Universität Passau

Fakultät für Informatik und Mathematik

Bachelorarbeit

Feature-orientierte Analyse von Service-orientierten Architekturen

Verfasser:

Rolf Daniel

18. Juni 2009

Betreuer:

Dr.-Ing. Sven Apel

Universität Passau
Fakultät für Informatik und Mathematik
Innstraße 33, D–94032 Passau

Daniel, Rolf:
Feature-orientierte Analyse von Service-orientierten Architekturen
Bachelorarbeit, Universität Passau, 2009.

Zusammenfassung

In dieser Bachelorarbeit geht es um eine Feature-orientierte Analyse von Service-orientierten Architekturen. Die grundlegenden Ideen hierfür stammen aus [AKL08].

Unter dem Begriff *Software Architektur* versteht man die Beschreibung der grundlegenden Komponenten eines Softwaresystems und deren Zusammenspiel. Die *Service-Oriented Architecture (SOA)* ist eine der viel versprechendsten Software Architekturen. Durch sie wird ein allgemeiner Ansatz zur Realisierung komplexer Systeme und der Abbildung von Geschäftsprozessen in solchen Systemen beschrieben.

Eine SOA zerlegt ein Softwaresystem in mehrere Services. Diese Services besitzen ein Interface, mit dem sie eine gewisse Funktionalität nach außen hin anbieten und Details ihrer Implementierung verbergen. Services, die in verschiedenen Programmiersprachen geschrieben und veröffentlicht wurden, können in einer gemeinsamen Service-Infrastruktur zusammengefasst werden.

Ein möglicher Ansatz zur Umsetzung solcher Architekturen sind *Web Services*. Sie bieten gemeinsam die Grundfunktionalität eines Systems an. Es stellt sich jedoch heraus, dass es gewisse Funktionalitäten gibt, die sich nicht innerhalb eines Services kapseln lassen. Zur Lösung dieses Problems werden *Features* benötigt. Sie können die Struktur eines Programms erweitern und modifizieren und bieten Konfigurationsmöglichkeiten an. Die zusätzliche Funktionalität liegt also verstreut in Form von Features vor.

Es werden Services und Features miteinander verglichen. Obwohl die beiden Konzepte große Gemeinsamkeiten bei der Konstruktion von Softwaresystemen aufweisen, lässt sich feststellen, dass Services und Features nicht dasselbe sind. Dabei ist auch herausgekommen, dass Features oft mehrere Services gleichzeitig betreffen.

Ein weiteres Ergebnis dieser Arbeit ist unter Anderem, dass es möglich ist Features aus einem System, bestehend aus mehreren Web Services, herauszukristallisieren. Die konkrete Fallstudie kann umgebaut werden, indem man die gefundenen Features modularisiert. Dies erleichtert die Entwicklung von *Varianten*. Unterschiedliche Varianten eines Softwaresystems können durch Features beschrieben und getrennt werden.

In der Zukunft kann man aus den Erfahrungen dieser Arbeit ein Konzept entwickeln für die Softwareentwicklung, auf Basis von Services und Features. Damit könnten sich einige Probleme, wie z.B. Modularität oder Kompatibilität, die keines der Konzepte einzeln bewältigen kann, lösen lassen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

AOP . . . . . Aspect-Oriented Programming

FOP . . . . . Feature-Oriented Programming

FTP . . . . . File Transfer Protocol

HTTP . . . . . Hypertext Transfer Protocol

IBM . . . . . International Business Machines Corporation

IDE . . . . . Integrated Development Environment

IT . . . . . Information Technology

JSP . . . . . JavaServer Pages

PC . . . . . Personal Computer

PHP . . . . . PHP: Hypertext Preprocessor

SMTP . . . . . Simple Mail Transfer Protocol

SOA . . . . . Service-Oriented Architecture

SOAP . . . . . Simple Object Access Protocol

SPL . . . . . Software Product Line

TCP . . . . . Transmission Control Protocol

UBR . . . . . UDDI Business Registry

UDDI . . . . . Universal Description, Discovery and Integration

URI . . . . . Uniform Resource Identifier

WSDL . . . . . Web Service Description Language

XML . . . . . Extensible Markup Language

Kapitel 1

Einleitung

1.1 Motivation

Wenn man, z.B. im Internet, einmal nach Systemen sucht, die Service-orientiert entwickelt wurden, lässt sich eine Vielzahl davon finden. Aber, warum ist das so? SOA ist einer der modernsten Trends der *Information Technology (IT)* und aufgrund seiner geschäftsprozessnahen Ausrichtung insbesondere im Bereich des Management sehr beliebt. Dieser Softwarearchitekturstil ist nicht nur für sehr große verteilte Systeme geeignet, sondern auch für die Strukturierung von kleinen Applikationen, die nur auf einen lokalen Bereich beschränkt sind.

Einer der größten Vorteile einer SOA ist, dass die Services und die sie benutzenden Anwendungen in grundverschiedenen Programmiersprachen und Plattformen implementiert sein können. Ein Service könnte z.B. in *.NET* implementiert sein, seine Schnittstelle in einem Verzeichnisdienst veröffentlichen und dann über ein bestimmtes Protokoll mit einem Java- oder *PHP: Hypertext Preprocessor (PHP)*-Client kommunizieren und Aufrufe seiner Funktionen ermöglichen.

Es gibt bereits einige Forschungen, die sich mit dem Konzept der SOA befassen. Dabei ist man jedoch noch auf einige Probleme gestoßen, was z.B. die Modularität oder Variabilität von Services anbelangt [AKL08]. Erstellt man ein Service-orientiertes System, kommt man schnell zu dem Ergebnis, dass es Funktionalitäten gibt, die man nicht so einfach in einem einzelnen Service kapseln kann, wie bereits in der Zusammenfassung beschrieben wurde.

Die *Feature-Oriented Programming (FOP)* könnte ein viel versprechender Versuch sein die Probleme, zusammen mit dem Konzept des Features, zu lösen [AKL08]. Das Querschneiden von Features führt zu einer nicht optimalen Systemstruktur und erschwert die Wartbarkeit und Weiterentwicklung von Sofwaresystemen. FOP ist in der Lage, den Quelltext, der verstreut in den Services vorliegt, zu modularisieren [AKL08]. Dadurch wird nicht nur die Lesbarkeit des Codes verbessert, man kann mit diesen Feature-Modulen auch verschiedene Varianten zusammenstellen und so ganz einfach neue Software erstellen.

Mit Hilfe von Web Services soll nun ein kleines Flugbuchung-Szenario erstellt und obige Aussagen überprüft werden.

1.2 Zielstellung

Ziel dieser Bachelorarbeit ist eine Feature-orientierte Analyse einer Service- orientierten Architektur. Da es aber keinen Zugang zur Implementierung eines bereits

bestehenden Service-orientierten Systems gibt und somit eine Analyse von querschneidenden Features unmöglich ist, soll selbst ein gebräuchliches Beispielsystem erstellt werden.

Anhand dieses Systems soll dann eine Feature-orientierte Analyse durchgeführt werden und es soll untersucht werden, in wie weit sich Features aus diesem System herauskristallisieren lassen. Wenn sich Features finden lassen, soll eine Annotation im Quelltext erfolgen. Dadurch wird eine spätere Modularisierung der Features vereinfacht. Desweiteren soll das Zusammenwirkung dieser beiden Ansätze analysiert und beurteilt werden, sowie die daraus entstehenden Nutzen, aber auch die damit verbundenen Probleme evaluiert werden.

Die Ergebnisse dieser Arbeit sollen für die Zukunft die Erarbeitung eines Konzepts für die Softwareentwicklung auf der Grundlage von Services und Features unterstützen.

1.3 Gliederung der Arbeit

Nach einer kurzen Einleitung durch Motivation und Zielstellung dieser Arbeit werden zunächst alle für das Verständis notwendigen Grundlagen in Kapitel 2 vermittelt. Dazu gehören unter Anderem Web Services, das *Simple Object Access Protocol (SOAP)*, die *Web Service Description Language (WSDL)* und der Verzeichnisdienst *Universal Description, Discovery and Integration (UDDI)*. Desweiteren wird ein Überblick zur Feature-Orientierung gegeben, der die Begriffe Produktlinien, Features und FOP beinhaltet.

Darauf folgt Kapitel 3 in dem die Ähnlichkeiten bzw. Unterschiede von Features und Services, sowie deren Zusammenspiel, analysiert werden. Es werden einige Vorteile, aber auch ein paar Herausforderungen, vorgestellt, welche die beiden Konzepte mit sich bringen.

Anschließend wird in Kapitel 4 ein Service-orientiertes Szenario, das als Fallstudie dient, und die damit erzielten Ergebnisse vorgestellt.

Am Ende der Arbeit wird dann eine Zusammenfassung der Resultate aufgeführt und ein kurzer Ausblick in weiterführende Forschungen, die auf dieser Arbeit aufbauen könnten, gegeben.

Kapitel 2

Grundlagen

2.1 Service-Orientierung

In diesem Abschnitt des Grundlagenkapitels sollen folgende Ausführungen einen kurzen Einblick in die Service-orientierte Programmierung geben.

2.1.1 Web Services

> „Ein Web Service ist eine durch einen *Uniform Resource Identifier (URI)* eindeutig identifizierte Softwareanwendung, deren Schnittstellen als *Extensible Markup Language (XML)*-Artefakte definiert, beschrieben und gefunden werden können. Ein Web Service unterstützt die direkte Interaktion mit anderen Softwareagenten durch XML-basierte Nachrichten, die über Internetprotokolle ausgetauscht werden.“ [1]

Zur Zeit sind Web Services der wohl aussichtsreichste Versuch, eine Service-orientierte Architektur umzusetzen und können daher als technische Realisierung einer SOA gesehen werden.

Sucht man in der Literatur nach Web Services, lassen sich unterschiedliche Auffassungen finden. Je nach Anbieter variieren die verschiedenen Definitionen:

IBM: „Web services are self-contained, modular applications that can be described, published, located, and invoked over a network, generally, the World-Wide Web.“ [2]

Microsoft: „A Web service is programmable application logic, accessible using standard Internet protocols.“[3]

Gartner Group: „A software component that represents a business function (or a business service) and can be accessed by another application (a client, a server or another Web service) over public networks using generally available ubiquitous protocols and transports (i.e. SOAP over HTTP).“[4]

[1] [AE03, Seite 67]
[2] [AE03, Seite 66]
[3] [AE03, Seite 66]
[4] [AE03, Seite 66]

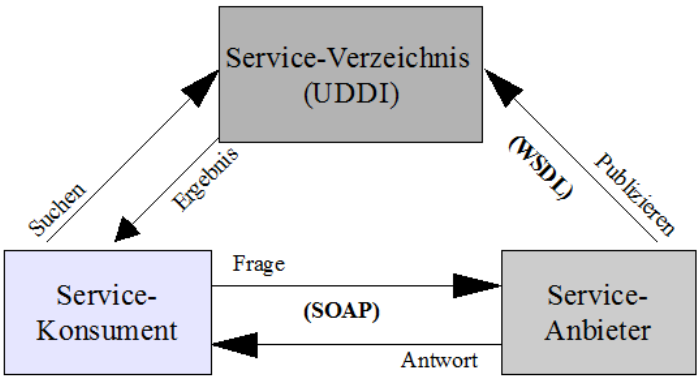

Abbildung 2.1: Funktionsweise eines Web Services

Hier werden Web Services folgendermaßen festgelegt:
Web Services

- basieren auf modernen und offenen Standards
- können über eine URI identifiziert werden
- besitzen eine klar definierte Schnittstelle
- bieten Dienste an
- tauschen Daten über ein XML-basiertes Protokoll aus (hier SOAP)
- sind unabhängig von Programmiersprache, Middleware, Hardware und Betriebssystem
- sind für Computer-Computer-Kommunikation gedacht

Vereinfacht kann man sagen, durch einen Web Service wird von einem Anbieter ein Dienst bereitgestellt, der von anderen Anwendungen angesprochen und genutzt werden kann. Die Veröffentlichung eines solchen Dienstes erfolgt durch den Verzeichnisdienst UDDI. Mit Hilfe der WSDL werden die von einem Web Service unterstützten Methoden und deren Parameter beschrieben. Die Kommunikation findet über SOAP statt.

Die Funktionsweise lässt sich anhand von Abb. 2.1 sehr gut veranschaulichen. Der Anbieter veröffentlicht in einem Verzeichnis die Beschreibung seiner Dienste. Der Konsument kann nun dieses Verzeichnis durchsuchen und den gewünschten Dienst auswählen. Daraufhin findet eine dynamische Anbindung des Konsumenten an den Anbieter statt und der Konsument kann auf die Methoden des Dienstes zugreifen. Dabei bilden die drei XML-basierten Standards SOAP, WSDL und UDDI die Grundlage. Sie sollen im Folgenden näher vorgestellt werden.

