Universität Stuttgart

Ausführbare Integrationsmuster

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Thorsten Scheibler

aus Stuttgart (Deutschland)

Hauptberichter: Prof. Dr. Frank Leymann

Mitberichter: Univ.-Prof. Dr. Schahram Dustdar

Tag der mündlichen Prüfung: 17. Juni 2010

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2010

Inhaltsverzeichnis

Zusammenfassung

Die Integration von großen IT Anwendungen ist für viele Firmen eine der bedeutendsten IT Herausforderungen, um die Geschäftsprozesse innerhalb eines Unternehmens effizient durchführen zu können. Zum einen gab es kein standardisiertes Vorgehen, wie man Integrationslandschaften konzipieren und spezifizieren konnte. Zum anderen bestand keine durchgängige Methode, die resultierenden zumeist abstrakten Architekturen in ausführbare Systeme zu überführen. Mit der Einführung von Integrationsmustern wurde ein großer Beitrag geleistet, um Integrationslandschaften einheitlich und technologieunabhängig darstellen zu können. Jedoch klaffte immer noch eine große Lücke zwischen der Modellierung und der Implementierung dieser Landschaften. Die tatsächliche Realisierung wurde durch Entwickler geleistet, indem die Landschaften interpretiert und entsprechend ausführbare Integrationslogik erstellt wurde. Diese Kluft muss geschlossen werden, damit Architekten und Entwickler besser miteinander kommunizieren und ausführbare Integrationslösungen effizienter realisiert werden können. Diese Dissertation leistet einen Beitrag, um diese Lücke zu schließen.

Im Rahmen dieser Arbeit wird eine Methode entwickelt, mit der ausgehend von Integrationsmustern automatisiert ausführbare Integrationslösungen erzeugt werden können. Es wird eigens ein Lebenszyklus konzipiert, der die unterschiedlichen Phasen der Erstellung von ausführbaren Integrationslösun-

gen beschreibt. Die Methode basiert auf der Modell-getriebenen Entwicklung. Die vormals visuellen und textuellen Repräsentationen der Integrationsmuster werden dazu in ein formales Modell überführt. Parametrisierbare Integrationsmuster dienen fortan als Grundlage der Methode. Durch die Parameter jedes einzelnen Musters kann das erwartete Verhalten eines Muster technologieunabhängig spezifiziert werden. Außerdem beschreibt das Modell die Zusammenhänge der einzelnen Muster und wie sie miteinander kommunizieren.

Das Modell der parametrisierbaren Integrationsmuster wird von einem Generierungsalgorithmus verwendet, um automatisiert ausführbare Integrationslösungen zu erstellen. Der Algorithmus fügt dazu plattformspezifische Informationen hinzu, so dass eine Integrationslösung auf einer bestimmten Zielinfrastruktur ausgeführt werden kann. Ein Modell kann dabei von verschiedenen Algorithmen verwendet werden und daher in verschiedene Ausführungsumgebungen übersetzt werden. Diese Methode ist daher nicht auf bestimmte Zielinfrastrukturen beschränkt. Im Verlauf der Arbeit werden parametrisierbare Integrationsmuster auf drei sehr unterschiedlichen Infrastrukturen abgebildet und ausgeführt. Dies verdeutlicht die Allgemeinheit der Methode und die leichte Erweiterbarkeit auf neue Technologien und veränderte Anforderungen.

Zur Unterstützung der Methode wird außerdem eine Werkzeugkette erstellt, die es Systemarchitekten erlaubt, eine Integrationslösung grafisch zu konzipieren. Das Werkzeug GENIUS umfasst eine graphische Modellierungsoberfläche, mit der parametrisierbare Integrationsmuster zu einer Integrationslösung zusammengefasst werden können. Außerdem werden zwei Algorithmen integriert, die ausführbare Integrationslösungen für zwei unterschiedliche Zielinfrastrukturen erzeugen.

Diese Dissertation schließt somit die Lücke zwischen Architektur und Entwicklung, indem eine auf der Modell-getriebenen Entwicklung basierenden Methode entwickelt wird, mit deren Hilfe parametrisierbare Integrationsmuster als direkte Spezifikation für ausführbare Systeme dienen und automatisiert in diese Systeme überführt werden.

Abstract

Integration of huge IT applications is one of the most important IT challenges companies are faced to enable the efficient execution of business processes. There was no standardized procedure how integration landscapes could be designed and specified on the one hand. And, on the other hand, no continuous method exists to transform those in most cases abstract architectures into executable systems. Due to the advent of integration patterns an important contribution was achieved for representing these landscapes in an uniform and technology independent manner. However, there was still a gap between the modeling of such integration solutions and the actual implementation. The realization was accomplished by developers through interpreting the landscapes and implementing appropriate executable systems on their own. This gap has to be closed to ease the communication between architects and developers, and let the realization of executable systems be more efficient. This dissertation contributes to closing this gap.

In the context of this dissertation a method will be developed for automatically generating executable integration solutions based on integration patterns. Especially for that purpose an integration process will be specified which includes the various phases while creating executable integration solutions. The method is based on model-driven development. For it, the formerly visual and textual representation of integration patterns is transformed into a formal

model. Parameterizable integration patterns serve from now on as the basis for the method. By use of the parameters of each pattern, the behavior of each pattern can be specified platform independently. Furthermore, the combination of individual patterns and the communication between them can be described by the model.

The model of parameterizable integration patterns is used by generation algorithms to automatically compile executable integration solutions. For that purpose, the algorithm adds platform specific information so that an integration solution can be executed on the target infrastructure. A single model can be used by various algorithms which will lead to executable solutions for various execution environments. Thus, the proposed method is not bound to a particular target infrastructure. In course of the dissertation parameterizable integration patterns will be mapped to and executed on three very different infrastructures. These mappings point out the general approach of the proposed method and the flexibility and extensibility due to new or changed technologies and requirements.

For a better support of the method a tool chain has been developed which allows a system architect to design an integration solution graphically. The tool GENIUS comprises a graphical modeling capability with which parameterizable integration patterns can be combined into an integration solution. Moreover, two different algorithms will be included which will produce executable integration solution for two different target infrastructures.

Hence, this dissertation closes the gap between the architects and the developers by introducing a method based on model-driven development which takes parameterizable integration patterns as direct specification of executable systems and transforms the patterns automatically into these systems.

Danksagungen

Während der Erstellung dieser Dissertation haben mich zahlreiche Personen unterstützt, gefördert und motiviert. Diesen möchte ich hiermit danken.

Zunächst geht mein Dank an Herrn Prof. Frank Leymann, der es mir durch die Einstellung am IAAS überhaupt erst ermöglicht hat, meine Dissertation zu schreiben. In den vergangenen Jahren habe ich in vielen interessanten Diskussionen mit ihm immer wieder neue Ideen für die Dissertation erarbeiten können. In der täglichen Arbeit konnte ich durch ihn wichtige Erfahrungen sammeln, die ich in den nächsten Jahren hoffentlich oft anwenden kann. Außerdem möchte ich mich bei Herrn Prof. Schahram Dustdar bedanken, der sich bereit erklärt hat, diese Dissertation zu begutachten.

Allen meinen Kollegen am IAAS gebührt mein Dank für die Unterstützung und die zahlreichen wissenschaftlichen Diskussionen. Ich möchte hier besonders die Kollegen herausheben, mit denen ich intensiv an Themen rund um die Dissertation arbeiten durfte. Dies sind Dr. Dimka Karastoyanova, Ralph Mietzner, Dieter Roller, Tobias Unger, Daniel Martin und Daniel Wutke.

Zum Gelingen dieser Dissertation haben auch einige Studenten beigetragen, die wichtige Themen der Dissertation ausgearbeitet haben: Bettina Druckenmüller, Xin Yuan, Oliver Eckhardt, Frederick Juchart, Pascal Kolb, Florian Schebelle, Frank Schmid und Christian Strempfer. Danke für eure Mitarbeit.

Eine Ausarbeitung ist nur halb so viel wert, wenn man nicht tolle Freunde

hat, die sich bereiterklären, diese Arbeit in ihrer Freizeit zu korrigieren und mir als Autor die Textblindheit nehmen. Ich danke daher von Herzen Tanja Lämmer, Jochen Brühl, Fabian Kaiser und Ralph Mietzner sowie Isabell und Edith Scheibler, die Fehler entdeckt haben und sich einige Stunden mit dieser Ausarbeitung beschäftig haben.

Zu guter Letzt darf ich meiner Familie danken, die mich durch alle Höhen und die paar wenigen Tiefen der Erstellung dieser Dissertation begleitet haben. Ihr wart mir immer hilfreiche Begleiter und Förderer. Nur durch eine starke Familie und den Zusammenhalt konnte ich diese Arbeit erfolgreich abschließen. Danke, dass ihr immer für mich da wart und mir die Kraft und Ausdauer gegeben habt, die ich benötigt habe, um die Arbeit zu vollenden. Danke Izzy, Jannick, Edith, Harry und Anke!

KAPITEL 1

EINFÜHRUNG

Aufgrund der steigenden Dynamik und des stetigen Wandels in einer globalisierten Welt sehen sich Unternehmen zunehmend mit zahlreichen Herausforderungen in der *IT (Informationstechnologie)* Umgebung konfrontiert. Eine der Bedeutendsten ist dabei die Integration von verschiedenen Anwendungssystemen. Unternehmen verwenden typischerweise hunderte, wenn nicht tausende, von unterschiedlichen Anwendungen, die selbst entwickelt oder von fremden Firmen gekauft wurden, Teil einer „legacy" Anwendung sind oder eine Kombination aus diesen Möglichkeiten darstellen. Diese Anwendungen arbeiten typischerweise auf unterschiedlichen Plattformen unter verschiedenen Betriebssystemen. Um die Geschäftstätigkeit und Geschäftsprozesse innerhalb eines Unternehmens zu unterstützen beziehungsweise in Software abzubilden, müssen diese unterschiedlichen Anwendungen miteinander arbeiten, d.h. integriert werden (*Anwendungsintegration*). In Zeiten von Umstrukturierungen innerhalb von Firmen oder durch den Zukauf neuer Firmen tritt das Problem der Anwendungsintegration verstärkt auf: es müssen heterogene Anwendungslandschaften verschiedenster Geschäftsbereiche oder gar von Unternehmen zusammen gefügt werden, damit die Umstrukturierung auch in der Anwendungslandschaft vollzogen werden kann. Somit können die nun bereichs-

beziehungsweise firmenübergreifenden Geschäftsprozesse durch die Integration der neuen Anwendungen mit den bereits im Unternehmen vorhandenen Anwendungen unterstützt werden. Die Herausforderung besteht darin, dass die einzelnen Systeme zum größten Teil nicht im Hinblick auf eine Integration mit anderen Systemen entwickelt wurden. Beispielsweise waren sie als große, monolithische, alleinstehende und unabhängige Anwendungen konzipiert; allerdings werden die Anforderungen im Laufe der Zeit komplexer, so dass ein solches System alleine nicht mehr ausreicht, um bestimmte Geschäftsvorfälle, die sich ebenfalls geändert haben, abwickeln zu können. Solche Änderungen in den Geschäftsvorfällen resultieren daher beispielsweise in zusätzlichen Anwendungen, die die geänderten Teile der Geschäftsprozesse unterstützen. Dies ist einer der Gründe, weshalb neue Anwendungssysteme zur Anwendungslandschaft hinzugefügt und mit bereits existierenden Anwendungssystemen integriert werden müssen.

Mechanismen und Technologien für die Integration von Anwendungen sind Gegenstand des Gebietes der Unternehmensanwendungsintegration (*EAI, Enterprise Application Integration* [Kel02][Lin99]) – hierbei wird der Präfix „Unternehmens-“ gelegentlich verwendet, um die Bedeutung der entsprechenden Anwendungen zur Unterstützung der Geschäftstätigkeit eines Unternehmens zu betonen. Die Unterstützung der Tätigkeit eines Unternehmens durch EAI bedeutet nicht die Entwicklung der entsprechenden Anwendungsfunktionalität im traditionellen Sinne der Softwaretechnologie, sondern die Nutzung und Koordinierung bereits existierender Anwendungsteile. Da diese Anwendungsteile über heterogene Umgebungen verteilt sind, wird eine spezielle Middleware-Landschaft benötigt, die diese Teile zugreifbar und koordinierbar macht. Diese Landschaft wird oft als *Integrationsinfrastruktur* bezeichnet und umfasst Technologien wie etwa *Message Queuing (MQ)* [BHL95][MHC00], *Workflow Management Systeme (WfMS)* [LR00] oder *Enterprise Service Bus (ESB)* [Cha04][SHLP05]. Die Entwicklung einer Lösung zu einem Integrationsproblem mit Hilfe dieser Integrationsinfrastruktur wird als *Integrationslösung* bezeichnet. Im Laufe der letzten Jahre sind eine Reihe von Mustern zur Lösung von Teilaspekten von Integrationsproblemen vorgeschlagen worden [HW03][TRH+04]. Integrationslösungen werden üblicherweise durch

Kombination solcher Muster modelliert. Allerdings müssen diese Modelle anschließend manuell in ausführbare Software umgesetzt werden.

Im Fokus dieser Arbeit steht die Modellierung von Integrationslösungen mit Hilfe der etablierten Muster und deren automatisierter Umsetzung in ausführbare Einheiten. Es wird eine neuartige Methode vorgestellt, mit der sich Integrationslösungen auf einer hohen Abstraktionsebene modellieren lassen, ohne technische Details der zugrunde liegenden Integrationsinfrastruktur betrachten zu müssen. Diese Methode ermöglicht es darüber hinaus, die erstellten Modelle unmittelbar in der Integrationsinfrastruktur ausführen zu können, indem sie automatisiert übersetzt werden.

1.1. Problembereich und Motivation

Die Erstellung von Integrationslösungen ist keine Aufgabe, die nur einmal durchlaufen werden muss. Vielmehr ist dieses Vorgehen iterativ, da immer wieder neue Anforderungen an bestehende (Integrations-) Lösungen gestellt werden, die eingearbeitet werden müssen (zum Beispiel Hinzufügen neuer Anwendungen). Darüber hinaus treten bei der Erstellung von Integrationslösungen immer wieder die gleichen sich wiederholenden Probleme auf. So müssen zum Beispiel ständig Daten zwischen verschiedenen Formaten transformiert werden. Diese Probleme werden dabei mit gleichen oder ähnlichen Lösungen gemeistert. Solche wiederkehrenden Probleme samt deren Lösungen werden im Allgemeinen als Muster (*Patterns*) bezeichnet. Der Begriff Muster, in dem hier verwendeten Sinne, wurde erstmals im Jahre 1977 durch den Architekten Christopher Alexander geprägt [Ale77]. Muster beschreiben dabei wiederkehrende Elemente beziehungsweise Problemstellungen im Bereich der Städte- und Hausbauarchitektur (zum Beispiel *Bettnische (bed alcove)* [Ale79]). Muster werden von Experten entwickelt, die über dieses Medium ihre jahrelange Erfahrung auf bestimmten Gebieten an weniger erfahrene Personen weitergeben. Dies hat sich als praktikabler Ansatz herausgestellt und führt daher zu weniger Fehlern in der Umsetzung. Ein Muster besitzt zunächst einen bezeichnenden Namen. Diesen Namen kann man auch in gesprochenen oder geschriebenen Sätzen verwenden, denn Muster sind dazu gedacht, von Menschen in ihrer

natürlichen Sprache eingesetzt zu werden („... Sollen wir in diesen Raum eine Nische für ein Bett integrieren?“). Neben dem bezeichnenden Namen des Musters beschreibt das Muster auch den Kontext, in dem es eingesetzt werden kann, was die treibenden Kräfte für seinen Einsatz sind und welche Probleme es löst. Darüber hinaus dienen Muster der Abstraktion, denn ein Muster sagt zunächst nichts darüber aus, wie es letztendlich umgesetzt (implementiert) werden soll. Bei dem Muster Bettnische etwa wird keine Aussage getroffen, aus welchem Material diese Nische gestaltet wird oder mit welchen Werkzeugen sie erstellt werden soll.

Der Begriff der Muster wurde Mitte der neunziger Jahre auch für den Bereich der Softwareentwicklung entdeckt. [GHJ95] beschreibt zum ersten Mal, wie wiederkehrende Probleme und Vorschläge zu deren Lösung im Bereich der *Objekt-orientierten Programmierung (OOP)* in Form von Mustern für den Entwurf von OO Systemen dargestellt werden können. Die Autoren beschreiben hierbei ihre Erfahrung bei der Implementierung von OO Systemen und erläutern, wie bestimmte Problemstellungen gemeistert werden können. Neben diesen weit verbreiteten und populären Entwurfsmustern werden neuerdings Muster auch für die Erstellung ganzer Systemarchitekturen eingesetzt. Komplette Anwendungslandschaften können mit Hilfe von Mustern konzipiert werden [FRF02][BMR+96]. Darüber hinaus existieren Muster zur Abbildung von komplexen Architekturen, die Integrationslösungen darstellen: *Enterprise Integration Patterns (EIP)* [HW03]. Diese Muster beschreiben, wie man Probleme bewältigen kann, die bei der Integration verschiedener heterogener Anwendungen auftreten. Die Muster basieren auf der *Pipes-and-Filters (PaF) Architektur* [Meu95][PW92] und kommunizieren mit Hilfe von *Message-oriented Middleware (MOM)* [HW03]. Integrationsmuster bieten die Möglichkeit, sowohl kleinste Problemstellungen in einer Integrationslösung zu bewältigen als auch zusammengesetzte, komplexe Probleme beherrschen zu können. So beschreibt zum Beispiel das Muster *Message Translator*, wie man ein eingehendes Nachrichtenformat auf ein ausgehendes Format übersetzt. Wohingegen mit Hilfe des Musters *Scatter-Gather* eine eingehende Nachricht in mehrere Teile aufgespalten, diese parallel abgearbeitet und am Ende wieder zusammengefügt wird.

Allerdings sind die Integrationsmuster bislang nur als Dokumentation gedacht. Sie ermöglichen es, Systemarchitekten über eine einheitliche Sprache untereinander und mit den beteiligten Entwicklern zu kommunizieren. Durch die Muster wird dabei jedoch keine Aussage getroffen, wie die Architektur letzten Endes in ein ausführbares System überführt werden kann, d.h. wie die Muster letztlich implementiert werden sollen. Dieser Schritt wird von Entwicklern manuell durchgeführt, indem sie die vorhandene Dokumentation (in Form von Zeichnungen und textueller Beschreibung) interpretieren und entsprechend ihrer Erfahrung in ausführbaren Programmcode umsetzen. Die Integrationsmuster stellen also ein Modell der Integrationslösung dar. Zwischen Modell und Implementierung gibt es in aller Regel nur eine gedankliche Verbindung.

Diese Verwendung von Modellen in einem Entwicklungsprozess wird *Modellbasierte Entwicklung* genannt [VS06]. Das Vorgehen birgt allerdings entscheidende Nachteile: ein Softwaresystem ist während der Entwicklung und Laufzeit ständigen Änderungen und Anpassungen unterworfen. Die Dokumentation muss deswegen ebenfalls ständig sorgfältig mitgezogen und angepasst werden, ansonsten wird die Dokumentation bezüglich der Implementierung schnell inkonsistent. Außerdem tragen diese Modelle (beziehungsweise die Dokumentation) nur unwesentlich zum Fortschritt bei, da erst durch die Interpretation der Modelle durch die Entwickler ausführbare Systeme entstehen. Im Gegensatz zu diesem Vorgehen stehen bei der Modell-getriebenen Softwareentwicklung (*model-driven (software) development, MDD*) [VS06] die Modelle im Mittelpunkt des Interesses; sie sind praktisch die treibenden Faktoren. Modelle dienen hierbei nicht nur der Dokumentation, sondern sind gleichzusetzen mit Quelltext: die Umsetzung in ausführbare Systeme wird hierbei automatisiert. Das Ziel von MDD ist, für spezielle Domänen (wie etwa EAI) ein formales Modell zu entwickeln, das durch Abstraktion Domänenexperten die Möglichkeit bietet, Lösungen zu konstruieren. Diese Lösungen können nun zur automatisierten Entwicklung von ausführbaren Systemen verwendet werden. Mit der Automatisierung geht in den meisten Fällen eine Produktivitätssteigerung einher: Änderungen an Anforderungen können im Modell integriert und danach durch Generierung rasch in ausführbare Systeme umgesetzt werden. Auch führt die

Ersetzung des manuellen Schrittes der Implementierung zu deutlichen Verbesserungen der Qualität und der Wartbarkeit [VS06]. Bildet man nun dieses Vorgehen auf die Implementierung von Integrationslösungen ab, so stellen die eingesetzten Integrationsmuster die Elemente des Modells dar. Eine automatisierte Übersetzung dieser Modelle in ausführbare Systeme, analog zum Vorgehen bei MDD, existiert jedoch noch nicht.

Neben der Verbesserung der Produktivität und Qualität durch die automatisierte Generierung von ausführbaren Lösungen ist die Werkzeugunterstützung essentiell, um ein solches Vorgehen praktikabel zu machen. Zu dieser Unterstützung zählt neben dem Algorithmus, der die automatisierte Generierung vornimmt, auch eine integrierte Werkzeugbox, die durch eine grafische Benutzungsoberfläche den Architekten bei der Spezifikation von Integrationslösungen unterstützt. Ein solches Werkzeug sollte die Möglichkeit bieten, einzelne Muster aus einem Katalog zu wählen, diese gegebenenfalls zu verfeinern und dann mit anderen vorhandenen Mustern zu kombinieren. Das Hauptproblem hierbei ist die Verfeinerung der Muster. Für gewöhnlich werden Muster durch Bilder oder Symbole repräsentiert, die allerdings keine Aussage über ihr Verhalten treffen. Bevor die Muster nun in ein Werkzeug eingebunden werden können, müssen Sie zunächst formal beschrieben werden. Darüber hinaus werden Variabilitätspunkte benötigt, um die unterschiedlichen Ausprägungen eines Musters beschreiben zu können. Variabilitätspunkte kommen aus dem Bereich der Softwareproduktlinien [JGJ97] und können auch für die Variabilität der (Integrations-) Muster angewendet werden. Am Beispiel des Musters *Bett* (aus dem Musterkatalog aus [Ale79]) bedeuten verschiedene Ausprägungen zum Beispiel, dass bestimmt wird, wie breit und hoch das Bett ist und aus welchem Material der Rahmen sein soll. Bei dem Muster Message Translator [HW03] beschreibt ein Variabilitätspunkt zum Beispiel die Art der Transformation von Nachrichten. Ein solches Muster samt ausgefüllten Variabilitätspunkten kann nun verwendet werden, um die Erzeugung der Implementierung automatisiert durchlaufen zu lassen. Die gesamten für die Automatisierung dieses Schrittes benötigten Informationen sind vorhanden und werden im nächsten Schritt in ausführbare Systeme übersetzt (oder im Fall des Bettes wird es von einem Handwerker verwendet, um exakt die Vorgaben umzusetzen). In dieser Arbeit

werden die Variabilitätspunkte mit dem Begriff Parameter belegt, da dieser Begriff in der Praxis weitaus gebräuchlicher ist.

Neben den Variabilitätspunkten ist auch ein Metamodell nötig, welches beschreibt, was die einzelnen Elemente des Modells (der Muster) bedeuten. Im Falle des Bettes muss beschrieben werden, was zum Beispiel ein rechteckiges Kästchen ausdrücken soll (zum Beispiel Breite und Länge). Der Automatisierungsschritt muss darüber hinaus wissen, dass es sich auch tatsächlich um ein Bett handelt und nicht zum Beispiel um einen Raum. Somit muss für jede Anwendungsdomäne (zum Beispiel Integrationsmuster oder Innenraummuster) ein entsprechendes Metamodell existieren, das die einzelnen Muster definiert und auch, so weit möglich, bereits erlaubte und nicht erlaubte Verbindungen vorgibt.

In dieser Arbeit wird ein Ansatz beschrieben, der mit Hilfe eines an MDD orientierten Vorgehens Integrationsmuster in ausführbare Systeme überführt. Dazu werden zunächst die Integrationsmuster, die in textueller und graphischer Form vorliegen, in eine formale Darstellung, ein Metamodell, überführt. Darüber hinaus werden Variabilitätspunkte eingeführt, die die einzelnen Muster und deren gewünschtes Verhalten genauer spezifizieren. Durch diese Darstellung bleibt die Abstraktion der Muster von der zugrundeliegenden Technologie erhalten. Somit ist zur Modellierungszeit kein Wissen über die konkret zur Implementierung eingesetzte Technologie der Integrationsinfrastruktur nötig. Dieses neue Vorgehen ermöglicht daher Systemarchitekten Integrationslösungen zu modellieren, die automatisiert in ausführbare Anwendungen münden. Mit Hilfe dieses Vorgehens ist es außerdem möglich, ausführbare Systeme für verschiedene Zielplattformen zu erzeugen, die jeweils auf unterschiedlichen Integrationsinfrastrukturen ausgeführt werden können. Ein Systemarchitekt ist also zur Modellierungszeit nicht an eine bestimmte Zielplattform gebunden, sondern kann diese erst vor dem eigentlichen Ausrollen (also vor der Generierung) bestimmen.

1.2. Motivierendes Beispielszenario

In diesem Abschnitt wird ein Szenario zur Motivation der Arbeit eingeführt. Darüber hinaus wird dieses Szenario im weiteren Verlauf der Arbeit in verschiedenen Kapiteln immer weiter verfeinert, so dass zum Abschluss der Arbeit ein Gesamtbild entstehen wird, anhand dessen die einzelnen Bestandteile der Arbeit und die Zusammenhänge erfasst werden können.

Das Szenario beschreibt einen Geschäftsvorfall der Bank *MehrVomGeld*. Diese Bank hat die Finanzkrise sehr gut überstanden und war in der Lage, weitere Banken zu übernehmen und damit ihr Portfolio auszubauen. Aus diesem Grund kann sie (unter anderem) verschiedene Kreditarten anbieten, die auf die Bedürfnisse bestimmter Kundenkreise angepasst sind. Es wurde eigens eine Abteilung gegründet, die für die Vermittlung geeigneter Kreditangebote zuständig ist (Abteilung Darlehensvermittlung). Diese Abteilung erhält Kundenanfragen und ermittelt daraufhin die besten Kreditoptionen.

Die IT Abteilung von MehrVomGeld wurde beauftragt, ein entsprechendes System zu entwickeln, mit dem Kundenanfragen möglichst einfach und schnell bedient werden können. Dieses System soll die bereits existierenden Anwendungen beziehungsweise deren Dienste der unterschiedlichen Banksparten verwenden. Um dieses Integrationsproblem möglichst verständlich zu formulieren und eine geeignete Lösung zu spezifizieren, verwendet der Systemarchitekt Integrationsmuster (1.1).

Die Lösung sieht vor, dass nach dem Eingang einer Anfrage zunächst ein externer Dienst aufgerufen wird, der die Kreditwürdigkeit des Kunden ermittelt und diese Information der Anfrage hinzufügt. Entsprechend der Kreditforderung (Höhe des Kredits und Kreditwürdigkeit) werden die verschiedenen Banksparten ermittelt, die ein entsprechendes Kreditangebot übermitteln sollen. Nachdem alle Sparten mit einem Angebot geantwortet haben, werden diese ausgewertet. Dabei wird das beste Angebot für den Kunden automatisch ausgewählt und an den Bearbeiter zurückgesendet.

Diese Darstellung soll den Entwicklern als Spezifikation dienen, mit der sie den neuen *Darlehensvermittlungsdienst (Darvien)* umzusetzen haben. Allerdings bleiben den Entwicklern viele Fragen offen. Das Modell trifft zum Beispiel keine

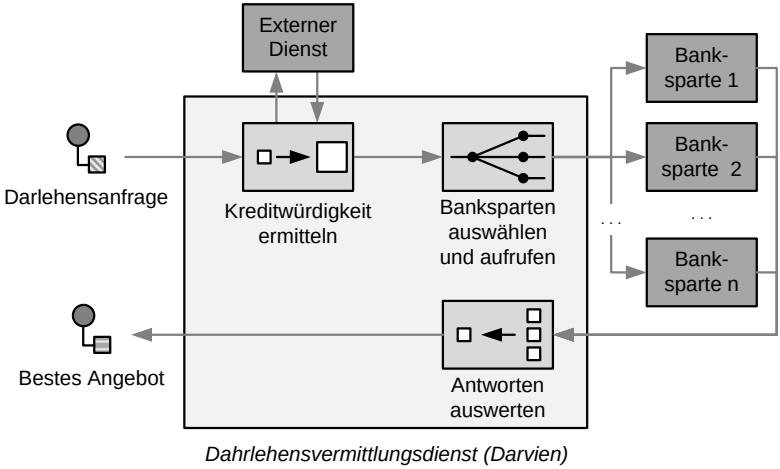


Abbildung 1.1.: Dahrlehensvermittlungsdienst Darvien der Bank MehrVomGeld

Aussage darüber, welcher Dienst zur Ermittlung der Kreditwürdigkeit angefragt werden soll, wie die eingehenden Nachrichten formatiert sind oder wie die unterschiedlichen Banksparten ausgewählt werden sollen. Darüber hinaus ist nicht klar, wie Änderungen am System eingearbeitet werden. Es fehlt somit eine einheitliche Methode mit einem Entwicklungsprozess, die diese offenen Punkte klären kann.

1.3. Beiträge der Arbeit

Die Beiträge dieser Dissertation sind eine Modell-getriebene Methode zur Erstellung ausführbarer Integrationslösungen, die Parametrisierung von Integrationsmustern, die Generierung von ausführbaren Artefakten für unterschiedliche Integrationsinfrastrukturen aus parametrisierten Integrationsmustern heraus, ein graphisches Werkzeug zur Modellierung von Integrationslösungen und ein allgemeines Format, um Integrationslösungen auf Basis parametrisierten Integrationsmustern in unterschiedlichen Werkzeugen einsetzen zu können. Weitere Beiträge der Arbeit haben sich aus den Hauptbeiträgen abgeleitet.

Diese sind (i) die Analyse von Instanz-basierter Verarbeitung von Nachrichten versus der Verarbeitung von Nachrichten im instanzlosen Pipes-and-Filters Ansatz, (ii) ein allgemeiner Ansatz zum Routing von SOAP Nachrichten und (iii) ein Architekturansatz samt einem entsprechenden Werkzeug zur Nutzung von Integrationslösungen im „as a Service“-Modus.

1.3.1. Modell-getriebene Methode zur Erstellung von Integrationslösungen

In dieser Arbeit wird eine Methode eingeführt, die die Modellierung von Integrationslösungen auf Basis parametrisierbarer Integrationsmustern ermöglicht. Die Muster werden dabei in einem Metamodell dargestellt, das von der Technologie der Integrationsinfrastruktur abstrahiert, in der das Modell implementiert werden soll. Während der Modellierung werden einzelne Integrationsmuster miteinander verbunden und somit zu einer so genannten Integrationslösung zusammen gefügt. Dieses Modell der Integrationslösung kann von Generierungsalgorithmen verwendet werden, um ausführbare Anwendungen zu erstellen. Ein Modell kann dabei von verschiedenen Algorithmen verwendet werden und daher in verschiedene Ausführungsumgebungen übersetzt werden. Als Grundlage dieser Methode dient ein Integrationsprozess, der alle Phasen beschreibt, die durchlaufen werden müssen, um ausführbare Integrationslösungen erstellen zu können.

1.3.2. Parametrisierung von Integrationsmustern

Integrationsmuster stehen üblicherweise in Form von graphischen Notationen und textueller Dokumentation zur Verfügung. In dieser Arbeit wird ein Metamodell für Integrationsmuster entwickelt, welches insbesondere deren Semantik definiert. Diese eindeutige Definition der Semantik der Integrationsmuster ist Grundlage der oben skizzierten Generierungen. Integrationsmuster sind variabel gestaltet und erfordern Entscheidungen, die ihr konkretes Verhalten repräsentieren. Diese Variabilitätspunkte sind bisher lediglich in der Dokumentation der Integrationsmuster beschrieben. Das in dieser Arbeit vorgestellte Metamodell erlaubt, solche Variabilitätspunkte präzise zu spezifizieren. Dadurch werden Integrationsmuster zu parametrisierbaren Integrationsmustern:

Parameter repräsentieren die Variabilitätspunkte. Für jedes vorhandene Muster werden entsprechende Parameter in das Metamodell aufgenommen.

1.3.3. Generierung von ausführbaren Integrationslösungen

Integrationslösungen, die auf Basis parametrisierbarer Integrationsmuster modelliert wurden, können durch die in dieser Arbeit entwickelten Algorithmen in ausführbare Artefakte überführt werden. Entscheidungen über die Integrationsinfrastruktur, die diese Artefakte letztendlich ausführen, müssen erst zum Zeitpunkt der Generierung zur Verfügung gestellt werden. Dadurch sind die Integrationslösungen technologieunabhängig, so dass unterschiedliche Integrationsinfrastrukturen als Laufzeitumgebung für dieselbe Integrationslösung verwendet werden können.

1.3.4. Architektur und prototypische Realisierung eines Werkzeugs zur Modellierung von Integrationslösungen

Im Rahmen dieser Arbeit wurde ein Werkzeug entwickelt, welches die Spezifikation parametrisierbarer Integrationsmuster und die Komposition solcher Integrationsmuster in Integrationslösungen gestattet. Darüber hinaus wurden zwei Algorithmen zur Generierung von Integrationslösungen in das Werkzeug eingebaut, so dass eine durchgängige Werkzeugkette von der Modellierung bis hin zur Ausführung besteht.

1.3.5. Austauschformat für Integrationslösungen

Damit man bei der Modellierung und Ausführung von Integrationslösungen nicht auf ein proprietäres Werkzeug festgelegt ist, wurde ein einheitliches Format entwickelt, durch das Integrationslösungen zwischen verschiedenen Werkzeugen ausgetauscht werden können. Das existierende Metamodell für Integrationsmuster gibt bereits einen Rahmen vor, und so wurde aufbauend auf diesem Metamodell eine Beschreibung in XML Schema entwickelt. Eine Integrationslösung, die diesem Schema genügt, kann nun von verschiedenen Werkzeugen importiert und anschließend verwendet werden.

Darüber hinaus wurde für die Spezifikation der benötigten Elemente zur Modellierung und Ausführung von Integrationslösungen ein Paketformat entwickelt, das alle benötigten Teile beschreibt: *EMod (EAI System Model)*. Ein EMod beinhaltet neben dem Metamodell der parametrisierbaren Integrationsmuster die eigentliche Integrationslösung in Form des allgemeinen Austauschformates, ein Modellierungswerkzeug und die so genannte *EAI Patterns Runtime*, die die modellierten Integrationslösungen in ausführbare Anwendungen überführt.

1.3.6. Vergleich und Analyse der PaF- und Workflow-basierten Verarbeitung von Nachrichten

Heutige Integrationslösungen basieren auf der Pipes-and-Filters Architektur, die die Komposition von Integrationsmustern via Kanäle vorsieht. Dem entsprechend basieren die Integrationsmuster ebenfalls auf dieser Architektur. Als Integrationsinfrastruktur wird eine nachrichtenorientierte Middleware vorausgesetzt. Middleware zur Unterstützung der *Service-orientierten Architekture (SOA)* sehen Kompositionen über Workflows vor. In dieser Arbeit wird u.a. eine Generierung vorgestellt, die Integrationsmuster auf Geschäftsprozesse abbildet. Da sich PaF und SOA unterscheiden, werden beide Ansätze konzeptionell verglichen und deren Leistungsfähigkeit analysiert.

1.3.7. Ansatz zum Routing von SOAP Nachrichten

In SOA basierten Integrationslösungen spielt das Routing von SOAP Nachrichten eine wesentliche Rolle. Integrationsmuster können zur Modellierung von Routingregeln verwendet werden. In dieser Dissertation werden verschiedene Ansätze zum Routing von SOAP Nachrichten vorgeschlagen und nach Durchführbarkeitskriterien analysiert. Der erfolgversprechendste Ansatz wird prototypisch umgesetzt.

1.3.8. Nutzung von Integrationslösungen im „as a Service“-Modus

Neben dem traditionellen Vorgehen der Modellierung eines Integrationslösungen mit Hilfe eines Modellierungswerkzeugs wurde auch ein Vorgehen

entwickelt, das neuen Anforderungen beziehungsweise Möglichkeiten im Bereich der Bereitstellung von Software gerecht wird. Software wird heute typisch *„on-premises“* verwendet. Das bedeutet, dass Software gekauft und auf Systemen installiert wird, die sich unter Kontrolle des Käufers befinden. Der aktuelle Trend ist, dass sich die Art der Bereitstellung verändert: die Software wird nur noch gemietet solange man sie auch tatsächlich benötigt (*„on-demand“*). Dabei ist die Software auf Systemen des Herstellers oder bei spezialisierten Anbietern installiert. Diese Form der Ausführung und Bereitstellung der Anwendungen wird unter dem Oberbegriff *Software as a Service (SaaS)* beschrieben. Allerdings sehen sich Firmen auch bei dieser Form der Bereitstellung mit der Integrationsproblematik konfrontiert. Anwendungen unterschiedlicher Anbieter müssen miteinander integriert werden, damit bestimmte Geschäftsprozesse erfolgreich ablaufen können. Mit dem so genannten *EaaS (EAI as a Service)* soll diesen neuen Anforderungen Rechnung getragen werden. Bei EaaS werden einzelne Integrationsmuster bis hin zu kompletten Integrationslösungen auf einer entsprechenden Infrastruktur zur Verfügung gestellt. Diese Infrastruktur wird unter dem Oberbegriff Cloud zusammengefasst [Ley09][LKN+09]. Mit EaaS können somit Integrationslösungen auf Basis parametrisierbarer Integrationsmuster erstellt werden. Diese werden anschließend automatisiert in der Cloud installiert und dort ausgeführt.

1.4. Aufbau der Arbeit

Zunächst werden in Kapitel 2 die in dieser Arbeit betroffenen Themengebiete (wie z.B. EAI) erörtert, um ein einheitliches Verständnis der eingeführten Begriffe und Technologien zu schaffen. Dieses Kapitel beinhaltet auch verwandte Arbeiten und die Diskussion, wie sich der vorgestellte Ansatz von existierenden Arbeiten unterscheidet.

In Kapitel 3 wird die allgemeine Vorgehensweise zur Erstellung von ausführbaren Integrationslösungen beschrieben. Hierbei wird erläutert, wie die parametrisierten Integrationsmuster aufgebaut sind und wie das zugrundeliegende Metamodell definiert ist.

Kapitel 4 befasst sich im Detail mit den parametrisierten Integrationsmus-

tern. Es wird zunächst beschrieben, wie atomare Muster parametrisiert werden können. Darauf aufbauend werden zusammengesetzte Muster definiert. Die vorhandene Literatur über Integrationsmuster beschäftigt sich nur am Rande mit der Behandlung von Ausnahmefällen. Da dies aber eine essentielle Eigenschaft in Geschäftsabläufen ist, auf die man explizit eingehen muss, befasst sich ein Unterkapitel gezielt mit dieser Problemstellung. Es wird aufgezeigt, welche Ausnahmesituationen (Fehlerfälle) auftreten können, wie diese mit den aktuellen Mitteln behandelt werden und welche Möglichkeiten in den existierenden Mustern fehlen. Das Kapitel schließt mit weiteren neu entwickelten Mustern, die in erster Linie die Modellierung durch den Systemarchitekten erleichtern.

Integrationsmuster basieren auf der Pipes-and-Filters Architektur. Moderne Integrationsinfrastrukturen werden heutzutage nach dem Architekturprinzip der Dienstorientierung (SOA) erstellt, die Geschäftsprozesse (*workflows*) als Integrationsschicht vorsieht. Diese beiden Architekturen sind allerdings von Grund auf verschieden. Daher beschäftigt sich Kapitel 5 mit dem Vergleich der beiden Architekturstile. Anhand eines Beispielszenarios werden verschiedene Implementierungen (jeweils auf Basis der entsprechenden Architektur) vorgestellt und diese mit Hilfe von entsprechenden Testfällen miteinander verglichen. Das Ergebnis dieses Kapitels ist zunächst ein theoretischer Vergleich der Charakteristika, sowie das Aufzeigen der Vor- und Nachteile der beiden Architekturstile und wird durch eine Leistungsfähigkeitsanalyse abgeschlossen.

Das Kapitel 6 präsentiert die Überführung des Metamodells der parametrisierbaren Integrationsmuster mit Hilfe eines auf MDD basierten Ansatzes in verschiedene ausführbare Integrationslösungen. Es wird hierbei zunächst das EAI System Model (EMod) eingeführt und alle benötigten Elemente dieses Ansatzes beschrieben. Im Anschluss wird der allgemeine Ansatz in Form eines Algorithmus zur Überführung eines Modells in ausführbare Artefakte vorgestellt. Danach werden im Detail drei verschiedene Umsetzungen des allgemeinen Algorithmus präsentiert, die jeweils ausführbare Artefakte für unterschiedliche Integrationsinfrastrukturen erzeugen: (i) Web Services und BPEL, (ii) Apache Camel und (iii) EaaS.

Integrationsmuster können außerdem ebenfalls verwendet werden, um das Weiterleiten von Nachrichten auf Transportebene zu modellieren. In Kapitel 7

wird eine Möglichkeit vorgeschlagen, wie die Muster genutzt werden, um mit Hilfe eines Geschäftsprozesses Nachrichtenweiterleitung auf der Transportebene zu realisieren.

Das darauf folgende Kapitel beschäftigt sich wieder mit der Parametrisierung von Integrationsmuster. Kapitel 8 präsentiert ein Werkzeug (GENIUS), mit dessen Hilfe Integrationslösungen durch parametrisierbare Integrationsmuster erstellen werden können. Das Werkzeug integriert zwei verschiedene Algorithmen, um unterschiedliche Zielplattformen zu unterstützen. In diesem Kapitel werden außerdem die Eigenschaften, Funktionalitäten und Erweiterungsmechansimen von GENIUS vorgestellt.

Mit Kapitel 9 wird die Arbeit abgeschlossen. Dabei wir zunächst eine Zusammenfassung der vorgestellten Beiträge geben. Die Arbeit endet mit dem Ausblick auf weitere Arbeiten, die an die vorliegende Arbeit anschließen können.

KAPITEL 2

GRUNDLAGEND UND VERWANDTE ARBEITEN

In diesem Kapitel werden die Grundlagen der Dissertation vorgestellt. Es gliedert sich dabei in fünf Teile. Zunächst werden die allgemeinen Probleme und Lösungsansätze in Bezug auf die Unternehmensintegration diskutiert (Abschnitt 2.1). Im nächsten Teil werden Architekturstile und Technologien vorgestellt, die im weiteren Verlauf der Arbeit benötigt werden (Abschnitt 2.2 und Abschnitt 2.3). Im Anschluss daran wird ein Überblick über die Herkunft des Begriffs Muster sowie existierende Muster aus verschiedenen Gebieten der IT gegeben (Abschnitt 2.4). Insbesondere werden dabei die Integrationsmuster beschrieben (Abschnitt 2.5). Daran schließt sich die Betrachtung von Modell-getriebener Entwicklung an (Abschnitt 2.6). Das Kapitel endet in einer Zusammenfassung und Abgrenzung verwandter Arbeiten (Abschnitt 2.7).

2.1. Enterprise Application Integration

Unternehmen versuchen, zunehmend Standard-Anwendungen zu kaufen oder zu mieten, anstatt eigene Anwendungen zu entwickeln. Das Ergebnis daraus ist, dass zunehmend Produkte eingesetzt werden, die nur bestimmte Aufgabenbereiche eines Unternehmens lösen, beispielsweise innerhalb der Buchhaltung oder Personalverwaltung. Für gewöhnlich verwaltet jedes dieser Produkte seine eigene Datenbasis in einer proprietären Form, die nur das Produkt selbst versteht. Allerdings müssen die Daten einer Anwendung in anderen Anwendungen weiter verarbeitet werden. Der Begriff *Enterprise Application Integration (EAI)* wurde durch das Bemühen geprägt, diese vielen unterschiedlichen Anwendungen, die nicht notwendigerweise für eine Zusammenarbeit entworfen wurden und auch nur Teilaufgaben von Geschäftsprozessen abdecken, dazu zu bringen, in einheitlichen Geschäftsprozessen zusammenzuspielen. EAI beschäftigt sich damit, heterogene Anwendungen eines Unternehmens so zu integrieren, dass sie sich möglichst so verhalten, als wären sie von Anfang an dafür entworfen worden [Kel02]. Dies erfordert, dass verschiedenartige Anwendungen miteinander verbunden werden müssen, um in einer großen integrierten Lösung aufzugehen. Diese Integration wird typischerweise durch eine so genannte Middleware realisiert [RMB00]. Dies ist eine anwendungsunabhängige Software, die Dienste zur Verfügung stellt, um Daten zu transportieren, transformieren und weiterzuleiten.

2.1.1. Herausforderungen der Integration

Per Definition muss sich EAI einer großen Anzahl von verschiedenen Anwendungen annehmen, die auf verschiedenen Plattformen an unterschiedlichen Orten ausgeführt werden. Es gibt Softwareanbieter von EAI Suiten, die eine Integration über verschiedene Plattformen und Implementierungssprachen und Schnittstellen zu populären Standard-Geschäftsanwendungen anbieten. Diese Anbieter geben Methodologien oder „Best Practices“ vor. Diese Richtlinien sind aber von den eigentlichen Produkten getrieben und bieten daher kein allgemein gültiges Prinzip zur Erstellung von Integrationslösungen [HW03].

Die Herausforderungen bei der Integration heterogener verteilter Systeme sind vielfältig [BSML06]. Zunächst muss man den gewünschten Grad der Abstraktion bei der Konzeption bestimmen. Je nach Verfeinerungsgrad können die Integrationslösungen ausführlicher und unübersichtlicher oder auch einfacher und überschaubarer werden. Unabhängig von der eingesetzten Technologie zur Integration verschiedener Systeme, kann dies eine Herausforderung darstellen. Nutzen zum Beispiel zwei Firmen, die ihre Systeme miteinander verbinden möchten, die gleiche Art der Kommunikationstechnologie (zum Beispiel Messaging), heißt das nicht, dass diese Systeme auch miteinander arbeiten können. Unterschiedliche Hersteller legen Spezifikationen unterschiedlich aus, was zu Interoperabilitätsproblemen führt. Dies sind nur einige der Herausforderungen, die bei EAI auftreten.

2.1.2. Arten der Integration

Integration bedeutet im weitesten Sinne die Verbindung zwischen Computersystemen, Firmen oder sogar Menschen. Zur Lösung der Integrationsherausforderungen gibt es sechs Ansätze, die in der Literatur am häufigsten auftreten [HW03][Kel02]: Informationsportale, Datenreplikationen, geteilte Geschäftsfunktionen, Service-orientierte Architekturen, verteilte Geschäftsprozesse und Business-to-Business Integration. Viele Integrationsprojekte bestehen aus einer Kombination dieser verschiedenen Typen.

Ausgehend von der drei-Schichten-Architektur in Informationssystemen (Benutzungsschnittstelle beziehungsweise Präsentation, Anwendungslogik, Datenhaltung), gibt es prinzipiell drei verschiedene Methoden [Kel02][BSML06], um die Anbindung von Anwendungen in Integrationslösungen zu ermöglichen: die Anbindung kann über die Präsentationsschicht erfolgen. Hier spricht man auch von *Screen Scraping*, also dem Abgreifen von Informationen an der Benutzungsoberfläche (bei einer Web Seite etwa der HTTP Datenstrom). Die zweite Methode bindet Anwendungen über Funktionsaufrufe oder Nachrichten ein. Diese funktionale Integration lässt sich allerdings nur dann realisieren, wenn die Anwendung zum Beispiel eine Programmierschnittstelle bietet, die die benötige Funktionalität bereitstellt. Die dritte Möglichkeit der Anbindung

erfolgt über die eigentlichen Daten. Nimmt man beispielsweise eine Datenbank als Datenhaltung, kann man mit Datenbankaufrufen (etwa SQL) oder Triggern in der Datenbank die Daten direkt ändern oder über Änderungen informiert werden. Die einzelnen Vor- und Nachteile sollen hier nicht diskutiert werden; dies kann in [Kel02][HW03] nachgelesen werden.

Hat man für einzelne Anwendungen den geeigneten Typ und die Methode der Integration gewählt, steht man vor der technischen Herausforderung, diese Anbindung umzusetzen. Über die Zeit hinweg haben sich zahlreiche Kommunikationsformen bewährt, die sich in vier Kategorien einordnen lassen [HW03]: Dateitransfer, geteilte Datenbanken, entfernte Methodenaufrufe (*remote procedure calls, RPC*) und *Message Queuing (MQ)*. Viele Integrationslösungen verwenden auch hier einen Mix der verschiedenen Technologien.

In dieser Arbeit wird nicht diskutiert, welche Integrationsmethode oder welcher Stil besser geeignet wäre, um ein EAI Problem zu lösen. Vielmehr wird davon ausgegangen, dass die entsprechenden Systeme mit Hilfe von Schnittstellenbeschreibungen definiert und dadurch aufgerufen werden können und dass die entsprechende Kommunikationsform umsetzbar ist.

2.2. Architekturstile

In diesem Abschnitt werden zwei Architekturstile und verwendete Technologien, die im weiteren Verlauf der Arbeit eingesetzt werden, vorgestellt. Die *Pipes-and-Filters (PaF)* Architektur ist die Basis, auf der die Integrationsmuster aufgebaut sind. Einerseits ist Message Queuing die am häufigsten eingesetzte Kommunikationsinfrastruktur bei PaF Implementierungen. Andererseits wird die *Service-orientierte Architektur (SOA)* eingeführt, die den aktuellen Stand der Technik bei der Umsetzung großer Unternehmensarchitekturen widerspiegelt. Eine Prozess-orientierte Architektur (mit Hilfe eines *Workflow Management System (WfMS)*) spielt hierbei eine bedeutende Rolle, denn einzelne Dienste einer SOA werden in Form von Geschäftsprozessen miteinander komponiert.

2.2.1. Pipes-and-Filters Architektur

Der Architekturstil Pipes-and-Filters ist sehr einfach und besteht im Grunde genommen nur aus zwei Komponenten: den Filtern und den Verbindungen zwischen den Filtern (*pipes*) [Meu95][PW92]. Die Filter sind die aktiven Komponenten der Architektur und übernehmen in einer Integrationslösung bestimmte Teilaufgaben. Ein solcher Filter nimmt Nachrichten über die Eingangsschnittstelle an und ergänzt, verfeinert, transformiert oder bearbeitet diese. Über die Ausgangsschnittstelle wird die bearbeitete Nachricht weitergegeben. Die Schnittstellen der Filter sind sehr einfach gehalten. Sie beschreiben nur, dass Nachrichten eines bestimmten Typs angenommen und versendet werden können. Pipes stellen die Verbindung zwischen den Filtern dar und nehmen daher keine aktive Rolle in der Architektur ein. Ihre Aufgabe besteht nur darin, die Nachrichten zu transportieren.

Das Konzept der PaF wurde vor allem durch das Betriebssystem UNIX bekannt, bei dem Dateien als Quellen und Ziele der Filter, also als die Verbindungsstücke zwischen den Filtern, eingesetzt werden[Bac86]. Im Umfeld von EAI ist MQ als Kommunikationsinfrastruktur für eine PaF Architektur weit verbreitet. In diesem Fall dienen *Queues* und *Topics* als Quelle und Ziel der Filter.

Eine wichtige Eigenschaft dieses Architekturstils ist die lose Kopplung der Filter. Charakteristisch ist, dass Filter komplett unabhängig von anderen Komponenten des Systems arbeiten. Kein Filter hat Kenntnis von der Existenz der anderen Filter. Insbesondere kennt ein Filter (innerhalb einer Integrationslösung) nicht die Sender und Empfänger von Nachrichten, was eine flexible Kombination der Filter ermöglicht. Darüber hinaus erlaubt die Architektur, dass ein Filter einen Datenstrom als Eingabe verarbeiten kann. Das bedeutet, dass die Bearbeitung erfolgen kann, sobald ein bestimmter Teil der Eingangsdaten angekommen ist. Der Filter muss daher nicht auf den kompletten Datensatz warten, sondern kann direkt mit der Bearbeitung beginnen. Wird diese Eigenschaft von der PaF Implementierung umgesetzt, kann sich der Nachrichtendurchsatz erhöhen.

2.2.2. Message Queuing

Die Kommunikationsinfrastruktur, die zum Beispiel bei Integrationslösungen auf Basis von PaF oftmals zum Einsatz kommt, ist *Message Queuing (MQ)* oder kurz *Messaging* [BHL95]. Die MQ Technologie ermöglicht die asynchrone und zuverlässige Kommunikation zwischen mehreren Systemen. Dies wird durch zwei Paradigmen ermöglicht: *„send-and-forget“* und *„store-and-forward“* (siehe Abbildung 2.1). Das erste Prinzip bedeutet, dass ein Sender eine Nachricht über ein *MQI (Message Queuing Interface)* in das MQ System (beziehungsweise ein *MQM, Message Queue Manager*) sendet und direkt mit seiner Arbeit fortfahren kann, sobald er eine Empfangsbestätigung erhält. MQ garantiert obendrein, dass die Nachricht nicht verloren geht und irgendwann ausgeliefert wird. Das zweite Prinzip bedeutet, dass Nachrichten zwischen MQ Systemen zuverlässig versandt werden. Eine Nachricht wird erst dann aus einer Queue gelöscht, wenn das Empfangssystem dieselbe Nachricht persistent abgelegt hat.

Asynchrone MQ Architekturen haben sich als beste Basis für EAI herausgestellt, da sie lose gekoppelte Systeme ermöglichen und die Beschränkungen der Remote Kommunikation, wie etwa Latenz und Unzuverlässigkeit, überwinden [HW03].

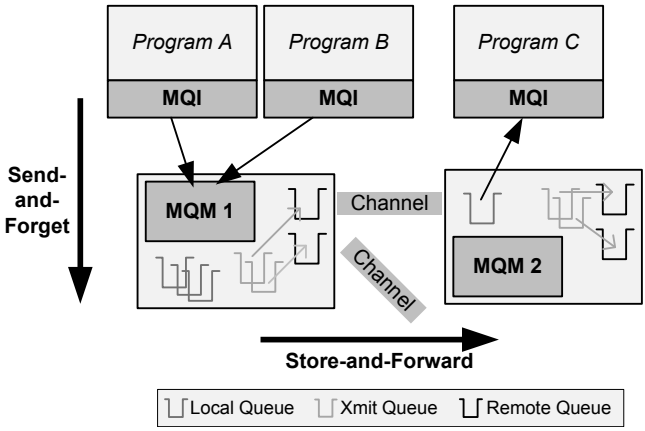


Abbildung 2.1.: Messaging Queuing Übersicht

Leider hat asynchrones Messaging auch eine Reihe von Nachteilen. Dies wird in lose gekoppelten Systemen auch als „Architect's dream, Developer's nightmare" bezeichnet [FRF02][Hoh07] und trifft auch für andere Architekturstile (wie etwa SOA) zu. Viele Annahmen, die bei der Entwicklung von einzelnen synchronen Anwendungen gelten, treffen hier nicht mehr zu. Die architektonischen Prinzipen der losen Kopplung und Indirektion – ein Sender kennt oftmals den Empfänger nicht und ist meist nicht dazu gedacht, auch Nachrichten anzunehmen – reduzieren die Annahmen, die Systeme gegenüber einander treffen, und steigern dadurch die Flexibilität. Allerdings erhöhen sie damit auch die Schwierigkeiten, ein solches System zu testen und zu überwachen [Hoh05]. MQ garantiert nur die Auslieferung der Nachricht, jedoch nicht, dass die Nachricht den eigentlichen Empfänger erreicht – bei Nicht-Verfügbarkeit landet eine Nachricht zum Beispiel in der Dead Letter Queue [MHC00][Kay03] – und auch nicht, in welcher Reihenfolge Nachrichten ausgeliefert werden. Dies erschwert es, zum Beispiel geeignete Testfälle zu entwickeln, um überprüfen zu können, ob der zeitliche Ablauf des Systems (zum Beispiel einer Integrationslösung) korrekt funktioniert.

2.2.3. Service-orientierte Architektur

Eine Service-orientierte Architektur [Erl05] ist mittlerweile der vorwiegende Standardstil, um Unternehmensarchitekturen zu erstellen. Eine solche Architektur baut sich dabei aus verschiedenen Diensten auf, die bereits existieren oder neu entwickelt werden. Ein Dienst wird dabei durch eine wohl definierte Schnittstelle beschrieben und bildet Geschäftsvorfälle eines Unternehmens ab. Die Dienste unterscheiden sich in der Granularität: atomare Dienste repräsentieren kleinste Vorgänge (etwa Abbuchung), wohingegen sich zusammengesetzte Dienste (*composite services*) aus anderen Diensten aufbauen. Die Schnittstellenbeschreibung definiert die funktionale Sicht auf den Dienst. Sie stellt noch keine Anforderungen, wie ein Dienst letztendlich implementiert werden soll.

Neben den Diensten besteht eine Architektur aus den Anbietern, die einen Dienst zur Verfügung stellen und Nutzern, die bestimmte Dienste suchen und anschließend aufrufen. Die Suche nach Diensten erfolgt über ein Register (*re-*

gistry), in der Dienstanbieter ihre Dienste veröffentlichen. Die Dienste werden dabei sowohl funktional als auch nicht funktional beschrieben (etwa durch bestimmte Dienstgüteeigenschaften, die ein Dienst erbringt). Diese drei Beteiligten gehen auch aus dem so genannten SOA-Dreieck hervor [Pap07] und werden durch unterschiedliche Schritte beschrieben: *„publish“* zur Veröffentlichung, *„find“* zum Auffinden und *„bind“* zum eigentlichen Aufruf des Dienstes.

Eine entscheidende Eigenschaft einer SOA ist die lose Kopplung [Kay03]: Dienste haben keinen Einfluss auf andere Dienste (wenn sich zum Beispiel die Schnittstelle eines Dienstes ändert) und haben keine Kenntnis über die Existenz anderer Dienst. Darüber hinaus werden sie nicht zwangsläufig entwickelt, um mit anderen Diensten zusammenarbeiten zu müssen. Allerdings müssen die Dienste miteinander verbunden werden, damit ein Geschäftsprozess ausgeführt werden kann – wie bereits erwähnt, repräsentieren Dienste kleine Geschäftsvorfälle. Somit müssen auch in einer SOA Dienste integriert werden. Typischerweise erfolgt diese Verbindung durch Komposition mit Hilfe eines Prozesses (*workflow*) [LR00][Pap07].

2.2.4. Workflow Management System

Damit innerhalb einer SOA ein Geschäftsprozess ausgeführt werden kann, wird ein *Workflow Management System (WfMS)* benötigt [LR00]. Ein solches System ermöglicht die Ausführung von Prozessen unter bestimmten Dienstgüteeigenschaften und nicht funktionalen Eigenschaften. Viele WfMS, wie beispielsweise der IBM Process Server [IBM09b], werden als Server oberhalb eines Anwendungsservers entwickelt und verwenden Transaktionen, um die nötige Robustheit zu unterstützen [LR00].

Die Struktur eines Geschäftsprozesses eines Unternehmens wird durch ein Prozessmodell beschrieben. Dieses Modell definiert alle möglichen Pfade, die in einem Geschäftsprozess durchlaufen werden können. Darin enthalten sind alle Regeln, die spezifizieren, welche Pfade wann (zeitlich und kontextabhängig) abgelaufen werden sollen und alle Aktivitäten, die entlang eines Pfades bearbeitet werden sollen. Das Modell dient als Vorlage, aus der jeder Prozess instanziiert wird. Eine Prozessinstanz wird somit auf Basis eines Prozessmo-

dells erzeugt. Ein WfMS verwendet ein Prozessmodell, navigiert anhand der Struktur durch diesen Prozess und führt die jeweiligen Aktivitäten aus.

2.3. Technologien

In diesem Abschnitt werden Technologien beschrieben, die verwendet werden, um eine der beschriebenen Architekturen beziehungsweise die entsprechenden Systeme umzusetzen.

2.3.1. Web Services

Der so genannte *Web Service Stack (WS-*)* fasst eine Menge von unterschiedlichen Technologien zusammen, mit denen man zum Beispiel eine SOA implementieren kann [WCL+05][Pap07]. Die Dienste einer SOA werden in WS-* als Web Services bezeichnet. Deren Schnittstellen werden mit Hilfe der *Web Service Description Language (WSDL)* [CCMW01] beschrieben. Dienste können in einem Verzeichnis veröffentlicht werden, das dem UDDI (Universal Description Discovery and Integration) Standard genügt. Nicht-funktionale Eigenschaften, die ebenfalls in diesem Verzeichnis abgelegt werden können, sind im WS-* Stack durch den *Web Service Policy (WS-Policy)* [W3C07b] Standard repräsentiert.

