Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Evaluation feature-basierter service-orientierter Architekturen am Beispiel eines Domotic-Szenarios

Verfasser:

Chau Le Minh

22. Juni 2009

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl. Wirtsch-Inf. Christian Kästner

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Le Minh, Chau:
Evaluation feature-basierter service-orientierter Architekturen am Beispiel eines Domotic-Szenarios
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2009.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Verzeichnis der Abkürzungen

AHEAD	Algebraic Hierarchical Equations for Application Design
API	Application Programming Interface
ATS	AHEAD Tool Suite
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
EAI	Enterprise Application Integration
EDI	Electronic Document Interchange
FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programming
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
JAX-RPC	Java API for XML based RPC
JAX-WS	Java API for XML-Based Web Services
JAXB	Java Architecture for XML Binding
JCP	Java Community Process
OASIS	Organization for the Advancement of Structured Information Standards
OOP	Object-Oriented Programming
REST	Representational State Transfer
RMI	Remote Method Invocation
SOA	Service-Oriented Architecture
SOAP	*orig.* Simple Object Access Protocol
SPL	Software Product Line
SWR	Step-Wise Refinement
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XML	Extensible Markup Language

Kapitel 1

Einleitung

1.1 Motivation

Als spezielle Art der Software-Architektur zieht seit geraumer Zeit service-orientierte Architektur (SOA) immer mehr Aufmerksamkeit von IT-Unternehmen auf sich. Dadurch, dass Software im SOA-Konzept in Services zerlegt werden und sich selbst wiederum auch als grobe Services darstellen können, sollten sie flexibler in ihrer Aufbaustruktur sein, sowie auch bessere Modularität in komplexen Systemen besitzen als traditionelle Architektur. Darüber hinaus verspricht SOA noch einen höchsten Wiederverwendungsgrad für die bereits vorhandenen Services. Allerdings zeigen Forschungen in letzter Zeit doch im konkreten Fall einige Probleme von SOA in dieser Hinsicht auf.

Feature-orientierte Programmierung (FOP) wurde bisher vor allem für Software-Produktlinien (SPL) eingesetzt. Dort spielt FOP große Rolle durch Trennung der Variabilität von der Gemeinsamkeit in Features innerhalb der Produktlinie. Die Modularität bzw. Wiederverwendung folgen auch daraus.

In [AKL08] wurden die beiden oben genannten Begriffe *FOP* und *SOA* gemeinsam in einen Kontext versetzt. Dort haben die Autoren auch gezeigt, welche Nutzenvorteile eine Kombination aus diesen beiden Konzepten zur aktuellen Software-Entwicklung beitragen würde. Die offenen Herausforderungen aus [AKL08] bereiten uns die Motivation, diese Arbeit zu schreiben.

1.2 Problemstellung und Zielsetzung

In [AKL08] untersuchten Apel et al. anhand eines Anwendungsfalls Probleme von SOA in der Variabilitätsimplementierung. Identifizierbare Probleme können wir wie folgt beschreiben:

- keine klare Trennung zwischen Gemeinsamkeit und Variabilität bei zwei Service-Varianten
- Code-Replikation in Varianten bei gemeinsamer Nutzung eines Services
- steigender Programmieraufwand bei sinkender Produktivität

- keine Möglichkeit zum übersichtlichen Management der verschiedenen Varianten

Dies impliziert weitere Probleme bezüglich der Modularität, Wiederverwendung und Kompatibilität in einer SOA-Anwendung.

Die Autoren haben den Vorteil von FOP in diesem Kontext erwähnt, und deren Einsatz als Abhilfe hierbei vorgeschlagen. Anschließend nannten sie die vermutlichen Vorteile dieses neuen Konzepts bezüglich fünf Aspekte: Modularität, Variabilität, Uniformität, Spezifikation und Kompatibilität. Diese Herausforderungen bedürfen allerdings noch einer systematischen Evaluation.

Ziel der Arbeit ist es, den Einsatz der feature-orientierten Programmierung für die Entwicklung von service-orientierten Architekturen in einem Anwendungskontext zu evaluieren. Dabei wird die offene, oben beschriebene Problemstellung in Betracht gezogen. Konkret sollen folgende Punkte behandelt werden:

- Modularität in SOA: Ist es möglich querschneidende Belange und Features auch im SOA-Kontext mit FOP zu modularisieren?
- Implementierung von Variabilität: Bietet FOP eine Verbesserung bei der Implementierung von Variabilität, verglichen mit traditionellen Mitteln?
- Anwendung von FOP über mehrere Sprachen hinweg: Ist FOP über Sprachgrenzen hinweg ausdrucksstark genug und praktikabel?
- Nutzung von Feature-Modellierung zur Spezifikation und Kompatibilitätsprüfung von Varianten: Kann man mit Hilfe der Feature-Modellierung Kompatibilität von Varianten spezifizieren und prüfen?

1.3 Gliederung

Diese Arbeit gliedert sich in fünf Kapitel.

1. Dieses Kapitel dient der Einführung in die Arbeit. Motivation, Zweck bzw. Struktur der Arbeit wurden hier gezeigt.
2. Im zweiten Kapitel wird notwendiges Hintergrundswissen als Übergang vermittelt, um die praktischen Umsetzungen in weiteren Kapiteln verständlich zu machen.
3. Das Kapitel 3 beschreibt den Kontext des Anwendungsbeispiels in einer Fallstudie. Ein Anwendungsszenario in SOA-Umgebung wird konstruiert und entsprechend implementiert. Letztlich findet eine Problemanalyse statt, welche Schwierigkeiten sowie Schwächen der angewandten Herangehensweise aufzeigt.
4. Im Kapitel 4 wird das neu entstandene Konzept aus der Problemanalyse im letzten Kapitel zunächst praktiziert und abschließend evaluiert.
5. Das letzte Kapitel fasst alles inhaltlich noch mal zusammen. Ein Ausblick auf weitere Forschungs- und Entwicklungsmöglichkeiten rundet die Arbeit dann ab.

Kapitel 2

Grundlagen

Dieses Kapitel befasst sich mit dem theoretischen Hintergrund, welcher als Grundlagen für die restlichen Kapitel dient. Service-Oriented Architecture (deut. *Service-orientierte Architektur*), Software Product Line (deut. *Software-Produktlinie*) und Feature-Oriented Programming (deut. *Feature-orientierte Programmierung*) sind hier zu behandelnde Schwerpunkte.

2.1 Service-orientierte Architektur

„*Service-orientierte Architektur*“ (SOA) ist für IT-Unternehmen seit Jahrzehnten kein fremder Begriff mehr. Bei vielen Fachkonferenzen taucht dieser Terminus immer wieder auf als Lösung für Unternehmen zur Verstärkung deren Wettbewerbsfähigkeit. Hunderte Bücher sowie Artikel über SOA wurden bereits veröffentlicht und proportional dazu ist auch die steigende Vielzahl von dessen Interpretationen. Im folgenden Abschnitt werden wir mehr über dieses Schlagwort bzw. die für diese Arbeit relevanten Definitionen aus dessen Umfeld erfahren.

2.1.1 SOA im Allgemeinen

Begriffserklärung und Definition

Was versteht man unter SOA? Wohin in der Schlagwortswolke des Software Engineering kann man SOA einordnen? In welchem Zusammenhang steht dieser Begriff? Um diesen Fragen vernünftige Antworten zuzuführen, werden wir SOA aus zwei Perspektiven betrachten, zum einen aus Sicht einer Architektur, und zum anderen die Service-Orientierung an sich.

Wir beginnen unsere Betrachtung zunächst mit einer Definition von Software-Architektur aus [BCK03], Seite 21:

> „The *software architecture* of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.“

Die Architektur eines Software-Systems definiert demzufolge Elemente, aus denen das System besteht, und beschreibt den Zusammenhang, wie diese miteinander interagieren. Ein bedeutendes Architekturmuster für leistungsfähige Software-Systeme ist die *Client/Server-Architektur*, bei der viele Clients bei Bedarf auf wenige Server zur Anforderungen von Ressourcen zugreifen. Ein Beispiel hierfür ist das bekannte Schema vom World Wide Web. Web-Browsers sind in diesem Fall die Clients, welche sich in Verbindung mit einem Web-Server setzen, um z.B. HTML-Dokumente anzufordern.

Weiterhin werden Informationen über einzelne Software-Elemente im Architekturentwurf möglichst abstrakt gehalten. Nur die Informationen, welche für Interaktion mit anderen Elementen relevant sind, werden in Schnittstellen beschrieben und sind somit nach außen sichtbar. Andere werden unterdrückt oder gar vernachlässigt.

Ferner wird großes komplexes System zur besseren Kontrolle bzw. einfachen Realisierung in mehrere kleinere Systeme aufgeteilt (*separation of concerns*). In diesem Sinne umfasst die Architektur vom gesamten System alle einzelnen Strukturen der kleinen Systemen. In [Erl05] wird auch von *Applikationsarchitektur* gesprochen, welche in der Praxis zumeist die mehrschichtige Struktur eines verteilten Systems besitzt.

Innerhalb eines Unternehmens sind üblicherweise viele Anwendungen produktiv. Diese finden in unterschiedlichen Abteilungen bzw. Einrichtungen des Unternehmens vielfältigen Einsatz und bilden gemeinsam die unternehmensweite Applikationslandschaft. Meistens ist es erforderlich, Informationen zwischen verschiedenen Informationssystemen auszutauschen, d.h. *Interoperabilität* muss anhand der Architektur sichergestellt sein. Das ist der grundlegende Ansatz, der einzelne Applikationsarchitekturen zu einer gesamten *Unternehmensarchitektur* überführt (vgl. [Erl05]).

Ein kritischer Punkt, welcher die Schlüsselstellung sowohl in Applikations- als auch in Unternehmensarchitektur einnimmt, ist die *Integration* – Fähigkeit, Kommunikation zwischen von Struktur unterschiedlich aufgebauten (heterogenen) Systemen zu schaffen (vgl. [Erl05]). Diese Aufgabe ist sehr aufwendig, insbesondere wenn neue Applikationen in bestehende Systeme (sog. *Legacy-Systeme*) integriert werden müssen z.B. bei Geschäftsübernahme oder Unternehmensfusion.

In der Praxis teilt man Integrationsaufgaben in fünf Ebenen ein: Daten-, Funktions-, Prozess-/Vorgangs-, Methoden- und Programmintegration (vgl. [Mer04], Seiten 1–3). Entsprechende Integrationsansätze werden je nach Ebene entwickelt und implementiert. Bekannte Lösungsansätze sind bisher z.B. *Middleware*[1] in verschiedenen Formen (s. [ACKM04], Seiten 29–66), Enterprise Application Integration (EAI) (s. [ACKM04], Seiten 67–92), oder auch SOA, welche in letzter Zeit diesbezüglich immer mehr an Bedeutung gewinnt (vgl. [Erl04]).

In diesem Zusammenhang wird SOA als eine spezielle Form von Software-Architektur gesehen, welche sich hauptsächlich mit den Integrationsaufgaben beschätigt (s. [WBFT04], Seite 33).

Darüber hinaus steht der Begriff *Service* in SOA zentral im Mittelpunkt. Unter Service versteht man einen Dienst mit einer klar definierter Leistung. Service stellt wiederverwendbare Geschäftslogik dar, die in einem oder mehreren Geschäftsprozessabläufen

[1] Auf der Webseite `http://www.middleware.org` der Firma *Defining Technology, Inc.* wird das Themengebiet *Middleware* seit 1997 behandelt. Hier findet man z.B. Informationen darüber, was Middleware ist, welche Formen von Middleware derzeit existieren, bzw. ausführliche Listen von Middleware-Herstellern zu jeweiligen Kategorien.

eingesetzt werden kann (vgl. [Erl05]). Service-Orientierung entstammt dem Begriff „separations of concerns“ in Software Engineering mit dem Ziel, einen komplexen Sachverhalt auf kleinere Services herunterzubrechen, die gelöst werden können.

Wir müssen an dieser Stelle feststellen, dass es keine einheitliche, allgemeingültige Definition für SOA gibt. Alle angeblichen Definitionen, welche wir in den Literaturen gefunden haben, liefern meistens nur Beschreibung, was SOA tut, wie SOA aussieht, etc. Nachfolgend möchten wir uns den allgemeinen Aufbau von SOA anschauen.

Rollen in SOA

Die drei Rollen Service-Anbieter, Service-Konsument und Service-Vermittler bzw. deren Interaktion sind fundamentale Bausteine von SOA (s. [WBFT04], Seiten 37–38):

- *Service-Anbieter* (*provider*) implementiert seine eigenen Services, und stellt sie dann zur Verfügung. Den letzten Vorgang nennt man das Publizieren von Services. Diese Services werden in einem Repository (auch *Service-Registry* genannt) beim *Service-Vermittler* abgelegt.
- *Service-Konsument* (*consumer*) findet Services beim *Service-Vermittler* auf. Die gefundenen Services werden mit eindeutigen Adressen (*endpoint address*) zurückgeliefert. Damit kann er den *Service-Anbieter* direkt kontaktieren, um die Services abzufragen.
- *Service-Vermittler* (*broker*) nimmt die publizierten Services entgegen, legt im Repository ab, und ermöglicht dem *Service-Konsumenten* das Finden der gespeicherten Services.

Abbildung 2.1 stellt das Interaktionsszenario zwischen diesen Rollen grafisch dar. Falls sich Anbieter und Konsument gegenseitig kennen, kann die Rolle des Vermittlers ausbleiben. Der Konsument fordert in diesem Fall die vom Anbieter angebotenen Services über bekannte Endpoint-Adressen direkt an.

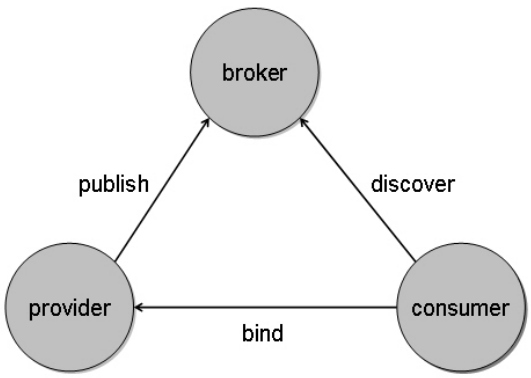


Abbildung 2.1: Rollen in SOA und deren Interaktionen

Nicht selten kommt es vor, dass der Bedarf an eine und dieselbe Funktionalität an mehreren Stellen von verschiedenen Abteilungen im Unternehmen zum unterschiedlichen Zeitpunkt gedeckt werden muss. Diese Funktionalität wird in der Praxis oftmals

mehrfach implementiert. Der Grund für diese Redundanz liegt daran, dass die Abteilungen nicht von der bereits vorhandenen Umsetzung der benötigten Funktionalität wissen. Durch Einsatz eines Service-Vermittlers, der alle wiederverwendbaren Services in einem zentralen Repository zur Verfügung stellt, wird dieses Problem ideal gelöst. Vorhandene Services werden dadurch leicht und schnell gefunden. Folglich lässt sich der Grad der Wiederverwendbarkeit einzelner Komponente drastisch erhöhen (vgl. [WBFT04], Seite 36).

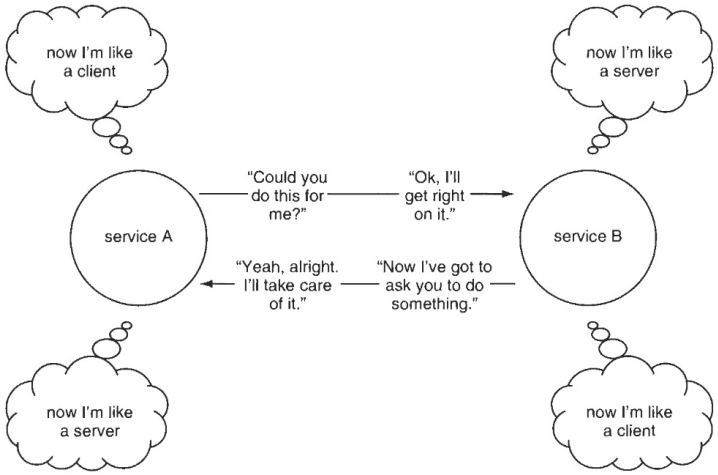


Abbildung 2.2: Client/Server-Rollen in SOA. Quelle: [Erl04], Seite 51.

Anbieter und Konsumenten sind theoretisch mit Server bzw. Clients in der *Client/Server-Architektur* vergleichbar. Der Unterschied besteht darin, dass diese Rollen in SOA nicht fest von vornherein definiert wurden. Ein Service-Anbieter kann gleichzeitig den Service-Konsumenten spielen, falls er Services wiederum von anderen Anbietern abfragt und konsumiert. Umgekehrt gilt der Rollentausch auch beim Service-Konsumenten (vgl. *Peer-to-Peer-Architektur*, wobei jede Komponente sowohl die Rolle eines Clients als auch eines Servers einnehmen kann, s. [Has05]). Dieses Verhalten ist in der Abbildung 2.2 deutlich zu erkennen.

Merkmale von SOA

Welche Merkmale zeichen SOA aus? Welchen Herausforderungen stellt sich SOA? Was möchte oder kann man mit SOA erzielen? Welche Nutzen bringt der SOA-Einsatz überhaupt? etc. Anhand dieser zahlreichen Fragestellung möchten wir an dieser Stelle einige bedeutende Charakteristiken von SOA erwähnen (vgl. [WBFT04], Seiten 38–42).

- *Interoperabilität* – Wie bereits besprochen, wird Interoperabilität in SOA durch Integrationsfähigkeit heterogenen Systeme erreicht.
- *Wiederverwendung* – Durch die Wiederverwendung bereits realisierten Services ergeben sich Zeit-, Qualitäts- und Kostenvorteile gegenüber einer Neuentwicklung.
- *Modularität und Granularität* – Geschäftsprozesse lassen sich mit SOA in modulare Services zerlegen. Grobgranulare Services haben weniger Abhängigkeiten (im Vergleich zu feingranularen Objekten), und erhöhen somit die Leistung des Systems.

- *Kapselung* – Die Implementierung von Services ist nicht für den Konsumenten sichtbar.
- *Flexibilität* - Durch *lose Kopplungen* zwischen Services können neue Geschäftsprozesse schnell konzipiert und entwickelt. Bei veränderten Anforderungen können unbrauchbare Services leicht durch Alternative ausgetauscht werden. Dies erhört die Flexibilität, sowie Wartbarkeit des Systems.
- *Dynamische Auffindung und Bindung* – Gewinn an Aktualität der Service-Ergebnisse
- *Zustandslosigkeit der Services* – Zustandslose Services sind skalierbar, zuverlässig.
- *Plattform- und Programmiersprachenunabhängigkeit*

Implementierung

Der Begriff SOA wurde 1996 in einer Studie von der Gartner Group zum ersten Mal geprägt[2]. Allerdings erst Ende der 90er Jahre, als Sun Microsystems ihr *Jini* Framework einführte[3], kam SOA öfter ins Gespräch.

Jini stellte damals ein Konzept zur dynamischen Auffindung und Nutzung von sogenannten *Services* in einem netzwerkfähigen Plug-and-Play-Szenario vor, wobei Services in einem *Lookup-Service* (vgl. Service-Repository von SOA) registriert und später über Discovery auffindig gemacht werden konnten (s. [OW01], Seiten 8–9).

Es gelang SOA erst 2001 nach der Einführung von Web Services mit ihrer Standardisierung den Durchbruch in die IT-Welt zu schaffen. Im Folgenden gehen wir auf diese Thematik näher ein.

2.1.2 Web Services: SOA-Implementierung

Wir wissen, dass Web Services und SOA heutzutage fast untrennbare Begriffe sind. Doch genau dies bereitet manchmal Probleme. Als bedeutendste Implementierung des SOA-Konzepts wurden Web Services leider immer öfter mit SOA selbst verwechselt. Um diese Trennung zwischen diesen beiden doch noch klar halten zu können, möchten wir uns in diesem Teilabschnitt Web Services genauer anschauen. Das erfolgt zunächst am besten mit einer Definition bzw. einem Vergleich mit „benachbarten“ Begriffen.

Definition und Abgrenzung

Das World Wide Web Consortium (W3C) hat Web Services wie folgt definiert [4,5]:

[2] im SSA Research Note SPA-401-068, am 12.04.1996, Service Oriented Architectures, Part 1.

[3] „*Sun shows off Jini*“, S. Shankland und M. Ricciuti, (`http://news.cnet.com/Sun-shows-off-Jini/2100-1001_3-218955.html?tag=mncol`

[4] Web Services Architecture Requirements, W3C Working Draft am 11.10.2002, `http://www.w3.org/TR/2002/WD-wsa-reqs-20021011`

[5] Auf der Webseite `http://www.jeckle.de/webServices/index.html#def` wurden noch viele anderen Definitionen über Web Services aus verschiedenen Quellen zusammengestellt.

> „A *Web service* is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A *Web service* supports direct interactions with other software agents using XML based messages exchanged via internet-based protocols.“

Laut Übersetzung in *Wikipedia* ist dies „eine Software-Anwendung, die mit einem URI eindeutig identifizierbar ist und deren Schnittstelle als XML-Artefakt definiert, beschrieben und gefunden werden kann. Ein Webservice unterstützt die direkte Interaktion mit anderen Software-Agenten unter Verwendung XML-basierter Nachrichten durch den Austausch über internetbasierte Protokolle.“

Wie bereits im vorigen Teilabschnitt erklärt wird jeder Service durch seine eindeutige Adresse identifiziert. Hierzu nutzen Web Services URI als Endpoint-Adressen. Für den Datenaustausch zwischen Rechnern im Internet oder zur Beschreibung eigenen strukturierten Daten wird üblicherweise XML eingesetzt. Web Services machen an dieser Stelle auch keine Ausnahme. Wie Web Services auf diesem technologischen Standard basieren, erfahren wir noch in diesem Teilabschnitt.

Den groben Vergleich zwischen Web Services mit anderen gängigen „benachbarten“ Middleware-Technologien wie OMG CORBA, Microsoft DCOM oder Sun RMI entnehmen wir der folgenden Tabelle 2.1 (s. angegebene Literaturen für mehr Details)

	CORBA	DCOM	RMI	Web Services
Initiative	OMG	Microsoft	Sun	W3C
Jahr	1996 (v2.0)	1996	1996	2001
Plattform	unabhängig	Microsoft	unabhängig	unabhängig
Sprache	unabhängig	unabhängig	Java	unabhängig
Protokoll	IIOP	DCE, RPC	JRMP, IIOP	SOAP, HTTP
Schnittstelle	IDL	-	-	WSDL
Verzeichnis	Naming Service	-	-	UDDI

Tabelle 2.1: Web Services und andere Middleware-Technologien. Quelle: [OHE99, Gro01, ACKM04], Microsoft Developer Network (MSDN).

Obwohl Web Services als konkrete Instanz der SOA zu verstehen sind, bedeutet es nicht zwangsläufig, dass eine Anwendung mit Web Services entwickelt wurde, gleich einer auf SOA basierten Anwendung zu setzen (vgl. [Erl04]). SOA ist in diesem Sinne ein abstraktes Konzept, und beinhaltet in sich keine technische Implementierung (vgl. [Erl05]).

Web Services stützen sich hauptsächlich auf drei zentrale technologische Standards (s. Abbildung 2.3), die wir im Folgenden betrachten möchten.

Web Services Standards

SOAP: ursprünglich für Simple Object Access Protocol
SOAP ist vergleichbar mit IIOP in CORBA (s. *oben*). Dieses fußt jedoch auf der XML-Technologie und bildet die Basis für Web Services Standards. Historisch gesehen stellt

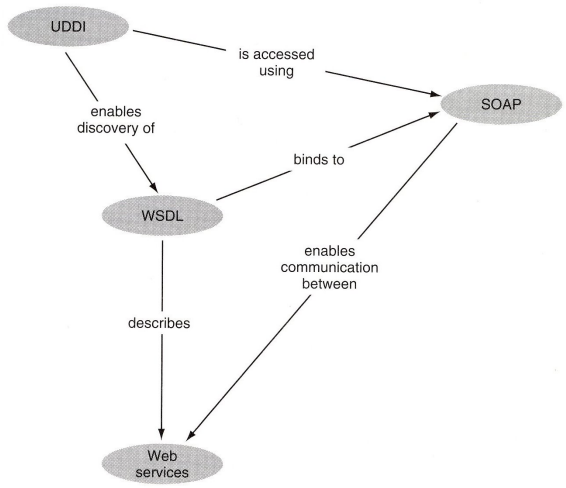


Abbildung 2.3: Web Services Technologien in Zusammenhang. Quelle: [Erl04], Seite 49.