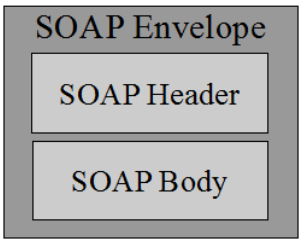

Abbildung 2.2: SOAP-Nachrichtenformat

2.1.2 SOAP - Simple Object Access Protocol

SOAP ist ein auf Basis von XML definiertes Protokoll, das zum Austausch von Daten mit Hilfe von Nachrichten genutzt wird. Zum Senden von Nachrichten können beliebige Transportprotokolle verwendet werden, beispielsweise *File Transfer Protocol (FTP)*, *Simple Mail Transfer Protocol (SMTP)* oder *Hypertext Transfer Protocol (HTTP)*. Am meisten verbreitet ist die Kombination SOAP mit HTTP und *Transmission Control Protocol (TCP)*.

Ursprünglich stand SOAP für *Simple Object Access Protocol*, aber inzwischen wird diese Abkürzung seit Version 1.2 nicht mehr verwendet, da sie als unzutreffend angesehen wird.

Eine SOAP-Nachricht ist wie folgt aufgebaut: Sie besteht zunächst einmal aus einem Wurzelelement, dem sogenannten *SOAP-Envelope*. Es bildet eine Art Umschlag für die beiden Elemente *SOAP-Header* und *SOAP-Body*[5]. Diese Struktur ist in Abb. 2.2 dargestellt.

Das *Header-Element* ist optional. In diesem können Verwaltungsinformationen und Metadaten, beispielsweise zum Routing oder zur Verschlüsselung, angeführt werden. Das obligatorische *Body-Element* enthält die eigentlichen Nutzdaten. Darin können sowohl Informationen zum Datenaustausch, als auch Anweisungen für einen entfernten Prozeduraufruf stehen.

Für die XML-Kodierung von SOAP-Nachrichten wird ein eigener Namensraum für alle Tags benutzt, die zum *Envelope* gehören. Normalerweise wird als Namensraum *env* verwendet [6]. Daraus ergibt sich der in List. 2.1[7] angeführte Rahmen für eine SOAP-Nachricht.

2.1.3 WSDL - Web Service Description Language

Über die WSDL werden die angebotenen Funktionen, Daten, Datentypen und Austauschprotokolle eines Web Services beschrieben. Diese Informationen werden in einem WSDL-Dokument abgelegt. Mit Hilfe dieses Dokuments können angebundene Anwendungen erkennen, welche Operationen, sowie deren Parameter und Rückgabewerte, von außen zugänglich sind.

Ähnlich wie bei einer SOAP-Nachricht, besteht das WSDL-Dokument zunächst

[5] [AE03, Seite 178]
[6] [AE03, Seite 179]
[7] [AE03, Seite 179]

```
<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
  ...
  </env:Header>
  <env:Body>
  ...
  </env:Body>
</env:Envelope>
```

Listing 2.1: XML-Darstellung einer SOAP-Nachricht

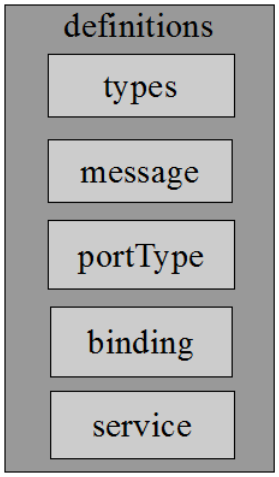

Abbildung 2.3: Aufbau eines WSDL-Dokuments

einmal aus einem Wurzelelement, das in List. 2.2 zu sehen ist. Es dient zur Definition eines Namensraums und umschließt die in Abb. 2.3 gezeigten Elemente.

Die Parameter, mit denen die Operationen des Web Services aufgerufen werden, können komplexe oder einfache Datentypen besitzen. Diese Typen werden im *Types-Element* beschrieben (siehe List. 2.3). Die Zeilen 2 - 9 beinhalten die Definition der Anfrageparameter. Der erste Parameter hat den Typ *int* und entspricht einer Währung, in die der zweite Parameter, der den Typ *double* besitzt, umgerechnet werden soll. In den Zeilen 10 - 14 wird der Rückgabeparameter definiert, der auch vom Typ *double* ist und dem umgerechneten Betrag entspricht.

Die SOAP-Nachrichten, die der Web Service bei seiner Kommunikation mit anderen Anwendungen verwendet, werden im *Message-Element* festgelegt. In List. 2.4 werden zwei Nachrichten definiert. Eine Anfragenachricht, die die Eingabeparamter festlegt (Zeile 1 - 3) und eine Antwortnachricht, die die Ausgabeparamter

```
<definitions ...>
  ...
</definitions>
```

Listing 2.2: Wurzelelement *definitions*

```
<types>
  <element name="calculateCurrency">
    <complexType>
      <element name="toCurrency" type="int"/>
    </complexType>
    <complexType>
      <element name="amount" type="double"/>
    </complexType>
  </element>
  <element name="calculateCurrencyResponse">
    <complexType>
      <element name="return" type="double"/>
    </complexType>
  </element>
</types>
```

Listing 2.3: Beschreibung der Parametertypen

```
<message name="calculateCurrency">
  <part name="parameters" element="tns:calculateCurrency"></part>
</message>
<message name="calculateCurrencyResponse">
  <part name="parameters" element="tns:calculateCurrencyResponse"></
     part>
</message>
```

Listing 2.4: Beschreibung der SOAP-Nachrichten

festlegt (Zeile 4 - 6).

Ein weiterer Bestandteil des WSDL-Dokuments, ist die Definition von Funktionen und Kommunikationstypen des Web Services. Diese Aufgabe übernimmt das *PortType-Element*, wie in List. 2.5 abgebildet. Hier wird eine *Request-Response-Operation* definiert: Ein Client sendet eine Anfragenachricht (Zeile 3) an den Web Service und empfängt daraufhin eine Antwortnachricht (Zeile 4) von ihm.

Im *Binding-Element* wird das Nachrichtenformat und das Protokoll, das der Web Service verwenden soll, definiert. In List. 2.6 wird beispielsweise *HTTP* als Protokoll festgelegt (Zeile 2), sowie die Ein- und Ausgabekodierung (Zeile 6 und 9).

Letzter Teil des WSDL-Dokuments ist das *Service-Element*. Hier werden In-

```
<portType name="WaehrungskonversionWS">
  <operation name="calculateCurrency">
    <input message="tns:calculateCurrency"></input>
    <output message="tns:calculateCurrencyResponse"></output>
  </operation>
</portType>
```

Listing 2.5: Beschreibung der Operationen

```
<binding ...>
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      ...></soap:binding>
  <operation name="calculateCurrency">
    ...
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
  </operation>
</binding>
```

Listing 2.6: Beschreibung von Nachrichtenformat und Protokoll

formationen, die beim Zugriff auf den Web Service benötigt werden, wie z. B. Netzwerkadresse und Portnummer, definiert.

In List. 2.7 ist der Java-Code des Web Services[8] aufgelistet, der zu diesem WSDL-Dokument gehört. Die Schnittstelle des Web Service wird mit Annotationen deklariert:

- **@WebService()** spezifiziert die Klasse als Web Service
- **@WebMethod()** kennzeichnet die Methoden der Service-Schnittstelle
- **@WebParam()** kennzeichnet die Parameter, die der Service als Eingabeparameter benötigt.

2.1.4 UDDI - Universal Description, Discovery and Integration

UDDI[9] ist ein standardisierter Verzeichnisdienst, der sowohl von Dienstanbietern verwendet wird, um ihre Services zu veröffentlichen, als auch von Dienstkonsumenten, um einen für ihre Bedürfnisse passenden Service zu finden. Die ursprüngliche Idee war, Geschäftsprozesse zwischen Unternehmen zu automatisieren und ohne menschlichen Kontakt stattfinden zu lassen. Allerdings konnte sich dieser Ansatz in der Praxis nicht durchsetzen und man kann heute UDDI eher als Management von Web Service Metadaten ansehen.

Erstmals vorgestellt wurde dieses Konzept von den drei Firmen *Microsoft*, *International Business Machines Corporation (IBM)* und *SAP* im Jahre 2000 durch das *UDDI Business Registry (UBR)* Projekt[10]. Sie sind bzw. waren auch zugleich die größten Unterstützer von UDDI bis Anfang 2006, als die Firmen ankündigten, das Projekt einzustellen.[11] Inzwischen wird UDDI von *Organisation for the Advancement of Structured Information Standards (OASIS)* standardisiert.

[8] Web Service ist Teil der Fallstudie
[9] http://uddi.org
[10] [AE03, Seite 301]
[11] Offiziell wird dies damit begründet, dass man nur die Robustheit und Interoperabilität von UDDI überprüfen wollte und man dies nun erreicht habe.

```
@WebService()
public class WaehrungskonversionWS {
  private double prize;
  @WebMethod(operationName = "calculateCurrency")
  public Double calculateCurrency(
          @WebParam(name = "toCurrency") int toCurrency,
          @WebParam(name = "amount") double amount) {
    prize = 0.0D;
    double exchangeRateGBP = 0.8826;
    double exchangeRateUSD = 1.2878;
    try {
      if (toCurrency == 2) {
        prize = amount * exchangeRateUSD;
      } else if (toCurrency == 3) {
        prize = amount * exchangeRateGBP;
      }
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    return prize;
  }
}
```

Listing 2.7: Java-Code eines Web Services

Die Informationen, die man von einer UDDI-Registry in Form von UDDI-Einträgen bekommt, kann man in drei verschiedene Kategorien einteilen [12]:

White Pages: Sie sind wie ein Telefonbuch und enthalten nur Basisinformationen über ein Unternehmen, das sich bei UDDI registriert hat, z.B. Adresse oder Telefon- und Faxnummer.

Yellow Pages: Sie entsprechen den allgemein bekannten *Gelben Seiten* und sind nach Branchen eingeteilt. Unternehmen sind sowohl in den White Pages als auch in den Yellow Pages vertreten.

Green Pages: Sie sind zuständig für die technische Beschreibung (Schnittstellenbeschreibung) der Web Services, die ein Unternehmen anbietet.

2.1.5 SOA - Service Oriented Architecture

Die Philosophie, die sich hinter Web Services verbirgt, nennt sich *Service-orientierte Architektur*. Erstmalig wurde dieser Begriff von einem Markforschungsunternehmen namens Gartner[13] genutzt. Es gilt daher auch als Erfinder der SOA. Eine allgemein akzeptierte Definition gibt es nicht, aber oft wird die in *OASIS Reference Model for Service Oriented Architecture 1.0* vorgestellte Definition zitiert:

[12] [AE03, Seite 304]
[13] http://www.gartner.com

> „Service-Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.“ [14]

Die drei wichtigsten Konzepte von SOA werden im Folgenden verdeutlicht:

Virtualisierung/Sprachenunabhängigkeit: Es existieren eine Vielzahl an unterschiedlichsten Programmierplattformen, die in Zukunft miteinander interagieren sollen. Die grundlegenden Konzepte von SOA haben als Ziel, die bestehenden Probleme bei der Integration und Interaktion dieser unterschiedlichen Teilsysteme zu lösen und Services, die in verschiedenen Programmiersprachen geschrieben und veröffentlicht wurden, in einer gemeinsamen Service-Infrastruktur zusammenzufassen.

Verteilung: Ein Anwendungsbeispiel hierfür ist eine Unternehmenslandschaft. Hier können verschiedene Unternehmen, also Service-Anbieter, gewisse Dienste, in Form von Web Services, anbieten, welche dann von Kunden, den Service-Konsumenten, genutzt werden können. Den Kunden werden diese Services zugänglich gemacht, indem sie von den Unternehmen in einem Verzeichnisdienst veröffentlicht werden. Der Kunde ist somit in der Lage, Services von verschiedenen Unternehmen, in seine eigenen Programme einzubauen. Es entsteht somit eine verteilte Funktionalität.

Entkopplung: Der Service-Konsument arbeitet aber nicht direkt mit einer Service-Implementierung zusammen, sondern sucht zunächst in einem Verzeichnisdienst nach verfügbaren Service-Anbietern, die die von ihm gewünschte Funktionalität anbieten. Erst dann greift er mittels den vom Verzeichnisdienst übermittelten Informationen auf einen Web Service zu. Dadurch wird eine lose Kopplung der einzelnen Komponenten erreicht.

Eine besondere Rolle in Service-orientierten Architekturen spielt dabei die Orientierung an Geschäftsprozessen. Dies soll anhand eines Beispiels - Kunde bestellt Artikel bei einem Versandhändler - verdeutlicht werden.

Es gibt jeweils einen Dienst für jeden Geschäftsprozessschritt. Die Dienste können in unterschiedlichen Programmiersprachen implementiert sein und auf unterschiedlichen Systemen laufen. So könnte beispielsweise die Zahlungsfähigkeit eines Kunden von einem unabhängigen Finanzdienstleister durchgeführt werden oder der Versand der bestellten Artikel durch einen Logistikdienstleister erfolgen.

Der Ablauf einer Bestellung muss nicht notwendigerweise so sequentiell erfolgen, wie in Abb. 2.4 gezeigt. Oft scheitern einige der Geschäftsprozessschritte, weil z.B. der gewünschte Artikel nicht im Lager vorhanden ist oder der Zahlungseingang noch nicht erfolgt ist. Dies führt zu Verzweigungen, die entsprechend verarbeitet werden müssen und auf die hier nicht weiter eingegangen werden soll.