WS-* trifft keine Aussage darüber, in welcher Implementierungssprache der Stack und insbesondere die einzelnen Dienste umgesetzt werden müssen. So gibt es entsprechende Implementierungen auf unterschiedlichen Plattformen (zum Beispiel Java oder C#). Damit die heterogenen Plattformen über Hard- und Softwaregrenzen hinweg miteinander kommunizieren können, wurde eine Protokoll geschaffen, das den Nachrichtenaustausch ermöglicht: SOAP [CLS+05][W3C07a], eine Spezifikation des WS-* Stacks (siehe Abschnitt 2.3.3).

2.3.2. Web Service Business Process Execution Language

Der Standard zur Modellierung von Web Service Kompositionen in Form eines Geschäftsprozesses in einer durch WS-* realisierten SOA ist *WS-BPEL (Web*

Service Business Process Execution Language) [Org07b]. WS-BPEL ist eine XML basierte Sprache und besitzt ein flexibles Aggregationsmodell. Denn WS-BPEL erlaubt die rekursive Aggregation von Web Services zu einem neuen Web Service: ein WS-BPEL Prozess steht nach außen hin als Web Service zur Verfügung.

Mit WS-BPEL können komplexe Geschäftsprozesse beschrieben und ausgeführt werden, die aus Aktivitäten und Kontrollflussverbindungen bestehen. Die Sprache beinhaltet zwei Arten von Aktivitäten: (i) einfache atomare Elemente, um mit Web Services zu interagieren (*interaction activities*), Daten zu manipulieren, Fehler und Kompensationen zu behandeln und auf Ereignisse zu reagieren, und (ii) strukturierte Elemente, um den Kontrollfluss zu modellieren (sowohl Graph- als auch Block-basiert).

2.3.3. SOAP

SOAP [W3C07a] definiert ein XML basiertes Nachrichtenformat, beschreibt Regeln, wie diese Nachrichten von SOAP Knoten bearbeitet werden müssen, und einen Mechanismus, um Nachrichten über unterschiedliche Transportprotokolle zu transportieren (*SOAP processing model*). Eine SOAP Nachricht wird vom Sender (*initial sender*) zu einem Empfänger (*ultimate receiver*) entlang eines Nachrichtenpfades über mögliche Zwischenknoten (*intermediaries*) versendet (siehe Abbildung 2.2). Das Transportprotokoll sowie die Dienstgüteeigenschaften können entlang des Nachrichtenpfades zwischen zwei Knoten verschieden sein. Neben der Weiterleitung einer Nachricht kann ein Zwischenknoten auch zusätzliche Dienste (*additional services*) auf einer Nachricht anwenden: zum Beispiel Verschlüsselung, Logging oder Transformation. Dies wird durch die *SOAP Message Header* ermöglicht. Eine SOAP Nachricht besteht immer aus einem (optionalen) Kopf (*Header*), der Metadaten enthält, und dem *Body*, der den eigentlichen Inhalt der Nachricht aufnimmt. Durch die Kopffelder kann das SOAP Abarbeitungsmodell erweitert werden und die Bearbeitung entsprechend der darin enthaltenen Informationen an einem SOAP Knoten erfolgen.

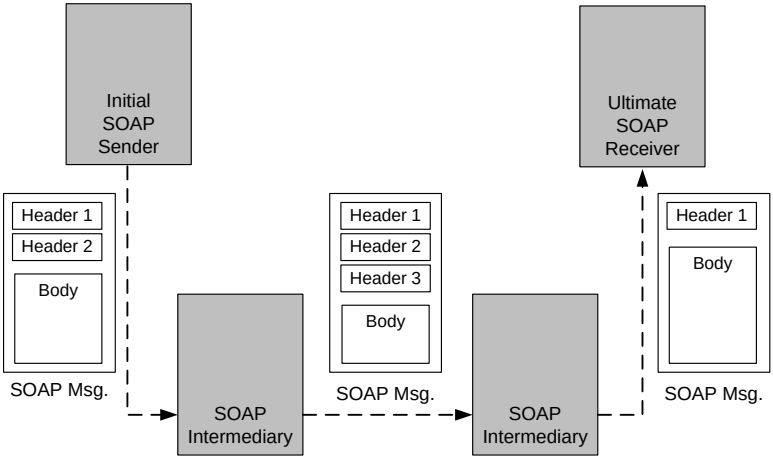


Abbildung 2.2.: SOAP Abarbeitungsmodell

2.4. Muster

Der Begriff des Musters (*pattern*) wie er im IT Umfeld verwendet wird, wurde in den späten siebziger Jahren durch den Hausbauarchitekten Alexander geprägt [Ale77][Ale79]. Seine Bücher beschreiben Muster im Bereich des Hausbaus und der Stadtplanung (zum Beispiel Bettnische, Balkon, Gehweg, usw.). Jedes Muster repräsentiert dabei eine Entscheidung, die ein Nutzer treffen muss, sowie die entsprechenden Überlegungen, die zu dieser Entscheidung führen. Eine Mustersprache (*pattern language*) ist eine Matrix aus zueinander in Beziehung stehenden Mustern, in der jedes Muster typischerweise zu einem anderen Muster führt und den Nutzer so durch den Entscheidungsfindungsprozess leitet. Ein ähnliches Vorgehen wird auch bei architektonischen Entscheidungsbäumen (*decision trees*) [ZGTL07] verwendet, bei der eine Entscheidung auf der vorhergehenden Entscheidung aufbaut und den Nutzer so zu einer geeigneten Lösung führt. Lösungsansätze in Form von Mustern werden von Experten erstellt, die jahrelange Erfahrung mit zusammenhängend auftretenden Problemen haben. Man spricht hierbei auch von so genannten *„nuggets of advice“*. Der Ansatz der Muster ist somit eine praktikable Technik, um Exper-

tenwissen zu dokumentieren und für Nicht-Experten lesbar und verständlich zu machen [HW03].

Ein Muster ist also eine mögliche Lösung eines bestimmten Problems innerhalb eines bestimmten Kontextes [Fow06]. Muster bestehen aus verschiedenen Aspekten, die die jeweiligen Muster detailliert beschreiben. Zunächst besitzt ein Muster einen Namen, mit dem die Lösung des Musters bezeichnet wird. Weiterhin kann ein Muster durch ein kleines Symbol repräsentiert werden, mit dem es innerhalb einer Abbildung leicht zu identifizieren ist. Zwei weitere sehr wichtige Aspekte sind zum einen das eigentlich zu lösende Problem, das in Form einer Frage formuliert wird, und zum anderen der Kontext, in dem das Problem auftreten kann. Der Aspekt Lösung stellt mögliche Alternativen zur Beseitigung des Problems dar. Unter dem Aspekt Erschwernisse (*Forces*) werden die Einflüsse und Einschränkungen beschrieben, die die Lösung des Problems beeinträchtigen können. Darüber hinaus besitzt jedes Muster eine Illustration der Problemlösung in Form einer übersichtlichen Zeichnung. Neben diesen Elementen gibt es noch Aspekte, die verwandte Muster beschreiben, beziehungsweise Muster angeben, die nachfolgend angewendet werden können. Weiterhin können Beispiele angegeben werden, die Lösungen für konkrete Probleme beschreiben (etwa eine Lösung in Form einer oder mehrerer Java Klassen).

In den folgenden Abschnitten werden Muster im Bereich der Softwareentwicklung vorgestellt. Diese Auflistung erhebt keinen Anspruch auf Vollständigkeit. Sie dient der Übersicht, in welchen unterschiedlichen Bereichen mittlerweile Muster zur Unterstützung des Entwicklungsprozesses von Software existieren.

2.4.1. Architekturmuster

Architekturmuster beschreiben Probleme und mögliche Lösungsalternativen, die beim Entwurf von Softwaresystemarchitekturen entstehen. Ein solches Muster beschreibt die fundamentale, strukturelle Organisation eines Softwaresystems. Ein Muster bietet eine Menge von vordefinierten Untersystemen, spezifiziert deren Verantwortlichkeiten und beinhaltet Regeln und Richtlinien,

um die Beziehungen zwischen diesen zu organisieren. Das bekannteste Buch in diesem Bereich ist [FRF02], welches Muster zum Beispiel aus dem Bereich Domain Logic, Data Source oder objektrelationales Verhalten beschreibt.

2.4.2. Entwurfsmuster

Entwurfsmuster (*design patterns*) sind die wohl bekanntesten Muster der Informatik. Diese Muster abstrahieren von der Implementierung zur Definition eines Schemas, um ein Untersystem oder eine Komponente eines Softwaresystems oder die Beziehungen untereinander zu verfeinern. Sie beschreiben häufig vorkommende Strukturen kommunizierender Komponenten, die ein allgemeines Entwurfsproblem innerhalb eines speziellen Kontextes lösen. Die Form eines Musters ist durch Mittel des Softwareentwurfs, wie beispielsweise Objekte, Klassen oder Vererbung, geprägt [App00]. Die bedeutendsten Entwurfsmuster sind die so genannten *Gang-of-Four-Patterns* [GHJ95]. Diese Sammlung beschreibt Entwurfsmuster, die für die Objekt-orientierte Programmierung hilfreich sind.

2.4.3. Implementierungsmuster

Implementierungsmuster (*coding patterns*) werden auch Idioms genannt und sind Muster, die auf der untersten (Implementierungs-) Ebene eingesetzt werden. Sie beziehen sich auf eine spezifische Programmiersprache und beschreiben, wie man spezielle Aspekte von Komponenten oder Beziehungen zwischen Komponenten, unter Ausnutzung der spezifischen Eigenschaften der entsprechenden Programmiersprache, implementiert [App00][Bec96]. Diese Art von Mustern adressieren viele Aspekte der Entwicklung und beinhalten Klassen, Status, Verhalten, Methoden und mehr. Auf dieser Ebene werden zum Beispiel bewährte Vorschläge, zur Variablenbennenung, die bei Fehlerbehandlungen (*exceptions*) eingesetzt werden, beschrieben [Bec06].

2.4.4. Weitere Muster

Neben den bereits erwähnten Mustern gibt es auch Muster, die das Verhalten eines Geschäftsprozesses beschreiben oder solche, die die Interaktion zwischen

Diensten beschreiben. Diese Musterarten sind weniger verbreitet, dennoch durchaus bekannt und sollen daher hier nicht unerwähnt bleiben.

2.4.4.1. Geschäftsprozessmuster

Die so genannten *Workflow Patterns* [ABHK00][AHKB03] sind Muster, um Geschäftsprozesse zu beschreiben beziehungsweise zu analysieren. Diese Muster können zum Beispiel verwendet werden, um die Mächtigkeit eines WfMS zu analysieren und um zu überprüfen, welche Konstrukte durch einen Workflow innerhalb des WfMS ausgeführt werden können und welche nicht. Darüber hinaus dienen sie als Hilfestellung, um gegebene Geschäftsanforderungen möglichst effizient durch einen Geschäftsprozess zu implementieren.

2.4.4.2. Interaktionsmuster

Für den Bereich der SOA und insbesondere der Implementierung durch Web Services wurden Muster entwickelt, die die Interaktion zwischen Diensten beschreiben (*service interaction patterns*) [BDH05a][BDH05b]. Diese Muster beschäftigen sich mit den Problemen beim Entwurf und der Implementierung von Web Services beziehungsweise deren Interaktion untereinander. Die Muster gehen dabei über bilaterale Interaktionen hinaus und beschäftigen sich auch mit multilateralen, konkurrierenden und kausal abhängigen Interaktionen.

2.5. Integrationsmuster

Neben den erwähnten Mustern haben sich über die Jahre auch Muster herauskristallisiert, die die Anforderungen, Probleme und deren mögliche Lösung im Bereich der Unternehmensintegration darstellen. In [TRH+04] wird beschrieben, wie mit *Integration Patterns* eine Integrationsarchitektur entworfen und entwickelt werden kann. Sie sind nicht zu verwechseln mit den *Enterprise Integration Patterns (EIP)* aus [HW03]. Die EIP gehen über die Integration Patterns hinaus und ermitteln feingranulare (*atomare*) bis hin zu größeren, zusammengesetzten (*composed*) Mustern, die wiederkehrende Probleme bei EAI und deren potentielle Lösungen beschreiben. Die Sammlung aus [HW03]

ist noch nicht komplett und wird im Laufe der Zeit weiter ausgebaut. Die Muster sind die Grundlage dieser Dissertation.

Die EIP (siehe Abbildung 2.3) basieren auf der Pipes-and-Filters Architektur und verwenden Message Queuing als Kommunikations- und Integrationsinfrastruktur. Die Muster sind in verschiedene Kategorien unterteilt: *Message Construction*, *Message Endpoints*, *Messaging Channels*, *Message Routing*, *Message Transformation* und *System Management*. Nachfolgend werden die einzelnen Kategorien kurz beschrieben, um ein besseres Verständnis für den Hauptteil der Arbeit zu schaffen.

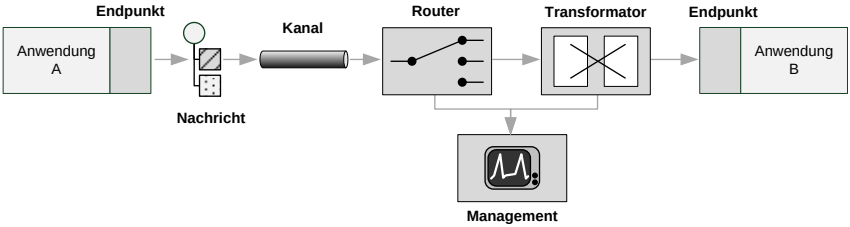


Abbildung 2.3.: Enterprise Integration Patterns Überblick

Message Construction: Diese Rubrik beschreibt die unterschiedlichen Arten von Nachrichten, die in einer Integrationslösung transportiert werden können. Eine Nachricht wird dabei als Datenpaket angesehen, das über einen Kanal versendet wird und von Filtern bearbeitet werden kann.

Message Endpoints: Die Kategorie der Endpunkte beschreibt Muster, durch die die Anbindung von Anwendungen an eine MQ Infrastruktur ermöglicht wird. Es wird durch die Muster somit eine Brücke zwischen Anwendungen und der Infrastruktur geschlagen, so dass eine Anwendung Nachrichten versenden und Nachrichten empfangen kann.

Messaging Channels: Nachrichtenkanäle beschreiben die unterschiedlichen Möglichkeiten, mit Hilfe derer Nachrichten versendet werden können. Ein Kanal verbindet immer einen bis mehrere Empfänger mit einem bis mehreren Sendern. Ein solcher Kanal kann bestimmte Dienstgüteeigenschaften besitzen, bestimmte Aufgaben übernehmen oder nur bestimmte Arten von Nachrichten übermitteln.

Message Routing: Die Muster dieser Kategorie beschreiben Möglichkeiten, mit denen der Weg einer Nachricht durch eine Integrationslösung bestimmt werden kann. Der eigentliche Sender einer Nachricht kennt den Empfänger nicht. Außerdem muss eine Nachricht unter Umständen auf ihrem Weg vom Sender zum Empfänger über verschiedene Wege bearbeitet und angepasst werden. Ein Router entscheidet daher anhand von Inhalten oder dem Kontext, über welche nachfolgenden Kanäle eine Nachricht verschickt wird.

Message Transformation: Auf Grund der Tatsache, dass unterschiedliche Anwendungen unterschiedliche, meist nicht kompatible Datenformate versenden und bearbeiten, müssen Nachrichten auf dem Weg von Sender zu Empfänger transformiert werden. Damit nicht der Sender beziehungsweise Empfänger angepasst werden muss, beschreiben die Muster dieser Kategorie Lösungsvorschläge, um dieser Herausforderung zu begegnen.

System Management: Lose gekoppelte und verteilte Anwendungen sind schwieriger zu überwachen und zu kontrollieren als große monolithische Systeme. Aus diesem Grund wurden Muster entwickelt, die die Probleme der Überwachung adressieren. Es werden Lösungen vorgeschlagen, um Integrationslösungen mit Messaging als Kommunikationsinfrastruktur zu überwachen und zu kontrollieren, zu beobachten und zu analysieren, sowie zu testen und zu debuggen.

2.6. Modell-getriebene Softwareentwicklung

Modell-getriebene Softwareentwicklung (*Model-driven Software Development, MDD*) ist ein Oberbegriff für Techniken, die formale Modelle erstellen und daraus automatisiert lauffähige Software erzeugen [VS06]. MDD besteht somit aus drei Teilen: formalen Modellen, automatisierten Transformationen und lauffähiger Software. Modelle treten in verschiedenen Formen während der Softwareentwicklung auf, zum Beispiel in Form von Architekturskizzen, UML-Diagrammen oder, wie in dieser Arbeit, durch Muster modellierte Integrationslösungen. Allerdings liegen diese Modelle selten formal vor, so dass sie für eine automatisierte Transformation nicht verwendet werden können. Der erste Schritt der MDD ist somit, aus den vorhandenen Modellen ein formales Modell

zu erzeugen, das in MDD-Werkzeugen eingesetzt werden kann.

Das Ziel der MDD ist ein lauffähiges Produkt. Dieses kann in MDD durch Generieren oder Interpretieren aus einem Modell erzeugt werden. Ein Generator ist ein Werkzeug, das aus einem Modell zum Beispiel ausführbaren Quelltext erstellt. Es handelt sich also im Prinzip um einen Übersetzer. Ein Interpreter liest ein Modell zur Laufzeit ein und führt, abhängig von seinem Inhalt, verschiedene Aktionen aus. In dieser Arbeit wird der erste Ansatz verfolgt, wodurch ein Modell (d.h. eine Integrationslösung) über einen Generator in ausführbare Systeme überführt wird.

Die Übersetzung eines Modells in ein ausführbares System soll außerdem automatisiert erfolgen. Der Kerngedanke dabei ist, dass die Modelle als Spezifikation dienen und praktisch die Rolle der Quelltexte einnehmen. So werden Änderungen im Modell vorgenommen und nicht, wie bei der üblichen Art der Softwareentwicklung, im Quelltext selbst. Dadurch erreicht man, dass die Modelle immer den aktuellen Stand der Entwicklung darstellen und der generierte Quelltext somit konsistent zu den Modellen ist. Das Modell ist also gleichzeitig Spezifikation und Dokumentation [Fra03][VS06].

Fälschlicherweise wird oftmals MDD mit der Modell-getriebenen Architektur (*Model-driven Architekture, MDA*) [Obj03][Fra03] gleichgestellt. MDA ist konzeptionell eine Spezialisierung von MDD. MDA sieht vor, dass als Metamodell für Modelle immer *MetaObject Facility (MOF)* [Obj06] verwendet wird und legt den Benutzern nahe, *UML (Unified Modeling Language)* [Obj09b] als Syntax eines Modells einzusetzen. Damit ist ein Modell im Kern auf UML festgelegt. Die grundlegende Eigenschaft, dass aus Modellen ausführbare Systeme erzeugt werden, bleibt erhalten.

2.6.1. Elemente der MDD

Der MDD Ansatz besteht im Wesentlichen aus wenigen Elementen, mit denen sich die Technik umsetzen lässt (siehe Abbildung 2.4). Modelle, die übersetzt werden sollen, werden in plattformunabhängiger Weise gestaltet. Sie werden daher *Platform Independent Model (PIM)* genannt. Es werden dadurch ausschließlich die rein fachlichen Aspekte betrachtet und modelliert.

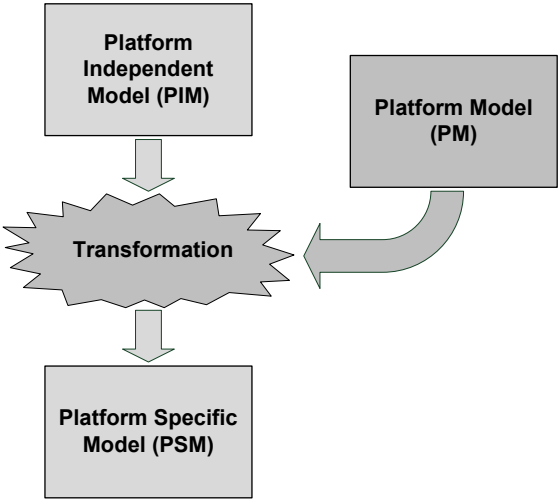


Abbildung 2.4.: Elemente des MDD Ansatzes

Ein PIM dient als Eingabe für einen *Transformator*. Ein solcher Transformator kann für ein und dasselbe PIM ausführbare Modelle für eine oder mehrere verschiedene Zielplattformen generieren. Das Produkt der Modelltransformation ist ein *Platform Specific Model (PSM)*.

Für die Generierung eines PSM wird neben dem PIM ein so genanntes *Platform Model (PM)* benötigt, um das PSM mit plattformspezifischen Aspekten anreichern zu können. PMs sind Metamodelle, die die Zielplattform des Systems beschreiben. Über die Kombination von einem PIM, also einer formalen semantischen Beschreibung der Zusammenhänge und Abläufe mit einem PM, kann letztendlich über Modelltransformation das Zielsystem (PSM) generiert werden.

2.6.2. Vor- und Nachteile bei MDD

Der MDD Ansatz bietet einige Vorteile, ist jedoch nicht ganz ohne Nachteile. In diesem Abschnitt werden einige Punkte kurz angesprochen. Eine ausführliche Diskussion kann in der entsprechenden Fachliteratur nachgelesen

werden [VS06][Fra03].

Ein Vorteil von MDD ist die Abstraktion von der Zielplattform. Dies ermöglicht eine vereinfachte Umstellung auf neue Zielplattformen, da das eigentliche Modell (PIM) nicht angepasst werden muss. Das bedeutet, dass eine Wiederverwendung von bestehenden Modellen erleichtert wird. Durch die Anpassung des PMs und der Transformationsalgorithmen kann eine neue Zielplattform eingeführt werden. Dies und der Umstand, dass neue Anforderungen einzig im Modell umgesetzt werden müssen und die ausführbaren Systeme nach der Generierung zur Verfügung stehen, führen zur Erhöhung der Entwicklungsgeschwindigkeit. Außerdem ist durch die automatisierte Generierung von PSMs eine gleichbleibende Softwarequalität gegeben. Dies ist allerdings nur dann der Fall wenn der Transformationsalgorithmus eine entsprechende Qualität besitzt.

Der letzte Punkt bezeichnet somit bereits einen Nachteil des MDD Ansatzes, denn die Güte des Algorithmus ist davon abhängig, wie viel Wissen der Entwickler des Algorithmus von der Abbildung auf die Zieltechnologie besitzt. Die Softwarequalität und damit auch die Produktivität stehen und fallen mit der Kompetenz des Entwicklers. Ein weiterer Nachteil ist, dass die Spezifikation des zu entwickelnden Systems zwar in Form der Modelle zur Verfügung steht, das System aber auf der Zielplattform ausgeführt wird. Testen und Debugging findet somit direkt in der Ausführungsumgebung statt. Eine Rückführung gefundener Fehler auf die eigentlichen Modelle und somit die Behebung dieser im Modell ist oftmals äußerst schwierig, denn Quelltext und Modell sind voneinander getrennt. Der Transformator arbeitet nur in eine Richtung, die andere Richtung wird meistens nicht unterstützt. Außerdem sind Änderungen an Modellen problematisch. Es muss ein Prozess existieren, der die Migration von bestehenden Systemen hin zu den neuen Systemen definiert und auch die Versionierung von Modellen unterstützt.

2.7. Verwandte Arbeiten

In den folgenden Abschnitten werden zunächst verwandte Arbeiten im Bereich der Modell-getriebenen Anwendungsintegration betrachtet. Im Anschluss

werden Arbeiten präsentiert, die Integrationsprobleme mit Hilfe von Integrationsmustern lösen.

2.7.1. Modell-getriebene Integration

Im Bereich der Modell-getriebenen Entwicklung existieren verschiedene Ansätze, um Integrationsproblemen zu begegnen. Die Gemeinsamkeit dieser Ansätze besteht darin, dass ein spezielles Modell oder eine Domänen-spezifische Sprache (DSL) entwickelt wird, womit die Architektur und somit die Integration bestehender Systeme beschrieben wird, um daraus eine ausführbare Integrationslösung zu erzeugen. Allerdings verwendet jeder Ansatz eigene proprietäre Modelle. Eine quasi standardisierte Notation, wie etwa die Integrationsmuster, wird nicht eingesetzt.

[BSML06] setzt zum Beispiel die proprietäre Sprache *SIML (System Integration Modeling Language)* ein. SIML ist eine DSL, die die funktionale Integration von komponentenbasierten Systemen vereinfacht. SIML abstrahiert dabei von der Technologie, in der die Integrationslösung implementiert werden soll. Innerhalb des MDD Ansatzes können verschiedene Plattformmodelle eingesetzt werden, um unterschiedliche Zielplattformen zu unterstützen, zum Beispiel Web Services implementiert in Java EJB Komponenten oder basierend auf der .NET Platform von Microsoft.

In [FYL+08] wird ein *Solution Template* für EAI eingeführt, um den Lebenszyklus einer Integrationslösung durch einen flexiblen Entwurf und Konfigurationsmöglichkeiten dieser Vorlagen zu vereinfachen. Das dazugehörige Werkzeug bietet verschiedene Ebenen der Abstraktion und wiederverwendbaren Einheiten an, um die Erstellung einer Lösungskomposition zu ermöglichen (zum Beispiel Konnektoren, Kollaborationen und Prozessmodelle). Die Vorlage dient dabei als PIM. Jede einzelne Vorlage kann mit Variabilitätspunkten auf die entsprechenden Belange angepasst werden (zum Beispiel Endpunkte von Diensten). Wenn man eine Integrationslösung mit Hilfe der Vorlagen erstellt hat, kann man dieses Modell in ein ausführbares *JEE (Java Enterprise Edition)* [Ora09] System übersetzten und auf einem entsprechenden Server ausführen. Die Erweiterung auf andere Technologien ist auch möglich.

[Her07] zeigt die Möglichkeit auf, Legacy-Anwendungen mit moderneren Frontends zu versehen. Der vorgestellte Ansatz beschränkt sich allerdings nicht auf die Benutzungsschnittstellen, sondern man kann diese auch auf Service-orientierte Architekturen anwenden. Die vorgestellte Methode modelliert dabei existierende Geschäftsprozesse in Form von PIMs und extrahiert Dienste von Legacy-Anwendungen nach dem *ADM (Architecture Driven Modernization)* Prinzip hin zu einem PIM. Die Integration erfolgt, indem diese beiden Modelle miteinander verbunden werden. Sie gehen gemeinsam in einem so genannten *Service Model* auf. Ein so erstelltes Modell kann mittels Generatoren auf verschiedene Zielplattformen überführt werden (zum Beispiel Java Application Server, IBM z/OS) und erzeugt unterschiedlichste Artefakte (Java Klassen, Portlets, Testklienten).

Die Firma E2E [E2E07] bietet mit der *E2EBridge* ein Produkt an, mit dessen Hilfe Integrationslösungen auf Basis von UML Diagrammen erstellt werden können. Diese Diagramme werden in einer *Modell-getriebenen Integration (MDI)*, basierend auf Event-getriebenen und Service-orientierten Architekturen, direkt ausgeführt [E2E06]. Dazu werden die UML Modelle in einer eigens implementierten virtuellen Maschine (*UML virtual machine*) ausgeführt. E2E setzt somit auch einen MDD Ansatz ein, verwendet dazu die allgemein bekannte UML Notation, konzentriert sich allerdings dabei sehr auf die technischen Aspekte. Die abstraktere Geschäftssicht, die durch Integrationsmuster erreicht wird, wird durch die E2EBridge nicht unterstützt. Außerdem unterscheidet sich dieser Ansatz von der Dissertation darin, dass eine proprietäre Laufzeitumgebung implementiert wurde, in der die Modelle interpretiert (nicht transformiert) werden.

2.7.2. Integration mit Hilfe von Integrationsmustern

Integrationsmuster als Grundlage der Spezifikation für direkt ausführbare Integrationslösungen werden in zwei unterschiedlichen Arbeiten verwendet. *Apache Cimero* [Apaf] und *Apache Camel* [Apab] nutzen die Muster und erstellen daraus Quelltext, der auf den entsprechenden Integrationsinfrastrukturen direkt ausgeführt werden kann. Cimero setzt *Apache ServiceMix* [Apad], einen

Enterprise Service Bus (ESB), als Infrastruktur ein. Cimero bietet einen Eclipse basierten Editor [Ecla], der dem Benutzer die Möglichkeit bietet, eine visuelle Repräsentation von Weiterleitungslogiken für ServiceMix zu modellieren. Nachdem man mit Hilfe der Integrationsmuster eine visuelle Repräsentation modelliert hat, kann man dieses Modell in ein Paketformat übersetzten, das auf ServiceMix installiert und ausgeführt werden kann. Die Zielplattform ist *JBI (Java Business Integration)* [Vin05] und eine entsprechende XML Konfigurationsdatei; Integrationsmuster werden in Anwendungen übersetzt, die auch unter dem Begriff *Service Assemblies* bekannt sind. Cimero unterstützt nur eine eingeschränkte Auswahl von Integrationsmustern, die darauf spezialisiert sind, in ServiceMix ausgeführt zu werden. Dieser Ansatz bietet somit kein generelles Konzept, um in einer anderen Zielplattform als ServiceMix verwendet zu werden.

Einen Schritt weiter geht Camel. Camel ist ein Java basiertes Open-Source Teilprojekt des Apache ActiveMQ Projektes [Apaa]. Camel unterstützt eine kleine Untermenge der Integrationsmuster und dient als Routing- und Mediationssystem. Es bietet ein Rahmenwerk, das es ermöglicht, Weiterleitungs- oder Mediationsregeln auf Basis von Integrationsmustern in einer Domänenspezifischen Sprache (DSL) oder durch eine Spring-basierte [Spr] XML Konfigurationsdatei zu erstellen. Es werden im Vergleich zu Cimero weitere Ausführungsumgebungen angeboten (ActiveMQ und ServiceMix). Camel unterstützt aber ebenfalls nur eine gewisse Untermenge von Integrationsmustern, genau die, die durch die zugrunde liegende Technologie letzten Endes bedient werden können. Somit ist auch dieser Ansatz an eine bestimmte Integrationsinfrastruktur gebunden.

In [Hoh04] wurde ein Messaging Werkzeugkasten vorgestellt, der viele der Integrationsmuster prototypisch als individuell ausführbare Komponenten implementiert. Diese Einheiten ermöglichen die Erstellung einer nachrichtenbasierten Integrationslösung, indem verschiedene Muster miteinander kombiniert beziehungsweise verbunden werden. Dies kann über die Kommandozeile erfolgen oder durch Konfigurationsdialoge, die für einige Muster existieren. Allerdings sind die Muster des Werkzeugkastens nicht generell einsetzbar. Es handelt sich lediglich um Beispiele für bestimmte Ausprägungen von Mus-

tern, die nur auf einer speziellen Infrastruktur (*Microsoft Message Queuing, MSMQ* [Mic]) ausgeführt werden können. Diese Anwendung kann somit auch nicht als genereller Ansatz dienen, der auf verschiedenen Integrationsinfrastrukturen eingesetzt werden kann.

Keiner der vorgestellten Ansätze kann somit die in dieser Arbeit vorgestellten Anforderungen erfüllen. Jeder Ansatz unterscheidet sich in der Art wie Integrationsmuster dargestellt werden, zum Beispiel als visuelle Repräsentation in einem Editor oder als Basis einer DSL, die in einer bestimmten Programmiersprache eingesetzt werden kann. Darüber hinaus scheint keiner der untersuchten Ansätze so generisch zu sein, dass mehrere verschiedene Zielplattformen unterstützt werden können. Außerdem hat keiner dieser Ansätze Anstrengungen unternommen oder angekündigt, diese Ansätze dahingehend zu erweitern.

KAPITEL 3

VORGEHENSMETHODE ZUR ERZEUGUNG AUSFÜHRBARER INTEGRATIONSLÖSUNGEN

Die Integration von verschiedenen Anwendungen ist, wie bereits erwähnt, eine sehr große Herausforderung für IT Unternehmen. Es gibt keinen Königsweg Integrationsprobleme zu lösen. Jeder Verantwortliche in diesem Bereich besitzt eine andere Herangehensweise, um eine Lösung zu konzipieren. Dabei erkennt man, dass Erfahrung eine große Rolle spielt, da wiederkehrende Problemstellungen existieren, die bereits mehrfach gelöst wurden. Das bedeutet, dass Experten typischerweise bereits zahlreiche Integrationsprobleme gelöst haben, so dass sie diese Lösungen mit neu aufkommenden Problemen vergleichen und daraus Lösungen für das aktuell betrachtete Problem ableiten können. Diese Personen kennen also die Muster in Problemstellungen und damit assoziierte Lösungen.

Die in [HW03] aufgeführten Integrationsmuster beschreiben solche wiederkehrenden Probleme und in welchem Kontext sie auftreten. Jedes Muster

bedeutet eine Entscheidung hinsichtlich einer Lösungsstrategie und erläutert die Überlegungen, die zu dieser Entscheidung geführt haben. Diese Muster stellen Problemstellungen und Lösungen auf einem hohen Abstraktionsniveau dar. Dabei werden technologische Entscheidungen weitestgehend außen vor gelassen. Das bedeutet, dass Integrationsentscheidungen auf einer abstrakten Ebene beschrieben werden, ohne darauf einzugehen, wie sie letzten Endes implementiert werden können beziehungsweise sollen.

Diese einzelnen atomaren Muster werden daraufhin miteinander verbunden und definieren eine vollständige Integrationslösung. Die einzelnen Muster stellen zusammen mit ihrer Verflechtung somit einen konzeptionellen Bauplan der Integrationslösung dar.

Die Konzeption einer Integrationslösung umfasst grundsätzlich drei unterschiedliche Ebenen (siehe Abbildung 3.1):

1. die Modellierung,
2. die Integration und
3. die Anwendungen, die Dienste oder Daten zur Verfügung stellen.

In [SL08] sind diese drei Ebenen dargestellt. Auf der obersten Ebene (Ebene 1) entwickeln und zeichnen Systemarchitekten eine Systemlandschaft in einer abstrakten Weise, um Integrationslösungen zu spezifizieren, ohne dabei auf die Technologie einzugehen, mit der diese Architektur zu einem späteren Zeitpunkt umgesetzt werden soll. Auf dieser Ebene können verschiedene Werkzeuge eingesetzt werden. Integrationsmuster sind dabei eine quasi standardisierte Möglichkeit, Integrationslösungen zu spezifizieren. Die Verwendung von Integrationsmustern auf dieser Ebene dient der Verbesserung der Kommunikation zwischen Architekten, die typischerweise kein detailliertes Wissen über die Technik der zugrundeliegenden Systeme besitzen, auf der einen Seite und dem technischen Personal (den Entwicklern) auf der anderen Seite, das diese Systeme im Detail kennt.

Auf der mittleren Ebene (Ebene 2), auch bekannt als Integrationsebene, ist die Logik beheimatet, die die einzelnen Anwendungen und deren Dienste entsprechend der Beschreibung aus Ebene 1 miteinander verbindet. Diese Integrationsebene repräsentiert die Geschäftslogik (oder Geschäftsprozesse/

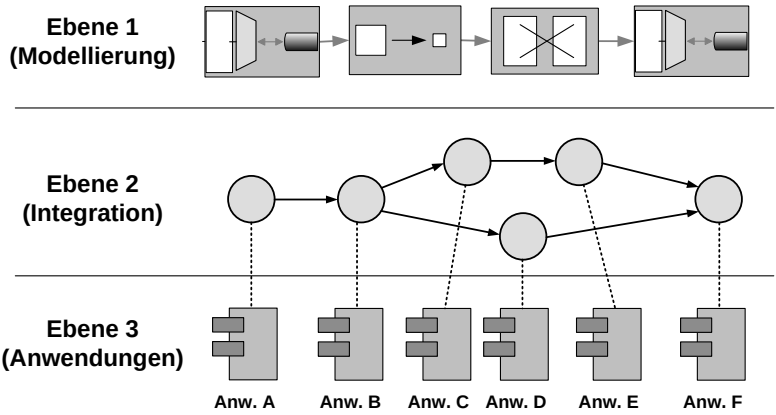


Abbildung 3.1.: Drei Ebenen bei der Erstellung von Integrationslösungen

-abläufe), mit der gewisse Geschäftsziele erreicht werden können (zum Beispiel einen PKW verkaufen). Auf dieser Ebene treten exakt die Probleme der Integration auf, wie sie in Abschnitt 2.1 beschrieben wurden (zum Beispiel verschiedene Datenformate). Unterschiedliche Herangehensweisen zur Lösung dieser Probleme können hier eingesetzt werden. In [SL05] und [SL08] wird beispielhaft ein Geschäftsprozess eingesetzt, um die Integrationsaufgaben umzusetzen. Auf der untersten Ebene (Ebene 3) befinden sich Anwendungen, die bestimmte Aufgaben innerhalb der Geschäftslogik übernehmen (zum Beispiel Datenbanken, selbst entwickelte Programme/Systeme oder Legacy Systeme). Diese Anwendungen stellen verschiedene Dienste zur Verfügung, um die entsprechenden Geschäftsziele zu erreichen. Diese Dienste werden von der Integrationsebene verwendet und somit miteinander verbunden.

Wenn man die unterschiedlichen Ebenen aus Abbildung 3.1 betrachtet, erkennt man, dass zwischen den beiden unteren Ebenen eine Verbindung besteht: die Integrationsebene ruft Dienste der darunterliegenden Ebene auf, und die Anwendungen führen bestimmte Aufgaben aus (zum Beispiel antworten sie mit entsprechenden Daten). Zwischen den Ebenen 1 und 2 existiert keine solche Verbindung: Architekturen in Form von Integrationslösungen werden mit Hilfe von Integrationsmustern entworfen, die keine Aussage über die tatsächliche

Implementierung machen. Die Entwickler, zuständig für die Erstellung der Integrationslösung, verwenden diese Architekturen und interpretieren, wie sie diese Vorgaben in der Integrationsschicht umsetzen können. Es existiert daher keine explizite Verknüpfung zwischen diesen Ebenen, lediglich eine gedankliche. Diese Problematik wird mit dem Begriff *Architecture-and-IT-Gap* bezeichnet [SL08]: es existiert eine Wissens- und Verständigungslücke zwischen den Personen, die mit der Architektur, Geschäftszielen und -prozessen vertraut sind, und den Personen, die das Verständnis der technologischen Sachverhalte haben. Letzteres ist allerdings oftmals abgekoppelt von den Geschäftszielen und der Architektur. Die daraus resultierende Frage ist, mit welchen Möglichkeiten diese Lücke geschlossen werden kann. Um diese Frage beantworten zu können, müssen zunächst die Integrationsmuster betrachtet und untersucht werden, um feststellen zu können, inwieweit diese mit der Integrationsebene direkt verknüpft werden können.

Integrationsmuster werden zurzeit lediglich zur Dokumentation von Integrationslösungen eingesetzt und bilden die Grundlage für die letztendliche Implementierung dieser Landschaften. Die ausführbaren Lösungen werden dagegen nach dem aktuellen Stand der Technik durch Entwickler, also einer anderen Personengruppe als den Systemarchitekten, manuell umgesetzt. Dabei stellt sich die Frage, ob eine Dokumentation auf Basis von Integrationsmustern nicht als unmittelbare Spezifikation dienen kann, um daraus eine ausführbare Anwendung zu erzeugen. Die Generierung von etwas Ausführbarem direkt aus der Dokumentation führt sogleich zu einer Erhöhung der Qualität und Produktivität in der Erstellung von Integrationslösungen [VS06] (vergleiche auch Abschnitt 2.6).

Wie bereits beschrieben, ist bei der Modell-getriebenen Softwareentwicklung das Modell im Zentrum des Entwicklungsprozesses. Hat man ein solches formales Modell erzeugt, können Transformationsalgorithmen darauf angewendet werden und ausführbare Systeme erzeugen. Änderungen an Anforderungen oder neue Forderungen können in das Modell eingebaut werden und sind nach einer anschließenden Neugenerierung direkt in das ausführbare System eingeflossen. Dies bedeutet eine Erhöhung der Produktivität und auch eine Erhöhung der Qualität, da bei vorhandenem Modell immer nachvollziehbar die

gleichen ausführbaren Systeme generiert werden. Ein wirklicher Gewinn ergibt sich, wenn ein System gewachsen ist und einige Änderungen hinter sich hat. In einer solchen Situation zahlt es sich aus, dass das System konform zu seiner Architekturbeschreibung ist und es automatisch aktuelle Entwurfsdokumentation gibt.

Die Nachteile eines solchen Vorgehens dürfen nicht außer Acht gelassen werden. Zunächst ist die Erstellung der Transformationsalgorithmen ein sehr aufwändiger Prozess. Diese Algorithmen können auch nur von Experten erstellt werden, also Personen, die genau wissen, wie ein bestimmtes Integrationsmuster letztendlich implementiert werden kann/muss. Außerdem spielt gerade in großen Softwaresystemen (und Integrationslösungen gehören in diese Familie) das Testen und Debuggen einzelner Komponenten oder des gesamten Systems eine wichtige Rolle. Dies muss in der Regel auf lauffähigen Systemen erfolgen. Das Simulieren auf den Modellen ist natürlich auch möglich, allerdings können nur die „echten“ Systeme Aufschluss darüber geben, ob die Komponenten korrekt arbeiten. Tritt jedoch ein Fehler im System auf, ist das Auffinden der Ursache beziehungsweise der verursachenden Komponente häufig schwierig. Bedingt ist dies dadurch, dass die Verbindung von System zu Modell (also die Rückwärtsrichtung) selten gegeben ist. Dieser Nachteil tritt auch in traditionellen Entwicklungsmethoden auf, da auch hier keine Verbindung zwischen Entwurf und Implementierung besteht.

Das Modell muss in einer geeigneten Form vorliegen, damit der Transformationsalgorithmus dieses verstehen und interpretieren kann. Aus diesem Grund müssen die Muster in einer formalen Struktur dargestellt werden, die jedoch bis dato nicht existiert, da die Modelle bisher einzig dokumentierenden Charakter haben. Ihre Eigenschaften und das Verhalten müssen so spezifiziert sein, dass der Algorithmus dies als Eingabe verwenden kann. Aus dieser formalen Eingabe kann anschließend eine ausführbare Anwendung generiert werden. Allerdings sollte diese Spezifikation immer noch von der eingesetzten Technologie (Integrationsinfrastruktur), die zur Ausführung benötigt wird, abstrahieren. Dies ist nötig, damit das Modell für verschiedene Zielplattformen verwendet werden kann. Das bringt den Vorteil mit sich, dass eine bestimmte Technologie zu Beginn der Modellierung nicht festgelegt werden muss. Diese Entscheidung

kann auf eine spätere Phase eines Projektes verschoben werden.

In diesem Kapitel wird zunächst ein so genannter Integrationsprozess beschrieben, der alle benötigten Schritte beinhaltet, um ausgehend von der Problemstellung eine ausführbare Integrationslösung zu erstellen. Dieser Prozess bedient sich dabei der Integrationsmuster und baut auf der Modell-getriebenen Entwicklung auf. Um diesen Ansatz anwenden zu können, werden im folgenden Abschnitt die Integrationsmuster in eine formale Form überführt: die parametrisierbaren Integrationsmuster bilden die Basis für das entsprechende Metamodell. Zum Abschluss des Kapitels werden zwei unterschiedliche Möglichkeiten beschrieben, wie die so geschaffene Grundlage zur Modellierung von Integrationslösungen eingesetzt werden kann: der traditionelle Weg über ein Modellierungswerkzeug und ein alternativer Weg, bei dem der Benutzer in Form eines Prozesses durch die Erstellung einer Integrationslösung geleitet wird.

3.1. Integrationsprozess

Ein vorrangiges Ziel bei der Entwicklung von ausführbaren Integrationslösungen im Rahmen dieser Arbeit ist, dass die verwendete Methode vielfältig eingesetzt werden kann. Integrationsmuster bieten, wie bereits erwähnt, eine hinreichend abstrakte Möglichkeit, Integrationslösungen zu beschreiben. Sie treffen dabei keine Aussage über die Technologie, die letzten Endes die Integrationslösung umsetzen soll. Ein Ziel der hier vorgestellten Methode ist es, diese Entscheidung auf einen möglichst späten Zeitpunkt im Integrationsprozess zu verschieben. Damit wird erreicht, die Integrationslösungen allgemeiner zu verwenden und auf unterschiedlichen Zielplattformen einzusetzen.

Es wurde daher ein Modell-getriebener Ansatz gewählt, in dem die Integrationsmuster als plattformunabhängiges Modell (PIM) verwendet werden, um Integrationslösungen zu erstellen, die auf unterschiedlichen Zielplattformen (PSM) ausgeführt werden können. Die Erstellung der Integrationslösung (also eines PIM) ist daher unabhängig vom später erzeugten PSM. Bisherige Ansätze zielen immer auf genau eine Zielarchitektur ab, stellen die Integrationsmuster nicht als PIM dar und unterstützen daher nicht mehrere Zielplattformen

(siehe Abschnitt 2.7). Eine Migration auf eine andere (neuere) Technologie ist dadurch nach Modellierung der Integrationslösung nicht möglich.

Aus diesem Grund wurde eine neue Methode entwickelt, der ein so genannter Integrationsprozess zugrunde liegt. Dieser Prozess kombiniert verschiedene Schritte, die benötigt werden, um Integrationsprobleme mit Hilfe eines Modell-getriebenen Ansatzes zu lösen. Der Integrationsprozess umfasst die Erstellung des Metamodells für parametrisierbare Integrationsmuster, die Modellierung der Integrationslösung in Form eines PIMs auf Basis des Metamodells samt Werkzeugunterstützung, die automatisierte Generierung des PSM und die letztendliche Ausführung der Integrationslösung. Darüber hinaus zählt der Schritt der eigentlichen Erstellung der Integrationsmuster selbst zum Prozess. Daraus ergeben sich fünf Schritte samt diverser Unterschritte, die durchlaufen werden, um eine Integrationslösung auszuführen. In Abbildung 3.2 ist der Prozess dargestellt. Systemarchitekten, die eine Integrationslösung modellieren, durchlaufen den Prozess für gewöhnlich ab dem dritten Schritt. Die ersten beiden Schritte sind Vorbereitungen, die für den eigentlichen MDD Ansatz benötigt werden.

Diese Arbeit beschäftigt sich in erster Linie mit der Modellierung und der Ausführung von Integrationslösungen. Außerdem wurde ein entsprechendes Metamodell entwickelt, um den MDD basierten Ansatz zu ermöglichen. Im

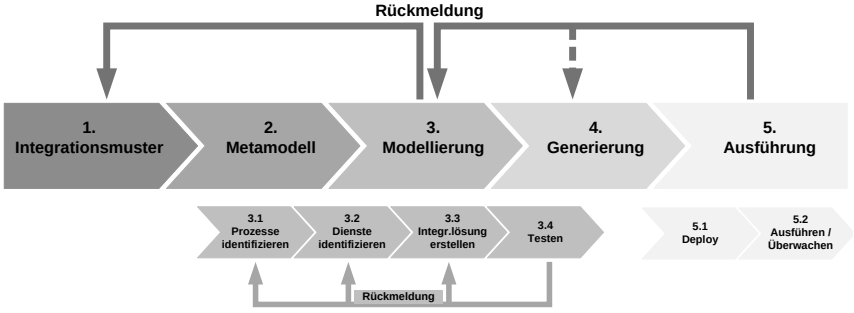


Abbildung 3.2.: Prozess zur Erstellung und Ausführung von Integrationslösungen

Verlauf des Abschnitts wird detailliert erläutert, welche Schritte die Arbeit berührt hat.

Der Integrationsprozess beginnt zunächst bei der Erstellung der Integrationsmuster. In [HW03] wurde diese Vorarbeit geleistet und ein entsprechender Katalog von Mustern erarbeitet, der als Basis für den Prozess dient (siehe auch Abschnitt 2.5). Da die Integrationsmuster momentan nur in Form von Dokumentation vorliegen, müssen diese in eine geeignete Form überführt werden, damit sie innerhalb des Integrationsprozesses als Basis für Generierungsalgorithmen dienen können. Dazu muss zunächst ein entsprechendes Metamodell entwickelt werden. Dieses Metamodell dient Entwicklern als Grundlage, um ein entsprechendes Werkzeug zu entwerfen, mit Hilfe dessen eine Integrationslösung auf Basis von Integrationsmustern erstellt werden kann. Der Schritt der Werkzeugerstellung ist innerhalb des Prozesses nicht explizit aufgeführt. Er befindet sich zwischen den Schritten zwei und drei.

Nachdem das Metamodell samt Werkzeug erstellt wurde, kann die eigentliche Arbeit der Konzeption von Integrationslösungen im dritten Schritt begonnen werden. In diesem Schritt sind noch diverse Unterschritte zu beachten. Bevor eine Integrationslösung entwickelt werden kann, müssen zunächst die Anforderungen identifiziert werden. Für gewöhnlich werden dazu Prozesse, die die Geschäftsziele eines Unternehmens repräsentieren, von entsprechenden Experten definiert. Im Anschluss daran erfolgt die Identifizierung von Diensten der eingesetzten Anwendungen, die dazu dienen können, die vorgegebenen Geschäftsziele zu erfüllen. Diese beiden Unterschritte sind nicht eigentlicher Bestandteil der Integrationslösung; vielmehr sind diese Vorarbeiten notwendig, um eine Integrationslösung überhaupt modellieren zu können. Nachdem also im dritten Unterschritt die eigentliche Integrationslösung erstellt wurde, kann abschließend ein Test der Integrationslösung durchgeführt werden. Dadurch kann die Lösung überprüft werden und unter Umständen entsprechende Fehler oder Unzulänglichkeiten behoben werden (dargestellt durch die Rückmeldungskante). Zu beachten ist, dass dieser Test auf einer entsprechenden Testumgebung (vergleichbar mit der letztendlichen Ausführungsumgebung) ausgeführt werden muss. Dazu muss die Integrationslösung ebenfalls in lauffähige Systeme überführt werden. Die entsprechenden Unter-

schritte der Generierung, Installation und Ausführung sind in Abbildung 3.2 nicht dargestellt. Sobald eine Integrationslösung fertiggestellt und getestet wurde, kann man in den nächsten Schritt übergehen.

Der vierte Schritt (die Generierung) ist das Herzstück des Integrationsprozesses. In diesem Schritt wird eine modellierte Integrationslösung in ausführbare Artefakte überführt. Erst zu diesem Zeitpunkt wird die Zielplattform (die entsprechende Integrationsinfrastruktur) gewählt, auf der die ausführbare Integrationslösung ausgerollt werden soll. Verschiedene Generierungsalgorithmen ermöglichen an dieser Stelle die Erzeugung von unterschiedlichen ausführbaren Artefakten für unterschiedliche Integrationsinfrastrukturen. Die entsprechenden Artefakte werden in eine geeignete Form überführt, damit die Integrationslösung im nächsten Schritt auf der Infrastruktur installiert (*deployed*) werden kann. Wenn die Infrastruktur beispielsweise auf einer SOA basiert, die durch WS-* implementiert wurde, dann werden neben dem BPEL Prozess die entsprechende WSDL Datei und der Deployment Descriptor zur Ausführung von BPEL-Prozessen auf einem entsprechenden WfMS erzeugt (siehe auch Abschnitt 8.7).

Der abschließende Schritt im Integrationsprozess ist die eigentliche Ausführung der Integrationslösung. Zu dieser Ausführung gehört der Unterschritt des Installierens der ausführbaren Artefakte (*Deployment*). Das Deployment kann automatisiert nach der Generierung erfolgen oder aber auch manuell vorgenommen werden. Letzteres ist dann nötig, wenn während des Deployments noch Informationen benötigt werden, die der Transformationsalgorithmus nicht wissen konnte (zum Beispiel wenn der Systemarchitekt keine konkreten Endpunkte der Dienste angibt, müssen diese nun nachgetragen werden). Ist die Integrationslösung auf der Infrastruktur installiert, kann diese ausgeführt werden.

Während der Ausführung wird die Lösung ständig überwacht (Monitoring). Beim Auftreten von Problemen oder wenn erkannt wird, dass die modellierte Lösung nicht den Anforderungen genügt, existiert eine Kante, die Rückmeldungen an unterschiedliche Schritte im Prozess fließen lässt. Typischerweise beeinflussen Fehler und Probleme während der Ausführung die Modellierung sowie eventuell die Generierung (Schritte 3 und 4) und nicht die Erstellung von

neuen Integrationsmustern. Daher zeigt die entsprechende Kante von Schritt 5 auf Schritt 3 und 4. Sollten allerdings bestimmte wiederkehrende Problemstellungen im Modellierungsschritt nicht realisierbar oder nur umständlich zu lösen sein, oder sollten sich existierende Integrationsmuster als fehlerhaft herausstellen (wenn etwa Parameter fehlen), wird auch dies an vorhergehende Schritte (an Schritt 1) gemeldet. In diesem Fall können neue Integrationsmuster erstellt oder existierende überarbeitet werden. Für diese Muster muss wiederum das Metamodell angepasst werden. Durch diese Rückmeldungskanten wird der Integrationsprozess also in sich abgeschlossen.

3.2. Modell-getriebene Entwicklung von Integrationslösungen

Bei der Modell-getriebenen Entwicklung von Softwaresystemen wird, wie in Abschnitt 2.6 beschrieben, ein plattformunabhängiges Modell (PIM) mit Hilfe eines Transformationsalgorithmus in ein plattformabhängiges Modell (PSM) überführt. Der Transformationsalgorithmus verwendet dazu ein Plattformmodell (PM), um ein PSM entsprechend den Vorgaben des PM zu erzeugen. Diesen Modell-getriebene Ansatz kann man, wie im vorherigen Kapitel beschrieben, auch auf die Generierung ausführbarer Integrationslösungen, die auf Integrationsmustern basieren, in Form eines Modell-getriebenen Ansatzes (MDD) anwenden. In diesem Fall sind die Integrationsmuster auf der Ebene des PIM angesiedelt: sie dienen der plattformunabhängigen Beschreibung von Integrationslösungen. Damit allerdings die Muster in einem solchen MDD Ansatz eingesetzt werden können, müssen sie zunächst in eine formalere Form überführt werden, die durch einen Generierungsalgorithmus interpretiert werden kann. Bei Grafiken inklusive textueller Beschreibung, in der Integrationslösungen bisher vorlagen, ist dies nicht möglich. Jedoch reicht die Überführung der Grafiken in ein Modell alleine nicht aus. Jedes Muster muss zusätzlich über bestimmte Eigenschaften näher beschrieben werden. Diese Eigenschaften sind von Muster zu Muster verschieden. Allerdings können Eigenschaften eines einzelnen Musters auch miteinander in Beziehung stehen: wenn eine bestimmte Eigenschaft gewählt wird, hat das zur Folge, dass eine weitere Eigenschaft gesetzt werden muss. Andererseits ist es auch möglich, dass die

zweite Eigenschaft in einem anderen Fall gar nicht zum Tragen kommt. Die nähere Spezifikation der Muster wird daher durch so genannte Parameter als variabler Teil der Muster beschrieben [SL08]. Die Eigenschaften dienen dem Transformationsalgorithmus als Spezifikation, welches Verhalten ein Muster zeigen soll und was dementsprechend daraus generiert werden soll. Denn ein einzelnes Muster führt nicht immer zwangsläufig zum gleichen ausführbaren System. Unterschiedliche Eigenschaften lösen unterschiedliche Aktionen beim Transformationsalgorithmus aus und resultieren somit in unterschiedlichen Systemen (siehe Kapitel 6). Damit ein Algorithmus diese Variabilitätspunkte, die zu unterschiedlichen Systemen führen, interpretieren kann, müssen sie ebenfalls in der formalen Beschreibung der Integrationsmuster enthalten sein.

3.2.1. Parametrisierbare Integrationsmuster

In Abschnitt 2.4 wurde erläutert, wie ein Muster grundsätzlich ein bestimmtes Problem in welchem Kontext lösen kann. Da auf unterschiedliche Gegebenheiten (Kontexte) unterschiedlich reagiert werden muss, müssen für ein und dasselbe Muster unterschiedliche Lösungsansätze geboten werden. Diese unterschiedlichen Ausprägungen lassen sich allerdings mit einem Symbol nicht darstellen. Dies erfolgt in der Regel durch textuelle Beschreibung. Da Integrationsmuster in dem hier dargestellten Verfahren als (formales) Modell dienen sollen, müssen diese unterschiedlichen Eigenschaften ebenfalls berücksichtigt werden. Dies kann mit Hilfe von Variabilitätspunkten beschrieben werden. Da dieser Begriff von vielen unterschiedlichen Bereichen verwendet wird und hier eine Abgrenzung zu diesen Arbeiten hergestellt werden soll, wird in dieser Arbeit ein neuer Begriff eingeführt: die *parametrisierbaren Integrationsmuster* (*Parameterizable Enterprise Integration Patterns, PEP*). Die Parameter solcher Integrationsmuster repräsentieren die Variabilität eines Musters.

Die textuellen Beschreibungen der Muster aus [HW03] müssen in Parameter überführt werden. Dabei stellt sich heraus, dass jedes Integrationsmuster prinzipiell durch vier verschiedene Kategorien an Parametern spezifiziert werden kann [SL08]. Für jedes Muster wird definiert, welche Eingaben das Muster benötigt, welche Ausgaben das Muster liefert, welche sonstigen Einstellungen

der Benutzer vor Gebrauch des Musters durchführen muss und wie das Muster von außerhalb kontrolliert werden kann, damit es seine Funktion erfüllen kann. In jeder Kategorie können ein bis mehrere Parameter das Muster genauer beschreiben. Darüber hinaus werden einzelne Kategorien (und somit Gruppen von Parametern) bei verschiedenen Mustern nicht benötigt (näheres dazu in Kapitel 4). Die Kategorien werden im Einzelnen folgendermaßen benannt:

- *Eingaben (Input):* Beschreibt die Anzahl der Nachrichten, die ein Muster als Eingaben entgegennehmen kann (es müssen aber nicht immer alle möglichen Nachrichten angenommen werden, vergleiche Vollständigkeitsbedingung des Musters *Aggregator* [HW03]).
- *Ausgaben (Output):* Beschreibt die Anzahl der ausgehenden Nachrichten, die ein Muster aussenden kann (auch hier müssen nicht alle möglichen Nachrichten versendet werden, vergleiche *Recipient List* Muster [HW03]).
- *Eigenschaften (Characteristics):* Diese ist die wichtigste der vier Kategorien. Die Parameter dieser Kategorie beschreiben das eigentliche Verhalten eines jeden Musters. Somit ist diese Kategorie maßgebend für die Generierung der ausführbaren Artefakte.
- *Management/Konfiguration (Control):* Dieses Gebiet beschreibt zwei verschiedene Bereiche. Zum einen kann ein Muster von außen gesteuert werden. Das bedeutet, ein Muster kann während der Laufzeit dynamisch konfiguriert werden. Zum anderen kann das Muster Informationen (Laufzeitverhalten) nach außen mitteilen (zum Beispiel für Überwachungszwecke). Die entsprechenden Nachrichten werden somit nicht über den normalen Nachrichtenkanal versendet, sondern über einen separaten Managementkanal (vergleiche Muster *Control Bus* [HW03]). In dieser Kategorie werden somit die Kanäle definiert, über die entsprechende Nachrichten empfangen oder gesendet werden. Außerdem werden die Strukturen dieser Nachrichten definiert.

Abbildung 3.3 zeigt eine Notation, in der ein Muster samt den verschiedenen Parametern dargestellt wird. Diese Notation wird zur Illustration der Parameter verwendet [SL08]. Mittig ist das eigentliche Muster zu finden. Der Text kann durch das entsprechende Symbol des Musters ersetzt werden, die Grafik

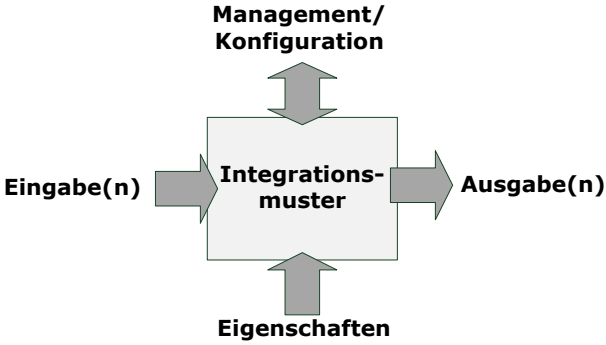


Abbildung 3.3.: Notation zur Darstellung der Parameter eines Integrationsmusters

orientiert sich an den Darstellungen aus [HW03]. Links und rechts des Musters befinden sich die Parameter für Eingaben beziehungsweise Ausgaben. Im unteren Bereich werden die Parameter der Kategorie Eigenschaften dargestellt. Am oberen Rand sind die Parameter für Verwaltung und Konfiguration des Musters zu finden.