SOAP die Weiterentwicklung für *XML-RPC*[6] dar, welches erstmalig eine Kombination von RPC-Architektur mit XML und HTTP zum entfernten Methodenaufruf durch verteilte Systeme 1998 vorstellte. Das Release v1.1 von SOAP[7] erfolgte im Jahr 2000, nachdem die entsprechende Spezifikation in Zusammenarbeit der 11 Software-Herstellern, u.a. HP, IBM, Compaq, Microsoft und SAP, beim W3C vorgelegt wurde. 2003 erhielt das nächste und aktuelle Release, SOAP 1.2[8], den Empfehlungsstatus von W3C. Ab dieser Version wird SOAP als Eigennamen behandelt und stellt kein Akronym mehr dar.

Dieser Standard dient als Protokoll zum Austausch von strukturierten Informationen in einer dezentralen, verteilten Umgebung (vgl. [WBFT04]). Auszutauschende Informationen, sog. *SOAP-Nachrichten*, werden in einem XML-Dokument mit vordefinierten Elementen verpackt. Eine solche Nachricht besteht in der Regel aus einem Umschlag (*SOAP envelope*), der einen optionalen *Header* und einen zwingend erforderlichen *Body* enthält. Wir verzichten im Rahmen dieser Arbeit auf eine detaillierte Beschreibung über den technischen Aufbau, sowie über die Verarbeitung der SOAP-Nachricht, da es zur Zeit reichliche Unterstützungstools gibt, welche diese Aufgaben gut erledigen können. Einige Hilfsframeworks werden wir im Verlauf dieses Kapitels auch noch vorstellen. Bei weiterem Interesse an das Thema SOAP verweisen wir auf [STK02].

Zur Hin- und Her-Übertragung von Nachrichten zwischen SOAP-Client und -Server können beliebige Transportprotokolle wie HTTP, SMTP oder FTP verwendet werden, wobei es in der Praxis meist auf HTTP oder HTTPS bei Verschlüsselung zurückgegriffen wird (vgl. [WBFT04]). Hinsichtlich des Kommunikationsstils wird bei SOAP zwischen *document-style* (oder auch Electronic Document Interchange (EDI) genannt) und *RPC-style* unterschieden. Es handelt sich hierbei um die Art und Weise der Dartellung vom Nachrichteninhalt innerhalb SOAP-Body (s. [ZTP03], Seiten 98–102).

Dr. Roy T. Fielding führte 2000 im Rahmen seiner Dissertation[9] den Begriff Representational State Transfer (REST) ein. Dieser wird heutzutage auch als eine Al-

[6] `http://www.xmlrpc.com/spec`

[7] `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`

[8] `http://www.w3.org/TR/2003/REC-soap12-part0-20030624/`

[9] „Architectural Styles and the Design of Network-based Software Architectures“, `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

ternative zu SOAP gesehen (s. [RR07]).

UDDI: Universal Description, Discovery and Integration
UDDI definiert Datenstruktur und API zur Veröffentlichung von Service-Beschreibungen im Repository (sog. *UDDI-Registry*) bzw. zur Abfrage der veröffentlichten Web Services (s. [ACKM04], Seite 174). Die erste UDDI-Spezifikation erschien im Jahr 2000. Aktuell liegt UDDI in der Version 3.0.2[10] seit 2005 bei OASIS vor. Analog zu einem Telefonbuch werden Informationen in diesem Verzeichnisdienst in drei Kategorien aufgeteilt. Diese sind White Pages, Yellow Pages und Green Pages (s. [WBFT04], Seiten 69–70).

- *White Pages* – enthalten Namen, Beschreibung und Kontaktinformationen der Unternehmen (z.B. Telefonnummer, Mailadresse), die Web Services publiziert haben.
- *Yellow Pages* – Alle Unternehmen sowie ihre sämtlichen Web Services werden nach bestimmter Verschlagwortung klassifiziert und kategorisiert, um die Suche nach den Services per Kategorieneingabe zu ermöglichen.
- *Green Pages* – enthalten Informationen darüber, wie Web Services aufgerufen werden können. Hier erhalten Konsumenten Verweise auf entsprechende Service-Beschreibungs-Dokumente, welche seitens des Service-Anbieters gespeichert wurden.

Diese Informationsaufspaltung der UDDI-Registry bietet den Konsumenten die Möglichkeit, seine Abfragen auf unterschiedliche Art und Weise formulieren und gewünschte Web Services schnell finden zu können. Allerdings wird UDDI nur in eher internen Firmennetzwerken verwendet und hat sich nie global durchgesetzt. Der Grund dafür liegt in der Identitätserkennung der Service-Anbieter. Das heißt, es existiert noch kein Mechanismus, um festzustellen, ob die dort registrierten Services tatsächlich den angegebenen Unternehmen entstammen (vgl. [ZTP03]). Ein bekanntes Beispiel für solchen globalen UDDI ist *XMethods*[11].

WSDL: Web Services Description Language
Mit WSDL liefert W3C eine XML-basierte Beschreibungssprache für Web Services Schnittstellen. Zwei Releases der WSDL-Spezifikation (Version 1.1[12] und 2.0[13]) sind momentan noch parallel im Einsatz. WSDL-Dokument stellt ein Vertrag dar, den der Kunde „akzeptieren“ muss, bevor er den Service nutzen kann. Dort wird z.B. geregelt, welche Operationen der Service besitzt bzw. wie der Kunde sie benutzen kann. Durch Abruf eines URI bekommt der Kunde dieses Dokument direkt vom Service-Anbieter zur Ansicht.

Wir werden uns später in dieser Arbeit noch viel mit WSDL v1.1 beschäftigen. Deshalb ist es an dieser Stelle sinnvoll, den Aufbau des WSDL-Dokuments anhand eines Beispiels (s. Listing 2.1) näher zu erläutern.

Listing 2.1: Beispiel eines WSDL-Dokuments

```
1 <definitions...>
2   <types>
3     <xsd:schema...>
```

[10] http://uddi.org/pubs/uddi-v3.0.2-20041019.htm
[11] http://xmethods.net/ve2/index.po
[12] http://www.w3.org/TR/2001/NOTE-wsdl-20010315
[13] http://www.w3.org/TR/2007/REC-wsdl20-20070626/

```
4       <element name="addNumbers" type="tns:addNumbers" />
5       <complexType name="addNumbers">
6         <sequence>
7           <element name="number1" type="xsd:int" />
8           <element name="number2" type="xsd:int" />
9         </sequence>
10      </complexType>
11      <element name="addNumbersResponse" type="tns:addNumbersResponse" />
12      <complexType name="addNumbersResponse">
13        <sequence>
14          <element name="return" type="xsd:int" />
15        </sequence>
16      </complexType>
17    </xsd:schema>
18  </types>
19
20  <message name="addNumbers">
21    <part name="parameters" element="tns:addNumbers" />
22  </message>
23  <message name="addNumbersResponse">
24    <part name="result" element="tns:addNumbersResponse" />
25  </message>
26
27  <portType name="AddNumbersPortType">
28    <operation name="addNumbers">
29      <input message="tns:addNumbers" name="add"/>
30      <output message="tns:addNumbersResponse" name="addResponse"/>
31    </operation>
32  </portType>
33
34  <binding name="AddNumbersBinding" type="tns:AddNumbersPortType">
35    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
36    <operation name="addNumbers">
37      <soap:operation soapAction="" />
38      <input>
39        <soap:body use="literal" />
40      </input>
41      <output>
42        <soap:body use="literal" />
43      </output>
44    </operation>
45  </binding>
46
47  <service name="AddNumbersService">
48    <port name="AddNumbersPort" binding="tns:AddNumbersBinding">
49      <soap:address location="http://localhost:8080/example/addnumbers" />
50    </port>
51  </service>
52</definitions>
```

Grundsätzlich besteht es aus zwei Teilen, einem abstrakten und einem konkreten Bereich (vgl. [ZTP03], Seiten 104–108). Der abstrakte Teil in WSDL beschreibt die verwendeten Datentypen und Operationen. Er stellt die Schnittstelle der Web Services mit folgenden Elementen dar:

- *types* – Hier werden alle Datentypen definiert, die zwischen Anbieter und Konsumenten ausgetauscht werden. Der WSDL-Standard hat nicht konkret spezifiziert, welche Sprachmittel verwendet werden sollen, so dass man hier freie Wahl hat, eine zwischen den gängigen Sprachen wie DTD, XML-Schema oder RELAX NG zu verwenden.

- *message* – liefert abstrakte Beschreibung der Nachrichten (als Anfrage oder Antwort). Innerhalb dieses Elements befindet sich eventuell ein `path`-Element, das Anfrageparameter bzw. Rückgabewert in Bezug auf vordefinierte Datentypen im `types`-Element festlegt.

- *portType*[14] – ist ein Definitions-Sammelpunkt für alle Service-Operationen. Jede einzelne Operation wird mit ihren bidirektionalen Nachrichten (`input` bzw. `output`) im `operation`-Element beschrieben.

- *part* – Unterelement von `message`. Dies bezieht auf einen vordefinierten Datentyp im `types`-Element.

Am Beispiel haben wir XML-Schema für die Beschreibung der zwei Datentypen *addNumbers* und *addNumbersResponse* verwendet. Der Datentyp *addNumbers* besteht aus zwei `int`-Werten *number1* und *number2*, und wird als Parameter für die *addNumbers*-Nachricht benutzt. Der *addNumbersResponse* aus einem `int`-Wert *return* wird als Rückgabewert für die *addNumbersResponse*-Nachricht. Beide Nachrichten sind In- bzw. Output für die Operation *addNumbers*, welche im `portType` erfasst wird.

Im konkreten Bereich beschreibt man die Web Services Implementierung. Die Bindung von der obigen abstrakten Beschreibung zu einem eindeutigen URI, einem Protokoll und konkreten Datenstrukturen (wie Daten serialisiert und codiert werden) wird hier definiert. Elemente zu diesem Bereich sind:

- *binding* – legt fest, über welchen Transportprotokoll (HTTP, FTP oder SMTP) die Nachrichten übertragen werden bzw. definiert SOAP-spezifische Informationen wie Kommunikationsstil der SOAP-Nachrichten (*rpc* oder *document*) und Datenkodierung (*literal* oder *encoded*).

- *service* – dient der Identifikation bzw. Lokalisierung des Service im Netzwerk. Im `port`-Element[15] wird eine eindeutige URI-Adresse festgelegt.

Unser Web Service *AddNumbersService* am Beispiel soll *RPC-style*, nicht kodierte (literal) SOAP-Nachrichten und HTTP als Übertragungsprotokoll verwenden.

Die Elemente eines WSDL-Dokuments sind nicht frei oder beliebig definierbar. Einige Restriktionen bezüglich des WSDL-Aufbaus werden in der sogenannten *Inhaltsstruktur* geregelt (s. Abbildung 2.4). Abbildung 2.5 veranschaulicht die logischen Beziehungen zwischen den WSDL-Elementen.

Außer den drei oben erwähnten Technologienstandards (SOAP, UDDI und WSDL) bringen Web Services noch einige andere mit sich heraus.

Weitere Standards in Web Services Interoperability Technology (WSIT) mit dem Präfix WS-*
WSIT spezifiziert Aspekte wie Metadaten, Sicherheit, Messaging und Transaktionen für Web Services Anwendungen. Da die WSIT-Standards für diese Arbeit nicht relevant sind, listen wir sie folgend nur stichwortartig auf. (s. `https://wsit.dev.java.net/` für aktuelle Information darüber)

- Metadaten
  - WS-Addressing
  - WS-Transfer

[14] WSDL v2.0 ändert diese Bezeichnung zu *interface*.
[15] WSDL v2.0 ändert diese Bezeichnung zu *endpoint*.

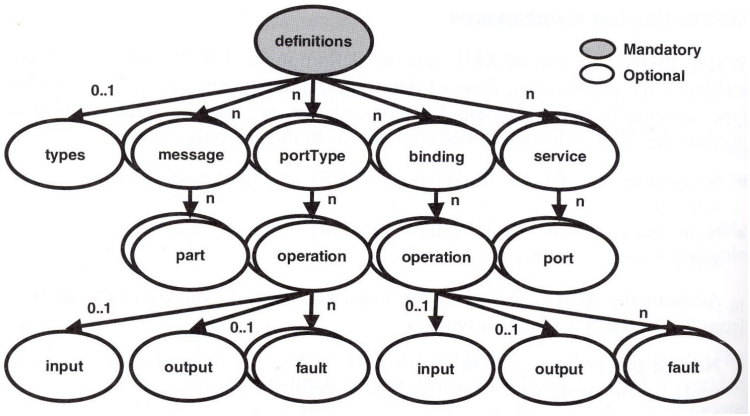


Abbildung 2.4: Inhaltsstruktur eines WSDL-Dokuments. Quelle: [ZTP03], Seite 110.

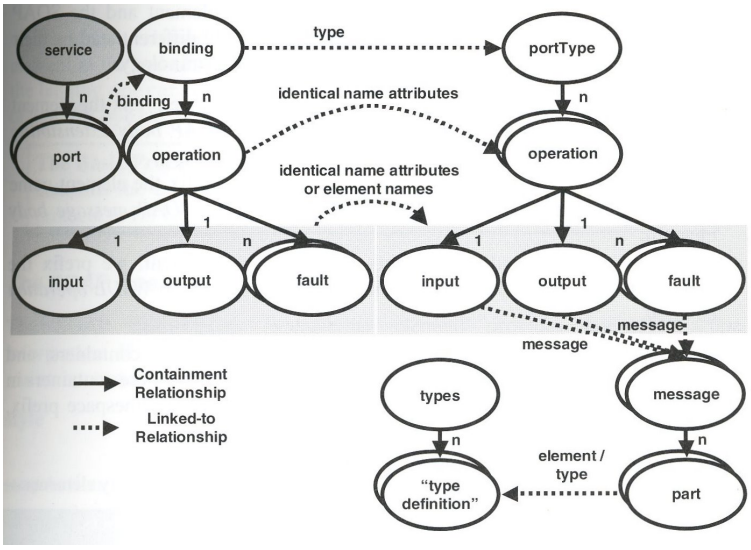


Abbildung 2.5: Logische Beziehungen zwischen den WSDL-Elementen. Quelle: [ZTP03], Seite 121.

  - WS-Policy

- Sicherheit
  - WS-Security
  - WS-SecureConversation
  - WS-Trust
  - WS-SecurityPolicy

- Messaging
  - WS-ReliableMessaging
  - WS-RMPolicy

- Transactionen
  - WS-Coordination
  - WS-AtomicTransaction

2.1.3 Web Services Entwicklung in Java

Ein Web Service besteht in der Regel aus einer Schnittstellenbeschreibung (sog. *Service Endpoint Interface*, SEI), welche den angebotenen Service beschreibt, und der Implementierungsklasse, die *Service Implementation Bean* (SIB) heißt.

Die Service-Schnittstelle liefert wiederum zwei Artefakte mit sich, nämlich ein mit Quelltext (sprich: Java *interface* – *code*-Artefakt) und ein ohne Quelltext (sprich: das WSDL-Dokument in XML – *non-code*-Artefakt). Sie beide müssen zu einander passen, da das eine Artefakt erstellt und das andere dementsprechend mittels eines bestimmten Schema-Mappings-Algorithmus generiert wird.

Vorgehensweise

Zur Implementierung von Web Services gibt es in der Praxis zwei verschiedene Ansätze, wobei der Unterschied zwischen denen in der Vorgehensweise liegt. Die Frage lautet hier, welches der beiden Artefakte der Service-Beschreibung zuerst erstellt werden soll. Falls die Antwort das code-Artefakt ist, geht man dem *Bottom-up*-Ansatz (*code-first*) nach. Andernfalls geht es um einen sogenannten *Top-down*-Ansatz (*contract-first*) (vgl. [FTW07]).

Toolunterstützung

Die erste Spezifikation zur Java Entwicklung von Web Services wurde 2002 mit dem JSR 109[16] von JCP[17] veröffentlicht. Da wurde JAX-RPC (JSR 101[18]) zur Verarbeitung von

[16] `http://jcp.org/en/jsr/detail?id=109`
[17] `http://jcp.org`
[18] Java APIs for XML based RPC, `http://jcp.org/en/jsr/detail?id=101`

XML-Daten hauptsächlich verwendet. Aktuell benutzt man JAX-WS (JSR 224[19]), den Nachfolger von JAX-RPC, sowie JAXB (JSR 222[20]) für die Web Services Entwicklung.

An sich ist JAX-WS jedoch nur eine reine Spezifikation. Das heißt, dies stellt keine Bibliothek im Sinne von API zur Verfügung. Um JAX-WS nutzen bzw. Java Web Services entwickeln zu können, muss man zusätzlich Hilfsframeworks erwerben, welche diese Spezifikation implementieren.

Derzeit finden folgende Frameworks populäre Verwendung[21]:

- *Apache Axis*[22] – bietet SOAP-Engine zur Konstruktion von darauf basierenden Web Services und Client-Anwendungen. Apache Axis ist eine Neuentwicklung und Nachfolger von Apache SOAP. Die höhere Geschwindigkeit erreicht AXIS durch Verwendung des SAX-Parsers, während Apache SOAP im Gegensatz dazu auf einem langsameren DOM-Parser aufbaute.

- *Apache Axis2*[23] – ist Neuentwicklung und Nachfolger von *Apache Axis*. Apache Axis2 unterstützt JAX-RPC-, und teilweise auch JAX-WS-Spezifikation.

- *Apache CXF*[24] – entstand aus einer Fusion der Projekte XFire und Ionas Celtix. Apache CXF unterstützt JAX-WS-Spezifikation und wird meistens für Integrationsaufgaben eingesetzt.

- *Sun Metro auf Glassfish*[25] – Sun Metro wird auch als Nachfolger von Sun JWSDP gesehen. Dieses enthält Referenzimplementationen (RI) für Java Web Services Spezifikationen wie JAX-WS, JAXB, etc. Die APIs sind bereits bei Auslieferung von dem Applikationsserver Glassfish mit drin.

Es ist schwer zu behaupten, welches Framework den anderen überlegen ist, da jedes eigene Stärken besitzt. Für die Implementierung in dieser Arbeit haben wir uns jedoch für Sun Metro bzw. den Glassfish Server entschieden, da uns die Web Services Entwicklung mit deren Unterstützung einfach und übersichtlich erscheint. Die Hersteller-Dokumentationen sind auch gut verständlich und ziemlich detailliert erfasst.

[19] Java API for XML-Based Web Services (JAX-WS) 2.0, `http://jcp.org/en/jsr/detail?id=224`

[20] Java Architecture for XML Binding (JAXB) 2.0, `http://jcp.org/en/jsr/detail?id=222`

[21] vgl. „*Apache Axis2, CXF und Sun JAX-WS RI im Vergleich*“, Thomas Bayer, `http://www.predic8.de/axis2-cxf-jax-ws-vergleich.htm`

[22] `http://ws.apache.org/axis/`

[23] `http://ws.apache.org/axis2/`

[24] `http://cxf.apache.org`

[25] `https://metro.dev.java.net`

2.2 Software-Produktlinie

Das der Industrie entstammte Konzept von *Produktlinie* ist nicht neu. Tatsächlich lässt sich dieses auf Henry Fords Idee zur Verbesserung der damals im Einsatz befindlichen Fließbandfertigung mit auswechselbaren Komponenten in der Automobilbranche zurückführen. Die Massenproduktion von qualitativ hochwertigen Produkten wurde dadurch viel schneller und auch günstiger als bei traditioneller Handfertigung (vgl. [PBvdL05]).

Wir geben nachfolgend zuerst einen kurzen Überblick darüber, wie der Begriff *Produktlinie* durch Probleme der Fertigung in der Industrie zu Stande kam. Ein Vergleich mit der herkömmlichen Herstellung von Software-Produkten ergibt die gleichartigen Probleme und schlussfolgert einen möglichen Zusammenhang zwischen Produktlinien und der Software-Entwicklung. Dieser Zusammenhang bzw. entstandene Begriffe aus dessen Umfeld sind abschließend unser letzter Fokus im diesem Abschnitt.

2.2.1 Historie und Herkunft

Massenproduktion, wie bereits erwähnt, hatte seit ihrer Einführung in der Industrie im vergangenen Jahrhundert deutlichen Vorteil gegenüber Handfertigung. Allerdings fehlen es an diesem Ansatz die Diversifikationsmöglichkeiten, die Flexibilität auf Veränderung. Die Hersteller mussten mit der stetig wachsenden Anforderung von Kunden an individualisierte Produkte konfrontieren. Diese Herausforderung führt erneut zu einem Problem der Industrie, das eines neuen Lösungsansatzes bedarf.

In den späten 80er Jahren wurde der Begriff *Mass Customization* (deut. *individualisierte Massenfertigung* oder auch *Massen-Maßfertigung*) in der Industrie eingeführt, welcher sich als „large-scale production of goods tailored to individual customers' needs“ definieren lässt (s. [PBvdL05], Seite 4). Unter diesem Begriff versteht man „eine Synthese aus Massenproduktion einerseits und der Befriedigung individueller Kundenbedürfnisse durch Produkte und Dienstleistungen, die der Kunde nach seinem Wunsch gestaltet, andererseits“ (s. [Mer04], Seite 59).

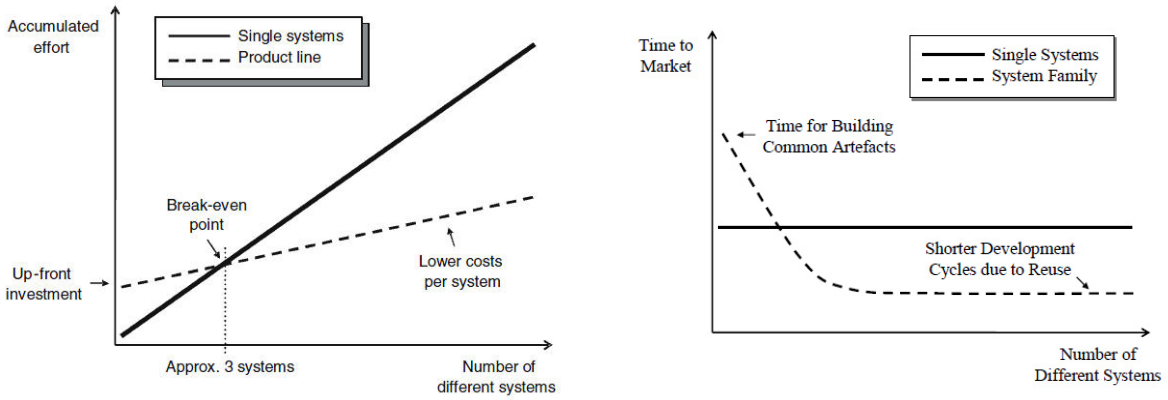
Durch Variation aus wenigen, aus Kundensicht jedoch entscheidenden Merkmalen des Produktes soll eine Individualisierung erreicht werden. Dies bedeutet, ein größeres Angebot an Produktvarianten muss erstellt werden (vgl. [Mer04]). Damit werden alle bekannten Vorteile der Massenproduktion wie Skaleneffekte, Niedigkosten, etc. weiter genutzt, und gleichzeitig individuelle Kundenwünsche auch mit berücksichtigt.

Mass Customization bedeutet jedoch für die Hersteller hohes Investment an Technologien. Demzufolge muss der Preis für individualisierte Produkte erhöht werden, was wiederum einen Nachteil gegenüber Kunden darstellt. Andernfalls muss das Unternehmen großen Teil dessen Gewinns selbst einbüßen (vgl. [PBvdL05]).

Um die genannten negativen Effekte des Mass Customization zu umgehen wurde sog. *gemeinsame Plattform* vorgestellt, welche das Grundgerüst mit allen Hauptkomponenten einer Produktreihe aufweist (s. [PBvdL05], Seite 5). Diese Komponenten sind als Basistechnologien für die Herstellung neuen Produkte nach Kundenanforderungen wiederverwendbar. Die Kombination von gemeinsamer Plattform mit Mass Customization führt zu einem neuen Begriff – die *Produktlinie*. Eine Produktlinie enthält in diesem Sinne Menge von Produkten, welche die Anforderungen eines gemeinsamen Anwendungsbereichs ab-

decken. Diese Produkte haben außerdem eine gemeinsame Basis und unterscheiden sich in variablen Teilen (vgl. [CN01]).

Ersichtliche Vorteile bezüglich der Entwicklungskosten bzw. Zeit zur Markteinführung von Produkten beim Einsatz von Produktlinien gegenüber Einzelproduktfertigung sind der Abbildung 2.6 zu entnehmen.



(a) Wirtschaftlichkeit (b) Zeit zur Markteinführung

Abbildung 2.6: Vorteile vom PL-Ansatz. Quelle: [vdLSR07, PBvdL05].

