttgart, Deutschland.
- [13] Hohpe, G. (2003). *Enterprise Integration Patterns - Asynchronous Messaging Architectures in Practice*. Retrieved 15 04, 2009, from [www.enterpriseintegrationpatterns.com/docs/hohpeg\\_enterpriseintegrationpatterns\\_sdbp.pdf](http://www.enterpriseintegrationpatterns.com/docs/hohpeg_enterpriseintegrationpatterns_sdbp.pdf)
- [14] Hohpe, G. (2005). *Your Coffee Shop Doesn't Use Two-Phase Commit*. *IEEE Software Design 2PC*, p. 3.
- [15] Hohpe, G. (2004, November 15). *Enterprise Integration Patterns Episode II*. Retrieved 04 15, 2009, from Tokyo Patterns Group Part2: [patterns-wg.fuka.info.waseda.ac.jp/event/ei2004/gregor-hohpe2.pdf](http://patterns-wg.fuka.info.waseda.ac.jp/event/ei2004/gregor-hohpe2.pdf)
- [16] Trautvetter, J. (31. August 2006). Diplomarbeit. *Analyse der "Pipes and Filter" Architektur gegenüber instanzbasierten Ansätzen bei Workflows*. Universität Stuttgart.

[17] T.Scheibler, G. H. (2008, 12 18). Message-basierte Anwendungsintegration Übung 5. Universität Stuttgart, B.-W., Deutschland.

[18] active endpoints. (2009). *ActiveVOS*. Retrieved März 15, 2009, from ActiveVOS:  
<http://www.activevos.com/>

[19] Tony Fletcher, A. G. (2003). *BPEL and Business Transaction Management: Choreology Submission to OASIS WS-BPEL Technical Committee*. Retrieved April 15, 2009, from <http://www.oasis-open.org>:  
<http://www.oasis-open.org/committees/download.php/3263/BPEL.and.Business.Transaction.Management.Choreology.Submission.html>

[20] Green, A. (2003, September). Transacting Business with Web Services Part1. *Web Services Journal* , p. 4.

[21] Green, A. (2003, September). Transacting Business with Web Services Part2. *Web Services Journal* , p. 8.

[22] *Web Tools Platform (WTP) Project*. (2009, März 18). Retrieved April 15, 2009, from Eclipse.org:  
<http://www.eclipse.org/webtools/>

[23] *Apache Tomcat*. (2009). Retrieved April 15, 2009, from The Apache Software Foundation:  
<http://tomcat.apache.org/download-55.cgi>

## Abbildungsverzeichnis

|                                                                                              |    |
|----------------------------------------------------------------------------------------------|----|
| Abbildung 1: Pipeline Verarbeitung mit Pipes und Filters (Quelle [3]) .....                  | 13 |
| Abbildung 2: EIP bei der Integration von Applikationen (Quelle [11]).....                    | 16 |
| Abbildung 3: Das Pattern Control Bus und die Management Konsole (Quelle [13]) .....          | 16 |
| Abbildung 4 : Grafische Notation eines Patterns .....                                        | 20 |
| Abbildung 5: Der SOA Stack.....                                                              | 22 |
| Abbildung 6: WS-BPEL und WS-Transaction (Quelle [7]) .....                                   | 28 |
| Abbildung 7: "caught exceptions" - Fehlernamen für Routing Entscheidungen (Quelle [9]) ..... | 31 |
| Abbildung 8: Koordinierte Transaktion mit 3 Teilnehmern (Quelle [15]) .....                  | 47 |
| Abbildung 9: Kompensationsaktion (Quelle [15]) .....                                         | 50 |
| Abbildung 10: Ausschnitt einer Kompensationsaktion für einen einfachen Prozess.....          | 51 |
| Abbildung 11: Retry mit redundanten Komponenten .....                                        | 54 |
| Abbildung 12: Correction Loop (Quelle [17]) .....                                            | 56 |
| Abbildung 13: CLIENT_ACKNOWLEDGE Modus in JMS .....                                          | 58 |
| Abbildung 14: Parametrisierung des Catch Fault Patterns .....                                | 71 |
| Abbildung 15: Parametrisierung der Global Sphere.....                                        | 75 |
| Abbildung 16: Änderungen am Model des ES.....                                                | 77 |
| Abbildung 17 : ES Dialogfenster mit Kompensation .....                                       | 78 |
| Abbildung 18 : Anpassung des Model Managers für Kompensation .....                           | 79 |
| Abbildung 19: Kennzeichnung kompensierter ES .....                                           | 80 |
| Abbildung 20: Änderungen an der Klasse Filter im Modell.....                                 | 81 |
| Abbildung 21: Das Modell der Klasse CatchFault .....                                         | 82 |
| Abbildung 22: Das Modell der Klasse GlobalSphere .....                                       | 83 |
| Abbildung 23: Das mit dem EAltoBPEL-Editor erstellte Szenario.....                           | 84 |
| Abbildung 24 : Die Web Service Implementierung des Ports Bank2 .....                         | 86 |
| Abbildung 25: Das Menü zur Generierung der Web Services .....                                | 87 |
| Abbildung 26: Das Menu für die Erstellung und das Deployment des BPR Files .....             | 88 |
| Abbildung 27: Oberfläche für Prozess Version und Details der ActiveBPEL Engine.....          | 89 |
| Abbildung 28: Ergebnisse der erfolgreichen Prozessausführung.....                            | 90 |
| Abbildung 29: Kompensierter Prozess nach Serverausfall.....                                  | 91 |