Obwohl ein Dienst von unterschiedlichen Prozessen genutzt wird, muss sichergestellt werden, dass beispielsweise die Verfügbarkeitsprüfung immer dieselbe ist.

[14] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

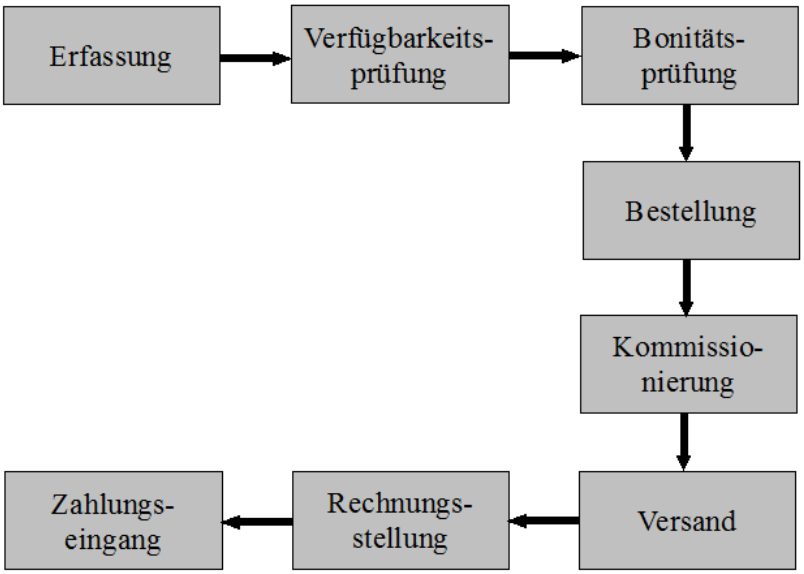

Abbildung 2.4: Service Architektur eines Warenhauses

Dadurch können die einzelnen Dienste besser gepflegt werden und es wird mehr Einheitlichkeit erreicht. So muss ein Dienst nur einmal implementiert werden und er muss nicht angepasst werden, falls sich Geschäftsprozesse ändern. Das spart Zeit und Geld.

Möchte man nun seine Waren nicht mehr mit der Deutschen Post sondern mit einem anderen Logistikunternehmen verschicken, kann man einfach einen anderen Anieter aufrufen und es müssen keine Änderungen an der Infrastruktur vorgenommen werden.

2.2 Feature-Orientierung

Da die *Objekt-orientierte Programmierung* durchaus Schwächen aufweist, die z.B. in [Ape08] genauer beschrieben werden, versucht man mit Hilfe von modernen Programmierparadigmen diese zu lösen. Eines dieser Programmmierparadigmen, die *Feature-orientierte Programmierung*, kann einen Teil dieser Probleme lösen und soll im Folgenden näher betrachtet werden.

2.2.1 Produktlinien

Der Begriff *Software-Produktlinie* wurde eingeführt vom *Software Engineering Institute (SEI)* der *Carnegie Mellon University* in Pittsburgh:

> „A *Software Product Line (SPL)* is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are

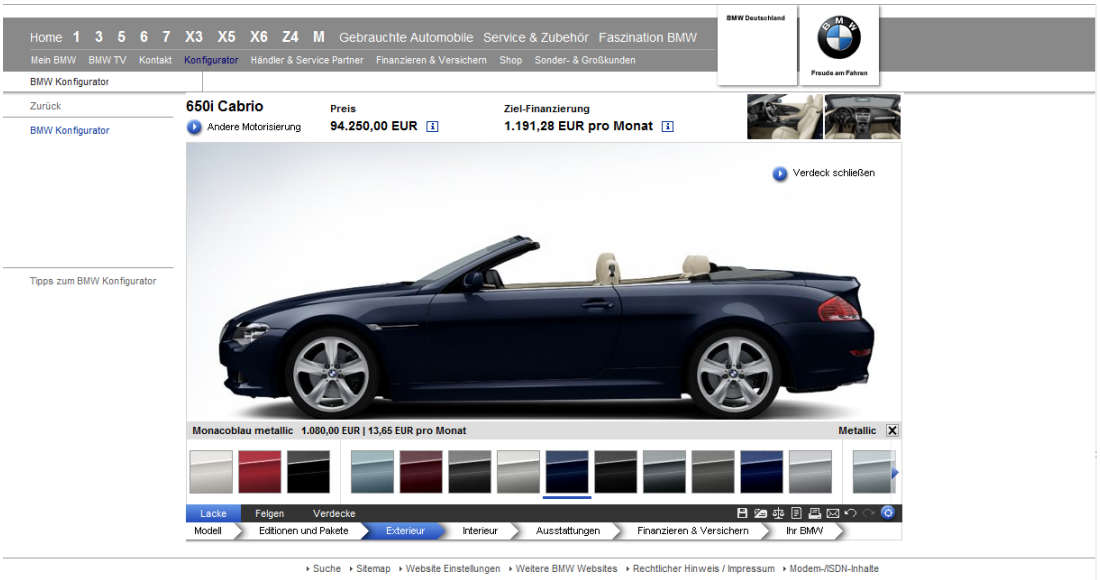

Abbildung 2.5: Produktlinie – Auto
[Quelle: http://www.bmw.de]

developed from a common set of core assets in a prescribed way.“ [15]

In dieser Definition tritt ein neuer Begriff auf: das *Feature*. Was ein Feature genau ist, wird in Kapitel 2.2.2 näher beschrieben.

Die Grundidee, die hinter Produktlinien steckt, ist die Entwicklung von maßgeschneiderter Software für verschiedene Anwendungsfälle, in Form von Varianten, die genau die benötigte Funktionalität enthält [Ape08]. Neue Varianten sollen schnell entwickelt und leicht hinzugefügt werden können, bewährte Funktionalität soll wiederverwendet werden können und die Software soll auf spezielle Kundenwünsche abgestimmt sein und sich an die verfügbaren Ressourcen anpassen [Ape08].

Ein gutes Beispiel für Produktlinien lässt sich in der Automobilindustrie finden. Ein Kunde möchte sich ein neues Auto kaufen. Im Katalog kann er sich ein Fahrzeug aussuchen, das eine gewisse Grundausstattung besitzt. Nun kann der Kunde dieses Fahrzeug an seine individuellen Wünschen anpassen (siehe Abb. 2.5). Aus einer Liste von verfügbaren Optionen, wie z.B. Motorleistung, Farbe und Art der Lackierung, Innenausstattung, etc., kann der Kunde entscheiden, was am besten für ihn und seine Bedürfnisse geeignet ist. So kann eine Vielzahl an verschiedenen Varianten von Autos entstehen, da jeder Kunde andere Wünsche hat.

Ein weiters Beispiel für Produktlinien sind Computer. Hier hat ein Kunde auch die Möglichkeit, sich einen *Personal Computer (PC)* genau nach seinen Wünschen zusammen zu stellen. Er kann sich für eine bestimmte Grafikkarte, Größe der Festplatte oder Art des Betriebssystems entscheiden. Die einzelnen Komponenten eines Computers können auf diese Art und Weise individuell kombiniert werden,

[15] http://www.sei.cmu.edu/productlines/index.html

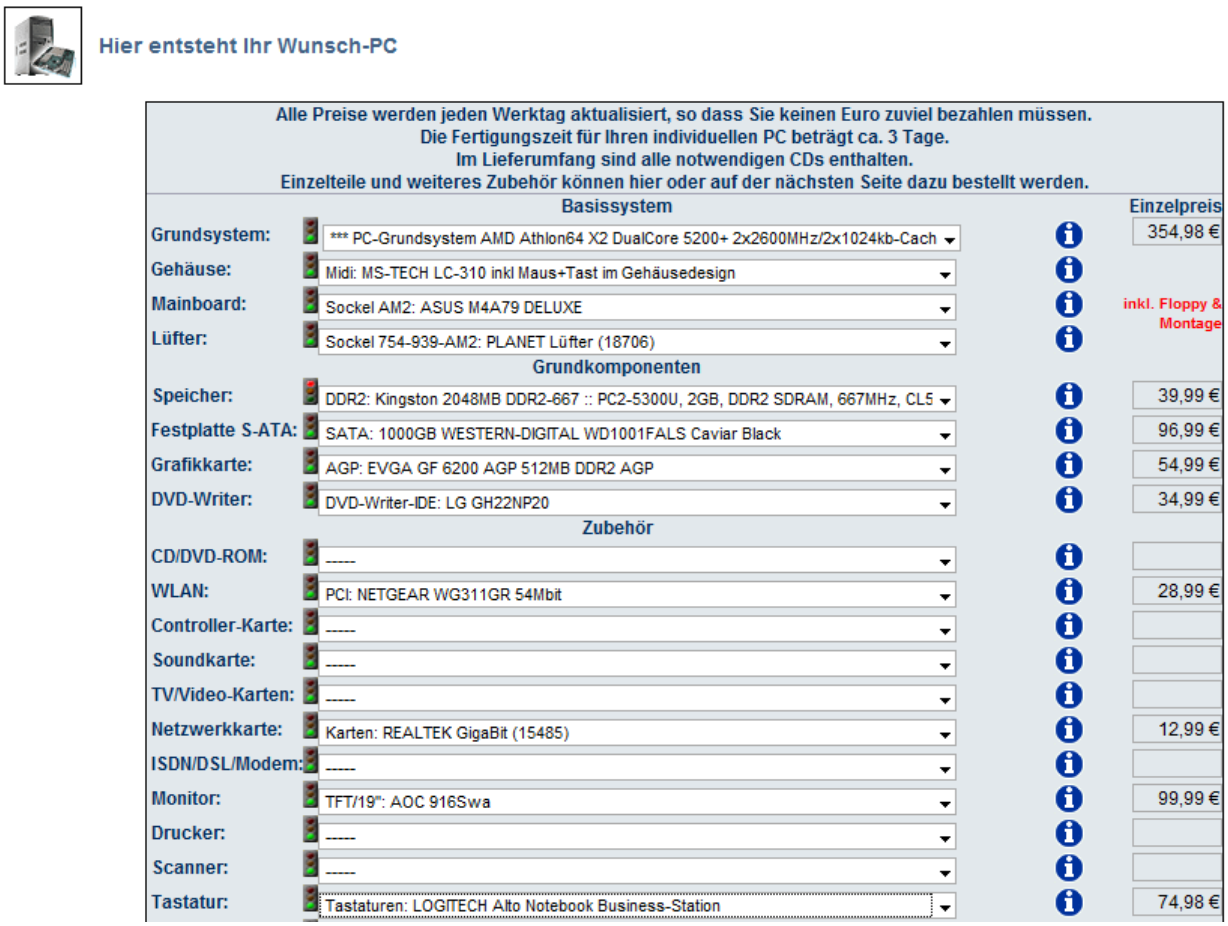

Abbildung 2.6: Produktlinie – PC
[Quelle: http://www.planet-elektronik.de/angebot.htm]

was auch hier zu einer sehr großen Variantenvielfalt führt (siehe Abb. 2.6).

Ein großer Vorteil von Produktlinien ist die hohe Wiederverwendbarkeit der einzelnen Funktionalitäten innerhalb der Produktfamilie. So ist es möglich, eine Implementation einer Funktionalität in der Regel für alle Produkte wieder zu verwenden. Durch die Wiederverwendung solcher gemeinsamer Teile, wird eine Senkung der Entwicklungskosten angestrebt, was eines der Hauptziele einer Software-Produktlinienentwicklung ist.

2.2.2 Was ist ein Feature?

Features werden dazu genutzt, zwischen verschiedenen Varianten eines Programms oder einer Software zu unterscheiden. Eine allgemeine Definition findet sich in [ALMK08]:

> „A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder‘s requirement, to implement a design decision, and to offer a configuration option.“

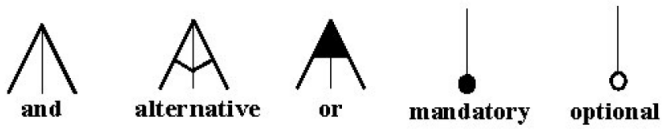

Abbildung 2.7: Kantenkennzeichnungen
[Quelle: [Bat05]]

Ein Feature [Ape08]:

- ist eine semantisch zusammenhängende Einheit, die einer Anforderung an ein Programm (Produkt) entspricht.
- ist eine Abstraktion einer Funktionalität.
- dient zur Spezifikation von Programmen (Produkten).
- repräsentiert Gemeinsamkeiten oder Unterschiede eines Programms (Produkts).
- bietet eine Konfigurationsmöglichkeit (vgl. Beispiel zu Produktlinien in Kapitel 2.2.1).