Um eine Brücke von Produktlinie hin zu ihrer Einführung im Bereich der Software-Entwicklung zu schlagen, beschäftigen wir uns zunächst mit der Fragestellung, welche Arten der gängigen Software-Produkte den Kunden angebotenen wurden? Durch Klassifizierung der Software auf dem Markt hinsichtlich der Kundenanforderung stellt sich heraus, dass man dort zwischen *Individual-* und *Standardsoftware* unterscheidet.

Standardsoftware wird definiert als „fertig entwickelte Lösung am Markt... ohne nennenswerten zusätzlichen Programmieraufwand im Unternehmen für die betriebliche Leistungserstellung...“ (s. [RS03], Seite 281). In der Regel wird sie für eine große Menge (potenzieller) Kunden entwickelt. Dagegen werden *Individualsoftware* gemäß den Anforderungen eines *einzelnen Kunden* maßgeschneidert erstellt[26].

Aus Kundensicht ist die Beschaffung einer Individualsoftware kostenintensiver gegenüber Standardsoftware. Allerdings fehlt es bei Standardsoftware genügende Diversifikation der Produktpalette. Deren Entwicklung zeigt sich inflexibel. Die Situation ist nahezu vergleichbar mit dem Problem im Industriebereich, das wir anfänglich beschrieben. Deshalb besteht es auch dort ein Anspruch, das Produktlinienkonzept zu adaptieren. Wie dieses definiert und dann umgesetzt wird, erfahren wir in den nachfolgenden Abschnitten.

2.2.2 Begriffserklärung

Hier im Vorfeld möchten wir einige Begriffe, die im Zusammenhang zu SPL entstehen und im Verlauf der Arbeit öfter verwendet werden, zum Verständnis bringen.

Wir betrachten zunächst folgende Definition zu *Software-Produktlinie* (SPL):

[26] `http://de.wikipedia.org/wiki/Individualsoftware`

> „A *software product line* is a set of software-intensive systems that share a common, managed set of features satisfying needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“ (laut *Software Engineering Institute* (SEI) der Carnegie Mellon Universität)

Analog zum originalen Konzept in der Industrie liefert Software-Produktlinie laut Definition eine Menge von Software-Produkten, die auch gemeinsame Features besitzen bzw. auf ein bestimmtes Marktsegment (vgl. *Domäne*) zugeschnitten sind. Diese Software-Produkte (hier als *software-intensive Systeme* genannt) können sowohl reine Anwendungssoftware als auch eingebettete Anwendungen im Hardwareteil sein. Sie werden aus vorhandenen Ressourcen in der gleichen Art und Weise kostensparend, schnell, flexibel und kundenorientiert entwickelt.

Begriffe, die von vielen Autoren interpretiert wurden, sind im Großen und Ganzen leider nicht einheitlich. Bei Software-*Feature* (deut. *Merkmal*) ist das auch keine Ausnahme.

Kang et al. (s. [KCH+90]) sahen z.B. Feature als „user-visible aspect, quality, or characteristic of s software system“ bzw. Getriebeart (hier: Automatik oder manuelles Getriebe) sei Entscheidungsorientierungsmerkmal beim Autokauf.

In [Bat05] wurde *Feature* aber als eine Erweiterung bezüglich der Programmfunktionalität betrachtet.

Sicherlich gibt es noch diffuse Menge an Definitionen von *Feature*. Aber für diese Arbeit halten wir uns an die Definition von Batory.

Zwei weitere Begriffe, die wir hier erwähnen möchten, sind *Gemeinsamkeit* (commonality) und *Variabilität*. Als Gemeinsamkeit wird in der Produktlinie Charakteristik bezeichnet, die bei allen Produkten allgemeingültig ist. Diese ist Teil der Plattform (vgl. [vdLSR07]).

Variabilität muss dagegen explizit modelliert werden, da sie nicht zur Plattform gehört. Weiterhin bei Repräsentation von Variabilität unterscheidet man noch zwischen *Varianten* und *Variationspunkt* (s. [vdLSR07], Seite 10).

Nun stellen wir im Folgenden ein Framework zur Software-Produktlinien vor.

2.2.3 Software-Produktlinien-Framework

Klassische Software-Entwicklung durchläuft im Allgemeinen sieben Phasen: Problemdefinition, Anforderungsanalyse, Spezifikation, Entwurf, Implementation, Erprobung und Auslieferung (vgl. [Dum03], Seiten 18–19). Diese Phasen bilden gemeinsam das Vorgehensmodell einer Entwicklung, z.B. *Wasserfallmodell* oder *V-Modell*. Der Einsatz eines solchen Modells ist durchaus sinnvoll, da die Entwicklung an sich damit überschaubar bzw. kontrollierbar wird.

In diesem Sinne empfahl das *Software Engineering Institute* (SEI) 1997 ein Framework zur Software-Produktlinien[27] (s. [CE00], Seiten 20–30), das wir hier vorstellen möchten.

[27] Das hier vorgestellte Framework von SEI ist mit dem in [PBvdL05], Seite 22 vergleichbar.

Das Framework beschreibt zwei parallel laufende Hauptphasen (s. [PBvdL05], Seiten 20–21), die jeweils auch eigene Prozesse enthalten (s. Abbildung 2.7).

- *Domain Engineering* (sog. *development for reuse*) – ist für den Aufbau von gemeinsamer Plattform sowie die Definition der Gemeinsamkeit und Variabilität in SPL verantwortlich
- *Application Engineering* (sog. *development with reuse*) – leitet Produktlinie von der Plattform ab, und bildet Fertigprodukte daraus

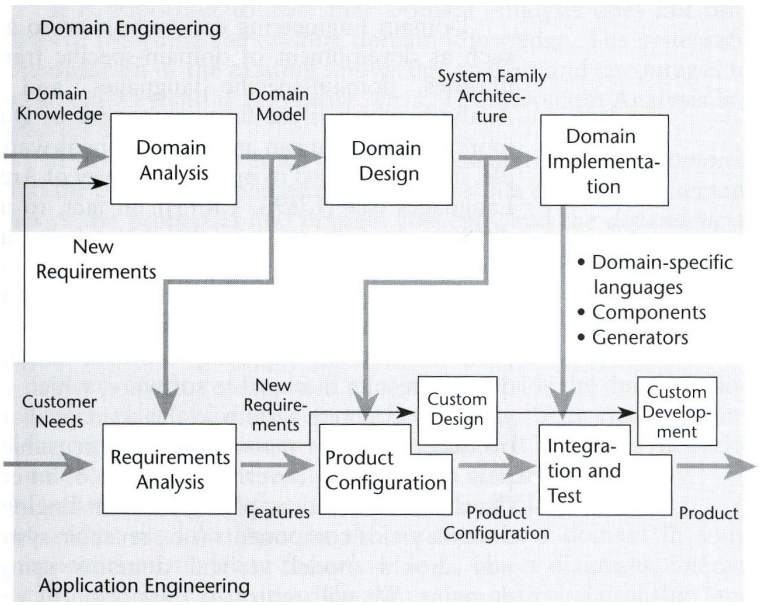


Abbildung 2.7: SPL-Framework. Quelle: [CE00], Seite 21.

Nachfolgend werden wir uns diese einzelnen Prozesse genauer anschauen.

Domain Engineering

Der Begriff *Domäne* bezeichnet ein Anwendungsgebiet, auf das alle Produkte entlang einer Produktlinie zugeschnitten sind. Erkenntnis über wiederverwendbare Objekte einer Domäne und deren Nutzung zur Entwicklung neuen Produkte ermöglichen den Unternehmen, ihre Produkte mit hoher Qualität, jedoch niedrigeren Kosten in kürzerer Zeit zum Markt zu bringen (s. [CE00], Seite 20). *Domain Engineering* gibt einen systematischen Ansatz, dieses Ziel zu erreichen.

Laut Definition in [PBvdL05], Seite 21 ist das der Prozess in SPL, in dem Gemeinsamkeit und Variabilität der Produktlinie bezüglich der Domäne definiert bzw. realisiert werden.

Wir betrachten jetzt folgende drei Prozesse des Domain Engineering: Domänenanalyse, Domänendesign und Domänenimplementierung.

Domänenanalyse

Domänenanalyse dient der Auswahl und Definition der Domäne. Hier werden relevante Informationen über die Domäne gesammelt, und in ein sogenanntes *Domänenmodell*

integriert, welches eine grafische Darstellung der gemeinsamen bzw. variablen Eigenschaften des Produktes dieser Domäne, sowie Darstellung der Abhängigkeiten zwischen den variablen Eigenschaften ist (vgl. [CE00], Seite 23).

In [KCH+90] wurde Feature-Oriented Domain Analysis (FODA) als erste Methode zur Domänenanalyse vorgestellt. Die FODA-Methode umfasst zwei Phasen: Kontextanalyse und Domänenmodellierug (oder auch Feature-Modellierung).

Ergebnis der Feature-Modellierung ist ein Feature-Modell, das die hierarchische Aufbaustruktur einer Gruppe von Features beschreibt. Zur grafischen Darstellung wird *Feature-Diagramm* verwendet. Abbildung 2.8 stellt alle möglichen Notationen zur Beschreibung der Beziehung zwischen den Eltern- (*compound*) und Kindknoten (*subfeatures*) in einem solchen Diagramm dar (s. [Bat05]).

- *And* – Alle Features müssen ausgewählt werden.
- *Alternative* – Nur ein der Features kann ausgewählt werden.
- *Or* – Ein oder mehrere Features können ausgewählt werden.
- *Mandatory* – Features sind zwingend erforderlich.
- *Optional* – Features sind optional auswählbar.

Abbildung 2.8: Notationen für Feature-Diagramm. Quelle: [Bat05].

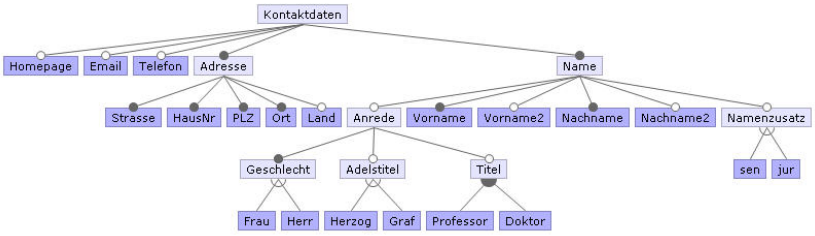


Abbildung 2.9: Beispiel eines Feature-Diagramms

Diese vielfältigen Darstellungsmöglichkeiten lassen sich an einem Beispiel einfach erklären (s. Abbildung 2.9). Demnach bestehen Kontaktdaten obligatorisch aus Namen und Adresse. Zusätzliche Informationen z.B. Email, Telefon, oder Webseite sind optional. Ein Name besteht wiederum aus einem Vornamen und einem Nachnamen. Zum Namen kann man auch eine Anrede oder deren Kombination hinzunehmen.

Domänendesign und Domänenimplementierung

- *Domänendesign* – entwickelt eine Architektur für Produktlinie der Domäne und entwirft Plan zur Produktion (hier: Features).
- *Domänenimplementierung* – implementiert die Architektur und den Produktionsplan (vgl. Feature-Implementierung später in FOP)

Application Engineering

Es handelt sich hierbei um die Prozesse zur Konfiguration bzw. Generierung des Fertigproduktes aufgrund der Ergebnisvorlage aus *Domain Engineering* (vgl. [CE00], Seite 30).

Application Engineering umfasst folgende Prozesse:

- *Anforderungsanalyse* – Unter Berücksichtigung der Kundenwünsche bzw. des Feature-Diagramms wird eine neue Anforderung zusammengestellt und an nächsten Prozess weitergereicht.
- *Produktkonfiguration* – erstellt das Produkt (vgl. Feature-Komposition später in FOP)
- *Integration und Test* – validiert bzw. verifiziert das Produkt gegenüber seiner Spezifikation

2.3 Feature-orientierte Programmierung

Im vorigen Abschnitt haben wir über Software-Produktlinien (SPL) als grobes Konzept in der Software-Entwicklung diskutiert. Hier werden wir eine Möglichkeit betrachten, dieses Konzept zu implementieren, nämlich mit feature-orientierter Programmierung (FOP).

Prehofer führte in seiner früheren Publikation [Pre97] die ersten Grundlagen zu FOP ein. Dort wurde FOP als eine Erweiterung für das seit langem etablierte objekt-orientierte Programmierparadigma (OOP) gesehen hinsichtlich der Flexibilität in der Wiederverwendung von Objekten gegenüber dem traditionellen OOP-Vererbungsprinzip.

Später stellten Batory et al. in [BSR04] das *Step-Wise Refinement* (deut. *schrittweise Verfeinerung*, SWR) vor. Dieses legt die Verfeinerung von Features zugrunde und bildet einen Mechanismus zur Entwicklung komplexen aus einer einfachen Software.

Beide Ansätze haben im Grunde genommen ein gemeinsames Ziel, nämlich die Modularisierung bzw. flexible Komposition komplexen Software zu erreichen. Welchen Weg zu diesem Ziel sie konkret genommen haben, werden wir uns folgend anschauen.

2.3.1 FOP-Methodik

Prehofer erkennt den Nachteil von der starren Klassenstruktur im klassischen Vererbungskontext der Objekte, so dass er die modularen Features in einem Feature-Repository ablegt. Features entstehen in diesem Sinne aus der Trennung der Kernfunktionalität einer Subklasse von zu überschreibenden Methoden deren Superklasse. Die Interaktionen zwischen ihnen erfolgen über den sogenanten *lifter* (vgl. [Pre97]). Dadurch ist die Komposition der Features auch flexibler.

Das war die grundlegende Idee in Prehofers Vorgehensweise. Wir gehen an dieser Stelle nicht mehr näher in diese ein, da wir uns später in dieser Arbeit hauptsächlich nur mit Batorys AHEAD-Modell beschäftigen werden. Für eine detaillierte Erläuterung verweisen wir deshalb auf seine oben erwähnte Arbeit ([Pre97]).

Der in [BSR04] vorgestellte SWR-Ansatz stellt eine allgemeine Methodik zur Entwicklung eines komplexen Programms dar, indem das einfache Ausgangsprogramm inkrementell um neue Features ergänzt wird. Unter einem Feature versteht man hier die Charakteristik, die zur Unterscheidung zwischen den einzelnen Produktvarianten innerhalb einer Produktlinie verwendet wird.

Anhand dieses Konzepts soll ein algebraisches Modell zur Komposition von Programmen erstellt werden. Das Ziel ist allerdings, dass die Darstellung der Komposition mittels dieses Modells einheitlich erscheinen muss, d.h. unabhängig davon, wie (einfach oder komplex) das Programm gestaltet wird, welche Artefakte (code- oder non-code-Artefakte) das Programm beinhaltet. Batory et al. konfrontieren mit dieser Herausforderung und entwickeln im Rahmen ihrer Arbeit das Algebraic Hierarchical Equations for Application Design (AHEAD) Modell (s. [BSR04]).

AHEAD basiert auf der Idee von *GenVoca*[28] zur Darstellung der Quelltexte einfachen Programme in *Equation*, und erweitert sie für komplexere Anwendungen, die nicht nur

[28] als zusammengeführtes Projekt aus *Genesis* und *Avoca* (s. [Ape07])

code-Artefakte, sondern auch non-code-Artefakte z.B. UML-Diagramme, Deployment Descriptors, Dokumentationen, Build-Skripte etc. beinhalten.

Die Begriffe aus GenVoca wie *Konstante* (Basis) bzw. *Funktion* (Refinement) werden im AHEAD weiter verwendet. Sie bleiben jedoch im AHEAD-Kontext nicht mehr atomar. Demzufolge können Konstanten bzw. Funktionen mehrere verschachtelte in gemischte Artefakte einkapseln. Diese werden auch als *collective* (vgl. Feature-Modul) bezeichnet, und in einer sogenannten *Inhalts-Hierarchie* dargestellt. Ein Beispiel in Abbildung 2.10 veranschaulicht diese Hierarchiendarstellung im AHEAD.

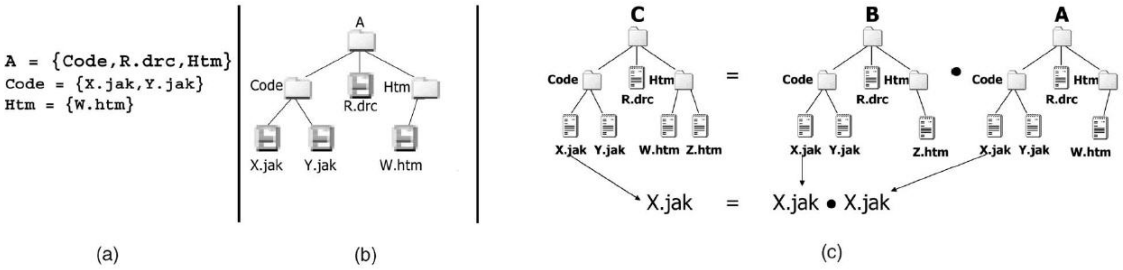


Abbildung 2.10: *collective* und Komposition mit AHEAD.
(a) *collective* A, (b) *Inhalts-Hierarchie*, (c) Komposition. Quelle: [BSR04].

Mit AHEAD ist es dann möglich, SWR in größerem Umfang zu verwenden. Dieses Modell bereitet somit auch die Grundlagen für FOP. Im nächsten Teilabschnitt stellen wir einige Tools und Entwicklungsumgebungen für die Programmierung in FOP vor.

2.3.2 Tools und Entwicklungsumgebungen

FOP-Unterstützungswerkzeuge

AHEAD Tool Suite
Das AHEAD Tool Suite (ATS) liefert die erste Implementierung für AHEAD-Modell. Die gängige hierarchische Struktur des Dateisystems wird zur Konstruktion von *collective* bei dem adaptiert und verwendet. Dieses ist eine Ansammlung von FOP-Kompositions-Tools, welche in Java bzw. Jakarta Tool Suite (JTS) entwickelt wurden (s. [BSR04]), z.B. *composer*, *jampack*, *mixin*, *unmixin*, *jak2java*, *guidsl*, *xak*, etc.

Im Folgenden möchten wir zwei ATS-Tools zum Arbeiten mit *Jak* (*composer*) und XML (*xak*) vorstellen.

Jak ist eine Java-Erweiterung bezüglich z.B. des Refinements (s. [BSR04]). Für die Java Entwicklung mit ATS wird Jak verwendet. Derzeitige ATS-Entwicklung unterstützt Java nur bis zur Version 1.4. In [Bat06] wurden die Einstiegsgrundlagen dazu zusammengestellt, auf das wir hier referenzieren werden. Wir erklären die Nutzung des *composer*-Tools am besten an einem konkreten Beispiel (s. Abbildung 2.11).

Zunächst ist klar zu stellen, dass jedes Feature bzw. Feature-Modul in ein sogenanntes Layer abgebildet wird (hier: *LampBase*, *Dimmer*, *Color*). Das Layer enthält alle Quelltext-Dateien, die das entsprechende Feature definieren (hier: *LampObject*). Der Aufruf von *composer*-Tool löst intern zwei aufeinander folgende Schritte.

Im Schritt 1 wird *jampack* oder *mixin* aufgerufen, um die Jak-Komposition zu erstellen. Nähere Erläuterung zu *jampack* bzw. *mixin* verweisen wir an dieser Stelle auf [Bat06].

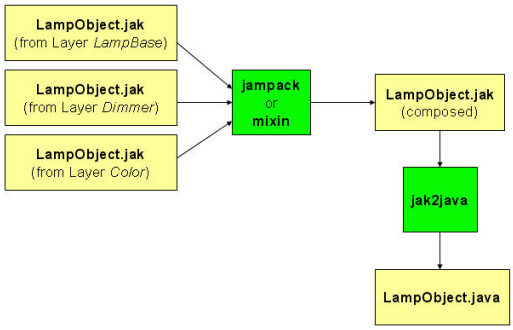


Abbildung 2.11: Komposition mit ATS-*composer*

Im Schritt 2 wird jak2java aufgerufen, um die Jak-Artefakte in die entsprechenden Java-Klassen zu übersetzen.

Listing 2.2 zeigt den Beispiels-Quelltext dazu.

Listing 2.2: LampObject.jak

```
1  //LampBase
2  layer LampBase;
3  public class LampObject {
4    protected int maxIntensity;
5    protected int status;
6    public LampObject(int maxIntensity, int status) {
7      setMaxIntensity(maxIntensity);
8      setStatus(status);
9    }
10   public int getMaxIntensity() {
11     return maxIntensity;
12   }
13   public void setMaxIntensity(int maxIntensity) {
14     this.maxIntensity = maxIntensity;
15   }
16   public int getStatus() {
17     return status;
18   }
19   public void setStatus(int status) {
20     this.status = status;
21   }
22 }
23
24 //Dimmer
25 layer Dimmer;
26
27 public refines class LampObject {
28   protected int precision;
29   public int getPrecision() {
30     return precision;
31   }
32   public void setPrecision(int precision) {
33     this.precision = precision;
34   }
35 }
36
37 //Color
38 layer Color;
39 public refines class LampObject {
40   protected int color;
41   public int getColor() {
42     return color;
43   }
44   public void setColor(int color) {
45     this.color = color;
46   }
```

```
47 }
```

Zum Refinement von XML-Artefakten bietet ATS das *xak*-Tool an. Gemeinsamkeit bzw. Variabilität werden hierbei auch separat erfasst. Derzeit besitzt dieses Tool an sich wiederum zwei unterschiedliche Vorgehensweisen, *xak* bzw. *xak2*.

In [ADT07] wurde *xak2* zur XML-Komposition vorgestellt. Alle XML-Artefakte müssen *xak* als Dateiendung in dieser Vorgehensweise haben. Vordefinierte Attribute (z.B. *xak:artifact*, *xak:feature* oder *xak:module*) werden in der Basisdatei eingesetzt, um die Positionierung des Knotens in der XML-Baumstruktur zu ermöglichen. Im Refinements-Dokument kommen zusätzlich noch Elemente wie *xak:refines*, *xak:keep-content* zum Einsatz (s. Beispiel in [ADT07]).

Aufgrund deren Übersichlichkeit werden wir die zweite Vorgehensweise zur Komposition der XML-Artefakte in dieser Arbeit anwenden. Da müssen nämlich keine zusätzlichen Elemente oder Attribute definiert werden. Die Positionierung des Knotens erfolgt mittels *XPath*.

FeatureHouse

In [AKL09] wurde *FeatureHouse* als Alternative für ATS bzw. dessen Nachfolger vorgestellt. Dieses Tool implementiert auch das AHEAD-Modell von Batory, gibt jedoch einen allgemeinen Ansatz zur Komposition von Software-Artefakten, die in unterschiedlichen Programmiersprachen entwickelt wurden. Momentan kann FeatureHouse bereits mit Java, C#, C, Haskell, JavaCC und XML arbeiten. Im Gegensatz zu ATS, bei dem das Kompositionsvorgehen von den Sprachen der Artefakte abhängig ist, bietet das seit 2007 an der Universität Magdeburg und Universität Passau entwickelte Tool in diesem Fall klaren Vorteil. Dies ist dem sogenannten *FSTComposer* bzw. *Feature Structure Tree* (FST) zu verdanken (s. [AKL09] für mehr Informationen).

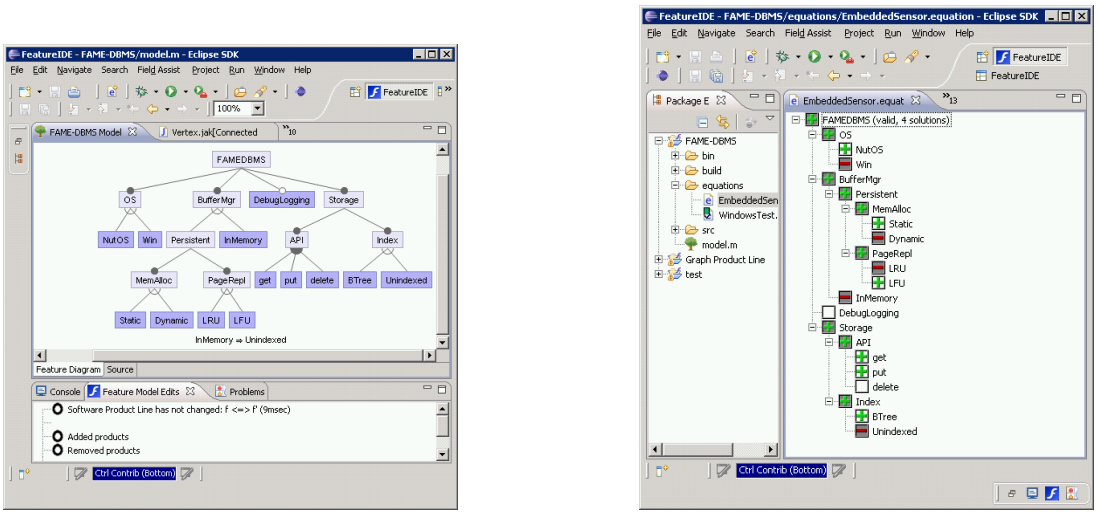
Bezüglich der Programmierung in Java können wir weiterhin folgende Unterschiede zwischen FeatureHouse und ATS feststellen:

- FeatureHouse arbeitet bereits mit JDK 5 zusammen, d.h. es kann z.B. mit Generics-Datentypen oder Annotationen umgehen, was bei ATS noch nicht der Fall ist.
- FeatureHouse unterstützt auch die gewöhnliche Nutzung von Java Packages (vgl. *Layer* in ATS).
- In FeatureHouse wird das Schlüsselwort *original* anstelle von *Super* verwendet, um Methoden oder Konstruktoren der „Superklasse“ aufzurufen. Ferner fällt die Nutzung vom Schlüsselwort *refines* auch weg.
- FeatureHouse verwendet *java*-Endung für die Klassendateien (vgl. *jak*-Endung bei ATS).
- Um weitere Unterschiede zu entdecken bedarf es einer intensiveren Arbeit mit beiden Werkzeugen.