Parametrisierte Integrationsmuster haben den entscheidenden Vorteil, dass jedes Muster für sich alleine steht. Das bedeutet, die Parameter beziehen sich nur auf das Muster selbst: Querbeziehungen zu anderen Mustern bestehen nicht. Das impliziert, dass Änderungen an einzelnen Mustern keine Auswirkungen auf andere Muster haben. Auch können dadurch neue Muster leicht integriert werden. Die Erweiterung und Anpassung der Integrationsmuster und der Parameter sind somit leicht umsetzbar.

Neben den atomaren Integrationsmustern existieren auch noch zusammengesetzte Integrationsmuster (vergleiche Abschnitt 2.5). Diese Muster bestehen aus einzelnen atomaren Mustern. Zusammengesetzte Muster besitzen keine eigenen Parameter außer der Anzahl der ein- und ausgehenden Nachrichten. Ihre Parameter bauen sich aus den Parametern der einzelnen atomaren Muster auf. Das bedeutet, dass an einem zusammengesetzten Muster keine Änderung der Parameter vorgenommen werden kann. Dies ist nur an den atomaren Mustern möglich. Daher ist die Modularität und Änderbarkeit auch bei zu-

sammengesetzten Integrationsmustern gegeben. Die einzige Querbeziehung, die einzelne Integrationsmuster haben, ist die Einschränkung, welche Muster miteinander verbunden werden dürfen. Bedingt durch die Pipes-and-Filters Architektur sind die Vorgaben allerdings klar definiert. Ein Filter darf nur mit ein oder mehreren Pipes in Verbindung stehen. Zwei Filter beziehungsweise zwei Pipes dürfen nicht miteinander verbunden werden. Diese Eigenschaft hat allerdings keine Auswirkung auf die Parameter.

Der modulare Aufbau der parametrisierten Integrationsmuster ermöglicht die einfache Anpassung der Muster an neue oder geänderte Anforderungen. So können neue Muster oder neue Parameter zu existierenden Mustern hinzugefügt werden, oder existierende Parameter angepasst oder entfernt werden, ohne dass diese Änderungen Auswirkungen auf die anderen (nicht geänderten) Muster haben. Die Effekte, die geänderte Anforderungen an die Integrationsmuster beziehungsweise das zugrundeliegende Metamodell haben, sind im nachfolgenden Abschnitt 3.2.2 beschrieben. Ein Generierungsalgorithmus, der PEP in ausführbare Artefakte überführt, muss nun ebenfalls nur um diese neuen Elemente erweitert werden. Das heißt, der bereits bestehende Algorithmus muss somit nicht angepasst werden (siehe Abschnitt 6.2).

3.2.2. Metamodell der parametrisierten Integrationsmuster

Die parametrisierten Integrationsmuster basieren auf der Pipes-and-Filters Architektur. Diese Architektur definiert einzig, dass eine Verbindung nur zwischen Komponenten verschiedener Klassen, also zwischen Filtern und Pipes, nicht jedoch zwischen Komponenten der gleichen Klasse – Pipes mit Pipes oder Filter mit Filter – existieren dürfen. Die Integrationsmuster verwenden dieses Meta-Metamodell eines Architekturstils und definieren darauf ein Metamodell (vergleiche Abbildung 3.4). In [HW03] wurde kein explizites Metamodell der Integrationsmuster beschrieben. Dieses ist aber unumgänglich, wenn man mit Integrationsmustern in einer Entwicklungsmethode basierend auf MDD arbeiten möchte. Daher wurde ein Metamodell für Integrationsmuster und PEP entworfen. Das Metamodell liegt in zwei Varianten vor, die je nach Situation verwendet werden (ER-Diagramm und XML Schema (XSD)). Das Schema etwa

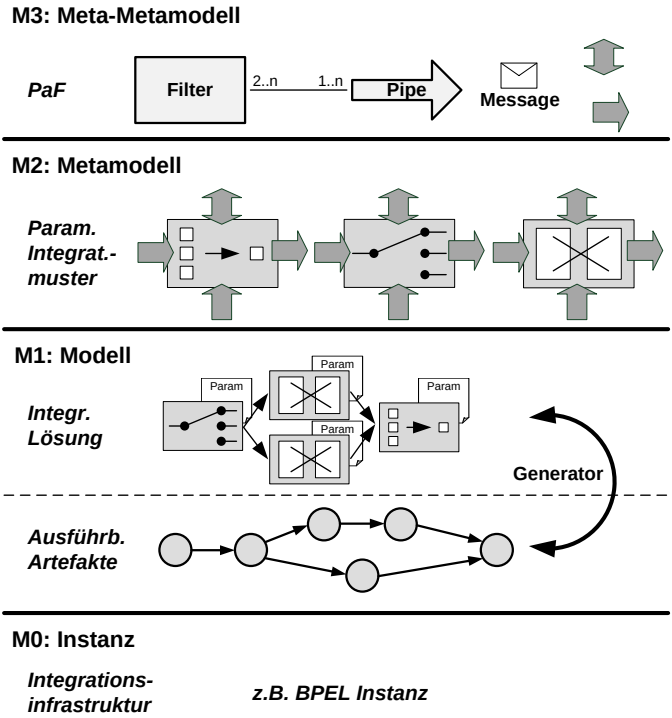


Abbildung 3.4.: Modellebenen der parametrisierbarer Integrationsmuster

ist Grundlage aller Algorithmen und Werkzeuge, die im Verlauf der Arbeit entstanden sind. Eine Instanz des Metamodells (d.h. ein Modell) stellt eine Integrationslösung dar. Auf der gleichen Ebene ist die Umsetzung in ein ausführbares Modell angesiedelt. Die beiden Modelle auf dieser Ebene sind äquivalent. Da allerdings für das Modell der Integrationslösung keine Ausführungsumgebung existiert, muss dieses Modell auf ein anderes Modell übersetzt werden, welches eine Laufzeitunterstützung bietet. Diese Übersetzung wird durch einen Generator ausgeführt. Die eigentlichen Instanzen – also ausgeführte Systeme – werden in einer weiteren Ebene darunter eingeordnet.

Das Metamodell für PEP ist sehr einfach gehalten, da durch die Vorgabe, dass Integrationsmuster auf der PaF Architektur basieren, die Eigenschaften

bereits klar spezifiziert sind. Aus diesem Grund gibt es zwischen verschiedenen Integrationsmustern keine Querbeziehungen. Jedes Muster steht somit für sich alleine und hängt von keinem anderen Muster ab. In Abbildung 3.5 ist ein Ausschnitt aus dem Metamodell dargestellt. In dieser Abbildung sind die einzelnen Parameter der Integrationsmuster der besseren Lesbarkeit wegen nicht abgebildet. Sie sind jedoch auch Teil des Metamodells und werden im weiteren Verlauf der Arbeit näher beschrieben (siehe Kapitel 4). Das Metamodell unterscheidet drei unterschiedliche Entitätstypen. Eine *PaF Struktur Entität* repräsentiert die Elemente der PaF Architektur. Sie sind abstrakt und können daher nicht instanziiert werden. Weitere abstrakte Entitäten sind alle *PEP abstrakte Entitäten*. Beide Entitätstypen dienen der Strukturierung des Metamodells und fassen Eigenschaften der erbenden Entitäten zusammen. Die Entitäten, die in einer Integrationslösung verwendet werden, sind die *PEP instanziierbaren Entitäten*. Sie repräsentieren die Integrationsmuster aus [HW03].

Das Metamodell spezifiziert zunächst, dass eine Integrationslösung ein Messaging System ist: daraus folgt, dass das Wurzelelement einer Integrationslösung ein Messaging System Element ist. Dieses Element hat einen eindeutigen Namen und als Attribut ein *Namespace*, über welches es identifizierbar ist. Innerhalb eines Messaging System Elements treten eine unbestimmte Anzahl an PEP auf mindestens jedoch drei, denn ein PaF System besteht immer aus mindestens zwei Filtern und einer Pipe (im Metamodell Message Channel genannt). Außerdem muss mindestens eine Nachricht das System durchlaufen. Ein Integrationsmuster kann dabei zur Kategorie Message Channel oder Filter gehören, d.h. alle Integrationsmuster erben von einer der beiden Kategorien. Eine Ausnahme bilden alle Muster der Kategorie *Message Construction* (in Abbildung 3.5 ist das Muster *Document Message* als Repräsentant dargestellt). Diese Muster erben von dem übergeordneten Element Message, das wiederum vom Element Integration Pattern erbt. Ein Element Message ist hierbei nicht zu verwechseln mit der Instanz einer Nachricht. Vielmehr ist im Metamodell die Struktur einer Nachricht gemeint.

Pipes werden in diesem Metamodell als Message Channel Elemente bezeichnet. Nachrichtenkanäle sind entweder *Point-to-Point Kanäle (P2P)*, bei denen genau zwei Filter durch einen Kanal verbunden werden, oder *Publish-Subscribe*

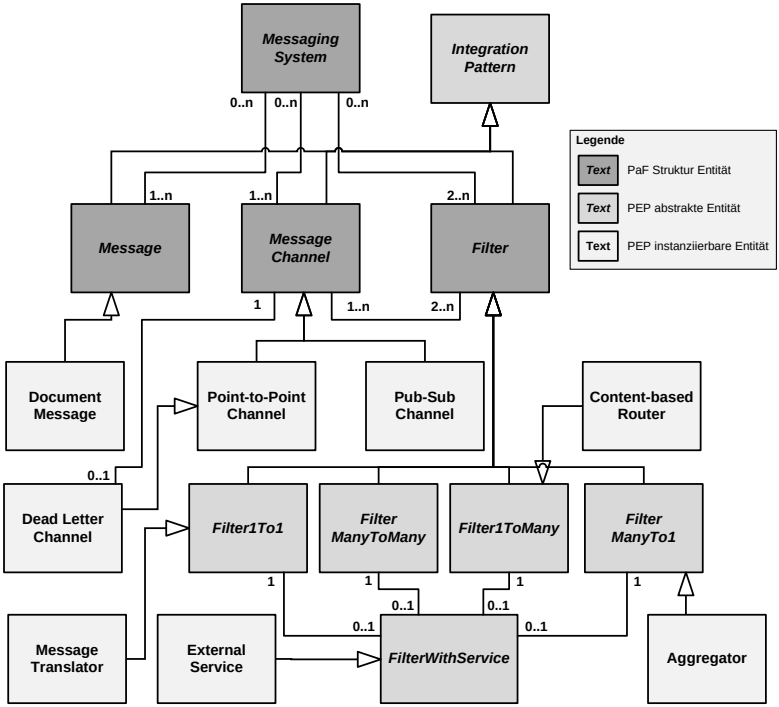


Abbildung 3.5.: Metamodell parametrisierbarer Integrationsmuster

Kanäle, bei denen ein Filter (Sender) an eine unbestimmte Anzahl von Filtern (Empfänger) eine Nachricht versendet. Weitere Muster, die in der Kategorie Messaging Channels auftreten (siehe Abschnitt 2.5), werden als zusätzliche Eigenschaft eines Nachrichtenkanals abgebildet: *Invalid Message Channel*, *Guaranteed Delivery*, *Message Expiration*, *Message Channel* oder *Datatype Channel*. Die zuletzt genannten Muster werden somit nicht als separate Entitäten spezifiziert. So kann zum Beispiel ein P2P Kanal ein Invalid Message Kanal sein, der gleichzeitig über die Eigenschaft der sichergestellten Auslieferung der Nachricht (guaranteed delivery) verfügt. Eine Ausnahme bildet das Muster *Dead Letter Channel (DLC)*. Dies ist eine besondere Form eines P2P Kanals und wird als separate Entität modelliert. Außerdem besitzt ein Kanal, der

nicht auslieferbare Nachrichten an einen DLC weiterleiten muss (vergleiche Kapitel 4 und [HW03]), eine direkte Verbindung zu einem DLC. Dies ist ein Widerspruch zur eigentlichen PaF Architektur. Eine andere Modellierung ist allerdings nicht möglich. Da ein Filter keine Kenntnis darüber hat, ob eine Nachricht an den nachfolgenden Filter durch den Kanal ausgeliefert werden kann (vergleiche Abschnitt 2.2.1), muss der Kanal diese Rolle übernehmen und mit einem separaten DLC Kanal verbunden sein. Die Integrationsmuster, die die Filter der PaF Architektur repräsentieren, erben alle von der Entität Filter. Das Metamodell unterscheidet nun zwischen verschiedenen Arten von Filtern. Das Unterscheidungsmerkmal ist die Anzahl der eingehenden beziehungsweise der ausgehenden Kanäle. Es existieren Filter, die genau einen Eingang und genau einen Ausgang besitzen (*Filter1to1*). Filter, die mehrere Ausgänge haben, werden als *Filter1toMany* Elemente dargestellt. Im Gegensatz dazu ist ein Filter mit mehreren Eingangskanälen und nur einem Ausgangskanal durch das Element *FilterManyTo1* repräsentiert. Kann ein Filter sowohl viele Eingänge als auch viele Ausgänge besitzen, wird er durch ein *FilterManyToMany* Element dargestellt. Alle Filter aus den Kategorien Message Routing, Message Transformation und Messaging Endpoint erben die Eigenschaften von einem dieser vier unterschiedlichen Elemente; in Abbildung 3.5 sind einige dieser Integrationsmuster exemplarisch dargestellt.

Die Entität *FilterWithService* stellt eine weitere Besonderheit des Metamodells dar. Durch diese Entität kann ein Filter um die zusätzliche Eigenschaft des Aufrufens eines externen Dienstes (in diesem Fall eines Web Service) erweitert werden. Diese Entität enthält zusätzliche Parameter, die für den Aufruf eines Web Services nötig sind (zum Beispiel *PortType* [CCMW01][CLS+05]). Das Muster *External Service* (vergleiche Abschnitt 4.4.2) erbt die Eigenschaften der Entität.

Durch den modularen Aufbau des Metamodells sind Änderungen und Ergänzungen infolge neuer Anforderungen oder neuer Muster beziehungsweise neuer Parameter leicht zu integrieren. Da die einzelnen Entitäten keine Querbeziehungen untereinander haben, können neue Elemente gut eingefügt und bestehende leicht angepasst werden. Die Auswirkungen dieser Änderungen auf die Werkzeuge und die Generierungsalgorithmen sind in Abschnitt 6.2 zu

finden.

3.3. Modellierung von Integrationslösungen

Durch die Einführung des Metamodells der parametrisierten Integrationsmuster ist es möglich, Integrationslösungen so zu modellieren, dass dieses Modell direkt als Spezifikation dient, um in einem Muster-getriebenen Ansatz automatisiert in ausführbare Artefakte überführt werden zu können. Dabei kann das Modell (eine Integrationslösung) auf unterschiedliche Arten erstellt werden. In dieser Arbeit werden zwei verschiedene Möglichkeiten der Modellierung von Integrationslösungen vorgestellt. Zum einen wird der Entwurf solcher Lösungen mit Hilfe eines traditionellen Werkzeugs beschrieben – vergleichbar mit einer Anwendung zur Modellierung von UML. Zum anderen wird gezeigt, wie Integrationsösungen Prozess-gesteuert entwickelt werden können. Dies wird ermöglicht, indem die Erstellung einer Integrationslösung durch einen Geschäftsprozess dargestellt wird. Darin enthalten ist auch die Konfiguration der Parameter, die ebenfalls als Prozess repräsentiert ist.

3.3.1. Modellierungswerkzeug

Die Entwicklung einer Integrationslösung kann durch ein Modellierungswerkzeug mit graphischer Benutzungsoberfläche erfolgen. Innerhalb eines solchen Werkzeugs können Integrationslösungen auf einer Zeichenfläche modelliert werden. Dazu werden die unterschiedlichen Integrationsmuster über eine Palette auf diese Zeichenfläche gezogen und miteinander verbunden. Die Integrationsmuster werden durch das Metamodell definiert und sind in einer maschinenlesbaren Form im Werkzeug repräsentiert. Diese PEP können in einem entsprechenden Repository abgelegt werden. In verschiedenen Integrationslösungen können Muster mit identischen Parameterwerten zum Einsatz kommen (zum Beispiel einzelne Muster, miteinander verbundene Muster oder gar ganze Integrationslösungen). Diese häufig wieder verwendeten Muster können mitsamt den Parameterwerten ebenfalls im Repository abgelegt werden. Somit wird das Repository ständig erweitert und die Wiederverwendung in anderen

Integrationslösungen ermöglicht. Das Werkzeug kann neben der eigentlichen Modellierung auch einen Simulationsmechanismus enthalten, mit dem fertige Integrationslösungen simuliert werden können, ohne dass sie auf entsprechenden Systemen installiert werden müssen. Darüber hinaus enthält das Werkzeug verschiedene Generierungsalgorithmen, um modellierte Integrationslösungen in ausführbare Artefakte für unterschiedliche Integrationsinfrastrukturen zu überführen. Zudem kann auch ein Mechanismus enthalten sein, mit dem eine ausführbare Integrationslösung automatisiert auf Produktivsysteme ausgerollt werden kann.

In dieser Dissertation wurde ein solches Werkzeug entwickelt. In Kapitel 8 wird das Werkzeug GENIUS präsentiert, das auf der Eclipse Plattform [Ecla] basiert und eine Teilmenge der beschriebenen Konzepte umsetzt.

3.3.2. Prozess-getriebene Entwicklung

Bei der „traditionellen“ Modellierung einer Integrationslösung durch ein Modellierungswerkzeug entsteht eine Integrationslösung durch das Anfügen und Parametrisieren von Integrationsmustern in einer nicht vorgegebenen Reihenfolge. Eine solche Erstellung kann aber auch gesteuert werden, indem die einzelnen Schritte während der Modellierung (Anfügen und Parametrisieren von Integrationsmustern) jeweils durch einen Subprozessen beschrieben wird. Diese Subprozesse werden in einem übergeordneten Prozess zusammengeführt, der beschreibt welche Schritte nacheinander durchgeführt werden müssen, um eine Integrationslösung zu erstellen. Ein existierendes Modellierungswerkzeug kann zum Beispiel diesen Prozess einsetzen, um den Endanwender durch die Modellierung zu leiten. Oder es kann eine generische Oberfläche der Laufzeitumgebung des Prozesses (zum Beispiel ein WfMS) eingesetzt werden, die direkt auf dem Prozess arbeitet.

Die separaten Subprozesse beschreiben somit in ihrer Gesamtheit exakt das Vorgehen zur Erstellung einer korrekten Integrationslösung. Ein einzelner Prozess definiert dabei zum Beispiel, welche Schritte nacheinander ausgeführt werden müssen, um ein Integrationsmuster so zu parametrisieren, dass es der Lösung hinzugefügt werden kann. Dem übergeordneten Prozess wird

aufgrund der PaF Architektur als Basis der Integrationsmuster eine Regelmäßigkeit vorgegeben: nach einem Filter kommt immer ein Kanal. Aufbauend auf dieser Regelmäßigkeit kann ein Prozess entwickelt werden, der die Modellierung einer PaF Architektur repräsentiert. In Abbildung 3.6 ist ein solcher Prozess in BPMN [Obj09a] Notation dargestellt. Er wird *Guideline Process* [SML08][SML09] genannt und fügt solange Filter und Kanäle der Integrationslösung hinzu, solange es keine verbundenen Ausgänge von Filtern oder Kanälen mit offenen Verbindungen gibt.

Prinzipiell kann man den übergeordneten Prozess auf zwei unterschiedliche Arten definieren:

1. Bei der Erstellung einer Integrationslösung beginnt man ohne Vorgabe und fügt nacheinander Filter und Kanäle in die Integrationslösung ein.
2. Man beginnt mit einer einfachen Integrationslösung (zwei Filter mit einem verbindenden Kanal) und fügt nun zwischen die Filter nach und nach weitere Muster ein, um eine komplette Integrationslösung zu erhalten.

Letztere Variante hat den Vorteil, dass man zu jedem Zeitpunkt eine vollständige Integrationslösung hat. Allerdings stellt bei diesem Vorgehen das Entfernen bereits eingefügter Muster ein Problem dar, da man auch nach dem Entfernen sicherstellen muss, dass die Integrationslösung vollständig und korrekt ist.

Neben der Modellierung (hinzufügen, löschen und ändern) der Integrationsmuster wird auch die Konfiguration der Parameter der einzelnen Muster durch

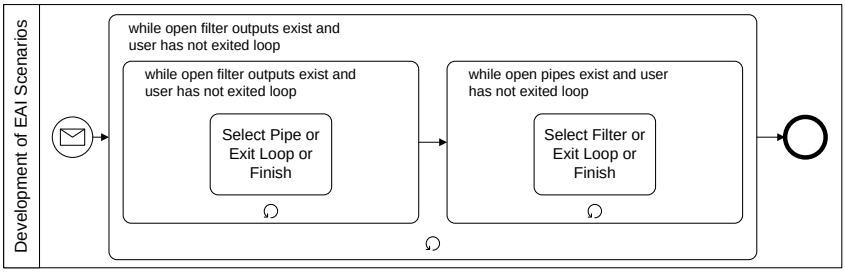


Abbildung 3.6.: Prozess zur Erstellung von Integrationslösungen

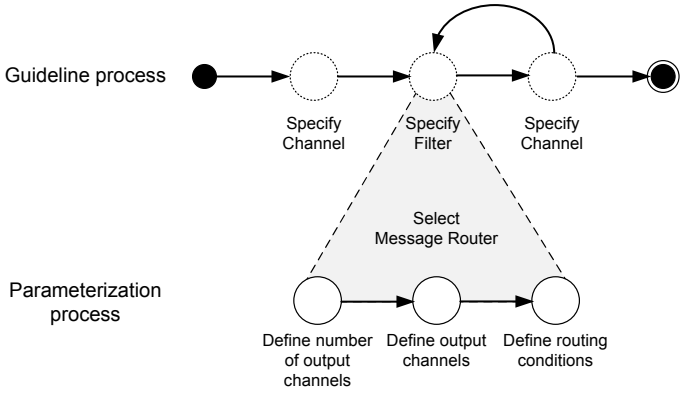


Abbildung 3.7.: Subprozess zur Parametrisierung eines Integrationsmusters

einzelne in sich abgeschlossene Prozesse spezifiziert. Da einzelne Parameter von anderen Parametern abhängen, ist die Reihenfolge des Ausfüllens der Werte von Bedeutung. Diese Abhängigkeiten führen dazu, dass man einen entsprechenden Prozess generieren kann, der das korrekte Ausfüllen garantiert [ML08a]. Für jedes Integrationsmuster steht somit ein eigener Prozess zur Verfügung. Dieser so genannte *Parameterization Process* [SML08] wird nach Auswahl eines speziellen Musters im übergeordneten Prozess aufgerufen. In Abbildung 3.7 ist ein Beispiel dargestellt: im Guideline Prozess wird ein *Message Router* als Integrationsmuster ausgewählt. Daraufhin wird der entsprechende Parametrisierungsprozess mit drei Schritten aufgerufen. Der Benutzer wird also durch die Erstellung einer Integrationslösung mit Hilfe mehrerer Prozesse geleitet. Das Resultat ist eine korrekt konfigurierte Integrationslösung.

3.4. Anwendung der Methode auf das Beispielszenario

In der IT Abteilung von MehrVomGeld wurde darüber nachgedacht, wie man die Modell-getriebene Methode auf die Entwicklung der Integrationslösung Darvien (siehe Abschnitt 1.2) anwenden kann. Da der Abteilungsleiter bereits weitere Integrationsprobleme sieht, die auf Grund des Zusammenschlusses

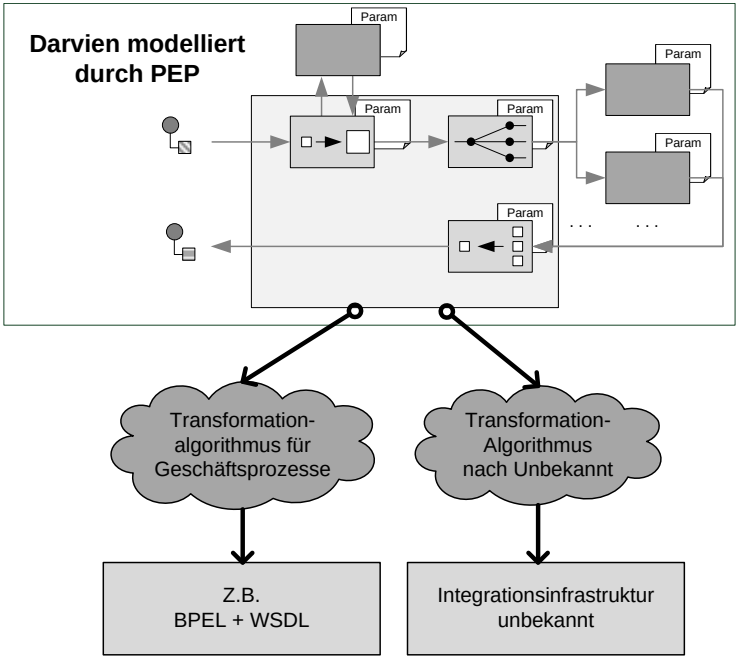


Abbildung 3.8.: Modell-getriebene Entwicklung von Darvien

mehrerer Banksparten von seiner Abteilung gelöst werden sollen, möchte er in seiner Abteilung diese Methode einsetzen. Er erkennt, dass die Entwicklungszeit neuer Integrationslösungen dadurch erheblich verkürzt wird, die Qualität darunter aber nicht leidet. Damit das Management von MehrVomGeld das nötige Budget zur Umsetzung der Methode sowie zur Entwicklung der entsprechenden Werkzeugunterstützung bewilligt, entwickelt er eine entsprechende Vision.

Das Ziel des Abteilungsleiters ist es, die existierende Systemlandschaft des Unternehmens MehrVomGeld auf eine SOA zu migrieren. Er sieht daher Geschäftsprozesse als Integrationsschicht für alle kommenden Anforderungen. Allerdings ist er sich bewusst, dass sich die Technologie in Zukunft schnell ändern kann. Daher möchte er mit der neu eingeführten Methode nicht auf eine

Zielplattform festgelegt sein. Mit der Muster-getriebenen Entwicklung ist dies möglich. In Abbildung 3.8 ist die Vision am Beispiel von Darvien dargestellt. Darvien wird durch parametrisierbare Integrationsmuster modelliert. Dadurch erhält das Modell aus Abbildung 1.1 nun eine präzisere Beschreibung. Entwickler könnten nun schon bedeutend besser die Anforderungen analysieren und interpretieren und dementsprechend ein passendes System erstellen. Der Abteilungsleiter präsentiert dem Management die MDD Methode und erklärt, dass man dadurch auf zukünftige Änderungen der Anforderungen – sei es Anforderungen an die Integrationslösungen oder an die Integrationsinfrastruktur – sehr viel effizienter reagieren kann.

KAPITEL 4

PARAMETRISIERUNG VON INTEGRATIONSMUSTERN

Ein Integrationsmuster wird durch verschiedene Eigenschaften näher spezifiziert und damit das Verhalten innerhalb einer Integrationslösung konkretisiert. Die Eigenschaften eines solchen Musters werden in dieser Arbeit mit dem Begriff Parameter bezeichnet. Wie in Abschnitt 3.2.1 beschrieben, besitzt jedes Integrationsmuster verschiedene Kategorien von Parametern. In den folgenden Abschnitten werden die Parameter verschiedener ausgewählter Integrationsmuster dargestellt. In [SL09a] sind alle existierenden Integrationsmuster aus [HW03] samt ihrer Parameter beschrieben. Zunächst werden einzelne Muster (so genannte atomare Muster) beschrieben. Im Anschluss folgen zusammengesetzte Muster. Der Bereich System Management beschreibt Muster, die dazu dienen, eine Integrationslösung zu verwalten und zu überwachen. Daraufhin werden Muster vorgestellt, die im Rahmen dieser Dissertation eingeführt wurden. Es handelt sich hierbei um Muster, die die Modellierung von Integrationslösungen erleichtern.

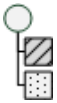
Der Bereich der Fehlerbehandlung wurde in [HW03] bisher außer Acht gelas-

sen. Da die Fehlerbehandlung allerdings essentiell ist, um eine Integrationslösung auch beim Auftreten von unerwarteten Ereignissen korrekt ausführen und beenden zu können, wurden bestehende Muster um entsprechende Eigenschaften erweitert beziehungsweise neue Muster zur Fehlerbehandlung hinzugefügt. Den Abschluss dieses Kapitels bildet die Anwendung der parametriesierbaren Integrationsmuster auf das Beispielszenario Darvien.

4.1. Atomare Muster

Die atomaren Muster werden durch die Parameter als eigenständige Einheit beschrieben ohne Verbindungen zu anderen Mustern. Parameter dieser Muster haben keinen Einfluss auf Parameter anderer Muster.

4.1.1. Nachrichten Muster

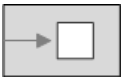


Das Muster einer Nachricht bezieht sich auf die Struktur und auf die Bedeutung der Nachricht (z.B. Dokumentennachricht, Ereignisnachricht). Innerhalb dieser Arbeit ist es nur wichtig, die Struktur der Nachricht zu spezifizieren. Die Bedeutung einer Nachricht wird nicht durch die Parameter dargestellt, sondern durch die Art des Musters. Tabelle 4.1 zeigt den einzigen Parameter, der benötigt wird, um eine Nachricht, egal welchen Typs, zu spezifizieren. Alle folgenden Tabellen sind immer nach dem gleichen Schema aufgebaut. Die Parameter eines Muster werden entsprechend der Kategorisierung aus Abschnitt 3.2.1 eingeteilt.

Tabelle 4.1.: Parameter eines Nachrichtenmusters

Kategorie	**Parameter**
Eingabe:	-
Ausgabe:	-
Eigenschaften:	Struktur der Nachricht
Konfiguration	-

4.1.2. Nachrichtenendpunkte



Die Gruppe der Nachrichtenendpunkte enthält Muster, mit denen beschrieben wird, wie ein (bestehendes) Anwendungssystem (zum Beispiel ein Warenwirtschaftssystem) mit der Integrationsinfrastruktur verbunden wird. Werden diese Muster implementiert, können zum einen Nachrichten aus den Systemen in die Infrastruktur gesendet, zum anderen Nachrichten aus der Infrastruktur an die Anwendungssysteme übermittelt werden. Realisierungen solcher Muster (Adapter) werden prinzipiell von den Anwendungsherstellern zur Verfügung gestellt und müssen somit nur in einer Integrationslösung Nachrichten empfangen können. Aus diesem Grund bestehen diese Muster aus wenigen Parametern, wie Eingangs- und Ausgangsnachrichtenstruktur (siehe Tabelle 4.2). Einzig das Muster *Transactional Client* besitzt einen weiteren Parameter: die transaktionalen Eigenschaften müssen spezifiziert werden. In einer Integrationsinfrastruktur, die auf der Web Service Technologie aufbaut, kann man sich diese Spezifikation als ein *Policy Attachment* innerhalb einer *WS-Policy* [CLS+05][W3C07b] Beschreibung vorstellen.

Tabelle 4.2.: Parameter der Muster der Gruppe Nachrichtenendpunkte

Kategorie	**Parameter**
Eingabe:	0..1 Nachricht(en)
Ausgabe:	0..1 Nachricht(en)
Eigenschaften:	• Strukturen der Eingangsnachricht • Strukturen der Ausgangsnachricht • Bei Transactional Client: – Spezifikation der transaktionalen Eigenschaften (zum Beispiel über Policy Attachement)
Konfiguration	-

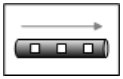
4.1.3. Nachrichtenkanäle

Nachrichtenkanäle repräsentieren die Pipes der Pipes-and-Filters Architektur. Es wird prinzipiell zwischen *Punkt-zu-Punkt* Kanälen und *Publish-Subscribe*

Kanälen unterschieden. Im Standardfall handelt es sich bei den Kanälen um eine Punkt-zu-Punkt Verbindung. Eine solche Verbindung zeichnet sich dadurch aus, dass nur genau ein Sender und ein Empfänger dem Kanal zugeordnet sind.

Darüber hinaus können auch noch spezielle Kanäle modelliert werden, die ausschließlich dazu dienen, Nachrichten, die von Empfängern nicht angenommen oder an Empfänger nicht ausgeliefert werden konnten, entgegenzunehmen und zu transportieren (z.B. *Dead Letter Channe*l). Die explizite Modellierung dieser Eigenschaft ist in bestehenden Systemen so nicht möglich. In einer Message-oriented Middleware (MOM) etwa steht eine *DeadLetterQueue (DLQ)* für jeden Message Queue Manager (MQM) implizit als Dienstgüteeigenschaft zur Verfügung (siehe Abschnitt 2.2.2). Typischerweise werden in einer MOM zahlreiche MQMs eingesetzt. Für einen Administrator ist diese Eigenschaft daher schwer zu gebrauchen, da zunächst die zahlreichen DLQs herausgefunden werden müssen, bevor die darin enthaltenen Nachrichten analysiert werden können. Durch die explizite Modellierung dieser Eigenschaften verbessern die Integrationsmuster diese Situation.

4.1.3.1. Datatype Channel



Das Muster *Datatype Channel* repräsentiert einen Kanal, der nur Nachrichten eines bestimmten Typs transportiert. Der Empfänger am Ende eines Kanals weiß somit ganz genau, welchen Nachrichtentyp er zu erwarten hat und verarbeiten muss.

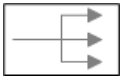
Die Parameter des Musters Datatype Channel sind in Tabelle 4.3 dargestellt. Ein solcher Kanal besitzt eine bestimmte Größe (Puffer), um Nachrichten aufzunehmen und zwischenzuspeichern, bevor die Nachrichten ausgeliefert werden können. Darüber hinaus kann man bestimmen, wie groß eine einzelne Nachricht sein darf, die ein Kanal aufnehmen kann. Die Option des *Dead Letter* Kanals beschreibt die Eigenschaft, dass Nachrichten, die zum Beispiel innerhalb eines bestimmten Zeitfensters nicht ausgeliefert werden können, an einen speziellen Kanal weitergeleitet werden. Dieser Dead Letter Kanal ist dem eigentlichen Kanal bekannt, muss aber über das entsprechende Muster separat modelliert werden. Neben diesen Eigenschaften besitzt das Muster Datatype

Tabelle 4.3.: Parameter des Musters Datatype Channel

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht (die gleiche wie die Eingabenachricht)
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachricht • Größe des Puffers • Maximale Größe einer einzelnen Nachricht • Dead Letter Kanal: – Ja/nein (ist der Kanal mit einem Dead Letter Kanal verbunden) – Bedingung für Weiterleitung einer Nachricht an den Dead Letter Kanal (zum Beispiel Verfalldatum, Sendeversuche) • Zusätzliche Dienstgüteeigenschaften – Zum Beispiel Verfügbarkeit, Sicherheit, Transaktionale Eigenschaften, sicheres Versenden (guaranteed delivery)
Konfiguration	-

Channel eine bestimmte Menge von Dienstgüteeigenschaften. In Tabelle 4.3 ist eine Auswahl dieser Eigenschaften dargestellt.

4.1.3.2. Publish-Subscribe Channel



Ein *Publish-Subscribe Channel* Muster stellt die Art der Kommunikation dar, mit der eine eingehende Nachricht über einen Kanal an mehrere Empfänger als Kopie gesendet wird. Man spricht hierbei auch von einem *Broadcast*-Mechanismus [MHC00]. Das Muster kann von außerhalb konfiguriert werden und ist daher mit einem *Control Bus* Muster verbunden (siehe Tabelle 4.4). Über diesen speziellen Kanal empfängt das Muster Kontrollnachrichten, mit denen sich Empfänger für den Erhalt von Nachrichten registrieren oder sich aus der Empfängerliste austragen.

Tabelle 4.4.: Parameter des Musters Publish-Subscribe Channel

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	N Nachrichten (Kopien der eingehenden Nachricht)
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachricht • Größe des Puffers • Maximale Größe einer einzelnen Nachricht • Dead Letter Kanal: – Ja/nein (ist der Kanal mit einem Dead Letter Kanal verbunden) – Bedingung für Weiterleitung einer Nachricht an den Dead Letter Kanal (zum Beispiel Verfalldatum, Sendeversuche) • Zusätzliche Dienstgüteeigenschaften – Zum Beispiel Verfügbarkeit, Sicherheit, Transaktionale Eigenschaften, sicheres Versenden
Konfiguration	• Verbindung zum Control Bus (Eingang) • Nachrichtentyp der Kontrollnachricht zum Einschreiben (subscribe) und Austragen (unsubscribe) – Enthält auch die Information, ob das Einschreiben dauerhaft erfolgt (durable subscription)

4.1.3.3. Invalid Message Channel

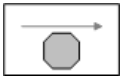
Das Muster *Invalid Message Channel* ist für das Weiterleiten von Nachrichten zuständig, die von einem Empfänger nicht verarbeitet werden konnten.

Der Parameter Ausgang beschreibt, ob eine eingehende Nachricht an einen nachgelagerten Filter (zum Beispiel *Management Console* [HW03]) gesendet werden oder ob die Nachricht intern abgespeichert werden soll. Im letzteren Fall ist der Kanal selbst dafür zuständig, die Nachricht persistent abzuspeichern.

Tabelle 4.5.: Parameter des Musters Invalid Message Channel

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	0..1 Nachricht (wenn Ausgabe auf „ja“ gesetzt ist, andernfalls keine Nachricht)
Eigenschaften:	• Struktur der Eingangsnachricht – Struktur der Ausgangsnachricht (wenn Ausgabe auf "ja"gesetzt ist) ist gleich der Struktur der Eingangsnachricht • Größe des Puffers • Maximale Größe einer einzelnen Nachricht • Zusätzliche Dienstgüteeigenschaften – Verfügbarkeit, Sicherheit, Transaktionale Eigenschaften, etc. • Ausgabe: ja oder nein – Wenn nein: Speichere Nachricht persistent: ja oder nein
Konfiguration	-

4.1.3.4. Dead Letter Channel

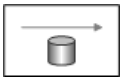


Das Muster *Dead Letter Channel* ist für das Annehmen und Weiterleiten von Nachrichten zuständig, die nicht an einen Empfänger gesendet wurden (abgelaufene Nachrichten). Der Unterschied zum Muster Invalid Message Channel liegt darin, dass die Nachrichten bei letzterem Muster bereits vom Empfänger angenommen wurden. Das Kriterium, wann eine Nachricht abgelaufen ist, wird nicht durch dieses Muster selbst bestimmt, sondern durch das übergeordnete Kanal Muster (siehe zum Beispiel Publish-Subscribe Channel).

Tabelle 4.6.: Parameter des Musters Dead Letter Channel

Kategorie	**Parameter**
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachrichten • Größe des Puffers • Maximale Größe einer einzigen Nachricht • Zusätzliche Dienstgüteeigenschaften – Zum Beispiel Verfügbarkeit, Sicherheit, Transaktionale Eigenschaften, sicheres Versenden • Ausgabe: ja oder nein – Wenn nein: Speichere Nachricht persistent: ja oder nein
Konfiguration	-

4.1.3.5. Guaranteed Delivery

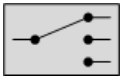


Das Muster *Guaranteed Delivery* ist bei den parametrisierbaren Integrationsmustern kein selbstständiges Muster. Es wird in dieser Arbeit als Zusatz zu bestehenden Kanal Mustern verwendet. Es ist somit eine Erweiterung eines Kanal Musters. Es hat keine weiteren Merkmale (Parameter) und wird nur als Ja/Nein-Option für Kanal Muster verwendet.

4.1.4. Routing Muster

Die Gruppe der Routing Muster beschreibt Integrationsmuster, die Nachrichten auf unterschiedlichen Wegen weiterleiten oder eine Nachricht in mehrere Nachrichten aufteilen beziehungsweise mehrere Nachrichten zu einer Nachricht zusammenfassen.

4.1.4.1. Content-based Router

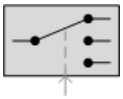


Das Muster *Content-based Router* besitzt einen Eingang und mehrere Ausgänge. Der Router bestimmt anhand des Inhaltes einer eingehenden Nachricht, an welchen Ausgang die Nachricht weitergeleitet werden soll. Es besitzt somit mehrere Ausgänge, leitet aber immer nur genau die eine eingehende Nachricht weiter. Es verändert darüber hinaus die eingehende Nachricht nicht. Neben dem Nachrichtentyp muss noch die Logik zum Weiterleiten der Nachricht angegeben werden. Die Sprache dieser Logik ist frei und wird durch die Parameter nicht vorgegeben. Sie muss durch den Generierungsalgorithmus für die Zielplattform übersetzt werden. Die einzelnen Parameter sind in Tabelle 4.7 dargestellt.

Tabelle 4.7.: Parameter des Musters Content-based Router

Kategorie	**Parameter**
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachricht • Anzahl der Ausgänge (Ausgangskanäle) • Weiterleitungslogik für Ausgänge 1 bis n – „else“ Ausgang (nicht gesetzt oder separater Ausgang)
Konfiguration	-

4.1.4.2. Dynamic Router



Das Muster *Dynamic Router* ist eine Erweiterung des Musters Content-based Router. Die Eigenschaften des Musters Content-based Router werden dadurch ergänzt, dass die Weiterleitungslogik und die Empfänger durch die Empfänger selbst zur Laufzeit dynamisch geändert werden können. Daher besitzt dieses Muster die gleichen Parameter wie das Muster Content-based Router. Zusätzlich kommen die Parameter zur Unterstützung der Dynamik hinzu (siehe Tabelle 4.8).

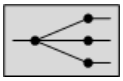
Tabelle 4.8.: Parameter des Musters Dynamic Router

Kategorie	**Parameter**
Eingabe:	Siehe Content-based Router
Ausgabe:	Siehe Content-based Router
Eigenschaften:	Siehe Content-based Router, zusätzlich: • Strategie zur Konfliktauflösung (eine der folgenden Optionen muss gewählt werden) – Ignoriere doppelte Einträge – Verwende Empfänger (beziehungsweise den entsprechenden Kanal), der die Routinglogik zuletzt/zuerst hinzugefügt hat – Zufällige Auswahl eines passenden Kanals
Konfiguration	• Struktur der Kontrollnachricht zur Änderung der Weiterleitungslogik eines Empfängers • Verbindung zum Control Bus

Wenn die Empfänger am Ende der Ausgangskanäle selbst ihre eigene Weiterleitungslogik definieren, können Konflikte innerhalb der Routinglogik entstehen, da die einzelnen Empfänger unabhängig voneinander sind. Es besteht daher die Möglichkeit, dass Empfänger doppelte Einträge (identische Weiterleitungslogik) definieren. Diese Konflikte müssen durch die Implementierung des Muster aufgelöst werden.

Neben den genannten Eigenschaften, spezifiziert das Muster, dass die Weiterleitungslogik persistent abgelegt werden muss, damit diese in einem Fehlerfall (d.h. nach erneutem Hochfahren nach einem Systemabsturz) noch zur Verfügung steht. Diese Eigenschaft muss allerdings nicht in den Parametern aufgenommen werden, da die Umsetzung des Musters dies immer (selbstständig) gewährleisten muss.

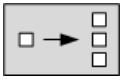
4.1.4.3. Recipient List



Das Muster *Recipient List* beschreibt, dass eine eingehende Nachricht an eine bestimmte Anzahl an Empfänger weitergeleitet wird. Die Nachricht wird somit kopiert und vervielfältigt. Das Muster ist vergleichbar mit einem Publish-Subscribe Kanal, es unterscheidet sich allerdings in der Auswahl der Empfänger. Bei dem Muster Recipient List obliegt die Auswahl der Empfänger dem Muster selbst. Die Empfänger haben somit keinen Einfluss darauf, welche Nachrichten sie empfangen werden und welche nicht. Bei einem Publish-Subscribe Kanal ist diese Entscheidung genau umgedreht. Hier melden sich die Empfänger für bestimmte Nachrichten an. Das Muster (Publish-Subscribe Kanal) hat somit keinen Einfluss, wohin die Nachrichten versendet werden. Bei beiden Mustern ist gleich, dass eine eingehende Nachricht unverändert an eine bestimmte Menge der eingetragenen Empfänger weitergeleitet wird.

Das Muster Recipient List besteht aus zwei unterschiedlichen Teilen: (i) die Bestimmung der Empfänger und (ii) die Verteilung der Nachricht an die verschiedenen Empfänger. Die eingehende Nachricht wird dabei für gewöhnlich nicht verändert. Es kann allerdings sein, dass die Liste der Empfänger in der Nachricht selbst beinhaltet ist. In diesem Fall kann die Liste aus der Nachricht entfernt werden, um sie vor Empfängern zu verbergen. Dazu wird ein *Message Translator* Muster vor die Verteilung der Nachrichten eingefügt. Außerdem muss ein Mapping angegeben werden, das spezifiziert, wie die Empfängerliste, die bereits in der Nachricht enthalten ist oder durch einen externen Dienst erstellt wird, auf die Ausgänge abgebildet wird. Dieser Schritt entfällt, wenn man Regeln angeben kann, die anhand des Nachrichteninhalts die Ausgänge berechnen (vergleiche Content-based Router).

4.1.4.4. Splitter



Das Muster *Splitter* spezifiziert, dass eine einkommende Nachricht in mehrere verschiedene Ausgangsnachrichten aufgeteilt wird. Dabei entstehen neue Nachrichten, die nacheinander an den Ausgang

Tabelle 4.9.: Parameter des Musters Recipient List

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	N Nachrichten
Eigenschaften:	• Struktur der Eingangsnachricht (und Ausgangsnachricht) • Falls die Liste der Empfänger in der Nachricht enthalten ist und entfernt werden soll: – Struktur der Ausgangsnachricht – Zusätzlich alle Parameter eines Message Translator Musters (siehe Tabelle 4.12) • Anzahl der Ausgänge (Ausgangskanäle) • Logik zur Bestimmung der Empfänger (eine der folgenden Optionen muss gewählt werden) – In der Nachricht enthalten (es muss spezifiziert werden, welches Element die Empfängerliste repräsentiert und welches einzelne Element welchen Ausgang repräsentiert) – Von einer externen Quelle (es muss die externe Quelle sowie Anfrage- und Antwortnachricht an diese Quelle spezifiziert werden (siehe Muster External Service (Tabelle 4.21)), außerdem muss spezifiziert werden, welches Element die Empfängerliste repräsentiert und welches einzelne Element welchen Ausgang repräsentiert) – Interne Berechnung (Berechnungslogik muss angegeben werden)
Konfiguration	-

weitergeleitet werden.

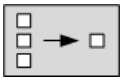
Bei der Aufteilung einer eingehenden Nachricht entstehen viele ausgehende Nachrichten. Diese können sich in ihrer Struktur unterscheiden – dies wird als *statischer Splitter* bezeichnet. Sie können auch in der Struktur gleich sein – in diesem Fall ist es ein *iterativer Splitter*. Im ersten Fall müssen alle Strukturen

Tabelle 4.10.: Parameter des Musters Splitters

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	N Nachrichten
Eigenschaften:	• Struktur der Eingangsnachricht • Die jeweiligen Strukturen der Ausgangsnachrichten (falls sich die Strukturen der einzelnen Nachrichten unterscheidet, ansonsten nur eine Nachrichtenstruktur) • Aufteilungslogik
Konfiguration	-

der ausgehenden Nachrichten angegeben werden. Im letzteren Fall natürlich nur eine Struktur.

4.1.4.5. Aggregator



Ein *Aggregator* Muster ist das Gegenstück zu einem Splitter. Es nimmt verschiedene eingehende Nachrichten an und erzeugt daraus eine einzelne ausgehende Nachricht. Der Aggregator besteht aus drei Hauptbestandteilen: (i) dem Aggregationsalgorithmus, (ii) der Vollständigkeitsbedingung und (iii) der Korrelation. Jede dieser drei Eigenschaften wird mit Hilfe eigener Parameter (siehe Tabelle 4.11) spezifiziert. Da ein Aggregator die angenommenen Nachrichten zwischenspeichern muss, um zum Beispiel die Vollständigkeitsbedingung zu überprüfen, unterscheidet sich dieses Muster von den anderen Mustern. Ein Aggregator muss den Zustand der Bearbeitung von Nachrichten immer persistent abspeichern. Kein anderes Integrationsmuster muss dies leisten. Diese verarbeiten ihre eingehende Nachricht und warten dann auf die nächste Nachricht, ohne dass bereits abgearbeitete und in der Bearbeitung befindliche Nachrichten gespeichert werden müssen.

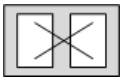
Tabelle 4.11.: Parameter des Musters Aggregator

Kategorie	**Parameter**
Eingabe:	N Nachrichten (auf n Eingangskanälen)
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur jeder Eingangsnachricht – Wenn sich die Struktur der Nachrichten an den verschiedenen Eingängen unterscheidet, muss für jeden Eingang die Struktur angegeben werden • Korrelationselement um verschiedene Nachrichten miteinander in Zusammenhang zu bringen (ein spezifisches Element jeder Nachricht) • Struktur der Ausgangsnachricht • Vollständigkeitsbedingung (eine der folgenden Optionen muss gewählt werden) – *Wait for all, timeout, first best, timeout with override, external event* [HW03] • Aggregationsalgorithmus (eine der folgenden Optionen muss gewählt werden) – Beste Antwort auswählen • Bestimmung, durch welche Eigenschaft die beste Nachricht bestimmt wird (zum Beispiel durch Größenvergleich eines bestimmten Elements) • Wenn Berechnung nötig ist: Transformationsalgorithmus angeben. – Alle eingehenden Nachrichten zu einer Nachricht zusammenführen – Falls Aggregation von einem externen Dienst durchgeführt wird: • Parameter siehe Muster External Service (Tabelle 4.21)
Konfiguration	• Struktur der Kontrollnachricht für das externe Ereignis oder um die Vollständigkeitsbedingung anzupassen • Verbindung zum Control Bus

4.1.5. Nachrichtenbearbeitung

Die Kategorie Nachrichtenbearbeitung beschreibt Muster, die eine eingehende Nachricht auf ein neues Format transformieren oder weitere Informationen der Nachricht hinzufügen beziehungsweise Informationen heraus löschen.

4.1.5.1. Message Translator

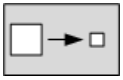


Das Muster *Message Translator* beschreibt, wie eine eingehende Nachricht auf ein anderes Nachrichtenformat transformiert werden kann. Der wesentliche Parameter ist die Logik zur Beschreibung der Transformation (siehe Tabelle 4.12).

Tabelle 4.12.: Parameter des Musters Message Translator

Kategorie	**Parameter**
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangsnachricht • Struktur der Ausgangsnachricht • Transformationslogik
Konfiguration	-

4.1.5.2. Content Filter

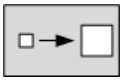


Ein *Content Filter* Muster ist ein Element in einer Integrationslösung, das beschreibt, wie Informationen aus einer eingehenden Nachricht entfernt und somit die Nachrichtengröße reduziert werden kann. Die gelöschten Daten werden nicht im System gespeichert. Die Filterlogik (siehe Tabelle 4.13) spezifiziert dabei die Teile der Nachricht, die entfernt werden sollen. Alle anderen Teile der Nachricht bleiben unberührt.

Tabelle 4.13.: Parameter des Musters Content Filter

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangsnachricht • Struktur der Ausgangsnachricht • Filterlogik (intern oder über externen Dienst möglich) – Bei externem Dienst siehe Parameter des Musters External Service aus Tabelle 21
Konfiguration	-

4.1.5.3. Content Enricher



Das Muster *Content Enricher* beschreibt den Fall, dass eine eingehende Nachricht mit zusätzlichen Informationen angereichert werden soll. Diese Anreicherung kann intern erfolgen, d.h. die zusätzlichen Daten können innerhalb des Musters bestimmt werden (zum Beispiel ein Zeitstempel). Zusätzliche Informationen können aber auch von externen Quellen (zum Beispiel mittels einer Datenbank oder eines externen Dienstes) erfragt werden.

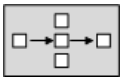
Tabelle 4.14.: Parameter des Musters Content Enricher

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangsnachricht • Struktur der Ausgangsnachricht • Anreicherungslogik (intern oder über externen Dienst möglich) – Bei externem Dienst siehe Parameter des Musters External Service aus Tabelle 21
Konfiguration	-

4.2. Zusammengesetzte Muster

Die zusammengesetzten Muster sind Kombinationen aus einzelnen atomaren Mustern. Sie dienen dazu, eine einzelne eingehende Nachricht aufzuteilen und auf unterschiedlichen, teils parallelen Wegen zu verarbeiten. Das Ergebnis dieser Muster ist häufig eine einzelne resultierende Nachricht.

4.2.1. Composed Message Processor



Das Muster *Composed Message Processor* ermöglicht die separate Abarbeitung von mehreren Elementen einer Nachricht und das Zusammenführen mehrerer Antworten zu einer Ausgangsnachricht. Das Muster ist aus den atomaren Mustern Splitter, Content-based Router, Aggregator und einzelnen Abarbeitungsschritten (*Processors*) aufgebaut (siehe Abbildung 4.1).

Die Parameter dieses Musters stellen sich aus den Parametern der einzelnen Muster (Splitter, Content-based Router, Aggregator und Abarbeitungsschritte) und zusätzlichen Parametern des eigentlichen Composed Message Processor Musters zusammen. Die jeweiligen Abarbeitungsschritte zwischen dem Verteilungsmuster und dem Aggregator können durch jedes beliebige Integrationsmuster mit genau einem Eingang und genau einem Ausgang realisiert werden.

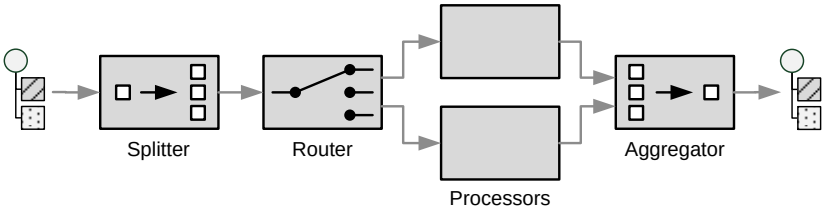


Abbildung 4.1.: Composed Message Processor Überblick

Tabelle 4.15.: Parameter des Musters Composed Message Processor

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangsnachricht • Struktur der Ausgangsnachricht • Anzahl der Abarbeitungsschritte zwischen Router und Aggregator (Processors) • Parameter der individuellen Komponenten: – Splitter – Content-based Router – Aggregator – Processors (entsprechende Muster dürfen nur genau einen Eingang und genau einen Ausgang besitzen)
Konfiguration	-

4.2.2. Scatter-Gather

Das Muster *Scatter-Gather* verteilt Kopien einer eingehenden Nachricht an verschiedene Empfänger und aggregiert die Ergebnisse der unterschiedlichen Pfade zu einer einzelnen Ausgangsnachricht (siehe Abbildung 4.2). Die Verteilung einer eingehenden Nachricht kann dabei über zwei Wege erfolgen:

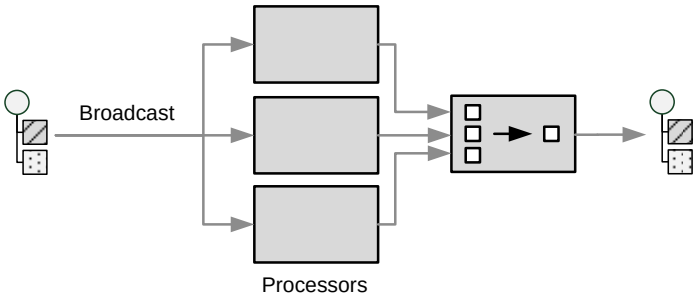


Abbildung 4.2.: Scatter-Gather Überblick

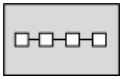
Tabelle 4.16.: Parameter des Musters Scatter-Gather

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangsnachricht • Struktur der Ausgangsnachricht • Verbreitungsmechanismus (Broadcasting) [eine der folgenden Optionen muss gewählt werden]: – Recipient List oder – Publish-Subscribe Channel • Anzahl der Abarbeitungsschritte • Parameter der individuellen Komponenten: – Recipient List oder Publish-Subscribe Channel, – Aggregator – Processors (entsprechende Muster dürfen nur genau einen Eingang und genau einen Ausgang besitzen)
Konfiguration	-

(i) über ein Recipient List Muster oder (ii) über einen Publish-Subscribe Kanal. Zwischen dem Verteilungsmuster und dem Aggregator können verschiedene Abarbeitungsschritte (Processors) folgen.

Die Parameter dieses Musters bestehen aus Parametern, mit denen das eigentliche Muster konfiguriert werden kann, und den Parametern der einzelnen Muster selbst (Recipient List oder Publish-Subscribe Channel, Aggregator, Processors). Die jeweiligen Abarbeitungsschritte zwischen dem Verteilungsmuster und dem Aggregator können durch jedes beliebige Integrationsmuster mit genau einem Eingang und genau einem Ausgang realisiert werden.

4.2.3. Routing Slip



Das Muster *Routing Slip* beschreibt die Eigenschaft, dass einer Nachricht eine geordnete Liste der nachfolgenden Abarbeitungsschritte (Processors) angehängt wird. Dabei folgt jedem Abarbeitungsschritt

Tabelle 4.17.: Parameter des Musters Routing Slip

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangsnachricht • Struktur der Ausgangsnachricht • Struktur der Routing Slip Liste – Spezifikation wie die Liste in die Nachricht integriert wird • Generierung der Routing Slip Liste (eine der folgenden Optionen muss gewählt werden): – Interne Berechnung durch entsprechende Logik – Externe Berechnung (siehe Parameter aus Tabelle 4.21) • Mechanismus zur Markierung abgearbeiteter Filter (eine der folgenden Optionen muss gewählt werden): – Markierung innerhalb der Liste oder – Löschen des entsprechenden Eintrags der Liste • Anzahl der Abarbeitungsschritte (Filter) • Parameter der Abarbeitungsschritte (entsprechende Muster dürfen nur genau einen Eingang und genau einen Ausgang besitzen)
Konfiguration	-

innerhalb des Musters ein Router, der die Nachricht an den nachfolgenden Schritt der Liste weiterleitet.

Die Abarbeitungsschritte innerhalb des Musters können jedes beliebige Muster mit genau einem Eingang und genau einem Ausgang sein. Die Parameter dieser Muster werden in den Parametern des Routing Slip Musters aufgenommen.

Da die einzelnen Abarbeitungsschritte innerhalb des Routing Slip Musters unabhängig von den anderen Filtern arbeiten und daher keine Information besitzen, welche Filter die Nachricht bereits bearbeitet haben, muss das Muster einen Mechanismus umsetzten, mit dem die bereits durchlaufenen Abarbei-

tungsschritte innerhalb der Nachricht (beziehungsweise der Routing Slip Liste) markiert werden. Dies erfolgt durch ein Message Translator, der jedem Abarbeitungsschritt nachgelagert ist. Die beiden Möglichkeiten sind hierbei, dass (i) der zuletzt durchlaufene Abarbeitungsschritt aus der Liste entfernt wird oder (ii) dieser Schritt markiert wird. Außerdem muss sichergestellt sein, dass jedes Content-based Router Muster die eingehende Nachricht an alle möglichen Abarbeitungsschritte des Routing Slip Musters weiterleiten kann.

4.3. System Management

Wenn Integrationslösungen mit Hilfe von Integrationsmustern entwickelt werden, sind die einzelnen Komponenten lose gekoppelt, kennen sich gegenseitig nicht und werden im Allgemeinen auf heterogenen Systemen ausgeführt. Um eine solche Lösung produktiv verwenden zu können, müssen Ausnahmen, Leistungsengpässe (zum Beispiel Ausfall von Systemen) und Änderungen an beteiligten Systemen beachtet werden. Der architektonische Vorteil der losen Kopplung [Kay03] erschwert dabei das Testen, die Fehlersuche (*debugging*) und das Überwachen solcher Anwendungen („Architect's dream, Developer's nightmare") [Hoh05] (siehe auch Kapitel 2). Aus diesem Grund beschreibt der Bereich *System Management* Muster, die das Überwachen, Analysieren, Kontrollieren, Verwalten, Testen, sowie die Fehlersuche von PaF basierten Integrationslösungen ermöglichen. Aus dem umfangreichen Katalog dieses Musterbereichs werden im folgenden Absatz exemplarisch drei Muster beschrieben. Die weiteren Muster sind in [SL09a] näher ausgeführt.

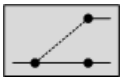
4.3.1. Control Bus

Das Muster *Control Bus* wird für die Verwaltung einer Integrationslösung eingesetzt. Dieser Bus verwendet die gleiche Kommunikationsinfrastruktur wie für die Anwendungsdaten (etwa Messaging), nutzt allerdings separate Kanäle, um den einzelnen Komponenten Verwaltungsdaten zukommen zu lassen. Komponenten können sich mit einem solchen Control Bus verbinden und darüber entsprechende Nachrichten empfangen oder aussenden. Der Control Bus ist

mit einer zentralen Stelle verbunden (*Management Console* [HW03]), die diese Nachrichten aussendet beziehungsweise empfängt und auswertet.