# Anhang A

## A.1 Die generierte BPEL Datei

### Itinerary.bpel

---

```
<process name="es-test" targetNamespace="http://example.com/bpel"
 xmlns:msg="http://example.com/message"
 xmlns:port="http://example.com/port"
 xmlns:home="http://example.com/wSDL"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

 <partnerLinks>
 <partnerLink name="PartnerLink1" partnerLinkType="home:PartnerLinkType1"
 partnerRole="PartnerRole1" />
 <partnerLink name="PartnerLink2" partnerLinkType="home:PartnerLinkType2"
 partnerRole="PartnerRole2" />
 <partnerLink name="PartnerLink3" partnerLinkType="home:PartnerLinkType3"
 partnerRole="PartnerRole3" />
 <partnerLink name="PartnerLink4" partnerLinkType="home:PartnerLinkType4"
 partnerRole="PartnerRole4" />
 <partnerLink name="PartnerLink5" partnerLinkType="home:PartnerLinkType5"
 myRole="MyRole5" />
 </partnerLinks>

 <variables>
 <variable name="Variable1" messageType="msg:itinerary" />
 <variable name="Variable2" messageType="msg:itinerary" />
 <variable name="Variable3" messageType="msg:itinerary" />
 <variable name="Variable4" messageType="msg:itinerary" />
 <variable name="Variable5" messageType="msg:itinerary" />
 </variables>

 <flow name="Flow1" suppressJoinFailure="yes">
 <links>
 <link name="Link0" />
 <link name="Link1" />
 <link name="Link2" />
 <link name="Link3" />
 <link name="Link4" />
 </links>

 <sequence name="ExternalService1" >
 <sources>
 <source linkName="Link0" />
 </sources>

 <receive partnerLink="PartnerLink5" portType="home:customerPT"
 operation="getItinerary" variable="Variable4" createInstance="yes" name="Receive1" >
 <correlations>
 </correlations>
 </receive>
 </sequence>
 </flow>
</process>
```

```

</sequence>
<sequence name="ExternalService2" >
 <targets>
 <target linkName="Link0" />
 </targets>
 <sources>
 <source linkName="Link1" />
 </sources>
 <invoke partnerLink="PartnerLink1" portType="port:PortHotel"
 operation="bookHotel" inputVariable="Variable4" outputVariable="Variable1"
 name="Invoke2" >
 <correlations>
 </correlations>
 <compensationHandler>
 <invoke partnerLink="PartnerLink1" portType="port:PortHotel"
 operation="cancelHotel" inputVariable="Variable1" name="Invoke1" >
 </invoke>
 </compensationHandler>
 </invoke>
</sequence>
<sequence name="ExternalService3" >
 <targets>
 <target linkName="Link1" />
 </targets>
 <sources>
 <source linkName="Link3" />
 </sources>
 <invoke partnerLink="PartnerLink2" portType="port:PortAirline"
 operation="bookFlight" inputVariable="Variable1"
 outputVariable="Variable2" name="Invoke4" >
 <correlations>
 </correlations>
 <compensationHandler>
 <invoke partnerLink="PartnerLink2" portType="port:PortAirline"
 operation="cancelFlight" inputVariable="Variable2" name="Invoke3" >
 </invoke>
 </compensationHandler>
 </invoke>
</sequence>
<sequence name="ExternalService4" >
 <targets>
 <target linkName="Link3" />
 </targets>
 <sources>
 <source linkName="Link2" />
 </sources>
 <invoke partnerLink="PartnerLink3" portType="port:Bank" operation="payHotel"
 inputVariable="Variable2" outputVariable="Variable3" name="Invoke6" >
 <correlations>
 </correlations>
 <compensationHandler>