Entwicklungsumgebungen

FeatureIDE ist eine Eclipse-basierte integrierte Entwicklungsumgebung (IDE) zur Programmierung mit FOP. Seit 2004 steht sie in der Entwicklung an der Universität Magdeburg, gemeinsam mit der Universität Passau und der METOP GmbH in Magdeburg.

FeatureIDE unterstützt bereits eine Menge an FOP-Tools u.a. ATS, FeatureHouse, FeatureC++, etc. (s. [KTS+09]). Diese geht den Prozessen im SPL-Framework einigermaßen nach, und erleichtert damit die Entwicklungs- und Wartungsarbeit. Sie bietet außerdem noch Tools zur grafischen Modellierung von Feature-Diagrammen bzw. zur Feature-Selektion für das fertige Produkt (s. Screenshots in Abbildung 2.12).



(a) Feature-Modellierung (b) Feature-Selektion

Abbildung 2.12: FeatureIDE

Weitere Entwicklungsumgebungen, die in diesem Sinne auch in Betracht kommen, sind z.B. *CIDE*[29] (s. [ALMK08]), *pure::variants* der Firma pure-systems GmbH[30] in Magdeburg (s. Technical White Paper von pure::variants auf der Firmenhomepage für mehr Information darüber) oder *XFeature* von P&P Software GmbH in Zürich[31] (s. [PR05]).

2.4 Zusammenfassung

Wir haben in diesem Kapitel bereits einen Überblick über die theoretischen Grundlagen zu Service-Oriented Architecture, Software Product Line bzw. Feature-Oriented Programming gegeben. Im nächsten Kapitel widmen wir uns einem Beispiel, um dann den Aufbau einer SOA-Anwendung aus der praktischen Perspektive zu zeigen.

[29] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/
[30] http://www.pure-systems.com
[31] http://www.pnp-software.com

Kapitel 3

Domotic – Eine Fallstudie

Anzustrebendes Ziel dieser Arbeit ist eine *Evaluierung* des Einsatzes von feature-orientierter Programmierung (FOP) in der Entwicklung von service-orientierten Architekturen (SOA). Im letzten Kapitel wurden bereits die notwendigen Grundlagen dazu vorgestellt. Was aber noch fehlt, ist eine Anwendung aus der Praxis, um die neuen Ansätze wirklich beurteilen zu können. Aufgrund dessen möchten wir uns in diesem Kapitel hauptsächlich mit einer Fallstudie beschäftigen. Die Fallstudie wählt zunächst eine klassische realisierte SOA-Umgebung mit Variabilität aus. Anhand einer Problemanalyse im weiteren Verlauf der Arbeit versuchen wir dann zu zeigen, dass die bisherige Entwicklung einige Schwierigkeiten bezüglich des gegebenen Kontexts bereitet. Am Ende des Kapitels geben wir einen Hinweis auf Alternative, welche diese Probleme bewältigen könnte.

Der erste Teil des Kapitels befasst sich mit der Vorstellung des Fallbeispiels. Es geht hier um *Domotic*-Systeme und deren Anwendungen. Ein solches System besteht in der Regel aus der Zusammenarbeit von mehreren Komponenten. Die Entscheidung, wie das System am Ende aussehen soll, liegt jedoch bei Kunden, die ihre Wünsche durch Auswahl der vorgegebenen Komponenten definieren und konfigurieren. Dies macht *Domotic* so variabel bzw. variationsreich. Aus diesem Grund ist dieses Anwendungsgebiet für unsere Fallstudie gut geeignet.

3.1 Domotic & Co.

Erfindungen wurden meistens gemacht, um das Leben von Menschen leichter und sinnvoller zu gestalten. Viele von denen bezogen auf die Automatisierung im Haushaltsbereich, um „lästige“ Hausarbeiten möglicherweise zu bewältigen. Typische Beispiele hierfür sind elektronische Haushaltsgeräte wie Waschmaschine, die seit ihrer Erfindung anfangs des 20. Jahrhunderts unheimlich viel Arbeit erleichtert gegenüber dem Vorgänger, dem im 19. Jahrhundert beliebten Waschbrett. Heutzutage wird Waschmaschine mit viel erweiterter Funktionalität in fast jedem Haushalt benutzt. Dies zeigt, wie sehr Menschen sich für Erfindungen engagiert, wenn sie seinen Wünschen entsprechen.

„Intelligente„ Geräte im Haushalt funktionieren seither wunderbar getrennt, unabhängig von einander. Man stellt sich nun ein automatisiertes oder zumindest halbautomatisiertes Privathaus vor, wo alle Gerät in einem oder mehreren Netzen miteinander verbunden sind, und in gegenseitiger Abstimmung agieren.

> „Stellen Sie sich vor: Sie sitzen auf der Couch, startbereit für einen Filmabend. Mit einem Tastendruck auf die Fernbedienung gehen die Rollladen herunter, wird die Beleuchtung gedämpft, der Fernseher schaltet auf den richtigen Kanal und die Home Cinema Anlage schaltet in den Dolby-Surround Stand... Für den einen ist Home Automation die Regulierung der Beleuchtung mit derselben Fernbedienung womit der Fernseher bedient wird, für den anderen ist es ein vollständig vom Rechner gesteuertes System, was sich an die eigenen Lebensgewohnheiten anpasst... "

Diese Beschreibung von der Firma Marmitek auf ihrer Homepage[1] bereichert unsere Vorstellung über Heimautomatisierung. Wir sehen, dass erdachte Szenarien nicht nur Vision von einer weit entfernten Zukunft waren. Sie sind der Realität näher als man denkt. Auf der *Domotechnica*[2], einer Domotic-Messe in Köln, kann man sein Wissen über dieses Gebiet jährlich wieder auf neuen Stand bringen.

Nachfolgend möchten wir diese neue Welt der Automatisierung näher vorstellen.

Begriffsdefinition

Domotic ist ein Kunstwort, das aus Zusammensetzung der zwei Begriffe *domus* (lat. *Haus*) und *Automatik* entstanden ist[3]. Zu diesem Thema sind noch viele weitere Bezeichnungen tauglich z.B. Home Automation, Heimautomatisierung, Smart Home, Intelligent Home oder E-Home etc. Wir setzen in dieser Arbeit all diese Begriffe gleich und sprechen einheitlich von *Domotic*. Darunter versteht man die Automatisierung in privaten Wohnhäusern, wobei Energieeffizienz, Komfort und Sicherheit die Schlüsselthemen sind.

Anwendungsgebiete

Beliebte Anwendungsgebiete einer Domotic-Lösung sind (vgl. [Che06], Seite 10) beispielsweise folgende:

- Energieeffizienz
  - Nutzung vom Bewegungsmelder zur automatischen Schaltung der Flurlichte oder der Außenbeleuchtung
  - Überprüfung und Steuerung von Beleuchtungssystem bzw. anderen elektronischen Geräten über Internet
- Komfort
  - Nutzung vom Wassersprinkler bzw. Wettersensor zur automatischen Bewässerung des Gartens bei Trockenheit
  - entfernte Regelung der Heizung zur Aufwärmung des Hauses mit Hilfe der mobilen Geräte

[1] `http://www.marmitek.com/de/`
[2] `http://www.domotechnica.de/`
[3] `http://knowledgerush.com/kr/encyclopedia/Domotics/`

- Sicherheit
  - Windmelder sorgt dafür, dass sich Fenster oder Rollläden bei Sturm oder Regen rechtzeitig schließen.
  - Wassermelder warnt vor Überschwemmung, falls die Waschmaschine einmal auslaufen sollte. Alle Wasserzuflüsse werden in diesem Fall gesperrt.

Diese Liste kann durch eine Vielzahl der Anwendungsfälle noch unendlich lang werden. Wir erkennen hiermit, wie variationsreich ein Domotic-System ist. Für das Fallstudie werden wir später ein konkretes Anwendungsszenario aussuchen.

Software-Markt

Auf der Suche nach einem realen Domotic-System für das Fallbeispiel sind uns folgende Software-Produkte am aktuellen Markt aufmerksam geworden:

- *ActiveHome Pro*[4]
  - Protokoll: X10
  - Plattform: Windows
  - Sprache: C++, Visual Basic, VB Script, JavaScript
- *MisterHouse*[5]
  - Protokoll: X10
  - Plattform: unabhängig
  - Sprache: Perl
- *FreakZ Zigbee*[6]
  - Protokoll: ZigBee
  - Plattform: Unix-basiert
  - Sprache: C
- *OpenRemote Controller*[7]
  - Protokoll: X10, KNX, Insteon, ZigBee
  - Plattform: unabhängig
  - Sprache: Java, C
- *Shion*[8]

[4] http://www.activehomepro.com
[5] http://misterhouse.sourceforge.net
[6] http://www.freaklabs.org/freakz/v0_7/html/index.html
[7] http://www.openremote.org/display/orb/OpenRemote+Controller+Software
[8] http://www.audacious-software.com/products/shion/

  - Protokoll: X10
  - Plattform: Mac
  - Sprache: AppleScript

- *Indigo*[9]
  - Protokoll: X10, Insteon
  - Plattform: Mac
  - Sprache: AppleScript

Es sind sicherlich noch viel mehr Software auf diesem Gebiet angeboten. Wir haben zur Kenntnis genommen, dass es außer den kommerziellen Produkten auch einige quelloffene (open source) Software gibt. Allerdings sind wir erstens nicht davon überzeugt, dass diese problemlos, ohne weiteres adaptieren lassen. Zweitens ist es uns kein System aufgefallen, dass dieses eine SOA-Umgebung besitzt. Aus diesen Gründen haben wir uns entschieden, selbst ein eigenes, zum Demonstrationszweck aber halbwegs reales System zu konzipieren und dann implementieren. Das erfolgt gleich im nächsten folgenden Abschnitt des Kapitels.

[9] `http://www.perceptiveautomation.com/indigo/index.html`

3.2 Der Demonstrator

Als nächstes betrachten wir ein typisches Szenario für das Beleuchtungsystem der Domotic-Lösung, um dessen konstruktiven Aufbau bzw. Funktionsweise zu untersuchen.

3.2.1 Ein Szenario des Beleuchtungssystems

In Wirklichkeit besteht eine solche Domotic-Anwendung grundsätzlich aus Sensoren, Aktuatoren und einem oder mehreren Controllers. Der Controller ermöglicht eine zentrale Steuerung aller Lampen, Lichte im Haus. Er kann fest an der Wand z.B. im Wohnzimmer eingebaut werden, besitzt noch einen kleinen Funkcontroller (Fernbedienung), und eventuell noch ein Front-End im mobilen Gerät.

Über ein Bussystem werden Signaldaten zwischen Sensoren und Controller bzw. zwischen Controller und Aktuatoren übermittelt. Ein Aktuator wird hier zwischen den Lampen und der Netzspannung eingebaut. Aktuatoren sind an das Bussystem angeschlossen und erhalten Signaldaten von diesem. Diese Daten stammen entweder direkt von einem Sensor (Schalter, Helligkeitssensoren oder Bewegungsmelder) oder indirekt von einem Controller (vgl. [PBvdL05], Seiten 40–52).

Bisher hat sich im Domotic-Bereich noch kein Standard bezüglich Bussysteme am Markt durchsetzen können. Aktive Protokolle sind momentan z.B. X10, CEBus (in den USA), EHS, EIB/KNX (in Europa), LonWorks, ZigBee, Z-Wave, etc. (vgl. [Osi08]).

Je nach Bedürfnissen können die Bewohner (Nutzer des Systems) dieses typische Szenario selbst konstruieren. Man kann hierbei z.B. Variabilität der Controller definieren, indem man die vorhandenen Sensoren bzw. Aktuatoren zur Auswahl stellt, welche der eine Controller dann steuern kann oder soll.

Unser fiktives Domotic-System besteht demzufolge auch aus Interaktion der drei Komponenten: Aktuatoren, Sensoren und einem zentralen Controller. Zum Demonstrationszweck ist es wichtig, die Variabilität in SOA-Umgebungen bzw. konkret in diesem System aufzuzeigen. Solange sonstige Detailliertheit keinen Einfluss diesbezüglich bewirkt, ist demzufolge irrelevant und wird außer Acht gelassen. Aufgrund dessen werden wir auch einige Restriktionen zur Systemvereinfachung annehmen.

Technische Einschränkungen

- Wir verzichten auf hardwareseitige Berücksichtigung und werden uns folgend nur auf die softwaremäßige Betrachtung unabhängig von den gängigen Protokollen konzentrieren.

- Alle Server-Komponenten werden nur auf einem Applikationsserver deployt. In der Realität können sie verteilt auf mehreren zur Performanzverbesserung installiert werden.

- Es wird kein Datenbanksystem zum Speichern der Anfangs- bzw. Zwischenzustände von den Geräten benutzt. Wir werden lediglich XML-basierte Konfigurationsdateien verwenden, um die Geräte zu registrieren und ihre Anfangszustände zu initialisieren.

- Wir trennen die Präsentationsschicht in der drei-schichtigen Client/Server-Architektur vom Server und lagern sie auf der Seite vom Client in Form einer einfachen Applikation. Diese Client-GUI kommuniziert mit dem Server auch über Web Services.
- Wir benutzen den *Top-down*-Ansatz zur Web Services Implementierung des SOA-Kontexts.
- Trotz dessen Wichtigkeit im allgemeinen Domotic-Szenario kümmern wir uns hier nicht um den Sicherheitsaspekt des Systems.

Hilfsframeworks

Die zur Verwendung kommenden technologischen Hilfsmittel sind:

- Java EE 5[10]
- Entwicklungsumgebung *Eclipse Ganymede*[11] für Enterprise Anwendungen
- Der kostenfreie Applikationsserver Glassfish v2.1[12] von Sun Microsystems
- Sun Metro v1.4[13] mit JAX-WS RI v2.1.5 und JAXB RI v2.1.9

Implementierung

Im Folgenden werden wir die Implementierung vom *Demonstrator* betrachten. Damit diese übersichtlicher erscheinen wird, teilen wir die gesamte Entwicklung in zwei Teilprobleme ein.

- *Variabilität bei Lampen* – hier kümmern wir uns nur um die Entwicklung verschiedenen Lampenmodule (Varianten), die später zur Kontruktion des Controllers verwendet werden können.
- *Variabilität bei Controller* – das eigentliche Gesamtsystem wird hier der Fokus. Hier definieren wir, aus welchen Komponennten in Kombination mit dem Controller ein solches System besteht. Interessant hierbei ist die Auswahlmöglichkeit auf der Seite des Controllers.

3.2.2 Implementierung: Variabilität bei Lampen

Beschreibung

Es werden vier Lampenvarianten (Aktuatoren) angeboten, die für das Beleuchtungssystem im Haus benutzt werden können. Diese sind folgende:

[10] `http://java.sun.com/javaee/`
[11] `http://www.eclipse.org/ganymede/`
[12] `https://glassfish.dev.java.net/downloads/v2.1-b60e.html`
[13] `https://metro.dev.java.net/1.4/`

- *LampSimple* – Das ist der einfachste Aktuator im System. Mit dessen Einsatz kann die angeschlossene Lampe entweder an- oder ausgeschaltet werden. Diese {an- und ausschalten}-Funktionen bilden auch die Basisfunktionalität aller Aktuatoren im Allgemeinen.
- *LampDimmer* – In manchen Situationen ist es wünschenswert, den Helligkeitswert einer Lampe regulieren zu können, z.B. beim Lesen, Fernsehen oder Abendessen. Das heißt, man möchte die Lichtstärke gegebenenfalls hoch- (*heller*) oder herunter- (*dunkler*) dimmen können. Mit *LampDimmer* lässt sich dies verwirklichen.
- *SimpleColor* – Angenommen, der Bewohner würde gerne die Lichtfarbe in manchen Zimmern z.B. im Bad, in der Küche, oder auch im Schlafzimmer wechseln können. So entwickeln wir *SimpleColor*. Dieser Aktuator enthält weiterhin die Standardfunktionen von *LampSimple* und zusätzlich noch eine Funktion zum Wechseln der Farbe dazu.
- *DimmerColor* – Einige Kunden hätten gerne die beiden Extra-Funktionen doch gleich in einem Aktuator, damit sie eine bestimmte Umgebung verzaubern können. Deshalb kommt *DimmerColor* hier zum Einsatz.

Die Anwendungsfälle (use cases) der jeweiligen Lampenvariante werden in Abbildung 3.1 und 3.2 noch mal veranschaulicht.

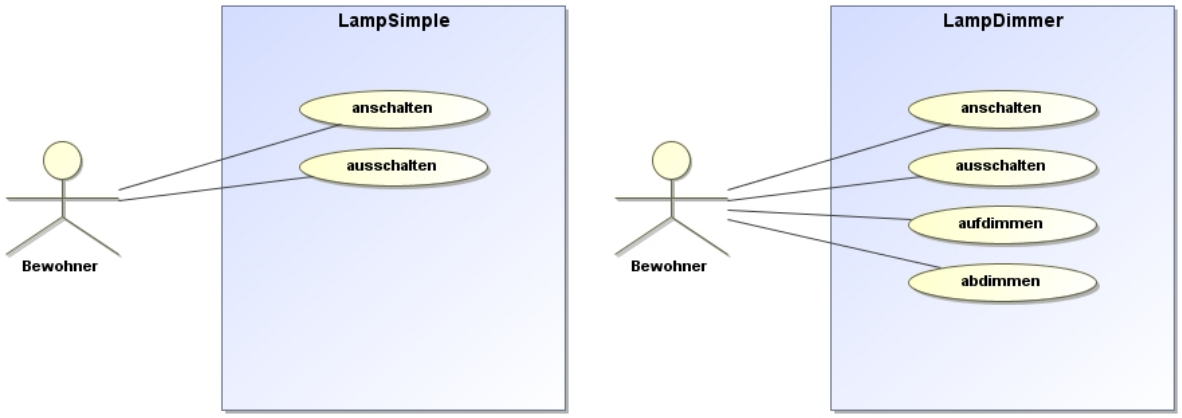
Service-Identifikation

Anhand der funktionalen Beschreibung (s. oben) sowie der entsprechenden Anwendungsfälle von einzelnen Aktuatoren können wir an dieser Stelle insgesamt folgende Funktionen bestimmen:

- *turnOn* – schaltet die Lampe an.
- *turnOff* – schaltet die Lampe aus.
- *dimUp* – dimmt hoch.
- *dimDown* – dimmt runter.
- *changeColor* – wechselt die Farbe.

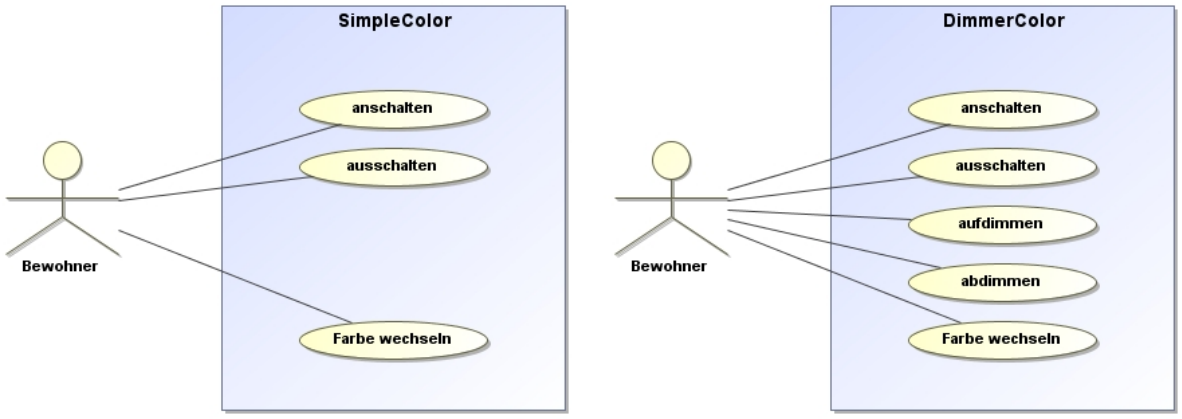
In SOA spielen Lampenaktuatoren die Rolle eines Service-Anbieters, wobei deren Konsument der am System angeschlossene Controller ist. Die oben aufgelisteten Funktionen sind entsprechende Web Services Operationen, die wir implementieren dann müssen.

Zunächst betrachten wir die Abbildung 3.3. Damit wir die Web Services Operationen in allen Lampenvarianten sinnvollerweise wiederverwenden können, entwickeln wir eine Basisplattform für sie. Dort sollen alle benötigten Operationen implementiert werden. Zur Nutzung dieser Operationen wird ein Service-Vertrag (Service-Beschreibung) verfasst, der die Operationen bzw. die Nutzungsregel näher beschreibt. Beispielsweise bezieht der Service-Vertrag *LampSimple* zwei Operationen {turnOn, turnOff} auf der Plattform.



(a) LampSimple (b) LampDimmer

Abbildung 3.1: Use Case Diagramme der Lampenmodule (1)



(a) SimpleColor (b) DimmerColor

Abbildung 3.2: Use Case Diagramme der Lampenmodule (2)

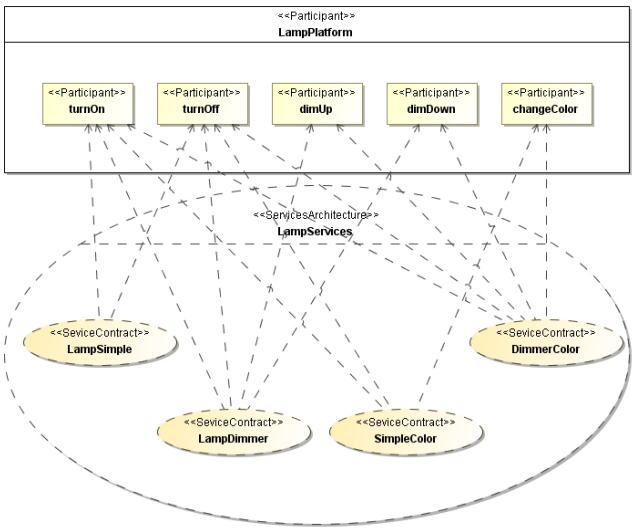


Abbildung 3.3: Service-Identifikation bei Lampen

Implementierung

Wir beginnen der Implementierung zunächst mit der Entwicklung der Basisplattform. Danach wenden wir den *Top-down*-Ansatz zur Web Services Entwicklung einzelnen Lampenvarianten an.

Entwicklung der Plattform
Zur Basisplattform gehören außer den benötigten Funktionen noch alle Lampenobjekte als objekt-orientierte Abbildung der realen Lampen im System. Die Beziehung zwischen diesen Objekte modellieren wir in einem Klassendiagramm (s. Abbilung 3.4).

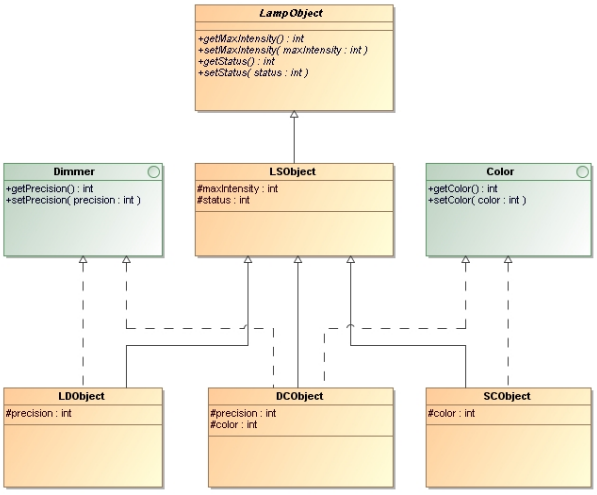


Abbildung 3.4: Beziehung zwischen Lampenvarianten

Einfache Lampen (*LSObject*) besitzen demzufolge zwei Attribute: *maxIntensity* (maximaler Helligkeitswert einer Lampe) und *status* (aktueller Lampenzustand).