Ein Control Bus kann verschiedene Arten von Nachrichten transportieren: Konfigurationsnachrichten (siehe etwa Dynamic Router), Heartbeat (zur Ermittlung, der Erreichbarkeit), Testnachrichten, Fehlermeldungen oder Statistiken (zum Beispiel Anzahl der verarbeiteten Nachrichten). Der Control Bus wird in erster Linie von den eigentlichen Filtern konfiguriert. Sie spezifizieren, welche Art der Nachricht über den Bus ausgesendet wird oder welche Nachrichten sie erwarten. Daher werden diese Parameter nicht beim Control Bus geführt. Einzig die zentrale Management Konsole muss bestimmt werden.

4.3.2. Detour



Das Muster *Detour* beschreibt einen Router, der anhand seiner aktuellen Konfiguration Nachrichten weiterleitet. Das Muster wird über einen Control Bus verwaltet. Über den Control Bus wird das Muster Detour so konfiguriert, dass eingehende Nachrichten auf eine alternative Route geleitet werden. Die gebräuchlichste Anwendung ist das Umleiten von Nachrichten auf Zwischenschritte, um Validierungen oder Tests zu ermöglichen. Ebenso kann hiermit ein Engpass umgangen werden, der infolge eines Systemausfalls entstanden ist.

4.3.3. Wire Tap

Das Muster *Wire Tap* ermöglicht das Abgreifen von Nachrichten, die über einen Punkt-zu-Punkt Kanal gesendet werden. Das Muster entspricht einem starren Recipient List Muster mit zwei Ausgangskanälen.

Ein Wire Tap Muster besitzt also genau zwei Ausgangskanäle. Ein Ausgang leitet die Nachricht auf den Kanal für den normalen Nachrichtenfluss weiter. Der zweite Ausgangskanal dient der Analyse oder Speicherung von eingegangenen Nachrichten. Typischerweise ist dieser Kanal ein Control Bus. Man könnte aber auch einen normalen Punkt-zu-Punkt Kanal anhängen, der die Nachricht zur Persistierung an einen *Message Store* [HW03] weiterleitet.

Tabelle 4.18.: Parameter des Musters Detour

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachricht • Anzahl der Ausgänge • Art der Anpassung der Umleitung (eine der folgenden Optionen muss gewählt werden) – Passiv: warten auf Konfigurationsnachrichten – Aktiv: Anfrage der aktuellen Konfiguration
Konfiguration	• Struktur der Konfigurationsnachricht • Struktur der Anfrage- und Antwortnachricht (wenn Detour aktiv Konfigurationen anfragt) • Verbindung zum Control Bus

Tabelle 4.19.: Parameter des Musters Wire Tap

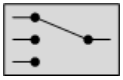
Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Nachrichtenstruktur • Anpassung der Umleitung (eine der folgenden Optionen muss gewählt werden) – Passiv: warten auf Konfigurationsnachrichten – Aktiv: Anfrage der aktuellen Konfiguration
Konfiguration	• Struktur der Konfigurationsnachricht • Struktur der Anfrage- und Antwortnachricht (wenn Wire Tap aktiv Konfigurationen anfrägt) • Verbindung zum Control Bus

Der zweite Kanal kann je nach Einstellung ein- und ausgeschaltet werden. Dies wird über einen Control Bus gesteuert, der entsprechende Konfigurationsnachrichten versendet. Diese Möglichkeit ist allerdings optional. Ohne Konfigurationsmöglichkeit wird immer eine Nachricht an beide Kanäle gesendet.

4.4. Neue Muster

Während der Entwicklung der parametrisierbaren Integrationsmuster sowie der Modellierung von Integrationslösungen mit diesen Mustern und der Generierung von ausführbarem Quelltext traten verschiedene Situationen auf, die teilweise gar nicht oder nur sehr aufwändig gemeistert werden konnten. Um diesen Schwierigkeiten zu begegnen, wurden weitere Integrationsmuster konzipiert.

4.4.1. Join Router



Das Muster *Join Router* ist eine Spezialisierung des Aggregator Musters. Das Muster ist zustandslos. Es nimmt die erste eintreffende Nachricht an und leitet diese an den nachfolgenden Kanal weiter. Dieses Muster ist in Situationen hilfreich, wenn verschiedene parallele Nachrichtenflüsse zusammengeführt werden müssen, von denen immer nur eine Abzweigung pro Durchlauf aktiviert sein kann. In einem solchen Fall muss man die Zusammenführung nicht über den sehr mächtigen Aggregator durchführen, sondern kann dies mit einem leichtgewichtigen Join Router Muster bewerkstelligen.

4.4.2. External Service

Das Muster *External Service* ist eine Erweiterung eines Filter-Musters (zum Beispiel des Musters Content Enricher), um einen Aufruf an einen externen Dienst zu spezifizieren. Dieses Muster wird benötigt, um zum einen die Eingänge und Ausgänge einer Integrationslösung darzustellen (siehe Abschnitt 4.1.2); zum anderen wird das Muster benötigt, um

Tabelle 4.20.: Parameter des Musters Join Router

Kategorie	Parameter
Eingabe:	1 Nachricht (auf n-Kanälen)
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachricht • Anzahl der eingehenden Kanäle
Konfiguration	-

Tabelle 4.21.: Parameter des Musters External Service

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	0..1 Nachricht(en) (bei Antwort auf Anfrage)
Eigenschaften:	• Struktur der Eingangsnachricht (wenn der externe Dienst aufgerufen wird) • Struktur der Ausgangsnachricht (wenn der externe Dienst eine Antwort sendet) • Aufrufmechanismus (eine der folgenden Optionen muss gewählt werden) – Synchroner Aufruf – Asynchroner Aufruf – nur Senden einer Nachricht
Konfiguration	-

den externen Aufruf eines Dienstes innerhalb eines bestimmten Musters zu definieren, das von Haus aus keinen externen Aufruf beinhaltet (zum Beispiel Aggregator mit externem Dienst zur Aggregation von Nachrichten).

Ein wichtiger Teil der Parameter ist die Art des Aufrufmechanismus. Mit Hilfe dieses Parameters wird definiert, wie der externe Dienst aufgerufen wird, ob dieser Dienst auf eine Anfrage antwortet oder ob das Muster nur auf eine Nachricht des Dienstes wartet (ohne eine Anfrage zu senden). Abhängig von der Auswahl des Aufrufmechanismus müssen Eingang und Ausgang sowie die

entsprechenden Nachrichtentypen bestimmt werden.

4.4.3. Dependency Channel

Das Muster *Dependency Channel* ist wichtig, um Abhängigkeiten bestimmter Muster zu definieren, die nicht durch Kanäle verbunden sind. Mit diesem Muster ist es möglich, strukturelle Abhängigkeiten zwischen zwei Mustern zu definieren. Eine strukturelle Abhängigkeit ist zum Beispiel die Zusammengehörigkeit eines Eingang-Ausgang-Paars einer Integrationslösung (zweier External Service Muster), das einen synchronen Aufruf einer Integrationslösung repräsentiert.

Tabelle 4.22.: Parameter des Musters Dependency Channel

Kategorie	**Parameter**
Eingabe:	-
Ausgabe:	-
Eigenschaften:	• Art der Abhängigkeit (eine der folgenden Optionen muss gewählt werden): – Asynchroner Aufruf einer Integrationslösung – Synchroner Aufruf einer Integrationslösung
Konfiguration	-

4.5. Fehlerbehandlung

Die im bisherigen Kapitel vorgestellte Sammlung von Integrationsmustern unterstützt die Fehlerbehandlung für Integrationslösungen bereits rudimentär. So existieren Muster wie I*nvalid Message Channel* und *Dead Letter Channel*, um auf bestimmte Fehlersituationen reagieren zu können. Ein Invalid Message Channel wird verwendet, um Nachrichten an eine dafür vorgesehene Stelle weiterzuleiten, deren Formate vom Filter nicht verarbeitet werden können beziehungsweise innerhalb derer benötigte Werte zur Verarbeitung an einem Filter nicht gesetzt sind. Das Muster Dead Letter Channel wird eingesetzt, um

Nachrichten, die an den eigentlichen Empfänger nicht ausgeliefert werden können (zum Beispiel weil dieser nicht verfügbar ist), an einen für diesen Zweck vorgesehenen Speicherort zu senden. Darüber hinaus ist die Gruppe der System Management Muster für gewisse Aspekte der Fehlererkennung, -diagnose und, wenn möglich auch -behebung, gedacht. Hier werden zum Beispiel auch die oben genannten Muster eingesetzt, um „ungültige“ Nachrichten zu analysieren. Allerdings geschieht diese Art der Fehlerbehebung in der Praxis manuell, d.h. Personen müssen das System überwachen und geeignete Maßnahmen einleiten, um das System wieder in einen konsistenten Zustand zu überführen. Die durch[HW03] beschriebenen Fehlerbehandlungsmechanismen der Integrationsmuster heben daher auf technische beziehungsweise infrastrukturelle Probleme ab. Die Anwendungsebene, also konkrete Geschäftsabläufe, werden mit den dort beschriebenen Möglichkeiten nur bedingt unterstützt.

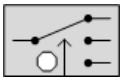
So helfen die bestehenden Integrationsmuster, um Integrationslösungen zwar zu einem definierten Ende zu bringen, der eigentliche Fehler wird dadurch allerdings nicht behoben. Zum Beispiel wird unter Umständen ein Geschäftsfall nicht korrekt zu Ende geführt, und das System befindet sich am Ende eines Durchlaufs in einem definierten (Fehler-) Endzustand: eine nicht auslieferbare Nachricht steht in der so genannten *DeadLetterQueue* [BHL95]. Es müssen weitere (manuelle) Korrekturmaßnahmen ergriffen werden, um den Geschäftsfall abzuschließen. Dies ist unter Umständen nicht erwünscht. Beispielsweise wird eine Bestellung bearbeitet, die Lieferung mit verschiedenen Artikeln zusammengestellt, verpackt und im Zwischenlager deponiert. Nun wird die Bestellung vor der Auslieferung storniert. Die Ware verbleibt allerdings verpackt im Zwischenlager und kann nicht weiter verkauft werden. Der Geschäftsfall ist daher nicht abgeschlossen. Auf Grund dessen sind weitere Mechanismen nötig, um Fehler auf der Anwendungsebene abfangen und korrekt behandeln zu können.

Ein Problem hierbei ist, dass in der Pipes-and-Filters Architektur kein globales Wissen vorhanden ist (siehe auch Kapitel 5). Aufgaben werden lokal abgearbeitet, Verbindungen zu anderen Filtern (vor allem zu Vorgängern) sind einem Filter nicht bekannt. Diese Eigenschaften sind bei der Realisierung fehlertoleranter Integrationslösungen allerdings von Bedeutung. Sollte ein Filter innerhalb der Integrationslösung einen Fehler hervorrufen, zum Beispiel

wenn er nicht erreichbar ist, müssen unter Umständen bereits durchgeführte Änderungen wieder rückgängig gemacht werden (Kompensation). Daher ist es notwendig, auch globales Wissen zu besitzen, über das die Verbindungen zwischen den einzelnen Filtern ermitteln werden können. Die konzeptionelle Architektur wurde daher um globales Wissen erweitert. Dies spiegelt sich durch vier neue Integrationsmuster wider:

- das Muster *Failover Router*, um Ausfallsicherheit für bestimmte Filter zu erreichen,
- das Muster *Compensation Sphere*, mit dem Bereiche einer Integrationslösung markiert werden können, innerhalb derer beim Auftreten eines Fehlers die bereits abgearbeiteten Schritte wieder rückgängig gemacht werden können,
- das Muster *Compensation Filter*, das einen bestehenden Filter um die Eigenschaften erweitert, eine Operation zur Verfügung zu stellen, um bereits abgeschlossene Änderungen rückgängig zu machen und
- das *Catch Fault* Muster, mit welchem bestimmt wird, welche Art von (erwarteten) Fehlern abgefangen und wie diese Fehler behandelt werden sollen.

4.5.1. Failover Router/Failover Processor



Das Muster *Failover Router* dient dazu, nicht aufrufbare Filter innerhalb einer Integrationslösung zu beherrschen, indem explizit alternative Filter modelliert werden, die im Fehlerfall aufgerufen werden können. Er hat somit eine ähnliche Funktion wie das Muster Detour aus dem System Management Bereich. Allerdings wird das Muster Detour von außerhalb über Konfigurationsnachrichten gesteuert, wohingegen der Failover Router eigenständig überprüft, ob Nachrichten an den nachfolgenden Filter versendet werden können und bei Misserfolg Gegenmaßnahmen einleitet (also einen alternativen Filter auswählt). Bei diesem Muster muss man die entsprechenden alternativen Filter explizit modellieren. Die Kanäle zwischen dem Router und den einzelnen Filtern besitzen einen Dead Letter Channel, der wiederum mit

dem Router verbunden ist. Der Router erhält somit die Information, dass eine Nachricht nicht ausgeliefert werden konnte. Daraufhin wird diese Nachricht an einen alternativen Filter gesendet. In Abbildung 4.3 ist die Modellierung von Ausfallsicherheit mit einem Failover Router dargestellt.

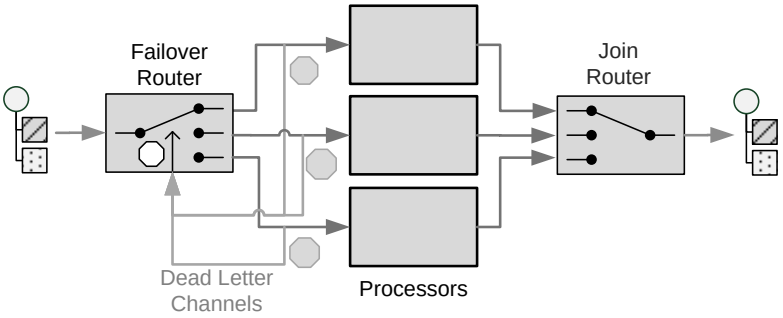


Abbildung 4.3.: Failover Router innerhalb eines Failover Processors

Dieses Muster bildet zusammen mit einem Join Router nach den alternativen Filtern ein zusammengesetztes Muster (*Failover Processor*). Die Parameter des Musters Failover Processor sind in Tabelle 4.23 dargestellt. Die Parameter der einzelnen alternativen Filter werden zu den Parametern des Musters Failover Processor hinzugefügt.

4.5.2. Kompensationssphären

Können fehlertolerante Integrationslösungen nicht mit der Sammlung der Integrationsmuster zur Fehlerbehandlung aus [HW03] modelliert werden, kann mit Hilfe des Musters *Compensation Sphere (CS)* ein globaler Kompensationsmechanismus konzipiert werden (siehe Abbildung 4.4). Dieses Muster lehnt sich an das gleich benannte Konzept bei Workflows an [Ley96]. In einem Workflow übernimmt das Workflow Management System (WfMS) bei Auftritt eines Fehlers die Koordination, um die korrigierenden Aktionen durchzuführen (so genannte Kompensation). Das WfMS ermittelt die bereits abgelaufenen Aktivitäten und berechnet daraus einen Kompensationsgraphen, der in umgekehrter Reihenfolge der ursprünglichen Ausführung die Aktivitäten anordnet. Dieses

Tabelle 4.23.: Parameter des Musters Failover Processor

Kategorie	Parameter
Eingabe:	1 Nachricht
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Eingangs- und Ausgangsnachricht • Anzahl der Ausgänge für alternative Filter • Korrelationsmechnismus zwischen eingehender Nachricht (die nicht ausgeliefert werden konnte) und bereits verwendeten Ausgängen • Algorithmus zur Bestimmung des nächsten Ausgangs (im Falle von nicht auslieferbaren Nachrichten) • Parameter des Musters Dead Letter Channel, mit dem jeder Kanal zwischen Router und Filtern parametrisiert ist • Parameter aller (alternativer) Filter (entsprechende Muster dürfen nur genau einen Eingang und genau einen Ausgang besitzen)
Konfiguration	Eingang des Kanals, über den nicht auslieferbare Nachrichten gesendet werden (Dead Letter Channel)

Vorgehen setzt voraus, dass für jede Aktivität auch eine kompensierende Aktivität zur Verfügung steht, die bereits durchgeführte Änderungen rückgängig macht (diskrete Kompensation). In Abbildung 4.4 sind die vorhandenen diskreten Kompensationen eines Filters durch das schwarze Kästchen mit einem „c“ darin an den jeweiligen Filtern dargestellt. Sollte allerdings die diskrete Kompensation, also das Umkehren des Ablaufgraphen und die Ausführung der einzelnen Kompensationen, nicht ausreichend sein, kann man eine benutzerdefinierte Kompensation einfügen. Diese Kompensation wird der CS angefügt und kann eine einzelne Aktivität bis hin zu mehreren verschachtelten Aktivitäten beinhalten.

Die Anwendung dieses Prinzips auf Integrationsmuster, die auf der PaF Architektur basieren, bedeutet, dass man die lose Kopplung der einzelnen Filter aufweichen und globales Prozesswissen hinzufügen muss. Damit der

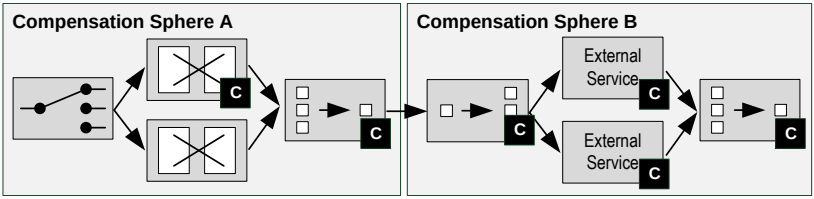


Abbildung 4.4.: Compensation Sphere Beispiel

Kompensationsgraph aufgebaut werden kann, müssen der Nachrichtenfluss durch die Integrationslösung sowie der aktuelle Status der Lösung bestimmbar sein. Wie dies im Einzelnen geschieht, wird durch die Algorithmen beschrieben, die eine Integrationslösung in eine ausführbare Anwendung überführen (siehe Kapitel 6).

Mit Hilfe des mächtigen Werkzeugs der Kompensationssphären ist es möglich, komplexe Verschachtelungen und Überlappungen dieser Sphären zu modellieren. Dies wird durch das Konzept nicht verboten. Allerdings wird im Verlauf der Arbeit (bei der Generierung von Integrationslösungen in Kapitel 6) das Konzept soweit eingeschränkt, dass sich Sphären nicht überlappen, nicht geschachtelt auftreten und nur zusammenhängende Graphen enthalten dürfen.

Die Parameter des Musters Compensation Sphere unterteilen sich in Fehlerbehandlung, Kompensationsbehandlung und die Bestimmung, welche Muster und Kanäle Bestandteil der Kompensationssphäre sind (siehe Tabelle 4.24).

4.5.3. Kompensierende Aktion (Compensation Filter)

Damit eine CS umgesetzt werden kann, muss bei einem Filter die Möglichkeit bestehen, eine kompensierende Aktivität zu definieren. Aus diesem Grund wurde ein weiteres Muster *Compensation Filter* eingeführt, das einen bestehenden Filter um die kompensierenden Eigenschaften erweitert. Nicht alle Filter müssen kompensierende Aktivitäten anbieten. Dies ist nur bei Filtern nötig, bei denen die eingehende Nachricht bearbeitet wird und sich von der ausgehenden Nachricht unterscheidet oder bei denen ein externer Dienst aufgerufen wird, der Daten manipuliert. Die Parameter des Musters bestehen,

Tabelle 4.24.: Parameter des Musters Compensation Sphere

Kategorie	Parameter
Eingabe:	-
Ausgabe:	-
Eigenschaften:	• Fehler innerhalb der CS auffangen – Catch all (alle Fehler werden innerhalb einer Aktivität aufgefangen) Weitere Parameter siehe Muster Catch Fault (Tabelle 4.26) – Catch (für jeden möglichen Fehler eine separate Fehlerbehandlung) Weitere Parameter siehe Muster Catch Fault (Tabelle 4.26) • Kompensationsbehandlung (Compensation Handler): siehe Parameter des Musters Compensation Filter (Tabelle 25) • Filter und Kanäle, die Teil der CS sind
Konfiguration	-

neben den eingehenden und ausgehenden Nachrichten, noch aus der kompensierenden Operation. Sollte allerdings solch eine einfache Kompensation nicht ausreichend sein, kann man die Kompensation erweitern. In diesem Fall modelliert man eine benutzerdefinierte Kompensation, die eine eigenständige Integrationslösung sein kann.

4.5.4. Catch Fault Muster

Das Muster *Catch Fault* wird benötigt, um einen Fehler, der innerhalb einer CS beziehungsweise innerhalb eines einzelnen Filters aufgetreten ist, abzufangen und darauf reagieren zu können.

Der Parameter Rethrow wird benötigt, um festzulegen, ob ein aufgetretener Fehler an den in der Hierarchie nächst höheren Fehlerbehandlungsmechanismus weitergereicht wird. Der Parameter Fehlerbehandlung spezifiziert, wie auf

Tabelle 4.25.: Parameter des Musters Compensation Filter

Kategorie	**Parameter**
Eingabe:	-
Ausgabe:	-
Eigenschaften:	• Bei mindestens einer der folgenden Eigenschaften muss "ja"gewählt werden: • Compensation Handler (ja/ nein) für benutzerdefinierte Kompensation (innerhalb einer CS): – Eine eigenständige Integrationslösung • Compensation Handler (ja/ nein) für benutzerdefinierte Kompensation (innerhalb eines Filters): – Wenn ja, zwei Alternativen (eine muss ausgewählt werden): Kompensation durch Operationsaufruf (Parameter siehe Muster External Service, Tabelle 4.21) Kompensation durch eigenständige Integrationslösung
Konfiguration	-

Tabelle 4.26.: Parameter des Musters Catch Fault

Kategorie	**Parameter**
Eingabe:	-
Ausgabe:	1 Nachricht
Eigenschaften:	• Struktur der Nachricht, die den Fehler repräsentiert • Rethrow (ja/ nein) • Fehlerbehandlung (ja/ nein) – Wenn ja, ist diese Kompensation eine eigene Integrationslösung
Konfiguration	-

einen Fehler reagiert werden soll. Innerhalb der Behandlung kann man eine komplett eigenständige Integrationslösung definieren, die die Fehlerbehandlung repräsentiert.

4.6. Parametrisierte Integrationsmuster für Darvien

Der Abteilungsleiter des IT-Bereichs der MehrVomGeld Bank konnte das Management dazu bewegen, die neue Methode zur Erstellung von Integrationslösungen zu bewilligen. Darvien ist der erste Prototyp, der mit Hilfe dieser Methode entwickelt wird. Ein erfahrener Systemarchitekt entwickelt nun das Modell, um Darvien in ein ausführbares System zu überführen. In Tabelle 4.27 sind die einzelnen Arbeitsschritte aus Darvien aufgeführt sowie die Muster, die die Schritte implementieren, samt der jeweiligen Parameterwerte.

Tabelle 4.27.: Parametrisierte Muster des Szenarios Darvien

Schritt	Muster	Parameterwerte
Bonitäts-prüfung	Content Enricher mit	1 Eingangsnachricht 1 Ausgangsnachricht • Eingangsnachrichtenstruktur: loanRequest (Typ definiert in der Datei darvien.xsd) • Ausgangsnachrichtenstruktur: creditBureauResponse (Typ definiert in der Datei darvien.xsd) • Anreicherungslogik (externer Dienst, s.u.)

Fortsetzung nächste Seite

Tabelle 4.27.: Parametrisierte Muster des Szenarios Darvien

Schritt	Muster	Parameterwerte
	External Service	1 Eingangsnachricht 1 Ausgangsnachricht • Eingangsnachrichtenstruktur: loanRequest • Ausgangsnachrichtenstruktur: creditBureauResponse • Aufrufmechanismus: synchroner Aufruf
Banken auswählen	Recipient List	1 Nachricht N Nachrichten • Nachrichtenstruktur: creditBureauResponse (gleich für Ein- und Ausgang, da Nachricht nicht verändert wird)

Fortsetzung nächste Seite

Tabelle 4.27.: Parametrisierte Muster des Szenarios Darvien

Schritt	Muster	Parameterwerte
		• Anzahl der Ausgänge: 3 • Logik zur Bestimmung der Empfänger : interne Berechnung (in XPath [W3C99]): – Ausgang 1: loanRequest/creditScore >= 500 && loanRequest/loanAmount >= 10000 – Ausgang 2: loanRequest/creditScore >= 700 && loanRequest/loanAmount >= 50000 – Ausgang 3: else
Banken	External Service	1 Eingangsnachricht 1 Ausgangsnachricht • Eingangsnachrichtenstruktur: creditBureauResponse • Ausgangsnachrichtenstruktur: loanQuote (Typ definiert in der Datei darvien.xsd) • Aufrufmechanismus: synchroner Aufruf
Antwort-auswertung	Aggregator	N Nachrichten (auf 3 Eingängen) 1 Nachricht • Eingangsnachrichtenstruktur: loanQuote • Korrelationselement: loanQuote/ requestID

Fortsetzung nächste Seite

Tabelle 4.27.: Parametrisierte Muster des Szenarios Darvien

Schritt	Muster	Parameterwerte
		• Ausgangsnachrichtenstruktur: bestLoanQuote (Typ definiert in der Datei darvien.xsd) • Vollständigkeitsbedingung: Wait for all • Aggregationsalgorithmus: – Beste Antwort auswählen: Minimum(bestLoanQuote/ interestRate)

Die Integrationslösung Darvien steht nun als Modell mit vollständig parametrisierten Integrationsmustern zur Verfügung. Dieses Modell muss nun in ausführbare Artefakte überführt werden. In Kapitel 6 wird dies an einzelnen Mustern umgesetzt. Am Ende des Kapitels werden die ausführbaren Artefakte resultierend aus dem Darvien Modell erläutert.

KAPITEL 5

PIPES-AND-FILTERS IM VERGLEICH ZU WORKFLOWS

Der Integrationsprozess zur Modellierung von Integrationslösungen auf Basis von Integrationsmustern (aus Kapitel 3) zielt darauf ab, verschiedene Zielplattformen zu unterstützen. Einer der Hauptbestandteile dieser Arbeit ist die Abbildung der Integrationsschicht auf einen Geschäftsprozess (siehe Kapitel 6). Allerdings tritt bei der Abbildung auf Geschäftsprozesse (*workflows*) die Frage auf, ob sich die Integrationsmuster, die auf der Pipes-and-Filters Architektur aufbauen, überhaupt auf Geschäftsprozesse beziehungsweise eine SOA, in der die Dienste durch Geschäftsprozesse komponiert werden, abbilden lassen. In diesem Kapitel wird wie in der Literatur üblich der Begriff Prozess verwendet, wenn von einem Workflow die Rede ist. Der Begriff Geschäftsprozess wird nur für einen Prozess der realen Welt verwendet.

Workflow Management Systeme (WfMS) als Ausführungsumgebung der Prozesse unterscheiden sich fundamental von der PaF Architektur. In WfMS tritt der Begriff einer Instanz auf: Prozesse stehen als Modelle zur Verfügung und laufen bei der Ausführung in unterschiedlichen Instanzen ab. Die einzelnen

Instanzen haben dabei keine Information über parallel laufende Instanzen und beeinflussen diese auch nicht. Darüber hinaus besitzen sie jederzeit das Wissen, in welchem Status sich die einzelne Prozessinstanz befindet und wie die Historie der Instanz aussieht. Diese Eigenschaft ist in der PaF Architektur nicht zu finden. Der Status einer Anfrage muss außerdem manuell im System implementiert werden. In den folgenden Abschnitten wird dieser Unterschied genauer herausgearbeitet. Zunächst werden beide Architekturstile beschrieben. Im Anschluss findet ein theoretischer Vergleich statt. Durch ein Szenario soll bestimmt werden, in welchen Fällen die eine Architektur eine bessere Leistungsfähigkeit bietet als die jeweils andere. Verschiedene Messungen geben Aufschluss über die Effektivität der einzelnen Architekturstile.

5.1. Vergleich der Architekturstile

In den folgenden Abschnitten werden die Pipes-and-Filters Architektur und die Architektur Workflow-basierter Systeme untersucht (basierend auf den Ausführungen in Abschnitt 2.2). Es werden zunächst die unterschiedlichen Stile einzeln betrachtet und anschließend miteinander verglichen.

5.1.1. Pipes-and-Filters

Die Pipes-and-Filters Architektur ist sehr einfach gehalten: es gibt Komponenten (*filter*), die Daten bearbeiten, und Verbindungen (*pipes*), die Daten, die von einer Komponente ausgesendet werden, an die nächste Komponente weiterleiten [Meu95][PW92]. Der Begriff Filter ist historisch gewachsen und impliziert nicht, dass Daten nur gefiltert werden (vergleiche *Message Filter* [HW03]). Vielmehr besteht die Aufgabe eines solchen Filters darin, eingehende Daten (Nachrichten) zu bearbeiten und in vielen EAI Szenarien operationale Daten zu manipulieren.

Durch eine Menge von unabhängigen Filtern und deren Verbindungen wird eine Anwendung aufgebaut. Jeder einzelne Filter führt unabhängig von anderen Filtern eine bestimmte Aufgabe durch, die Teil der übergeordneten Anwendung ist. Dies geschieht, indem ein Datenstrom über die Eingabeschnittstelle

eines Filters gelesen wird, auf diesen Daten operiert wird und die bearbeiteten Daten über die Ausgabeschnittstelle weitergegeben werden. Die Architektur erlaubt dabei die Bearbeitung des Eingabestroms sobald die ersten Teile der Daten vorhanden sind. Es muss also nicht auf den kompletten Datensatz gewartet werden. Dies ermöglicht eine performantere Abarbeitung eingehender Daten.

Verbindungen (*pipes*) sind typischerweise als separate Komponenten, wie etwa *Data Repositories* [BD94][HL93] oder *Queues* [BHL95], vorhanden. Deren einzige Aufgabe liegt darin, Schnittstellen anzubieten, um Daten an sie zu übergeben (`put`) oder Daten von ihnen zu holen (`get`). Eine Verbindung ist damit nur für die Übertragung der Daten zwischen den Filtern zuständig; sie führt keine Datenbearbeitung durch.

Die Hauptcharakteristik einer PaF Architektur ist die völlige Isolation jeder einzelnen Komponente; jede Einheit, sei es Verbindung oder Filter, arbeitet unabhängig von jeder anderen Einheit eines Systems. Wenn zum Beispiel die Ausgaben mehrerer Filter zusammengefügt werden müssen, besteht die einzige Möglichkeit innerhalb einer PaF Architektur darin, dieses Zusammenfügen in einer Sequenz von Filtern zu realisieren, in der die Ausgabe eines Filters im nächsten Filter verfeinert wird. Da Filter isoliert sind, kennen sie auch nicht ihren Vorgänger beziehungsweise Nachfolger.

Die PaF Architektur erhält durch die soeben beschriebenen Eigenschaften eine Menge an attraktiven nicht-funktionalen Eigenschaften [Meu95]. Zunächst versteht ein Entwickler das letztendliche Verhalten einer Anwendung als eine einfache Komposition des Verhaltens individueller Filter. Des Weiteren sind Filter bedingt durch die Einfachheit der Schnittstellen gut wiederverwendbar. Jegliche Filter können miteinander verbunden werden. Der nachfolgende Filter geht einzig davon aus, dass er die Daten seines Vorgängers verarbeiten kann [MWSL07]. Außerdem können durch die einfache Struktur eines PaF Systems Durchsatz und Verklemmungen (*deadlock*) relativ leicht analysiert werden [Meu95].

Implementierungen einer PaF Architektur profitieren ebenfalls von der Einfachheit dieses Architekturstils. Sie sind relativ einfach zu verstehen, zu warten und zu erweitern. Neue Filter können leicht hinzugefügt werden, bestehen-

de Filter können durch neue oder verbesserte Filter ersetzt werden. Darüber hinaus ist aber auch das Testen leicht möglich, da jeder Filter separat für sich getestet werden kann (es existieren keine Abhängigkeiten zu anderen Komponenten). Und letztlich können PaF Architekturen gut die Nebenläufigkeit der Bearbeitung von Daten implementieren, indem jeder Filter eine eigenständige Aufgabe oder mehrere zusammenhängende Aufgaben realisiert. In [Isa07] werden einige Beispiele beschrieben, mit denen Skalierbarkeit durch entsprechende Konfigurationen erreicht werden kann.

Allerdings haben PaF Systeme auch Nachteile. Da Filter komplett eigenständige Artefakte darstellen, werden sie oftmals so entworfen, dass sie immer nur einen kompletten Datensatz verarbeiten können und entsprechend auch nur einen kompletten Datensatz ausgeben. Dadurch wird die Eigenschaft der inkrementellen Abarbeitung von Daten nicht genutzt. Dies führt zu einer gebündelten Ausführung (*batched processing*), bei der ein Filter erst mit der Arbeit beginnt, wenn der vorhergehende Filter seine Arbeit abgeschlossen hat. Deshalb wird das Potenzial der parallelen Ausführung von einzelnen Filtern nicht genutzt, was sich negativ auf die Effizienz auswirken kann. Ein weiterer Nachteil entsteht, wenn aufgrund der Implementierung der Filter ein einheitliches Datenformat eingeführt werden muss. Dies resultiert in zusätzlicher Arbeit eines Filters (Transformation von und in das Datenformat), führt zu verringerter Leistung des Systems und erhöht die Komplexität beim Implementieren eines Filters.

Nachteilig ist auch der Ressourcenverbrauch einer Verbindung. Für jede Verbindung, die zwei Filter koppelt, muss eine separate Ressource allokiert werden. In komplexen Szenarien mit zahlreichen Filtern und entsprechend vielen Verbindungen erhöht sich die Anzahl benötigter Ressourcen sehr schnell. Darüber hinaus sind PaF Systeme ungeeignet, um mit interaktiven Szenarien umzugehen [GS93]. Filter sind nicht dazu konzipiert, mit der Umwelt zu interagieren. Sie sind dazu gedacht, kleine, in sich geschlossene Funktionalitäten anzubieten. Außerdem kann ein PaF System nicht ohne Weiteres die Verbindung zwischen zwei separaten, aber zusammengehörenden Datenströmen herstellen. Dadurch, dass PaF Systeme inhärent lose gekoppelt sind, wird die gemeinsame Nutzung von Zustandsinformationen oder gar ein globaler

Zustand des kompletten Systems nicht angeboten. Aus diesem Grund bieten PaF Systeme von Haus aus keine Funktionalität für die Überwachung (*Monitoring*) oder Überprüfung (*Auditing*) eines Systems. Allerdings sind letztere Eigenschaften wesentliche Voraussetzungen beim Betrieb einer produktiven Integrationslösung.

5.1.2. Workflow Management Systeme

Wendet man das Prinzip der Filter auf eine Service-orientierte Architektur an, dann stehen die Filter als einzelne Dienste zur Verfügung. Auch diese Dienste sind lose gekoppelt und kennen nicht den Kontext, in dem sie verwendet werden. Aus diesem Grund müssen auch hier die Dienste miteinander verbunden werden, damit sie in einer Anwendung gemeinsam aufgehen. Diese Integration (Komposition) erfolgt in der Regel mit Hilfe von Prozessen (*workflows*).

Die Struktur eines Geschäftsprozesses (der realen Welt) wird durch ein Prozessmodell [LR00] beschrieben. Dieses Modell definiert alle möglichen Pfade, die in einem Prozess durchlaufen werden können. Darin enthalten sind auch alle Regeln, die spezifizieren, welche Pfade wann (zeitlich und kontextabhängig) abgelaufen werden, sowie alle Aktivitäten, die entlang eines Pfades bearbeitet werden sollen. Das Modell dient als Vorlage, aus der jeder Prozess instanziiert wird: eine Prozessinstanz wird auf Basis eines Prozessmodells erzeugt. Sie wird gemäß einer Menge an Werten ausgeführt, durch die der tatsächliche Pfad durch das Prozessmodell bestimmt wird.

Geschäftsprozesse beschreiben also eine Menge an Aktivitäten und die Reihenfolge, in welcher diese Aktivitäten abgearbeitet werden sollen. Der Standard zum Modellieren und Ausführen von Geschäftsprozessen in einer SOA, realisiert durch WS-*, ist WS-BPEL [Org07a]. In WS-BPEL werden die assoziierten Daten durch eine Menge an Variablen spezifiziert. Die Daten zur Abarbeitung innerhalb eines Prozesses werden aus diesen Variablen gelesen und nach der Bearbeitung dort abgelegt [LR05]. Ein WS-BPEL Prozess beschreibt somit einen Kontrollfluss. Im Gegensatz hierzu steht die PaF Architektur, die Datenfluss-orientiert ist.

5.1.3. Analyse beider Architekturen

Die PaF-Architektur und WfMS ähneln sich in einigen Eigenschaften. Beide Ansätze ermöglichen die technische Unterstützung von Geschäftsprozessen und genügen daher auch den Anforderungen an EAI. Beide Ansätze basieren auf eigenständigen Komponenten, die nicht auf anderen Komponenten aufbauen.

Ein erster Unterschied zeigt sich bei der Modellierung von Prozessen. Bei WfMS steht ein entsprechendes Modellierungswerkzeug zur Verfügung, mit dem ein Prozessmodell unabhängig vom eigentlichen System erstellt werden kann. Bei PaF werden die Kommunikationskanäle zwischen den Filtern angelegt, was in einem fertigen System resultiert und nicht in einem separaten Modell. Hieraus ergibt sich ein weiterer Unterschied.

Im Falle eines WfMS wird zum Start eines Prozesses eine entsprechende Instanz aus dem Prozessmodell heraus erzeugt. In einem PaF System wird die Aufrufnachricht an die erste Komponente des Systems übergeben. Im Anschluss wird diese Nachricht verarbeitet und an die nachfolgenden Komponenten weitergeleitet. Die Nachricht wandert also durch das System. In einer solchen Struktur befinden sich demnach mehrere Datenobjekte an verschiedenen Stellen des Systems. Da es keine globale Sicht auf das System gibt, kann man auch nicht analysieren, in welchem Zustand sich ein bestimmter Geschäftsprozess befindet. Besonders problematisch ist dies, wenn Nachrichten innerhalb eines Systems aufgeteilt werden. In diesem Fall ist es nahezu unmöglich, den Zustand zu bestimmen. Darüber hinaus ist bei PaF von Haus aus kein Mechanismus vorgesehen, um abgearbeitete Nachrichten abzulegen und um zu einem späteren Zeitpunkt Analysen über bearbeitete Geschäftsprozesse fahren zu können.

Bei einem WfMS ist dies anders. Hier ist eine eintreffende Nachricht an eine bestimmte Prozessinstanz gekoppelt. Man kann daher zu jedem Zeitpunkt den Zustand des Prozesses und somit den Fortschritt der Bearbeitung feststellen. Außerdem werden Prozessinstanzen – sowohl laufende als auch abgeschlossene – persistent abgelegt. Dies ermöglicht Überwachung und nachträgliche Überprüfung von Prozessen. Letztere Eigenschaft ist essentiell bei produktiven Systemen und ist daher auch für die Lösung von Integrationsproblemen von großem Vorteil.

Von bedeutendem Unterschied ist auch die Struktur der Systeme. Bei einer PaF Architektur ist die Struktur eines Systems starr: eine Instanz eines Filters ist mit festen Kommunikationskanälen verbunden. Sie empfangen ihre Eingaben von und senden ihre Ausgaben immer an jeweils ein und denselben Kanal. In einer SOA sendet ein Dienst abhängig von der Empfangsadresse, die in einer eingehenden Nachricht definiert ist, die Antwort an den oder die entsprechenden Empfänger. Daher kann ein System, das auf PaF basiert, auch immer nur einen Prozess unterstützen beziehungsweise eine Instanz eines Filters kann immer nur in genau einem System eingesetzt werden. Soll ein weiterer Prozess unterstützt werden, muss ein neues PaF System mit neuen Instanzen der Filter aufgebaut werden. Bei WfMS wird die Struktur hingegen durch das Prozessmodell vorgegeben, d.h. es existiert eine zentrale Stelle, die entscheidet, welche Dienste aufgerufen werden und wem diese aufgerufenen Dienste antworten. Ein WfMS ist daher in der Lage, viele verschiedene Prozessmodelle (gleichzeitig) auszuführen und dabei einen Dienst in unterschiedlichen Prozessmodellen zu verwenden.

Ein weiterer Unterschied besteht darin, dass PaF Systeme Datenfluss-orientiert sind und typischerweise WfMS Kontrollfluss-orientiert. Allerdings besitzt keine der Orientierungen entscheidende Vor- oder Nachteile im Hinblick auf die Anforderungen, die Integrationsprobleme stellen.

Zusammenfassend ist zu sagen, dass der Begriff der Instanz den bedeutendsten Unterschied der beiden Architekturen darstellt. Vor allem im Hinblick auf die Anforderungen, die eine produktive Umgebung stellt (Überwachung, Überprüfung, Dienstgüteeigenschaften, etc.), ist dies besonders wichtig. In nachfolgenden Kapiteln wird zwar beschrieben, wie diese so genannten System Management Komponenten mit PaF nachgebildet werden können, allerdings resultiert das in einem aufwändigen und komplexen System. Ein WfMS bringt diese Eigenschaften bereits von Haus aus mit. In Abschnitt 5.3 werden die verschiedenen Architekturen auch praktisch miteinander verglichen und somit herausgearbeitet, ob die theoretischen Vorteile eines WfMS sich letztlich in der Ausführung als nachteilig erweisen.

5.2. Szenario zur Evaluation der Architekturen

Zunächst wurde ein allgemeines Szenario entwickelt, damit die unterschiedlichen architektonischen Ansätze miteinander verglichen werden können. Das Szenario orientiert sich an dem Beispielszenario Darvien und wurde plattformunabhängig mit Hilfe von Integrationsmustern modelliert. Auf der Basis dieses Szenarios wurde jeweils eine Implementierung für die PaF Architektur und eine für Geschäftsprozesse umgesetzt.

5.2.1. Szenario

Das Szenario lehnt sich an das Darvien Szenario dieser Dissertation an. Es wird bei einem Broker ein Kreditantrag eingereicht. Der Broker ermittelt über einen externen Dienst die Kreditwürdigkeit des Kunden. Er leitet diese Daten zusammen mit der Kreditanfrage an verschiedene passende Banken weiter, sammelt die Antworten und wählt den passenden Kredit für den Kunden aus. Das Szenario wurde um zwei zusätzliche Arbeitsschritte ergänzt, um eine verfeinerte Analyse der Eigenschaften der Implementierungen zu ermöglichen. Beim erweiterten Darvien Szenario kommt nun eine Überprüfung auf staatliche Förderprogramme und eine interne Überprüfung der Tilgungsfähigkeit hinzu. Dadurch kann der Kunde auf eventuelle Risiken hingewiesen werden. Das erweiterte Szenario ist in Abbildung 5.1 dargestellt und kann ausführlich in [Eck08] nachgelesen werden.

Wie in der Abbildung 5.1 zu sehen ist, wird jeder einzelne Schritt des Szenarios durch ein Integrationsmuster repräsentiert. Der Rating Agent wird durch ein *Content Enricher* Muster realisiert. Der nachfolgende Schritt der Förderprogramme wird nur ausgeführt, wenn bestimmte Kriterien erfüllt sind (modelliert durch das Muster *Content-based Router*). Der Schritt selbst ist mit Hilfe des Musters *Complex Message Processor* beschrieben, eingeleitet durch ein *Splitter* Muster und abgeschlossen durch ein *Aggregator* Muster. Die Banken werden mit einem *Scatter-Gather* Muster dargestellt, durch welches eine Anfrage mit einem *Recipient List* Muster an mehrere Banken gesendet wird und die Antworten durch ein *Aggregator* Muster gesammelt und ausgewertet werden. Im

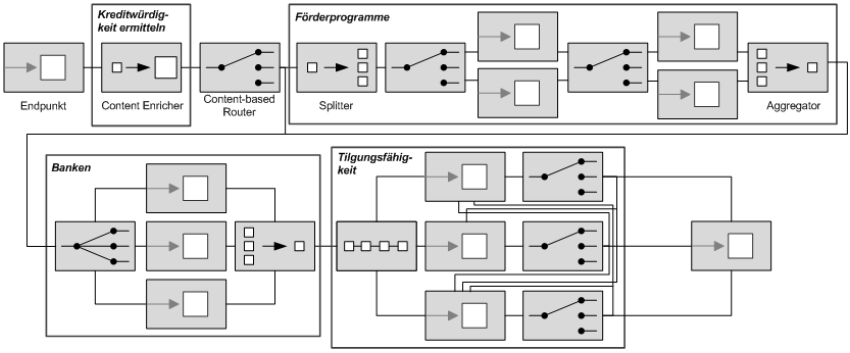


Abbildung 5.1.: Erweitertes Darvien Szenario

letzten Schritt, der Berechnung der Tilgungsfähigkeit, werden verschiedene Informationen nacheinander gesammelt, die Ergebnisse analysiert und eine Information für den Kunden erstellt. Die Reihenfolge der Abfragen variiert von Anfrage zu Anfrage und wird daher mit Hilfe eines *Routing Slip* Musters realisiert.

5.2.2. Implementierungen

Dieses Szenario wurde nun auf verschiedenen Plattformen implementiert, um zum einen PaF zu realisieren und zum anderen einen Geschäftsprozess als Repräsentation der Integrationslösung zu erhalten. Als PaF Implementierung wurden so genannte *Mediation Flows* verwendet, die durch das Produkt *WebSphere Enterprise Service Bus (ESB)* [IBM09a] [IBM09a] der Firma IBM angeboten werden. Als WfMS Implementierung wurde ebenfalls ein Produkt der Firma IBM verwendet: *WebSphere Process Server* [IBM09b] [IBM09b]. Es wurden genau diese beiden Produkte aus zwei Gründen gewählt: (i) die zugrundeliegende Technologie ist die Gleiche (IBM WebSphere), wodurch Unterschiede im Leistungsverhalten durch verschiedene Plattformen eliminiert werden. (ii) das Fähigkeitslevel der Entwicklungsteams der beiden Produkte kann als gleichwertig angesehen werden. Im Übrigen wird der ESB von IBM als Teil des Process Servers ausgeliefert.

5.2.2.1. Pipes-and-Filters (Mediation Flow)

Eine mögliche Implementierung der PaF Architektur sind Mediation Flows. Nachrichten, die zwischen verschiedenen Ausführungsaktivitäten versendet werden, repräsentieren den benötigten Datenfluss. Jeder Ausführungsschritt (*mediation primitive*) bearbeitet eine eingehende Nachricht und leitet sie an den nachfolgenden Schritt weiter. Jeder einzelne Schritt arbeitet unabhängig von den anderen Schritten. Darüber hinaus besitzt ein Mediation Flow kein globales Wissen und implementiert auch kein Fehlerverhalten. In Abbildung 5.2 ist ein Auszug aus dem Loan Broker Mediation Flow dargestellt; zu sehen sind die Aufrufe der Banken.

Jede Aktivität des Szenarios wird im Mediation Flow durch ein oder mehrere *Mediation Primitive* Elemente implementiert. Externe Dienste (wie die Bankaufrufe) werden durch ein Invoke Primitiv realisiert. Routing Entscheidungen, wie etwa bei einem Content-based Router, werden mit Hilfe von *Request Routern* umgesetzt. Transformationen auf ein anderes Nachrichtenformat werden durch einen *Business Object Mapper* implementiert. Das Muster Recipient List wird mit Hilfe eines *Object Mapper* Primitives repräsentiert, welches die Empfänger berechnet, gefolgt von einem *Splitter* Primitiv, das Kopien der Nachrichten entsprechend der Empfängerliste weiterleitet. Die nachfolgenden Routen sen-

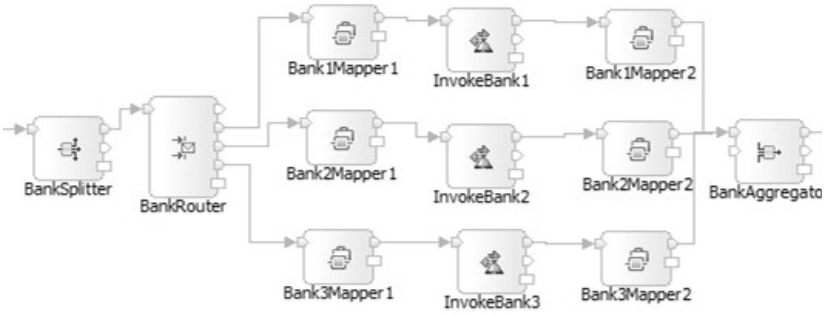


Abbildung 5.2.: Implementierung des Szenarios durch einen Mediation Flow (Ausschnitt)

den jeweils eine Nachricht an die entsprechende Bankaufruf Aktivität (*invoke*). Ein *Join* Primitiv zusammen mit einem *Mapper* Primitiv repräsentieren den Aggregator. Das Muster Routing Slip wird auf ein Mapper Primitiv abgebildet, welches die Reihenfolge der aufzurufenden Aktivitäten berechnet. Im Anschluss werden ein Splitter und ein Router eingefügt, der die Nachricht nacheinander an die entsprechenden Aktivitäten sendet. Nach jedem Aufrufprimitiv folgt ein *Mapper*, der die Ergebnisse des vorhergehenden Primitives in die Nachricht einbaut, den Routing Slip aktualisiert (d.h. abgearbeitete Primitive werden markiert) und diese wieder an den Splitter sendet, der die Nachricht entsprechend weiterleitet.

5.2.2.2. Workflow (WS-BPEL)

Zur Implementierung des Szenarios in Form eines Geschäftsprozesses wurde das Szenario als WS-BPEL Prozess modelliert. Jedes Integrationsmuster wurde dabei durch ein oder mehrere BPEL Aktivitäten implementiert. Der Content-Enricher wurde in Form einer *Invoke* gefolgt von einer *Assign* Aktvität realisiert. Die Auswahl, ob Förderprogramme in die Berechnung mit einbezogen werden sollen, erfolgt über eine *If* Aktivität. Das Muster Recipient List, um verschiedene Banken zu kontaktieren, wird durch eine *Assign* Aktivität umgesetzt, die die zu kontaktierenden Banken bestimmt und von der nachfolgenden *ForEach* Aktivität verwendet wird, die für die parallele Anfrage der Banken zuständig ist. Nach dieser Aktivität folgt eine *If* Aktivität, die die Antwort der Banken überprüft, gefolgt von einer *Assign* Aktivität, die das beste Angebot für den Kunden berechnet. Der abschließende Teil des Szenarios (Routing Slip) wird durch eine *While* Aktivität realisiert. Innerhalb dieses Konstrukts wird immer der nächste Schritt anhand der Informationen innerhalb der Nachricht berechnet und der entsprechende Dienst durch eine korrespondierende *Invoke* Aktivität aufgerufen. Die letzten beiden Schritte umfassen die Erzeugung der Antwort für den Kunden (mittels *Assign* Aktivität) und die Versendung der Antwort an den Kunden (durch eine *Reply* Aktivität). In Abbildung 5.3 ist der Ausschnitt des Musters Recipient List dargestellt.

Abbildung 5.3.: Implementierung des Szenarios in WS-BPEL (Ausschnitt)

5.3. Ergebnisse der Ausführung

Die vorgestellten Implementierungen wurden auf den unterschiedlichen Umgebungen ausgeführt. Dabei wurden verschiedene Varianten getestet, um das Verhalten einer solchen Implementierung unter verschiedenen Rahmenbedingungen analysieren zu können. In diesem Abschnitt wird zunächst die Testumgebung beschrieben, auf der die verschiedenen Testfälle mit unterschiedlicher Ausprägung ausgeführt wurden. Abgeschlossen wird dieser Abschnitt mit der Analyse der Ergebnisse.

5.3.1. Testumgebung

Die Testfälle wurden auf folgender Hard- und Softwareumgebung ausgeführt.

5.3.1.1. Hardware

Es kamen zwei verschiedene Systeme zum Einsatz, eines um die eigentlichen Testszenarien (die PaF und Workflow Implementierungen) auszuführen, und ein weiteres System, um die Web Services ablaufen zu lassen, die aus den Implementierungen heraus aufgerufen werden (zum Beispiel Bank Dienste). Der Testserver besaß einen Intel Core 2 Quad Prozessor mit 2.4 GHz und 4 GB RAM. Der Server zur Ausführung der Web Services lief unter einem Pentium M Prozessor mit 1.6 GHz und 512 MB RAM. Auf beiden Systemen war als Betriebssystem Windows XP Professional mit Service Pack 2 installiert. Die Rechner wurden über ein 100 MBit Ethernet Netzwerk miteinander verbunden.

5.3.1.2. Software

Auf dem Testserver wurde ein IBM WebSphere Process Server in der Version 6.1.2 verwendet, um die Geschäftsprozesse auszuführen. Zur Ausführung der Mediation Flows wurde der darin enthaltene Enterprise Service Bus verwendet. Auf dem Webserver war Apache Tomcat 6.0.14 zur Ausführung der Web Services installiert. Die einzelnen Produkte wurden in der Standardkonfiguration eingesetzt.

5.3.2. Testfälle

Zum Vergleich der verschiedenen Architekturen wurden mehrere Testfälle aufgestellt. Diese Testfälle variieren dabei in zwei entscheidenden Eigenschaften: (i) Skalierbarkeit und (ii) Veränderung der Nachrichtengröße. Die Skalierbarkeit wird getestet, indem die Anzahl der parallel ausgeführten Darvien Szenarien und die Anzahl der aufgerufenen Dienste innerhalb des Szenarios variiert werden. Dieser Test überprüft den Einfluss bei der Erhöhung nacheinander oder gleichzeitig eingehender Anfragen an den Loan Broker und die Auswirkung bei Erhöhung der Dienstanfragen (zum Beispiel Bank-Dienste) innerhalb des Szenarios. Der Einfluss der Nachrichtengröße wird analysiert, indem die Skalierbarkeitstests durchgeführt werden und gleichzeitig die Nachrichtengröße deutlich erhöht wird. Aus diesen Anforderungen resultieren fünf

verschiedene Testfälle, die in Tabelle 5.1 zusammengefasst sind.

Tabelle 5.1.: Testfälle des Szenarios

	Testfälle				
Parameter	**1.**	**2.**	**3.**	**4.**	**5.**
Gleichzeitige Anfragen	1	30	1	30	30
Nachrichtengröße	10kB	10kB	500kB	500kB	500kB
Dienstaufrufe	3	3	3	3	30

Alle Testfälle wurden mit Hilfe von *soapUI* [evi09] ausgeführt. Dieses Werkzeug ermöglicht die Erstellung geeigneter Aufrufnachrichten, überprüft die eingehenden Anworten auf Korrektheit und erstellt entsprechende Statistiken (zum Beispiel über Antwortzeiten). Darüber hinaus wurde der Ereignislog des WebSphere Servers verwendet, um für eine geeignete Auswertung alle benötigten Daten zu erhalten. Sowohl Mediation Flows als auch die Microflows (die BPEL Prozesse auf dem IBM Process Server) werden in einer einzigen Transaktion durchgeführt. Darüber hinaus wurde auch das Verhalten von Macroflows durch die Testfälle gemessen. Diese Art des Prozesses in WebSphere ermöglicht die Persistierung von Nachrichten und des Zustandes während des Ablaufs, also nicht nur am Ende. Dieser Test erfolgte, um zu analysieren, wie sich die kontinuierliche Abspeicherung von Prozessinstanzdaten auf das Laufzeitverhalten eines Prozesses auswirkt. Die Tests können allerdings nicht direkt mit den anderen Testergebnissen verglichen werden, da die Dienstgüteeigenschaften grundverschieden sind. Die Ergebnisse werden daher im folgenden Abschnitt nicht diskutiert. Eine Betrachtung der Ergebnisse findet im letzten Abschnitt des Kapitels statt.

5.3.3. Vergleich der Ergebnisse

Die in diesem Abschnitt präsentierten Zahlen geben die durchschnittlichen Antwortzeiten der Prozesse wieder. Weitere Werte, wie Durchsatz, werden hier nicht erwähnt. Einen ausführlichen Überblick aller Zahlen und Ergebnisse sind in [Eck08] nachzulesen. Alle Testfälle wurden insgesamt zehn Mal ausgeführt.

Die Ergebnisse repräsentieren somit die Durchschnittswerte. Entsprechend der Tradition bei Transaktion- und Datenbank-Benchmarking [GR93] werden die Ergebnisse in Instanzen/Sekunde dargestellt.

In Tabelle 5.2 werden die verschiedenen Testfälle bezüglich der Inter-Prozess Skalierbarkeit miteinander verglichen. Diese Testfälle repräsentieren das Verhalten einer Implementierung, wenn die Anzahl der parallelen Threads, die einen Prozess aufrufen, vergrößert wird, bei gleichbleibender Nachrichtengröße. In den Testfällen 1 und 2 wird die Skalierbarkeit bei gleichzeitig geringer Nachrichtengröße verglichen. Der Vergleich der Testfälle 3 und 4 zeigt die Veränderung bei erhöhter Nachrichtengröße. Die Zahlen legen zwei bedeutende Beobachtungen offen. Zunächst skaliert die Workflow Maschine deutlich besser als die Mediation Flow Maschine (der Service Bus) bei Erhöhung der Last, also der Erhöhung gleichzeitiger, paralleler Anfragen. Außerdem skaliert die Mediation Flow Implementierung bei erhöhten Nachrichtengrößen nicht.

Tabelle 5.2.: Vergleich der Inter-Prozess Skalierbarkeit (in Instanzen/Sekunde)

Testfall	**1.**	**2.**	**3.**	**4.**
Mediation flow	12.7	32.6	4.3	4.5
Microflow	12.6	55.4	6.2	16.9
Macroflow	1.4	2.4	0.9	2.5

Tabelle 5.3 zeigt, wie die Nachrichtengröße die unterschiedlichen Implementierungen beeinflusst. Auch aus diesen Ergebnissen lässt sich die gleiche Folgerung wie aus Tabelle 5.2 schließen: die Mediation Flow Maschine reagiert signifikant empfindlicher auf die Nachrichtengröße als die Workflow Maschine.

Tabelle 5.3.: Vergleich des Einflusses der Nachrichtengröße (in Instanzen/Sekunde)

Testfall	**1.**	**3.**	**2.**	**4.**
Mediation flow	12.7	4.3	32.6	4.5
Microflow	12.6	6.2	55.4	16.9
Macroflow	1.4	0.3	2.4	2.5

In Tabelle 5.4 wird die Intra-Prozess Skalierbarkeit dargestellt. Hierunter ist das Verhalten eines Prozesses zu verstehen, wenn mehrere parallele Dienstaufrufe koordiniert werden müssen (zum Beispiel im Falle des parallelen Aufrufs mehrerer Bankdienste). Es ist offensichtlich, dass durch die zusätzlich benötigte Abarbeitung die Anfragerate, die eine Implementierung bearbeiten kann, geringer wird. Dies ist bei allen Implementierungen der Fall. Allerdings erkennt man auch hier, dass die Mediation Flow Implementierung mit Parallelität deutlich schlechter umgehen kann als die Workflow Maschine.

Tabelle 5.4.: Vergleich der Intra-Prozess Skalierbarkeit (in Instanzen/Sekunde)

Testfall	**4.**	**5.**
Mediation flow	4.5	1.9
Microflow	16.9	11.0
Macroflow	2.5	0.9

5.4. Bewertung

Beide vorgestellten Architekturstile können dazu verwendet werden, Integrationslösungen (auf Basis von Integrationsmustern) auszuführen. Beide Stile haben verschieden Vor- und Nachteile und sind unterschiedlich mächtig, können aber beide das betrachtete Integrationsproblem lösen.

Es wurde gezeigt, dass eine Pipes-and-Filters Architektur mit Hilfe eines Workflow Management Systems prinzipiell umsetzbar ist. Ein WfMS kann alle funktionalen Eigenschaften einer PaF Architektur realisieren. Darüber hinaus bietet die Ausführung einer PaF Architektur durch einen Geschäftsprozess aufgrund des Vorhandenseins von Instanzen weitere Eigenschaften (Monitoring, Auditing, Fehlerbehandlung). Außerdem wurde herausgestellt, dass dieses Verhalten noch dazu mit einer besseren Leistung umgesetzt werden kann. Die Ergebnisse der Testfälle zeigen, dass WfMS in den vorliegenden Szenarien deutlich besser skalieren als PaF Systeme. Dies liegt in der Natur der Sache, da WfMS genau für die Anforderungen der Testfälle ausgelegt sind: viele (gleiche) Prozesse müssen parallel ausgeführt werden und dürfen nicht zu einer

erheblich schlechteren Leistung führen. Allerdings sind dies auch genau die Anforderungen, die Integrationslösungen ebenfalls an Integrationsinfrastrukturen stellen.

Für die hier vorgestellte Umsetzung der Integrationsmuster unter anderem auf eine Workflow Umgebung (siehe Abschnitt 6.2.1) sind dies ausreichend Gründe, um diese Abbildung zu rechtfertigen und weiter zu verfolgen. Eine weitere interessante Betrachtung wäre die Analyse des Verhaltens von Macroflows und der Vergleich mit einer Umsetzung der Eigenschaften des Macroflows auf eine PaF Ausführungsumgebung. Dazu müssten in der PaF Architektur weitere Vorkehrungen getroffen werden, um die erhöhten Dienstgüteeigenschaften von Macroflows (etwa Persistierung von Prozessinstanzdaten) mit Hilfe der PaF Architektur umzusetzen.