```

```

 <invoke partnerLink="PartnerLink3" portType="port:Bank"
 operation="cancelHotelPayment" inputVariable="Variable3"
 name="Invoke5" >
 </invoke>
 </compensationHandler>
</invoke>
</sequence>
<sequence name="ExternalService5" >
 <targets>
 <target linkName="Link2" />
 </targets>
 <sources>
 <source linkName="Link4" />
 </sources>
 <invoke partnerLink="PartnerLink4" portType="port:Bank2" operation="payAirline"
 inputVariable="Variable3" outputVariable="Variable5" name="Invoke8" >
 <correlations>
 </correlations>
 <compensationHandler>
 <invoke partnerLink="PartnerLink4" portType="port:Bank2"
 operation="cancelAirlinePayment" inputVariable="Variable5"
 name="Invoke7" >
 </invoke>
 </compensationHandler>
 </invoke>
</sequence>
<sequence name="ExternalService6" >
 <targets>
 <target linkName="Link4" />
 </targets>
 <reply partnerLink="PartnerLink5" portType="home:customerPT" operation="getItinerary"
 variable="Variable5" name="Reply1" >
 <correlations>
 </correlations>
 </reply>
</sequence>
</flow>
</process>

```

---

## A.2 Die durch ActiveVOS modifizierte BPEL Datei

### Itinerary.bpel

---

```
<?xml version="1.0" encoding="UTF-8"?>
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:home="http://example.com/wSDL" xmlns:msg="http://example.com/message"
xmlns:port="http://example.com/port" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="es-
test" targetNamespace="http://example.com/bpel">
 <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="../wsdl/itinerary.wsdl"
namespace="http://example.com/wSDL"/>

 <bpel:partnerLinks>
 <bpel:partnerLink name="PartnerLink1" partnerLinkType="home:PartnerLinkType1"
partnerRole="PartnerRole1"/>
 <bpel:partnerLink name="PartnerLink2" partnerLinkType="home:PartnerLinkType2"
partnerRole="PartnerRole2"/>
 <bpel:partnerLink name="PartnerLink3" partnerLinkType="home:PartnerLinkType3"
partnerRole="PartnerRole3"/>
 <bpel:partnerLink name="PartnerLink4" partnerLinkType="home:PartnerLinkType4"
partnerRole="PartnerRole4"/>
 <bpel:partnerLink myRole="MyRole5" name="PartnerLink5"
partnerLinkType="home:PartnerLinkType5"/>
 </bpel:partnerLinks>

 <bpel:variables>
 <bpel:variable messageType="home:itinerary" name="Variable1"/>
 <bpel:variable messageType="home:itinerary" name="Variable2"/>
 <bpel:variable messageType="home:itinerary" name="Variable3"/>
 <bpel:variable messageType="home:itinerary" name="Variable4"/>
 <bpel:variable messageType="home:itinerary" name="Variable5"/>
 </bpel:variables>

 <bpel:flow name="Flow1" suppressJoinFailure="yes">
 <bpel:links>
 <bpel:link name="Link0"/>
 <bpel:link name="Link1"/>
 <bpel:link name="Link3"/>
 <bpel:link name="Link2"/>
 <bpel:link name="Link4"/>
 </bpel:links>

 <bpel:sequence name="ExternalService1">
 <bpel:sources>
 <bpel:source linkName="Link0"/>
 </bpel:sources>
 <bpel:receive createInstance="yes" name="Receive1"
operation="getItinerary" partnerLink="PartnerLink5" variable="Variable4"/>
 </bpel:sequence>

 <bpel:sequence name="ExternalService2">
 <bpel:targets>
 <bpel:target linkName="Link0"/>
 </bpel:targets>
 <bpel:sources>
 <bpel:source linkName="Link1"/>
 </bpel:sources>
 <bpel:invoke inputVariable="Variable4" name="Invoke2">