Lampenaktuatoren, die dimmen kann, bekommen zusätzlich noch ein *precision*-Attribut (Genauigkeit des Dimmens). Dies gibt den Stufenunterschied bezüglich der Helligkeit zwischen zwei Dimm-Aktionen an. Je kleiner dieser Wert ist, desto feiner wird der Aufhellungs- bzw. Verdunkelungsprozess. *LDObject* und *DCObject* besitzen jeweils dieses Attribut, da sie *Dimmer*-Funktion unterstützen.

DCObject hat zusätzlich noch das *color*-Attribut, weil die Lichtfarbe von dieser Lampenart auch wechselbar ist. Das ist bei *SCObject* auch der Fall.

Das *Dimmer*- bzw. *Color*-Verhalten haben wir deswegen in Schnittstellen (interface) modelliert, weil Java keine Mehrfachvererbung erlaubt. Das Problem an dieser Stelle ist, dass wir die gleichen Codes z.B. in *LDObject* und *DCObject* zur Implementierung gleichen Sachverhalten (*getPrecision* und *setPrecision*) wiederholen müssen (Stichwort: Code-Replikation) (s. Listings 3.1 und 3.2). Auf dieses Problem werden wir bei der Problemanalyse im späteren Teil des Kapitels noch mal zurückkommen.

Listing 3.1: LDObject.java

```
1 package de.lampplatform.objects;
2
3 public class LDObject extends LSObject implements Dimmer {
4
5   protected int precision;
6
```

```
7   public LDObject(int maxIntensity, int status, int precision) {
8     super(maxIntensity, status);
9     setPrecision(precision);
10  }
11
12  public int getPrecision() {
13    return precision;
14  }
15
16  public void setPrecision(int precision) {
17    this.precision = precision;
18  }
19
20 }
```

Listing 3.2: DCObject.java

```
1  package de.lampplatform.objects;
2
3  public class DCObject extends LSObject implements Color, Dimmer {
4
5    protected int precision;
6    protected int color;
7
8    public DCObject(int maxIntensity, int status, int precision, int color) {
9      super(maxIntensity, status);
10     setPrecision(precision);
11     setColor(color);
12   }
13
14   public int getPrecision() {
15     return precision;
16   }
17
18   public void setPrecision(int precision) {
19     this.precision = precision;
20   }
21
22   public int getColor() {
23     return color;
24   }
25
26   public void setColor(int color) {
27     this.color = color;
28   }
29 }
```

Web Services sind zustandslos, d.h. sie merken nicht, welche Anfragen sie in der Vergangheit zur Verarbeitung bekommen, sowie welche Ergebnisse sie zurückgeliefert haben. Für unseren Demonstrator bedeutet, dass z.B. nach einem Absturz des Client-Programms (Controller-GUI) die letzten Lampenzustände nicht gemerkt werden. Alle im System befindlichen Lampen erhalten wieder die initialisierten Zustände am Anfang. obwohl die letzte Zustandskonfiguration durch den Benutzer erfolgreich war. Das ist so nicht realisch.

Da wir keine Datenbanken zum Speichern der Lampenzustände verwenden (vgl. Restriktionen des Demonstrators ganz anfangs der Implementierung), sind uns nur noch zwei Lösungsvorschläge entgegen gekommen, um dieses Problem zu umgehen. Der eine Vorschlag ist, dass man die Zustände nach jeder Änderung auf Dateien zurückschreibt. Diese Lösung scheint uns schon auf den ersten Blick zu aufwendig. Deshalb haben wir uns für die zweite Vorgehensweise entschieden. Wir verwenden das Entwurfsmuster *Singleton* (s. [GHJV95]) zur Implementierung der zustandsbehafteten Bean-Objekte. Ein bedeutender Nachteil dieses Vorgehens, den wir doch in Kauf nehmen müssen, ist der Verzicht

auf Vererbung dieser Objekte, da sie nur einmal während ihrer gesamten Laufzeit instanziiert werden dürfen. Dies ist akzeptabel, da *LampBean* alle Lampenobjekte auf der Plattform enthält, und bedarf keiner Spezifizierung (s. Listing 3.4).

In Listing 3.3 zeigen wir ein Beispiel für Konfigurationsdateien, die wir zur Installation der Lampen ins System bzw. zur Initialisierung deren Zustände benutzen. Wir haben hier z.B. 5 Lampen vom Typ *LS*, 3 vom Typ *LD*, etc. Der Anfangszustand einer Lampe wird in *status* angegeben (*0 = ausgeschaltet*). Die Lampen jeweiliger Art sind außerdem nummeriert zum Zweck der Identifikation im System. Der Schlüssel wird in diesem Fall durch Kombination von *type* und *id* gebildet (z.B. *LS1*, *LD3*, *DC2*, etc.)

Listing 3.3: Konfiguration für alle Lampen im System

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

  <lamp type="LS" id="1" maxIntensity="40" status="0"/>
  <lamp type="LS" id="2" maxIntensity="60" status="0"/>
  <lamp type="LS" id="3" maxIntensity="20" status="0"/>
  <lamp type="LS" id="4" maxIntensity="30" status="0"/>
  <lamp type="LS" id="5" maxIntensity="50" status="0"/>

  <lamp type="LD" id="1" maxIntensity="20" status="0" precision="5"/>
  <lamp type="LD" id="2" maxIntensity="30" status="0" precision="10"/>
  <lamp type="LD" id="3" maxIntensity="60" status="0" precision="15"/>

  <lamp type="SC" id="1" maxIntensity="20" status="0" color="2" />
  <lamp type="SC" id="2" maxIntensity="30" status="0" color="1" />
  <lamp type="SC" id="3" maxIntensity="50" status="0" color="3" />

  <lamp type="DC" id="1" maxIntensity="20" status="0" precision="5" color="5" />
  <lamp type="DC" id="2" maxIntensity="30" status="0" precision="10" color="7" />

</lamps>
```

Listing 3.4: LampBean.java

```
package de.lampplatform.beans;

import de.lampplatform.objects.*;
...
public class LampBean {
  public static enum LampType {LS, LD, SC, DC}
  private static LampBean bean;
  private final String pathConfigXML = "config-lamps.xml";
  private Hashtable<String, LampObject> devices = new Hashtable<String, LampObject>();

  private LampBean() { //singleton
    NodeList nl = null;
    try {
      nl = loadXML(pathConfigXML);
    ...
    if (nl != null) {
      for (int i=0; i<nl.getLength(); i++) {
        Element lamp = (Element) nl.item(i);
        String type = lamp.getAttribute("type");
        String id = type + lamp.getAttribute("id");
        LampObject obj = null;
        if (type.equals(LampType.LS.toString())) {
          obj = new LSObject(
              Integer.valueOf(lamp.getAttribute("maxIntensity")).intValue(),
              Integer.valueOf(lamp.getAttribute("status")).intValue());
        }
        if (type.equals(LampType.LD.toString())) {
          obj = new LDObject(
              Integer.valueOf(lamp.getAttribute("maxIntensity")).intValue(),
              Integer.valueOf(lamp.getAttribute("status")).intValue(),
```

```
31                Integer.valueOf(lamp.getAttribute("precision")).intValue());
32          }
33          if (type.equals(LampType.SC.toString())) {
34            ...
35          }
36          if (type.equals(LampType.DC.toString())) {
37            ...
38          }
39          if (obj != null) {
40            setObject(id, obj);
41          }
42        }
43      }
44    }
45
46    public LampType checkType(String id) {
47      String type = id.substring(0, 2);
48      if (type.equals(LampType.LS.toString()))
49        return LampType.LS;
50      if (type.equals(LampType.LD.toString()))
51        return LampType.LD;
52      if (type.equals(LampType.SC.toString()))
53        return LampType.SC;
54      if (type.equals(LampType.DC.toString()))
55        return LampType.DC;
56      return null;
57    }
58
59    public boolean checkDimmer(String id) {
60      String type = id.substring(0, 2);
61      if (type.equals(LampType.LD.toString()) || type.equals(LampType.DC.toString()))
62        return true;
63      return false;
64    }
65
66    public boolean checkColor(String id) {
67      String type = id.substring(0, 2);
68      if (type.equals(LampType.SC.toString()) || type.equals(LampType.DC.toString()))
69        return true;
70      return false;
71    }
72    ...
73  }
```

Nun werden die benötigten Operationen der Lampenaktuatoren implementiert (s. Listings 3.5 und 3.6). Die Rückgabewerte der allen Operationen sind vom Typ *integer*. Anhand von diesen Werten können wir entstandene Fehler eventuell klassifizieren, dokumentieren und auswerten.

Listing 3.5: LampPlatform.java

```
1  package de.lampplatform.webservice;
2
3  public interface LampPlatform {
4    //basic functions
5    public int turnOn(String lampID);
6    public int turnOff(String lampID);
7    public int dimUp(String lampID);
8    public int dimDown(String lampID);
9    public int changeColor(String lampID, int color);
10
11   //monitoring functions
12   public int getStatus(String lampID);
13   public int getColor(String lampID);
14 }
```

Listing 3.6: LampPlatformServiceImpl.java

```
1  package de.lampplatform.webservice;
2
3  import de.lampplatform.beans.LampBean;
4  import de.lampplatform.beans.LampBean.LampType;
5  import de.lampplatform.objects.*;
6
7  public class LampPlatformServiceImpl implements LampPlatform {
8    LampBean bean = LampBean.getIntance();
9    LampObject obj;
10
11   private void setLampObject(String lampID) {
12     if (bean.checkType(lampID) == LampType.LS) {
13       obj = (LSObject) bean.getObject(lampID);
14     }
15     if (bean.checkType(lampID) == LampType.LD) {
16       obj = (LDObject) bean.getObject(lampID);
17     }
18     if (bean.checkType(lampID) == LampType.SC) {
19       obj = (SCObject) bean.getObject(lampID);
20     }
21     if (bean.checkType(lampID) == LampType.DC) {
22       obj = (DCObject) bean.getObject(lampID);
23     }
24   }
25
26   public int turnOn(String lampID) {
27     int result = 0;
28     setLampObject(lampID);
29     if (obj == null) {
30       return -1;
31     }
32     obj.setStatus(obj.getMaxIntensity()); //turnOn
33     return result;
34   }
35   ...
36   public int dimUp(String lampID) {
37     int result = 0;
38     setLampObject(lampID);
39     if (obj == null) {
40       return -1;
41     }
42     if (bean.checkDimmer(lampID)) { //check compatibility
43       int newIntensity = obj.getStatus() + ((Dimmer) obj).getPrecision();
44       if (newIntensity < obj.getMaxIntensity()){
45         obj.setStatus(newIntensity);
46       } else {
47         obj.setStatus(obj.getMaxIntensity());
48       }
49       return result;
50     }
51     return -2;
52   }
53   ...
54   public int changeColor(String lampID, int color) {
55     int result = 0;
56     setLampObject(lampID);
57     if (obj == null) {
58       return -1;
59     }
60     if (bean.checkColor(lampID)) { //check compatibility
61       ((Color) obj).setColor(color);
62       return result;
63     }
64     return -2;
65   }
66   ...
67 }
```

Wir können an dieser Stelle weiter mit der Entwicklung von *Top-down*-Web Services starten. Das heißt, wir beschäftigen uns zunächst mit der Service-Beschreibung (WSDL-

Dokument).

Service-Beschreibung
Wie ein WSDL-Dokument strukturiert ist, wurde bereits im letzten Grundlagen-Kapitel vorgestellt.

Als erstes kümmern wir uns um die Definition der benötigten Datentypen (z.B. für *LampSimple*). Der Standard *XML-Schema* wird für die Beschreibung der Datenstruktur verwendet. Zugunsten der Übersichtlichkeit, sowie der besseren Pflege trennen wir diesen Teil von der eigentlichen WSDL-Datei. Listing 3.7 zeigt einen Ausschnitt aus dem Schema.

Listing 3.7: LampSimpleSchema.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema...>
  <xs:element name="turnOn" type="tns:turnOn"/>
  <xs:complexType name="turnOn">
    <xs:sequence>
      <xs:element name="lampID" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="turnOnResponse" type="tns:turnOnResponse"/>
  <xs:complexType name="turnOnResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="turnOff" type="tns:turnOff"/>
  <xs:complexType name="turnOff">
    <xs:sequence>
      <xs:element name="lampID" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="turnOffResponse" type="tns:turnOffResponse"/>
  <xs:complexType name="turnOffResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Wir spezifizieren dort vier Datentypen *turnOn*, *turnOnResponse*, *turnOff* und *turnOffResponse*, die später in der WSDL-Datei weiter verwendet werden. *turnOn* und *turnOff* enthalten ein einziges Element vom Typ *String*, nämlich *lampID* (zur Identifikation der zusteuernden Lampe). *turnOnResponse* und *turnOffResponse* besitzen ein *integer*-Element *return* (um den Erfolg des Operations-Aufrufs zu signalisieren).

Im abstrakten Teil der WSDL-Datei wird das Schema zunächst innerhalb des `types`-Elements importiert. Dann definieren wir in den `message`-Elementen die allen Nachrichten, die zwischen Anbieter und Konsumenten ausgetauscht werden. Zu jeder Nachricht wird gegebenenfalls ein bereits im Schema spezifizierter Datentyp als Parameter oder Rückgabewert zugemappt. Im `portType` werden die Operation *turnOn* bzw. *turnOff* mit entsprechenden In- und Outputnachrichten definiert (s. Listing 3.8).

Listing 3.8: LampSimple.wsdl

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

```
4       <xsd:schema>
5         <xsd:import schemaLocation="LampSimpleSchema.xsd".../>
6       </xsd:schema>
7     </types>
8
9     <message name="turnOn">
10      <part name="parameters" element="tns:turnOn"/>
11    </message>
12    <message name="turnOnResponse">
13      <part name="result" element="tns:turnOnResponse"/>
14    </message>
15    <message name="turnOff">
16      <part name="parameters" element="tns:turnOff"/>
17    </message>
18    <message name="turnOffResponse">
19      <part name="result" element="tns:turnOffResponse"/>
20    </message>
21
22    <portType name="LampSimple">
23      <operation name="turnOn">
24        <input message="tns:turnOn"/>
25        <output message="tns:turnOnResponse"/>
26      </operation>
27      <operation name="turnOff">
28        <input message="tns:turnOff"/>
29        <output message="tns:turnOffResponse"/>
30      </operation>
31    </portType>
32
33    <binding name="LampSimplePortBinding" type="tns:LampSimple">
34      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
35      <operation name="turnOn">
36        <soap:operation soapAction=""/>
37        <input>
38          <soap:body use="literal"/>
39        </input>
40        <output>
41          <soap:body use="literal"/>
42        </output>
43      </operation>
44      <operation name="turnOff">
45        <soap:operation soapAction=""/>
46        <input>
47          <soap:body use="literal"/>
48        </input>
49        <output>
50          <soap:body use="literal"/>
51        </output>
52      </operation>
53    </binding>
54
55    <service name="LampSimple">
56      <port name="LampSimplePort" binding="tns:LampSimplePortBinding">
57        <soap:address location="http://localhost:8080/LampPlatform/LampSimple" />
58      </port>
59    </service>
60  </definitions>
```

Es folgt für jede Operation im konkreten Teil eine konkrete Bindungs-Beschreibung, wie der SOAP-Body dann aussehen soll. Ebenfalls wird eine eindeutige Service-Endpoint-Adresse im `port`- bzw. `service`-Element bestimmt (vgl. Kapitel 2).

Um auch die WSDL-Dateien für restliche Lampenaktuatoren (*LampDimmer*, *SimpleColor*, *DimmerColor*) zu erstellen, wiederholen wir diesen ganzen Vorgang jeweils von Anfang an, oder benennen wir die bereits erstellte WSDL-Datei um, und passt sie an. Als Beispiel zeigt Listing 3.9 die WSDL-Datei von *SimpleColor*. Dort wird die Operation *changeColor* für den *SimpleColor* entsprechend ergänzt.

Listing 3.9: SimpleColor.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions...>
  <types>
    <xsd:schema>
      <xsd:import schemaLocation="SimpleColorSchema.xsd".../>
    </xsd:schema>
  </types>

  <message name="turnOn">
    <part name="parameters" element="tns:turnOn"/>
  </message>
  <message name="turnOnResponse">
    <part name="result" element="tns:turnOnResponse"/>
  </message>
  ...
  <message name="changeColor">
    <part name="parameters" element="tns:changeColor"/>
  </message>
  <message name="changeColorResponse">
    <part name="result" element="tns:changeColorResponse"/>
  </message>

  <portType name="SimpleColor">
    <operation name="turnOn">
      <input message="tns:turnOn"/>
      <output message="tns:turnOnResponse"/>
    </operation>
    ...
    <operation name="changeColor">
      <input message="tns:changeColor"/>
      <output message="tns:changeColorResponse"/>
    </operation>
  </portType>

  <binding name="SimpleColorPortBinding" type="tns:SimpleColor">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
      <operation name="turnOn">
        ...
      </operation>
      <operation name="turnOff">
        ...
      </operation>
      <operation name="changeColor">
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
  </binding>

  <service name="SimpleColor">
    <port name="SimpleColorPort" binding="tns:SimpleColorPortBinding">
      <soap:address location="http://localhost:8080/LampPlatform/SimpleColor" .../>
    </port>
  </service>
</definitions>
```

Dieses Vorgehen ist ziemlich mühsam, und zeigt keine Möglichkeit, das Wiederverwendungspotenzial zu nutzen. Hier sehen wir, dass es momentan keinen vorhandenen Mechanismus gibt, um diese Anpassung etwas dynamsicher zu gestalten. In der Problemanalyse wird dies auch wieder zum Gespräch kommen.

Wir führen nun das Tool `wsimport` vom Glassfish aus, um notwendige code-Artefakte (Java-Quelltext) entsprechend den WSDL-Dokumenten generieren zu lassen. Dann

können wir uns weiter mit der Service Implementation Bean (SIB) (Web Services Anwendungslogik-Implementierung) beschäftigen.

Service-Implementierung
Die Erkennung der eigentlichen *SIB* erfolgt durch *Annotations*-Angabe `@WebService` vor der Klassendeklaration. Das Service-Interface wird dort durch das `endpointInterface`-Attribut auch lokalisiert. Die Implementierung an sich ist an dieser Stelle einfach, da die einzelne Logik schon in der Plattform implementiert wurde (s. Listing 3.10).

Listing 3.10: LampDimmerServiceImpl.java

```
1  package de.lampdimmer.webservice;
2
3  import javax.jws.WebService;
4
5  import de.lampplatform.webservice.LampServiceServiceImpl;
6
7  @WebService(endpointInterface = "de.lampdimmer.webservice.LampDimmer",
8      serviceName = "LampDimmer",
9      portName = "LampDimmerPort")
10 public class LampDimmerServiceImpl {
11   LampPlatform lamp = new LampPlatformServiceImpl();
12   public int turnOn(String lampID) {
13     return lamp.turnOn(lampID);
14   }
15   public int turnOff(String lampID) {
16     return lamp.turnOff(lampID);
17   }
18   public int dimUp(String lampID) {
19     return lamp.dimUp(lampID);
20   }
21   public int dimDown(String lampID) {
22     return lamp.dimDown(lampID);
23   }
24 }
```

Nach dem Deployment auf den Glassfish-Applikationsserver stehen die Web Services der Lampenvarianten dem Controller zur Verfügung. Die Entwicklung des Controllers wird im nächsten Teilabschnitt beschrieben. Dort wird auch gezeigt, wie man vorhandene Web Services konsumiert und verwendet.

3.2.3 Implementierung: Variabilität bei Controller

Beschreibung

Wir erinnern uns zunächst an das Szenario, welches wir in dieser Fallstudie aufbauen möchten. Bisher sind die Lampenaktuatoren definiert. Es fehlen noch Sensoren und Controller dazu.

Die Sensoren
Bevor wir uns auf die Entwicklung der Controller-Varianten konzentrieren können, möchten wir die Entwicklung der Sensoren hier kurz vorwegnehmen. Wir werden *Helligkeitssensoren* (*IntensitySensor*) für das gesamte Beleuchtungssystem einsetzen.

Helligkeitssensor soll die Intensität der Lichtstärke von einer Umgebung observieren (beobachten), und informiert den angeschlossenen Controller über deren Änderung (fiktive Zustände: *1* für *hell* bzw. *2* für *dunkel*). Darauf reagiert der Controller, indem er die bereits in der Konfiguration registrierten Lampen dortiger Umgebung automatisch

steuern. Falls es dunkel ist, schaltet er alle Lampen an. Ansonsten bleiben die Lampen aus. Dieser Sachverhalt erklärt sich in der Abbildung 3.5.

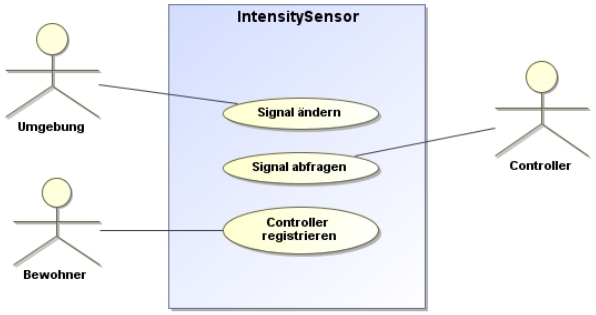


Abbildung 3.5: Use Case Diagramm des Sensors

Ein Sensor dieser Art stellt folgende Operationen zur Verfügung:

- *setController* – registriert den aktiven Controller, der Signale vom Sersor übermittelt bekommen wird.
- *getSignal* – zur Abfrage des Sensor-Signals
- *setSignal* – Theoretisch wechseln Sensoren als Hardware automatisch ihr Signal durch Veränderung der Naturbedingung in der Umgebung, wo er eingesetzt wird. Für unser fiktives Fallbeispiel erlauben wir manuelle Änderung über den Controller mittels dieser Operation. Auch diesbezüglich zeigt keine nachteilige Wirkung auf unser Vorhaben.

Durch letzte Annahme entsteht ein gegenseitiger Austausch von Informationen zwischen Controller und Sensor, da der Sensor wiederum Operation von Controller aufruft, um diesen über die veränderte Umgebung zu informieren.

Die Implementierung der Web Services von *IntensitySensor* an sich ist nichts spannendes. Deshalb ist es hier nicht zwingend erforderlich, sie detaillierter zu erläutern. Wir zeigen einen Ausschnitt aus dessen SIB-Klasse in Listing 3.11.

Listing 3.11: IntensitySensorServiceImpl.java

```
1  package de.intensitysensor.webservice;
2
3  import javax.jws.WebService;
4  import de.intensitysensor.beans.ISBean;
5  import de.intensitysensor.objects.ISObject;
6  ...
7  @WebService(endpointInterface = "de.intensitysensor.webservice.IntensitySensor",
8      serviceName = "IntensitySensor",
9      portName = "IntensitySensorPort")
10 public class IntensitySensorServiceImpl {
11   ISBean bean = ISBean.getInstance();
12
13   public void setController(int controller) {
14     bean.setRegController(controller);
15   }
16
17   public int getSignal(String id) {
18     if (id.startsWith(ISBean.ID)) { //check compatibility
19       ISObject obj = bean.getObject(id);
```

```
20        if (obj == null) {
21          return -1;
22        }
23        return obj.getSignal();
24      }
25      return -2;
26    }
27
28    public int setSignal(String id, int signal) {
29      if (id.startsWith(ISBean.ID)) { //check compatibility
30        ISObject obj = bean.getObject(id);
31        if (obj == null) {
32          return -1;
33        }
34        if (obj.getSignal() != signal) {
35          obj.setSignal(signal);
36          switch (bean.getRegController()) { //inform approprivate controller about new
                signal
37          case 1:
38            bean.getControllerBasicService().update(id, signal);
39            break;
40          case 2:
41            bean.getControllerPremiumService().update(id, signal);
42            break;
43          default:
44            break;
45          }
46          return 0;
47        }
48      }
49      return -2;
50    }
51
52  }
```

Jetzt können wir uns der Entwicklung der Controller widmen.

Controller-Varianten

Die Variabilität bei Controller lässt sich bei Auswahl der vorhandenen Lampenvarianten definieren. *IntensitySensor* ist dabei ein Muss für jeden Controller. Das gesamte Beleuchtungsszenario wird in Abbilung 3.6 veranschaulicht.