KAPITEL 6

GENERIERUNG AUSFÜHRBARER INTEGRATIONSMUSTER

Mit parametrisierbaren Integrationsmustern wurde die Grundlage geschaffen, um Integrationslösungen zu modellieren und aus dieser Beschreibung direkt ausführbare Systeme zu erzeugen. Eine Integrationslösung stellt somit ein plattformunabhängiges Modell (PIM) dar. Ein solches PIM kann mit Hilfe eines Modell-getriebenen Entwicklungsansatzes in ausführbare Integrationslösungen überführt werden. Aus diesem Grund wurde eine Struktur entwickelt, das als Dreh- und Angelpunkt des Generierungsansatzes dient. Das so genannte *EMod (EAI System Model)* beschreibt parametrisierbare Integrationsmuster in einer allgemeinen Darstellungsform sowie die Elemente und Komponenten, die benötigt werden, um Integrationslösungen ausführen zu können. Ein EMod besteht aus verschiedenen Einzelteilen, die im folgenden Abschnitt beschrieben werden. Im Anschluss wird der allgemeine Ansatz erläutert, bei dem mit Hilfe der EMod Komponenten ausführbare Integrationslösungen erzeugt werden können. Daraufhin folgen drei konkrete Umsetzungsbeispiele der Generierung von ausführbaren Integrationsmustern für drei unterschiedliche Integrationsin-

frastrukturen. Jede dieser Generierungen verwendet dabei die gleiche Basis zur Erzeugung der ausführbaren Einheiten. Der vorletzte Abschnitt beschreibt die Verwendung einer Cloud Infrastruktur zur Ausführung und Bereitstellung von Integrationslösungen. Auch diese Variante basiert auf parametrisierbaren Integrationsmustern und verdeutlicht, dass diese Form der Beschreibung der Integrationsmuster ein allgemeiner Ansatz ist, der verschiedene Plattformen unterstützen kann. Das Kapitel schließt mit der Betrachtung der ausführbaren Artefakte, die auf Grundlage des Beispielszenarios Darvien generiert werden.

6.1. EMod - EAI System Model als Basis für ausführbare Integrationslösungen

Ein EAI System Modell beschreibt alle benötigten Elemente, um eine Integrationslösung mit Hilfe parametrisierbarer Integrationsmuster zu modellieren und mit dem unter Abschnitt 6.2 beschriebenen Ansatz in ausführbare Systeme zu überführen. Ein solches Modell kann von unterschiedlichen Anwendungen erstellt und als Spezifikation für ausführbare Integrationslösungen verwendet werden. Allerdings existiert noch kein einheitliches Modell, das alle benötigten Bestandteile einer Integrationslösung beschreibt. Dies ist beispielsweise nötig, wenn Integrationslösungen in verschiedenen Werkzeugen modelliert und diese Spezifikationen zwischen den Werkzeugen ausgetauscht werden (siehe Abbildung 6.1).

EMod ist ein Ansatz, der diese Anforderungen realisiert. Mit Hilfe von EMod ist es möglich, genau zu spezifizieren, welche Artefakte benötigt werden, um (i) eine Integrationslösung zu beschreiben und (ii) um aus dieser Beschreibung ausführbare Systeme zu generieren. Im weiteren Verlauf dieses Kapitels werden in Abschnitt 6.3ff mehrere Möglichkeiten der Nutzung von EMod beschrieben. EMod besteht aus verschiedenen einzelnen Elementen (vergleiche Abbildung 6.2). Die gestrichelten Komponenten auf der rechten Seite repräsentieren konkrete Implementierungen der allgemeinen Konzepte auf der linken Seite. In den nachfolgenden Unterabschnitten werden die einzelnen Bestandteile näher erläutert.

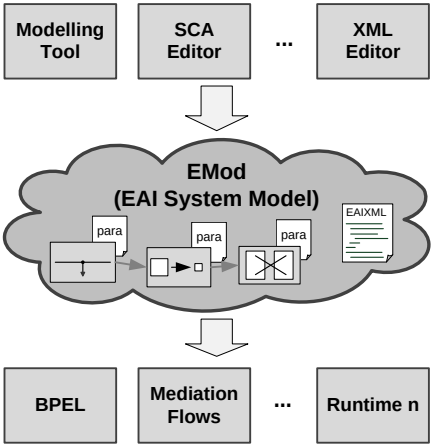


Abbildung 6.1.: Übersicht der Verwendung von EMod

6.1.1. EAI System Modell

Ein EAI System Modell repräsentiert ein Modell einer Integrationslösung eines speziellen Integrationsproblems. Es besteht zunächst aus einer Menge parametrisierbarer Integrationsmuster (*Parameterizable EAI Pattern, PEP*). Modelliert man mit Hilfe dieser Muster eine Integrationslösung, erhält man miteinander verbundene parametrisierte Integrationsmuster. Diese werden in einer geeigneten Form serialisiert, damit die Lösung von einem Generierungsalgorithmus in dem entwickelten Werkzeug übersetzt werden kann. In Abbildung 6.2 ist EAIXML als eine mögliche Realisierung dargestellt.

6.1.2. Parametrisierbare Integrationsmuster

Integrationsmuster sind als Ratschläge (*nuggets of advice*) gedacht, um Integrationslösungen auf einer konzeptionellen Ebene zu realisieren. Mit den parametrisierbaren Integrationsmustern aus Kapitel 4 werden die Muster mit Hilfe eines Metamodells spezifiziert. Einzelne Muster sind daher Bausteine einer Integrationslösung. Der Detailierungsgrad ist dabei so gewählt, dass sie automatisiert in ausführbare Artefakte überführt werden können. EMod

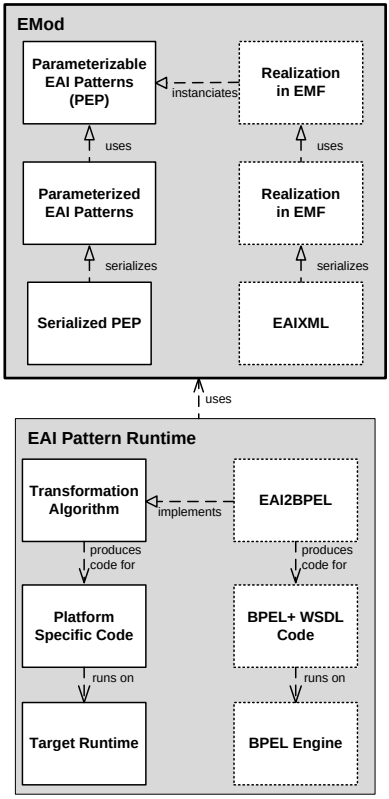


Abbildung 6.2.: EMod Bestandteile

beinhaltet alle parametrisierbaren Muster, die auf dem in Abschnitt 3.2.2 beschriebenen Metamodell basieren. Die einzelnen Muster werden zu einer Integrationslösung zusammengefügt und sind in Form von *parametrisierten Integrationsmustern* als eine konkrete Realisierung einer Integrationslösung in EMod enthalten. Diese Lösung wird in EMod serialisiert, um den Austausch zwischen einzelnen Integrationswerkzeugen zu ermöglichen und um als Basis für Generierungsalgorithmen zu dienen.

6.1.3. Serialisierung von Integrationsmustern

Parametrisierbare Integrationsmuster sollen plattformunabhängig für verschiedene Modellierungs- und Generierungswerkzeuge zur Verfügung stehen. Daher werden die Muster in eine generische, serialisierte Form überführt. EAIXML ist eine solche Serialisierung, welche eine XML basierte, plattformunabhängige Repräsentation parametrisierter Integrationsmuster darstellt. Diese XML Datei dient als generisches Austauschformat zwischen Werkzeugen und Integrationsinfrastrukturen.

In Listing 6.1 ist ein Ausschnitt einer EAIXML Datei dargestellt. Sie zeigt drei verschiedene Integrationsmuster (External Service, Message Translator und Aggregator), die bereits parametrisiert wurden. Die XML Datei ist eine Repräsentation des Metamodells, das unter Abschnitt 3.2.2 beschrieben wurde. Diese Datei dient als Eingabe zur Generierung von ausführbaren Artefakten für unterschiedliche Integrationsinfrastrukturen (siehe Abschnitte 6.3ff).

```
<messageSystem name="integrationSolution" xmlns:is="http://www.example.com/
        integrationsolution" targetNamespace=" http://iaas.uni−stuttgart.de/emod">
...
<externalService id="1" name="IncomingMessageEndpoint">
   <interfaceToExternalWebservice>
      <call>receive</call>
      <portTypeForInvokeCall>is:incomingPT</portTypeForInvokeCall>
      <operationForInvokeCall>is:incomingOp</operationForInvokeCall>
   </interfaceToExternalWebservice>
   <connections>
      <outputChannel channelId="2"/>
   </connections>
</externalService>
...
<messageTranslator id="5">
   <connections>
      <inputChannel channelId="4"/>
      <outputChannel channelId="6"/>
   </connections>
   <stylesheetURI>stylesheet.xsl</stylesheetURI>
</messageTranslator>
...
<aggregator id="7">
```

```
    <interfaceToExternalWebservice>
      <call>invoke</call>
      <portTypeForInvokeCall>is:aggregatePT</portTypeForInvokeCall>
      <operationForInvokeCall>is:aggregateOp</operationForInvokeCall>
    </interfaceToExternalWebservice>
    <connections>
      <inputChannel channelId="6"/>
      <outputChannel channelId="8"/>
    </connections>
    <channelNameForExternalEvent/>
    <callbackOperation>is:aggregateCallbackOp</callbackOperation>
    <completenessCondition>wait for all</completenessCondition>
  </aggregator>
  ...
  </messageSystem>
```

Listing 6.1: Ausschnitt einer EAIXML Datei

6.1.4. EMod Modellierungswerkzeuge

Zu einem EMod gehören auch ein oder mehrere Modellierungswerkzeuge. Ein EMod Modellierungswerkzeug bietet die Funktionalität, eine valide EMod Instanz, also parametrisierte Integrationsmuster, zu erzeugen. Verwendet man die Standard-Serialisierung von EMod – eine EAIXML Datei – kann man jeden XML- oder Text-Editor verwenden, um eine solche Datei zu erzeugen. Da die Erstellung einer Integrationslösung durch einen Text-Editor sehr aufwändig und fehleranfällig ist, erleichtert ein Editor mit graphischer Benutzungsoberfläche die Erstellung einer solchen Lösung. In Kapitel 8 wird exemplarisch das entsprechende Werkzeug GENIUS beschrieben. Es unterstützt EMod, indem es parametrisierbare Integrationsmuster als Modellierungselemente anbietet und die dadurch entstehenden Modelle in Form von EAIXML ablegt. Diese EAIXML Datei wird von Algorithmen, die in GENIUS enthalten sind, verwendet, um daraus ausführbare Artefakte zu erzeugen, die auf unterschiedlichen Integrationsinfrastrukturen ausgeführt werden können. Die Algorithmen sind aufgrund der modularen Architektur allerdings nicht eng mit GENIUS verwoben, sondern können auch unabhängig von GENIUS genutzt werden (Details sind in

Abschnitt 8.6 zu finden).

6.1.5. Laufzeitumgebung für Integrationsmuster

Eine Laufzeitumgebung für Integrationsmuster (*EAI Pattern Runtime*) ist ein System, das eine EMod Serialisierung entgegennimmt (zum Beispiel in Form einer EAIXML Datei), diese in ausführbare Artefakte überführt, die Artefakte auf einer Integrationsinfrastruktur installiert und die Integrationslösung ausführt, um verschiedene Komponenten zu integrieren. Beispiele für eine solche Laufzeitumgebung sind Apache Cimero [Apaf] und Apache Camel [Apab]. Sie führen die Domänen-spezifische Beschreibung einer Integrationslösung (siehe Abschnitt 2.7) aus und überführen parametrisierte Integrationsmuster auf eine auf Web Service basierte Integrationsinfrastruktur. Auf einer anderen Infratruktur wird die Integrationslösung in Form eines WS-BPEL Geschäftsprozesses repräsentiert (siehe Abschnitt 6.2.1). Eine Laufzeitumgebung besteht also im Wesentlichen aus zwei Teilen: (i) dem Generierungsalgorithmus, der Integrationsmuster in ausführbare Artefakte überführt und (ii) der eigentlichen Ausführungsumgebung für die Integrationslösung der Integrationsinfrastruktur.

6.2. Allgemeiner Ansatz zur Erzeugung von ausführbaren Integrationslösungen

Liegt eine Integrationslösung in Form einer EMod Serialisierung vor, wird diese Serialisierung im nächsten Schritt von einem Generierungsalgorithmus in ausführbare Systeme überführt. Diesem Schritt liegt ein generischer Algorithmus zu Grunde, der für unterschiedliche Integrationsinfrastrukturen immer dem gleichen Schema folgt. In Listing 6.2 ist dieser Algorithmus in Pseudo-Quelltext dargestellt.

```
/* part 1 – local part:
   transform each parameterized integration pattern (pep) into
   platform specific code (psc) */
while pep in emod do
   get pep from emod;
   psc := doTransformation(pep);
```

```
    listOfSuccesors := pep.getOutput();
    put (psc, listOfSuccesors) on heap;
end while;
/* part 2 – global part:
   connect psc patterns according to the description in emod
   generate executable artefacts (possibly create package format)
   generate required additional artefacts (e.g. deployment
   descriptor) */
for each pair(pep1, pep2) in connected patterns do
   /* pep1 and pep 2 are connected
      connection can be determined via listOfSuccesors entry of every
        psc in heap **/
   get psc1 and psc2 from heap;
   psc := generateConnection (psc1, psc2);
   put psc on heap;
end for;
/* Generation of additional artefacts for deployment */
if (additional artefacts required) then
   addArtefacts := generateAdditionalArtefacts (heap);
end if;
/* Generation of the deployable unit(s) containing all
   required artefacts */
deploymentUnit := generatedDeployableUnit (heap, addArtefacts);
return deploymentUnit;
```

Listing 6.2: Generischer Generierungsalgorithmus

Da das Integrationsmuster aufgrund der Pipes-and-Filters Architektur zunächst aus eigenständigen, lose gekoppelten Einheiten besteht (siehe Kapitel 5), zeigt sich diese Eigenschaft auch in dem Generierungsalgorithmus. So werden zunächst die einzelnen Muster in einzelne Artefakte überführt. Abhängig von der Zielintegrationsinfrastruktur (PSM) sind die erzeugten Artefakte für sich bereits ausführbar (vergleiche Abschnitt 6.5). Bei anderen Infrastrukturen handelt es sich um einzeln für sich nicht ausführbare Artefakte (vergleiche Abschnitt 6.3). Zur Übersetzung der einzelnen Muster existiert ebenfalls eine generische Funktion (siehe Listing 6.3), die alle möglichen Muster entgegennimmt und abhängig vom Typ des Musters die entsprechende Funktion aufruft. Am Ende der ersten Phase – diese Phase wird auch als lokale Phase bezeichnet – liegen alle Integrationsmuster somit einzeln in der Sprache der Zielplattform

vor.

```
procedure doTransformation(PEP pep) return PSC
begin
  switch (pep.type)
    case "Content−Enrichter"
      begin
        /* here starts the transformation of one single pattern
          according to the parameter settings */
        psc := doTransformationContentEnrichter(pep);
      end;
    case "Message Translator"
      begin
        /* here starts the transformation of one single pattern
          according to the parameter settings */
        psc := doTransformationMessageTranslator(pep);
      end;
    ...
  end switch;
  return psc;
end doTransformation;
```

Listing 6.3: Generischer Generierungsalgorithmus der atomaren Muster

In der zweiten Phase – der globalen Phase – werden die einzelnen Muster gemäß der Spezifikation der Integrationslösung miteinander verbunden. Dieser Schritt unterscheidet sich nun von Zielplattform zu Zielplattform. Würde die EMod Serialisierung auf eine (native) Pipes-and-Filters Implementierung abgebildet werden, müsste man den einzelnen Filtern nur die entsprechenden eingehende(n) und ausgehende(n) Pipes bekannt machen. Im Fall der Abbildung auf WS-* mit einem BPEL Prozess als Integrationslogik werden die einzelnen Integrationsmuster innerhalb eines Prozesses miteinander verbunden. Der Generierungsalgorithmus endet mit der Erstellung aller weiteren Artefakte, die zur Installation auf der Zielinfrastruktur benötigt werden (zum Beispiel *Deployment Descriptor*), sowie der Zusammenführung aller dieser Dateien. Dies kann zum Beispiel ein vordefiniertes Paketformat sein oder einfach nur das Ablegen aller Dateien in einem bestimmten Ordner. In Abbildung 6.3 ist der schematische Aufbau eines solchen Algorithmus dargestellt, mit einer EAIXML Serialisierung als Eingabe und einem installierbaren Paket als Ausgabe.

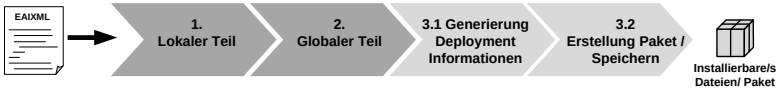


Abbildung 6.3.: Schematischer Aufbau des Generierungsalgorithmus

Prinzipiell existiert bei den in dieser Arbeit favorisierten Abbildungen auf WS-* ein inhärenter Unterschied zur Pipes-and-Filters Architektur (vergleiche Kapitel 5). Eine Alternative wäre gewesen, dass eine Integrationslösung direkt auf eine Implementierung basierend auf PaF abgebildet wird. Allerdings existiert keine standardisierte Möglichkeit, ein PaF System zu beschreiben und dies auf einer entsprechenden Laufzeitumgebung auszuführen. Bei WS-BPEL ist das hingegen anders. Hier existiert der Standard, mit dem Prozesse beschrieben und ausgeführt werden können. Diese Prozesse werden auf einer entsprechenden Laufzeitumgebung (*BPEL Engine*) ausgeführt. Im Falle von PaF existiert keine solche Umgebung. Es müssen proprietäre Systeme verwendet werden, um PaF vergleichbare Ausführungen zu ermöglichen. Oder es muss eine eigene Maschine entwickelt werden, auf der ein PaF System nativ ausgeführt werden kann. In dieser Arbeit wurde dieser Weg allerdings nicht eingeschlagen, da eine Neuimplementierung einer PaF Engine sehr aufwändig und das Ergebnis erneut eine proprietäre Lösung gewesen wäre. Im Verlauf der Arbeit werden daher PaF basierte Integrationsmuster auf vergleichbare beziehungsweise andere Architekturstile überführt.

Der allgemeine Generierungsalgorithmus teilt sich in zwei Phasen. Durch die getrennte Überführung der Integrationsmuster in alleinstehende Teile bietet sich die Möglichkeit, Änderungen an den parametrisierbaren Integrationsmustern leicht in Algorithmen abzubilden. Eine Änderung an einzelnen Mustern (Hinzufügen, Ändern, Löschen von Parametern bei existierenden Filtern oder das Hinzufügen neuer Filter samt Parameter) hat keine Auswirkung auf bereits bestehende Muster. Dies bedingt der modulare Aufbau des Metamodells der parametrisierbaren Integrationsmuster (siehe Abschnitt 3.2.2). Änderungen an den Mustern führen also zu Änderungen am Metamodell (aus Metamodell PIM wird Metamodell PIM', siehe Abbildung 6.4). Der lokale Teil des Algorithmus A

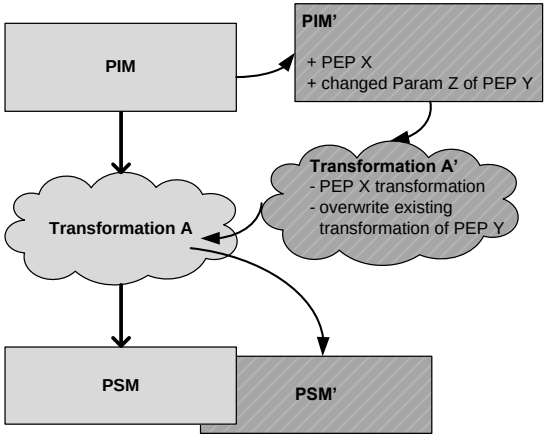


Abbildung 6.4.: Auswirkung von Änderungen an PEP auf Algorithmen

muss nun nur für die veränderten Teile erweitert oder modifiziert werden und resultiert in einem erweiterten Algorithmus A'. Der lokale Teil des Algorithmus A muss dabei für bereits existierende Muster, die nicht angepasst wurden, auch nicht modifiziert werden. Selbst der globale Teil des Algorithmus muss nur bedingt angepasst werden. Hierbei ist nur zu beachten, dass neu hinzugekommene Implementierungen von Integrationsmustern in die ausführbare Integrationslösung aufgenommen werden.

6.2.1. Anforderungen an die Systeme

Prinzipiell müssen im vorgestellten Modell-getriebenen Integrationsprozess keine besonderen Anforderungen an die existierenden oder neu anzufertigenden Systeme gestellt werden. Damit diese allerdings in einer Integrationsschicht aufgerufen werden können, müssen die Schnittstellen der Systeme als Dienste zur Verfügung stehen. Stand der Technik zur Beschreibung solcher Schnittstellen ist die Web Service Technologie, insbesondere WSDL. Die Dienste der Systeme sollten daher als Web Services vorhanden sein. Das bedeutet, dass alle Schnittstellen durch WSDL beschrieben sind und entsprechende Aufrufmechanismen (so genannte *Bindings*) bestehen, durch die eine solche Schnittstelle

beziehungsweise ein dahinter liegender Dienst aufgerufen werden kann.

6.3. Erzeugung von BPEL und Web Services

Im vorherigen Abschnitt wurde das allgemeine Vorgehen bei der Erzeugung von ausführbaren Integrationslösungen vorgestellt. In diesem Abschnitt wird beschrieben, wie eine EMod Serialisierung auf BPEL als Integrationslogik und Web Service Technologie unter Zuhilfenahme des Ansatzes abgebildet werden kann. Zunächst werden die einzelnen atomaren Muster auf BPEL Fragmente abgebildet. Die gesetzten Parameter der einzelnen Integrationsmuster steuern dabei den Algorithmus. Das bedeutet, dass für unterschiedliche Konfigurationen der Parameter auch unterschiedliche BPEL Fragmente erstellt werden. Aufrufe von externen Diensten (External Service) werden mit Hilfe von *Partner Links* [WCL+05] realisiert. Darüber hinaus kann es vorkommen, dass die Funktionalität der Integrationsmuster nicht direkt durch BPEL Artefakte abgebildet werden kann. In einem solchen Fall muss ein externer Dienst aufgerufen werden, der diese Funktionalität bereit stellt. Dieser Dienst wird allerdings nicht generiert. Er muss bereits existieren oder erstellt werden. Sollten externe Dienste involviert sein, kann für jede Interaktion mit diesen Diensten entsprechende Korrelationsinformationen verwendet werden, die während der Parametrisierung spezifiziert wurden. Diese Informationen werden bei der Generierung ind BPEL correlation Elemente überführt.

Ein generiertes BPEL Artefakt wird immer mit einem `Scope` Konstrukt [Org07a] umschlossen. Dies dient zum einen der Strukturierung des BPEL Prozesses, aber vor allem dem, dass die Fehlerbehandlung (siehe Abschnitt 6.3.6) durch die Semantik des Scope Konstrukts erleichtert wird.

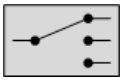
Im zweiten Schritt des Algorithmus werden die einzelnen Artefakte zu einem zusammenhängenden Prozess vereinigt. Dies erfolgt mit BPEL `Links` [Org07a]. Darüber hinaus wird die WSDL des Prozesses erzeugt. Soll nun der Prozess auf einem BPEL Server ausgerollt werden, müssen das entsprechende Paketformat und gegebenenfalls auch die Konfigurationsdateien (sog. *Deployment Descriptor, DD*) erstellt werden. Diese Dateien können nun auf dem entsprechenden Server installiert werden. Die Integrationslösung steht dann zur Ausführung bereit.

In Kapitel 8 wird dieser Mechanismus am Beispiel des Darvien Szenarios angewendet.

In den folgenden Abschnitten wird eine Auswahl der Abbildungen einzelner atomarer Muster aus den verschiedenen Kategorien des Musterkatalogs beschrieben. Insbesondere sind alle diese Muster in den Transformationsalgorithmen des GENIUS Werkzeuges enthalten. Dabei wird auch gezeigt, welche Auswirkungen die unterschiedlichen Konfigurationen einzelner Parameter auf die Generierung haben. Abschließend wird ein zusammengesetztes Muster beschrieben.

Die Abbildungen auf BPEL werden zunächst in BPEL Quelltext dargestellt. Im Laufe des Kapitels werden der besseren Übersicht halber die Quelltexte durch schematische Darstellung in Eclipse BPEL Designer Notation dargestellt [Ecl09].

6.3.1. Content-based Router in BPEL



Der generierte BPEL Quelltext des Musters *Content-based Router* unterscheidet sich je nach Ausprägung der Weiterleitungslogik. Die Weiterleitungslogik kann in Form eines *XPath* [W3C99] Ausdruckes angegeben werden, mit dem anhand des Inhalts der eingehenden Nachricht bestimmt wird, welchen Weg die Nachricht nehmen soll. Ist die Ausdrucksmöglichkeit von XPath allerdings nicht mächtig genug, muss ein externer Dienst aufgerufen werden, der die Berechnung des ausgehenden Kanals bewerkstelligt.

Kann die Weiterleitungslogik durch XPath Ausdrücke spezifiziert werden, kann der Router in zwei unterschiedlichen Ausprägungen übersetzt werden. Die erste Möglichkeit ist in Listing 6.4 dargestellt. Nach einer `Empty` Aktivität [Org07a] (in Listing 6.4 entry genannt) folgen weitere Empty Aktivitäten, die die einzelnen Ausgänge repräsentieren (in Listing 6.4 `exit1`, `exit2`, `exitN` genannt). Die Eingangsaktivität wird mit den einzelnen Ausgangsaktivitäten über Links verbunden. Jeder Link besitzt eine *Transition Condition* [Org07a], die den XPath Ausdruck für den entsprechenden Ausgang enthält und ebenso alle XPath Ausdrücke der anderen Ausgänge in negierter Form. Dadurch wird verhindert, dass eine Nachricht über mehrere Zweige versendet

werden kann. Falls ein `Else` Zweig vorhanden ist, werden alle XPath Ausdrücke in negierter Form als Transition Condition dieses Zweiges verwendet.

```
<bpel:scope name="CBRouterScope">
  <bpel:flow name="CBRouter">
    <bpel:empty name="Entry">
      <bpel:sources>
        <bpel:source linkName="link1">
          <bpel:transitionCondition expressionLanguage="xpath1.0">
            <![CDATA[XPathExpresion1]]>
          </bpel:transitionCondition>
        </bpel:source>
        <bpel:source linkName="link2">
          <bpel:transitionCondition expressionLanguage=" xpath1.0">
            <![CDATA[XPathExpresion2]]>
          </bpel:transitionCondition>
        </bpel:source>
        <bpel:source linkName="link3">
          <bpel:transitionCondition expressionLanguage=" xpath1.0">
            <![CDATA[XPathExpresion3]]>
          </bpel:transitionCondition>
        </bpel:source>
      </bpel:sources>
    </bpel:empty>
    <bpel:empty name="Exit1">
      <bpel:targets>
        <bpel:target linkName="link1"/>
      </bpel:targets>
    </bpel:empty>
    <bpel:empty name="Exit2">
      <bpel:targets>
        <bpel:target linkName="link2"/>
      </bpel:targets>
    </bpel:empty>
    <bpel:empty name="ExitElse">
      <bpel:targets>
        <bpel:target linkName="link3"/>
      </bpel:targets>
    </bpel:empty>
    <bpel:links>
      <bpel:link name="link1"/>
```

```
            <bpel:link name="link2"/>
            <bpel:link name="link3"/>
        </bpel:links>
    </bpel:flow>
</bpel:scope>
```

Listing 6.4: Content-based Router in BPEL

Die zweite Möglichkeit ist die Realisierung der Weiterleitungslogik in Form eines `If` Konstrukts [Org07a]. Bei dieser Variante werden die Überprüfungen in Form von einzelnen `If` beziehungsweise `Elseif` Bedingungen dargestellt. Wird ein *Else* Ausgang benötigt, wird dieser mit einer `else` Bedingung repräsentiert. Verwendet man diese Variante der Umsetzung eines Content-based Router Musters, müssen keine Vorkehrungen getroffen werden insofern, dass Nachrichten doppelt versendet werden. Sobald eine `If`-Bedingung zu *true* evaluiert, wird dieser Zweig beschritten und die anderen Zweige automatisch auf *false* gesetzt. Da dies die übersichtlichere Form im Vergleich zur ersten Variante darstellt, wurde im BPEL Generator dieser Ansatz umgesetzt (siehe Listing 6.5).

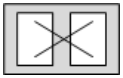
```
<bpel:scope name="CBRouterScope">
    <bpel:flow name="CBRouter">
        <bpel:if name="CBRouterDecision">
            <bpel:condition expressionLanguage="Xpath1.0">
                <![CDATA[XPathExpresion1]]>
            </bpel:condition>
            <bpel:empty name="Exit1"/>
            <bpel:elseif>
                <bpel:condition expressionLanguage="Xpath1.0">
                    <![CDATA[XPathExpresion2]]>
                </bpel:condition>
                <bpel:empty name="Exit2"/>
            </bpel:elseif>
            <bpel:else><bpel:empty name="ExitElse"/></bpel:else>
        </bpel:if>
    </bpel:flow>
</bpel:scope>
```

Listing 6.5: Content-based Router in BPEL mit If-Konstrukt

Eine dritte Möglichkeit der Realisierung des Content-based Router Musters ist die Modellierung der Weiterleitungslogik innerhalb eines `While` Konstrukts [Org07a]. In diesem Fall wird in jedem Schleifendurchlauf für den aktuellen Ausgang überprüft, ob der XPath Ausdruck positiv ausgewertet werden kann. Trifft das zu, kann die Bearbeitung der Schleife sofort abgebrochen werden. In diesem Fall muss keine Vorkehrung getroffen werden, um keine Nachricht doppelt zu versenden. Allerdings ändert sich das Laufzeitverhalten des Prozesses, da die Abarbeitung sequentiell erfolgt und somit länger dauert. Außerdem müssen hier ebenfalls im Anschluss die Ausgänge mit Links und Transition Conditions generiert werden. Aufgrund des Laufzeitverhaltens und des Mehraufwandes im Vergleich zu den beiden zuvor genannten Varianten wurde dieser Ansatz ebenfalls nicht im Generator umgesetzt.

Sollte die Weiterleitungslogik nicht durch XPath berechnet werden können, muss ein externer Dienst aufgerufen werden. Dieser Dienst berechnet anhand der eingehenden Nachricht, an welchen Ausgang die Nachricht weitergeleitet wird. Hier muss eine Abbildung des Ergebnisses des Dienstes auf den Ausgangskanal vorgenommen werden. Dieses Mapping wird ebenfalls über die Parameter vorgenommen. Dabei antwortet der Dienst mit einem bestimmten Wert, der nun auf den entsprechenden Ausgang abgebildet wird. Dies erfolgt wiederum über XPath Ausdrücke. Die eigentliche Weiterleitung der Nachricht erfolgt dann analog zum bereits beschriebenen Vorgehen aus Listing 6.5.

6.3.2. Message Translator in BPEL



Das Muster *Message Translator* kann zwei unterschiedliche Ausprägungen haben, die abhängig von den Werten des Parameters *Transformationslogik* generiert werden. Wird in der Transformationslogik ein XSLT Stylesheet als Transformationslogik angegeben, dann wird eine Assign Aktivität verwendet, die per doXslTransform Operation die eingehende Nachricht in die ausgehende Nachricht umwandelt (siehe Listing 6.6).

```
<bpel:scope name="MessageTranslatorScope">
  <bpel:assign validate="no" name="MessageTranslator">
    <bpel:copy>
```

```
            <bpel:from>
               bpel:doXslTransform("transformation.xsl", inputMessage)
            </bpel:from>
            <to variable="outputMessage"/>
         </bpel:copy>
      </bpel:assign>
   </bpel:scope>
```

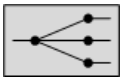
Listing 6.6: Message Translator in BPEL 1

Kann die Transformation nicht durch ein XSLT Stylesheet realisiert werden, muss die Transformation durch einen externen Dienst geleistet werden. Das Vorgehen ist in Listing 6.7 dargestellt. Die im Muster spezifizierte eingehende Nachricht wird auf die eingehende Nachricht des externen Dienstes kopiert. Die Antwort des externen Dienstes wird auf die Ausgangsnachricht des Musters kopiert.

```
<bpel:scope name="MessageTranslatorScope">
   <bpel:sequence>
      <bpel:assign validate="no" name="Assign">
         <bpel:copy>
            <bpel:from variable="inputMessage"></bpel:from>
            <bpel:to variable="externalServiceIn"></bpel:to>
         </bpel:copy>
      </bpel:assign>
      <bpel:invoke name="ExternalService"></bpel:invoke>
      <bpel:assign validate="no" name="Assign1">
         <bpel:copy>
            <bpel:from variable="externalServiceOut"></bpel:from>
            <bpel:to variable="outputMessage"></bpel:to>
         </bpel:copy>
      </bpel:assign>
   </bpel:sequence>
</bpel:scope>
```

Listing 6.7: Message Translator in BPEL 2

6.3.3. Recipient List in BPEL



Die Umsetzung des Musters *Recipient List* in BPEL orientiert sich an der Umsetzung des Content-based Router Musters. Prinzipiell ist es das gleiche Muster, lediglich mit dem Unterschied, dass nicht nur eine Nachricht versendet wird, sondern mehrere Empfänger mit Nachrichten bedient werden. Die Überprüfung der Weiterleitung kann hierbei allerdings nicht mit `If` Konstrukten umgesetzt werden, da normalerweise mehr als ein Ausgangspfad gewählt wird. Daher wird hier auf die Variante mit Transition Conditions zurückgegriffen (vergleiche Listing 6.4). Die Generierung des BPEL Artefakts unterscheidet sich durch die Art der Empfängerliste. Wird die Liste anhand des Inhalts einer Nachricht bestimmt, muss man für jeden Ausgang eine entsprechende Regel formulieren. Dies erfolgt in BPEL durch die Verwendung von XPath Regeln. Das bedeutet, dass jede Ausgangskante (`Link`) eine Transition Condition erhält.

```
<bpel:scope name="RecipientListScope">
  <bpel:flow name="RecipientListFlow">
    <bpel:assign validate="no" name="RemoveList">
      <bpel:sources>
        <bpel:source linkName="link1">
          <bpel:transitionCondition expressionLanguage="Xpath1.0">
            <![CDATA[XPathExpresion1]>
          </bpel:transitionCondition>
        </bpel:source>
        <bpel:source linkName="link2">
          <bpel:transitionCondition expressionLanguage="Xpath1.0">
            <![CDATA[XPathExpresion2]>
          </bpel:transitionCondition>
        </bpel:source>
        <bpel:source linkName="link3">
          <bpel:transitionCondition expressionLanguage="Xpath1.0">
            <![CDATA[XPathExpresion3]>
          </bpel:transitionCondition>
        </bpel:source>
      </bpel:sources>
      <bpel:copy>
        <bpel:from variable="inputWithRecipientList"/>
```

```
            <bpel:to variabl="inputForRecipients"/>
          </bpel:copy>
        </bpel:assign>
        <bpel:empty name="Exit1">
          <bpel:targets><bpel:target linkName="link1"/></bpel:targets>
        </bpel:empty>
        <bpel:empty name="Exit2">
          <bpel:targets><bpel:target linkName="link2"/></bpel:targets>
        </bpel:empty>
          <bpel:empty name="Exit3">
            <bpel:targets><bpel:target linkName="link3"/></bpel:targets>
        </bpel:empty>
        <bpel:links>
          <bpel:link name="link1"/>
          <bpel:link name="link2"/>
          <bpel:link name="link3"/>
        </bpel:links>
      </bpel:flow>
    </bpel:scope>
```

Listing 6.8: Recipient List in BPEL

Ist die Liste Teil der eingehenden Nachricht, können ebenfalls Transition Conditions für jeden Ausgangskanal erstellt werden. Gleiches gilt, wenn die Liste durch einen externen Dienst berechnet werden muss. In diesem Fall wird zunächst der entsprechende Dienst über eine Invoke Aktivität aufgerufen. Die Antwort des Dienstes enthält die Liste aller Empfänger. Soll die eingehende Nachricht dahingehend verändert werden, dass die Empfängerliste aus der weiterzuleitenden Nachricht entfernt werden soll, muss eine Transformationslogik angegeben werden. Hier wird analog zum Muster Message Translator (siehe Abschnitt 6.3.2) der entsprechende BPEL Quelltext erzeugt und vor der Weiterleitungslogik der Nachrichten eingefügt (siehe Listing 6.8).

Der vorgestellte Ansatz ist dann praktikabel, wenn die Anzahl der Empfänger zur Modellierungszeit bereits bekannt sind. Sollte dies nicht der Fall sein, muss eine andere Konstruktion zur Abbildung des Recipient List Musters generiert werden. In diesem Fall wird eine `ForEach` Aktivität eingesetzt, die über ein Array einer Variablen iteriert und entsprechend dem Inhalt des Arrays die entsprechenden Empfänger aufruft. Da die Empfänger erst zur Laufzeit ermit-

telt werden können, insbesondere sind die Adressen erst zur Laufzeit bekannt, beginnt die Berechnung (also Befüllung) des Arrays mit einer `Invoke` Aktivität, die einen entsprechenden Dienst aufruft. Die Antwort enthält das Array mit allen Empfängern (inklusive der entsprechenden Adressen), die aufgerufen werden müssen. Die anschließende `Assign` Aktivität ermittelt die Anzahl der Empfänger, welche von der `ForEach` Aktivität verwendet wird, um die Anzahl der zu erzeugenden `Scopes` zu berechnen. Die `ForEach` Aktivität verwendet die Empfängerliste und erzeugt für jeden Empfänger einen separaten `Scope`, die jeweils parallel zueinander ausgeführt werden. Innerhalb dieser `Scopes` werden die Empfänger aufgerufen. Zu beachten ist, dass in diesem Fall jeder Pfad nach dem Recipient List Muster die gleiche Eigenschaft hat. Vor allem muss gewährleistet sein, dass die jeweiligen Schnittstellen übereinstimmen. Das bedeutet, dass bei jedem Empfänger die Struktur der Eingangsvariable und die Struktur der Ausgangsvariable übereinstimmen muss. Im Falle, dass die maximale Anzahl der Empfänger zur Modellierungszeit bekannt ist, können die verschiedenen Pfade unterschiedliche Bearbeitungen der Nachrichten beinhalten, da diese jeweils separat modelliert werden können. Im dynamischen Fall ist diese Unterscheidung nicht möglich. Nach dem Aufruf des Empfängers (repräsentiert durch die `Empty` Aktivität *„IntermediateStep“*), wird die Antwort des Empfängers auf die Ergebnisvariable der `ForEach` Aktivität kopiert. Sobald alle Empfänger aufgerufen wurden und geantwortet haben, wird die `ForEach` Aktivität beendet und alle Antworten der Empfänger stehen in der Ergebnisvariable bereit zur weiteren Verwendung nachfolgender Muster. Der BPEL Quelltext des Musters Recipient List ist in Listing 6.9 dargestellt.

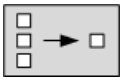
```
<bpel:scope name="RecipientListScope">
  <bpel:flow ame="RecipientListFlow">
    <bpel:links>
      <bpel:link name="link1"/>
      <bpel:link name="link2"/>
    </bpel:links>
  <bpel:invoke name="DetermineRecipients" operation="determineRecipients"
    partnerLink="determineRecipientsPLT" portType="ns1:DetermineRecipientsPT"
    inputVariable="determineRecipientsMsg" outputVariable="recipientsListMsg">
   <bpel:sources><bpel:source linkName="link1"/></bpel:sources>
  </bpel:invoke>
```

```
<bpel:assign validate="no" name="AssignRecipientsCount">
  <bpel:sources><bpel:source linkName="link2"/></bpel:sources>
  <bpel:targets><bpel:target linkName="link1"/></bpel:targets>
  <bpel:copy>
    <bpel:from>count(recipientsList.list/Recipient)</bpel:from>
    <bpel:to varibale="numberOfRecipients"></bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:forEach parallel="yes" counterName="Counter" name="ForEachRecipient">
  <bpel:startCounterValue><![CDATA[1]]></bpel:startCounterValue>
  <bpel:finalCounterValue><![CDATA[numberOfRecipients]]></bpel:finalCounterValue>
  <bpel:completionCondition/>
  <bpel:targets><bpel:target linkName="link2"/></bpel:targets>
    <bpel:scope name="IntermediateSteps">
    <bpel:flow name="RecipientFlow">
      <bpel:links>
          <bpel:link name="link3"/>
          <bpel:link name="link4"/>
      </bpel:links>
      <bpel:assign validate="no" name="AssignRecipient">
        <bpel:sources><source linkName="link3"/></bpel:sources>
          <bpel:copy>
            <bpel:from>(recipientsList.list/Recipient[counter])</bpel:from>
            <bpel:to partnerLink="recipientPLT"></bpel:to>
          </bpel:copy>
      </bpel:assign>
        <bpel:empty name="IntermediateSteps">
        <bpel:sources><bpel:source linkName="link4"/></bpel:sources>
        <bpel:targets><bpel:target linkName="link3"/></bpel:targets>
      </bpel:empty>
      <bpel:assign validate="no" name="AddResponse">
        <bpel:targets><target linkName="link4"/></bpel:targets>
        <bpel:copy>
            <bpel:from variable="recipientResponse"/>
            <bpel:to>
              (recipientsListAnswer.answers/recipientAnswer[counter])
            </bpel:to>
          </bpel:copy>
      </bpel:assign>
    </bpel:flow>
  </bpel:scope>
```

```
    </bpel:forEach>
    </bpel:flow>
</bpel:scope>
```

Listing 6.9: Recipient List mit variabler Anzahl von Empfängern

6.3.4. Aggregator in BPEL



Die Umsetzung des *Aggregator* Musters in BPEL ist komplizierter, da dieses Muster im Vergleich zu den bisher beschriebenen Mustern zustandsbehaftet ist. In einem Aggregator werden die abgearbeiteten Nachrichten gespeichert, um zu überprüfen, wann die Vollständigkeitsbedingung erfüllt ist und um aus allen eingegangenen, zusammengehörenden Nachrichten eine neue Nachricht zu erstellen. Allerdings erleichtert die Eigenschaft eines Geschäftsprozesses als Instanz ausgeführt zu werden in der eine Aggregatorimplementierung ausgeführt wird die Realisierung des Zustandes. Der Aggregator innerhalb eines Prozesses behandelt nur Nachrichten der zugehörigen Prozessinstanz. Somit ist die Korrelation bereits realisiert. Die Umsetzung eines Aggregators in BPEL unterscheidet sich je nach gewählter Vollständigkeitsbedingung und Aggregationsalgorithmus. Es gibt zum einen die Möglichkeit, dies mit BPEL-eigenen Mitteln zu bewerkstelligen oder man verwendet einen externen Dienst, der die Aggregation der Nachrichten vornimmt. In letzterem Fall müssen allerdings die Korrelation der Nachrichten innerhalb des Aggregationsdienstes manuell behandelt werden, da die Nachrichten außerhalb des Kontextes des Prozesses bearbeitet werden.

Zunächst wird die Umsetzung des Aggregators in BPEL samt Berechnung innerhalb des Prozesses beschrieben. Bei der Vollständigkeitsbedingung *wait-for-all* weisen alle eingehenden `Links` auf eine `Assign` Aktivität (siehe Abbildung 6.5). Die eingehenden Links werden in den Abbildungen durch `Empty` Aktivitäten (`EntryN`) repräsentiert. Jede dieser Aktivitäten steht am Ende einer Verbindung vom vorgelagerten Muster zu dem Aggregator Muster. Die `Assign` Aktivität nimmt die eigentliche Aggregation vor. Ist die Aggregationsstrategie dabei *condense*, dann bedeutet die Aggregation das Kopieren der einzelnen Eingangsnachrichten auf eine Ausgangsnachricht. Soll die beste Antwort bestimmt

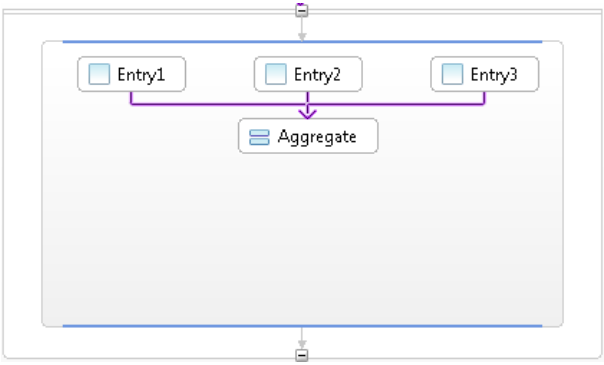


Abbildung 6.5.: Aggregator (wait for all) in BPEL

werden, kann dies zum Beispiel über ein XSL Stylesheet erfolgen. Basiert die Bestimmung der besten Antwort auf dem einfachen Vergleich atomarer Werte (etwa dem Preis), kann man diesen Vergleich auch mit verschiedenen Links und entsprechenden Transition Conditions realisieren (siehe Abbildung 6.6).

Wurde die Vollständigkeitsbedingung *first-best* gewählt, wird immer die zuerst eintreffende Nachricht als Ausgabe des Aggregators verwendet. In diesem

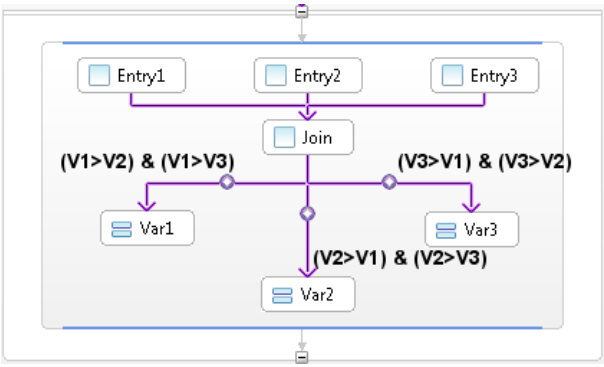


Abbildung 6.6.: Aggregator (wait for all, best answer) bei einfachem Vergleich in BPEL

Fall stehen zu Beginn des Aggregator Musters `Assign` Aktivitäten, die alle Eingänge repräsentieren. Evaluiert ein Link, der auf eine Eingangsaktivität zeigt, zu *true*, so wird im Anschluss ein Fehler vom Typ *FirstBestCompletenessReached* geworfen (siehe Abbildung 6.7). Dieser Fehler wird im entsprechenden *Fault Handler* des `Scopes` gefangen. Dadurch wird der `Scope` abgebrochen, und es können keine weiteren Nachrichten empfangen werden. Im Anschluss kopiert eine weitere Assign Aktivität die eingetroffene Nachricht (*first best*) auf die Ausgangsnachricht (in Abbildung 6.7 `Aggregate` Aktivität). Damit zwischen dem Eintreffen der ersten Nachricht und dem Abbrechen des `Scopes` durch den Fault Handler keine weiteren Nachrichten angenommen werden können beziehungsweise die erste eingetroffene Nachricht eindeutig identifiziert werden kann, werden mehrere Mechanismen eingeführt, die dies bewerkstelligen. Dazu wird eine temporäre Variable angelegt, die die ID des Kanals, über den eine Nachricht eingegangen ist, in einer geordneten Liste (*Array*) in der korrekten Reihenfolge aufnimmt. Eine Eingangsaktivität (`Assign`) kopiert somit zunächst den Wert, der den Nachrichtenkanal identifiziert, in diese Liste.

Die temporäre Variable wird bei den eingehenden Links der Eingangsaktivitäten zur Evaluierung der Transition Condition verwendet. Wenn ein Wert

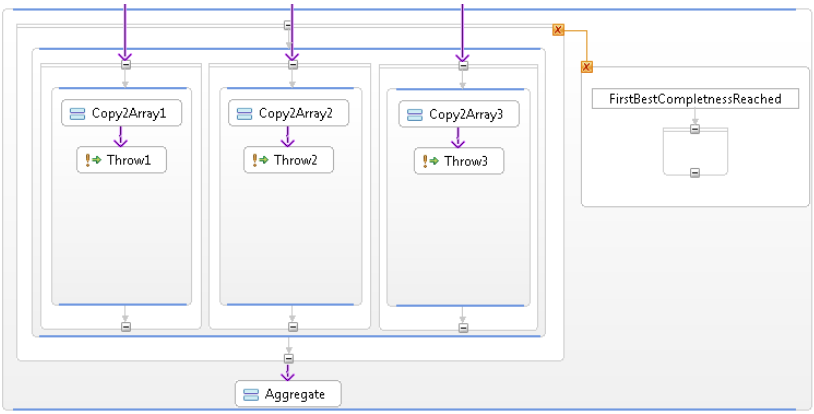


Abbildung 6.7.: Aggregator (first best) in BPEL

in der Liste enthalten ist, evaluiert die Bedingung zu *false*. Dies verhindert, dass eine andere Eingangsaktivität Nachrichten annehmen kann, d.h. den Wert des Nachrichtenkanals in die Liste kopiert, bevor der Fault Handler den `Scope` terminiert. In der Liste steht nun immer zu Beginn der Kanal, über den die erste Nachricht eingetroffen ist. Weitere eingegangene Nachrichten würden über den Kanal in der Liste dahinter geführt. Damit die Eingangsaktivitäten nicht gleichzeitig auf die temporäre Variable schreiben können, sind die einzelnen Eingangsaktivitäten sowie die anschließenden `Throw` Aktivitäten jeweils in einem separaten `Scope` enthalten. Diese `Scopes` besitzen das Attribut `isolated=true`. Dieses Attribut garantiert, dass verschiedene `Scopes` die auf eine gemeinsame Variable zugreifen serialisiert werden. Dadurch wird sichergestellt, dass immer nur eine Aktivität auf die temporäre, gemeinsame Variable zugreifen kann, während den anderen Aktivitäten innerhalb der anderen Scopes der Zugriff verweigert wird. Die letzte `Assign` Aktivität nach diesem `Scope` (Aggregate in Abbildung 6.7) ist dafür zuständig, die eigentliche Ausgangsvariable zu bestimmen. Dazu überprüft sie die Liste und bestimmt, welche Eingangsnachricht als erstes eingetroffen ist. Der erste Wert der Liste repräsentiert den entsprechenden Kanal. Anhand des Kanals kann die Nachricht identifiziert werden. Durch die verschiedenen Schritte wird somit sichergestellt, dass auch tatsächlich die erste eintreffende Nachricht als Ergebnis verwendet wird.

Bei der Vollständigkeitsbedingung *timeout* wird eine weitere Eigenschaft von BPEL verwendet. Zur Überprüfung der Zeit wird ein *Event Handler* für den Aggregator `Scope` definiert, der eine `onAlarm` Bedingung enthält (siehe Abbildung 6.8). Innerhalb dieser Bedingung wird die Zeitdauer eingesetzt, die über die Parameter konfiguriert wurde. Innerhalb des `Scopes` repräsentieren `Assign` Aktivitäten die Eingänge des Aggregators, die wiederum in einzelnen `Scopes` angeordnet sind. Jede `Assign` Aktivität hängt dabei die eingehende Nachricht an eine temporäre Nachricht an. Auch diese `Scopes` werden als *isolated* konfiguriert, damit immer nur genau eine Aktivität auf die Variable zugreifen kann. Sind alle möglichen Nachrichten eingegangen oder ist die Zeit abgelaufen, wird eine weitere `Assign` Aktivität angestoßen, die aus der temporären Variablen die ausgehende Nachricht berechnet. Im Falle eines *timeouts*

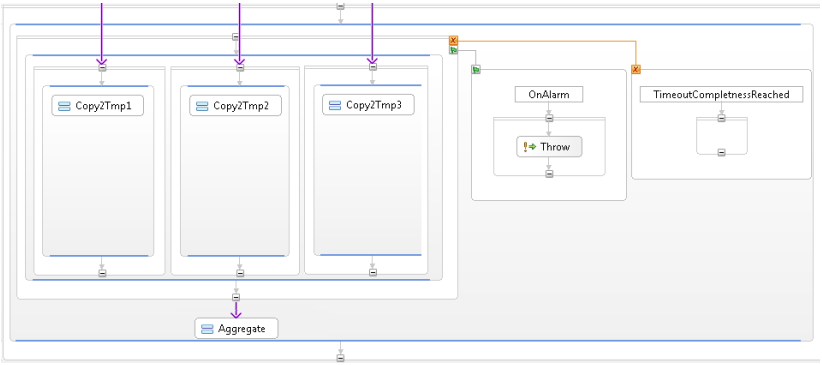


Abbildung 6.8.: Aggregator (timeout) in BPEL

wird der Event Handler mit der `onAlarm` Bedingung aktiv. Innerhalb des Event Handlers wird ein Fehler erzeugt, der vom Fault Handler des `Scopes` abgefangen wird. In diesem Fault Handler befindet sich nur eine Empty Aktivität. Der Scope wird also nach dem Erreichen des timeouts beendet (es werden nun keine weiteren Nachrichten angenommen), und die nachfolgende `Assign` Aktivität errechnet das Ergebnis des Aggregators.

Die Bedingung *timeout-with-overwrite* verwendet das eben vorgestellte Konstrukt eines BPEL Prozesses ebenfalls. Darüber hinaus erhalten die `Assign` Aktivitäten eine nachfolgende Aktivität, die einen Fehler wirft (siehe Abbildung 6.9). Dieser Fehler wird allerdings nur geworfen, wenn die Transition Condition des `Links` zwischen `Assign` und `Throw` zu *true* evaluiert. Dies geschieht, wenn zum Beispiel ein bestimmter Grenzwert überschritten wurde. In einem solchen Fall wird ein Fehler (*TimeoutWithOverwriteCompletenessReached*) geworfen, der vom entsprechenden Fault Handler gefangen wird. Im Anschluss geht es, wie im vorherigen Absatz beschrieben, weiter.

Im Falle der Vollständigkeitsbedingung *External-Event* wird der Aggregator analog zur Bedingung *timeout* erstellt. Der Unterschied liegt darin, dass der onAlarm Handler durch einen `onEvent` Handler ausgetauscht wird (siehe Abbildung 6.10). Dieser Handler wird von außerhalb aktiviert und signalisiert dem Prozess, die Aggregation zu stoppen und aus den bisher erhaltenen

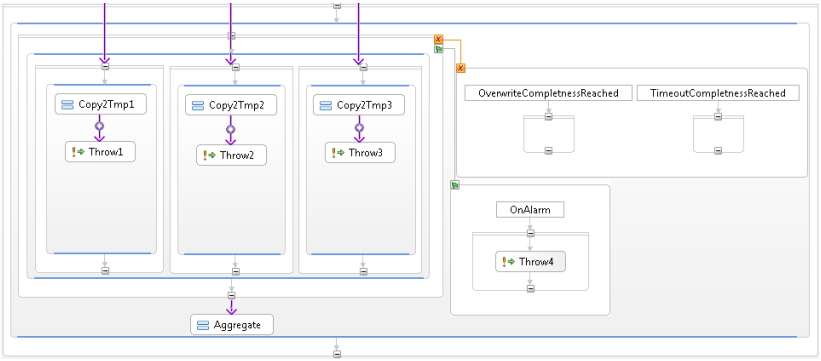


Abbildung 6.9.: Aggregator (timeout with overwrite) in BPEL

Nachrichten ein Aggregat zu erstellen.

Sollte die Aggregation mehrerer Nachrichten so komplex sein, dass sie mit BPEL-eigenen Mitteln nicht möglich ist oder sprechen andere Gründe dafür, die Aggregation auszulagern, wird ein externer Dienst verwendet, der die Aggregation vornimmt. Dieser Dienst muss die Korrelation verschiedener Nachrichten unterstützen, da er Nachrichten von unterschiedlichen Sendern (Prozessinstanzen) entgegennehmen muss. Die Kontrolle über die Vollständigkeitsbedingungen verbleibt allerdings im Prozess. So wird zum Beispiel ein *Timeout* vom

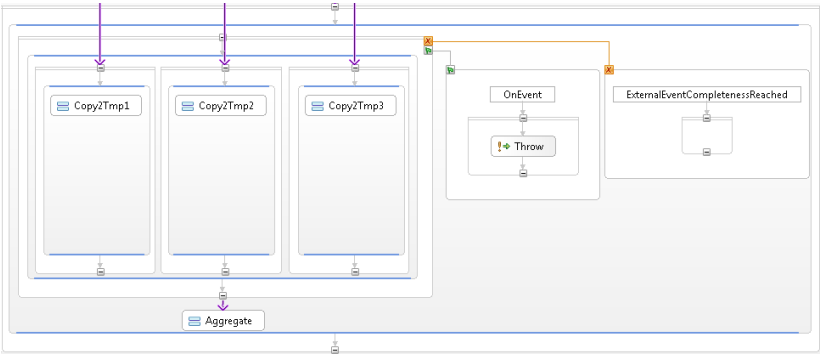


Abbildung 6.10.: Aggregator (external event) in BPEL

Prozess überprüft und nicht vom Dienst selbst. Die Aggregation beim Dienst erfolgt dann in zwei Stufen: zunächst werden die eingegangenen Nachrichten an den Dienst übertragen (*add* Operation). In BPEL ist dies eine `one-way invoke` Aktivität. Zur Korrelation mehrerer Nachrichten verschiedener Prozesse wird in den einzelnen Prozessinstanzen eine eindeutige ID erzeugt, die der Nachricht angefügt wird (manche BPEL Maschinen unterstützen dies, indem sie die Prozess ID über eine entsprechende Funktion direkt in BPEL zur Verfügung stellen). In diesem Fall können die Eigenschaften der internen BPEL Korrelation nicht eingesetzt werden, da diese Fähigkeiten nur innerhalb des BPEL Prozesses beziehungsweise innerhalb der BPEL Maschine zum Tragen kommen. Da allerdings Nachrichten an einen externen (Aggregations-) Dienst gesendet werden, müssen die Korrelationsinformationen somit im Nachrichteninhalt enthalten sein.

Ist die Vollständigkeitsbedingung erfüllt, wird die Annahme von Nachrichten im BPEL Prozess beendet. Das Vorgehen ist hierbei das Gleiche wie in den oben erwähnten Szenarien. Nun ruft der Prozess eine entsprechende Funktion des Dienstes auf (`Aggregate`), er übergibt dabei die Korrelations-ID und erhält als Antwort die aggregierte Nachricht zurück, die an die nachfolgenden Muster weitergeleitet werden kann. Dies wird in BPEL mit einer `two-way invoke` Aktivität dargestellt. Somit unterscheiden sich die vorgestellten Lösungen in BPEL darin, dass bei der Zuhilfenahme eines externen Dienstes die `Assign` Aktivitäten durch `Invoke` Aktivitäten ersetzt werden (vergleiche Abbildung 6.11).

Bei den bisher vorgestellten Aggregator Umsetzungen wird davon ausgegangen, dass die maximale Anzahl der eingehenden Nachrichten zu Modellierungszeit bekannt sind. Ist dies nicht der Fall, muss ähnlich wie bei der Umsetzung des Musters Recipient List ein anderes Vorgehen gewählt werden. Auch in diesem Fall wird das BPEL Konstrukt `ForEach` eingesetzt. Der Aggregator ermittelt mit Hilfe der Zählvariable vorhergehender Aktivitäten, die den Nachrichtenfluss aufgeteilt haben (etwa eine Recipient List), die Anzahl der parallelen Pfade, die erstellt werden müssen, um die Antwortnachrichten entgegenzunehmen. Diese Nachrichten werden innerhalb eines `Scopes` mit dem Attribut `isolated=true` durch eine `Pick` Aktivität angenommen. Die

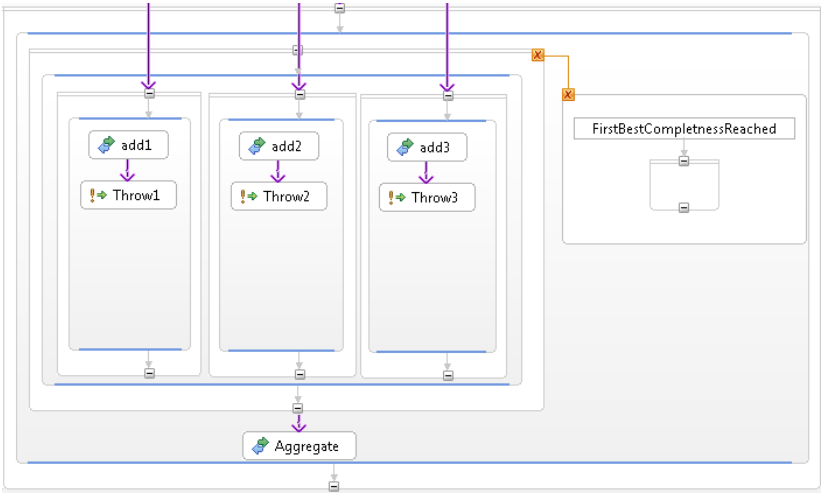


Abbildung 6.11.: Aggregator (first best) mit externem Aggregationsdienst in BPEL

Umsetzung der Vollständigkeitsbedingung und des Aggregierungsalgorithmus erfolgt analog zu den bereits beschriebenen Umsetzungen eines Aggregators. In Abbildung 6.12 ist exemplarisch die Umsetzung mit einem `ForEach` Konstrukt und der Vollständigkeitsbedingung *first best* dargestellt.