```

```

 operation="bookHotel" outputVariable="Variable1" partnerLink="PartnerLink1">
 <bpel:compensationHandler>
 <bpel:invoke inputVariable="Variable1" name="Invoke1"
 operation="cancelHotel" partnerLink="PartnerLink1"/>
 </bpel:compensationHandler>
 </bpel:invoke>
</bpel:sequence>

<bpel:sequence name="ExternalService3">
 <bpel:targets>
 <bpel:target linkName="Link1"/>
 </bpel:targets>
 <bpel:sources>
 <bpel:source linkName="Link3"/>
 </bpel:sources>
 <bpel:invoke inputVariable="Variable1" name="Invoke4"
 operation="bookFlight" outputVariable="Variable2" partnerLink="PartnerLink2">
 <bpel:compensationHandler>
 <bpel:invoke inputVariable="Variable2" name="Invoke3"
 operation="cancelFlight" partnerLink="PartnerLink2"/>
 </bpel:compensationHandler>
 </bpel:invoke>
</bpel:sequence>

<bpel:sequence name="ExternalService4">
 <bpel:targets>
 <bpel:target linkName="Link3"/>
 </bpel:targets>
 <bpel:sources>
 <bpel:source linkName="Link2"/>
 </bpel:sources>
 <bpel:invoke inputVariable="Variable2" name="Invoke6"
 operation="payHotel" outputVariable="Variable3" partnerLink="PartnerLink3">
 <bpel:compensationHandler>
 <bpel:invoke inputVariable="Variable3" name="Invoke5"
 operation="cancelHotelPayment" partnerLink="PartnerLink3"/>
 </bpel:compensationHandler>
 </bpel:invoke>
</bpel:sequence>

<bpel:sequence name="ExternalService5">
 <bpel:targets>
 <bpel:target linkName="Link2"/>
 </bpel:targets>
 <bpel:sources>
 <bpel:source linkName="Link4"/>
 </bpel:sources>
 <bpel:invoke inputVariable="Variable3" name="Invoke8"
 operation="payAirline" outputVariable="Variable5" partnerLink="PartnerLink4">
 <bpel:compensationHandler>
 <bpel:invoke inputVariable="Variable5" name="Invoke7"
 operation="cancelAirlinePayment" partnerLink="PartnerLink4"/>
 </bpel:compensationHandler>
 </bpel:invoke>
</bpel:sequence>

<bpel:sequence name="ExternalService6">
 <bpel:targets>
 <bpel:target linkName="Link4"/>
 </bpel:targets>
 <bpel:reply name="Reply1" operation="getItinerary"

```

```
 partnerLink="PartnerLink5" variable="Variable5"/>
 </bpel:sequence>

</bpel:flow>

</bpel:process>
```

---

## A.3 Die generierte WSDL Datei

### Itinerary.wsdl

---

```
<definitions targetNamespace="http://example.com/wsdl"
 xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:port="http://example.com/port"
 xmlns:msg="http://example.com/message"
 xmlns:home="http://example.com/wsdl">

 <portType name="customerPT">
 <operation name="getItinerary">
 <input message="msg:itinerary"/>
 <output message="msg:itinerary"/>
 </operation>
 </portType>

 <plnk:partnerLinkType name="PartnerLinkType1">
 <plnk:role name="PartnerRole1"
 portType ="port:PortHotel"/>
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="PartnerLinkType2">
 <plnk:role name="PartnerRole2"
 portType ="port:PortAirline"/>
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="PartnerLinkType3">
 <plnk:role name="PartnerRole3"
 portType ="port:Bank"/>
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="PartnerLinkType4">
 <plnk:role name="PartnerRole4"
 portType ="port:Bank2"/>
 </plnk:partnerLinkType>
 <plnk:partnerLinkType name="PartnerLinkType5">
 <plnk:role name="MyRole5"
 portType ="home:customerPT"/>
 </plnk:partnerLinkType>

</definitions>
```

---

## A.4 Die editierte WSDL Datei

### Itinerary.wsdl

---

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://example.com/wsdl"
 xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:port="http://example.com/port"
 xmlns:msg="http://example.com/message"
 xmlns:home="http://example.com/wsdl"
 xmlns:impl="http://DefaultNamespace"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

 <wsdl:message name="itinerary">
 <wsdl:part name="customerInfo" type="xsd:string"/>
 </wsdl:message>