Meistens sind die Programme einer Produktlinie auf einen bestimmten Markt, auch *Domäne* genannt, zugeschnitten, damit sie nicht zu unterschiedlich werden [Ape08]. Die *Feature-Modellierung* dient zur Darstellung der Features einer solchen Domäne [Ape08]. Durch ein *Feature-Modell* werden die elementaren Abstraktionen einer Domäne und deren Beziehungen, sowie die Menge der Programme einer Produktlinie beschrieben [Ape08]. Feature-Modelle sind hierarchisch angeordnete Mengen von Features. Die Beziehungen zwischen Eltern-Features und Kind-Features lassen sich wie folgt definieren [Bat05]:

- *and*: alle Kind-Features müssen ausgewählt werden
- *alternative*: genau eins der Kind-Features kann ausgewählt werden
- *or*: mindestens eins der Kind-Features muss ausgewählt werden
- *mandatory*: Kind-Feature muss ausgewählt werden
- *optional*: Kind-Feature kann ausgewählt werden

Mit Hilfe eines *Feature-Diagramms* werden Features und deren Beziehungen graphisch dargestellt [Bat05]. Hierbei handelt es sich um Bäume. Die inneren Knoten entsprechen dabei den zusammengesetzten Features (Eltern-Features) und die Blätter entsprechen primitiven, d.h. atomaren Features [Bat05]. Die Kanten des Baumes tragen die Information über die Art der o.g. Eltern-Kind-Beziehungen, wie in Abb. 2.7 zu sehen. Abb. 2.8 zeigt ein Beispiel eines Feature-Modells.

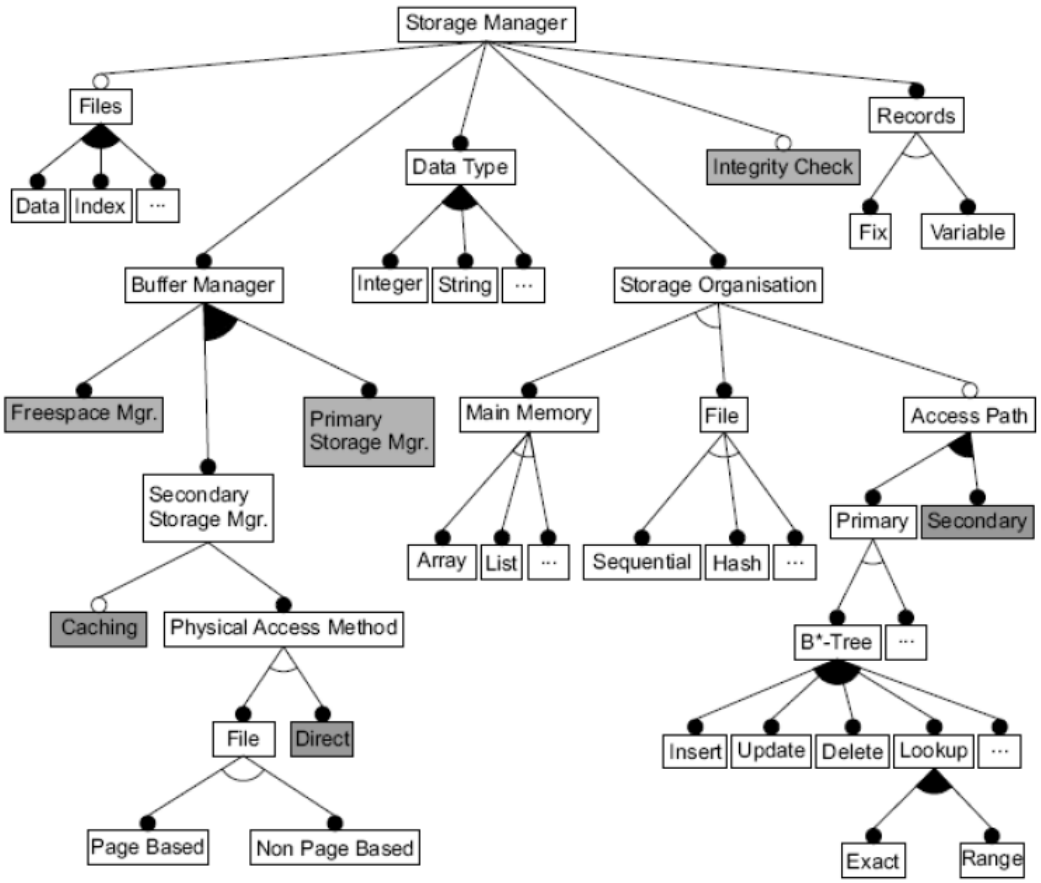

Abbildung 2.8: Feature-Modell eines Speichermanagers
[Quelle: [Ape08]]

```
C = { Base, BigI, BigD, Iadd, Idiv,
      Isub, Dadd, Ddivd, Ddivu, Dsub }
```

Abbildung 2.9: Feature-Orientiertes Modell einer Produktfamilie
[Quelle: [Bat06]]

2.2.3 FOP - Feature-Oriented Programming

Features sind, wie oben beschrieben, semantisch zusammenhängende Einheiten, deren Implementierung jedoch im Quelltext verstreut vorliegt. Grund hierfür ist eine mangelnde *Feature-Kohäsion*, die in [Ape08] definiert ist als

> „Eigenschaft eines Programms alle Implementierungseinheiten eines Features an einer Stelle im Code zu lokalisieren“

und die das *Feature Traceability Problem* zur Folge hat. In [Ape08] wird es definiert als

> „Problem die Features einer Domäne bzw. eines konkreten Programms im Code wiederzufinden.“

Die *Feature-orientierte Programmierung* ist in der Lage, dieses Problem zu lösen, indem jedes Feature durch ein *Feature-Modul* implementiert wird [Ape08]. Dadurch wird eine Trennung und Modularisierung von Features erreicht und die Komposition von Features vereinfacht. Die Modularisierung und Komposition von Features setzt allerdings eine Programmiersprache voraus, die eine gewisse Ausdruckskraft besitzt [AKL08]. Ein Beispiel hierfür ist Java und eine Erweiterung davon mit dem Namen *Jak* (kurz für „Jakarta“), die im Folgenden anhand eines Beispiels[16] kurz vorgestellt wird.

Es wird eine Produktfamilie von Taschenrechnern betrachtet. Die Taschenrechner unterscheiden sich in:

1. ihren arithmetischen Konstanten *BigInteger*[17] oder *BigDecimal*[18]
2. durch die Menge der Operationen, die sie durchführen können.

Zu den unterstützten Operationen gehören Addition, Division und Subtraktion. Das Modell *c* in Abb. 2.9 beschreibt diese Produktfamilie. Die einzige Konstante in diesem Modell ist *Base*. Sie definiert einen leeren Taschenrechner (siehe List. 2.8[19]).

Die in List. 2.9[20] und List. 2.10[21] dargestellten Klassen sind Erweiterungen des leeren Taschenrechners. Sie fügen Konstanten und Methoden hinzu. Sie schließen sich aber gegenseitig aus, d.h. ein Taschenrechner arbeitet entweder auf

[16] Übernommen aus [Bat06]
[17] **BigInteger** sind Ganzzahlen von beliebiger Größe
[18] **BigDecimal** sind Dezimalzahlen mit beliebiger Genauigkeit
[19] [Quelle: [Bat06]]
[20] [Quelle: [Bat06]]
[21] [Quelle: [Bat06]]

```
class calc { }
```

Listing 2.8: Basis Taschenrechner

```
import java.math.BigInteger;

refines class calc {
  static BigInteger zero = BigInteger.ZERO;
  BigInteger e0 = zero, e1 = zero, e2 = zero;

  void enter( String val ) {
    e2 = e1;
    e1 = e0;
    e0 = new BigInteger(val);
  }

  void clear() {
    e0 = e1 = e2 = zero;
  }

  String top() { return e0.toString(); }
}
```

Listing 2.9: Erweiterung BigInteger

BigInteger- oder auf BigDecimal-Werten. Erweiterungen werden mit dem Schlüsselwort *refines* gekennzeichnet.

Die in List. 2.11[22] gezeigten Erweiterungen entsprechen den Operationen bzgl. BigInteger und BigDecimal. Zu Beachten ist, dass es zwei Arten der Division von BigDecimal-Werten gibt, abgerundet und aufgerundet.

Die Modularität erlaubt nun eine Programmerstellung mit Hilfe einer *Feature-Selektion* [Ape08]. Dabei wird aus allen vorhandenen Features eine Menge von Features ausgewählt, die dann das Programm bildet. Ein Taschenrechner könnte dann wie folgt aussehen [Bat06]:

$$calcI = Idiv \bullet Iadd \bullet BigI \bullet Base$$

Der Taschenrechner arbeitet mit BigInteger-Werten und unterstützt die Operationen Addition und Division. Abb. 2.10 zeigt den dazugehörigen Quelltext.

Dadurch ergibt sich jedoch häufig ein Problem, denn die Anzahl an Features ist meist sehr viel größer als in diesem kleinen Beispiel (mehr als 100). Ein möglicher Lösungsansatz hierfür ist eine *partielle Selektion* [Ape08]. Die Grundidee ist, ein Anwender wählt nur die Features aus, die für ihn am wichtigsten sind und daraufhin werden *passende* Features automatisch ausgewählt. Welches passende Features sind, wird durch Einschränkungen und Regeln der Domäne oder durch Expertenwissen festgelegt [Ape08]. Durch diese automatische Selektion wird eine Optimierung erreicht.

[22] [Quelle: [Bat06]]

```
import java.math.BigDecimal;

refines class calc {
  static BigDecimal zero = new BigDecimal("0");
  BigDecimal e0 = zero, e1 = zero, e2 = zero;

  void enter( String val ) {
    e2 = e1;
    e1 = e0;
    e0 = new BigDecimal(val);
  }

  void clear() {
    e0 = e1 = e2 = zero;
  }

  String top() { return e0.toString(); }
}
```

Listing 2.10: Erweiterung BigDecimal

```
* Division für BigDecimal (abgerundet)
import java.math.BigDecimal;

refines class calc {
  void divide() {
    e0 = e0.divide( e1,
    BigDecimal.ROUND_DOWN );
    e1 = e2;
  }
}

* Division für BigInteger
refines class calc {
  void divide() {
    e0 = e0.divide( e1 );
    e1 = e2;
  }
}

* Addition für BigInteger und BigDecimal
refines class calc {
  void add() {
    e0 = e0.add(e1);
    e1 = e2;
  }
}
```

Listing 2.11: Operationen für BigInteger und BigDecimal

```
import java.math.BigInteger;

class calc {
   static BigInteger zero = BigInteger.ZERO;
   BigInteger e0 = zero, e1 = zero, e2 = zero;

   void add() {
      e0 = e0.add(e1);
      e1 = e2;
   }

   void clear() {
      e0 = e1 = e2 = zero;
   }

   void divide() {
      e0 = e0.divide( e1 );
      e1 = e2;
   }

   void enter( String val ) {
      e2 = e1;
      e1 = e0;
      e0 = new BigInteger(val);
   }

   String top() { return e0.toString(); }
}
```

Abbildung 2.10: Taschenrechner *calcI*
[Quelle: [Bat06]]

Kapitel 3

Analyse

In diesem Kapitel werden die beiden Konzepte Feature und Service näher betrachtet. Dabei sollen Unterschiede bzw. Ähnlichkeiten und ihr Zusammenwirken analysiert werden. Dies geschieht mit Hilfe eines kleinen Beispiel-Szenarios.

3.1 Beispiel-Szenario

Ausgangspunkt für die Analyse ist ein Warenhaus-Szenario[1] (siehe Abb. 3.1). Es besteht aus folgenden sieben Services, die zuständig für die Bearbeitung einer Kundenbestellung sind:

- *Aquisition:* Annehmen der Kundenbestellung
- *Availability Checking:* Prüfen, ob angeforderte Artikel verfügbar sind
- *Credit Ranking:* Überprüfung der Bonität (Kreditwürdigkeit) des Kunden
- *Ordering:* Kommissionierung
- *Shipping:* Versenden der Artikel an den Kunden
- *Billing:* Abrechnung erstellen
- *Payment Checking:* Überprüfung der Bezahlung des Kunden

3.2 Problemstellung

Wie bereits in Kapitel 1.1 beschrieben, weisen SOAs noch einige Probleme auf, die bereits anhand dieses kleinen Beispiels sehr gut verdeutlicht werden können. Die Services in Abb. 3.1 sind jeweils in ein Modul gekapselt und kommunizieren untereinander über Interfaces. Betrachtet man das Beispiel aber etwas genauer, stößt man auf einige Probleme [AKL08].