Diese Umsetzung ist allerdings nur dann möglich, (i) wenn die Empfänger des Recipient List Musters asynchron aufgerufen werden können und (ii) wenn nach dem Aufruf eines Empfängers keine weiteren Schritte (wie etwa eine Nachrichtentransformation) ausgeführt werden müssen. Kann dies nicht gewährleistet werden, muss die Umsetzung des Aggregator Musters anders erfolgen. Eine Option besteht darin, das Muster direkt in das Verteilungsmuster zu integrieren und dort die kompletten Möglichkeiten eines Aggregators auszuschöpfen. Wenn diese Verzahnung der unterschiedlichen Muster vermieden werden soll, bleibt der Ansatz den Aggregator wie bereits beschrieben nach dem Verteilungsmuster anzuhängen. In diesem Fall kann nur die Vollständigkeitsbedingung *wait for all* umgesetzt werden, da aus einem Scope innerhalb einer `ForEach` Aktivität kein Link gezogen werden kann. Aus die-

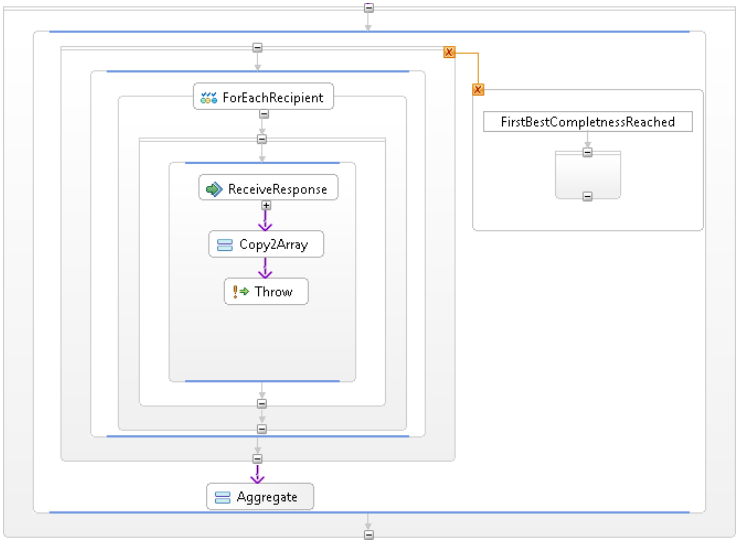


Abbildung 6.12.: Aggregator (first best) mit unbekannter Anzahl von Nachrichten in BPEL

sem Grund kann die der `ForEach` Aktivität folgende Aktivität erst aktiviert werden, wenn die Abarbeitung aller parallelen `Scopes` der `ForEach` Aktivität komplett abgeschlossen ist.

6.3.5. Composed Message Processor in BPEL

In den vorhergegangenen Abschnitten wurden atomare Muster und deren Umsetzung in BPEL diskutiert. Das Muster *Composed Message Processor (CMP)* ist dagegen ein zusammengesetztes Muster, das aus einzelnen kleinen atomaren Mustern aufgebaut ist (vergleiche Abschnitt 4.2.1). Ein solches Muster verwendet daher die generierten BPEL Artefakte und verbindet die Ein- und Ausgänge der Muster über BPEL Links miteinander. Die Kanäle innerhalb des zusammengesetzten Musters werden also auf Links abgebildet. In Abbildung 6.13 ist ein CMP dargestellt, der aus einer Recipient List, drei parallelen Filtern und einem Aggregator mit der *wait-for-all* Strategie besteht. Die Ausprägungen der

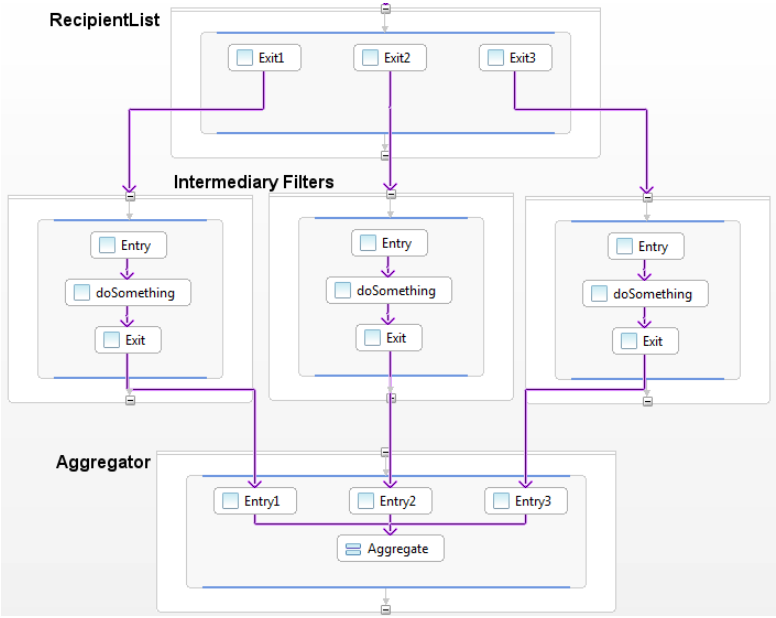


Abbildung 6.13.: Composed Message Processor in BPEL

einzelnen Muster sind in der Abbildung der Einfachheit halber nicht dargestellt.

6.3.6. Fehlerbehandlung in BPEL

In Abschnitt 4.5 wurden zusätzliche Muster beschrieben, die Fehlerbehandlung in Integrationslösungen erlauben. Diese Muster müssen ebenfalls auf BPEL abgebildet werden. Die größte Problematik bei der Umsetzung der Fehlerbehandlung in Integrationsmustern stellt die bisher nicht vorhandene globale Sicht dar. Wie Kapitel 5 verdeutlicht, kann die PaF Architektur nur über Umwege globales Wissen verfügbar machen. Dieses Problem existiert in Workflow-basierten Systemen nicht, da einzelne Prozesse immer in Instanzen laufen, die das globale Wissen einer Instanz einer Integrationslösung vorhalten. Somit kann auch in BPEL dieses Wissen genutzt und für die Fehlerbehandlung verwendet werden. Das Muster Kompensationssphäre (Abschnitt 4.5.2) entspricht dem Konzept

des `Scopes` in BPEL. Eine `Scope` Aktivität ist eine Art Container, in dem BPEL Aktivitäten eines Prozesses zusammengefasst werden und dadurch einen gemeinsamen Kontext besitzen, in welchem sie eine gemeinsame Fehler- und Kompensationsbehandlung teilen [Pap07][Org07a]. Dieses Konzept entspricht eins zu eins dem Konzept der Kompensationssphäre. Wird also ein Bereich einer Integrationslösung (sei es ein atomares Muster oder mehrere Muster) als Kompensationsphäre zusammengefasst und für diese Sphäre eine Fehlerbehandlung beziehungsweise eine Kompensationsbehandlung definiert, wird diese als *Fault Handler* beziehungsweise *Compensation Handler* des Scopes abgebildet. Wie in Abschnitt 4.5.2 erwähnt, wurde das Konzept momentan soweit eingeschränkt, dass sich Scopes nicht überlappen und nicht geschachtelt auftreten dürfen. Diese Einschränkung könnte aber in zukünftigen Versionen aufgelocktert werden.

Darüber hinaus kann für ein Integrationsmuster eine kompensierende Operation modelliert werden (*Compensation Filter*). Da, wie zu Beginn des Abschnitts 6.3 beschrieben, die einzelnen Integrationsmuster jeweils auf einen `Scope` als umschließenden Container abgebildet werden, kann dieser `Scope` auch für die Fehlerbehandlung des einzelnen Musters verwendet werden. Das bedeutet, dass innerhalb des `Scopes` des Integrationsmusters wiederum der Compensation Handler eingesetzt wird, in welchem ein Aufruf an die modellierte Operation (den Compensation Filter) als `Invoke` Aktivität eingefügt wird.

6.4. Erzeugung von Apache Camel

Bei der Generierung von ausführbaren Integrationsmustern lag das Hauptaugenmerk auf der Erzeugung von BPEL. Daneben wurde auch die allgemeine Anwendbarkeit der beschriebenen Methode überprüft. Daher wurden neben BPEL verschiedene Technologien als Integrationsinfrastruktur gewählt, die sich von BPEL deutlich unterscheiden.

Apache Camel bietet ein Rahmenwerk, um die Integration von verschiedenen Komponenten zu ermöglichen. Es wird als Routing- und Mediationsimplementierung von verschiedenen weiteren Apache Projekten verwendet: *Apache MQ*

(ein Messagebroker) oder *Apache ServiceMix* (ein ESB). Zur Konfiguration der Routingregeln bietet Apache Camel zwei Möglichkeiten. Ein Ansatz ermöglicht die Konfiguration über Spring-basierte XML Konfigurationsdateien. Eine weitere Möglichkeit ist die Konfiguration von Camel mit Hilfe einer Java basierten *DSL (Domain Specific Language)*, über die Camel durch die Aneinanderreihung von Java Methodenaufrufen konfiguriert wird. Letztere Variante ist ausdrucksstärker und wird deswegen durch Apache Camel Entwickler favorisiert [Apab]. Aus diesem Grund wurde in dieser Arbeit die Abbildung der parametrisierten Integrationsmuster auf diese DSL vorgenommen.

Die Abbildung der parametrisierten Integrationsmuster auf Apache Camel orientiert sich an dem allgemeinen Generierungsalgorithmus (siehe Abschnitt 6.2): zunächst werden die einzelnen Muster übersetzt, und im zweiten Schritt werden die Verbindungen zwischen den Mustern hinzugefügt. Das Ergebnis ist eine Apache Camel Routinginformation, die auf den entsprechenden Routingsystemen ausgeführt werden kann. Da als Zieltechnologie die DSL von Camel verwendet wurde, ist das Resultat des Generierungsalgorithmus eine eigenständige Java Anwendung, die die Integrationslösung repräsentiert. Diese Anwendung läuft als ein so genannter *Apache Camel Context*. Sie interagiert mit der Außenwelt, indem sie als Web Service zur Verfügung gestellt wird und die Komponenten über Web Services aufruft, die durch das Muster External Service modelliert wurden.

Bei der Generierung wird die Integrationslösung dadurch implementiert, dass Routen dem Kontext hinzugefügt werden, die die einzelnen Filtermuster verbinden und instanziieren. Die Parameter der einzelnen Filter beeinflussen die entsprechende Umsetzung. Im Folgenden wird die Umsetzung einzelner Muster beschrieben. Die detaillierte Beschreibung der Überführung der meisten Integrationsmuster ist in [Kol08] zu finden.

6.4.1. Message Translator in Apache Camel

Apache Camel unterstützt in seinem Rahmenwerk das Muster *Message Translator* bereits auf unterschiedliche Weise. Transformationen auf Transportebene werden implizit von den Apache Camel *Component* Elementen übernommen.

Transformationen der Datenstruktur werden durch speziell zu implementierende *Processor* Elemente realisiert. Diese erlauben die Transformation mit Hilfe von frei wählbarem Java Quelltext oder durch einen Ausdruck (*Expression*), deren Ergebnis den ursprünglichen Nachrichteninhalt ersetzt (siehe Listing 6.10).

```
// general translation by a custom Processor:
from("myComponent:endpointA1").process(myCustomTranslatorProcessor)
   .to("myComponent:endpointB1");
// translation by an Expression:
from("myComponent:endpointA2").setBody(myTranslatorExpression)  .to("myComponent:
      endpointB2");
```

Listing 6.10: Message Translator in Apache Camel DSL

Die Umsetzung der Parameter des Musters *Massage Translator* wurde derart realisiert, dass die Apache Camel Komponente MessageTranslator erweitert wurde. Damit wird erreicht, dass die eingehende Nachricht an eine so genannte *WebServiceFilter* Komponente weitergeleitet wird, wenn die Nachricht durch einen externen Dienst transformiert werden soll. Als Ergebnis erhält die MessageTranslator Komponente die umgewandelte Nachricht. Wird ein XSLT Stylesheet als Transformationsregel in den Parametern angegeben, so verwendet die erweiterte Komponente MessageTranslator die Transformations-API, die durch JAXP [Sun05] bereitgestellt wird. In Listing 6.11 werden beide Ausprägungen des erweiterten MessageTranslators dargestellt: die erste Variante verwendet ein XSLT Stylesheet zur Transformation. Hierbei wird der Komponente MessageTranslator die entsprechende XSL Datei übergeben. Die zweite Variante bedient sich eines externen Dienstes, der die Nachricht transformiert. Dabei wird der Komponente die WSDL-Datei des Dienstes sowie der *ServiceQName* und *PortQName* [CCMW01] übergeben. Die Antwort des externen Dienstes ist die neue Nachricht, die an den ausgehenden Kanal weitergeleitet wird.

```
// messageTranslator with XSLT
URL messageTranslatorXsltLocation = (new    File("generateRecipientList.xsl")).toURI().toURL();
MessageTranslator messageTranslator1 = new  MessageTranslator(
      messageTranslator1XsltLocation);
```

```
// messageTranslator with external Web service
URL messageTranslatorWsdlLocation =
    (new File("messageTranslator.wsdl")).toURI().toURL();
QName messageTranslatorServiceQName = new
    QName(".../MessageTranslator", "messageTranslator");
QName messageTranslatorPortQName = new
    QName(".../MessageTranslator", "messageTranslatorSOAP");
MessageTranslator messageTranslator = new
    MessageTranslator(messageTranslator1WsdlLocation,
                      messageTranslator1serviceQName,
                      messageTranslator1portQName);
```

Listing 6.11: Erweiterterte Message Translator in Apache Camel DSL

6.4.2. Recipient List in Apache Camel

Mit der Klasse *RecipientList* bietet Apache Camel die Möglichkeit, das Muster Recipient List umzusetzen. Die Klasse berechnet anhand eines übergebenen Ausdrucks, an welche Empfänger (in Form von Endpoint-URIs) die Nachricht weitergeleitet werden soll (siehe Listing 6.12). Es wird nicht näher spezifiziert, welche Art dieser Ausdruck haben soll, nur dass der Ausdruck eine Liste mit Empfängern zurückgeben muss (zum Beispiel in Form einer *Collection* oder eines *Arrays*). Camels RecipientList Klasse iteriert dann über diese Liste und sendet Kopien der Nachrichten an die entsprechenden Endpunkte. Das Aussenden der Nachrichten erfolgt sequentiell.

```
from("myComponent:endpointA").recipientList(myExpression);
```

Listing 6.12: Recipient List in Apache Camel DSL

Listing 10: Recipient List in Apache Camel DSL Damit die Parameter aus Abschnitt 4.1.4 unterstützt werden, wurde die Klasse RecipientList um weitere Funktionalität erweitert. Die Erweiterung wurde analog zur Erweiterung bei der Klasse MessageTranslator vorgenommen. Ist die Liste innerhalb der Nachricht enthalten, wird ein XSL Stylesheet angegeben, das aus der eingehenden Nachricht eine Liste mit den Empfängern erzeugt. Analog verhält es sich, wenn die Liste der Empfänger zwar nicht in der Nachricht enthalten ist, diese aber

anhand des Inhaltes berechnet werden kann. Auch in diesem Fall wird ein entsprechendes Stylesheet angegeben. Sind die Empfänger nicht in der Nachricht enthalten und können sie nur durch einen externen Dienst berechnet werden, muss auch hier eine WebServiceFilter Komponente eingesetzt werden. Diese erhält als Übergabewerte die WSDL-Datei des externen Dienstes sowie den *ServiceQName* und *PortQName*. Die Antwort des externen Dienstes ist eine Liste mit Empfängern, die von der ursprünglichen RecipientList Klasse in die entsprechende Form überführt werden kann. Im Anschluss wird die eingegangene Nachricht an die Empfänger weitergeleitet. Auf die Angabe des Quelltextes wird hier verzichtet, da er sich von Listing 6.11 nicht unterscheidet.

6.4.3. Aggregator in Apache Camel

Das Muster *Aggregator* wird durch Apache Camel nicht vollständig unterstützt, allerdings kann man das Konzept so erweitern, dass es alle Anforderungen erfüllt, die durch die Parameter beschrieben werden. Ein Aggregator erhält als Konfiguration unter anderem eine *AggregationCollection*. Diese Sammlung beinhaltet die drei benötigten Eigenschaften des Aggregators: Korrelation, Aggregation und Vollständigkeitsbedingung. In Listing 6.13 ist der schematische Aufbau der Erstellung eines Aggregators dargestellt.

```
AggregationCollection myAggregationCollection = new
    MyAggregationCollection (myCorrelationExpression,
                             myAggregationStrategy,
                            myCompletnessConditionPredicate);
Aggregator myAggregator = new Aggregator(
    getEndpoint("myComponent:endpointA"),
    getEndpoint("myComponent:endpointB").createProducer(),
    myAggregationCollection);
```

Listing 6.13: Aggregator in Apache Camel DSL

Damit nun die Parameter auf die Camel DSL abgebildet werden können, müssen die entsprechenden Strategien implementiert werden. In Listing 6.14 ist eine beispielhafte Umsetzung eines Aggregators mit der Vollständigkeitsbedingung *Timeout* und dem Aggregationsalgorithmus, um die *beste Antwort* zu ermitteln, dargestellt. Zunächst muss die Korrelationsbedingung de-

finiert werden. In dem Beispiel wird hierzu ein XPath Ausdruck definiert (`/*/@lqrq:requestID`), der aus der eingehenden Nachricht das Attribut requestID als Korrelation verwendet. Das bedeutet, dass alle eingehenden Nachrichten innerhalb des Aggregators mit dem gleichen Wert dieses Feldes zu einem Aggregat zusammengefasst werden. Im Anschluss wird die Aggregationsstrategie beziehungsweise der Algorithmus gewählt. In diesem Fall wird die beste Antwort ausgewählt. Die Klasse *AggregationStrategyBest* ist eine selbst entwickelte Klasse. Ihr wird zunächst ein XPath Element übergeben, das das Feld selektiert, über das die beste Antwort bestimmt wird. Hierbei wird wieder ein XPath Ausdruck übergeben. Die nächste Variable definiert, wie die Werte verglichen werden sollen. In diesem Fall wird die Nachricht mit dem geringsten Wert selektiert. Im Anschluss erfolgt die Definition der Vollständigkeitsbedingung. In diesem Fall wird die Bedingung *Never* gewählt, da ein Zeitwert angegeben werden soll, der bestimmt, nach wie vielen Millisekunden die Aggregation angehalten werden soll. Dies wird wenige Zeilen darunter spezifiert (Wert = 3000ms). Die letzten Zeilen des Quelltextbeispiels setzen die Routinginformationen auf und fügen den Aggregator dem Camel Kontext hinzu.

```
// create correlation expression
XPathBuilder<Exchange> aggregatorCorrelationExpr = new
   XPathBuilder<Exchange> ("/*/@lqrq:requestID"); // correlation id
aggregatorCorrelationExpr.setNamespaceContext(namespaceContext);
aggregatorCorrelationExpr.setResultType(XPathConstants.STRING);
// specify aggregation strategy
AggregationStrategy aggregatorAggregationStrategy = new
   AggregationStrategyBest(
      "//lqrq:interestRate",
      AggregationStrategyBest.SelectionStrategy.SELECT_MINIMUM,
      namespaceContext);
// define completeness condition
Predicate<AggregateExchange> aggregatorCompletenessPredicate = new
   CompletnessPredicateNever(); //never complete->only after timeout
// define aggregationCollection with timeout 3000ms
AggregationCollection aggregator1 = new
   ExtendedAggregationCollection(aggregator1correlationExpression,
      aggregator1aggregationStrategy,
```

```
    aggregator1completnessPredicate,
      ExtendedAggregationCollection.
        CORRELATION_IDENTIFIER_RETENTION_PERIOD_NONE, 3000);
  // set up routing
  from("eai2camel:queue:queue10").thread(1).aggregator(aggregator1).
    to("eai2camel:queue:queue11");
  // add aggregator to camel context
  this.getContext().addService(new
    Aggregator(this.getContext().
      getEndpoint("eai2camel:queue:queue10"),
    this.getContext().getEndpoint("eai2camel:queue:queue11").
      createProducer(), aggregator1));
```

Listing 6.14: Aggregator mit Timeout und Bester Antwort in Apache Camel DSL

Alle weiteren Ausprägungen des Aggregators können analog zu dem erwähnten Beispiel umgesetzt werden. Die Implementierungen der unterschiedlichen Aggregationsstrategien können beliebig komplex werden. Es müssen unter Umständen Vorkehrungen getroffen werden, um zum Beispiel das Persistieren von Nachrichten zu ermöglichen, um im Falle eines fehlerbedingten Neustarts wieder den zuletzt gültigen Zustand herstellen zu können. Nähere Informationen dazu können in [Kol08] nachgelesen werden.

6.5. EaaS (EAI as a Service)

Eine weitere Technologie zur Umsetzung von ausführbaren Integrationsmustern wurde aufgrund des aufkommenden Interesses für *Cloud Computing* [Ley09][LKN+09] gewählt. Integrationsmuster (beziehungsweise deren Implementierung) sind Kandidaten, um in einer Cloud Umgebung als wiederverwendbare Artefakte zur Verfügung gestellt zu werden. Integrationsmuster können als einzelne selbstständige Dienste angeboten werden. Dadurch kann *EAI as a Service (EaaS)* ermöglicht werden. Die Lösung ist hierbei rekursiv, denn eine komplette Integrationslösung kann wiederum als eigenständiger Dienst angeboten werden, der in einer anderen Integrationslösung eingebunden werden kann. Die letzte Eigenschaft gilt allerdings auch für die Umsetzung

von ausführbaren Integrationslösungen in WS-BPEL und Apache Camel.

6.5.1. Mandantenfähigkeit

Der Kerngedanke bei EaaS ist, dass Integrationsmuster (in erster Linie Filter Muster wie etwa ein *Message Translator*) als eigenständige lauffähige Komponenten in einer Cloud als Dienste angeboten werden. Da Integrationsmuster auf der PaF Architektur aufbauen, ist diese Abbildung möglich. In PaF existieren einzelne Bausteine (Filter), die eigenständig arbeiten (siehe Abschnitt 2.2.1) und in verschiedenen Szenarien wiederverwendet werden können. Dies ist auch der Grundgedanke bei Cloud Architekturen. Auch hier werden Dienste für mehrere Benutzer beziehungsweise Mandanten (so genannte *Tenants*) in unterschiedlichen Kontexten angeboten. Damit allerdings die einzelnen Komponenten von mehreren Mandanten verwendet werden können, müssen sie konfigurierbar sein. Sie hängen daher von multi-tenancy („Mandantenfähigkeit“) Patterns [MLP08][CC06] ab, die die Benutzung der Integrationsmuster beschreiben und das Deployment einer Integrationslösung in der Cloud definieren. Es existieren drei verschiedene Arten von multi-tenancy Patterns: (i) *single instance*, (ii) *single configurable instance*, und (iii) *multiple instances*. Im ersten Fall ist eine Instanz eines Integrationsmusters für alle Anfragen von mehreren Mandanten zuständig (zum Beispiel *Wire Tap* Muster, das eingehende Nachrichten für Analysezwecke immer an die gleiche Adresse kopiert). Sollte diese Anforderung nicht ausreichen, kann man im zweiten Fall einzelne Integrationsmuster nach den Bedürfnissen des Mandanten konfigurieren. Hier spielen neben Dienstgüteeigenschaften wie Verfügbarkeit und Skalierbarkeit die Parameter der Integrationsmuster eine große Rolle. Je nach Ausprägung der Parameter kann eine laufende Instanz eines Integrationsmusters so angepasst werden, dass man es in individuellen Integrationslösungen verwenden kann. Ein Message Translator Muster ist ein Beispiel hierfür: über die Parameter wird der Transformationsalgorithmus bestimmt (zum Beispiel ein XSL Stylesheet). Eine Instanz des Musters verwendet nun diesen Algorithmus, wenn Nachrichten eines bestimmten Mandanten eintreffen. Ein anderer Mandant verwendet die gleiche Instanz der Musters, nur hat dieser einen anderen Transforma-

tionsalgorithmus gewählt. Mit Hilfe des *single configurable instance* Musters ist die Instanz nun in der Lage, auf verschiedene Kontexte einzugehen. Der dritte Fall bedeutet, dass ein Integrationsmuster aufgrund der Parameter nicht so konfigurierbar ist, dass es von mehreren Mandanten verwendet werden kann. Dies ist zum Beispiel bei einem Aggregator der Fall, der eine komplexe Vollständigkeitsbedingung und einen Algorithmus verwendet, der nicht generisch von einer Instanz abgebildet werden kann. Ein weiteres Beispiel ist, wenn zusätzliche Eigenschaften zu den eigentlichen Parametern hinzukommen (wie etwa Sicherheitseinstellungen), die dazu führen, dass nur dieses Muster gewählt werden kann.

6.5.2. Erzeugung ausführbarer (EaaS) Lösungen

In diesem Abschnitt wird angenommen, dass eine Integrationslösung in einer serialisierten Form zum Beispiel als EAIXML vorliegt. Darin enthalten sind die Integrationsmuster samt Parameter und die Verbindungen zwischen den Mustern. Darüber hinaus ist die Serialisierung mit Provisionierungsinformationen angereichert. Diese Informationen werden von der Cloud Infrastruktur benötigt, um ausführbare Artefakte entsprechend der *multi-tenancy Patterns* zu erstellen und auf der Infrastruktur zu installieren [SML08]. Der Abschnitt beschreibt die Generierung von ausführbaren Integrationslösungen für die Cloud und nicht die Modellierung dieser Integrationslösungen.

Die Generierung von ausführbaren Artefakten für die Cloud als Integrationsinfrastruktur läuft im Wesentlichen wie die Generierung für andere Zielplattformen ab. Zunächst werden die Muster für sich alleine betrachtet und entsprechende Artefakte generiert. Im nächsten Schritt werden die Einzelteile zu einem Ganzen zusammengefügt und auf der Infrastruktur installiert. Im Detail unterscheidet sich die Erzeugung einer ausführbaren Integrationslösung von den bisher vorgestellten Szenarien.

Die Generierung der ausführbaren Integrationslösung hängt in erster Linie von den *multi-tenancy Patterns* ab, mit denen jedes Muster zusätzlich zu den Parametern spezifiziert ist. Damit eine Integrationslösung auf der Cloud Infrastruktur ausführbar ist, werden folgende Artefakte generiert: (i) je nach

multi-tenancy Pattern unterschiedliche ausführbare Komponenten inklusive Deployment- und Konfigurationsdaten für jedes Muster, (ii) ein *Pipes Prozess*, der die einzelnen Muster miteinander verbindet (dies ist ein BPEL Prozess) und (iii) ein Provisioning Flow, der die einzelnen Komponenten aus (i) und (ii) auf der Integrationsinfrastruktur (in der Cloud) installiert.

Abbildung 6.14 zeigt ein einfaches Beispiel eines EMod mit Integrationsmustern, die durch multi-tenancy Patterns annotiert sind. Die weiteren Parameter wurden ausgelassen, da sie für das Beispiel unerheblich sind. Aus diesem EMod resultiert nach der Generierung ein Pipes Prozess und drei unterschiedliche

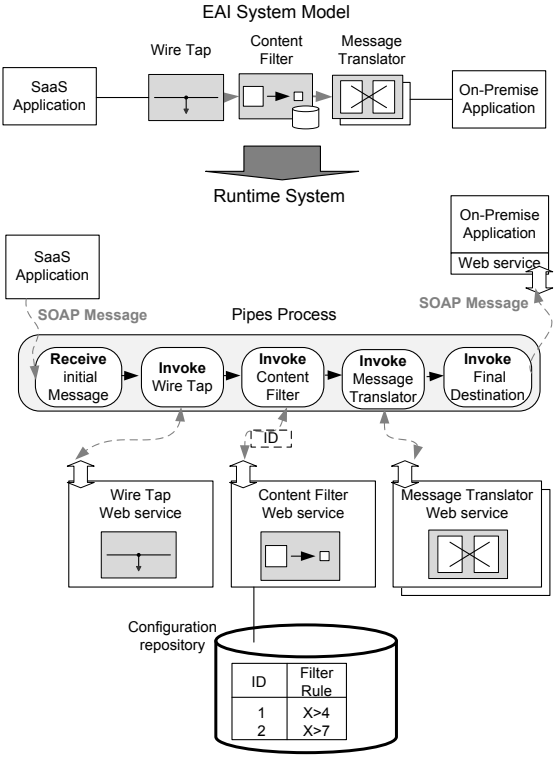


Abbildung 6.14.: EMod und Laufzeitkomponenten für EaaS

Artefakte, die allesamt in der Cloud ausgeführt werden. Das *Wire Tap* Muster ist mit dem *single instance* Muster annotiert. Dies bedeutet, dass eine Instanz dieses Musters alle Anfragen aus allen Integrationslösungen unterschiedlicher Mandanten entgegennimmt und verarbeitet. Darüber hinaus muss dieses Muster nicht für einzelne Lösungen weiter konfiguriert werden. Daher wird eine existierende Instanz des ausführbaren Musters verwendet, und es muss nichts zusätzlich installiert werden. Das Muster *Content Filter* aus Abbildung 6.14 ist als *single configurable instance* gekennzeichnet (erkennbar an dem kleinen Datenbanksymbol). Dieses *multi-tenancy* Muster beschreibt, dass eine laufende Instanz eines Integrationsmusters alle eingehenden Anfragen aller Integrationslösungen bearbeitet. Allerdings werden darüber hinaus noch Konfigurationsdaten erzeugt, über die ein Muster unterschiedliche Eigenschaften zur Laufzeit einnehmen kann. Das bedeutet, dass eine Instanz von verschiedenen Integrationslösungen auf unterschiedliche Weise verwendet werden kann (abhängig von der Konfiguration), ohne dass das Integrationsmuster mehrfach installiert werden muss.

Im Falle, dass ein Integrationsmuster mit dem *multiple instances* Muster (vergleiche Message Translator in Abbildung 6.14) annotiert ist, muss das Integrationsmuster komplett generiert und installiert werden. Das *multiple instances* Muster spezifiziert, dass für jede neue Konfiguration (für jeden Einsatz innerhalb einer einzelnen Integrationslösung) ein entsprechendes Artefakt generiert und dieses als einzelne Komponente installiert werden muss. Dies kann zum Beispiel ein Java Programm Archiv sein, das auf einem Anwendungsserver installiert wird. In solch einem Fall muss der Generierungsalgorithmus zunächst eine Quelltextvorlage aus einem Quelltextrepository abrufen. Anschließend wird der änderbare Teil der Vorlage entsprechend den Parametern des Integrationsmusters angepasst. Zum Abschluss werden alle Artefakte (Klassen, Deployment Descriptor, etc.) zu einem Archiv zusammengefügt und auf der Zielplattform installiert.

Nachdem die installierbaren Artefakte generiert wurden, wird ein Pipes Prozess erzeugt, der die einzelnen ausführbaren Integrationsmuster miteinander verbindet. Dieser Prozess wird durch einen BPEL Prozess repräsentiert, der im Grunde die einzelnen ausführbaren Integrationsmuster nacheinander (oder

parallel) über invoke Aktivitäten aufruft.

Sobald alle beschriebenen Artefakte erzeugt wurden, müssen diese auf der Cloud Integrationsinfrastruktur installiert werden. Dies geschieht mit Hilfe eines so genannten *Provisioning Flows* [KB04][ML08b]. Die Generierung hängt von der Anzahl der Integrationsmuster und der Konfiguration der einzelnen Muster ab. In Abbildung 6.15 ist exemplarisch ein Provisionierungsprozess dargestellt. Er repräsentiert die benötigten Schritte, um das Szenario aus Abbildung 6.14 auf einer Cloud Infrastruktur auszurollen. Die eigentliche Generierung des Prozesses wird in dieser Arbeit nicht betrachtet. Dies kann in 6.14 nachgelesen werden.

Ein Provisionierungsprozess erhält zunächst die kompletten Konfigurationsdaten (Annotation der *multi-tenancy* Muster und Werte der Parameter) aus der EAIXML Datei. Im Anschluss ruft er die Provisionierungsdienste (*Provisioning Services* [KB04][ML08b]) der Cloud Infrastruktur auf. Für jede Ausprägung der *multi-tenancy Patterns* gibt es einen entsprechenden Dienst. Für den *sin-*

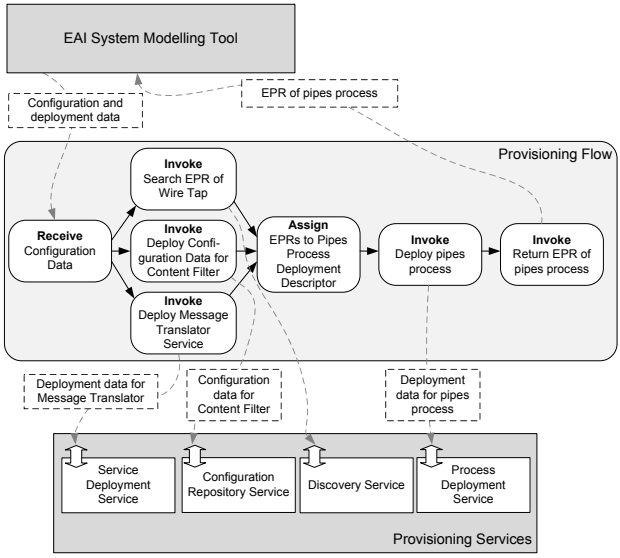


Abbildung 6.15.: EaaS Provisioning Flow

gle instance Fall wird der Endpunkt des entsprechenden Dienstes bestimmt. Handelt es sich um eine konfigurierbare Instanz, werden dem Dienst die Konfigurationsdaten mitgeteilt. Der Dienst speichert diese Informationen in der Konfigurationsdatenbank der Implementierung des ausführbaren Integrationsmusters. Der Provisionierungsdienst gibt als Antwort den Endpunkt des Integrationsmusters zurück, sowie eine eindeutige ID, mit der zur Laufzeit die Konfigurationsdaten durch das Integrationsmuster aus der Datenbank gelesen werden können. Diese ID wird im Pipes Prozess beim Aufruf des Integrationsmusters verwendet (siehe Abbildung 6.14). Die ID ist von Mandant zu Mandant verschieden und ermöglicht somit die Verwendung ein und derselben Instanz in verschiedenen Integrationslösungen durch unterschiedliche Mandanten. Im Falle des Message Translator Musters aus Abbildung 6.14 muss das gesamte Artefakt (zum Beispiel Java Archiv) mit Hilfe eines Dienstes auf der Infrastruktur installiert werden. Auch dieser Dienst liefert nach erfolgreicher Installation einen Endpunkt auf das ausführbare Integrationsmuster zurück.

Wurden alle Dienste aufgerufen und somit die Artefakte der Integrationsmuster installiert, kann der Provisionierungsprozess nun den Pipes Prozess konfigurieren. Dazu wird der *Deployment Descriptor (DD)* des Prozesses mit den erhaltenen Endpunkten (EPRs) gefüllt. Nachdem der DD des Pipes Prozesses konfiguriert wurde, muss der Pipes Prozess selbst mitsamt des DD noch auf die Infrastruktur ausgerollt werden. Ein spezieller Dienst (*process deployment service*), der wiederum den entsprechenden Dienst einer BPEL Maschine aufruft, ist dafür zuständig. Wenn dies erfolgreich abgelaufen ist, ist die Integrationslösung installiert und kann in der Cloud durch Mandanten verwendet werden.

6.6. Ausführbares System für Darvien

Der IT Abteilungsleiter von MehrVomGeld hat entschieden, dass Darvien auf einer SOA implementiert wird, die auf WS-* basiert. Er kann dabei auf Schnittstellenbeschreibungen in WSDL der verschiedenen Anwendungen der einzelnen Banksparten aufbauen, die schon bereitgestellt wurden. Daher können die Sparten ebenfalls über WS-* Technologie aufgerufen werden. Auch der externe

Dienst zur Prüfung der Kreditwürdigkeit besitzt eine WSDL Schnittstelle. Die Komposition der einzelnen Dienste erfolgt daher in Form eines BPEL Prozesses. Die Transformation erfolgt noch manuell, da das Management noch kein Budget für die Erstellung des Werkzeugs (also des Generierungsalgorithmus) freigegeben hat. Der Abteilungsleiter präsentiert daher dem Management die nächste Ausbaustufe der Methode in Form einer Übersichtsdarstellung (siehe Abbildung 6.16).

In zentraler Position ist der BPEL Geschäftsprozess dargestellt. Er wird manuell aus dem modellierten Darvien EMod System abgeleitet, dass in Form einer EAIXML Serialisierung vorliegt. Der BPEL Prozess ruft die einzelnen Dienste als Web Services auf. Die einzelnen BPEL Konstrukte der Muster Content Enricher (das Muster wurde in diesem Kapitel nicht erläutert), Recipient List und Aggregator werden, wie in Abbildung 6.13 illustriert, zu einem gesamten BPEL Prozess zusammengefügt. Der ausführliche BPEL Prozess ist im Anhang dargestellt.

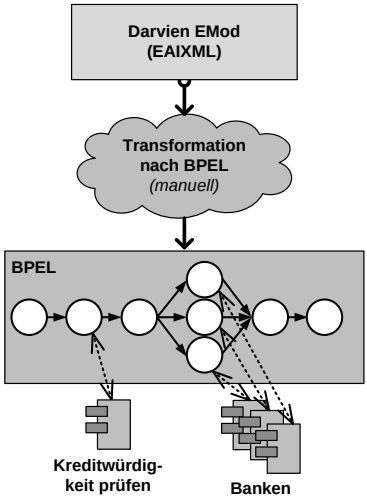


Abbildung 6.16.: Darvien auf einer SOA (Übersicht)

KAPITEL 7

DYNAMISCHES ROUTING VON NACHRICHTEN

Der Begriff des Dienstes bietet eine Abstraktionsmöglichkeit, um ein heterogenes System homogen darzustellen. Durch diese Technik wird auch die Anwendungsintegration erleichtert. Dienste bieten Funktionalität als und werden in einer plattformunabhängigen Art und Weise beschrieben. Die Verbindung zu Transportprotokollen und Serialisierungsformaten wird getrennt von den Schnittstellen beschrieben, um Konfigurierbarkeit zu ermöglichen und lose Kopplung aufrechtzuerhalten. Die *Service-orientierte Architektur (SOA)* verkörpert diese Eigenschaften. Die Basis Middleware einer SOA ist der *Enterprise Service Bus (ESB)*. Der ESB unterstützt *Messaging* als grundlegendes Kommunikationsparadigma.

Die *Web Service Technologie (WS-*)* ist zurzeit die wesentliche SOA Implementierung, die sowohl durch die Industrie als auch durch die Forschung unterstützt wird. Um Interoperabilität zu ermöglichen, wird SOAP als Standard Nachrichtenformat und Nachrichtenverarbeitungsmodell eingesetzt. Ein ESB innerhalb einer SOA realisiert durch WS-* muss daher SOAP unterstützen. Die

SOAP Spezifikation definiert allerdings nicht, in welcher Reihenfolge Nachrichten auf dem Weg von dem Sender zum Empfänger über Zwischenknoten geleitet werden können.

Integrationsmuster, die in erster Linie für die Integration von großen Anwendungen konzipiert wurden, können allerdings auch dazu verwendet werden, Weiterleitungslogik von Nachrichten auf dieser Ebene zu beschreiben. Diese Logik kann in Form von Geschäftsprozessen ausgeführt und für das Weiterleiten von SOAP Nachrichten eingesetzt werden. In diesem Kapitel wird der Ansatz des SOAP Nachrichtenroutings mit Hilfe eines BPEL Prozesses beschrieben. Zunächst wird eine Übersicht des Problembereichs und der momentan ungelösten Probleme bei SOAP Routing gegeben. Im Anschluss wird der neue Ansatz im Detail erläutert und mit der Beschreibung einer prototypischen Implementierung abgeschlossen.

7.1. Routing von Nachrichten in Service-orientierte Architekturen

Die SOAP Spezifikation definiert ein Modell, um SOAP Nachrichten auf einem Nachrichtenpfad abzuarbeiten. Dieser Pfad kann mehrere Zwischenknoten beinhalten und verschiedene Transportprotokolle mit unterschiedlichen Dienstgüteeigenschaften zwischen ihnen unterstützen (siehe Abschnitt 2.3.3). Die Zwischenknoten (*intermediary*) leiten Nachrichten weiter und können die eingehenden Nachrichten selbst mittels zusätzlicher Dienste (*additional services*) bearbeiten (zum Beispiel Logging, Verschlüsselung, etc.). Diese werden durch den Nachrichtenkopf (*header*) einer SOAP Nachricht konfiguriert.

Das eigentliche Weiterleiten einer SOAP Nachricht von einem Zwischenknoten zum nächsten wird durch die SOAP Spezifikation nicht behandelt. Existierende Erweiterungen (siehe Abschnitt 7.1.2) haben folgende Nachteile: (i) die Reihenfolge der zusätzlichen Dienste, die in einem Zwischenknoten abgearbeitet werden sollen, kann nicht spezifiziert werden. Dies ist aber eine wichtige Eigenschaft, denn zum Beispiel sollte die Entschlüsselung erfolgen, bevor Teile der Nachricht bearbeitet werden. (ii) die parallele Ausführung

von Diensten an unterschiedlichen Zwischenknoten wird nicht unterstützt und (iii) zuverlässiges Abarbeiten und Zustellen von Nachrichten im Falle eines fehlerhaften Knotens auf dem Nachrichtenpfad kann nicht garantiert werden.

Aus diesem Grund wurde im Zuge dieser Arbeit eine Lösung entwickelt, die einen Prozess-basierten Ansatz verwendet, um genau diese Nachteile zu adressieren [SKL09]. Da BPEL der Standard für Web Service Kompositionen ist, wurde diese Prozessbeschreibungssprache im Rahmen der Lösung verwendet (allerdings können auch andere Sprachen für den Zweck eingesetzt werden). Generell reduziert sich der Ansatz darauf, den SOAP Nachrichtenpfad auf einen BPEL Prozess abzubilden, in dem jeder Knoten des Pfades durch eine WS Interaktionsaktivität innerhalb des BPEL Prozesses repräsentiert wird. Der BPEL Prozess treibt daher die Entscheidung, welcher Knoten als nächstes ausgewählt wird und welche Dienste dort abgearbeitet werden sollen.

7.1.1. SOAP Processing Model

SOAP definiert ein standardisiertes XML basiertes Nachrichtenformat, eine Menge an Regeln, die beschreiben, wie Dienste Nachrichten bearbeiten sollen und einen Mechanismus, der es erlaubt, SOAP Nachrichten über verschiedene Netzwerktransportprotokolle zu versenden. SOAP Nachrichten werden von einem Ursprungssender (*initial sender*) zu einem Zielempfänger (*ultimate receiver*) gesendet; optional auch über mehrere dazwischenliegende SOAP Knoten (*intermediaries*) (siehe Abbildung 2.2).

Das Transportprotokoll auf dem Nachrichtenpfad kann zwischen einzelnen Knoten unterschiedlich sein. Darüber hinaus können sich die Dienstgüteeigenschaften auf den unterschiedlichen Teilstrecken unterscheiden. Eine SOAP Nachricht besteht aus zwei Teilen: einem Kopf (*header*) und dem eigentlichen Inhalt (*body*) der Nachricht, der für den Zielempfänger bestimmt ist. Der Kopf einer Nachricht beinhaltet Metadaten, die in erster Linie für die Zwischenknoten entlang des Nachrichtenpfades bestimmt sind. Diese Knoten können durch so genannte Rollen adressiert werden. Bestimmte Felder innerhalb des Kopfes definieren, wie eine Nachricht oder bestimmte Teile der Nachricht durch zusätzliche Dienste eines Knotens bearbeitet werden sollen. Allerdings spezifiziert

das SOAP Abarbeitungsmodell nicht, in welcher Reihenfolge die einzelnen Kopffelder abgearbeitet werden sollen. Das widerspricht unter Umständen den Anforderungen einer Anwendungsdomäne oder Sicherheitsmaßnahmen. Außerdem besteht manchmal keine Abhängigkeit zwischen einzelnen Diensten der unterschiedlichen Zwischenknoten. Dadurch wäre eine parallele Abarbeitung der Nachricht an mehreren Knoten gleichzeitig möglich. Diese Möglichkeit wird durch die SOAP Spezifikation nicht ausgeschlossen, allerdings wird dies durch aktuelle SOAP Routing Verfahren nicht unterstützt.

7.1.2. Offene Punkte in SOAP Routing

Der Mechanismus um ein Routing von SOAP Nachrichten zu definieren ist nicht Bestandteil der SOAP Spezifikation. Es gibt bereits verschiedene Ansätze, dieses Problem zu lösen: WS-Routing [NT01], WS-Referral [NCLL01] und WS-Addressing[W3C04][WCL+05].

In den meisten Fällen wird eine statische Route vor dem Versenden der Nachricht definiert, und die Abarbeitung der Nachricht erfolgt nur sequentiell. Beim Auftreten eines Fehlers an einem Knoten oder dem Ausfall eines Knotens muss ein separater Fehlerbehandlungsmechanismus eingeführt werden, der sich um das Auffangen und Abarbeiten des Fehlers kümmert. Dieser Punkt wird in der SOAP Spezifikation nicht berücksichtigt. Somit kann die zuverlässige Auslieferung einer SOAP Nachricht nicht garantiert werden, solange dies nicht durch das Transportprotokoll unterstützt wird.

Die Spezifikation WS-Routing definiert ein Format einer festen Nachrichtenroute zwischen Sender und Empfänger mit Hilfe des Nachrichtenkopfes. Darüber hinaus erlaubt WS-Routing, dass Zwischenknoten zusätzliche Knoten auf dem Nachrichtenpfad einfügen aber nicht löschen dürfen. WS-Referral ist eine weitere Spezifikation, die in Kombination mit WS-Routing eingesetzt werden kann, um die Flexibilität der Nachrichtenroute zu erhöhen. Obwohl auch hiermit definiert werden kann, welche zusätzlichen Dienste an jedem Knoten ausgeführt werden sollen, kann die Reihenfolge der Abarbeitung der zusätzlichen Dienste an einem Knoten nicht bestimmt werden.

Die WS-Addressing Spezifikation [W3C04] definiert ein Modell und ein For-

mat, um Dienstendpunkte in einer technologisch unabhängigen Art und Weise zu beschreiben. Dies wird durch so genannte *Endpoint References (EPR)* [W3C04] ermöglicht. Darüber hinaus beschreibt die Spezifikation die Serialisierung von EPR Informationen in eine SOAP Nachricht. Andere Spezifikationen können diese Information zur Identifikation des nächsten Knotens auf dem Nachrichtenpfad verwenden. Jeder Zwischenknoten kann die Endpunktadresse entsprechend setzen und dadurch den Routingpfad mitbestimmen. Allerdings benötigt jeder Zwischenknoten Wissen über den entsprechenden Nachrichtenpfad, was eine ausführliche Konfiguration der Zwischenknoten mit Routingregeln voraussetzt. Dieser Ansatz wird beispielsweise durch das *Apache Synapse* Projekt [Apae] verfolgt.

Die existierenden Lösungen bieten allerdings immer noch nicht die Möglichkeit, (i) einen SOAP Nachrichtenpfad mit alternativer oder paralleler Abarbeitung von Nachrichten zu spezifizieren und (ii) die Reihenfolge der zusätzlichen Dienste an einem Zwischenknoten zu bestimmen. (iii) Außerdem resultiert die Tatsache, dass Nachrichtenpfade vordefiniert sind und keine Verzweigung erlauben, in einer Fehlerintoleranz der Nachrichtenbearbeitung. Auftretende Fehlersituation können nicht behandelt werden, da die Reaktion auf Fehler nicht spezifizierbar ist. So können zum Beispiel Mechanismen wie Kompensationsbehandlung bereits abgelaufener Nachrichtenbearbeitungen oder das dynamische Auswechseln eines fehlerhaften Knotens im Nachrichtenpfad nicht umgesetzt werden. Diese Nachteile verdeutlichen die Probleme, die durch den nachfolgend vorgestellten Ansatz adressiert werden. Durch die Spezifikation der Nachrichtenroute auf Basis von Geschäftsprozessen (im Folgenden nur noch Prozess genannt) werden diese Probleme gelöst.

7.2. Dynamisches Weiterleiten von Nachrichten mit Hilfe von Geschäftsprozessen

Typischerweise besteht eine Route aus mehreren Knoten, die durch Konnektoren (zum Beispiel Nachrichtenflüsse) miteinander verbunden sind. Ein solcher Fluss kann nach bestimmten Knoten aufgespalten und vor anderen Knoten zu-

sammengeführt werden. Dieses Verhalten entspricht einem Prozess (*workflow*), bei dem Aktivitäten durch Kontrollkonnektoren verbunden sind und dieser Kontrollfluss aufgeteilt (*split*) oder zusammengeführt (*join*) werden kann. Daher sind Prozesse prädestiniert, Routinglogik zum einen zu beschreiben und zu definieren, und zum anderen auch auszuführen. Aus diesem Grund wird BPEL als Prozessbeschreibungs- und Ausführungssprache eingesetzt.

Bei der Betrachtung des Ansatzes wurden zwei fundamental verschiedene Lösungen untersucht: (i) die Routinginformationen werden mit jeder Nachricht von Knoten zu Knoten mitgesendet und (ii) die Routinginformation wird an einer separaten Stelle außerhalb der eigentlichen Nachricht gehalten. Es wird jedes Mal von einem Zwischenknoten darauf zugegriffen, sobald eine SOAP Nachricht eintrifft.

Wenn man den ersten Fall betrachtet, sind alle benötigten Informationen in der SOAP Nachricht enthalten. Sie wächst allerdings auch in der Größe. Dadurch erhöht sich der Aufwand beim Versenden der Nachricht. Darüber hinaus muss jeder Zwischenknoten in der Lage sein, für sich alleine einen BPEL Prozess verstehen und ausführen zu können. Dies ist oftmals nicht realisierbar, da dadurch eine Infrastruktur vorausgesetzt wird, die durch Zwischenknoten selten unterstützt werden kann. Außerdem werden Knoten so zu komplexen Diensten, wohingegen sie ursprünglich nur für einfache Aufgaben, wie etwa Verschlüsselung, gedacht waren. Ferner muss neben dem Prozess auch immer der aktuelle Status (Zustand des Routingprozesses) mitgeliefert werden. Daher muss der Prozess samt seinem Status zunächst serialisiert, später wieder deserialisiert und anschließend in der BPEL Maschine entsprechend verarbeitet werden. Zum aktuellen Zeitpunkt gibt es hierfür keinen Standard, der dies plattformunabhängig spezifiziert. Außerdem wird die Funktionalität, einen bereits gestarteten Prozess zu importieren, danach weiter auszuführen und anschließend den Prozess samt Status wieder zu exportieren, von keiner BPEL Implementierung angeboten.

Im Gegensatz dazu hat die zweite Alternative keine große Auswirkung auf die SOAP Zwischenknoten. Da der BPEL Prozess nicht mit jeder Nachricht mitgeschickt wird, sondern diese nur ein Zeiger (eine Adresse) auf den Prozess (beziehungsweise auf den Web Service, der den Prozess repräsentiert) im Nach-

richtenkopf beinhaltet, muss der Knoten BPEL nicht verstehen und ausführen können. Die Nachricht selbst bleibt ebenfalls klein. Auf Grund dieser Vorteile und der Probleme, die ein Mitführen des Zustandes eines BPEL-Prozesses mit sich bringt, wird in dem hier vorgestellten Ansatz die Lösung mit einem zentralen BPEL Prozess gewählt. Das bedeutet, das Weiterleiten von SOAP Nachrichten wird von einem (externen) Prozess-getrieben, der nicht lokal auf den Knoten ausgeführt wird, sondern zentral in einer entsprechenden BPEL Maschine (vergleiche Abbildung 7.1).

Um diesen Ansatz zu realisieren, muss das SOAP Abarbeitungsmodell im Rahmen der Erweiterungsmöglichkeiten der SOAP Spezifikation angepasst werden. Zunächst muss dazu ein weiteres Kopffeld eingeführt werden, das vor allen anderen Kopfinformationen interpretiert werden muss. Darüber hinaus wird ein neues Protokoll eingeführt, das die Interaktion zwischen den SOAP Knoten und der BPEL Maschine mit dem Routingprozess spezifiziert. Dieser Ansatz kann auch angewendet werden, wenn ein BPEL-Prozesses verteilt ist und ist daher nicht auf eine einzige zentrale Maschine limitiert. Eine Routingbe-

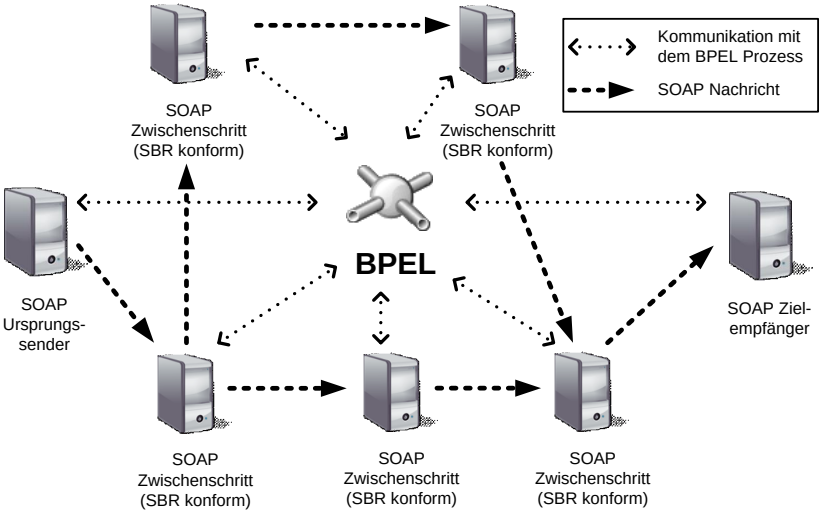


Abbildung 7.1.: Übersicht SOAP Routing über einen zentralen BPEL Prozess

schreibung kann sich über verschiedene selbstständige Prozesse hin erstrecken, wobei jeder einzelne (Teil-) Prozess auf einer separaten Prozessmaschine ausgeführt werden kann. Darüber hinaus ist es generell üblich, eine Infrastruktur anzubieten, in der es keinen „*single point of failure*“ gibt (zum Beispiel ein Cluster von BPEL Maschinen). Der Begriff des „zentralen BPEL Prozesses“ soll den Sachverhalt widerspiegeln, dass SOAP Knoten mit einer externen Maschine kommunizieren und nicht den Prozess selbst ausführen.

Durch den Prozess-basierten Ansatz wird SOAP Routing mit erweiterten Eigenschaften angeboten. (i) Es wird ermöglicht, alternative oder parallele Pfade in der Routinglogik zu spezifizieren. (ii) Der Ansatz unterstützt einen Mechanismus, um die Kopfinformationen an einem Knoten zu ordnen und daher auch die Reihenfolge der Abarbeitung durch Zusatzdienste zu bestimmen. (iii) Der Umstand, dass BPEL Prozesse mit Fehlerbehandlungsmechanismen ausgestattet sind, ermöglicht die Realisierung zuverlässiger Nachrichtenverarbeitung im Falle des Ausfalls von einzelnen Knoten. Dieser neuartige Ansatz ist konform zur SOAP Spezifikation, da er ausschließlich die Erweiterungsmöglichkeiten der SOAP Spezifikation nutzt und keine Auswirkung auf die Spezifikation selbst hat.

7.3. SOAP BPEL Routing (SBR)

In den folgenden Abschnitten wird der neue Mechanismus des Routings von SOAP Nachrichten durch BPEL Prozesse im Detail erläutert. Zunächst werden die neuen Kopffelder beschrieben. Danach werden das Protokoll zwischen Knoten und BPEL Prozess sowie die entsprechenden Schnittstellen und Nachrichten erläutert. Den Abschluss bilden eine ausführliche Betrachtung der Verarbeitung von Routinginformationen an einem Zwischenknoten sowie ein kurzes Beispiel.

7.3.1. Kopfinformationen

Um Routing mit Hilfe von BPEL an SOAP Zwischenknoten anwenden zu können, müssen zunächst die Kopfinformationen einer SOAP Nachricht erweitert werden. Dazu muss die EPR des Routing Prozesses in der Nachricht enthalten

sein, damit der Knoten mit dem Prozess kommunizieren kann. Zur Korrelation eingehender Nachrichten werden entsprechende Daten in der SOAP Nachricht benötigt (zum Beispiel Prozessinstanz ID, Nachrichten ID). Da eine Nachricht innerhalb einer Routinglogik mehrere parallele Pfade durchlaufen kann, muss der entsprechende Pfad ebenfalls noch identifiziert werden. Darüber hinaus muss im Kopf die Reihenfolge der abzuarbeitenden Dienste an einem Knoten enthalten sein. Sollte es sich bei dem Knoten um einen Join-Knoten handeln, müssen außerdem Informationen über die Aggregation mehrerer zusammengehörender Nachrichten mitgeliefert werden (Anzahl der Nachrichten und zugehöriger Aggregationsdienst).

In Abbildung 7.2 sind alle benötigten Kopffelder als XML Schema dargestellt. Das Schemaelement `RoutingInfo` enthält alle benötigten Elemente, um Routinginformationen zu spezifizieren. `messageID` ermöglicht die Korrelation einer Nachricht zu einer bestimmten Prozessinstanz. Die optionalen Felder `replyTo`, `faultTo` und `relatesTo` geben an, wohin eine Antwort oder ein Fehler gesendet werden soll oder definieren eine Beziehung zu einer früher gesendeten Nachricht. Das Element `nodeType` enthält die Pfadidentifikation (im Falle einer parallelen Abarbeitung), den *URI (Uniform Ressource Identifier)* des Knotens an dem die Nachricht bearbeitet wird (`nodeURI`), den URI des Prozesses (`processURI`) sowie die Information, welche Dienste in welcher Reihenfolge ausgeführt werden sollen (service), und die Aggregationsinformationen (`aggregate`). Die Werte der Elemente `service` und `aggregate` verweisen dabei durch einen *QName* auf den entsprechenden Kopf innerhalb der SOAP Nachricht, die den jeweiligen Dienst adressiert.

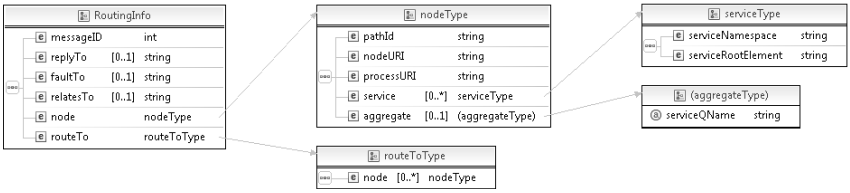


Abbildung 7.2.: Routingkopf Elemente (in XML Schema)

7.3.2. Protokoll zwischen Knoten und Prozess

Der BPEL Prozess, der die Routinglogik beschreibt, steht als Web Service zur Verfügung. Die Schnittstelle ist daher durch eine WSDL-Datei beschrieben. Jeder SOAP Knoten, der mit dem Prozess interagiert, sendet eine RoutingRequest Nachricht an die Schnittstelle, d.h. den Web Service, um die benötigten Informationen zu erhalten (siehe Abbildung 7.3). Die Korrelation der Anfragenachricht und der Prozessinstanz wird durch eine systemweit eindeutige Nachrichten-ID sowie die Prozessinstanz-ID der BPEL Maschine ermöglicht.

Die Antwort des Prozesses beinhaltet die Routinginformationen und wird an den SOAP Knoten in Form einer `RoutingResponse` Nachricht versendet. Die Routinginformationen enthalten eine Liste der nachfolgenden Knoten, an die die bearbeitete Nachricht versendet wird. Jeder Knoten wird durch einen URI adressiert. Diesem Knoten wird optional zusätzlich eine Liste von abzuarbeitenden Zusatzdiensten zugewiesen. Darüber hinaus ist der URI des als nächsten aufzurufenden Routingprozesses (d.h. des Web Services) sowie die Pfad-ID des Nachrichtenflusses enthalten. Das Zusammenführen mehrerer Nachrichten wird durch die optionale Liste des Aggregationsdienstes mit Nachrichteninformationen und dem Dienstnamen ermöglicht.