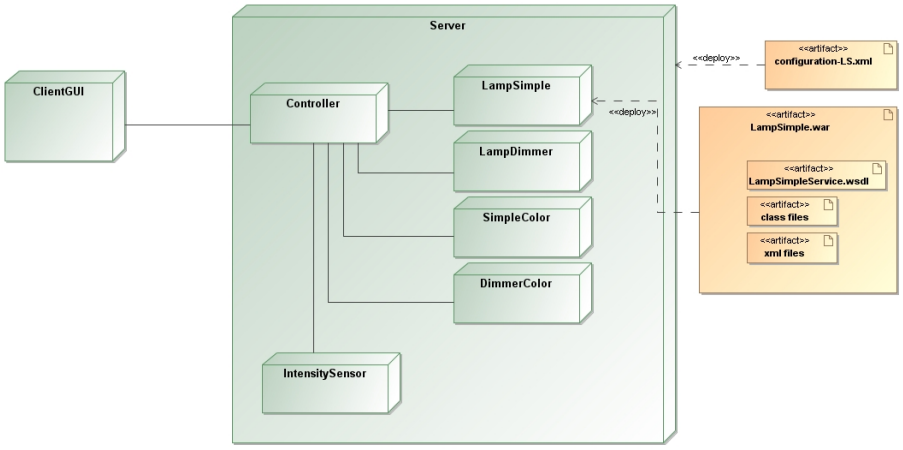


Abbildung 3.6: Ein komplettes Szenario

In diesem Szenario kommunizieren Lampenaktuatoren bzw. Sensoren mit dem zentrallen Controller über Web Services. Ferner hat der Controller seinerseits auch eigens Front-End (GUI), wo der Benutzer (Bewohner) per Befehlseingabe seine Wünsche an

das System übermitteln kann.

Zur Demonstration liefern wir zwei Varianten: *ControllerBasic* und *ControllerPremium*. *ControllerBasic* können mit *LampSimple* und *LampDimmer* arbeiten, und *ControllerPremium* können dagegen alle vier ansprechen.

Service-Identifikation

Ein Controller als Service-Konsument benutzt Web Services Operationen der Lampenaktuatoren, um die Lampen anzusprechen. Aus Sicht des Clients (Nutzer des Systems) ist er jedoch ein Service-Anbieter.

ControllerBasic bietet deshalb folgende Operationen an:

- *register*/*deregister* – zur An- bzw. Abmeldung der Module (Sensor oder Lampenaktuator) am System.
- *turnOnLamp*/*turnOffLamp* – damit hat der Nutzer die Möglichkeit, die einzelnen Lampen an der Benutzeroberfläche vom Controller an- oder auszuschalten.
- *dimUpLamp*/*dimDownLamp* – damit hat der Nutzer die Möglichkeit, die einzelnen Lampen an der Benutzeroberfläche vom Controller auf- oder abdimmen.
- *getStatus* – Monitorings-Funktion des Controllers zur Abfrage der Lampenzustände. Der Nutzer kann den momentanen Status einer bestimmten Lampe abfragen, ob sie gerade leuchtet bzw. wie stark diese Beleuchtung (bei *Dimmer*) ist.
- *changeSignal*/*getSignal* – zur Kommunikation mit Sensoren
- *update* – zur Aktualisierung der Zustände von Lampen, die durch Änderung des Sensor-Signals bewirkt sind.

ControllerPremium hat zusätzlich noch weitere Operationen, um *SimpleColor* bzw. *DimmerColor* ansprechen zu können:

- *changeColor* – zur Änderung der aktuellen Lichtfarbe
- *getColor* – zur Abfrage der aktuellen Lichtfarbe

Implementierung

Um die Web Services Operationen vom Sensoren und Lampenaktuatoren nutzen zu können, konsumieren wir sie zunächst innerhalb vom Controller. Dies geschieht mittels `wsimport`-Tools des Glassfish-Servers. Es wird dann eine Menge von Klassen mittels JAX-WS bzw. JAXB automatisch angelegt, die wir später zum Aufruf einer Web Services Operation brauchen werden.

In Listing 3.12 zeigen wir einen Ausschnitt aus dem WSDL-Dokument des *ControllerBasic*.

Listing 3.12: ControllerBasic.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions...>
  <types>
    <xsd:schema>
      <xsd:import schemaLocation="ControllerBasicSchema.xsd".../>
    </xsd:schema>
  </types>

  <message name="register"> ... </message>
  <message name="registerResponse"> ... </message>
  ...
  <message name="turnOnLamp"> ... </message>
  <message name="turnOnLampResponse"> ... </message>
  <message name="turnOffLamp">  ... </message>
  <message name="turnOffLampResponse"> ...  </message>
  <message name="dimUpLamp">  ... </message>
  <message name="dimUpLampResponse"> ...  </message>
  <message name="dimDownLamp">  ... </message>
  <message name="dimDownLampResponse"> ...  </message>
  ...
  <message name="update"> ... </message>
  <message name="updateResponse"> ... </message>

  <portType name="ControllerBasic">
    <operation name="register"> ... </operation>
    ...
    <operation name="turnOnLamp">
      <input message="tns:turnOnLamp"/>
      <output message="tns:turnOnLampResponse"/>
    </operation>
    ...
    <operation name="update"> ... </operation>
  </portType>

  <binding name="ControllerBasicPortBinding" type="tns:ControllerBasic">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="register"> ... </operation>
    <operation name="deregister"> ... </operation>
    <operation name="changeSignal"> ... </operation>
    <operation name="turnOnLamp"> ... </operation>
    <operation name="turnOffLamp"> ... </operation>
    <operation name="dimUpLamp"> ... </operation>
    <operation name="dimDownLamp"> ... </operation>
    <operation name="getSignal"> ... </operation>
    <operation name="getStatus"> ... </operation>
    <operation name="update"> ... </operation>
  </binding>

  <service name="ControllerBasic">
    <port name="ControllerBasicPort" binding="tns:ControllerBasicPortBinding">
      <soap:address location="http://localhost:8080/ControllerPlatform/ControllerBasic"
          .../>
    </port>
  </service>
</definitions>
```

Bei *ControllerPremium* wird der selbe Inhalt der WSDL-Datei um zwei Operationen *setColorValue* und *getColorValue* entsprechend erweitert (s. Listing 3.13).

Listing 3.13: ControllerPremium.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions...>
  <types>
    <xsd:schema>
      <xsd:import schemaLocation="ControllerPremiumSchema.xsd".../>
    </xsd:schema>
  </types>

```

```
9   ...
10  <message name="getColor">
11    <part name="parameters" element="tns:getColor"/>
12  </message>
13  <message name="getColorResponse">
14    <part name="result" element="tns:getColorResponse"/>
15  </message>
16  <message name="changeColor">
17    <part name="parameters" element="tns:changeColor"/>
18  </message>
19  <message name="changeColorResponse">
20    <part name="result" element="tns:changeColorResponse"/>
21  </message>
22
23  <portType name="ControllerPremium">
24    ...
25    <operation name="getColor">
26      <input message="tns:getColor"/>
27      <output message="tns:getColorResponse"/>
28    </operation>
29    <operation name="changeColor">
30      <input message="tns:changeColor"/>
31      <output message="tns:changeColorResponse"/>
32    </operation>
33  </portType>
34
35  <binding name="ControllerPremiumPortBinding" type="tns:ControllerPremium">
36    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
37    ...
38    <operation name="getColor"> ... </operation>
39    <operation name="changeColor"> ... </operation>
40  </binding>
41
42  <service name="ControllerPremium">
43    <port name="ControllerPremiumPort" binding="tns:ControllerPremiumPortBinding">
44      <soap:address location="http://localhost:8080/ControllerPlatform/ControllerPremium
          " .../>
45    </port>
46  </service>
47 </definitions>
```

Zur Einstellung der automatischen Schaltungsregelung für die Lampenaktuatoren durch Sensoren konfigurieren wir eine XML-Datei wie in Listing 3.14 gezeigt, z.B. der Sensor *IS1* observiert die Umgebung, wo die Lampen *LS1* und *LS3* aktiv sind.

Listing 3.14: Konfiguration für Controller

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <controller>
3    <reg sensor="IS1" device="LS1" />
4    <reg sensor="IS1" device="LS3" />
5
6    <reg sensor="IS2" device="LS2" />
7    <reg sensor="IS2" device="LD1" />
8    <reg sensor="IS2" device="LD2" />
9
10   <reg sensor="IS3" device="LS4" />
11   <reg sensor="IS3" device="LS5" />
12   <reg sensor="IS3" device="LD3" />
13 </controller>
```

Damit die automatische Schaltung der Lampen bei Änderung des Sensor-Signals möglich ist, wurde das *Observer*-Entwurfsmuster (s. [GHJV95]) adaptiert eingesetzt (s. Listing 3.15). Die Funktion *process* wird dann über Service-Operation *update* ausgeführt, falls die Web Services Operation *setSignal* für den Sensor aufgerufen wurde. Das ankommende Signal eines Sensors wird vom System geprüft, bevor Schaltungsbefehle an entsprechende Lampenaktuatoren losgeschickt werden.

Listing 3.15: ControllerBean.java

```
1  package de.controllerplatform.beans;
2  ...
3  public class ControllerBean {
4    ...
5    public LampSimple getLampSimpleService() {
6      return new LampSimple_Service().getLampSimplePort();
7    }
8    public LampDimmer getLampDimmerService() {
9      return new LampDimmer_Service().getLampDimmerPort();
10   }
11   public IntensitySensor getIntensitySensorService() {
12     return new IntensitySensor_Service().getIntensitySensorPort();
13   }
14   ...
15   public void process(String sensorID, int signal) {
16     if (!controller.get(sensorID).isEmpty()) {
17       switch (getSensorType(sensorID)) {
18       case 1: // IntensitySensor
19         Vector<String> devices = controller.get(sensorID);
20         for (int i=0; i<devices.size(); i++) {
21           String deviceID = devices.get(i);
22           if (isRegDevice(deviceID)) {
23             switch (getDeviceType(deviceID)) {
24             case 1: // LampSimple
25               switch (signal) {
26               case 1: // bright -> turnOff
27                 getLampSimpleService().turnOff(deviceID);
28                 break;
29               case 2: // dark -> turnOn
30                 getLampSimpleService().turnOn(deviceID);
31                 break;
32               default:
33                 break;
34               }
35               break;
36
37             case 2: // LampDimmer
38               switch (signal) {
39               case 1: // bright -> turnOff
40                 getLampDimmerService().turnOff(deviceID);
41                 break;
42               case 2: // dark -> turnOn
43                 getLampDimmerService().turnOn(deviceID);
44                 break;
45               default:
46                 break;
47               }
48               break;
49
50             default:
51               break;
52             }
53           }
54         }
55         break;
56
57       default:
58         break;
59       }
60     }
61   }
62   ...
63 }
```

Listing 3.16: ControllerBasicServiceImpl.java

```
1  package de.controller.webservice;
2
3  import java.util.ArrayList;
```

```
4  import javax.jws.WebService;
5  import de.controller.beans.ControllerBean;
6  @WebService(endpointInterface = "de.controller.webservice.ControllerBasic",
7      serviceName = "ControllerBasic",
8      portName = "ControllerBasicPort")
9  public class ControllerBasicServiceImpl {
10   ControllerBean bean = ControllerBean.getIntance();
11   ...
12   public void update(String sensorID, int signal) {
13     if (bean.isRegSensor(sensorID)) {
14       bean.process(sensorID, signal);
15     }
16   }
17   public void changeSignal(String sensorID, int signal) {
18     if (bean.isRegSensor(sensorID)) {
19       switch (bean.getSensorType(sensorID)) {
20       case 1: // IntensitySensor
21         bean.getIntensitySensorService().setSignal(sensorID, signal);
22         break;
23       default:
24         break;
25       }
26     }
27   }
28   public void turnOnLamp(String deviceID) {
29     if (bean.isRegDevice(deviceID)) {
30       switch (bean.getDeviceType(deviceID)) {
31       case 1: // LampSimple
32         bean.getLampSimpleService().turnOn(deviceID);
33         break;
34       case 2: // LampDimmer
35         bean.getLampDimmerService().turnOn(deviceID);
36         break;
37       default:
38         break;
39       }
40     }
41   }
42   ...
43   public void dimUpLamp(String deviceID) {
44     if (bean.isRegDevice(deviceID)) {
45       switch (bean.getDeviceType(deviceID)) {
46       case 2: // LampDimmer
47         bean.getLampDimmerService().dimUp(deviceID);
48         break;
49       default:
50         break;
51       }
52     }
53   }
54   ...
55 }
```

Listing 3.17 zeigt die Erweiterung bei der Service-Implementierung von *Controller-Premium* in mehreren Stellen. Dies stellt ein Problem der Modularität dar. Dieses Problem werden wir später noch analysieren.

Listing 3.17: ControllerPremiumServiceImpl.java

```
1  public class ControllerPremiumServiceImpl {
2    ControllerBean  bean = ControllerBean.getIntance();
3    ...
4    public void turnOnLamp(String deviceID) {
5      if (bean.isRegDevice(deviceID)) {
6        switch (bean.getDeviceType(deviceID)) {
7        ...
8        case 3: // SimpleColor
9          bean.getSimpleColorService().turnOn(deviceID);
10         break;
11       case 4: // DimmerColor
```

```
12          bean.getDimmerColorService().turnOn(deviceID);
13          break;
14        default:
15          break;
16        }
17      }
18    }
19    ...
20    public void dimUpLamp(String deviceID) {
21      if (bean.isRegDevice(deviceID)) {
22        switch (bean.getDeviceType(deviceID)) {
23        ...
24        case 4: // DimmerColor
25          bean.getDimmerColorService().dimUp(deviceID);
26          break;
27        default:
28          break;
29        }
30      }
31    }
32    ...
33    public void changeColor(String deviceID, int color) {
34      if (bean.isRegDevice(deviceID)) {
35        switch (bean.getDeviceType(deviceID)) {
36        case 3: // SimpleColor
37          bean.getSimpleColorService().changeColor(deviceID, color);
38          break;
39        case 4: // DimmerColor
40          bean.getDimmerColorService().changeColor(deviceID, color);
41          break;
42        default:
43          break;
44        }
45      }
46    }
47    ...
48  }
```

Abschließend entwickeln wir ein kleines Front-End (GUI), um das komplette Szenario zu testen (s. Abbildung 3.7).

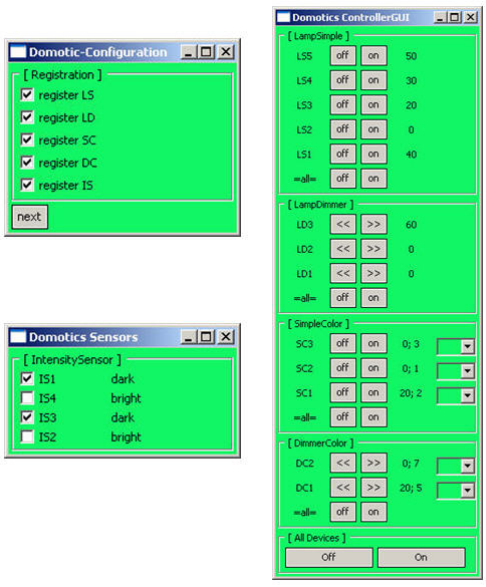


Abbildung 3.7: ClientGUI

3.3 Problemanalyse

Während der Implementierung des SOA-Kontexts der Fallstudie sind uns bereits einige Schwierigkeiten entgegengekommen. Dort haben wir auch einen Hinweis auf diesen Abschnitt gegeben. Wir werden nachfolgend versuchen, all diese Probleme zusammenzuführen, bzw. zu klassifizieren, sowie einen möglichen Lösungsansatz gegebenenfalls vorzuschlagen.

Problem der Variabilität

Wir erinnern uns zunächst an die vier Lampenvarianten des Demonstrators mit ihrer Hauptfunktionalität:

- *LampSimple* = {anschalten, ausschalten}
- *LampDimmer* = {*anschalten*, *ausschalten*, aufdimmen, abdimmen}
- *SimpleColor* = {*anschalten*, *ausschalten*, Farbe wechseln}
- *DimmerColor* = {anschalten, ausschalten, aufdimmen, abdimmen, Farbe wechseln}

In der Software-Entwicklung spielt die *Wiederverwendung* von Code-Artefakten besonders wichtige Rolle. Dadurch wird Redundanz in der Entwicklung vermieden. Je höher der Wiederverwendungsgrad ist, desto mehr verringert sich der Programmieraufwand.

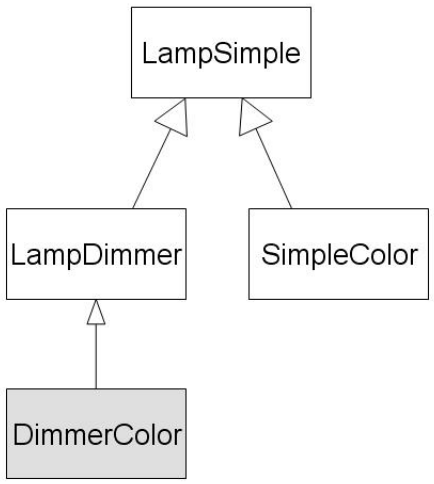
Zur Lösung dieses Aspekts wird in OOP das *Vererbungsprinzip* (oder auch *Generalisierungs-Spezialisierungs-Prinzip* genannt) eingeprägt: „Klassen können Spezialisierungen anderer Klassen darstellen, d.h. Klassen können hierarchisch angeordnet werden, übernehmen ('erben') dabei die Eigenschaften der ihnen übergeordneten Klassen und können diese bedarfsweise spezialisieren ('überschreiben'), aber nicht eliminieren.“(s. [Oes01], Seite 45)

Wir können diese in einer Vererbungshierarchie so aufbauen, dass *LampDimmer* und *SimpleColor* direkt von *LampSimple* erben, d.h. die Funktionen {an- und ausschalten} von *LampSimple* werden bei *LampDimmer* und *SimpleColor* wiederverwendet. *DimmerColor* spezialisiert wiederum *LampDimmer* durch Ergänzung der Funktion {Farbe wechseln}. Man kann allerdings *DimmerColor* von *SimpleColor* auch genauso gut erben lassen (s. Abbildung 3.8). In diesem Fall erweitert dies die vorhandenen Funktionen von *SimpleColor* um {auf- und abdimmen}.

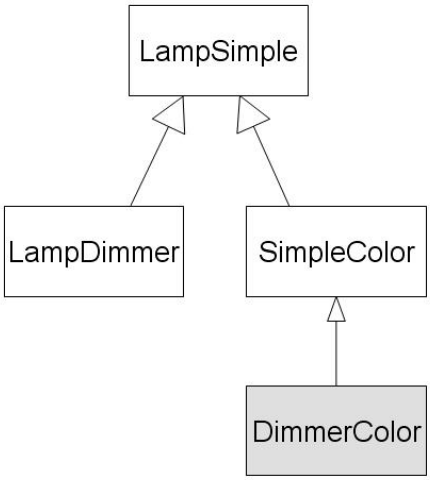
Das Problem bezieht sich nun auf die *Code-Replikation*, welche man dadurch in Kauf nehmen muss. An Beispiel in Abbildung 3.8.a müssen bestehende Funktionen von *SimpleColor* wiederholt bei *DimmerColor* implementiert werden.

Je mehr Variabilität eingebracht wird, desto massiver wird die Redundanz im Code, die wir feststellen können. Wir stellen uns beispielsweise noch ein neues *Timing*-Feature vor. Dieses bereichert unsere Lampen z.B. mit der Fähigkeit, Beleuchtungszeit einzustellen. Eine Möglichkeit, die existierende Hierarchie zu erweitern, zeigt die Abbildung 3.9.a.

Selbst die Tatsache, dass es mehrere Möglichkeiten zum Aufbau einer Vererbungshierarchie gibt, ist irreführend, und erschwert die Übersichtlichkeit, sowie die Wartung der Software. Diese Art der Vererbung zeigt sich bei Änderung auch inflexibel.

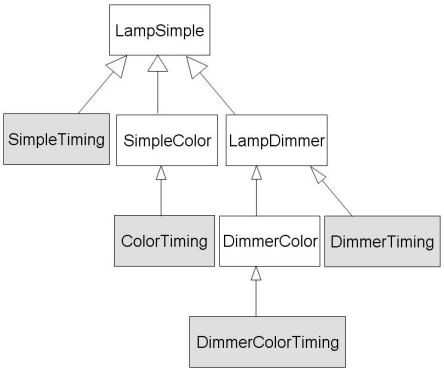


(a) LampDimmer spezialisieren

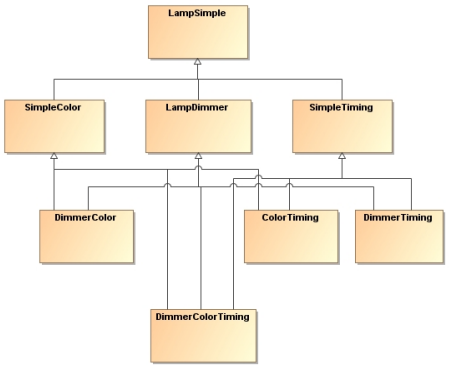


(b) SimpleColor spezialisieren

Abbildung 3.8: Vererbungshierarchie der Lampenmodule



(a) Einfache Vererbungshierarchie



(b) Mehrfache Vererbungshierarchie

Abbildung 3.9: Vererbungshierarchie der Lampenmodule

Als Lösung für dieses Problem wäre Mehrfachvererbung gut vorstellbar (s. Abbildung 3.9.b), da in den Subklassen tatsächlich nur die Differenzen zu deren Superklassen implementiert werden sollen. Allerdings wird diese nur in wenigen Programmiersprachen unterstützt. In Java kann man Mehrfachvererbung noch mit Hilfe von Schnittstellen (Stichwort: Kapselung, Polymorphismus) realisieren. Dies haben wir in unserem Fallbeispiel versucht. Aber das Problem der massiven Code-Replikation wird dadurch nicht behoben (vgl. Listing 3.1 mit 3.2). In beiden Klassen *DCObject* und *LDObject* müssen wir jeweils die Methoden *getPrecision* bzw. *setPrecision* implementieren.

In Listings 3.18 und 3.19 zeigen wir die Service-Interfaces der *ControllerBasic* und *ControllerPremium*.

Listing 3.18: ControllerBasic.java

```
1 package de.controllerbasic.webservice;
2
3 import java.util.ArrayList;
4
5 import javax.jws.WebMethod;
```

```
6  import javax.jws.WebService;
7
8  @WebService(name = "ControllerBasic")
9  public interface ControllerBasic {
10   @WebMethod
11   public boolean register(String key);
12   @WebMethod
13   public boolean deregister(String key);
14   @WebMethod
15   public void update(String sensorID, int signal);
16   @WebMethod
17   public void changeSignal(String sensorID, int signal);
18   @WebMethod
19   public void turnOnLamp(String deviceID);
20   @WebMethod
21   public void turnOn(int type);
22   @WebMethod
23   public void turnOnAll();
24   @WebMethod
25   public void turnOffLamp(String deviceID);
26   @WebMethod
27   public void turnOff(int type);
28   @WebMethod
29   public void turnOffAll();
30   @WebMethod
31   public void dimUpLamp(String deviceID);
32   @WebMethod
33   public void dimDownLamp(String deviceID);
34   @WebMethod
35   public int getSignal(String sensorID);
36   @WebMethod
37   public int getStatus(String deviceID);
38   @WebMethod
39   public ArrayList<String> getLamps(int type);
40   @WebMethod
41   public ArrayList<String> getSensors(int type);
42 }
```

Listing 3.19: ControllerPremium.java

```
1  package de.controllerpremium.webservice;
2
3  import java.util.ArrayList;
4
5  import javax.jws.WebMethod;
6  import javax.jws.WebService;
7
8  @WebService(name = "ControllerPremium")
9  public interface ControllerPremium {
10   @WebMethod
11   public boolean register(String key);
12   @WebMethod
13   public boolean deregister(String key);
14   @WebMethod
15   public void update(String sensorID, int signal);
16   @WebMethod
17   public void changeSignal(String sensorID, int signal);
18   @WebMethod
19   public void turnOnLamp(String deviceID);
20   @WebMethod
21   public void turnOn(int type);
22   @WebMethod
23   public void turnOnAll();
24   @WebMethod
25   public void turnOffLamp(String deviceID);
26   @WebMethod
27   public void turnOff(int type);
28   @WebMethod
29   public void turnOffAll();
30   @WebMethod
31   public void dimUpLamp(String deviceID);
```