Die oben beschriebenen Services können in verschiedenen Varianten eines Warenhauses benutzt werden:

[1] Übernommen aus [AKL08]

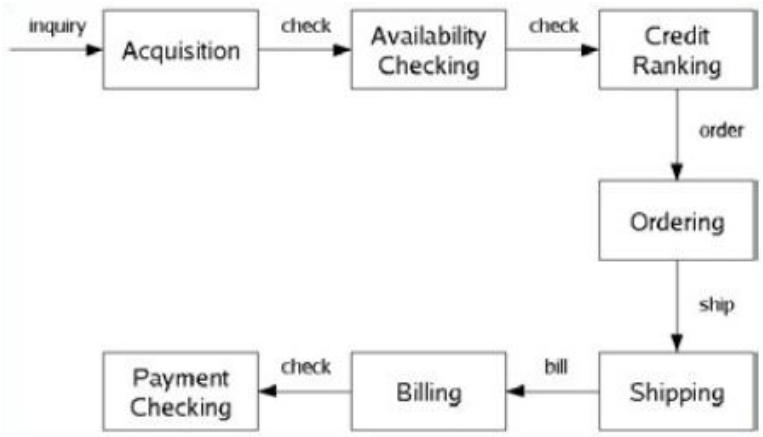

Abbildung 3.1: Service-Übersicht des Warenhauses
[Quelle: [AKL08]]

Discounting: Kunden können einen Rabatt auf gewisse Produkte ab einer bestimmten Bestellmenge erhalten. Um diese Funktionalität anbieten zu können, müssen die Implementierungen einiger Services angepasst werden, hier *Acquisition*, *Credit Ranking*, *Billing* und *Payment Checking* [AKL08].

Overseas Orders: In dieser Variante ist es möglich Güter in Überseeländer zu verschiffen. Dabei müssen aber Dinge, wie ausländische Währungen oder Zölle berücksichtigt werden. Dies betrifft die Implementierungen der Services *Credit Ranking*, *Shipping*, *Billing* und *Payment Checking* [AKL08].

Es fällt sofort auf, dass sich die Unterschiede zwischen den beiden Varianten gleich auf mehrere Services auswirken. Die Varianten unterscheiden sich nur geringfügig von der Service-Architektur am Anfang. Die Kernfunktionalität der Services und der Gesamtarchitektur bleibt allerdings unverändert. Es wird lediglich zusätzliche Funktionalität, repräsentiert durch Features, in den einzelnen Varianten hinzugefügt [AKL08].

Beispielsweise wird in der Variante *DISCOUNTING* das Interface (und die Implementierung) des Services *Acquisition* so erweitert, dass der Kunde in der Lage ist, sich über die Höhe des Rabatts zu informieren [AKL08]. Ein Feature kann aber auch nur die Implementierung eines Services, wie z.B. bei dem Service *Billing*, betreffen. Hier wird nur die Berechnung des Preises verändert und nicht das Interface selbst [AKL08].

Ziel ist es nun von den Gemeinsamkeiten aller Varianten zu profitieren, indem man z.B. gemeinsamen Quelltext wiederverwendet, um neue Varianten zu erstellen [AKL08]. Die Varianten unterscheiden sich dann nur durch unterschiedliche Feature-Module. Aber Services, die in verschiedenen Varianten verwendet werden, müssen sich leicht anpassen lassen, ansonsten würde der Aufwand für die Entwicklung von neuen Varianten stark erhöht werden. Hinzu kommen unter Anderem noch folgende Probleme [AKL08]:

- Wenn man einen neuen Service integrieren will, woher weiß man, zu welcher Variante des Szenarios dieser Service gehört?

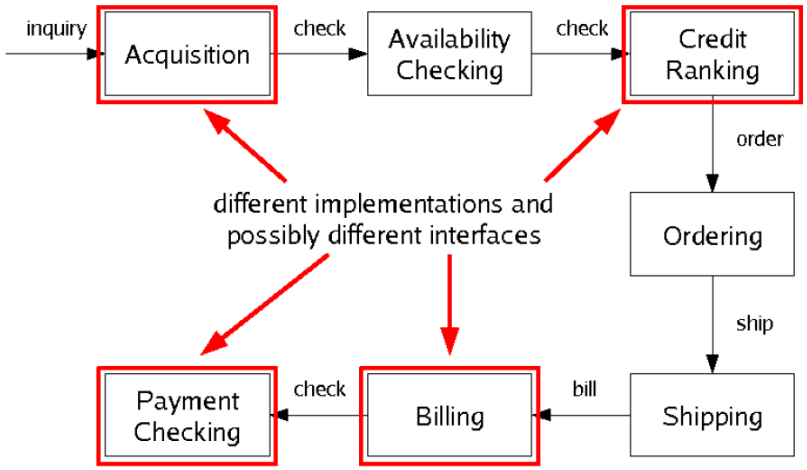

Abbildung 3.2: Auswirkungen von Feature DISCOUNTING auf Services des Warenhauses
[Quelle: [AKL08]]

- Welche Varianten sind mit einem anderen Service überhaupt kompatibel?

FOP hat das Potenzial diese Probleme zu lösen.

3.3 Features und Services

Die unterschiedlichen Varianten eines Szenarios lassen sich durch unterschiedliche Features beschreiben und trennen. Features erweitern die Implementierung von Services und sind somit ein Inkrement an Service-Funktionalität [AKL08]. Die Varianten in diesem Beispiel-Szenario unterscheiden sich jeweils in einem Feature, wie z.B. die Variante *Discounting*, die ähnlich zum Basiswarenhaus ist, nur dass sie noch zusätzlich das Feature *DISCOUNTING* anbietet [AKL08].

Abb. 3.2 zeigt die Service-Architektur des Warenhauses zusammen mit dem Feature *DISCOUNTING*. Hier kann man gut erkennen, dass das Feature mehrere Services gleichzeitig betrifft (rot markiert). Dabei kann es das Interface und die Implementierung (z.B. *Acquisition*) oder nur die Implementierung (z.B. *Billing*) eines Services verändern. Daraus ergibt sich folgende Beziehung zwischen Features und Services [AKL08]:

- Services bietet Grundfunktionalität und eine Menge von Features, die diese erweitern, an.
- Features können sich auf mehrere Services und deren Interface bzw. Implementierung auswirken.

Mit Hilfe der FOP kann man nun den Quelltext eines Features in ein separates Feature-Modul packen [AKL08]. Abb. 3.3 zeigt die beiden Feature-Module für das Basiswarenhaus und das Feature *DISCOUNTING*. Die roten Pfeile stehen

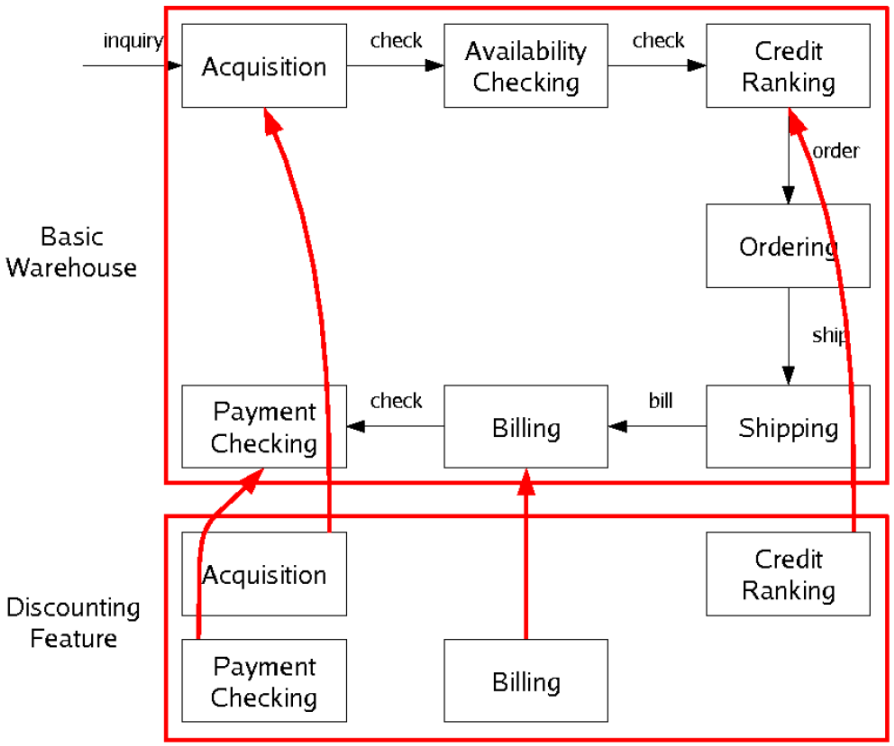

Abbildung 3.3: Feature DISCOUNTING getrennt von Warenhausarchitektur [Quelle: [AKL08]]

```
1  class Bill { ...
2    double getPrice(Order o) { ... }
3  }
```

```
4  refines class Bill { ...
5    double getPrice(Order o) {
6      if (qualifiesForDiscount(o))
7        return original(o) * 0.8;
8      else
9        return original(o);
10     }
11   }
12   boolean qualifiesForDiscount(Order o) { ... }
13 }
```

Abbildung 3.4: Java-Klasse und eine Erweiterung davon
[Quelle: [AKL08]]

für Erweiterungen von Services durch das Feature. Die Komposition von Features erfolgt durch das Zusammenfügen des Quelltexts eines Services mit dem Quelltext seiner Features. Die Modularisierung und Komposition von Features setzt eine Programmiersprache mit einer gewissen Ausdruckskraft voraus [AKL08]. Abb. 3.4 zeigt einen Service, der in Java/Jak implementiert ist und eine Erweiterung des Services durch ein Feature (vgl. Beispiel in Kapitel 2.2.3).

Die Klasse *Bill* gehört zu dem Service *Billing* aus dem Basiswarenhaus und wird durch das Feature *DISCOUNTING* erweitert. Gekennzeichnet wird dies mit dem Schlüsselwort „refines“. Dabei wird die Methode *getPrice* überschrieben und es wird eine neue Methode *qualifiesForDiscount* hinzugefügt. Damit kann man einem Kunden einen Rabatt von 20% geben [AKL08].

Es gibt Erweiterungen für XML (ähnlich wie Jak für Java), die es ermöglichen, das Interface eines Services, das in WSDL geschrieben ist, zu erweitern [AKL08]. Komponiert man nun das Basiswarenhaus mit dem Feature *DISCOUNTING* wird z.B. das Interface des Services *Acquisition* mit einer Erweiterung, die denselben Namen hat, überlagert. Dabei wird eine neue Operation *discountResponse* hinzugefügt, mit der der Kunde die Höhe des Rabatts erfahren kann (siehe Abb. 3.5) [AKL08].

Ein Ergebnis bisher ist, dass Features die Services bzw. deren Implementierungen und Interfaces aus dem Warenhaus-Szenario erweitern. Dies lässt sich in Form eines Feature-orientierten Modells darstellen. Das Warenhaus (WH) besteht aus der grundlegenden Service-Architektur (BASE) und den beiden Features *DISCOUNTING* (DISC) und *OVERSEAS ORDERS* (OVER) (Gleichung 3.1 und 3.2) [AKL08]:

$$WH = \{BASE, DISC, OVER\} \tag{3.1}$$

$$BASE = \{Acqu, Avail, Cred, Ord, Ship, Bill, Pay\} \tag{3.2}$$

```
1  <definitions name="Acquisition">
2  ...
3    <message name="priceRequest">
4      <part name="orderNumber" type="xsd:int">
5    </message>
6  ...
7  </definitions>
```

```
8  <definitions name="Acquisition">
9  ...
10   <message name="discountResponse">
11     <part name="discountPercentage" type="xsd:float">
12   </message>
13 ...
14 </definitions>
```

Abbildung 3.5: WSDL Definition eines Interfaces und eine Erweiterung davon [Quelle: [AKL08]]

Da das Feature *DISCOUNTING* die Services *Acquisition*, *Credit Ranking*, *Billing* und *Payment Checking* erweitert sieht DISC folgendermaßen aus [AKL08]:

$$DISC = \{Acqu, Cred, Bill, Pay\} \tag{3.3}$$

Anhand von Gleichung 3.3 kann man auch sehr gut sehen, dass sich Features auf mehrere Services gleichzeitig auswirken.

Mit Hilfe dieses Feature-Modells lassen sich nun ganz einfach und auch automatisch Varianten des Warenhauses generieren[2], indem *BASE* mit den verfügbaren Features komponiert wird (dargestellt durch •) [AKL08]. Gleichung 3.4 zeigt eine Variante, die das Feature *DISCOUNTING* enthält und Gleichung 3.5 zeigt eine Variante mit beiden Features:

$$WH_1 = DISC \bullet BASE \tag{3.4}$$

$$WH_2 = OVER \bullet DISC \bullet BASE \tag{3.5}$$

3.4 Voraussichtlicher Nutzen

Durch das gemeinsame Nutzen der beiden Konzepte Feature und Service ergeben sich unter Anderem folgende Vorteile [AKL08]:

Modularität: Zerlegt man eine Service-Architektur und ihre Services in Features kann man die verschiedenen Anforderungen an das System besser separieren. Wie im vorherigen Abschnitt gezeigt, betreffen Features meist mehrere

[2] z.B. durch AHEAD Tool Suite

Services. Der Quelltext eines Features kann jedoch in ein Feature-Modul gekapselt werden. Dadurch wird die Basisimplementierung eines Services nicht mit dem Quelltext eines Features verunreinigt, was das Verständins und die Wartbarkeit des Quelltexts erheblich verbessert. Es gibt bereits einige Fallstudien für SPLs in denen gezeigt wurde, das FOP die Modularität steigert. Man geht davon aus, dass FOP auch die Modularität von Service-Architekturen und ähnlichen Anzätzen verbessern kann.