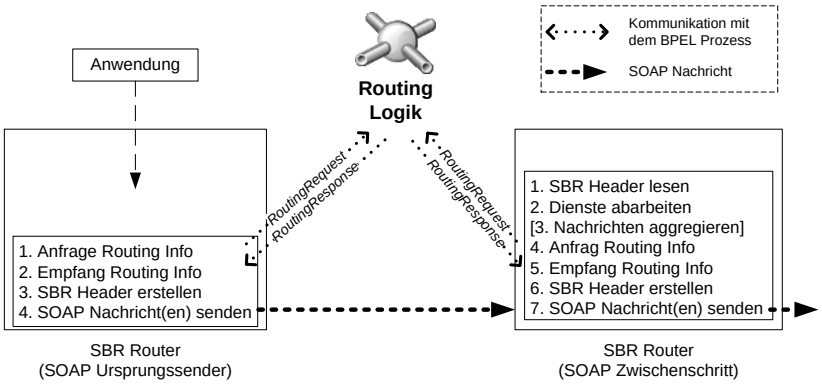


Abbildung 7.3.: Routing Protokoll im Detail

7.3.3. Detailliertes Routingprotokoll

Damit die zusätzlichen Kopffelder an den SOAP Knoten verarbeitet werden können, muss das bestehende SOAP Processing Modell erweitert werden (siehe Abbildung 7.3). Der erste Sender kontaktiert im ersten Schritt den Web Service des Routingprozesses, um die Daten für den ersten Routingschritt zu erhalten. Die Adresse des entsprechenden Prozesses wird dem initialen Sender durch die Konfiguration des Knotens durch einen Administrator vorgegeben. Die Zuordnung einer SOAP Nachricht zu einem Prozess (also eine RoutingRequest Nachricht) erfolgt über eine eindeutige Nachrichten-ID. Die Zuordnung innerhalb der BPEL Maschine erfolgt über BPEL Korrelationsmechanismen. Nach Erhalt der Daten konstruiert der Knoten daraus den *SBR Header*. Die Elemente `messageID`, `pathID`, `nodeID` und `processURI` werden gesetzt. Wenn die Anwendung eine *Request-Response* Kommunikation aufbauen möchte (d.h. der letztendliche Empfänger soll eine Antwort auf die Anfrage an den Ursprungssender zurücksenden), muss das Kopffeld `replyTo` entsprechend gesetzt werden. Nach der Erstellung des SBR Headers wird die SOAP Nachricht an die Empfänger, die unterhalb des node Elements aufgeführt sind, gesendet. Für jede Versendung wird pro Empfänger eine separate Nachricht erstellt.

Nachdem ein SBR kompatibler SOAP Knoten eine Nachricht empfangen hat, liest und interpretiert dieser zunächst den SBR Header. Aus diesem Grund muss das Attribut `mustUnderstand` des Kopffeldes auf `true` gesetzt sowie die Rolle `next` definiert sein. Entsprechend den Informationen des `service` Elements bearbeitet der Knoten die Nachricht, indem alle aufgeführten Dienste in der entsprechenden Reihenfolge aufgerufen werden. Diese Dienste sind auf dem SOAP Knoten vorhanden (d.h. installiert). Nach der Bearbeitung der Nachricht muss der Knoten unter Umständen diese Nachricht mit weiteren Nachrichten aggregieren, und zwar genau dann, wenn der entsprechende Kopf gesetzt ist. Der zuständige Aggregationsdienst ist ebenfalls auf dem SOAP Knoten installiert. Nach erfolgreicher und vollständiger Aggregation kontaktiert der Zwischenknoten den Routingprozess, erhält die Routinginformationen und erstellt den entsprechenden SBR Header wie oben beschrieben. Im Anschluss wird die SOAP Nachricht an den oder die nächsten SOAP Knoten gesendet.

Auch beim letztendlichen Empfänger der Nachricht ist der Mechanismus zur Bearbeitung der Nachricht identisch: zunächst wird der Nachrichteninhalt entsprechend der Kopffelder bearbeitet, danach wird der Routingprozess kontaktiert. Wenn die Antwort des Prozesses ein leeres `routeTo` Element beinhaltet, erkennt der Sender, dass er der letzte Empfänger ist und die Nachricht nicht an weitere SOAP Knoten gesendet werden soll. In diesem Fall leitet der Knoten die Nachricht an die angeschlossene Anwendung weiter. Aufgrund der Korrektheit des BPEL Routingprozesses ist sichergestellt, dass der letztendliche Empfänger auch tatsächlich diese Rolle erfüllen kann.

Spezialfälle innerhalb des Routings von SOAP Nachrichten sind die Knoten, an denen sich der Nachrichtenfluss teilt oder vereinigt wird. Es existieren prinzipiell zwei unterschiedliche Szenarien, um Nachrichten zu teilen: (i) die komplette Nachricht als Ganzes wird kopiert und an die nachfolgenden Zwischenknoten gesendet oder (ii) die Nachricht wird in verschiedene Teile aufgespalten, welche auch überlappend sein können, und wird an verschiedene Nachfolger geschickt. Beide Alternativen führen zu unterschiedlichen Problemen, die durch die Aggregationsknoten beziehungsweise die entsprechenden Dienste behandelt werden müssen. Im ersten Fall müssen verschiedene Kopien der Nachrichten zusammengefügt werden. Dies wird dann problematisch, wenn einzelne Teile der Nachricht von unterschiedlichen Knoten bearbeitet wurden und die bearbeiteten Teile verschiedene Inhalte haben. Ist dies nicht der Fall, ist die Aggregation kein Problem. Gleiches gilt für den Fall, wenn eine Nachricht disjunkt aufgespalten wird und die Zwischenknoten keine identischen Informationen hinzufügen, sondern nur die vorhandenen Informationen bearbeiten. Wurden allerdings die Nachrichten auf den parallelen Wegen an identischen Teilen bearbeitet, führt das ebenfalls zu einem komplexen Verhalten der Aggregation. Allerdings ist die eigentliche Aggregation nicht Teil des SOAP Routing Mechanismus. Es wird davon ausgegangen, dass die Aggregationsdienste auf den entsprechenden Knoten vorhanden sind. Aus diesem Grund wird diese Problematik hier nicht betrachtet. In der aktuellen SBR Implementierung wird nur die Variante der Weiterleitung ganzer Nachrichten unterstützt.

7.3.4. Beispielszenario für SOAP BPEL Routing

Um SBR mit einem praktischen Beispiel zu illustrieren, wird in diesem Kapitel ein einfaches Szenario dargestellt. In Abbildung 7.4 wird die Routinglogik durch Integrationsmuster beschrieben. Die Logik umfasst insgesamt fünf Routingschritte. Im ersten Schritt wird die Nachricht aufgeteilt und anschließend an zwei nachfolgende Router weitergeleitet. Nun durchlaufen die Nachrichten zwei parallele Pfade. Im oberen Pfad wird in Router 2 die Nachricht nochmals per *Message Translator* Muster [HW03] bearbeitet. Im unteren Pfad wird die Nachricht in zwei unterschiedlichen Routern mit Inhalten durch *Content Enricher* Muster [HW03] angereichert. Der letzte Schritt umfasst die Aggregation der beiden Nachrichten, die in Router 1 aus einer Nachricht hervorgegangen sind.

Diese Integrationslösung (also die Routinglogik) wird nun auf einen BPEL Prozess abgebildet. Auch hier kann die Methode der vorhergehenden Kapitel eingesetzt werden, wenn parametrisierte Integrationsmuster zur Modellierung verwendet werden. Sie unterscheidet sich allerdings von der Übersetzung aus Abschnitt 6.3: Bei SBR erzeugt der BPEL Prozess nur die Routinginformationen für jeden Knoten, die eigentliche Bearbeitung und Weiterleitung der Nachricht erfolgt in den SOAP Knoten; in Abschnitt 6.3 hingegen ist der BPEL Prozess selbst für die Bearbeitung und Weiterleitung der eingehenden Nachricht(en) zuständig. Der resultierende (SBR) BPEL Prozess ist in Abbildung 7.5 dargestellt. Die dazugehörige WSDL Datei ist im Anhang zu finden. Die BPEL Datei ist eine

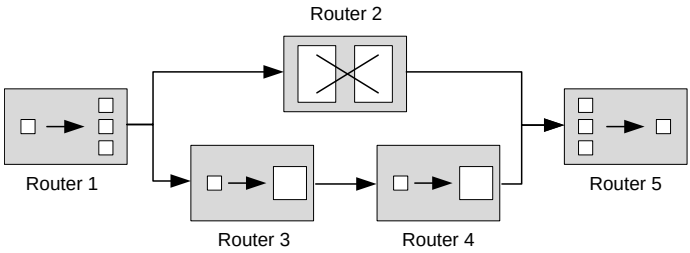


Abbildung 7.4.: Routinglogik beschrieben durch Integrationsmuster

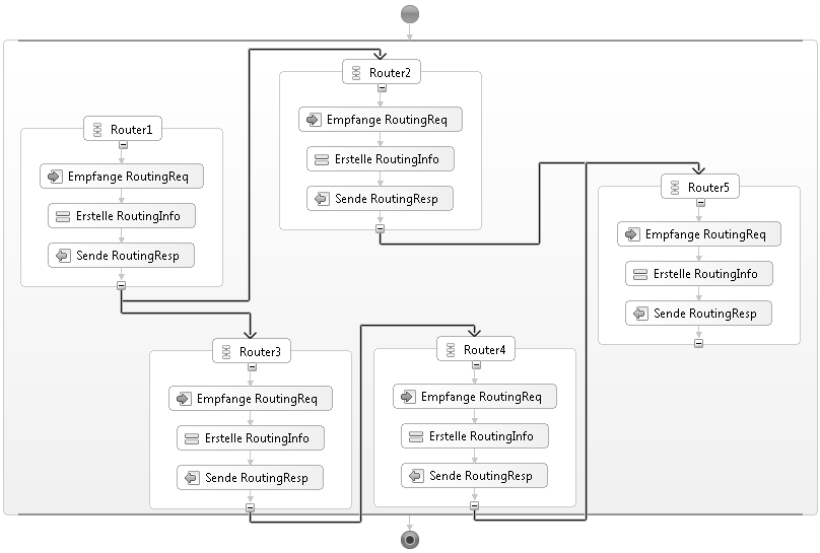


Abbildung 7.5.: BPEL Prozess einer Routinglogik

eins-zu-eins Abbildung der Integrationslösung aus Abbildung 7.4. Jeder Routingknoten wird durch drei Aktivitäten repräsentiert. Eine `Receive` Aktivität nimmt eine `RoutingRequest` Nachricht an. Die `Assign` Aktivität erzeugt die Informationen, die ein SOAP Zwischenknoten benötigt, um den SBR Header zu erzeugen. Die anschließende `Reply` Aktivität antwortet dem SOAP Knoten mit einer `RoutingResponse` Nachricht. Im Falle des ersten Routers umfasst die Antwort den Dienst, der die Aufteilung der Nachrichten durchführt, und eine Liste mit den zwei nachfolgenden Knoten, die als nächstes aufgerufen werden sollen (Router 2 und Router 3). Da in der aktuellen Implementierung keine Aufteilung der Nachrichten unterstützt wird, sondern nur die Weiterleitung der vollständigen Nachrichten, ist die Liste der aufzurufenden Dienste leer.

Der Ablauf des Szenarios sieht dabei wie folgt aus. Zunächst erzeugt der anfängliche Sender den SBR Header für die SOAP Nachricht (siehe Listing 7.1) und sendet diese Nachricht an den ersten Routingknoten. Aus Gründen der Übersichtlichkeit werden in den nachfolgenden Quelltextbeispielen nur Aus-

schnitte dargestellt; insbesondere wird der Nachrichteninhalt ausgelassen. Die ausführlichen Quellen sind in [Juc06] zu finden.

```
<ns1:RoutingInfo soapenv:mustUnderstand="1" soapenv:role="next"
   xmlns:ns1="urn:iaas.uni−stuttgart.de/sbr/2006/08"
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

   <messageId>33ea4f−d5eg41−ab4ca5−5efa3b−7cd901</messageId>
   <replyTo>http://origsender.example.org:8079/resp</replyTo>
   <faultTo>http://origsender.example.org:8079/resp</faultTo>

   <node>
      <pathId>1</pathId>
      <nodeURI>
         http://router1.example.org:8081/SBR−Router/RoutingService
      </nodeURI>
      <processURI>
         http://proc.example.org:8080/active−bpel/services/route1
      </processURI>
   </node>
   ...
```

Listing 7.1: SBR Header des initialen Senders

Der erste Routingknoten interpretiert nach Erhalt der SOAP Nachricht den Nachrichtenkopf, lässt die Nachricht durch Zusatzdienste bearbeiten, soweit diese im Header gelistet sind, und sendet im Anschluss eine Anfrage an den Prozess, den er durch das Element `processURI` adressieren kann. Als Antwort erhält er die Information, dass die Nachricht an zwei nachfolgende Zwischenknoten kopiert werden muss. Damit die Korrelation möglich ist, werden die Pfad-IDs der jeweiligen Pfade angegeben. In Listing 7.2 ist der entsprechende Ausschnitt aus der Antwort des Prozesses dargestellt.

```
<RoutingResponse>
   <messageId>33ea4f−d5eg41−ab4ca5−5efa3b−7cd901</messageId>
   <routeTo>
      <node>
         <pathId>2</pathId>
         <nodeURI>
```

```
            http://router2.example.org:8081/SBR−Service/router2
          </nodeURI>
          <processURI>
            http://proc.example.org:8080/active−bpel/services/route1
          </processURI>
        </node>
        <node>
          <pathId>3</pathId>
          <nodeURI>
            http://router3.example.org:8081/SBR−Service/router3
          </nodeURI>
          <processURI>
            http://proc.example.org:8080/active−bpel/services/route1
          </processURI>
        </node>
      </routeTo>
    </RoutingResponse>
```

Listing 7.2: RoutingResponse des Prozesses für den ersten Routingschritt

Unterhalb des Elementes `routeTo` existieren zwei `node` Elemente, die die nachfolgenden SOAP Knoten angeben. Anhand dieser Information erkennt der SOAP Knoten (Router 1), dass die Nachricht kopiert werden muss und erstellt entsprechende SBR Header für die Nachrichten, die an die nachfolgenden Knoten gesendet werden (siehe Listing 7.3).

```
...
<messageId>33ea4f−d5eg41−ab4ca5−5efa3b−7cd901</messageId>
  ...
    <node>
      <pathId>2</pathId>  (<pathId>3</pathId>)
      <nodeURI>
        http://router2.example.org:8081/SBR−Service/router2
        (http://router3.example.org:8081/SBR−Service/router3)
      </nodeURI>
      <processURI>
        http://proc.example.org:8080/active−bpel/services/route1
      </processURI>
    </node>
```

```
...
```

Listing 7.3: SBR Header für Router 2 (Information für Router 3 in Klammern)

Die nachfolgenden Schritte laufen analog zu den vorherigen Schritten ab. Die jeweiligen Antworten des Prozesses und die daraus resultierenden SBR Header werden hier nicht dargestellt. Diese sind in [Juc06] zu finden. Allerdings unterscheidet sich der letzte Routingschritt an Router 5 von den vorhergehenden. An dieser Stelle müssen zwei eingehende Nachrichten aggregiert werden. Aus diesem Grund unterscheiden sich auch die SBR Header Felder, die der Router bearbeiten muss. In Listing 7.4 ist der Ausschnitt der entsprechenden SOAP Nachricht dargestellt. Über das Element `aggregate` wird dem Dienst mitgeteilt, dass er eingehende Nachrichten durch einen bei sich installierten Aggregationsdienst zusammenführen muss. Über die Elemente `pathId` kann er die Anzahl der eingehenden Nachrichten berechnen (in diesem Fall zwei Nachrichten), die aggregiert werden müssen. Über das Feld `messageId` kann der Aggregationsdienst die entsprechenden Nachrichten korrelieren.

```
...
<messageId>33ea4f−d5eg41−ab4ca5−5efa3b−7cd901</messageId>
...
<aggregate service="http://www.example.org/services/aggregation/a1">
   <pathId>2</pathId>
   <pathId>3</pathId>
</aggregate>
...
```

Listing 7.4: SBR Header für Router 5 (nur Ausschnitt des Aggregationsdiensts)

Nachdem die Nachrichten durch den Aggregationsdienst zusammengeführt wurden, konstruiert der SOAP Knoten die entsprechende Nachricht. Diese erhält er vom Aggregationsdienst. Im Anschluss kontaktiert der Knoten den Routingprozess. Aufgrund des leeren routeTo Elements erkennt der SOAP Knoten, dass er der finale Empfänger ist und sendet die Nachricht an seine angeschlossene Anwendung.

KAPITEL 8

GENIUS - EIN WERKZEUG ZUR ERZEUGUNG AUSFÜHRBARER INTEGRATIONSLÖSUNGEN

Die bisherigen Kapitel haben gezeigt, dass parametrisierbare Integrationsmuster für unterschiedliche Zwecke der Integration eingesetzt werden können. Damit dieser Ansatz tatsächlich anwendbar wird, muss eine entsprechende Werkzeugunterstützung geschaffen werden. Eine solche Werkzeugkette umfasst neben der Abbildung der Integrationsmuster auf eine maschinenlesbare Sprache (siehe EMod, Abschnitt 6.1) und der Bereitstellung von Algorithmen, die diese Sprache in ausführbare Systeme übersetzen (siehe Kapitel 6), auch ein Modellierungswerkzeug, mit dem Integrationslösungen erstellt werden können. Mit einem solchen Modellierungswerkzeug können die Integrationsmuster auf graphische Art und Weise zusammengefügt werden. Die Verwendung eines Texteditors ist ebenfalls möglich (vergleiche Abbildung 6.1). Allerdings wird der vorgestellte Ansatz nur dann wirklich praktikabel, wenn er durch ein Werkzeug mit grafischer Benutzungsoberfläche unterstützt wird und dadurch von

der eigentlichen Zielgruppe genutzt werden kann: Dies sind in erster Linie Systemarchitekten und Integrationsspezialisten und keine Entwickler. Ihre Stärke liegt in der Domäne der Architektur und dem Zusammenspiel von einzelnen Anwendungen beziehungsweise Komponenten und nicht in der tatsächlichen Entwicklung von Anwendungen.

In diesem Kapitel wird ein Werkzeug vorgestellt, das die Erstellung von Integrationslösungen mit Hilfe parametrisierbarer Integrationsmuster auf Basis einer Modell-getriebenen Entwicklung anbietet: *GENIUS (Generating Enterprise Integration Executable Scenarios)* [SL09b]. In den folgenden Abschnitten wird zunächst auf die Eigenschaften des Werkzeugs eingegangen und erläutert, welche Funktionalitäten es bietet. Im Anschluss wird ein Überblick über die Architektur der Software gegeben, zusammen mit der Beschreibung der Erweiterbarkeit bei sich ändernden Anforderungen der Integrationsmuster. Es werden auch Bedienbarkeitsaspekte betrachtet und die Umsetzung der Generierungsalgorithmen aus Abschnitt 6.3 und Abschnitt 6.4 beschrieben. Den Abschluss der Werkzeugbetrachtung bildet ein Überblick über die grafische Benutzungsoberfläche sowie die Modellierung und Ausführung des Beispielszenarios Darvien in GENIUS.

8.1. Funktionalität und Eigenschaften

GENIUS bietet eine grafische Oberfläche, in der ein Benutzer aus einer Symbolleiste heraus Integrationsmuster auf eine Fläche ziehen und diese Muster dort miteinander verbinden kann. Er erstellt somit eine Integrationslösung, die in GENIUS als *Messaging System* bezeichnet wird - analog zum zugrunde liegenden Metamodell, das als Wurzel das Element `MessagingSystem` besitzt (siehe Abschnitt 3.2.2). Jedes einzelne Integrationsmuster muss konfiguriert werden, damit die Integrationslösung, in der es enthalten ist, in ausführbare Systeme transformiert werden kann. Die Parametrisierung der Muster ist gleichbedeutend mit dem Ausfüllen der Parameter aus Kapitel 4. Hierbei ist wichtig, dass GENIUS nicht immer alle Parameter zur Konfiguration anbietet. Kontextbezogen können einzelne Parameter ausgeblendet werden: wenn beispielsweise der Transformationsalgorithmus bei einem *Message Translator* Muster nicht von

einem externen Dienst geleistet wird, dann werden die entsprechenden Parameter für den externen Dienst nicht angeboten. Diese Regeln werden innerhalb der Konfigurationsdialoge jedes einzelnen Musters implementiert. Zusätzlich zu der Konfiguration einzelner Muster kann auch die gesamte Integrationslösung konfiguriert werden. Dies ist zum Beispiel dann nötig, wenn WSDL-Dateien, die von mehreren Mustern verwendet werden, in das System importiert werden sollen. Andere Parameter sind der Name der Integrationslösung oder auch die Angabe des Namespaces der Lösung, falls dies vom Benutzer gewünscht wird.

Bei der Parametrisierung der Muster und der Integrationslösung wird der Benutzer durch GENIUS unterstützt, um fehlerhafter Modellierung vorzubeugen. GENIUS überprüft in bestimmten Fällen, ob der eingegebene Wert eines Parameters erlaubt ist. Dazu gibt es bei manchen Parametern Wertebereiche. Neben dieser Überprüfung kontrolliert GENIUS auch noch anhand der Regeln innerhalb der Konfigurationsdialoge, ob ein einzelnes Muster vollständig und korrekt parametrisiert ist. Wenn dies nicht der Fall ist, erhält der Benutzer eine entsprechende Meldung. Eine weitere Eigenschaft ist, dass GENIUS den Benutzer bei der Erstellung einer Integrationslösung insofern unterstützt, als es nur die Verbindung von zwei Mustern erlaubt, die auch miteinander kombiniert werden dürfen (zum Beispiel ein *Content-based Router* Muster nach einem *Data Type Channel* Muster). Die direkte Verbindung zweier Filtern zum Beispiel kann somit nicht modelliert werden.

Neben diesen syntaktischen Überprüfungen können teilweise auch semantische Überprüfungen vorgenommen werden. Beispielsweise kontrolliert GENIUS, ob der Typ einer ausgehenden Nachricht eines Musters mit dem der eingehenden Nachricht des darauffolgenden Musters übereinstimmt. Unterscheiden sich die entsprechenden Nachrichtenstrukturen (bestimmt durch die Parameter der verbundenen Muster), meldet GENIUS dies als Warnung an den Benutzer. Der Benutzer hat somit die Möglichkeit auf diese Warnung zu reagieren. Diese Regeln sind zum einen durch das Metamodell vorgegeben. Zum anderen sind diese Regeln direkt im jeweiligen Quelltext implementiert (z.B. innerhalb eines Parametrisierungsdialogs).

Eine Einschränkung bei der Modellierung von Integrationslösungen in GENIUS ist, dass zur Modellierungszeit die Anzahl der möglichen Empfänger (etwa

bei einem Recipient List Muster) bzw. die Anzahl der möglichen Ausgänge (beispielsweise bei einem Content Based Router Muster) bekannt sein muss. Eine Modellierung mit etwa nur einem Ausgangskanal, der während der Laufzeit mehrfach „instanziiert“ wird, ist zurzeit nicht möglich. Diese Beschränkung ist allerdings nur durch das Werkzeug vorgegeben. Wie in Abschnitt 6.3 beschrieben, bieten die Generierungsalgorithmen prinzipiell die Möglichkeit, ein solch dynamisches Verhalten abzubilden.

Nachdem eine Integrationslösung modelliert wurde, überführt GENIUS dieses Modell in ein ausführbares System. Dazu besitzt GENIUS eine Reihe von Algorithmen, die ausführbare Artefakte für entsprechende Integrationsinfrastrukturen erzeugen. Die Generierung dieser Artefakte erfolgt allerdings nur dann, wenn GENIUS keinen Fehler in der modellierten Integrationslösung gefunden hat. Das bedeutet, dass auch hier versucht wurde, die Fehlerquote so gering wie möglich zu halten und möglichst kein System zu erzeugen, das nicht korrekt ausgeführt werden kann. Bis zu diesem Schritt sind technische Details der Zielinfrastruktur weitestgehend unberücksichtigt geblieben. Einzig die Verwendung von WSDL-Dateien oder XSLT Stylesheets zur Transformation von Nachrichten sowie Verwendung von XPath bei Vergleichen, sind technische Details, die nicht vor dem Benutzer verborgen werden können. Bei der Generierung von ausführbaren Artefakten ist teilweise das Eingreifen des Benutzers noch notwendig, wenn weitere technische Informationen verlangt werden. Falls in der WSDL-Datei eines Dienstes noch kein konkreter Endpunkt angegeben ist, muss dies während des Deployments nachgeholt werden. Darüber hinaus ist es nötig, dass der Benutzer den Ort angeben muss, an dem die erzeugten Artefakte und die zusätzlichen Dateien (WSDL, Deployment Descriptor, etc.) abgelegt werden müssen, damit die Integrationslösungen ausgeführt werden können (zum Beispiel Deployment Verzeichnis eines Anwendungsservers). Ist es nicht möglich, dass GENIUS auf bestimmte Systeme zugreifen kann, etwa wenn die ausführbaren Artefakte einzig über eine Weboberfläche installiert werden können, erfolgt der letzte Schritt manuell.

8.2. Architektur

Eine der primären Anforderungen an GENIUS war, die Software nicht als komplett eigenständiges Werkzeug mit komplett neuer Benutzungsoberfläche zu entwickeln, sondern, wenn möglich, existierende Programme zu nutzen. Aus diesem Grund fiel die Entscheidung, GENIUS auf Basis der Eclipse Plattform [Ecla] zu entwickeln. Eclipse ist ein quelloffenes Programmierwerkzeug zur Entwicklung von Software unterschiedlichster Arten. Ursprünglich wurde Eclipse als integrierte Entwicklungsumgebung (IDE) für die Programmiersprache Java genutzt. Aufgrund seiner Erweiterbarkeit wird es aber mittlerweile auch für viele andere Entwicklungsaufgaben eingesetzt. Die Plattform bietet durch den Plugin Mechanismus die Möglichkeit, die Entwicklungsumgebung mit weiteren Werkzeugen (*Tools*) zu erweitern (vergleiche Abbildung 8.1). Eclipse selbst ist dabei nur der Kern, der einzelne Plugins lädt, die dann die eigentliche Funktionalität zur Verfügung stellen. Ein entscheidendes Plugin ist das *EMF (Eclipse Modelling Framework)* [Eclb], mit dem sich ein Datenmodell (zum Beispiel auf Basis von UML) erstellen und daraus Java-Quelltext erzeugen lässt. EMF ist somit ein Modell-getriebener Ansatz zur Modellierung und Implementierung von Objektmodellen. Die erzeugten Java Klassen können Instanzen des Modells beziehungsweise der Modellelemente erstellen, abfragen, manipulieren, serialisieren, validieren und Änderungen überwachen. Diese Klassen können auch in einer grafischen Oberfläche verwendet und bearbeitet werden. Auch hierfür bietet die Eclipse Plattform ein entsprechendes Plugin an: *Graphical Editing Framework (GEF)* [Eclc]. GEF erlaubt Entwicklern, mit Hilfe eines existierenden Datenmodells schnell grafische Editoren zu erstellen. Änderungen im Datenmodell werden somit auch in den grafischen Editoren automatisch nachgezogen.

GENIUS besteht im Wesentlichen aus drei Komponenten: Modellierung, Datenmodell und Transformation (siehe Abbildung 8.2). Die Modellierungskomponenten beinhalten alle Module der grafischen Benutzungsoberfläche. Dies sind die Musterpalette, die Zeichenfläche, die Konfigurationsdialoge und die Verifizierungslogik. Das Datenmodell umfasst die Datenhaltung des Metamodells sowie einer modellierten Integrationslösung. Darüber hinaus arbeitet

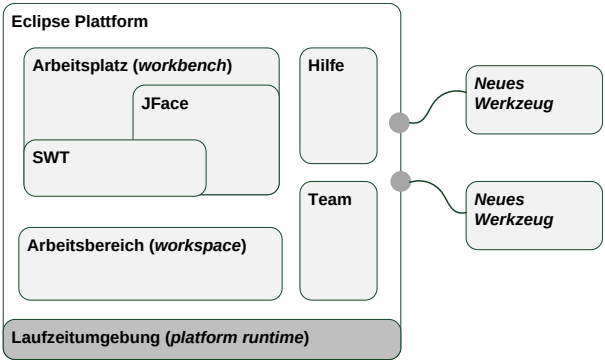


Abbildung 8.1.: Eclipse Plattform Überblick

die Serialisierungskomponente, die für den Ex- und Import von Integrationslösungen zuständig ist, mit diesem Modul. Der dritte Bereich umfasst die Generierungslogik. Jeder mögliche Generierungsalgorithmus steht als separates Modul zur Verfügung. Wie im vorherigen Absatz erläutert, wird GENIUS als Plugin der Eclipse-Plattform implementiert. Um diesem Ansatz treu zu bleiben, wurden die einzelnen Module ebenfalls als Plugin der Plattform implementiert ist. Die Verbindung der Module erfolgt dann mit Hilfe der Eclipse Plattform.

Das Datenmodell ist der Dreh- und Angelpunkt der eigentlichen Oberfläche des Werkzeugs. Da die Integrationsmuster bereits durch ein Metamodell beschrieben wurden, bietet es sich an, EMF zu verwenden, um das Metamodell zu realisieren. Ein EMF Modell wird in Form einer XMI Datei (*XML Metadata Interchange* [Obj07]) serialisiert. Durch die Abbildung der Integrationsmuster auf EMF stehen die Muster nun also in maschinenlesbarer Form zu Verfügung. Das Modell wird als *MessageSystem* bezeichnet und enthält alle Integrationsmuster, die zu einer Integrationslösung hinzugefügt werden können. Die Muster der Kategorie Kanal sind als Unterklassen des Elements *Pipe* repräsentiert, und die Muster der unterschiedlichen Filter Kategorien werden als Unterklassen des Elements Filter dargestellt. Die Parameter der einzelnen Muster werden als Attribute der jeweiligen Klasse in das Modell aufgenommen. Zum Beispiel besitzt das Element *Aggregator* ein Attribut *completenessCondition*, welches die

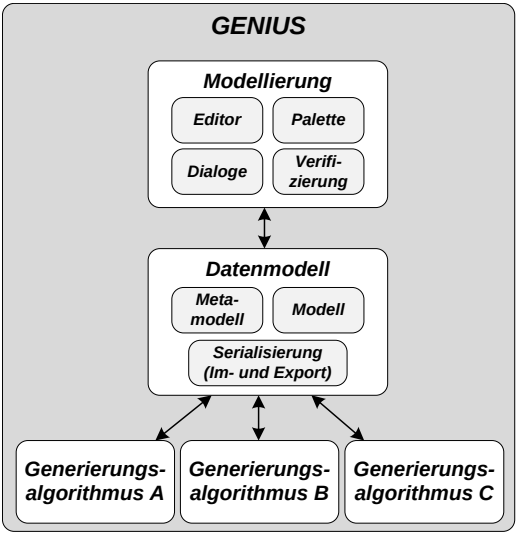


Abbildung 8.2.: GENIUS Architektur Übersicht

Vollständigkeitsbedingung des Aggregators beschreibt.

Die Generierungsalgorithmen, die Systeme für verschiedene Zielinfrastrukturen erzeugen, werden ebenfalls als Plugins in GENIUS integriert. Die Arbeitsweise der Algorithmen ist in Abschnitt 6.2 beschrieben und wird in Abschnitt 8.6 näher erläutert. Die jeweiligen Algorithmen arbeiten immer auf dem Datenmodell von GENIUS, d.h. sie lesen das Modell und erstellen abhängig von der Parametrisierung der einzelnen Muster entsprechende ausführbare Artefakte. Der Kern von GENIUS und die Algorithmen sind somit entkoppelt.

8.3. Erweiterungsmechanismus

Eine Hauptanforderung an GENIUS war, dass die Software leicht anpassbar und erweiterbar sein soll. Dies sollte vor allem in zwei Bereichen möglich sein. (i) Zum einen sollen Änderungen der Integrationsmuster relativ problemlos eingebaut werden können. (ii) Zum anderen sollen Vorkehrungen getroffen werden, um bestehende Algorithmen leicht zu erweitern und weitere

Algorithmen hinzuzufügen.

Die erste Eigenschaft wurde durch den Einsatz von EMF bereits ermöglicht. Durch die Repräsentation der Integrationsmuster in EMF können Änderungen an den Parametern umgehend in EMF abgebildet werden. Änderungen bedeuten, dass neue Integrationsmuster oder neue Parameter für einzelne Muster hinzukommen. Neue Muster werden durch neue Elemente im EMF Modell abgebildet. Neue Parameter werden nachgezogen, indem bei existierenden Klassen die entsprechenden Attribute hinzugefügt werden. Darüber hinaus können existierende Parameter verändert werden, indem die Attribute der Klassen angepasst werden. Sobald die Änderung im EMF Modell erfolgt ist, kann durch Generierung der entsprechenden Klassenhierarchie der Implementierungsquelltext erzeugt werden. Die neuen Elemente stehen somit direkt für GENIUS zur Verfügung, und auch für die Dialoge werden die geänderten Klassen verwendet. Aufgrund dieser Eigenschaft muss nur der anwendungsspezifische Quelltext der unterschiedlichen Klassen manuell implementiert werden. Alles andere nimmt das EMF Plugin dem Entwickler ab.

Neben der Anpassung der Integrationsmuster sollte auch die Integration der Generierungsalgorithmen möglichst lose gekoppelt sein. Aus diesem Grund klinkt sich ein Algorithmus als eigenständiges Plugin über eine so genannte *Action* in GENIUS ein. GENIUS kann somit über diese Verbindung entsprechende Algorithmen aufrufen. Ein Algorithmus verwendet zur Generierung ausführbarer Artefakte das EMF Modell beziehungsweise die Instanz des EMF Modells. Weitere Verbindungen zwischen Algorithmus und dem GENIUS Kern gibt es somit nicht, d.h. Änderungen am Algorithmus haben keine Auswirkungen auf den Kern und insbesondere auch keine Auswirkung auf andere Algorithmen, die als Plugin eingebunden werden können.

Durch den modularen Aufbau der Generierungsalgorithmen (vergleiche Abschnitt 6.2 und Abschnitt 8.6) kann auf Änderungen einzelner Muster des Modells relativ einfach reagiert werden. Allerdings können Änderungen am Modell nicht automatisch in den Algorithmen repräsentiert werden. Zur Generierung der ausführbaren Artefakte ist Wissen notwendig, dass nicht im Modell abgelegt werden kann. Die Algorithmen werden daher manuell angepasst. Man unterscheidet zwei Arten von Änderungen: (i) Anpassungen an existierenden

Mustern, d.h. an den Parametern, (ii) Löschen von existierenden Parametern oder Mustern oder (iii) Hinzufügen von neuen Mustern beziehungsweise Parametern. In den ersten beiden Fällen wird der lokale Teil des Algorithmus direkt abgeändert. Im dritten Fall werden Plugins erstellt, die von dem existierenden Algorithmus geladen werden, sobald dieser auf ein Element trifft, dass ihm nicht bekannt ist. Der entsprechende Algorithmus ist über einen eindeutigen Namen in einer Registry abgelegt und wird darüber gefunden. Somit muss der existierende Algorithmus nicht angepasst werden. Hat die Anpassung des Modells Auswirkungen auf den globalen Teil des Algorithmus, kann auch hier ein entsprechendes Plugin erstellt werden, das bei der Generierung geladen wird.

8.4. Benutzerunterstützung durch GENIUS

GENIUS richtet sich in erster Linie an Personen, die nicht unbedingt fundiertes Wissen der zugrundeliegenden Integrationstechnologie besitzen, auf der eine Integrationslösung später ausgeführt werden soll. Diese Personen kennen sich allerdings sehr gut mit Integrationsmustern, deren Parametrisierung und der Verwendung für Integrationslösungen aus. Allerdings wird das Finden und Erkennen von Fehlern immer komplizierter, je komplexer eine Integrationslösung wird. Damit die Fehlerquote nicht mit der Komplexität ansteigt, bietet GENIUS, wie oben beschrieben, einige Konzepte, um die Erstellung von Fehlern während der Modellierungszeit auf ein Minimum zu reduzieren.

Zwei unterschiedliche Eigenschaften einer Integrationslösung müssen kontinuierlich überprüft werden, um die Erzeugung eines fehlerhaften Modells zu vermeiden: (i) sind alle in der Lösung enthaltenen Integrationsmuster korrekt konfiguriert (parametrisiert) und (ii) sind alle Muster korrekt miteinander verbunden. Die Logik, die die einzelnen Muster überprüft, verifiziert, ob alle benötigten Parameter gesetzt sind und ob die eingegebenen Werte stimmen. Innerhalb des EMF Modells können diese Eigenschaften teilweise modelliert werden. Somit betrachtet dieser Teil der Logik nur den lokalen Teil der Muster, ohne den globalen Zusammenhang mit einzubeziehen. Dies ist ebenfalls eine Eigenschaft, die die Erweiterung und Anpassung von Integrationsmuster

erleichtert: Änderungen an einzelnen Mustern führen nur zu Änderungen an lokalen Teilen der Logik.

Die zweite Eigenschaft betrifft das globale Verhalten einer Integrationslösung. Hierfür gibt es einen zweiten Teil der Logik, der sich um die Verbindung der einzelnen Muster untereinander kümmert. Dieser Teil überprüft zum Beispiel, ob ein ausgehender Nachrichtentyp eines Musters mit dem entsprechenden eingehenden Nachrichtentyp des nachfolgenden Musters korrespondiert. Darüber hinaus verhindert die Logik, dass Muster miteinander verbunden werden, die nicht zusammen passen. Das bedeutet, dass ein Benutzer keine zwei Muster miteinander verbinden kann, die nicht verbunden werden dürfen (etwa zwei Kanäle hintereinander). Zu guter Letzt verhindert die Logik, dass keine Generierungsalgorithmen auf dem Modell arbeiten und ausführbare Artefakte generieren können, solange noch erkannte Fehler in der Integrationslösung enthalten sind. Die kontinuierliche Überprüfung dieser Kriterien während der Modellierung führt dazu, dass nur syntaktisch korrekte Lösungen an Generierungsalgorithmen übergeben werden. Somit wird die Erzeugung ausführbarer Artefakte aus bereits offensichtlich falschen Integrationslösungen zur Modellierungszeit verhindert.

8.5. Grafische Benutzungsoberfläche

Die grafische Benutzungsoberfläche von GENIUS basiert auf der von Eclipse verwendeten Bibliothek zur Erstellung von grafischen Oberflächen *SWT (Standard Widget Toolkit)* [Ecld]. Die Dialoge der einzelnen Muster wurden mit GEF entwickelt. Die Oberfläche von Eclipse bietet drei unterschiedliche Konzepte, um die Arbeit mit Quelltext oder anderen Ressourcen zu vereinfachen: Sichten (*Views*), Editoren und Perspektiven. GENIUS verwendet davon die ersten zwei. Abbildung 8.3 stellt einen Screenshot von GENIUS dar. Die Hauptarbeitsfläche (1) ist ein graphischer Editor, der es erlaubt, Integrationsmuster von einer Palette (2) zu ziehen und darauf abzulegen. In der Palette sind alle verfügbaren Integrationsmuster enthalten. Neben dem graphischen Editor befindet sich die Ressourcenansicht aller Dateien, die in einem Projekt enthalten sind und zu einer oder mehreren Integrationslösungen gehören (3). Hier können zum

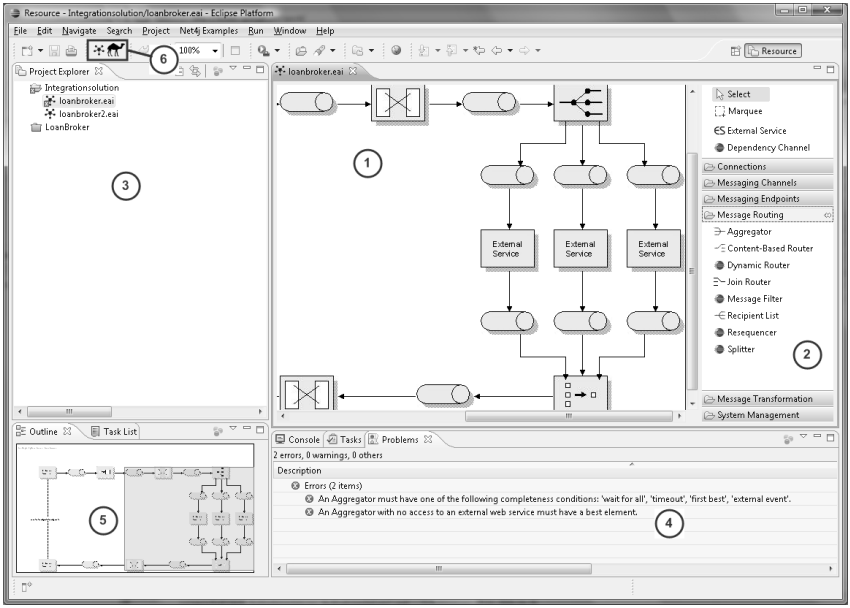


Abbildung 8.3.: Screenshot von GENIUS

Beispiel auch WSDL-Dateien oder XSLT Stylesheets enthalten sein, falls diese von einer Integrationslösung benötigt werden.

Im unteren Bereich des Hauptfensters von GENIUS findet sich eine Sicht, auf der die Fehler und Warnungen, die in einer modellierten Integrationslösung enthalten sind, dargestellt werden (4). Diese Meldungen entspringen der Überprüfung der modellierten Lösung nach den unter Abschnitt 8.4 beschriebenen Konzepten. Der letzte Bereich (5) ist ein Grundriss der Integrationslösung (*Outline-View*). Mit Hilfe dieser Ansicht können komplexe Integrationslösungen immer im Überblick betrachtet werden. Das grau unterlegte Rechteck veranschaulicht den Bereich, der momentan im Editor angezeigt wird. In der Symbolleiste oberhalb des Editors befindet sich eine Schaltfläche, über die man die Generierung einer ausführbaren Integrationslösung starten kann (6). Jeder vorhandene Algorithmus (in diesem Fall *EAI2BPEL* und *EAI2Camel*) ist durch die zugeordneten Schaltflächen repräsentiert.

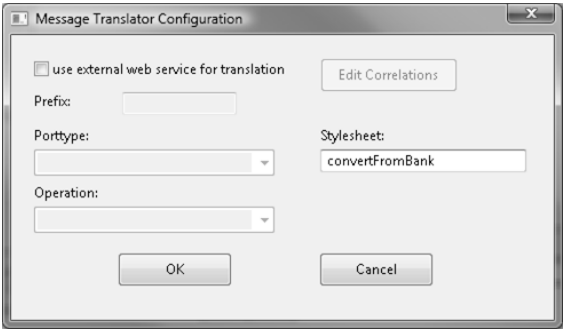


Abbildung 8.4.: Parametrisierungsdialog eines Message Translators

Eine Integrationslösung kann zunächst ohne Parametrisierung modelliert werden. Dies ermöglicht dem Benutzer schnell, den Überblick zu erlangen, wie die endgültige Integrationslösung aussehen soll. Einen Generierungsalgorithmus kann man allerdings nicht aufrufen, da zunächst die Lösung analog zu den Regeln in Abschnitt 8.4 parametrisiert werden muss. Aus diesem Grund bietet jedes Integrationsmuster einen eigenen Parametrisierungsdialog. In Abbildung 8.4 ist der Dialog eines Message Translator Musters dargestellt. Es ist zu erkennen, dass bestimmte Parameter nicht gesetzt werden können (die Felder „Port Type“ und „Operation“ sind nicht aktiviert). Der Grund dafür ist, dass das Muster nicht mit einem externen Dienst arbeitet, sondern ein Stylesheet verwendet (`convertFromBank`).

Ein weiterer Dialog ist in Abbildung 8.5 dargestellt. Hier sind die Parameter des Aggregator Musters zu sehen. Es sind die verschiedenen Eigenschaften des Musters und die entsprechenden Parameter zu erkennen. Zunächst kann die Vollständigkeitsbedingung angegeben werden. Im Anschluss kann bestimmt werden, wie die Aggregation vorgenommen werden kann. Auch hier ist zu sehen, dass bestimmte Parameter nicht ausgefüllt werden können, da sie in der aktuellen Konfiguration nicht benötigt werden. Die zuletzt aufgeführten Parameter sind notwendig, um eine Korrelation der Ausgänge des Musters (jeder Ausgang wird durch eine Nummer identifiziert) zu den nachfolgenden Kanal Mustern herzustellen.

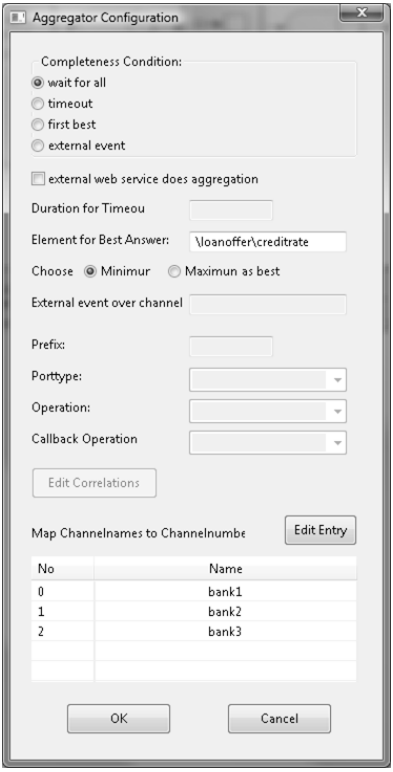


Abbildung 8.5.: Parametrisierungsdialog eines Aggregators

Ein wichtiger Bereich während der Modellierung sind Hinweise auf Fehler und Warnungen. Dies erfolgt über eine separate Sicht. In Abbildung 8.6 ist ein solcher Hinweis dargestellt. Der Benutzer wird darüber in Kenntnis gesetzt, dass ein Muster nicht korrekt parametrisiert ist (in diesem Fall ein Aggregator). Diese Fehlermeldung wird darüber hinaus auch durch ein kleines rotes Kreuz dargestellt, das an die die Integrationslösung repräsentierende Datei in der Ressourcenübersicht angefügt wird. Solange die Datei mit einem solchen Kreuz markiert ist, erkennt der Benutzer, dass die Integrationslösung nicht in ein ausführbares System überführt werden kann. Momentan werden fehlerhafte

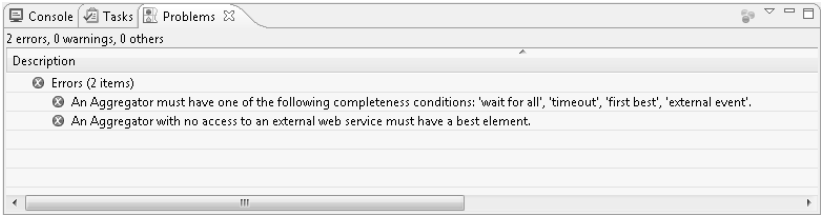


Abbildung 8.6.: Fehler und Warnmeldungen in GENIUS

Muster in dem grafischen Editor noch nicht markiert. Dies würde die Suche der Fehler nochmals erleichtern. Diese Erweiterung ist jedoch für die zukünftigen Versionen angedacht.

8.6. Generierungsalgorithmen

Wie in Abschnitt 6.2 bereits beschrieben wurde, haben alle Algorithmen den gleichen Ansatz, um ausführbare Systeme aus den parametrisierten Integrationsmustern zu erzeugen. Ein Algorithmus klinkt sich in den GENIUS Kern über eine so genannte *Action* ein. Er erhält dadurch Zugriff auf das Datenmodell von GENIUS. D.h. ein Algorithmus kann über die Muster einer *Messaging System* Instanz (eine Integrationslösung) iterieren und entsprechend der gesetzten Parameter Quelltext erzeugen. Im ersten Schritt werden die einzelnen Muster separat betrachtet und Quelltextteile für dieses Muster erzeugt. Im nächsten Schritt werden die einzelnen Muster entsprechend der Verbindungen in der Integrationslösung zusammengefügt. In manchen Algorithmen ist es möglich, dass nicht nur ein ausführbares System erzeugt wird (zum Beispiel bei BPEL wird die Integrationslösung als ein einzelnes BPEL Prozessmodell realisiert), sondern mehrere unabhängige Anwendungen, die über eine ebenfalls durch den Algorithmus erzeugte Integrationsbeschreibung, miteinander verbunden werden (vergleiche Erzeugung für EaaS in Abschnitt 6.5). Somit erzeugt der Algorithmus alle für die Ausführung der Integrationslösung relevanten Daten. Dazu gehören auch die Deployment Informationen und die Erzeugung des entsprechenden Formats, mit dem die Artefakte auf der Integrationsinfrastruktur

ausgerollt und ausgeführt werden können.

In GENIUS sind aktuell zwei Algorithmen enthalten, die ausführbare Artefakte für zwei unterschiedliche Zielintegrationsinfrastrukturen erzeugen. Im Folgenden wird zunächst der Algorithmus zur Generierung von BPEL präsentiert. Im Anschluss daran wird der Algorithmus erläutert, der Code für Apache Camel als Zielinfrastruktur produziert.

8.6.1. EAI2BPEL

In Abschnitt 6.2.1 wurden Beispiele gezeigt, die erläutern, welche unterschiedlichen BPEL Artefakte für einzelne Integrationsmuster abhängig von den gesetzten Parametern erzeugt werden. In diesem Abschnitt wird der dahinterstehende Algorithmus erläutert und gezeigt, wie er aus dem vorhandenen Datenmodell die entsprechenden BPEL Prozesse und zugehörigen Dateien generiert. Zur Erzeugung von BPEL wurde ein eigenständiges Framework entwickelt, mit dessen Hilfe ein BPEL Prozess durch Java Klassen erstellt werden kann. Dazu wurde das BPEL Schema in ein separates BPEL Modell überführt. Die Hauptklassen des Generierungsalgorithmus sind die Klassen *BPELWriter* und *WSDLWriter*. Die Generierung eines BPEL Prozesses unterteilt sich daher prinzipiell in zwei unterschiedliche Schritte, einen zur Erzeugung des BPEL Prozesses und zugehöriger WSDL Datei und einen abschließenden Schritt zur Erzeugung der zusätzlichen Deployment Informationen und des entsprechenden Formats zur Installation. In Abbildung 8.7 ist der Generierungsalgorithmus schematisch dargestellt.

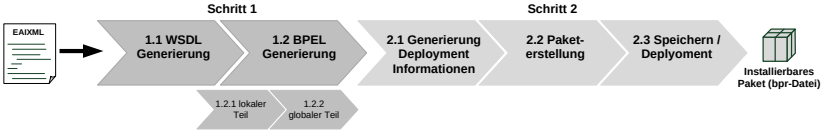


Abbildung 8.7.: Schematischer Aufbau des Generierungsalgorithmus für BPEL

Im ersten Schritt wird zunächst die WSDL Datei generiert, die die Web Service Schnittstelle des BPEL Prozesses darstellt. Die Klasse *WSDLWriter* erzeugt anhand der Informationen der Schnittstelle der Integrationslösung die

entsprechenden WSDL Informationen. So setzt sie zum Beispiel die Nachrichtentypen der eingehenden und ausgehenden Nachrichten (`message type schema`, `input message`, `output message`, `fault message`). Darüber hinaus werden die Operationen des Web Services erstellt und die *Partner Link Typen* entsprechend generiert. Nach diesem Unterschritt ist somit eine vollständige WSDL Datei entstanden, die die Schnittstellen des Prozesses beschreibt.

Im Anschluss erfolgt die Erzeugung des eigentlichen BPEL Prozesses durch die Klasse *BPELWriter*. Dieser Schritt unterteilt sich wiederum in zwei verschiedene Unterschritte. Im ersten werden die BPEL Artefakte der einzelnen atomaren Muster generiert und im zweiten die einzelnen Artefakte zu einem Prozess zusammengeführt. Die Verbindung der einzelnen Artefakte erfolgt durch BPEL Links. Die eingehenden und ausgehenden Nachrichten von Integrationsmustern werden im BPEL Prozess als Variablen repräsentiert. Zur Korrelation eingehender Nachrichten zur korrekten BPEL Prozessinstanz wird der Korrelationsmechnismus von BPEL verwendet. Entsprechende Definitionen werden während der Generierung dem BPEL Prozess hinzugefügt. Die Struktur der Nachrichten wird durch WSDL oder XSD Dateien definiert, die während der Modellierung in die Integrationslösung importiert werden. Die Umsetzung von parametrisierten Integrationsmustern in BPEL wurde anhand von einzelnen Beispielen ausführlich in Abschnitt 6.2.1 beschrieben. Am Ende dieses zweiten Schrittes ist also ein kompletter BPEL Prozess mitsamt der zugehörigen WSDL Datei entstanden. Diesen beiden Dateien müssen nun noch alle weiteren Dateien, unter anderem die Deployment Informationen, hinzugefügt werden, damit die Integrationslösung auf dem entsprechenden BPEL Server ausgeführt werden kann.

Das Deployment beziehungsweise die Vorbereitung für das Deployment erfolgt somit im letzten Schritt des Algorithmus. Prinzipiell können hierbei verschiedene BPEL Systeme als Zielplattform dienen. Im aktuellen GENIUS Prototyp wird ActiveBPEL [Act] der Firma Active Endpoints als Plattform unterstützt. Weitere Plattformen wurden noch nicht in den Algorithmus integriert, im manuellen Versuch wurde allerdings ein generiertes Integrationssystem auch auf der Apache ODE Engine [Apac] erfolgreich ausgeführt. ActiveBPEL erwartet zur Installation eines BPEL Prozesses ein bestimmtes Format. Dies ist

ein Zip-Archiv mit einer vorgegebenen Ordnerstruktur und benötigten Dateien. Aus diesem Grund wird zunächst eine Deployment Descriptor Datei erzeugt. Die Informationen, die hier benötigt werden (zum Beispiel Ports und Server Adresse), werden während des Generierungsvorgangs vom Algorithmus mittels Dialogfenster vom Benutzer erfragt. Alle weiteren Dateien wurden bereits vom Algorithmus erzeugt und werden nun entsprechend der Spezifikation von ActiveBPEL in einem Archiv zusammengefügt. Dieses so erzeugte Archiv kann man nun direkt auf dem Server ablegen. ActiveBPEL installiert im Anschluss daran den neuen BPEL Prozess automatisch auf dem Server. Nun steht die Integrationslösung zur Ausführung zur Verfügung.

8.6.2. EAI2Camel

Die zweite unterstützte Zielplattform ist Apache Camel (vergleiche Abschnitt 6.4). Auch für diesen Algorithmus wurde ein entsprechendes Rahmenwerk implementiert, das mit Hilfe eines Datenmodells ermöglicht, Quelltext für die Zielplattform zu erzeugen. Der *EAI2Camel* Generierungsalgorithmus iteriert wie der *EAI2BPEL* Algorithmus ebenfalls über die Muster einer Instanz einer *Messaging System* Instanz und erzeugt ein Java Projekt innerhalb der Eclipse Umgebung. Das generierte Java Projekt ist eine eigenständige Anwendung, welche einen Apache Camel Kontext aufbaut, der die Integrationslösung verkörpert. Die Anwendung interagiert mit der Umgebung, indem sie einen Web Service nach außen anbietet und unter Umständen Web Services aufruft, wenn dies in der Integrationslösung konfiguriert wurde.

Das Rahmenwerk repräsentiert die parametrisierbaren Integrationsmuster als eigenständige Klassen, die auf der von Apache entwickelten Camel Bibliothek aufbauen. Jedes Muster, das in der Bibliothek noch nicht oder nicht vollständig vorhanden ist, muss dazu einen sogenannten *Processor* implementieren, der mit den unterschiedlichen Parametern der Muster umgehen kann. Zum Beispiel wird ein Processor benötigt, um eine Nachricht mit Hilfe von XSLT zu transformieren. Zur Überführung einer Integrationslösung in die entsprechende Java Anwendung müssen diese Klassen nur noch entsprechend der Parametrisierung instanziiert werden, durch Apache Camel Routen miteinander verbunden und

Abbildung 8.8.: EAI2Camel Dialog für Port und Service eines Web Services

dem Java Projekt hinzugefügt werden.

Die Generierung wird auch bei diesem Algorithmus durch eine Hauptklasse getrieben: die Klasse *CamelWriter*. Diese Klasse erzeugt zunächst für die einzelnen Muster entsprechende Instanzen der Klassen des Frameworks. Zu diesem Zweck wurde eine entsprechende Klasse entwickelt (*CamelFilterPatternGenerator*), die die jeweiligen Klassen instanziiert. Im Anschluss wird eine Klasse *MessagingSystem* erstellt, die die Integrationslösung letztlich repräsentiert. Darin enthalten sind die Routen, die den Nachrichtenfluss innerhalb der Integrationslösung darstellen.

Da die Integrationslösung noch keine Daten über den konkreten Endpunkt eines Web Services enthält (vergleiche Abschnitt 8.6.1), müssen diese Informationen während der Erzeugung der Java Klassen ebenfalls noch eingegeben werden. Dies erfolgt über einen Dialog für jedes Muster, das einen Web Service verwendet. In diesem Dialog muss der Port beziehungsweise der Endpunkt jedes Porttypen eingegeben werden. In Abbildung 8.8 ist ein entsprechender Dialog zur Parametrisierung eines External Service Musters abgebildet.

Damit die Integrationslösung ausgeführt werden kann, wird im letzten Schritt im aktuellen Eclipse Arbeitsbereich (*workspace*) ein neues Java Projekt angelegt, in das die zuvor erstellten Klassen in der entsprechenden Ordnerstruktur eingefügt werden. Darüber hinaus werden alle benötigten Bibliotheken dem Projekt hinzugefügt. So ist zum Beispiel die Apache Camel Bibliothek wie auch die Apache ActiveMQ Bibliothek nötig, da die Implementierung des *EAI2Camel* Algorithmus diese Bibliotheken verwendet.

Eine Alternative zur Erzeugung eines Java Projektes ist die Erstellung eines ausführbaren Java Archivs. In diesem Fall ruft der Generierungsalgorithmus einen Java Compiler auf, der aus den übergebenen Java Klassen ein Java Archiv erstellt (jar-Datei). Diese Möglichkeit wurde im aktuellen Prototyp noch nicht realisiert.

8.7. Darvien in GENIUS

Die IT Abteilung von MehrVomGeld setzt GENIUS als Modellierungswerkzeug für Darvien ein. Ein Systemarchitekt modelliert dazu das bereits vorhandene Darvien EMod in EAIXML Serialisierung nach. In Abbildung 8.9 ist Darvien mit Hilfe von GENIUS bereits fertig modelliert. Die Kreise sowie die Beschriftung sind nicht Teil des Modellierungswerkzeugs, sondern wurden zur besseren Übersicht hinzugefügt. Es besteht aus insgesamt fünf unterschiedlichen Teilen. Der Ein- und Ausgang – die Annahme von Kreditanfragen und die Rückgabe der Angebote – werden durch das External Service Muster modelliert. Im nächsten Schritt wird die eingegangene Nachricht mit Hilfe eines Content Enricher Musters um Informationen über die Kreditwürdigkeit angereichert. Das Muster verwendet dazu einen externen Dienst; dieser ist in der Abbildung nicht ersichtlich, da das Muster Bestandteil des Content Enricher Musters ist. Anhand der Informationen innerhalb der Kreditanfrage (Kreditwürdigkeit und Kreditsumme, siehe Tabelle 4.27) werden Kopien der Anfragenachricht durch ein Recipient List Muster an verschiedene Dienste der Banksparten weitergeleitet. Die Dienste werden durch External Service Muster aufgerufen. Die jeweiligen Antworten der Banken werden durch einen Aggregator zusammengefasst und ausgewertet. Die resultierende Nachricht – das beste Angebot – wird als Antwort zurückgesendet.

Die Schnittstellen der externen Dienste sind durch WSDL Dateien beschrieben. Darin sind auch die Nachrichten definiert. Zusätzlich können noch XML Schema Dateien existieren, die die Nachrichtenformate beinhalten (soweit diese nicht in der WSDL bereits definiert wurden). Diese WSDL und XSD Dateien können in das Messaging System importiert werden. Der Systemarchitekt kann daraufhin in den entsprechenden Dialogen auf die Definitionen der

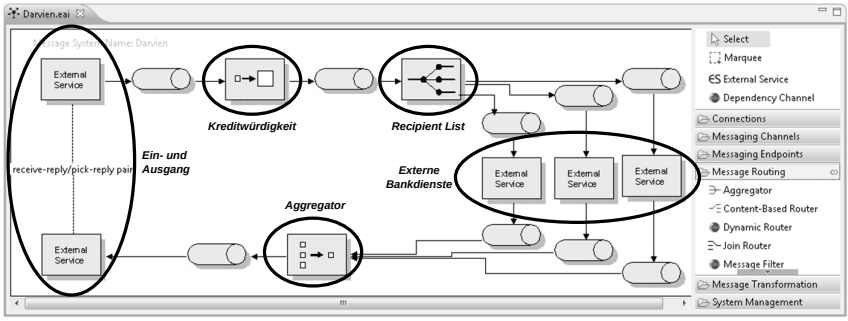


Abbildung 8.9.: Darvien Szenario in GENIUS

WSDL Datei (Operationen, Nachrichten) zugreifen, ohne diese Daten manuell eingeben zu müssen. Es wird an dieser Stelle darauf verzichtet, die einzelnen Konfigurationsdialoge darzustellen. Einige sind im Verlauf dieses Kapitels bereits beschrieben worden. Die Parameter der einzelnen Muster wurden in Tabelle 4.27 festgelegt.