 <wsdl:portType name="customerPT">
 <wsdl:operation name="getItinerary">
 <wsdl:input message="home:itinerary"/>
 <wsdl:output message="home:itinerary"/>
 </wsdl:operation>
 </wsdl:portType>

 <wsdl:portType name="Bank2">
 <wsdl:operation name="payAirline">
 <wsdl:input message="home:itinerary" name="payAirlineRequest"/>
 <wsdl:output
 message="home:itinerary" name="payAirlineResponse"/>
 </wsdl:operation>
 <wsdl:operation name="cancelAirlinePayment">
 <wsdl:input
 message="home:itinerary"
 name="cancelAirlinePaymentRequest"/>
 </wsdl:operation>
 </wsdl:portType>

 <wsdl:portType name="Bank">
 <wsdl:operation name="payHotel">
 <wsdl:input message="home:itinerary" name="payHotelRequest"/>
 <wsdl:output message="home:itinerary" name="payHotelResponse"/>
 </wsdl:operation>
 <wsdl:operation name="cancelHotelPayment">
 <wsdl:input
 message="home:itinerary"
 name="cancelHotelPaymentRequest"/>
 </wsdl:operation>
 </wsdl:portType>

 <wsdl:portType name="PortAirline">
 <wsdl:operation name="bookFlight">
 <wsdl:input message="home:itinerary" name="bookFlightRequest"/>
 <wsdl:output
 message="home:itinerary"
 name="bookFlightResponse"/>
 </wsdl:operation>
 </wsdl:portType>
</wsdl:definitions>
```

```

</wsdl:operation>
<wsdl:operation name="cancelFlight">
 <wsdl:input
 message="home:itinerary"
 name="cancelFlightRequest"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:portType name="PortHotel">
 <wsdl:operation name="bookHotel">
 <wsdl:input message="home:itinerary" name="bookHotelRequest"/>
 <wsdl:output
 message="home:itinerary"
 name="bookHotelResponse"/>
 </wsdl:operation>
 <wsdl:operation name="cancelHotel">
 <wsdl:input
 message="home:itinerary"
 name="cancelHotelRequest"/>
 </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="PortHotelSoapBinding" type="home:PortHotel">
 <soap:binding style="rpc"
 transport="http://schemas.xmlsoap.org/soap/http"/>
 <wsdl:operation name="bookHotel">
 <soap:operation soapAction=""/>
 <wsdl:input name="bookHotelRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 <wsdl:output name="bookHotelResponse">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:output>
 </wsdl:operation>

 <wsdl:operation name="cancelHotel">
 <soap:operation soapAction=""/>
 <wsdl:input name="cancelHotelRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 </wsdl:operation>
</wsdl:binding>

<wsdl:service name="PortHotelService">
 <wsdl:port binding="home:PortHotelSoapBinding" name="PortHotel">
 <soap:address location=
 "http://localhost:8080/WebProj/services/PortHotel"/>
 </wsdl:port>
</wsdl:service>

<wsdl:binding name="Bank2SoapBinding" type="home:Bank2">
 <soap:binding style="rpc" transport=

```

```

"http://schemas.xmlsoap.org/soap/http"/>
 <wsdl:operation name="payAirline">
 <soap:operation soapAction=""/>
 <wsdl:input name="payAirlineRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 <wsdl:output name="payAirlineResponse">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:output>
 </wsdl:operation>
 <wsdl:operation name="cancelAirlinePayment">
 <soap:operation soapAction=""/>
 <wsdl:input name="cancelAirlinePaymentRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 </wsdl:operation>
</wsdl:binding>

<wsdl:service name="Bank2Service">
 <wsdl:port binding="home:Bank2SoapBinding" name="Bank2">
 <soap:address location=
 "http://localhost:8081/Server2/services/Bank2"/>
 </wsdl:port>
</wsdl:service>

<wsdl:binding name="BankSoapBinding" type="home:Bank">
 <soap:binding style="rpc" transport=
 "http://schemas.xmlsoap.org/soap/http"/>
 <wsdl:operation name="payHotel">
 <soap:operation soapAction=""/>
 <wsdl:input name="payHotelRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 <wsdl:output name="payHotelResponse">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:output>
 </wsdl:operation>
 <wsdl:operation name="cancelHotelPayment">
 <soap:operation soapAction=""/>
 <wsdl:input name="cancelHotelPaymentRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 </wsdl:operation>
</wsdl:binding>