Variabilität: Ein Anwender oder Programmierer kann verschiedene Varianten eines Services generieren, da der Quelltext des Services vom Quelltext eines Features getrennt vorliegt. Ein Benutzer kann dann seine gewünschten Features, mit Hilfe der Feature-Komposition, zu einem Programm kombinieren. Dabei werden die entsprechenden Code-Fragmente von Services und Features miteinander verschmolzen (vgl. Gleichung 3.5).

Uniformität: Die einzelnen Services einer Service-Architektur können in verschiedenen Programmiersprachen implementiert sein. Dies beinhaltet die Implementierung der Services (z.B. in Java oder C++) und die Spezifikation ihrer Interfaces (z.B. in WSDL). Die Komposition von Features ist sprachenunabhängig und kann auf jede Komponente eines Services angewendet werden [AL08].

3.5 Herausforderungen

Neben den Vorteilen, die sich aus der gemeinsamen Nutzung von Services und Features ergeben, treten auch einige Herausforderungen auf die im Folgenden näher beschrieben werden sollen [AKL08]:

Modularität: Wie bereits beobachtet wurde, erschwert das Querschneiden in den Services und deren Komponenten die Modularität. Um dieses Problem lösen zu können, gibt es bereits mehrere Ansätze. Besonders hervorzuheben darunter ist die *Aspect-Oriented Programming (AOP)*. Die FOP ist eng mit all diesen Ansätzen verwandt und eben dieser Feature-orientierte Ansatz ist sehr Erfolg versprechend auf Grund der Theorie von Features, die eine automatische Komposition von Features und Services erlaubt. Eine Herausforderung ist es, die Praktikabilität und Skalierbarkeit dieses Ansatzes, mit Hilfe von nicht trivialen Fallstudien zu SOAs, zu beweisen.

Variabilität: Da die Anzahl an Varianten stetig wächst, wird ein automatisiertes Management der Varianten immer wichtiger. Betrachtet man eine SOA, dann ist dies eine besondere Herausforderung, bedingt durch eine meist große Anzahl an Services, die in vielen verschiedenen Varianten auftreten können. Von den Erfahrungen und Tools, die man bereits aus dem Bereich der SPLs kennt, kann ein Feature-basierter Ansatz nur profitieren. Aber wie bereits schon bei der Modularität erwähnt, müssen auch hier erst noch aussagekräftige Fallstudien durchgeführt werden.

Uniformität: Wie bereits vorher erwähnt wurde, können Services in verschiedenen Sprachen implementiert sein. Die Grundidee, die hinter dem Konzept der Web Services steckt, basiert auf der Virtualisierung der Services (vgl. Kapitel 2.1.5). Services können von unterschiedlichen Anbietern implementiert und angeboten werden und in einer Infrastruktur, die eine sprachenunabhängige Kommunikation erlaubt, integriert und genutzt werden. Um die Generizität, d.h. die Eigenschaft der Feature-Komposition sprachenunabhängig alle Teilkomponenten eines Services zu kombinieren, zu beurteilen, ist die SOA ein sehr gutes Anwendungsbeispiel. Ob die Generizität der Feature-Komposition auch für sehr große Service-Architekturen, die aus sehr unterschiedlichen Services aufgebaut sein können, gilt, muss allerdings noch untersucht werden.

Zusammenfassend kann festgehalten werden, dass sich Services und Features gegenseitig ergänzen können und eine gemeinsame Nutzung einige Vorteile mit sich bringen kann. Aber es gibt auch noch ein paar Herausforderungen, die man dabei bewältigen muss. Im nachfolgenden Kapitel soll das Zusammenwirken beider Konzepte anhand eines praktischen Anwendungsfalls gezeigt werden.

Kapitel 4

Fallstudie Flugbuchung

Im Rahmen dieser Bachelorarbeit wird ein Flugbuchung-Szenario implementiert, um das Zusammenspiel von Services und Features zu untersuchen. In diesem Kapitel werden die einzelnen Web Services, ihre Funktionalität, sowie die gefundenen Features und die Änderungen, die sie an den Web Services vornehmen genauer vorgestellt. Zunächst wird aber noch kurz auf die Werkzeugumgebung eingegangen, die zur Erstellung der Fallstudie genutzt wird.

4.1 Werkzeugumgebung

Um Web Services zu implementieren, muss zunächst eine geeignete Entwicklungsumgebung gefunden werden, die folgenden Anforderungen genügt:

- Sie soll kostenlos verfügbar sein und
- eine möglichst gute Unterstützung bei der Entwicklung von Web Services bieten.

Eine integrierte Entwicklungsumgebung ist ein Anwendungsprogramm zur Entwicklung von Software. Integrierte Entwicklungsumgebungen verfügen in der Regel über folgende Komponenten:

- Texteditor
- Compiler bzw. Interpreter
- Linker
- Debugger
- Quelltextformatierungsfunktion

In erster Linie sind IDEs hilfreiche Werkzeuge, die dem Software-Entwickler häufig wiederkehrende Aufgaben abnehmen und einen schnellen Zugriff auf wichtige Funktionen bieten. Es gibt sie für nahezu alle Programmiersprachen und Plattformen. Das NetBeans 6.5 *Integrated Development Environment (IDE)* stellt sich für die angegebenen Anforderungen als gute Lösung heraus und arbeite mit der Programmiersprache Java:

- NetBeans steht im Bündel mit einem *GlassFish V2.1* Applikationsserver und diversen Tools zur Verfügung ⇒ alle benötigten Komponenten sind nahtlos integriert, was viel Konfigurationsaufwand erspart.
- Auf der Hompage von NetBeans findet man eine gute Dokumentation der IDE.

Aufsetzen der Entwicklungsumgebung: Wer schnell zu einer vorkonfigurierten Entwicklungsumgebung kommen möchte, kann sich auf der Homepage von NetBeans[1] das oben genannte „NetBeans IDE Bundle“ herunterladen.

Einzige Vorraussetzung ist ein vorinstalliertes *Java SE Development Kit*, welches auf der Homepage von Java[2] zum Download bereit steht.

Projekte Grundlage für jede Java-Entwicklung innerhalb dieser Entwicklungsumgebung ist ein Projekt. Dies ist eine Gruppe von Quelldateien und Einstellungen, wie man diese Dateien anlegen, ausführen und auf Fehler überprüfen kann. Die IDE speichert alle Informationen eines Projekts in einem sog. *Ant*[3] Skript, in einer Java-Properties-Datei und in ein paar XML-Konfigurationsdateien ab. Für das Flugbuchung-Szenario werden zwei solcher Projekte erstellt, die alle Web Services bzw. Web Service Clients mit den dazugehörigen Quelldateien, sowie die oben beschriebenen Informationen, enthalten.

Web Services Ein Web Service ist nichts anderes als eine Java-Klasse. Diese Klasse enthält die Methoden, die der Web Service nach außen hin anbietet und die von einem Konsumenten aufgerufen werden können (vgl. 2.1.1), nachdem der Web Service von einem Applikationsserver zur Verfügung gestellt wurde.

Web Services lassen sich in der NetBeans IDE, mit Hilfe von Assistenten zum Anlegen der Java-Klassen und der Methodenköpfe, sehr leicht erstellen. Es müssen dann nur noch die Methodenrümpfe von einem Anwender implementiert werden. Auf der Homepage von NetBeans finden sich einige Anleitungen zum Erstellen von Web Services[4].

Web Service Tester Mit NetBeans hat man die Möglichkeit die erstellten Web Services zu testen. Dies geschieht mit dem sog. *Web Service Tester*. Ruft man diese Anwendung mit einem Web Service auf, öffnet sich ein Fenster im Webbrowser, wie in Abb. 4.1 gezeigt. In diesem Formular können den Methoden des Web Services Werte übergeben werden.

Der in den Abbildungen gezeigte Web Service hat die Aufgabe, Beträge in eine ausgewählte Währung umzurechnen. In diesem Beispiel, soll der Betrag von 2,50 Euro in US-Dollar umgerechnet werden. Der Wert 2 für den ersten Parameter

[1] http://www.netbeans.org/
[2] http://java.sun.com
[3] Apache Ant (Ant englisch für Ameise) ist ein in Java geschriebenes Werkzeug zum automatisierten Erzeugen von Programmen aus Quelltext.
[4] http://www.netbeans.org/kb/docs/web/tutorial-webapps.html

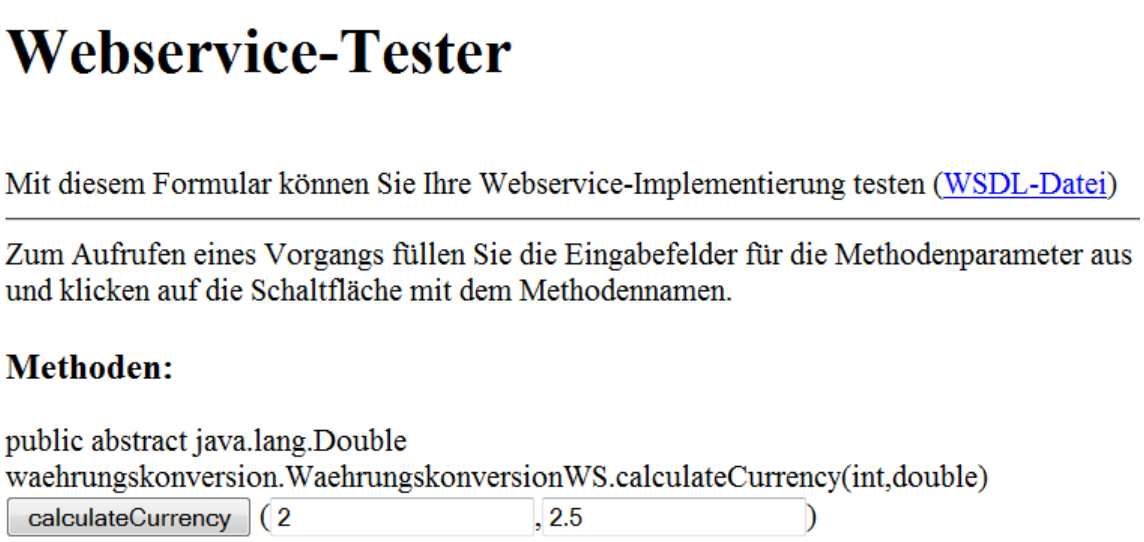

Abbildung 4.1: Seite zum Testen des Web Services

entspricht dabei der Währung US-Dollar und der Wert 2.5 dem Betrag, der umgerechnet werden soll. Danach können die Ergebnisse, die zurückgegeben werden, auf ihre Richtigkeit überprüft werden (siehe Abb. 4.2).

Die entsprechenden SOAP-Nachrichten für die Anforderung und Antwort sind in Abb. 4.3 zu sehen.

Web Service Client Möchte man nun als Entwickler eine Anwendung programmieren, die den oben erwähnten Web Service zum Umrechnen von Geldbeträgen nutzt, so benötigt man einen *Web Service Client*. Dieser besitzt eine Referenz auf das WSDL-Dokument des Web Services und somit weiß man genau, wie man die Methode des Web Services aufrufen kann. Der Aufruf lässt sich dann ganz leicht in den eigenen Quelltext einbetten, dargestellt in List. 4.1.

Zuerst wird in Zeile 4 aus dem WSDL-Dokument eine Instanz des Web Services erzeugt und danach in Zeile 5 ein Port, mit dem man den Web Service erreichen kann. In den Zeilen 6 und 7 werden die beiden Parameter *toCurrency* und *amount* festgelegt. Danach wird in Zeile 8 die Methode des Web Services aufgerufen und das Ergebnis in der Variablen *result* gespeichert. Nun kann die Anwendung mit diesem Ergebnis weitere Berechnungen durchführen.