Darvien wird nun über den in GENIUS integrierten Algorithmus nach BPEL transformiert. Der resultierende BPEL Prozess ist unter Abschnitt 6.6 beschrieben und im Anhang dargestellt. Die zugehörige WSDL Datei ist ebenfalls im Anhang aufgeführt. Während der Erstellung der BPEL und WSDL Datei müssen unter Umständen noch weitere Informationen hinzugefügt werden, die dem Systemarchitekten zur Modellierungszeit nicht vorgelegen haben (zum Beispiel Port und Service der externen Dienste). Fehlen die Informationen, wird der Benutzer über entsprechende Dialoge zur Eingabe aufgefordert. Falls diese Informationen bereits in der WSDL enthalten sind, wird dieser Schritt übersprungen. Der Generierungsalgorithmus erzeugt auch den benötigten Deployment Descriptor für ActiveBPEL (siehe Listing 8.1).

```
<?xml version="1.0" encoding="UTF−8"?>
<process xmlns="http://schemas.active−
    endpoints.com/pdd/2005/09/pdd.xsd"
    xmlns:bpelns="http://www.mehrvomgeld.de/darvien"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
    location="bpel/darvien/darvien.bpel" name="bpelns:darvien">
```

```
  <partnerLinks>
    <partnerLink name="bplens:DarvienProcessPLT">
      <myRole allowedRoles="" binding="RPC−LIT"
              service="bpelns:darvien"/>
    </partnerLink>
  </partnerLinks>

  <wsdlReferences>
    <wsdl location="project:/darvien/darvien.wsdl"
          namespace="http://www.mehrvomgeld.de/darvien"/>
    <wsdl location="project:/darvien/WSDL/creditBureau.wsdl"
          namespace="http://www.kreditkontrolle.de/kwk"/>
    <wsdl location="project:/darvien/WSDL/bank1.wsdl"
          namespace="http://www.mehrvomgeld.de/bank1"/>
    <wsdl location="project:/darvien/WSDL/bank2.wsdl"
          namespace="http://www.mehrvomgeld.de/bank2"/>
    <wsdl location="project:/darvien/WSDL/bank3.wsdl"
          namespace="http://www.mehrvomgeld.de/bank3"/>
  </wsdlReferences>
</process>
```

Listing 8.1: Deployment Descriptor ActiveBPEL Engine

Im Anschluss erzeugt der Algorithmus ein Paket (eine bpr Datei), in dem alle benötigen Dateien enthalten sind: Darvien BPEL Prozess, Darvien Prozess WSDL Datei, WSDL Datei des Kreditinstituts, WSDL Dateien aller Banken und der Deployment Descriptor. Dieses Paket legt er im Wurzelverzeichnis des Projektes ab. Daraufhin wird dieses Paket manuell auf dem ActiveBPEL Server installiert. Im Anschluss steht die ausführbare Integrationslösung als Web Service zur Verfügung.

8.8. Mögliche GENIUS Erweiterungen

GENIUS ist momentan ein Prototyp, d.h. das Werkzeug kann an einigen Stellen noch verbessert und ausgebaut werden. Verbesserung an der Benutzungsoberfläche, Verfeinerung von Dialogen oder ähnliches wird in diesem Abschnitt nicht besprochen. Vielmehr gibt es einige wenige konzeptionelle Verbesserungen,

die noch eingeführt werden können.

Bei der Erstellung von Integrationsmustern fällt auf, dass immer wieder die gleichen Arten von Musterkombinationen gewählt werden. Diese Kombinationen sind häufig bereits durch zusammengesetzte Muster repräsentiert. Allerdings haben auch diese Muster immer noch eine Varianz. Zum Beispiel ist eine häufige Ausprägung des Composed Message Processor Musters die Verwendung einer Recipient List als Verteilungsmuster, gefolgt von einem Message Translator, einem externen Dienstaufruf und wiederum einem Message Translator Muster als Abarbeitungsschritt. Da eine solche Ausprägung häufig verwendet wird, ist dies auch als eine Art Schablone anzusehen. Eine solche Schablone kann ausgewählt werden, wenn ein Composed Message Processor Muster der Integrationslösung hinzugefügt werden kann. Solche „Best Practices“ können ebenfalls Bestandteil der Palette der Integrationsmuster sein. Dieses Vorgehen kann noch weiter ausgebaut werden, wenn man erkennt, dass ein und dasselbe Muster immer mit den gleichen Parameterwerten eingesetzt wird. Auch in einem solchen Fall kann dem Modellierer Arbeit erspart werden, indem dieses vorparametrisierte Muster in der Palette angeboten wird.

Der Ansatz der häufig verwendeten Muster lässt sich noch weiter ausbauen. Ein Modellierer selbst sollte die Möglichkeit haben, seine eigenen häufig verwendeten Muster samt Parametrisierung abzuspeichern und in weiteren Integrationslösungen wiederverwenden zu können. Dies ist zum Beispiel dann realisierbar, wenn man die parametrisierbaren Integrationsmuster, die Schablonen und sogar parametrisierte Muster in einem Repository ablegt. Die Palette der Integrationsmuster könnte direkt mit dem Repository verknüpft werden und somit Muster aus dem Repository über die Palette in eine Integrationslösung geladen werden. Mit einem solchen Repository würden zahlreiche Vorteile einhergehen: zum Beispiel Versionierung von Mustern, Suche von Mustern oder Schablonen und verbesserte Austauschmöglichkeiten zwischen Integrationslösungen und den Modellierern selbst. Da diese Betrachtung (insbesondere die Analyse von Repositories) nicht Teil dieser Arbeit ist, wird die Diskussion hier nicht aufgegriffen. Wichtig ist, dass GENIUS diese Erweiterungsmöglichkeiten bietet. Durch die Existenz eines klar spezifizierten Metamodells ist die Grundlage gegeben, ein entsprechendes Repository zur Verfügung stellen zu

können.

Neben den aktuellen Generierungsalgorithmen sollten weitere Zielplattformen unterstützt werden. Ein Beispiel wäre die Unterstützung des EaaS Ansatzes, so dass GENIUS dahingehend erweitert wird, dass Integrationslösungen in der Cloud ausgeführt werden können. Außerdem kann man die GENIUS Plattform einsetzen, um die Modellierung von Weiterleitungslogiken für SOAP Nachrichten zu ermöglichen. Somit hätte der vorgestellte SBR Ansatz auch eine Werkzeugunterstützung.

KAPITEL 9

ZUSAMMENFASSUNG UND AUSBLICK

Dieses Kapitel fasst die vorgestellten Arbeiten zusammen und bietet einen Ausblick auf zukünftige Arbeiten. Die Zusammenfassung gibt einen Überblick über die Beiträge der Dissertation, zeigt die Bereiche auf in denen parametrisierbare Integrationsmuster eingesetzt werden und beschreibt anhand des Beispielszenarios Darvien wie die einzelnen Kapitel zusammen hängen. Das Kapitel schließt mit dem Ausblick auf Arbeiten, die auf den vorgestellten Beiträgen aufbauen und die die Beiträge somit erweitern können.

9.1. Zusammenfassung

In dieser Dissertation wurde eine Modell-getriebene Methode zur automatisierten Erstellung von ausführbaren Integrationslösungen vorgestellt. Die Methode baut dabei auf einem Integrationsprozess auf, der jeden einzelnen durchzuführenden Schritt beschreibt. Integrationsmuster zur Modellierung von Integrationslösungen werden dabei als plattformunabhängige Modellie-

rungselemente eingesetzt. Diese Muster lagen vor Beginn der Dissertation nur in Form von textuellen Beschreibungen mit visuellen Repräsentationen vor. Sie wurden daher in ein formales Modell überführt, um die Modell-getrieben Methode anwenden zu können. Aus diesem Grund wurde ein entsprechendes Metamodell erstellt, welches parametrisierbare Integrationsmuster und die Verbindungen untereinander definiert. Die Parameter eines Integrationsmusters zeigen dabei die Variabilität jedes einzelnen Musters auf. Mit Hilfe des Metamodells ist es möglich, eine Integrationslösung genau zu spezifizieren. Eine solche Spezifikation dient einem Generierungsalgorithmus als Eingabe, um automatisiert ausführbare Integrationsmuster zu erzeugen.

Um die Methode generell einsetzbar zu machen, zum Beispiel zur Unterstützung durch eine Vielzahl an Werkzeugen, wurde ein Gerüst entworfen, das alle Komponenten definiert und beinhaltet, um Integrationslösungen automatisiert ausführen zu können. EMod beschreibt alle benötigten Bestandteile, um eine Integrationslösung mit Hilfe von parametrisierbaren Integrationsmustern zu modellieren und in ausführbare Systeme zu überführen. Basiert eine Anwendung auf dieser Spezifikation, kann das Produkt (eine Integrationslösung) durch verschiedene Werkzeuge erstellt und auf unterschiedlichen Integrationsinfrastrukturen ausgeführt werden. EMod enthält folgende Elemente: das Metamodell, parametrisierbare Integrationsmuster (PEP), eine Serialisierung dieser PEP, ein EMod Modellierungswerkzeug und eine Laufzeitumgebung für PEP.

Basierend auf der EMod Spezifikation wurde ein genereller Algorithmus entwickelt, auf dem alle Generierungsalgorithmen für die unterschiedlichen Zielplattformen aufsetzen. Er unterteilt sich prinzipiell in zwei Phasen. Im ersten Schritt werden die einzelnen Muster übersetzt, und im zweiten Schritt werden diese übersetzten Muster miteinander verbunden. Abgeschlossen wird ein solcher Generierungsalgorithmus mit der Erzeugung der installierbaren Pakete oder Dateien. Aufbauend auf diesem Vorgehen wurden zwei verschiedene spezielle Algorithmen entwickelt. Der erste Algorithmus generiert ausführbare Artefakte, um eine Integrationslösung als BPEL Prozess auszuführen. Der zweite Algorithmus erzeugt Dateien, die unter Apache Camel ausgeführt werden können.

Neben diesen beiden Technologien wurde auch ein Ansatz verfolgt, der Integrationslösungen im „as a Service“-Modus nutzt. Dazu wurde das existierende PEP Modell um Informationen für Mandantenfähigkeit erweitert. Dieses erweiterte Modell wird von einem entsprechenden Generierungsalgorithmus verwendet. Der Algorithmus nutzt dazu Provisionierungsdienste der Cloud Infrastruktur und installiert automatisiert alle benötigten Artefakte auf dieser Infrastruktur. Auch in diesem Fall dient ein BPEL Prozess dazu, die verschiedenen Anwendungen, die in der Cloud vorhanden sind, entsprechend der modellierten Integrationslösung miteinander zu integrieren.

Neben diesen drei Zielplattformen wurde die Abbildung auf eine weitere Plattform untersucht. Die *Service Component Architecture (SCA)* erlaubt zum Beispiel die Strukturierung von Geschäftsanwendungen in kleinen Komponenten [Ope07]. Allerdings treten auch in SCA gewisse Integrationsprobleme auf [SZC+07]. Diesen Problemen kann ebenfalls durch den Einsatz von parametrisierten Integrationsmustern begegnet werden. Dazu wurde ein neuer Implementierungstyp in SCA definiert: *Executable EAI Pattern (EEP)*. Dieser neue Ansatz wurde mit dem Begriff *PICS (Pattern-based Integration of Components)* belegt [SML09]. Wie der Name suggeriert, besteht eine solche Komponente aus einer Menge an parametrisierten Integrationsmustern, die die Integrationslogik repräsentieren (siehe Abbildung 9.1). Die Implementierungslogik eines solchen EEP Typs ist somit als EMod Datei definiert. Da EMod als Implementierungstyp verwendet wurde, ist die Ausführung dieser Muster von der Spezifikation getrennt. Denn durch Verwendung von EMod wird die eigentliche Laufzeitumgebung (*Pattern Runtime*) erst während des Deployments der Komponente bestimmt. Außerdem ist der Klient durch die explizite Modellierung einer separaten Mediationskomponente an den Dienst sehr viel weniger gekoppelt.

In Abbildung 9.2 sind alle in dieser Dissertation vorgestellten Zielplattformen für ausführbare Integrationsmuster zusammengefasst. Basis sind Integrationslösungen, die als EAIXML serialisiert sind. Auf dieser Datei können verschiedene Algorithmen arbeiten, um ausführbare Artefakte generieren zu können. Die gestrichelten Linien zum Transformationsalgorithmus für einen ESB, der in dieser Arbeit nicht diskutiert wurde, sowie die Verbindung zu PICS deuten an, dass diese Konzepte momentan noch nicht praktisch umgesetzt wurden. Es

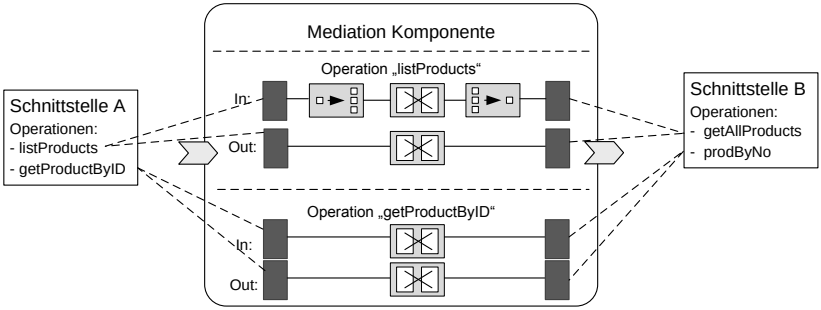


Abbildung 9.1.: SCA Komponente und Integrationsmuster

sind theoretische Überlegungen, um EMod in weiteren Bereichen nutzen zu können [Mie08][SML09].

Zur Erstellung einer Integrationslösung wurde im Rahmen der Dissertation ein eigenständiges Werkzeug entwickelt. *GENIUS (Generating Enterprise Integration Executable Scenarios)* bietet die Möglichkeit, eine Integrationslösung mit Hilfe von PEP graphisch zu modellieren. Es wurden zwei Generierungsalgorithmen in das auf Eclipse basierende Werkzeug eingebaut (*EAI2BPEL* und *EAI2Camel*). Darüber hinaus bietet GENIUS einige Konzepte, die den Modellierer während der Entwurfsphase unterstützen und so die Benutzerfreundlichkeit erhöhen.

Neben diesen Hauptbeiträgen sind noch einige weitere Beiträge während der Arbeit zur Dissertation entstanden. So wurde der Ansatz der Pipes-and-Filters Architektur mit dem Ansatz der Instanz-basierten Verarbeitung von Nachrichten mit Hilfe eines Geschäftsprozesses verglichen. Die Analyse beider Ansätze zeigte, dass beide prinzipiell als Infrastruktur für ausführbare Integrationslösungen dienen können. Allerdings besitzt der Instanz-basierte Ansatz deutliche Vorteile im Bereich des produktiven Einsatzes (wie etwa Überwachung und Überprüfung) bei gleichzeitig vergleichbarem Laufzeitverhalten.

Integrationsmuster können nicht nur dazu verwendet werden, ganze Anwendungslandschaften miteinander zu verbinden. Mit PICS wurde beschrieben, wie diese Muster auch zur Integration von einzelnen Komponenten innerhalb

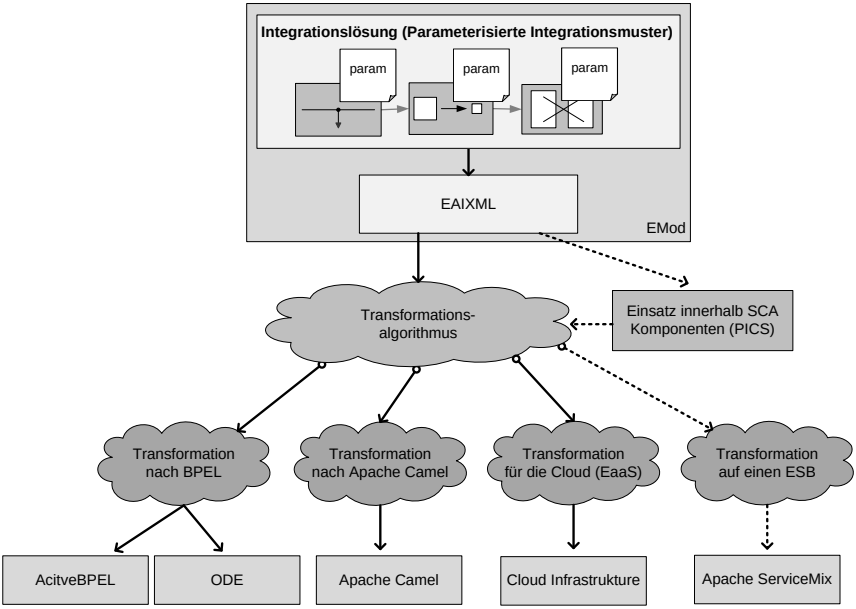


Abbildung 9.2.: Überblick Transformationsmöglichkeiten

einer SCA eingesetzt werden können. Darüber hinaus müssen Nachrichten auch auf der Transportebene (SOAP) weitergeleitet und bearbeitet werden, bevor sie beim eigentlichen Empfänger ankommen. Auch zu diesem Zweck, also zur Beschreibung einer Weiterleitungslogik auf Nachrichtenebene, können Integrationsmuster verwendet werden. Darüber hinaus können diese Muster wiederum auf BPEL Prozesse abgebildet werden. In einer SOA, basierend auf Web Services, ermöglicht dies die Einführung eines neuen Mechanismus: SOAP BPEL Routing (SBR). Mit Hilfe dieses Ansatzes ist es möglich, SOAP Nachrichten zwischen einzelnen Knoten anhand eines Prozesses weiterzuleiten und die Reihenfolge der Bearbeitung an einem Knoten durch zusätzliche Dienste zu spezifizieren.

9.1.1. Darvien im Kontext der Arbeit

Um ein abschließendes Bild über die Ergebnisse der Arbeit zu erhalten, wird ein weiteres Mal das Beispielszenario Darvien verwendet. In Abbildung 9.3 werden der Aufbau der Dissertation anhand von Darvien geschildert und die jeweiligen Kapitel aufgezeigt. Die IT Abteilung von MehrVomGeld modellierte zunächst ihre Anforderungen mit Hilfe von Integrationsmustern (Kapitel 1). Da diese allerdings nicht aussagekräftig genug waren, wurden diese Muster formal definiert (Kapitel 3 und Kapitel4). Es entstand ein formales Modell mit dessen Hilfe man Integrationslösungen spezifizieren konnte. Gleichzeitig zu dieser Entwicklung wurde ein Prozess entwickelt, der die Methode unterstützt (Kapitel 3). Die IT Abteilung erkannte, dass eine Modell-getriebene Methode angewendet werden konnte, um eine solche Integrationslösung in ausführbare Artefakte zu überführen. Daraufhin wurde ein Algorithmus eingesetzt, der dieses Modell auf einen BPEL Prozess als Integrationslogik abbildet (Kapitel 6). Allerdings war dem Abteilungsleiter auch klar, dass er seine Systemintegratoren nicht XML Dateien mit einem XML Editor erstellen lassen konnte. Aus diesem Grund wurde ein entsprechendes Werkzeug (GENIUS) verwendet, mit dem Integrationsmuster visuell modelliert werden können (Kapitel 8). Am Ende besitzt MehrVomGeld eine umfassende und durchgängige Methode samt Prozess, um Integrationslösungen modellieren und automatisiert ausführen zu können. Aufgrund der Erweiterbarkeit des Ansatzes kann MehrVomGeld auf zukünftige Änderungen der Anforderungen schnell und einfach reagieren. Außerdem ist es nun möglich, neue Integrationsinfrastrukturen zu unterstützen, falls neue Technologien eingesetzt werden sollen.

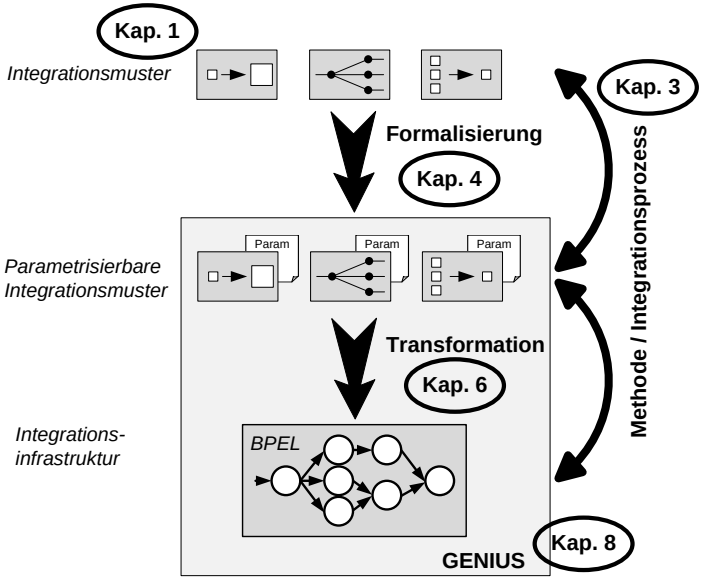


Abbildung 9.3.: Beiträge der Arbeit anhand des Beispielszenarios Darvien

9.2. Ausblick

In diesem Abschnitt werden mögliche zukünftige Arbeiten im Bereich der parametrisierbaren Integrationsmuster beschrieben. Diese Arbeiten teilen sich in konzeptionelle Anpassungen der Methode sowie Verbesserungen und Erweiterungen der Werkzeugkette. Abgeschlossen wird der Abschnitt mit einer Betrachtung der verbesserten Modellierung von Integrationslösungen durch Entscheidungsbäume.

Die vorgestellte Methode erlaubt es, aus Integrationsmustern ausführbare Artefakte zu erzeugen. Allerdings wird, wie in MDD basierten Verfahren üblich, keine Aussage über die entgegengesetzte Richtung getroffen: zum Beispiel wie kann ein Integrationsmuster innerhalb eines BPEL Prozesses identifiziert werden. Diese Richtung ist insbesondere dann von Bedeutung, wenn das Monitoring von laufenden Integrationslösungen oder die Überprüfung von bereits

abgelaufenen Lösungen betrachtet wird (vergleiche Abbildung 9.4). Der generierte Quelltext ist teilweise sehr komplex, daher nicht einfach zu verstehen und die Rückübersetzung auf die Muster nur schwer möglich. Am Beispiel des Musters Aggregator soll dies verdeutlicht werden: Wenn ein Aggregator mit *first best* Vollständigkeitsbedingung modelliert wird, führt das in BPEL dazu, dass beim Eintreffen der ersten Nachricht innerhalb des Scope Konstrukts des Aggregator Musters ein Fehler erzeugt wird, der abgefangen und innerhalb von BPEL behandelt wird, um den Aggregator erfolgreich zu beenden (siehe Abschnitt 6.3.4). In einer Überwachungskomponente der BPEL Maschine wird der abgelaufene Prozess beziehungsweise das Teilstück mit dem Zustand Fehler markiert. Auf der Ebene der Integrationsmuster ist dieser Fehlerzustand allerdings nicht relevant, denn der Aggregator wurde entsprechend der Spezifikation der Integrationsmuster erfolgreich abgeschlossen. Daher muss eine Übersetzung der Laufzeitinformationen auf die Modellebene erfolgen. Ein ähnlich gelagertes Problem tritt beim Testen von Integrationslösungen auf: Tritt im BPEL Prozess ein Fehler auf, muss dies auf die Modellebene übersetzt werden, da auf dieser Ebene das Problem behoben werden muss. Zu dieser Abbildungsproblematik existieren bereits erste konzeptionelle Arbeiten. Allerdings muss dieser Ansatz noch in die Methode beziehungsweise in die Werkzeugkette integriert werden.

Ein weiterer wichtiger Bereich der Integration sind menschliche Interaktionen innerhalb einer Integrationslösung. Die Integrationsmuster aus [HW03] treffen darüber keine Aussage. Es wird angenommen, dass einzig automatische Dienste in den Integrationslösungen eingebunden werden müssen. Offensichtlich ist aber, dass auch menschliche Interaktion in Geschäftsprozessen erforderlich ist [LR00]. Für BPEL wurde zum Beispiel ebenfalls eine Erweiterung erstellt, um Menschen in den Prozess einbinden zu können: BPEL4People [Org08]. Aus diesem Grund sollten die Integrationsmuster dahingehend erweitert werden, damit die Einbindung von Personen ermöglicht werden kann.

Momentan unterstützt das Muster External Service der PEP den Aufruf eines Web Services. Hierzu verwendet das Muster eine WSDL Datei, anhand derer die Generierungsalgorithmen entsprechende Anwendungen zum Aufruf

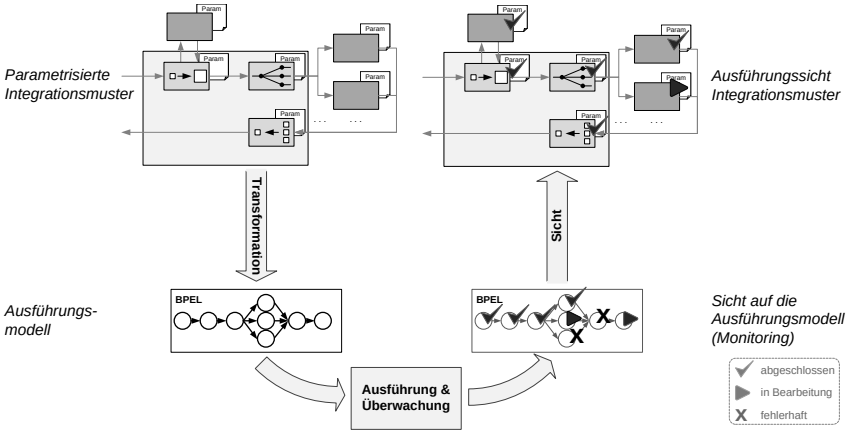


Abbildung 9.4.: Monitoring von ausführbaren Integrationsmustern

eines Web Service erzeugen. Allerdings kann nicht garantiert werden, dass alle Anwendungen beziehungsweise deren Dienste, die in einer Integrationslösung eingebunden werden sollen, per WSDL Schnittstelle aufrufbar sind. Daher sollte das Muster um die Möglichkeit erweitert werden, zusätzliche Aufrufmechanismen und -protokolle zu unterstützen.

Neben diesen konzeptionellen Erweiterungen, kann auch das Werkzeug weiter verbessert werden. Einige Punkte wurden in Abschnitt 8.8 bereits beschrieben. Durch die Einführung einer Simulationsmöglichkeit für modellierte Integrationsmuster können Systemarchitekten vor der Generierung ausführbarer Systeme das Modell einer Integrationslösung innerhalb GENIUS simulieren. Somit können Modellierungsfehler, die durch die in GENIUS integrierten Überprüfungslogiken nicht festgestellt werden, bereits frühzeitig erkannt werden.

In Abschnitt 6.5 wurde beschrieben, wie der Ansatz der parametrisierten Integrationsmuster für den Einsatz in der Cloud verwendet werden kann. Dieses Vorgehen kann mit der aktuellen GENIUS Implementierung kombiniert werden. Dazu muss das zugrundeliegende Modell um Eigenschaften bezüglich der *multi-tenancy* Muster erweitert werden. Außerdem muss ein weiterer Algorithmus entwickelt werden, der sich in den GENIUS Kern einklinkt. Auch dieser

Algorithmus verwendet das Datenmodell und erzeugt entsprechende Artefakte für die Cloud Umgebung. Dieser Ansatz hat allerdings den Nachteil, dass die bestehenden Algorithmen ebenfalls angepasst werden müssen, denn sie können *multi-tenancy* Eigenschaften nicht verarbeiten. Vielmehr können diese Eigenschaften nicht ohne weiteres auf BPEL oder Camel abgebildet werden, da bestehende Integrationsinfrastrukturen (etwa eine BPEL Maschine) nicht in der Lage sind, multi-tenancy zu unterstützen. Darüber hinaus könnte man die Modellierung einer Integrationslösung ebenfalls „as a Service“ anbieten. So könnte die Erstellung einer Integrationslösung über das Web auf Basis eines Prozesses erfolgen, die Modelle in der Cloud gespeichert und automatisiert übersetzt und ausgeführt werden. Auch hierfür kann der GENIUS Kern (die Datenhaltung) als Basis dienen und erweitert werden.

Die Modellierung von Integrationslösungen kann mit Hilfe des Prozessgesteuerten Ansatzes deutlich erleichtert werden. Allerdings treffen Systemarchitekten zumeist ad-hoc aufgrund ihrer Erfahrung Entscheidungen, welche Muster der Lösung hinzugefügt werden sollten, um bestimmte Probleme zu lösen. Durch die Einführung von Entscheidungsbäumen kann die Auswahl der passenden Muster strukturiert werden. In [ZGTL07] und [ZZGL08] wird ein Rahmenwerk (*SOA Decision Modeling, SOAD*) beschrieben, mit dem der Entwurf eines Systems, basierend auf einer SOA, unterstützt wird. Die Nutzung des SOAD Rahmenwerkes ist jedoch nicht auf den SOA Entwurf beschränkt, es kann auf zahlreiche Applikationsgenres und Architekturstile angewendet werden. Daher kann SOAD auch dahingehend erweitert werden, Integrationsmuster in Entscheidungsbäume einzubinden und Entscheidungen während der Modellierung von Integrationslösungen zu erleichten.

LITERATURVERZEICHNIS

[ABHK00] AALST, W.M.P. van d. ; BARROS, A. P. ; HOFSTEDE, A.H.M. ter ; KIEPUSZEWSKI, B.: Advanced Workflow Patterns. In: *CooplS '02: Proceedings of the 7th International Conference on Cooperative Information Systems*. London, UK : Springer-Verlag, 2000. – ISBN 3–540–41021–X, S. 18–29

[Act] ACTIVEENDPOINTS: *ActiveBPEL Engine*. – http://www.activebpel.org

[AHKB03] AALST, W.M.P. van d. ; HOFSTEDE, A.H.M. ter ; KIEPUSZEWSKI, B. ; BARROS, A.P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Juli, S. 5–51. – http://dx.doi.org/10.1023/A:1022883727209

[Ale77] ALEXANDER, C.: *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. – ISBN 0195019199

[Ale79] ALEXANDER, C.: *The Timeless Way of Building*. B&T, 1979. – 552 S. – ISBN 0195024028

[Apaa] APACHE SOFTWARE FOUNDATION: *Apache ActiveMQ*. – http://activemq.apache.org/

[Apab] Apache Software Foundation: *Apache Camel: a Spring based Integration Framework*. – http://activemq.apache.org/camel

[Apac] Apache Software Foundation: *Apache ODE (Orchestration Director Engine)*. – http://ode.apache.org/

[Apad] Apache Software Foundation: *Apache ServiceMix: an Open Source ESB (Enterprise Service Bus)*. – http://servicemix.apache.org

[Apae] Apache Software Foundation: *Apache Synapse Enterprise Service Bus (ESB)*. – http://synapse.apache.org

[Apaf] Apache Software Foundation: *CIMERO: a graphical Eclipse tool for ServiceMix*. – http://servicemix.apache.org/cimero-editor.html

[App00] Appelton, B.: *Patterns and Software: Essential Concepts and Terminology*. 2000. – http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

[Bac86] Bach, M. J.: *The Design of the Unix Operating System*. Prentice Hall, 1986. – ISBN 0132017997

[BD94] Bernstein, P. A. ; Dayal, U.: An Overview of Repository Technology. In: *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1994. – ISBN 1–55860–153–8, S. 705–713

[BDH05a] Barros, A. ; Dumas, M. ; Hofstede, A. ter: *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection*. März 2005. – http://eprints.qut.edu.au/2940/

[BDH05b] Barros, A. ; Dumas, M. ; Hofstede, A.H.M. ter: Service Interaction Patterns. In: *Business Process Management*. Springer Berlin, 2005. – ISBN 978–3–540–28238–9, S. 302–318

[Bec96] Beck, K.: *Smalltalk Best Practice Patterns*. Prentice Hall PTR, 1996. – ISBN 013476904X

[Bec06] Beck, K.: *Implementation Patterns*. Addison-Wesley Professional, 2006. – ISBN 0321413091

[BHL95] Blakeley, B. ; Harris, H. ; Lewis, R.: *Messaging and queueing using the MQI*. New York, NY, USA : McGraw-Hill, Inc., 1995. – ISBN 0–07–005730–3

[BMR+96] Buschmann, F. ; Meunier, R. ; Rohnert, H. ; Sommerlad, P. ; Stal, M.: *Pattern-Oriented Software Architecture*. Wiley, 1996. – ISBN 978–0–471–95869–7

[BSML06] Balasubramanian, K. ; Schmidt, D. C. ; Molnár, Z. ; Lédeczi, Á.: *System Integration Using Model-Driven Engineering*. 2006. – http://www.dre.vanderbilt.edu/ kitty/pubs/bookchapter-final.pdf

[CC06] Chong, F. ; Carraro, G.: *Building Distributed Applications Architecture Strategies for Catching the Long Tail*. http://msdn2.microsoft.com/en-us/library/aa479069.aspx : MSDN architecture center, 2006

[CCMW01] Christensen, E. ; Curbera, F. ; Meredith, G. ; Weerawarana, S.: *Web Services Description Language (WSDL) 1.1*. März 2001. – www.w3.org/TR/wsdl

[Cha04] Chappell, D. A.: *Enterprise Service Bus. Theory in Practice*. O'Reilly Media, 2004. – ISBN 978–0596006754

[CLS+05] Curbera, F. ; Leymann, F. ; Storey, T. ; Ferguson, D. ; Weerawarana, S.: *Web Services Platform Architecture: Soap, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005. – ISBN 0131488740

[E2E06] E2E Technologies Ltd.: *MDI - Model Driven Integration: a quantum shift in systems integration*. 2006. – http://www.e2e.ch/live/files/E2E-WP-MDI-351-EN.pdf

[E2E07] E2E TECHNOLOGIES LTD.: *E2E Bridge*. 2007. – http://www.e2ebridge.com

[Eck08] ECKHARDT, O.: *„Pipes and Filter“ – Architektur und Workflow-basierte Systeme: ein praktischer Vergleich*. Oktober 2008 http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2166

[Ecla] ECLIPSE FOUNDATION: *Eclipse Framework*. – http://www.eclipse.org/

[Eclb] ECLIPSE FOUNDATION: *Eclipse Modeling Framework (EMF)*. – http://www.eclipse.org/emf

[Eclc] ECLIPSE FOUNDATION: *Graphical Editor Framework (GEF)*. – http://www.eclipse.org/gef

[Ecld] ECLIPSE FOUNDATION: *The Standard Widget Toolkit (SWT)*. – http://www.eclipse.org/swt/

[Ecl09] ECLIPSE FOUNDATION: *Eclipse BPEL Project*. 2009. – http://www.eclipse.org/bpel

[Erl05] ERL, T.: *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2005. – ISBN 0131858580

[evi09] EVIWARE: *soapUI*. 2009. – htpp://www.eviware.com

[Fow06] FOWLER, M.: *Writing Software Patterns*. 2006. – http://martinfowler.com/articles/writingPatterns.html

[Fra03] FRANKEL, D. S.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003. – 352 S. – ISBN 0471319201

[FRF02] FOWLER, M. ; RICE, D. ; FOEMMEL, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, Amsterdam, 2002. – 560 S. – ISBN 0321127420

[FYL+08] Fu, S. S. ; Yang, J. ; Laredo, J. ; Huang, Y. ; Chang, H. ; Kumaran, S. ; Chung, J.-Y. ; Kosov, Y.: Solution Templates Tool for Enterprise Business Applications Integration. In: *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on* 0 (2008), S. 314–319. ISBN 978–0–7695–3158–8

[GHJ95] Gamma, E. ; Helm, R. ; Johnson, R. E.: *Design Patterns. Elements of Reusable Object-Oriented Software.* 1st ed., Reprint. Addison-Wesley Longman, Amsterdam, 1995. – 416 S. – ISBN 0201633612

[GR93] Gray, J. ; Reuter, A.: *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993. – ISBN 978–1558601901

[GS93] Garlan, D. ; Shaw, M.: An Introduction to Software Architecture. In: Ambriola, V. (Hrsg.) ; Tortora, G. (Hrsg.): *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 1993, S. 1–39

[Her07] Herbst, M.: Model Driven Legacy Integration / Transformation of Legacy-Software, Dagstuhl Workshop. 2007. – Forschungsbericht. – Transformation of Legacy-Software, Dagstuhl Workshop, http://www.tu-chemnitz.de/informatik/PI/transbs/aktuelles/folien/Dagstuhl_Herbst.pdf

[HL93] Habermann, H.-J. ; Leymann, F.: *Repository. Eine Einführung*. Oldenbourg, 1993. – ISBN 3486222007

[Hoh04] Hohpe, G.: *Enterprise Integration Patterns: Asynchronous Messaging Architectures in Practice*. 2004. – 5th Internation Middleware Conference, Tutorial, http://www.eecg.toronto.edu/middleware2004/tp04.htm

[Hoh05] Hohpe, G.: Developing Software in a Service-Oriented World. In: Vossen, Gottfried (Hrsg.) ; Leymann, Frank (Hrsg.) ; Lockemann,

Peter C. (Hrsg.) ; STUCKY, Wolffried (Hrsg.): *Datenbanksysteme in Business, Technologie und Web, 11. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)* Bd. 65, GI, 2005. – ISBN 3–88579–394–6, S. 476–484

[Hoh07] HOHPE, G.: Architect's dream or developer's nightmare? In: *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–665–3, S. 188–188

[HW03] HOHPE, G. ; WOOLF, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. – 736 S. – ISBN 0321200683

[IBM09a] IBM CORPORATION: *IBM WebSphere Enterprise Service Bus*. 2009. – http://www-01.ibm.com/software/integration/wsesb/

[IBM09b] IBM CORPORATION: *IBM WebSphere Process Server*. 2009. – http://www-01.ibm.com/software/integration/wps/library/

[Isa07] ISAACSON, C.: *Software Pipelines: A New Approach to High-Performance Business Applications*. http://www.softwarepipelines.org/wiki/Software_Pipelines_A_New_Approach_to_High-Performance_Business_Applications, 2007

[JGJ97] JACOBSON, I. ; GRISS, M. ; JONSSON, P.: *Software reuse: architecture, process and organization for business success*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1997. – ISBN 0–201–92476–5

[Juc06] JUCHART, F.: *Entwicklung eines Routing-Verfahrens für SOAP-Nachrichten*, Universität Stuttgart, Diplomarbeit, 2006. – 90 S. – http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2460

[Kay03] KAYE, D.: *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003. – ISBN 978–1881378242

[KB04] KELLER, A. ; BADONNEL, R.: Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. In: *Utility Computing: 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2004, Davis, CA, USA, November 15-17, 2004: Proceedings* (2004)

[Kel02] KELLER, W.: *Enterprise Application Integration. Erfahrungen aus der Praxis*. Dpunkt Verlag, 2002. – ISBN 3898641864

[Kol08] KOLB, P.: *Realisierung von EAI Patterns in Apache Camel*, Universität Stuttgart, Studienarbeit, April 2008. – http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2127

[Ley96] LEYMANN, Frank: Transaction concepts for workflow management systems, International Thomson Publ., Januar 1996 (Geschäftsprozeßmodellierung und Workflow-Management - Modelle, Methoden, Werkzeuge). – ISBN 3826601246

[Ley09] LEYMANN, F.: Cloud Computing. In: *Proc. 52th Photogrammetric Week*, Online, September 2009, S. 1–10. – http://www.ifp.uni-stuttgart.de/publications/phowo09/010Leymann.pdf

[Lin99] LINTHICUM, D. S.: *Enterprise Application Integration*. Addison-Wesley Professional, 1999. – ISBN 978–0201615838

[LKN+09] LENK, A. ; KLEMS, M. ; NIMIS, J. ; TAI, S. ; SANDHOLM, T.: What's inside the Cloud? An architectural map of the Cloud landscape. In: *ICSE Workshop on Software Engineering Challenges of Cloud Computing* (2009), S. 23–31. ISBN 978–1–4244–3713–9

[LR00] LEYMANN, F. ; ROLLER, D.: *Production Workflow: Concepts and Techniques*. Upper Saddle River, New Jersey : Prentice-Hall, 2000. – 479 S. – ISBN 0130217530

[LR05] LEYMANN, F. ; ROLLER, D.: Modeling Business Processes with BPEL4WS. In: *Information Systems and e-Business Management (ISeB)* (2005)

[Meu95] MEUNIER, R.: The pipes and filters architecture. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1995. – ISBN 0–201–60734–4, S. 427–440

[MHC00] MONSON-HAEFEL, R. ; CHAPPELL, D.: *Java Message Service*. O'Reilly, 2000. – ISBN 978–0596522049

[Mic] MICROSOFT COOPERATION: *Microsoft Message Queuing (MSMQ)*. – http://www.microsoft.com/windowsserver2003/technologies/msmq/defau

[Mie08] MIERZWA, C.: *Architektur eines ESBs zur Unterstützung von EAI Patterns*, Universität Stuttgart, Diplomarbeit, März 2008. – http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2665

[ML08a] MIETZNER, R. ; LEYMANN, F.: Generation of BPEL customization processes for SaaS applications from variability descriptors. In: *IEEE International Conference on Services Computing, 2008. SCC'08* Bd. 2, 2008

[ML08b] MIETZNER, R. ; LEYMANN, F.: Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In: *IEEE International Congress on Services (SERVICES 2008)*, 2008

[MLP08] MIETZNER, R. ; LEYMANN, F. ; PAPAZOGLOU, M. P.: Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and SaaS Multi-Tenancy Patterns. In: *Proceedings of the 3rd Intl. Conf. on Internet and Web Applications and Services ICIW 2008*. Athens, Greece : IEEE, Januar 2008

[MWSL07] MARTIN, D. ; WUTKE, D. ; SCHEIBLER, T. ; LEYMANN, F.: An EAI Pattern-Based Comparison of Spaces and Messaging. In: *Proceedings*

of the 11th IEEE International Enterprise Distributed Object Computing Conference : EDOC 2007 ; Annapolis, Maryland, October 15-19, 2007 (2007)

[NCLL01] Nielsen, H.F. ; Christensen, E. ; Lucco, S. ; Levin, D.: *Web Services Referral Protocol (WS-Referral)*. 2001. – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-referral.asp

[NT01] Nielsen, H.F. ; Thatte, S.: *Web Services Routing Protocol (WS-Routing)*. 2001. – http://msdn.microsoft.com/en-us/library/ms951249.aspx

[Obj03] Object Management Group (OMG): *Model-driven Architecture (MDA) Guide Version 1.0.1*. 2003. – http://www.omg.org/cgi-bin/doc?omg/03-06-01

[Obj06] Object Management Group (OMG): *MetaObject Facility (MOF) Version 2.0*. 2006. – http://www.omg.org/spec/MOF/2.0/

[Obj07] Object Management Group (OMG): *XMI Mapping Specification V 2.1.1*. 2007. – http://www.omg.org/technology/documents/formal/xmi.htm

[Obj09a] Object Management Group (OMG): *Business Process Model and Notation (BPMN) 1.2*. 2009. – http://www.omg.org/spec/BPMN/1.2/

[Obj09b] Object Management Group (OMG): *Unified Modeling Language (UML) Version 2.2*. 2009. – http://www.omg.org/technology/documents/formal/uml.htm

[Ope07] Open SOA Collaboration (OSOA): *SCA Service Component Architecture, Assembly Model Specification Version 1.00*. 2007. – http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[Ora09] ORACLE: *Java(TM) Platform, Enterprise Edition 6 (Java EE 6)*. 2009. – http://java.sun.com/javaee

[Org07a] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS): *Web Services Business Process Execution Language (WS-BPEL) Version 2.0 - OASIS Standard*. 2007. – http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.htm

[Org07b] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS): *Web Services Business Process Execution Language (WS-BPEL) Version 2.0 - Primer*. 2007. – http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0-Primer.htm

[Org08] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS): *WS-BPEL Extension for People (BPEL4People) TC*. 2008. – http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=bpel4people

[Pap07] PAPAZOGLOU, M. P.: *Web Services: Principles and Technology*. Prentice Hall, 2007. – ISBN 0321155556

[PW92] PERRY, D. E. ; WOLF, A. L.: Foundations for the Study of Software Architecture. In: *ACM SIGSOFT Software Engineering Notes* 17 (1992), S. 40–52

[RMB00] RUH, Wi. A. ; MAGINNIS, F. X. ; BROWN, W. J.: *Enterprise Application Integration: A Wiley Tech Brief*. 1. Wiley, 2000. – ISBN 0471376418

[SHLP05] SCHMIDT, M.-T. ; HUTCHINSON, B. ; LAMBROS, P. ; PHIPPEN, R.: The Enterprise Service Bus: Making service-oriented architecture real. In: *IBM Systems Journal, 44(4)* (2005)

[SKL09] SCHEIBLER, T. ; KARASTOYANOVA, D. ; LEYMANN, F.: Dynamic Message Routing Using Processes. In: *Proceedings of 16th Fachtagung Kommunikation in Verteilten Systemen (KiVS 09)*, Springer, 2009

[SL05] SCHEIBLER, T. ; LEYMANN, F.: Realizing Enterprise Integration Patterns in WebSphere / Universität Stuttgart. Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Oktober 2005. – Technischer Bericht Informatik. – http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2005-09&engl=0

[SL08] SCHEIBLER, T. ; LEYMANN, F.: A Framework for Executable Enterprise Application Integration Patterns. In: *4th International Conference Interoperability for Enterprise Software and Applications (I-ESA 2008)*, 2008

[SL09a] SCHEIBLER, T. ; LEYMANN, F.: Parameterization of integration patterns / Universität Stuttgart. Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Dezember 2009. – Technischer Bericht Informatik. – http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2009-06

[SL09b] SCHEIBLER, Thorsten ; LEYMANN, Frank: From Modelling to Execution of Enterprise Integration Scenarios: the GENIUS tool. In: *Proceedings of 16th Fachtagung Kommunikation in Verteilten Systemen (KiVS 09)*, Springer, 2009

[SML08] SCHEIBLER, T. ; MIETZNER, R. ; LEYMANN, F.: EAI as a Service - Combining the Power of Executable EAI Patterns and SaaS. In: *International EDOC Conference (EDOC 2008)*, Springer, 2008

[SML09] SCHEIBLER, T. ; MIETZNER, R. ; LEYMANN, F.: EMod: platform independent modelling, description and enactment of parameterisable EAI patterns. In: *Enterp. Inf. Syst.* 3 (2009), Nr. 3, S. 299–317. – ISSN 1751–7575

[Spr] SPRINGSOURCE: *Spring Framework*. – http://www.springsource.org/

[Sun05] SUN MICROSYSTEMS, INC.: *JSR 206 Java API for XML Processing (JAXP) 1.4*. 2005. – https://jaxp-sources.dev.java.net/nonav/docs/spec/html/

[SZC+07] Sun, W. ; Zhang, K. ; Chen, S. ; Zhang, X. ; Liang, H.: Software as a Service: An Integration Perspective. In: Kramer, B. J. (Hrsg.) ; Kwei-Jay, L. (Hrsg.): *Service-Oriented Computing - ICSOC 2007*, Springer, Berlin, 2007, S. 558

[TRH+04] Trowbridge, D. ; Roxburgh, U. ; Hohpe, G. ; Manolescu, D. ; Nedhan, E.G.: *Integration Patterns*. 1. Microsoft Press, 2004. – ISBN 073561850X

[Vin05] Vinoski, S.: Java Business Integration. In: *Internet Computing, IEEE* 9 (2005), Nr. 4, S. 89–91

[VS06] Völter, M. ; Stahl, T.: *Model-Driven Software Development*. Wiley & Sons, 2006. – 444 S. – ISBN 0470025700

[W3C99] W3C Recommendation: *XML Path Language (XPath) Version 1.0*. 199. – http://www.w3.org/TR/1999/REC-xpath-19991116

[W3C04] W3C: *Web Services Addressing (WS-Addressing)*. 2004. – http://www.w3.org/Submission/ws-addressing/

[W3C07a] W3C: *SOAP Version 1.2*, 2007. – http://www.w3.org/TR/soap/

[W3C07b] W3C: *Web Services Policy 1.5 Framework*, 2007. – http://www.w3.org/TR/2007/CR-ws-policy-20070330/

[WCL+05] Weerawarana, S. ; Curbera, F. ; Leymann, F. ; Storey, T. ; Ferguson, D. F.: *Web Services Platform Architecture*. Prentice Hall, 2005

[ZGTL07] Zimmermann, O. ; Grundler, J. ; Tai, S. ; Leymann, F.: Architectural Decisions and Patterns for Transactional Workflows in SOA. In: *Proceedings of 5th International Conference of Service-Oriented Computing (ICSOC)* (2007), September

[ZZGL08] Zimmermann, O. ; Zdun, U. ; Gschwind, T. ; Leymann, F.: Combining Pattern Languages and Reusable Architectural Decision Models into

a Comprehensive and Comprehensible Design Method. In: *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3092–5, S. 157–166

Alle Links wurden zuletzt am 08.04.2010 überprüft.

ABBILDUNGSVERZEICHNIS

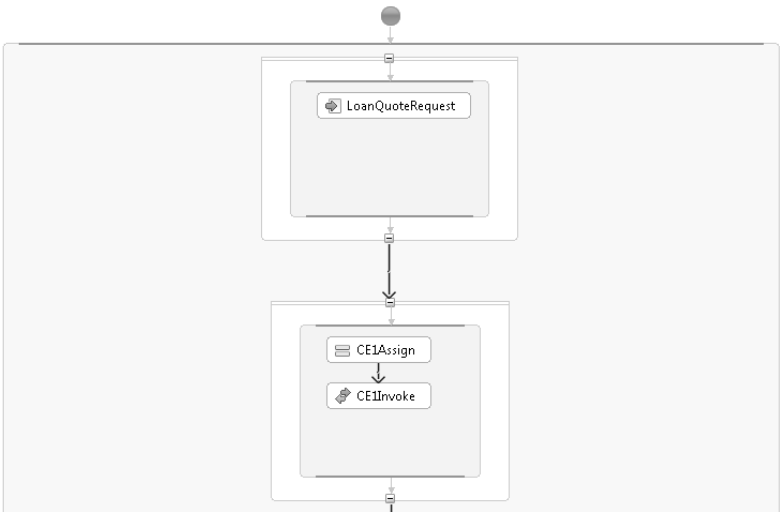
TABELLENVERZEICHNIS

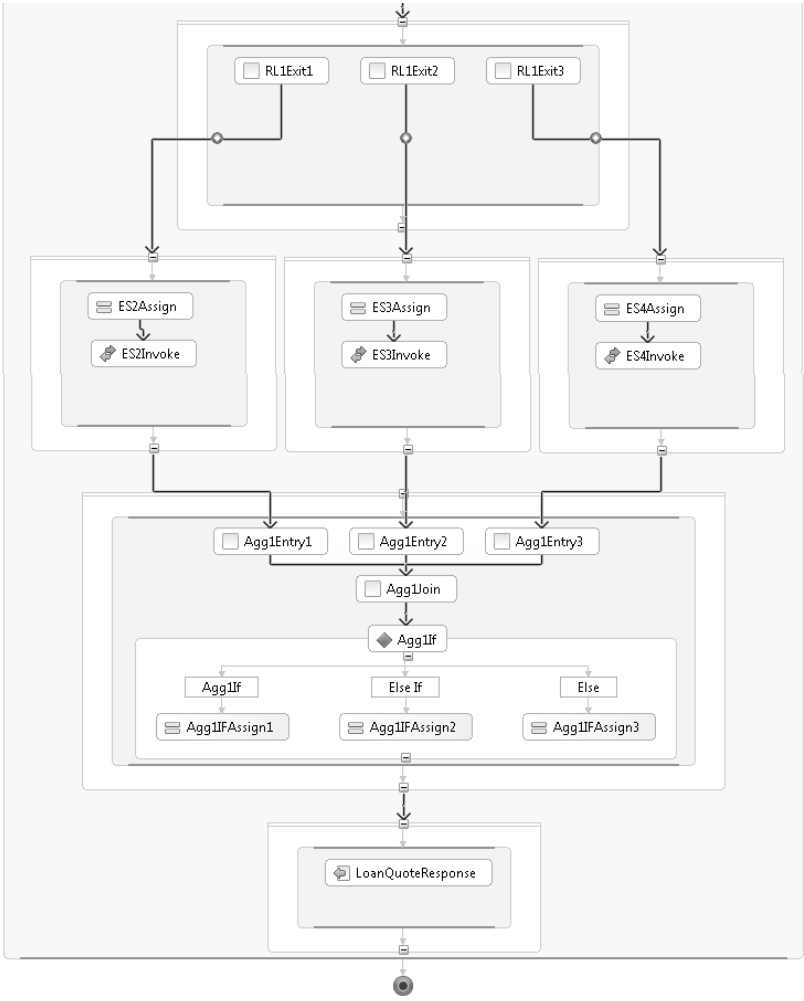
Listings

ANHANG A

DARVIEN BPEL IMPLEMENTIERUNG

A.1. BPEL Umsetzung (graphische Darstellung)





A.2. WSDL Datei des Darvien BPEL Prozesses

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:plnk="http://docs.oasis−open.org/wsbpel/2.0/plnktype"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://merhvomgeld.de/darvien"
```

```
    xmlns:vprop="http://docs.oasis−open.org/wsbpel/2.0/varprop"
    xmlns:wsdl="http://creditbureau.de/creditBureau/" name="darvien"
    targetNamespace="http://merhvomgeld.de/darvien"
    xmlns:p="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://mehrvomgeld.de/darvien">
  <plnk:partnerLinkType name="CreditBureauPLT">
      <plnk:role name="creditBureau" portType="wsdl:creditBureau"/>
    </plnk:partnerLinkType>
    <import location="creditBureau.wsdl"
       namespace="http://creditbureau.de/creditBureau/"/>
    <types>
      <schema xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:Q1="http://mehrvomgeld.de/darvien"
          attributeFormDefault="unqualified" elementFormDefault="qualified"
          targetNamespace="http://merhvomgeld.de/darvien">
        <import namespace="http://mehrvomgeld.de/darvien"
          schemaLocation="darvienMessages.xsd"/>
          <element name="loanQuoteRequest" type="Q1:loanQuoteRequestT"/>
          <element name="loanQuoteResponse" type="Q1:loanQuoteResponseT"/>
      </schema>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <xsd:import namespace="http://mehrvomgeld.de/darvien"
          schemaLocation="darvienMessages.xsd">
        </xsd:import>
      </xsd:schema>
    </types>
    <message name="darvienRequestMessage">
      <part element="tns:loanQuoteRequest" name="payload"/>
    </message>
    <message name="darvienResponseMessage">
      <part element="tns:loanQuoteResponse" name="payload"/>
    </message>
    <message name="getLoanQuoteRequest">
      <part name="payload" type="xsd:creditBureauResponseT"></part>
    </message>
    <message name="getLoanQuoteResponse">
      <part name="payload" type="xsd:bankResponseT"></part>
    </message>
    <portType name="darvien">
      <operation name="getLoanQuote">
        <input message="tns:darvienRequestMessage"/>
        <output message="tns:darvienResponseMessage"/>
      </operation>
    </portType>
    <plnk:partnerLinkType name="darvien">
      <plnk:role name="darvienProvider" portType="tns:darvien"/>
    </plnk:partnerLinkType>
    <plnk:partnerLinkType name="bankDepartment">
      <plnk:role name="bankDepartment" portType="tns:bankDepartment"/>
```

```
    </plnk:partnerLinkType>
    <portType name="bankDepartment">
      <operation name="getLoanQuote">
        <input message="tns:getLoanQuoteRequest"></input>
        <output message="tns:getLoanQuoteResponse"></output>
      </operation>
    </portType>
    <binding name="darvienServiceSOAPBinding" type="tns:darvien">
      <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="getLoanQuote">
        <soap:operation soapAction="http://merhvomgeld.de/darvien/getLoanQuote"/>
        <input><soap:body use="literal"/></input>
        <output><soap:body use="literal"/></output>
      </operation>
    </binding>
    <binding name="BankDepartmentSOAP" type="tns:bankDepartment">
      <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="getLoanQuote">
        <soap:operation
          soapAction="http://merhvomgeld.de/darvien/getLoanQuote" />
        <input><soap:body use="literal" /></input>
        <output><soap:body use="literal" /></output>
      </operation>
    </binding>
    <service name="darvienService">
      <port binding="tns:darvienServiceSOAPBinding" name="darvienServicePort">
        <soap:address location="http://localhost:9080/darvien"/>
      </port>
    </service>
    <service name="bankDepartment">
      <port name="getLoanQuoteBank1SOAP" binding="tns:BankDepartmentSOAP">
        <soap:address location="http://localhost:8080/bankdepartment1" />
      </port>
      <port name="getLoanQuoteBank2SOAP" binding="tns:BankDepartmentSOAP">
        <soap:address location="http://localhost:8080/bankdepartment2" />
      </port>
      <port name="getLoanQuoteBank3SOAP" binding="tns:BankDepartmentSOAP">
        <soap:address location="http://localhost:8080/bankdepartment3" />
      </port>
    </service>
  </definitions>
```

ANHANG B

SOAP BPEL ROUTING

B.1. WSDL-Datei des Prozesses

```
<?xml version="1.0" encoding="UTF−8"?>
<wsdl:definitions name="routingservice"
  targetNamespace="urn:iaas.uni−stuttgart.de/proposals/sbr/2006/08/routingservice"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business−process/"
  xmlns:sbrt="urn:iaas.uni−stuttgart.de/proposals/sbr/2006/08/types"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner−link/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sbr="urn:iaas.uni−stuttgart.de/proposals/sbr/2006/08/routingservice"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema elementFormDefault="qualified"
      targetNamespace="urn:iaas.uni−stuttgart.de/proposals/sbr/2006/08/types"
      xmlns:sbrt="urn:iaas.uni−stuttgart.de/proposals/sbr/2006/08/types"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:complexType name="serviceType">
      <xsd:sequence>
        <xsd:element name="serviceNamespace" type="xsd:anyURI"/>
        <xsd:element name="serviceRootElement" type="xsd:NCName"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="aggregationType">
      <xsd:sequence>
```

```
            <xsd:element maxOccurs="unbounded" name="pathId"
                type="xsd:positiveInteger"/>
        </xsd:sequence>
        <xsd:attribute name="service" type="xsd:QName" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="nodeType">
        <xsd:sequence>
            <xsd:element name="pathId" type="xsd:positiveInteger"/>
            <xsd:element name="nodeURI" type="xsd:anyURI"/>
            <xsd:element name="processURI" type="xsd:anyURI"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0"
                name="service" type="sbrt:serviceType"/>
            <xsd:element minOccurs="0" name="aggregate" type="sbrt:aggregationType"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="routeToType">
        <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0"
                name="node" type="sbrt:nodeType">
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    </xsd:schema>
</wsdl:types>
<wsdl:message name="getNextHopsRequest">
    <wsdl:part name="messageId" type="xsd:string"/>
    <wsdl:part name="pathId" type="xsd:positiveInteger"/>
</wsdl:message>
<wsdl:message name="getNextHopsResponse">
    <wsdl:part name="messageId" type="xsd:string"/>
    <wsdl:part name="routeTo" type="sbrt:routeToType"/>
</wsdl:message>
<wsdl:portType name="routingservicePortType">
    <wsdl:operation name="getNextHops">
        <wsdl:input message="sbr:getNextHopsRequest"/>
        <wsdl:output message="sbr:getNextHopsResponse"/>
    </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="routingserviceSOAP" type="sbr:routingservicePortType">
    <soap:binding xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getNextHops">
        <soap:operation xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
            soapAction="urn:iaas.uni−stuttgart.de/proposals/sbr/
            2006/08/routingservice/getNextHops"/>
        <wsdl:input>
            <soap:body xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                namespace="urn:iaas.uni−stuttgart.de/proposals/sbr/2006/
                08/routingservice" use="literal"/>
```

```
            </wsdl:input>
            <wsdl:output>
               <soap:body xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  namespace="urn:iaas.uni−stuttgart.de/proposals/sbr/2006
                  /08/routingservice" use="literal"/>
            </wsdl:output>
         </wsdl:operation>
      </wsdl:binding>
      <wsdl:service name="routingservice">
         <wsdl:port name="routingserviceSOAP" binding="sbr:routingserviceSOAP">
            <soap:address xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
               location="http://localhost:8080/active−bpel/services/RoutingService"/>
         </wsdl:port>
      </wsdl:service>
   </wsdl:definitions>
```