<wsdl:service name="BankService">

```

```

 <wsdl:port binding="home:BankSoapBinding" name="Bank">
 <soap:address location=
 "http://localhost:8080/WebProj/services/Bank"/>
 </wsdl:port>
 </wsdl:service>

 <wsdl:binding name="PortAirlineSoapBinding" type="home:PortAirline">
 <soap:binding style="rpc" transport=
 "http://schemas.xmlsoap.org/soap/http"/>
 <wsdl:operation name="bookFlight">
 <soap:operation soapAction=""/>
 <wsdl:input name="bookFlightRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 <wsdl:output name="bookFlightResponse">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:output>
 </wsdl:operation>
 <wsdl:operation name="cancelFlight">
 <soap:operation soapAction=""/>
 <wsdl:input name="cancelFlightRequest">
 <soap:body encodingStyle=
 "http://schemas.xmlsoap.org/soap/encoding/" namespace=
 "http://example.com/wsdl" use="encoded"/>
 </wsdl:input>
 </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="PortAirlineService">
 <wsdl:port binding=
 "home:PortAirlineSoapBinding" name="PortAirline">
 <soap:address location=
 "http://localhost:8080/WebProj/services/PortAirline"/>
 </wsdl:port>
 </wsdl:service>

 <plnk:partnerLinkType name="PartnerLinkType2">
 <plnk:role name="PartnerRole2" portType="home:PortAirline"/>
 </plnk:partnerLinkType>

 <plnk:partnerLinkType name="PartnerLinkType1">
 <plnk:role name="PartnerRole1" portType="home:PortHotel"/>
 </plnk:partnerLinkType>

 <plnk:partnerLinkType name="PartnerLinkType3">
 <plnk:role name="PartnerRole3" portType="home:Bank"/>
 </plnk:partnerLinkType>

 <plnk:partnerLinkType name="PartnerLinkType4">
 <plnk:role name="PartnerRole4" portType="home:Bank2"/>
 </plnk:partnerLinkType>

 <plnk:partnerLinkType name="PartnerLinkType5">
 <plnk:role name="MyRole5" portType="home:customerPT"/>
 </plnk:partnerLinkType>

</wsdl:definitions

```

## A.5 Der Process Deployment Descriptor

### Itinerary.pdd

---

```
<?xml version="1.0" encoding="UTF-8"?>

<pdd:process xmlns:bpeln="http://example.com/bpel"
xmlns:pdd="http://schemas.active-endpoints.com/pdd/2006/08/pdd.xsd"
location="bpel/test3/bpel/itinerary.bpel" name="bpeln:es-test"
platform="opensource">
 <pdd:partnerLinks>

 <pdd:partnerLink name="PartnerLink1">
 <pdd:partnerRole endpointReference="static" invokeHandler=
"default:Address">
 <wsa:EndpointReference xmlns:ns5=
"http://example.com/wsdl" xmlns:soapenv=
"http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa=
"http://www.w3.org/2005/08/addressing" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
 <wsa:Address>
 http://localhost:8080/WebProj/services/PortHotel
 </wsa:Address>
 <wsa:Metadata>
 <wsa:ServiceName PortName="PortHotel">
 ns5:PortHotelService
 </wsa:ServiceName>
 </wsa:Metadata>
 </wsa:EndpointReference>
 </pdd:partnerRole>
 </pdd:partnerLink>

 <pdd:partnerLink name="PartnerLink2">
 <pdd:partnerRole endpointReference="static" invokeHandler=
"default:Address">
 <wsa:EndpointReference xmlns:ns5=
"http://example.com/wsdl" xmlns:soapenv=
"http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa=
"http://www.w3.org/2005/08/addressing" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
 <wsa:Address>
 http://localhost:8080/WebProj/services/PortAirline
 </wsa:Address>
 <wsa:Metadata>
 <wsa:ServiceName PortName="PortAirline">
 ns5:PortAirlineService</wsa:ServiceName>
 </wsa:Metadata>
 </wsa:EndpointReference>
 </pdd:partnerRole>
 </pdd:partnerLink>

 <pdd:partnerLink name="PartnerLink3">
 <pdd:partnerRole endpointReference="static" invokeHandler=
"default:Address">
 <wsa:EndpointReference xmlns:ns5=
"http://example.com/wsdl" xmlns:soapenv=
"http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa=
```