```
32    @WebMethod
33    public void dimDownLamp(String deviceID);
34    @WebMethod
35    public void changeColor(String deviceID, int color);
36    @WebMethod
37    public int getSignal(String sensorID);
38    @WebMethod
39    public int getStatus(String deviceID);
40    @WebMethod
41    public int getColor(String deviceID);
42    @WebMethod
43    public ArrayList<String> getLamps(int type);
44    @WebMethod
45    public ArrayList<String> getSensors(int type);
46  }
```

Hier besteht es keine Möglichkeit zur Trennung zwischen Variabilität und Gemeinsamkeit der Service-Varianten. Schon bei der Menge von Operationen verliert man die Übersicht, wo sich die beiden Services unterscheiden bzw. was sie anbieten können.

Problem der querschneidenden Belange

Crosscutting concerns (deut. *querschneidende Belange*) sind entstandene Probleme bei der Modularisierung bzw. „separation of concerns“ (deut. Aufteilung der Belange) in der Software-Entwicklung. Diese klassifizieren sich in drei Hauptprobleme: vermischten (tangling), verstreuten (scattering) und replizierten (replication) Code. Dadurch erschweren sich die Wartungsarbeit, Lesbarkeit sowie Anpassbarkeit der Anwendung.

Anhand der Strukturierung in Java-Paketen (s. Abbildung 3.10 ist uns nicht gelungen, Sachverhalten zu modularisieren, die zu einem Belang gehören.

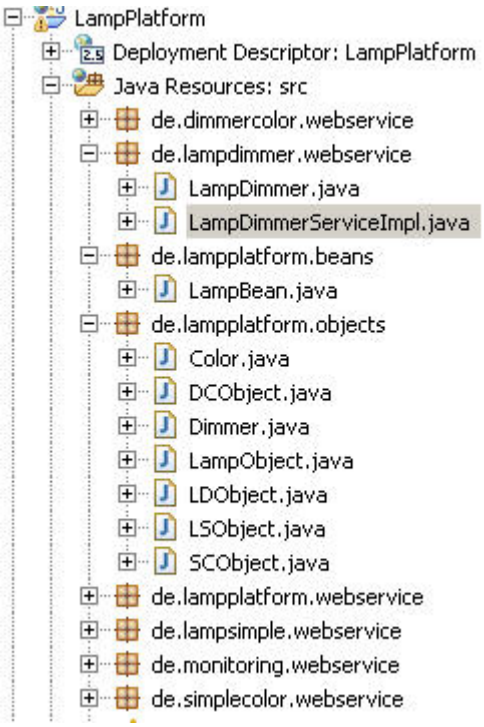


Abbildung 3.10: Paketstruktur in Java

Hier wurde der Code z.B. für Dimmer-Funktion in mehreren Klassen verstreut, in

beans-, *objects*- oder auch in *webservice*-Paketen.

Weitere Probleme und Lösungsvorschlag

In einem Szenario mit vielen Service-Variaten ist die Verfassung einzelner Service-Beschreibungs-Dokumente (WSDL) ziemlich mühsam. Hier kann man zwar ein gemeinsames Schema zur Beschreibung der benötigten Datenstrukturen erstellen, auf dieses WSDL-Dokumente referenzieren werden. Aber die WSDL-Dateien selbst kann man noch nicht wiederverwenden, obwohl der Inhalt (Operation) dieser Dateien zum größten Teil ähnlich sind.

Im Kapitel 2 haben wir bereits das Konzept der Software-Produktlinie eingeführt, welches für die Lösung der Variabilitätsprobleme geeignet ist. Einen SPL-Implementierungsansatz habe wir auch vorgestellt, nämlich Feature-Oriented Programming. Ob der Einsatz dieses Konzepts im gegebenen SOA-Kontext erfolgreich ist, sehen wir nach deren Umsetzung im Evaluations-Abschnitt des nächsten Kapitels.

3.4 Zusammenfassung

Dieses Kapitel umfasst eine Fallstudie zu *Domotic*. Im ersten Teil dieses Kapitels beschäftigten wir uns hauptsächlich mit dem Thema Domotic, konkret mit dessen Definition, Anwengungsgebieten, etc. Im zweiten Teil haben wir ein Domotic-Szenario in einer SOA-Umgebung mit Web Services in zwei Schritten implementiert. Wir haben den Kontext des Demonstrators bewusst so aufgebaut, dass er gewisse Variabilitäten enthält, die aber von der Realität nicht weit entfernt sind. Eine Beurteilung von dem hier vorgestellten Implementierungsansatz erfolgte in der Problemanalyse im dritten Teil dieses Kapitels. Im nächsten Kapitel, Kapitel 4, werden wir die hier vorgestellte Variabilitätsimplementierung mit FOP noch einmal umzusetzen, bevor wir die beiden Vorgehensweisen miteinander vergleichen können.

Kapitel 4

Evaluation

In diesem Kapitel handelt es sich um die Evaluierung des FOP-Einsatzes zur Implementierung der Variabilität im vorhandenen SOA-Kontext aus dem Kapitel 3. Die dort durchgeführte Problemanalyse zeigte bereits Schwächen in der Entwicklung bezüglich der Variabilitätsimplementierung. Diese gibt abschließend auch Hinweis auf eine alternative Lösung, nämlich FOP als Verbesserungsvorschlag für die Situation. Um festzustellen, ob und wo der FOP-Einsatz bezüglich des Kontexts der Fallstudie sinnvoll erscheinen könnte, werden wir uns im ersten Teil dieses Kapitel mit dessen Realisierung beschäftigen. Ein Vergleich zwischen den zwei Vorgehensweisen ermöglicht uns danach, die Fragestellung ganz am Anfang dieser Arbeit auszuarbeiten.

4.1 FOP-Umsetzung zu Demonstrator

Wir haben anhand der Analyse gesehen, dass unser Demonstrator genügende Probleme bereitet hat. Nun versuchen wir ihn mit FOP zu verbessern. Unser Ziel ist die gleiche Variabilität im Szenario nach dem Produktlinien-Ansatz zu implementieren. Erfahrungen, die wir nach diesem Abschnitt sammeln werden, werden dann mit den Erfahrungen aus der bisherigen Vorgehensweise verglichen, um den Einsatz von FOP evaluieren zu können.

Für die Implementierung verwenden wir folgende technologische Hilfsmittel:

- FeatureIDE[1]
- AHEAD Tool Suite v2009.03.27[2]: *xak*-Tool[3]
- FeatureHouse v0.2.2[4]

Aus dem Demonstrator-Beispiel können wir zwei Produktlinien erkennen: eine für *Lampenmodule* und eine für *Controller*. Diese werden nacheinander jeweils in vier Schritten implementiert, welche zu den Hauptprozessen des SPL-Frameworks gehören (vgl. Kapitel 2):

[1] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`
[2] `http://www.cs.utexas.edu/~schwartz/ATS.html`
[3] `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/xak.html`
[4] `http://www.fosd.de/fh`

- *Feature-Modellierung* – kommt in der Phase *Domänenanalyse* des Domain Engineering zum Einsatz. Wir werden FODA-Methode zur Modellierung des Feature-Diagramms verwenden.

- *Feature-Implementierung* – ist die letzte Phase des Domain Engineering. Die ursprünglichen Klassen werden in Kollaborationslayer mit verschiedenen Rollen aufgesplittet, um einzelne Feature-Module zu implementieren. Es kommen non-code-Artefakte (hier: WSDL-Dateien und evtl. Deployment Descriptors oder Build-Skripte) auch noch ins Spiel.

- *Feature-Selektion* – wählt Features aus zur Konfiguration eines Produktes.

- *Feature-Komposition* – Durch Komposition entsteht ein neues Produkt für die Produktlinie.

4.1.1 Produktlinie der *Lampenmodule*

Feature-Modellierung

Kontextanalyse
Aus dem vorhandenen Kontext heraus haben die vier Lampenvarianten folgende Beziehung:

$LampDimmer = \{DimmerFunktion\} \bullet LampSimple$
$SimpleColor = \{ColorFunktion\} \bullet LampSimple$
$DimmerColor = \{ColorFunktion\} \bullet \{DimmerFunktion\} \bullet LampSimple$

Das heißt, die Funktionen von *LampSimple* bilden die *gemeinsame Plattform* (*LampBase*). *Variabilität* wird hier jeweils durch die Dimmer- bzw. Color-Funktion gekennzeichnet.

Modellierung
Anhand dieser Kontextanalyse können wir den Sachverhalt als Feature-Diagramm modelliert (s. Abbildung 4.1). *LampBase* ist sozusagen die Basis des Produktes, ist deshalb ein notwendiges Element. *LampFeatures* ist dagegen optional. Bei dessen Auswahl hat man dann die Möglichkeit, *Dimmer* oder *Color* oder auch beides zu selektieren.

Feature-Implementierung

Bei Service-Implementierung fängt man standardgemäß erst mit der Verfassung von WSDL-Dateien (Service Endpoint Interface, SEI) an. Danach implementiert man die Service Implemention Bean (SIB)-Klassen (vgl. *Top-down*-Ansatz der Web Services Implementierung im letzten Kapitel).

Wir werden den Ablauf dieses Vorgehens folgend so belassen, jedoch mit einer kleinen Änderung. Der Schritt zur Erstellung eines *Kollaborationsdiagramms* wird nämlich zum Anfang vorgerückt aufgrund der Tatsache, dass wir den Demonstrator an dieser Stelle nicht von Grund auf entwerfen, sondern den bereits konzipierten Kontext übernehmen.

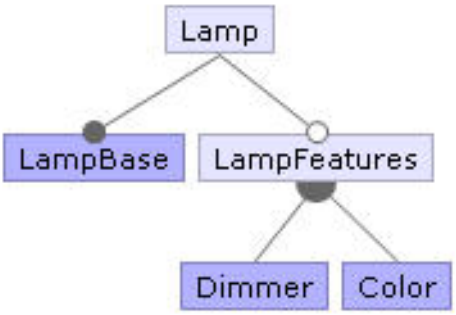


Abbildung 4.1: Feature-Diagramm der Lampenmodule

Das heißt, dass wir im Voraus wissen oder ahnen, welche Web Services Operationen in *LampBase* bzw. in den einzelnen Feature-Modulen erwartet werden.

Mittels Kollaborationsdiagramms strukturieren wir die zu implementierenden Artefakte, indem wir die benötigten Klassen in Kollaborationslayers (vgl. Feature-Module) und Rollen aufteilen (s. Kapitel 2). Das Kollaborationsdiagramm zu unseren Lampenmodulen wird in Abbildung 4.2 grafisch dargestellt.

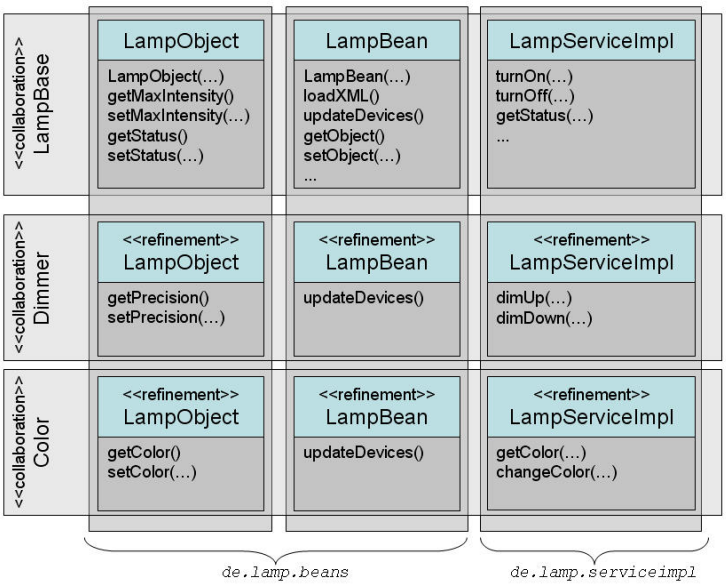


Abbildung 4.2: Kollaborationsdiagramm der Lampenmodule

Dem Feature-Diagramm entsprechend haben wir 3 Module, nämlich *LampBase*, *Dimmer* und *Color*. Diese bilden im Einzelnen sogenanntes Kollaborationslayer. Als nächstes zu erkennen sind die 3 Klassen (`LampObject`, `LampBean` und `LampServiceImpl`), die sich jeweils in den allen 3 Layers befinden, zuzüglich deren Methoden.

Aus dem Kollaborationsdiagramm sehen wir, dass Refinements sowohl bei *Dimmer*- als auch bei *Color*-Modul vorgenommen werden müssen, da diese jeweils zwei neue Operationen definieren.

Wie am Anfang dieses Abschnitts erwähnt, werden wir Sprachmittel des *xak*-Tools

im AHEAD Tool Suite zum Refinement der WSDL-Dokumente einsetzen.

Wir beginnen mit dem Refinement des Schemas. Hier definieren wir für *Color*-Feature 4 neue Datentypen: *getColor*, *getColorResponse*, *changeColor*, *changeColorResponse* (s. Listing 4.1), und für *Dimmer*-Feature: *dimUp*, *dimUpResponse*, *dimDown*, *dimDownResponse* (s. Listing 4.2).

Listing 4.1: Verfeierung von LampSchema.xsd für *Color*

```
<?xml version="1.0" encoding="UTF-8"?>
<xr:refine xmlns:xr="http://www.atarix.org/xmlRefinement"...>

  <xr:at select="/xs:schema">
    <xr:append>
      <xs:element name="getColor" type="tns:getColor" />
      <xs:complexType name="getColor">
        <xs:sequence>
          <xs:element name="id" type="xs:string" minOccurs="0" />
        </xs:sequence>
      </xs:complexType>

      <xs:element name="getColorResponse" type="tns:getColorResponse" />
      <xs:complexType name="getColorResponse">
        <xs:sequence>
          <xs:element name="return" type="xs:int" />
        </xs:sequence>
      </xs:complexType>

      <xs:element name="changeColor" type="tns:changeColor" />
      <xs:complexType name="changeColor">
        <xs:sequence>
          <xs:element name="id" type="xs:string" minOccurs="0" />
          <xs:element name="color" type="xs:int" />
        </xs:sequence>
      </xs:complexType>

      <xs:element name="changeColorResponse" type="tns:changeColorResponse" />
      <xs:complexType name="changeColorResponse">
        <xs:sequence>
          <xs:element name="return" type="xs:int" />
        </xs:sequence>
      </xs:complexType>
    </xr:append>
  </xr:at>

</xr:refine>
```

Listing 4.2: Verfeierung von LampSchema.xsd für *Dimmer*

```
<?xml version="1.0" encoding="UTF-8"?>
<xr:refine xmlns:xr="http://www.atarix.org/xmlRefinement"...>

  <xr:at select="/xs:schema">
    <xr:append>
      <xs:element name="dimUp" type="tns:dimUp" />
      <xs:complexType name="dimUp">
        <xs:sequence>
          <xs:element name="id" type="xs:string" minOccurs="0" />
        </xs:sequence>
      </xs:complexType>

      <xs:element name="dimUpResponse" type="tns:dimUpResponse" />
      <xs:complexType name="dimUpResponse">
        <xs:sequence>
          <xs:element name="return" type="xs:int" />
        </xs:sequence>
      </xs:complexType>

      <xs:element name="dimDown" type="tns:dimDown" />
```

```
21      <xs:complexType name="dimDown">
22        <xs:sequence>
23          <xs:element name="id" type="xs:string" minOccurs="0" />
24        </xs:sequence>
25      </xs:complexType>
26
27      <xs:element name="dimDownResponse" type="tns:dimDownResponse" />
28      <xs:complexType name="dimDownResponse">
29        <xs:sequence>
30          <xs:element name="return" type="xs:int" />
31        </xs:sequence>
32      </xs:complexType>
33    </xr:append>
34  </xr:at>
35
36 </xr:refine>
```

Dann wird die Verfeinerung für das WSDL-Dokument vorgenommen. Nachrichten bzw. Operationen und deren Bindungsbeschreibungen müssen hier ergänzt werden (s. Listing 4.3).

Listing 4.3: Verfeinerung von LampService.wsdl

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xr:refine xmlns:xr="http://www.atarix.org/xmlRefinement"...>
3
4    <xr:at select="/xs:schema/messages">
5      <!-- messages-Element manuell in Basisdatei einbauen, um message-Elemente richtig zu
           positionieren -->
6      <xr:append>
7        <message name="getColor">
8          <part name="parameters" element="tns:getColor"/>
9        </message>
10       <message name="getColorResponse">
11         <part name="result" element="tns:getColorResponse"/>
12       </message>
13       ...
14     </xr:append>
15   </xr:at>
16
17   <xr:at select="/xs:schema/portType">
18     <xr:append>
19       <operation name="getColor">
20         <input message="tns:getColor"/>
21         <output message="tns:getColorResponse"/>
22       </operation>
23       <operation name="changeColor">
24       ...
25       </operation>
26     </xr:append>
27   </xr:at>
28
29   <xr:at select="/xs:schema/binding">
30     <xr:append>
31       <operation name="getColor">
32         <soap:operation soapAction=""/>
33         <input>
34           <soap:body use="literal"/>
35         </input>
36         <output>
37           <soap:body use="literal"/>
38         </output>
39       </operation>
40       <operation name="changeColor">
41       ...
42       </operation>
43     </xr:append>
44   </xr:at>
45
46 </xr:refine>
```

Ein kleines Problem, was jedoch an dieser Stelle auffällig ist, liegt an der Positionierung von den `message`-Elementen. Es gibt derzeit noch keine Möglichkeit mit *xak* bzw. *XPath*, die zulässige Position von denen in der WSDL-Datei zu finden, und dort die Refinements-Artefakte einzusetzen. Diese Artefakte werden immer automatisch am Ende des aktuellen XML-Knotens hinzugefügt.

Abhilfe hierfür finden wir beim manuellen Hinzufügen eines temporären Elements namens `messages` als Wrapper für alle `message`-Elemente (s. Listing 4.4). Nach der Komposition muss dieses Element in der resultierenden WSDL-Datei wieder manuell entfernt werden, um die Gültigkeit des WSDL-Dokuments sicherzustellen. Da dieses Vorgehen unser Vorhaben nicht großartig beeinträchtigt, lassen wir es hier gelten.

Listing 4.4: Basis von LampService.wsdl

```
<definitions...>
  <types>
    <xsd:schema>
      <xsd:import... schemaLocation="LampSchema.xsd" ...>
    </xsd:schema>
  </types>

<messages> <!-- manuell hinzugefügt -->
  <message name="turnOn">
    <part name="parameters" element="tns:turnOn"/>
  </message>
  <message name="turnOnResponse">
    <part name="result" element="tns:turnOnResponse"/>
  </message>
  ...
</messages> <!-- manuell hinzugefügt -->

  <portType name="LampPortType">
    ...
  </portType>

  <binding name="LampBinding" type="tns:LampPortType">
    ...
  </binding>

  <service name="LampService">
    <port name="LampPort" binding="tns:LampBinding">
      <soap:address location="http://localhost:8080/LampSimple/LampService" .../>
    </port>
  </service>
</definitions>
```

Nun wird an dieser Stelle der erste Kompositionsprozess mittels *xak* stattfinden, um die entsprechende Service-Beschreibung (WSDL-Datei) für das neue Produkt zu generieren.

Das Tool `wsimport` wird dann aufgerufen, um die benötigten Java-Klassen bereitzustellen.

Die eigentliche Service-Implementierung erfolgt jetzt, und zwar mit der Implementierung einzelnen Feature-Module für die Lampenvarianten (vgl. Kollaborationsdiagramm 4.2). Listing 4.5 zeigt einen kleinen Quelltext-Ausschnitt des Refinements in *Dimmer*- bzw. *Color*-Modul.

Listing 4.5: Verfeinerung von LampServiceImpl.java

```
// Dimmer
package de.lamp.serviceimpl;

public class LampServiceImpl {
```

```
5     public int dimUp(String id) {
6       if (id.startsWith(LampBean.ID)) {
7         obj = bean.getObject(id);
8         if (obj == null) {
9           return -1;
10        }
11        int newIntensity = obj.getStatus() + obj.getPrecision();
12        if (newIntensity < obj.getMaxIntensity()){
13          obj.setStatus(newIntensity);
14        } else {
15          obj.setStatus(obj.getMaxIntensity());
16        }
17        return 0;
18      }
19      return -2;
20    }
21    ...
22  }
23
24  // Color
25  package de.lamp.serviceimpl;
26
27  public class LampServiceImpl {
28    public int changeColor(String id, int color) {
29      if (id.startsWith(LampBean.ID)) {
30        obj = bean.getObject(id);
31        if (obj == null) {
32          return -1;
33        }
34        obj.setColor(color);
35        return 0;
36      }
37      return -2;
38    }
39    ...
40  }
41
42  // LampBase
43  package de.lamp.serviceimpl;
44  ...
45  import de.lamp.beans.*;
46
47  @WebService(endpointInterface="de.lamp.webservice.LampPortType",serviceName="LampService
        ",portName="LampPort")
48  public class LampServiceImpl {
49    LampBean bean = LampBean.getIntance();
50    LampObject obj;
51
52    public int turnOn(String id) {
53      if (id.startsWith(LampBean.ID)) {
54        obj = bean.getObject(id);
55        if (obj == null) {
56          return -1;
57        }
58        obj.setStatus(obj.getMaxIntensity());
59        return 0;
60      }
61      return -2;
62    }
63    ...
64  }
```

Feature-Selektion

Die Konfiguration eines zu generierenden Produktes (hier: Lampenvariante) erfolgt durch Auswahl der vorhandenen Features. Dies geschieht mit der Erfassung von einer sogenannten *expression*-Datei in FeatureHouse (oder *equation*-Datei mit ATS). Abbildung 4.3

zeigt die grafische Oberfläche zur Feature-Selektion in FeatureIDE. Hier sehen wir auch die Anzahl der möglichen Varianten aufgrund der bereits ausgewählten Features schön dargestellt.

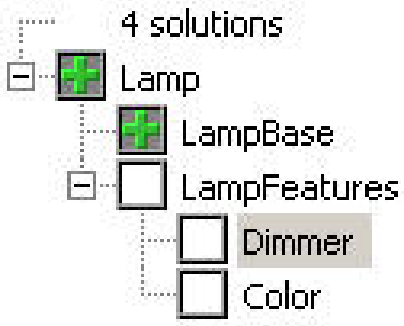


Abbildung 4.3: Feature-Selektion für Lampenmodule

Feature-Komposition

Wir kommen nun zum letzten Schritt der Entwicklung, nämlich zu der Komposition der Feature-Module. Eine Ausführung von FeatureHouse löst automatisch den internen Kompositionsprozess dazu, und erledigt diesen Schritt. Der zusammengestellte Quelltext wird für uns als Ergebnis generiert. Wir können ihn danach wie gewöhnlich mit `javac` kompilieren.

Das Produkt ist bereits für die Verpackung (*war*-Datei) bzw. das Deployment später auf den Glassfish.

Die Struktur einer solchen *war*-Datei sieht folgendermaßen aus (hier als Beispiel *LampDimmer.war*):

- META-INF
  - MANIFEST.MF
- WEB-INF
  - classes – enthält generierte JAX-WS-Quelltexte, u.a. die Service-Schnittstelle (SEI)
  - lib
    - *featurelampdimmer.jar* – Produktvariant *LampDimmer* nach Komposition
  - wsdl – WSDL nach Komposition
    - *LampSchema.xsd*
    - *LampService.wsdl*
  - *sun-jaxws.xml* – Deployment Descriptor
  - *web.xml* – Deployment Descriptor

4.1.2 Produktlinie der *Controller*

Feature-Modellierung

Die *gemeinsame Plattform* für Controller-Produktlinie bildet sich aus der Basisfunktionalität des Controllers im Umgang mit den Basislampen *LampSimple* und *LampDimmer* bzw. mit dem Sensor *IntensitySensor* (*ControllerBase*). *Variabilität* entsteht in diesem Zusammenhang durch Hinzufügen von erweiterten Lampenmodulen (*SimpleColor*, *DimmerColor*).