4.2 Szenario – Flugbuchung

Ausgangspunkt für die Entwicklung dieser Fallstudie war eine Recherche im Internet über SOAs mit dem Ziel ein solches System Feature-orientiert zu analysieren. Aber eine bestehende SOA auf querschneidende Features zu untersuchen ist nicht möglich, da kein Zugang zur Implementierung der Services besteht. Die Hersteller von SOAs möchten Implementierungsdetails der Web Services aus wirtschaftlichen Gründen nicht preisgeben. Daher wird ein gebräuchliches Beispielsystem selbst erstellt, um anschließend Feature-orientiert analysiert zu werden.

calculateCurrency Methodenaufruf

Method parameter(s)

Type	Value
int	2
double	2.5

Zurückgegebene Methode

java.lang.Double : "**3.2195**"

Abbildung 4.2: Parameterwerte und Ergebnis der Umrechnung

SOAP-Anforderung

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:calculateCurrency xmlns:ns2="http://waehrungskonversion/">
            <toCurrency>2</toCurrency>
            <amount>2.5</amount>
        </ns2:calculateCurrency>
    </S:Body>
</S:Envelope>
```

SOAP-Antwort

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:calculateCurrencyResponse xmlns:ns2="http://waehrungskonversion/">
            <return>3.2195</return>
        </ns2:calculateCurrencyResponse>
    </S:Body>
</S:Envelope>
```

Abbildung 4.3: SOAP-Anforderung und SOAP-Antwort

```
1  ...
2  <%-- Aufruf des Webservice für Währungsumrechnung --%>
3  try {
4    WebService service = new WebService();
5    WebServicePort port = service.getWebServicePort();
6    int toCurrency = 2;
7    double amount = 35.5;
8    double result = port.calculateCurrency(toCurrency, amount);
9  ...
10 } catch (Exception ex) {
11 ...
12 }
13 <%-- Ende des Webservice Aufrufs! --%>
14 ...
```

Listing 4.1: Aufruf des Web Services

Zunächst gilt es die Funktionalität des Systems festzulegen:

1. Was soll es können und
2. welche Services soll es enthalten?

Wenn man im Internet Seiten wie

- http://www.lufthansa.com
- http://www.flugbuchung.com
- http://www.fairliners.com
- http://www.flug.de
- http://www.unitedairlines.de
- http://www.tuifly.com
- http://www.easyjet.com
- u.v.m.

näher betrachtet, stellt man einige Gemeinsamkeiten fest. Fast alle bieten Funktionen, wie z.B. das Buchen von Flügen, Hotels oder Mietwagen an. Hinzu kommen häufig noch viele weitere Funktionen. Dieses Flugbuchung-Szenario orientiert sich an diesen Reiseportalen und besteht aus Web Services, die auch in den meisten anderen zu finden sind. Hier eine Übersicht:

- Flugbuchung
- Hotelbuchung
- Mietwagenbuchung

Abbildung 4.4: Ausschnitt der Testumgebung für das Flugbuchung-Szenario (1)

- Events und Tickets
- Info und Service
- Baukasten
- Währungskonversion

Diese Web Services werden von entsprechenden Web Service Clients genutzt, die in eine Testumgebung integriert werden. Abb. 4.4 und Abb. 4.5 zeigen kleine Ausschnitte davon. Der gesamte Quelltext der Web Services, Web Service Clients und der Testumgebung ist auf der beigefügten CD-ROM zu finden.

Nachdem man die Web Services festgelegt hat, muss man sich noch Gedanken über ihre Interfaces, also ihre Methoden machen:

1. Welche Parameter sollen den Methoden übergeben werden und
2. was sollen sie zurückliefern?

Auf diese Fragen wird im Abschnitt 4.3 genauer eingegangen. Dort werden alle Web Services vorgestellt.

Es gibt jedoch auch einige Funktionalitäten, wie z.B. die Stornierung eines Fluges oder einer Hotelbuchung, das Einlösen von Bonusmeilen oder das Abschließen einer Reiseversicherung, die sich nicht in einem einzelnen Web Service kapseln

Abbildung 4.5: Ausschnitt der Testumgebung für das Flugbuchung-Szenario (2)

lassen. Vielmehr sind diese Funktionalitäten, auch als Features bezeichnet, auf mehrere Web Services verstreut. Eine Aufgabe ist es nun, vorhandene Features in diesem System zu finden und dann zu versuchen diese heraus zu kristallisieren. Mit Hilfe dieser Features können dann verschiedene Varianten des Szenarios erstellt werden (vgl. 2.2.3).

Welche Features vorhanden sind, wofür sie benötigt werden und welche Änderungen sie genau an den Web Services vornehmen wird im Abschnitt 4.4 bzw. im Abschnitt 4.5 erläutert.

4.3 Web Services

Abb. 4.6 zeigt die Web Service Architektur des Flugbuchung-Szenarios und die Methoden der einzelnen Web Services. Wobei hier auch schon Methoden aufgelistet sind, die von Features hinzugefügt werden. Das sind die Methoden *storno-Reservation* und *encashMiles*. In den nachfolgenden Teilabschnitten werden die Web Services und ihre Methoden genauer betrachtet.

Flugbuchung Dieser Web Service ermöglicht einem Kunden die Suche nach und das Buchen von Flügen. Wenn ein Kunde nach einem Flug suchen möchte, so kann er dies mit der Methode **searchForFlights** tun. Er muss nur einige Suchkriterien, wie z.B. Abflug- und Zielort oder Hin- und Rückflugdatum angeben und erhält als Ergebnis eine Liste mit Flügen, aus der er sich einen Flug auswählen kann, um dann mit der Buchung dieses Fluges fortzufahren.

Für die Buchung ist dann die Methode **doBooking** zuständig. Anhand der Anzahl der Reisenden und der Flug-ID können Flugtickets gebucht werden. War die Buchung erfolgreich erhält der Kunde eine Bestätigung dafür, ansonsten eine Fehlermeldung.

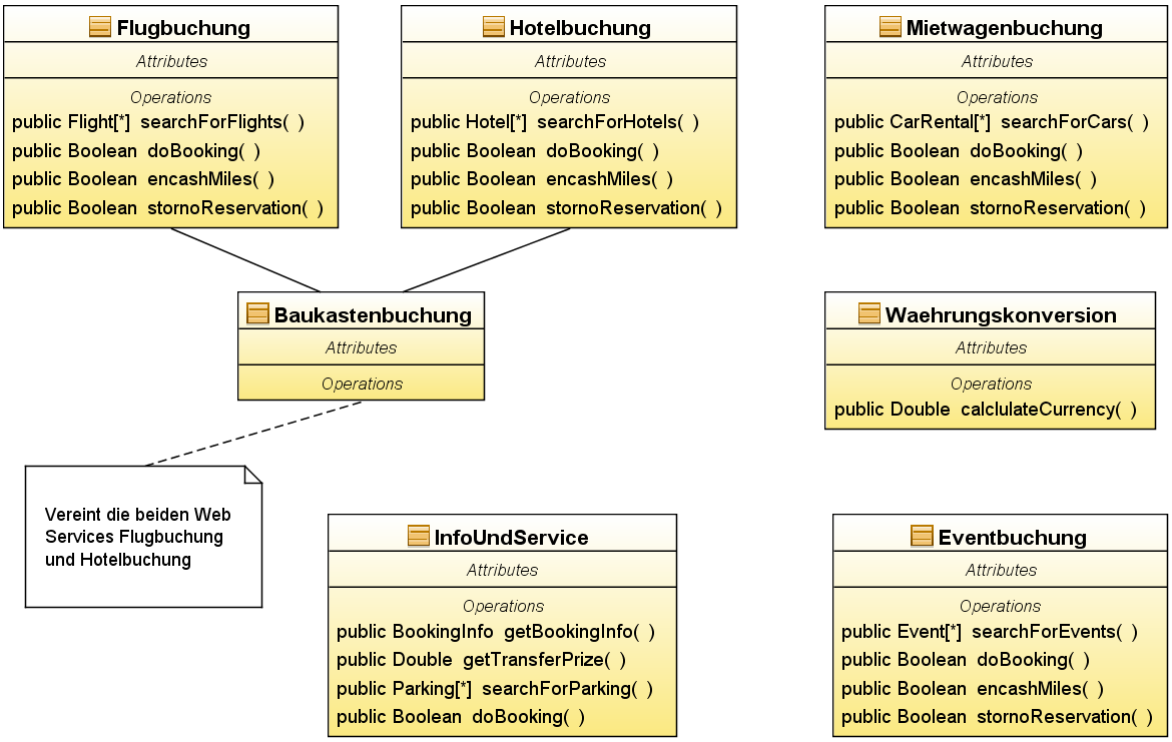

Abbildung 4.6: Web Service Architektur

Hat ein Kunde bereits einen Flug gebucht, möchte diesen aber wieder stornieren, so kann er dies mit Hilfe der Methode **stornoReservation** tun. Auch das Einlösen von Bonusmeilen[5] ist möglich. Dazu wird die Methode **encashMiles** benötigt. Als Parameter müssen ihr die Kundennummer und die Geheimzahl des Kunden übergeben werden.

Hotelbuchung Mit diesem Web Services kann ein Kunde nach Hotels suchen und diese dann buchen oder ein bereits gebuchtes Hotel wieder stornieren. Der Web Service besitzt im Prinzip die gleichen Methoden wie der Web Service *Flugbuchung*, nur heißt hier die Methode **searchForHotels** und ihr werden andere Suchkriterien übergeben. Darunter sind z.B. Hotelkategorie, Land und Region. Das Ergebnis wird dann wieder in Form einer Liste zurückgegeben, aber diesmal mit Hotels.

Die Methoden **doBooking**, **stornoReservation** und **encashMiles** haben dieselbe Aufgabe wie vorhin, aber hier für ein Hotel.

Mietwagenbuchung Ähnlich wie bei den gerade vorgestellten Web Services, kann ein Kunde mit diesem Web Service nach einem passenden Mietwagen suchen und diesen dann buchen. Für die Suche ist die Methode **searchForCars** zuständig. Als Parameter werden dieser Methode unter Anderem ein Land, ein

[5] Bonusmeilen können bei jedem Flug gesammelt und ab einer bestimmten Anzahl eingelöst werden (hier: für verbilligte Flugtickets)

Ort und ein Begin- und Endedatum übergeben.

Aus der zurückgegebenen Liste mit Mietwagen, kann sich der Kunden dann einen passenden aussuchen und diesen anschließend mit der Methode **doBooking** buchen. Es erscheint eine Erfolgsmeldung, falls der Vorgang erfolgreich war, ansonsten wird eine Fehlermeldung ausgegeben. Auch dieser Web Service besitzt die Methoden **stornoReservation** und **encashMiles**, mit denen ein gebuchter Mietwagen storniert werden kann bzw. Bonusmeilen für eine Mietwagenbuchung eingelöst werden können.

Events und Tickets Neben der Suche nach Flügen, Hotels und Mietwagen bietet das Flugbuchung-Szenario auch noch die Möglichkeit nach Events, wie z.B. Fussballspielen, Formel-1-Rennen oder Musikkonzerten, zu suchen und für diese Tickets zu bestellen. Die Suche kann mit der Methode **searchForEvents** durchgeführt werden. Als Suchkriterien werden neben dem Datum, z.B. auch noch die Kategorie übergeben. Der Kunde erhält als Rückgabe wiederum eine Liste mit verfügbaren Events, aus denen er wählen kann.

Auch dieser Web Service besitzt die Methoden **doBooking**, **stornoReservation** und **encashMiles**, die die gleiche Funktionalität besitzen, wie die Methoden der bisher vorgestellten Web Services, nur hier für Events.

Info und Service Mit diesem Web Service kann sich der Kunde über, z.B. aktuelle Wechselkurse, rechtliche Bestimmungen, Konditionen für die Benutzung von Bonusmeilen oder angebotene Reiseversicherungen informieren. Außerdem können Informationen zu bereits gebuchten Flügen oder Hotels abgerufen werden. Diese Aufgabe übernimmt die Methode **getBookingInfo**. Parameter der Methode ist der Reservierungs-Code der Buchung. Als Ergebnis erhält der Kunde alle benötigten Informationen zu seiner Reservierung.

Der Web Service bietet zudem eine Auswahl an mehreren Anreisemöglichkeiten zu einem gewünschten Flughafen:

- Shuttle-Transfer zum Flughafen
- Parken am Flughafen
- Anreise mit der Bahn

Die Methode **getTransferPrize** erlaubt das Suchen nach einem Shuttle-Transfer. Ihr werden Parameter, wie z.B. Flughafen, Ort und Anzahl Personen übergeben. Als Rückgabe erhält der Kunde den Preis für den Transfer zum Flughafen und hat die Möglichkeit diesen zu buchen.

Zuständig für die Suche nach Parkplätzen am oder in der Nähe eines Flughafens, ist die Methode **searchForParking**. Ihr werden die Parameter Flughafen und Anfangs- und Endedatum übergeben. Aus der Liste, die zurückgegeben wird, kann der Kunde dann mit Hilfe der Methode **doBooking** einen Parkplatz buchen. Diese Methode ist ebenfalls für die Buchung eines Shuttle-Transfers und eines Bahntickets zuständig.

Abbildung 4.7: Kombination von Flugbuchung und Hotelbuchung in der Testumgebung

Baukasten Der Baukasten ist in dem Sinn eigentlich kein richtiger Web Service. Hier werden die beiden Web Services *Flugbuchung* und *Hotelbuchung* lediglich miteinander vereint. Durch die Kombination von Flug und Hotel kann eine Vergünstigung im Preis erzielt werden. Abb. 4.7 zeigt diese Kombination in der Testumgebung.

Währungskonversion Der Web Service *Währungkonversion* ist für die Umrechnung aller Preise in eine ausgewählte Währung zuständig und besitzt nur die Methode **calculateCurrency**. Der Methode werden als Parameter eine Zielwährung und ein Betrag, der umgerechnet werden soll, übergeben. Rückgabewert ist der umgerechnete Betrag.