```

"http://www.w3.org/2005/08/addressing" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
 <wsa:Address>
 http://localhost:8080/WebProj/services/Bank
 </wsa:Address>
 <wsa:Metadata>
 <wsa:ServiceName PortName="Bank">
 ns5:BankService</wsa:ServiceName>
 </wsa:Metadata>
 </wsa:EndpointReference>
</pdd:partnerRole>
</pdd:partnerLink>

<pdd:partnerLink name="PartnerLink4">
 <pdd:partnerRole endpointReference="static" invokeHandler=
 "default:Address">
 <wsa:EndpointReference xmlns:ns5=
 "http://example.com/wsdl" xmlns:soapenv=
 "http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsa=
 "http://www.w3.org/2005/08/addressing" xmlns:xsd=
 "http://www.w3.org/2001/XMLSchema" xmlns:xsi=
 "http://www.w3.org/2001/XMLSchema-instance">
 <wsa:Address>
 http://localhost:8081/Server2/services/Bank2
 </wsa:Address>
 <wsa:Metadata>
 <wsa:ServiceName PortName="Bank2">
 ns5:Bank2Service</wsa:ServiceName>
 </wsa:Metadata>
 </wsa:EndpointReference>
 </pdd:partnerRole>
 </pdd:partnerLink>

<pdd:partnerLink name="PartnerLink5">
 <pdd:myRole allowedRoles="" binding="RPC-LIT" service=
 "PartnerLink5Service"/>
</pdd:partnerLink>

</pdd:partnerLinks>

<pdd:references>
 <pdd:wsdl location="project:/test3/wsdl/itinerary.wsdl" namespace=
 "http://example.com/wsdl"/>
</pdd:references>