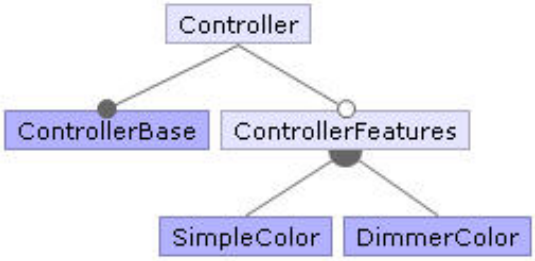


Abbildung 4.4: Feature-Diagramm der Controller

Feature-Implementierung

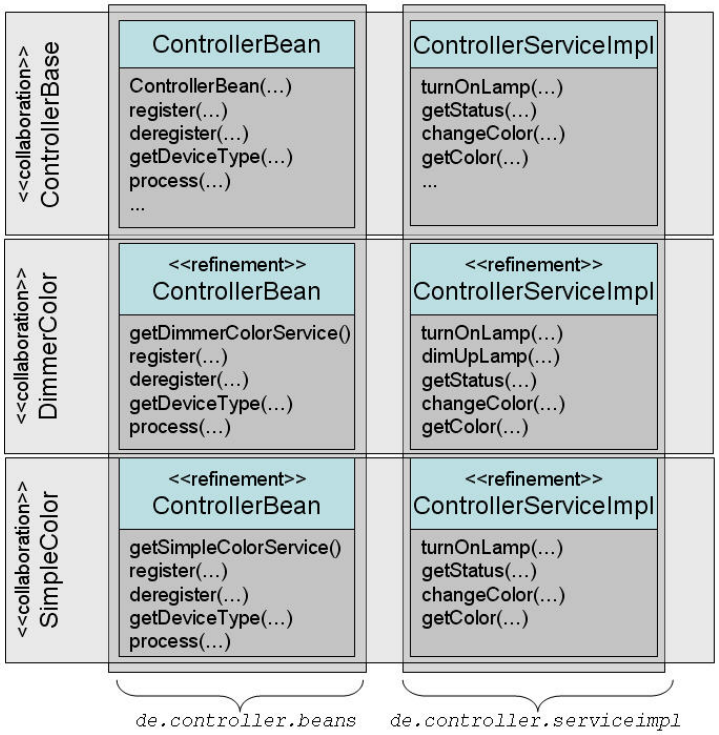


Abbildung 4.5: Kollaborationsdiagramm der Controller

Aus technischem Grund müssen die beiden Operationen *getColor* und *changeColor* der Module *SimpleColor* bzw. *DimmerColor* auch präsent im *ControllerBase* sein (s.

Kollaborationsdiagramm in Abbildung 4.5). Auf eine konkrete Begründung werden wir später noch mal zurückkommen. An dieser Stelle nehmen wir dies einfach so hin.

Da alle Web Services Operationen schon im *ControllerBase* bekannt sind, bleibt die WSDL-Datei unabhängig von der Feature-Selektion unverändert. Somit besteht es hier kein Anspruch auf Refinements bezüglich der Service-Beschreibung. Das bedeutet, dass WSDL-Dokument in diesem Fall nur einmal erfasst wird, und zwar mit allen brauchbaren Operationen in den Feature-Modulen.

Theoretisch macht es keinen Sinn, die Operationen *getColor* bzw. *changeColor* auch im *ControllerBase* implementieren zu lassen, weil diese keine Basisfunktionalität des Controllers (im Sinne der Zusammenarbeit mit *LampSimple* und *LampDimmer*) darstellen.

Technisch muss man so realisieren, da es möglich sein muss, dass die eine Operation bei Auswahl beider Features dieselbe in anderem Modul aufruft. Dies erfolgt in FeatureHouse mit *ogininal*-Methode. Aus diesem Grund müssen wir *getColor* und *changeColor* in *ControllerBase* aufnehmen. Da sie dort eigentlich nur die Rolle einer Maske spielen, lassen wir sie leer. Diese werden entsprechend in sowohl *SimpleColor* als auch *DimmerColor* verfeinert (s. Listings 4.6, 4.7).

Listing 4.6: Verfeinerung von ControllerBean.java in *DimmerColor*

```
1  //ControllerBase
2  package de.controller.beans;
3  ...
4  public class ControllerBean {
5    ...
6    public de.lampsimple.webservice.LampService getLampSimpleService() {
7      return new de.lampsimple.webservice.LampPortType().getLampServicePort();
8    }
9    public de.lampdimmer.webservice.LampService getLampDimmerService() {
10     return new de.lampdimmer.webservice.LampPortType().getLampServicePort();
11   }
12   public de.intensitysensor.webservice.SensorService getIntensitySensorService() {
13     return new de.intensitysensor.webservice.SensorPortType().getSensorServicePort();
14   }
15   public boolean register(String key) {
16     if (regs.contains(key)) {
17       return true;
18     }
19     //--devices
20     if (key.equals("LS")) {
21       regs.add(key);
22       devices.addAll(getLampSimpleService().getDevices());
23       return true;
24     }
25     if (key.equals("LD")) {
26       regs.add(key);
27       devices.addAll(getLampDimmerService().getDevices());
28       return true;
29     }
30     //--sensors
31     ...
32     return false;
33   }
34   public int getDeviceType(String deviceID) {
35     if (deviceID.startsWith("LS")) {
36       return 1; // LampSimple
37     }
38     if (deviceID.startsWith("LD")) {
39       return 2; // LampDimmer
40     }
41     return 0;
42   }
```

```
43    ...
44  }
45  //----------------
46  //DimmerColor
47    ...
48    public de.dimmercolor.webservice.LampService getDimmerColorService() {
49      return new de.dimmercolor.webservice.LampPortType().getLampServicePort();
50    }
51    public boolean register(String key) {
52      boolean result = original(key);
53      //--devices
54      if (key.equals("DC")) {
55        regs.add(key);
56        devices.addAll(getDimmerColorService().getDevices());
57        return true;
58      }
59      return result;
60    }
61    public int getDeviceType(String deviceID) {
62      int result = original(deviceID);
63      if (deviceID.startsWith("DC")) {
64        return 4; // DimmerColor
65      }
66      return result;
67    }
68    public void process(String sensorID, int signal) {
69      original(sensorID, int signal);
70      if (!controller.get(sensorID).isEmpty()) {
71        ...
72              switch (getDeviceType(deviceID)) {
73              case 4: // DimmerColor
74                switch (signal) {
75                case 1: // bright -> turnOff
76                  getDimmerColorService().turnOff(deviceID);
77                  break;
78                case 2: // dark -> turnOn
79                  getDimmerColorService().turnOn(deviceID);
80                  break;
81                default:
82                  break;
83                }
84                break;
85
86              default:
87                break;
88
89              }
90        ...
91      }
92    }
93    ...
```

Listing 4.7: Verfeinerung von ControllerServiceImpl.java in *DimmerColor*

```
1   //ControllerBase
2   package de.controller.serviceimpl;
3   ...
4   @WebService(endpointInterface="de.controller.webservice.ControllerPortType",serviceName=
        "ControllerService",portName="ControllerPort")
5   public class ControllerServiceImpl {
6     protected ControllerBean bean = ControllerBean.getIntance();
7     ...
8     public void turnOnLamp(String deviceID) {
9       if (bean.isRegDevice(deviceID)) {
10        switch (bean.getDeviceType(deviceID)) {
11        case 1: // LampSimple
12          bean.getLampSimpleService().turnOn(deviceID);
13          break;
14        case 2: // LampDimmer
15          bean.getLampDimmerService().turnOn(deviceID);
16          break;
```

```
17        default:
18          break;
19        }
20      }
21    }
22    public void changeColor(String deviceID, int color) {
23    }
24    public int getColor(String deviceID) {
25      return -1;
26    }
27    ...
28  }
29
30  //---------------
31  //DimmerColor
32    ...
33    public void turnOnLamp(String deviceID) {
34      original(deviceID);
35      if (bean.isRegDevice(deviceID)) {
36        switch (bean.getDeviceType(deviceID)) {
37        case 4: // DimmerColor
38          bean.getDimmerColorService().turnOn(deviceID);
39          break;
40        default:
41          break;
42        }
43      }
44    }
45    public void changeColor(String deviceID, int color) {
46      original(deviceID, color);
47      if (bean.isRegDevice(deviceID)) {
48        switch (bean.getDeviceType(deviceID)) {
49        case 4: // DimmerColor
50          bean.getDimmerColorService().setColor(deviceID, color);
51          break;
52        default:
53          break;
54        }
55      }
56    }
57    public int getColor(String deviceID) {
58      int result = original(deviceID);
59      if (bean.isRegDevice(deviceID)) {
60        switch (bean.getDeviceType(deviceID)) {
61        case 4: // DimmerColor
62          return bean.getDimmerColorService().getColor(deviceID);
63        default:
64          break;
65        }
66      }
67      return result;
68    }
69    ...
```

Feature-Selektion

Wir haben die *expression*-Datei (da wir mit FeatureHouse arbeiten) z.B. mit dem Inhalt „*ControllerBase SimpleColor*“ verfasst. Die entsprechende grafische Darstellung in FeatureIDE liefert Abbildung 4.6.

Feature-Komposition

Wie bei Komposition der Lampenvariante geschildert, verläuft der Kompositionsvorgang hier auch auf der gleichen Art und Weise.

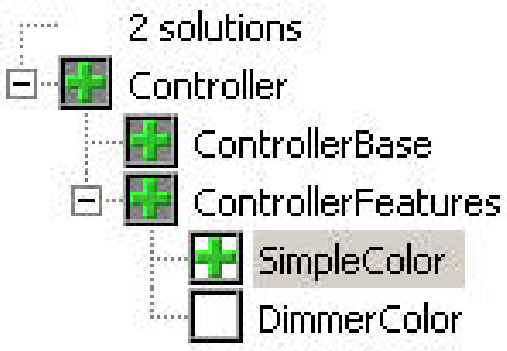


Abbildung 4.6: Feature-Selektion für Controller

4.2 Evaluierung

Im Kapitel 3 haben wir versucht, einen SOA-Kontext mit Variabilität (Demonstrator) zu schaffen und zu implementieren. Die Problemanalyse am Anfang dieses Kapitel hat gezeigt, dass es hinter der aktuellen Implementierung einige Schwächen hinsichtlich bestimmten Gesichtspunkte stecken. Als alternative Lösung zur Verbesserung der Situation stellt sich FOP. Letztes haben wir den Demonstrator dann noch einmal mit FOP realisiert, um diesen Lösungsvorschlag umzusetzen.

Wir werden nachfolgend die beiden Implementierungsansätze (*A* - ohne FOP; *B* - mit FOP) noch einmal zu Gesicht bekommen, und sie bezüglich der bekannten Problemfelder (vgl. Problemstellung im ersten Kapitel) anhand der bereits durchgeführten Problemanalyse miteinander vergleichen. Fazite, die wir daraus schlussfolgern, werden die Problemstellung bzw. Zielsetzung der Arbeit aufklären.

4.2.1 Modularität

Probleme bezüglich den querschneidenden Belangen aus der Problemanalyse zeigen, dass die Zerlegung der Software in modularen Services am Szenario, wo Variabilität bzw. Service-Varianten mit Gemeinsamkeit enthalten, nicht ganz möglich ist.

Im Ansatz B (mit FOP) lässt sich dies gut, praktikabel lösen. Alles, was z.B. zum *Color*-Feature gehört, wird im Feature-Modul *Color* eingekapselt (s. Listing 4.8). Einzelne Klassen bestehen dabei aus Rollen, die sich auf dieses Feature beziehen. Dies ermöglicht bessere „separation of concerns“. Änderung bezüglich dieses Features muss auch nur in dem entsprechenden Modul vorgenommen werden, ohne in den ganzen Paketen nach Problemstellen erst durchzuforschen.

Listing 4.8: Feature-Modul *Color*

```
1  //Bean
2  package de.lamp.beans;
3
4  public class LampBean {
5
6    public void updateDevices() {
7      NodeList nl = null;
8      try {
9        nl = loadXML(pathConfigXML);
10     } catch (SAXException e) {
```

```
11        e.printStackTrace();
12      } catch (IOException e) {
13        e.printStackTrace();
14      } catch (ParserConfigurationException e) {
15        e.printStackTrace();
16      }
17      for (int i=0; i<nl.getLength(); i++) {
18        Element lamp = (Element) nl.item(i);
19        String id = lamp.getAttribute("id");
20        int color = Integer.valueOf(lamp.getAttribute("color")).intValue();
21        devices.get(id).setColor(color);
22      }
23    }
24
25  }
26
27  //Object
28  package de.lamp.beans;
29
30  public class LampObject {
31    protected int color;
32    public int getColor() {
33      return color;
34    }
35    public void setColor(int color) {
36      this.color = color;
37    }
38  }
39
40  //ServiceImpl
41  package de.lamp.serviceimpl;
42
43  public class LampServiceImpl {
44
45    public int changeColor(String id, int color) {
46      if (id.startsWith(LampBean.ID)) {
47        obj = bean.getObject(id);
48        if (obj == null) {
49          return -1;
50        }
51        obj.setColor(color);
52        return 0;
53      }
54      return -2;
55    }
56
57    public int getColor(String id) {
58      if (id.startsWith(LampBean.ID)) {
59        obj = bean.getObject(id);
60        if (obj == null) {
61          return -1;
62        }
63        return obj.getColor();
64      }
65      return -2;
66    }
67
68  }
```

4.2.2 Variabilität

Die Problemanalyse hat sich ergeben, dass Variabilitätsimplementierung nach Ansatz A Probleme mit sich bringt. SOA bietet hierbei keinen Mechanismus, um alle Service-Varianten dynamisch zu verwalten, d.h. alle benötigten Service-Implementierungen müssen fest implementiert werden. Je mehr Variationen in Services angebotenen werden, desto unübersichtlicher wird es. Dieses erschwert auch die Wartbarkeit der Service-

Varianten.

Ansatz B erlaubt die automatische Service-Komposition nach einer Selektion der benötigten Features. Dadurch können wir eine neue Service-Variante schnell, dynamisch zusammenstellen. Das Management der Features in Feature-Modulen erleichtert auch die Wartungsarbeit, und hält eine gute Übersicht über alle Features auch bei größerem Problemumfang.

4.2.3 Uniformität

SOA liefert eine ideale Umgebung zum Testen der Plattform- bzw. Sprachunabhängigkeit eines Tools. Zur Web Services Entwicklung in der Fallstudie kommen Java-Artefakte und WSDL (XML-Artefakte) zur Verwendung. Deshalb können wir hier bezüglich der Uniformität nur eingeschränkt auf diese beiden Sprachen beurteilen. Um noch den Einsatz von FOP in anderer Umgebung zu evaluieren, sind weiere Fallstudien gefragt.

WSDL (XML-Anwendung) spielt hier in der Web Services Programmierung die wichtigste Rolle zur Service-Beschreibung, da sie den standardisierten Kommunikationsweg zur „Außenwelt“ darstellen (vgl. Kapitel 2). Modularität bzw. Variabilität in diesen können auch sehr sinnvoll sein, insbesondere beim steigendem Umfang des SOA-Projektes mit viel mehr Variantenzahl. Einem allgemeinen Mechanismus, um diese zu ermöglichen, fehlt es bis zur Nutzung von FOP.

Mit Hilfe vom *xak*-Tool (ATS) lässt sich Variabilität von Gemeinsamkeit auch hier trotz der eigenen Art zur Komposition (vgl. *Jak*-Komposition) trennen. Service-Beschreibungen der Varianten können auf besseren Weg gemanagt werden.

Einen kleinen Nachteil stellt aber das derzeitige *xak*-Tool dar, und zwar, manuelle Eingriffe müssen in manchen Fällen noch getätigt werden. Hierzu könnten weitere Sprachelemente entwickelt werden.

4.2.4 Spezifikation und Kompatibilität

Feature-Modellierung nach FODA-Methode (vgl. Kapitel 2) wurde eingesetzt, um Produktvariante zu spezifizieren. Anhand vom Feature-Diagramm lassen sich alle Service-Varianten auch problemlos modellieren (hier: Lampen, Controller).

Damit ein Diagramm jedoch allgemein verständlich und lesbar erscheint, bedarf es einer internen Abmachung zur Begriffswahl. Beispielsweise wurde das letzte Diagramm zu Service-Varianten der Controller etwas flach gehalten (s. Abbildung 4.4), *Controller-Base* umfasste dort intern noch *LampSimple*, *LampDimmer* und *IntensitySensor* (vgl. *ControllerBasic*-Variante der letzten Implementierung im Kapitel 3). Aufgrund von „separation of concerns“ kann man dies z.B. wie in Abbilung 4.7 modellieren.

Auch die Modellierung von Abhängigkeiten lässt sich verwirklichen. Beispielsweise sollte *IntensitySensor* sollte nur mit *LampSimple* und *LampDimmer* zusammenarbeiten. Dort würde man den Sensor unterhalb *LampBase* als deren Feature unterbringen.

Aufgrund der Komplexität einer formalen Kompatibilitätsprüfung (vgl. *typeability* in [AKL08]) werden wir diesen Punkt nicht in dieser Arbeit weiter behandeln. Hierzu bedarf es künftig einer weiteren Fallstudie.

(a) Erweitertes Feature-Diagramm (b) Feature-Selektion

Abbildung 4.7: Produktlinie der Controller

4.2.5 Unterstützungswerkzeuge

Für den Einsatz von FOP in dieser Arbeit haben wir unterschiedliche Tools verwendet, zum einen AHEAD Tool Suite (*xak*), und zum anderen FeatureHouse, da sie auch eigene Stärken anbieten. Deshalb ist die Vorgehensweise noch umständlich. Die Nutzung von FeatureIDE ist hier relativ eingeschränkt, da sich die Einbindungsschnittstelle für FeatureHouse in FeatureIDE zum Zeitpunkt dieses Schreibens noch in der Entwicklung bzw. Erprobung befindet.

4.3 Zusammenfassung

Somit ist dieses Kapitel beendet. Hier haben wir mit der FOP-Umsetzung des im vorigen Kapitel vorgestellten Demonstrators nach dem SPL-Framework begonnen. Es folgte die Evaluation hinsichtlich der Problemstellung und Zielsetzung der Arbeit aus dem ersten Kapitel. Diese schloss dann dieses Hauptkapitel ab.

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit konfrontierten wir mit den Herausforderungen aus [AKL08] für eine Evaluation des Einsatzes feature-orientierter Programmierung (FOP) zur Implementierunng von Variabilität in service-orientierten Architekturen (SOA), konkret an einem selbst gewählten Szenario. Diese Herausforderungen zu evaluieren war zum größten Teil unsere Zielsetzung, welche wir nach der Arbeit erreichen wollten.

Die Arbeit begann zum Einstieg mit einem einleitenden Kapitel, in dem wir die Motivation gezeigt, Problemstellung beschrieben, Ziel definiert bzw. die Arbeit strukturiert haben.

Das nächste Kapitel haben wir genutzt, um uns mit den Grundlagen zu SOA, SPL sowie FOP zu beschäftigen. Diese sollten der Verständlichkeit des gesamten Inhalts von dieser Arbeit dienen. Dort wurden zu jedem einzelnen Themengebiet Definitionen, Begriffe, Methoden, Frameworks, Tools, etc. im Zusammenhang eingeführt, welche wir im weiteren Verlauf der Arbeit verwendet haben.

Danach stellten wir in einem anderen Kapitel ein ausgewähltes Fallbeispiel mit variablen Sachverhalten im Domotic-Kontext vor. Wir gaben dort auch einen kleinen Blick auf die reale Domotic-Welt, um unseren Demonstrator noch realitätsnah darzustellen. Anschließend implementierten wir diesen mit Java Web Services zum Aufbau einer SOA-Umgebung. Die Problemanalyse am Ende dieses Kapitels fasste alle Probleme zusammen, die während der Implementierung in Frage kamen, und diskutierte über diese Probleme, um einen möglichen geeigneten Lösungsvorschlag zu finden.

Der erste Teil im nächsten Kapitel zeigte die nochmalige Umsetzung des Szenarios mit FOP-Einsatz. Im zweiten Teil haben wir diese Implementierung anhand der vorigen Problemanalyse genauer unter die Lupe genommen, um dann ihren Einsatz zu evaluieren.

Das letzte Kapitel der Arbeit, dieses Kapitel, nutzen wir, um den Inhalt der Arbeit grob zusammenzufassen. Dies gilt als der Wegweiser für unsere Arbeit von der Motivation bis zur Zielerreichung.

Die Evaluierung feature-basierter service-orientier Architekturen bei der Variabilitätsimplementierung hat sich ergeben, dass die erwarteten Effekte in [AKL08] bezüglich der Modularität, Variabilität, Uniformität und Spezifikation nach diesem Einsatz zutreffen. Die Kompatibilitätsprüfung haben wir im Rahmen dieser Arbeit nicht behandelt. Diese könnte durchaus ein interessantes Thema für weitere Arbeiten später werden.

Literaturverzeichnis

[ACKM04] Alonso, G.; Casati, F.; Kuno, H.; Machiraju, V.: *Web Services - Concepts, Architectures and Applications*. Springer Verlag, 2004.

[ADT07] Anfurrutia, F. I.; Díaz, O.; Trujillo, S.: On Refining XML Artifacts. In *Lecture Notes in Computer Science (LNCS) 4607*, S. 473–478. Springer Verlag, 2007.

[AKL08] Apel, S.; Kästner, C.; Lengauer, C.: Research Challenges in the Tension Between Features and Services. In *Proc. of ICSE Workshop on Systems Development in SOA Environments (SDSOA)*, S. 53–58. ACM, 2008.

[AKL09] Apel, S.; Kästner, C.; Lengauer, C.: FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. of the 31th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009.

[ALMK08] Apel, S.; Lengauer, C.; Möller, B.; Kästner, C.: An Algebra for Features and Feature Composition. In *Proc. of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, S. 36–50. Springer Verlag, 2008.

[Ape07] Apel, S.: *The Role of Features and Aspects in Software Development*. Dissertation, School of Computer Science, University of Magdeburg, 2007.

[Bat05] Batory, D.: Feature Models, Grammars, and Propositional Formulas. In *Proc. of Software Product Line Conference (SPLC)*, S. 4, 2005.

[Bat06] Batory, D.: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Lecture Notes in Computer Science (LNCS) 4143*, S. 3–35. Springer Verlag, 2006.

[BCK03] Bass, L.; Clements, P.; Kazman, R.: *Software Architecture in Practice*. Addison-Wesley, 2. Auflage, 2003.

[BSR04] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (IEEE TSE)*, 2004.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming - Methods, Tools, and Applications*. Addision-Wesley, 2000.

[Che06] Cherry, N.: *Linux Smart Homes for Dummies*. Wiley Publishing, 2006.

[CN01] Clements, P.; Northrop, L.: *Software Product Lines : Practices and Patterns*. Addison-Wesley, 2001.

[Dum03] Dumke, R.: *Software Engineering - Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools*. Vieweg Verlag, 4. Auflage, 2003.

[Erl04] Erl, T.: *Service-Oriented Architecture - A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, 2004.

[Erl05] Erl, T.: *Service-Oriented Architecture - Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[FTW07] Frotscher, T.; Teufel, M.; Wang, D.: *Java Web Services mit Apache Axis2*. entwickler.press, 2007.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addision-Wesley, 1995.

[Gro01] Grosso, W.: *Java RMI*. O'Reilly, 2001.

[Has05] Hasselbring, W.: Software-Architektur. *Informatik Spektrum 29(1)*, S. 48–52, 2005.

[KCH+90] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University*, 1990.

[KTS+09] Kästner, C.; Thüm, T.; Saake, G.; Feigenspan, J.; Leich, T.; Wielgorz, F.; Apel, S.: FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proc. of the 31th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009.

[Mer04] Mertens, P.: *Integrierte Informationsverarbeitung 1 - Operative Systeme in der Industrie*. Springer Verlag, 14. Auflage, 2004.

[Oes01] Oestereich, B.: *Objektorientierte Softwareentwicklung - Analyse und Design mit der Unified Modeling Language*. Oldenbourg Verlag, 5. Auflage, 2001.

[OHE99] Orfali, R.; Harkey, D.; Edwards, J.: *Instant CORBA - Führung durch die CORBA-Welt*. Addision-Wesley, 1999.

[Osi08] Osipov, M.: Home Automation with ZigBee. In *Lecture Notes in Computer Science (LNCS) 5174*, S. 263–270. Springer Verlag, 2008.

[OW01] Oaks, S.; Wong, H.: *Jini In a Nutshell - Deutsche Ausgabe für Jini 1.0 und 1.1*. O'Reilly, 2001.

[PBvdL05] Pohl, K.; Böckle, G.; Linden, F. v. d.: *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer Verlag, 2005.

[PR05] Pasetti, A.; Rohlik, O.: Concept for the XFeature Tool. Technischer Bericht, P&P Software GmbH and ETH-Zurich, 2005.

[Pre97] Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. *ECOOP '97*, 1997.

[RR07] Richardson, L.; Ruby, S.: *RESTful Web Services*. O'Reilly, 2007.

[RS03] Rautenstrauch, C.; Schulze, T.: *Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker*. Springer Verlag, 2003.

[STK02] Snell, J.; Tidwell, D.; Kulchenko, P.: *Webservice-Programmierung mit SOAP*. O'Reilly, 2002.

[vdLSR07] Linden, F. v. d.; Schmid, K.; Rommes, E.: *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer Verlag, 2007.

[WBFT04] Wang, D.; Bayer, T.; Frotscher, T.; Teufel, M.: *Java Web Services mit Apache Axis*. entwickler.press, 2004.

[ZTP03] Zimmermann, O.; Tomlinson, M.; Peuser, S.: *Perspectives on Web Services - Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer Verlag, 2003.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 22. Juni 2009

Chau Le Minh