</pdd:process>

```

---

## A.6 Die Logdatei des fehlgeschlagenen Prozesses

**active endpoints** Active Process Detail:

Outline Log

Filter: Execution Up Down Goto

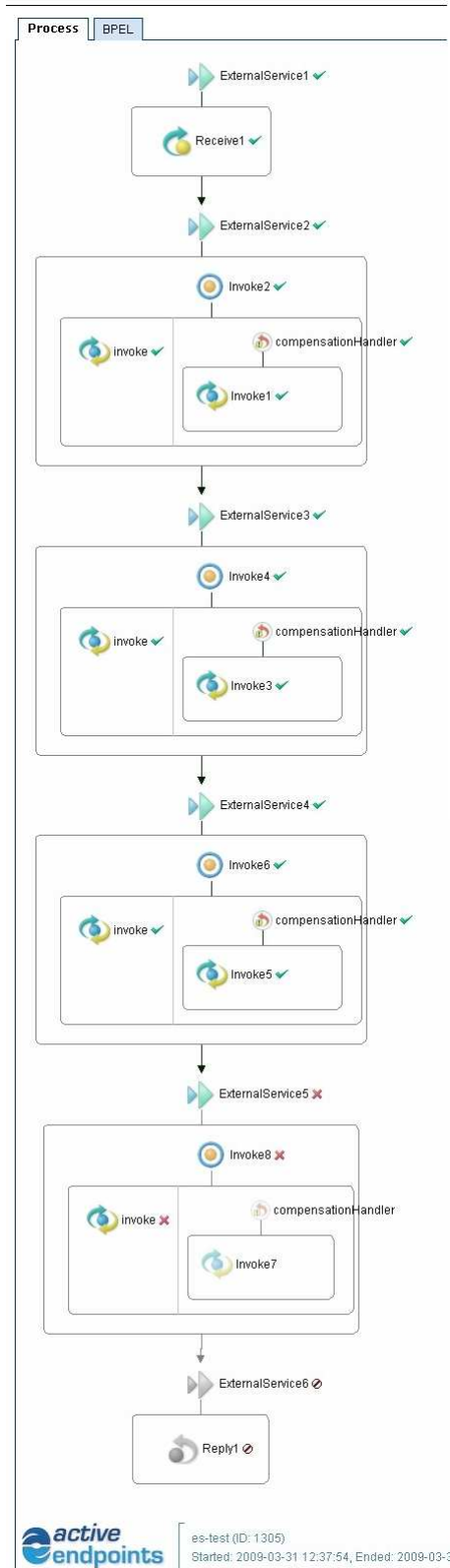
|   |                                                         |                  |
|---|---------------------------------------------------------|------------------|
| ▶ | process                                                 | 2009-03-31 12:37 |
| ▶ | Flow1                                                   | 2009-03-31 12:37 |
| ▶ | ExternalService1                                        | 2009-03-31 12:37 |
| ▶ | Receive1                                                | 2009-03-31 12:37 |
| ✓ | Receive1                                                | 2009-03-31 12:37 |
| ✓ | ExternalService1                                        | 2009-03-31 12:37 |
| ▶ | ExternalService2                                        | 2009-03-31 12:37 |
| ▶ | Invoke2                                                 | 2009-03-31 12:37 |
| ▶ | invoke                                                  | 2009-03-31 12:37 |
| ✓ | invoke                                                  | 2009-03-31 12:37 |
| ✓ | Invoke2                                                 | 2009-03-31 12:37 |
| ✓ | ExternalService2                                        | 2009-03-31 12:37 |
| ▶ | ExternalService3                                        | 2009-03-31 12:37 |
| ▶ | Invoke4                                                 | 2009-03-31 12:37 |
| ▶ | invoke                                                  | 2009-03-31 12:37 |
| ✓ | invoke                                                  | 2009-03-31 12:37 |
| ✓ | Invoke4                                                 | 2009-03-31 12:37 |
| ✓ | ExternalService3                                        | 2009-03-31 12:37 |
| ▶ | ExternalService4                                        | 2009-03-31 12:37 |
| ▶ | Invoke5                                                 | 2009-03-31 12:37 |
| ▶ | invoke                                                  | 2009-03-31 12:37 |
| ✓ | invoke                                                  | 2009-03-31 12:37 |
| ✓ | Invoke5                                                 | 2009-03-31 12:37 |
| ✓ | ExternalService4                                        | 2009-03-31 12:37 |
| ▶ | ExternalService5                                        | 2009-03-31 12:37 |
| ▶ | Invoke8                                                 | 2009-03-31 12:37 |
| ▶ | invoke                                                  | 2009-03-31 12:37 |
| ✗ | invoke                                                  | 2009-03-31 12:37 |
| ▶ | Invoke8                                                 | 2009-03-31 12:37 |
| ▶ | Invoke8                                                 | 2009-03-31 12:37 |
| ✓ | Invoke8                                                 | 2009-03-31 12:37 |
| ✓ | Invoke8                                                 | 2009-03-31 12:37 |
| ✗ | Invoke8                                                 | 2009-03-31 12:37 |
| ✗ | ExternalService5                                        | 2009-03-31 12:37 |
| ✗ | Flow1                                                   | 2009-03-31 12:37 |
| ▶ | process_implicitFaultHandler                            | 2009-03-31 12:37 |
| ▶ | process_implicitFaultHandler_implicitCompensateActivity | 2009-03-31 12:37 |
| ▶ | compensationHandler                                     | 2009-03-31 12:37 |
| ▶ | Invoke5                                                 | 2009-03-31 12:37 |
| ✓ | Invoke5                                                 | 2009-03-31 12:37 |
| ✓ | compensationHandler                                     | 2009-03-31 12:37 |
| ▶ | compensationHandler                                     | 2009-03-31 12:37 |
| ▶ | Invoke3                                                 | 2009-03-31 12:37 |
| ✓ | Invoke3                                                 | 2009-03-31 12:37 |
| ✓ | compensationHandler                                     | 2009-03-31 12:37 |
| ▶ | compensationHandler                                     | 2009-03-31 12:37 |
| ▶ | Invoke1                                                 | 2009-03-31 12:37 |
| ✓ | Invoke1                                                 | 2009-03-31 12:37 |
| ✓ | compensationHandler                                     | 2009-03-31 12:37 |
| ✓ | process_implicitFaultHandler_implicitCompensateActivity | 2009-03-31 12:37 |
| ✓ | process_implicitFaultHandler                            | 2009-03-31 12:37 |
| ✗ | process                                                 | 2009-03-31 12:37 |

scope

|            |                                                                                            |
|------------|--------------------------------------------------------------------------------------------|
| Date       | 2009-03-31 12:37:54.234                                                                    |
| Name       | Invoke6                                                                                    |
| Event Type | EXECUTING                                                                                  |
| Path       | /process/flow[@name='Flow1']<br>/sequence[@name='ExternalService4']/scope[@name='Invoke6'] |

## A.7 Abbildung des fehlgeschlagenen Prozesses

Active Process Detail: es-test (ID 1305)



**Erklärung**

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Stuttgart, den 16.04.2009

---

(Frank Schmid)