4.4 Features

Betrachtet man die im vorherigen Abschnitt vorgestellten Web Services, fällt auf, dass einige sehr ähnliche Methoden, wie z.B. **stornoReservation** oder **encashMiles** besitzen. Dies ist zurückzuführen auf das Querschneiden von Features. Die Implementierung der Features ist auf mehrere Web Services verstreut. Folgende Features können aus dem Flugbuchung-Szenario herauskristallisiert werden:

Währungsumrechnung Durch dieses Feature können alle Preise, z.B. für Flüge, Hotels oder Mietwagen in verschiedene Währungen umgerechnet werden. Im Moment kann man zwischen den Währungen Euro, US-Dollar und Britischen Pfund wählen, was sich aber um zusätzliche Währungen leicht erweitern lässt.

Storno Wenn man einen Flug oder ein Hotel gebucht hat, aber dann aus irgendwelchen Gründen die Reise nicht antreten kann, möchte man natürlich die Möglichkeit haben, eine Reservierung zu stornieren. Dafür ist das Feature *Storno* zuständig.

Discount Mit Hilfe dieses Features ist es möglich einen Rabatt auf eine Buchung zu bekommen, z.B. in Form eines Frühbucherrabatts für eine Ferienreise oder eines Last Minute Rabatts für einen Flug.

Miles and More Als Vielflieger hat man die Möglichkeit Bonusmeilen zu sammeln. Hat man dann eine gewisse Anzahl an solchen Meilen gesammelt, kann man diese für Prämien einlösen. Zu den Prämien gehören unter Anderem Flüge, kurze Wochenendreisen, Reisetaschen, Uhren, Elektronikgeräte und diverse andere Dinge.

Reiserversicherung Als Reiseversicherung bezeichnet man Versicherungsverträge, die verschiedene Risiken im Zusammenhang mit Reisen abdecken. Dazu gehören z.B. eine Reiserücktrittskostenversicherung, eine Reisekrankenversicherung, eine Reiseunfallversicherung oder eine Reisegepäckversicherung. Es empfielt sich bei Reisen eine Versicherung abzuschließen, damit man sich im Urlaub keine Sorgen über Krankheit, Diebstahl oder Streitigkeiten machen muss.

Rechtliche Bestimmungen Ein Kunde kann sich über die rechtlichen Bestimmungen informieren, z.B. was er bei einer Buchung zu beachten hat oder welche Kosten bei einer Stornierung auf ihn zukommen würden.

Mit Hilfe der gefundenen Features lassen sich verschiedene Varianten des Systems bauen. Je nachdem welche Features man auswählt, entsteht ein ähnliches, aber unterschiedliches System. Möchte man z.B. eine Preisermäßigung für Flugtickets nicht erlauben, so lässt man einfach das Feature *Discount* weg. Somit können Kunden keinen Rabatt bekommen. Soll das System in einem Bereich eingesetzt werden, in dem alle Preise in nur einer einzigen Währung angegeben werden, kann man das Feature *Währungsumrechnung* vernachlässigen.

4.5 Zusammenspiel der Web Services und Features

In diesem Abschnitt werden die Änderungen, die die Features in den Web Services machen, genauer untersucht. Diese Änderungen sind im Quelltext der Web Services bzw. der Web Service Clients annotiert, um besser nachvollziehen zu

Webservice / Feature	Flugbuchung	Hotelbuchung	Mietwagenbuchung	Events & Tickets	Info & Service	Baukasten
Währungsumrechnung	C	C	C	C	C	C
Storno	I/C	I/C	I/C	I/C	----	I/C
Discount	I/C	I/C	I/C	I/C	C	I/C
Miles and More	I/C	I/C	I/C	I/C	C	----
Reiseversicherung	I/C	I/C	----	----	C	I/C
Rechtliche Bestimmungen	C	C	C	C	C	C

Abbildung 4.8: Übersicht Web Services und Features

Feature	# Web Services	# Methoden-rümpfe	# Signaturen	# Neue Methoden	# Veränderter HTML Seiten	LOC Java	LOC HTML
Währungsumrechung	6	----	----	----	12	145	87
Storno	5	4	----	4	15	425	262
Discount	6	5	4	----	9	76	181
Miles and More	5	9	4	4	14	473	783
Reiseversicherung	4	3	2	----	6	50	211
Rechtliche Bestimmungen	6	----	----	----	11	----	310

# Features, die Interface und Methoden verändern	4
# Features, die nur Code hinzufügen oder ändern	2

Abbildung 4.9: Änderungen durch Features (1)

können, durch welches Feature sie hervorgerufen werden. Dies erleichtert zudem auch das Finden aller Code-Stellen, die zu einem Feature gehören.

Es gibt zwei verschiedene Arten von Änderungen, die ein Feature an einem Web Service vornehmen kann (vlg. Abb. 4.8):

1. Es fügt zusätzlichen Quelltext in einen Web Service ein (**C**).

2. Es ändert das Interface eines Web Services, indem es neue Methoden hinzufügt oder die Signatur einer bestehenden Methode verändert ⇒ zusätzlicher Quelltext wird hinzugefügt (**I/C**).

Für die beiden Web Services *Mietwagenbuchung* und *Events und Tickets* lässt sich keine Versicherung abschließen, was mit – gekennzeichnet ist. Ebenso können Bonusmeilen nicht für den Web Service *Baukasten* genutzt werden, da hier eine Kombination von Flug und Hotel bereits eine Vergünstigung darstellt. Dies sind Annahmen die festgelegt worden sind und könnten auch anders interpretiert werden. Der Web Service *Währungskonversion* ist nicht in Abb. 4.8 zu sehen, da sich keines der Features auf das Interface oder die Implementierung auswirkt.

Abb. 4.9 zeigt eine Übersicht der Änderungen, die durch die Features hervorgerufen werden. Hier kann man unter Anderem sehen wie viele Web Services die einzelnen Features jeweils betreffen. Die Features *Währungsumrechnung*, *Discount*

```
1  ...
2  @WebMethod(operationName = "searchForFlights")
3  public Flight[] searchForFlights(
4    @WebParam(name = "fromAirport") String fromAirport,
5    @WebParam(name = "toAirport") String toAirport,
6    @WebParam(name = "departure") String departure,
7    @WebParam(name = "returnFlight") String returnFlight,
8    @WebParam(name = "number") int number,
9    // zusätzlicher Parameter
10   @WebParam(name = "discount") int discount) {
11   ...
12   }
```

Listing 4.2: Änderung der Methodensignatur (1)

```
1  ...
2  @WebMethod(operationName = "searchForHotels")
3  public Hotel[] searchForHotels(@WebParam(name = "land") String land,
4    @WebParam(name = "region") String region,
5    @WebParam(name = "city") String city,
6    @WebParam(name = "arrival") String arrival,
7    @WebParam(name = "departure") String departure,
8    @WebParam(name = "number") int number,
9    @WebParam(name = "category") int category,
10   @WebParam(name = "meals") String meals,
11   // zusätzlicher Parameter
12   @WebParam(name = "versicherung") int type) {
13 ...
14 }
```

Listing 4.3: Änderung der Methodensignatur (2)

und *rechtliche Bestimmungen* betreffen sechs Web Services, *Storno* und *Miles and More* nur fünf und das Feature *Reiseversicherung* betrifft nur vier Web Services. Die Zahlen ergeben sich aus Abb. 4.8. Dies zeigt wieder, dass Features mehrere Web Services gleichzeitig betreffen (vgl. Abschnitt 3.2).

In der Abb. 4.9 kann man auch erkennen, dass die Features *Discount*, *Miles and More* und *Reiseversicherung* Signaturen von Methoden der Web Services ändern. *Discount* ändert z.B. die Signatur der Methode **searchForFlights** des Web Services *Flugbuchung*, indem es einen zusätzlichen Parameter *discount* einfügt (vgl. List. 4.2 Zeile 10). Dadurch kann ein Kunden einen Rabatt auf Flugtickets bekommen.

Das Feature *Reiseversicherung* ändert z.B. auch die Signatur einer Methode und zwar **searchForHotels** des Web Services *Hotelbuchung*. Hier wird ebenfalls ein zusätzlicher Parameter *versicherung* hinzugefügt, was in List. 4.3 in Zeile 12 zu sehen ist. Somit hat ein Kunde die Möglichkeit eine Reiseversicherung abzuschließen.

Andere Features können den Web Services sogar neue Methoden hinzufügen. Das Feature *Storno* fügt z.B. dem Web Service *Flugbuchung* eine komplett neue

```
1  ...
2  <message name="stornoReservation">
3  <part name="parameters" element="tns:stornoReservation"></part>
4  </message>
5  ...
6  <operation name="stornoReservation">
7  <input message="tns:stornoReservation"></input>
8  <output message="tns:stornoReservationResponse"></output>
9  </operation>
10 ...
```

Listing 4.4: Änderungen im WSDL-Dokument durch das Feature *Storno*

Methode **stornoReservation** hinzu. Damit können bereits gebuchte Flüge wieder storniert werden. Diese Änderungen lassen sich auch im WSDL-Dokument des Web Services wiederfinden. List 4.4 zeigt einen Auschnitt aus dem WSDL-Dokument des Web Services *Flugbuchung*. Hier wird in den Zeilen 2 - 4 eine SOAP-Nachricht, die der Web Service bei seiner Kommunikation mit anderen Anwendungen verwendet, festgelegt und in den Zeilen 6 - 9 wird eine Request-Response-Operation definiert. Ein weiteres Feature, das eine Methode hinzufügt, ist *Miles and More*. Die Methode **encashMiles** ermöglicht, wie im vorherigen Abschnitt bereits beschrieben, das Einlösen von Bonusmilen für verschiedene Arten von Prämien.

Die Features *Währungsumrechnung* und *rechtliche Bestimmungen* fügen nur Änderungen im Quelltext der Web Services bzw. der Web Service Clients hinzu. Dies kann z.B. in Form eines Links geschehen über den man auf eine andere Seite kommt, auf der z.B. Informationen zur Stornierung von Mietwagen stehen (vgl. zweiten Link in Abb. 4.5) oder in Form einer Auswahlliste für die bevorzugte Währung, in der alle Preise angegeben werden sollen (vgl. Abb. 4.4).

In Abb. 4.9 ist auch noch zu sehen, dass Features HTML-Code und auch Java-Code hinzufügen. Dies ist zurückzuführen auf die Methodenaufrufe der Web Services (vgl. List. 4.1) und auf den zusätzlichen Quelltext der Features selbst. Man sieht in der Abbildung auch, wieviele HTML-Seiten die Features beeinflussen, was wieder das Querschneiden von Features deutlich macht. Ein Feature betrifft meist mehrere Web Services, Methoden, Signaturen und HTML-Seiten.

Eine Übersicht der Web Services und wie viele Features etwas ändern zeigt die Abb. 4.10. Wenn man den Web Service *Flugbuchung* betrachtet, kann man sehen, dass zwei Features dem Web Service Methoden hinzufügen, drei Features Signaturen von Methoden ändern und sechs Features Quell-Code verändern oder hinzufügen. Desweiteren ist angegeben, wie oft die Methoden des Web Services aufgerufen werden.

4.6 Fazit

Nach anfänglichen Schwierigkeiten bei der Suche nach einer bereits bestehenden SOA, um diese dann Feature-orientiert zu analysieren, wird kurzer Hand ein

Web Service	# Features die Methoden hinzufügen	# Features die Signaturen ändern	# Methodenaufrufe durch Features	# Features die Code verändern	LOC HTML + Java (ohne Features)
Flugbuchung	2	3	7	6	726
Hotelbuchung	2	3	7	6	724
Mietwagenbuchung	2	2	4	5	664
Events & Tickets	2	2	4	5	709
Info & Service	----	----	5	5	909
Baukasten	1	1	----	5	423
Währungskonversion	----	----	15	----	44

Abbildung 4.10: Änderungen durch Features (2)

eigenes System mit Hilfe von Web Services entwickelt. Zusammenfassend können die folgenden Punkte festgehalten werden:

Entwicklungsumgebung: Mit der verwendeten NetBeans 6.5 IDE lassen sich Web Services sehr leicht erstellen. Im Internet und auf der Homepage von NetBeans finden sich einige sehr hilfreiche Anleitungen, wie man Web Services entwickeln kann. Desweiteren beinhaltet das NetBeans Bundle einen vorkonfigurierten Applikationsserver, der für die Bereitstellung der Web Services zuständig ist, damit diese von Web Service Clients genutzt werden können.

Flugbuchung-Szenario: Das erstellte System ist nicht völlig aus dem Raum gegriffen, sondern orientiert sich an bereits bestehenden Reiseportalen. Es werden mehrere solcher Reiseportale im Internet untersucht und dabei stellt sich heraus, dass viele davon sehr ähnliche Funktionalitäten, wie z.B. Flug-, Hotel- und Mietwagenbuchung anbieten.

Dieses Flugbuchung-Szenario beschränkt sich auf die sieben Web Services, die in Abb. 4.10 dargestellt sind. Aber nicht jede, für dieses System benötigte, Funktionalität lässt sich in einem separatem Web Service kapseln. Daher werden Features benötigt. Mit Hilfe dieser Features ist man in der Lage, verschiedene Varianten des Systems zu bauen. Insgesamt werden sechs Features gefunden, die in Abb. 4.9 zu sehen sind.

Web Services und Features: Es werden die einzelnen Web Services, ihre Funktionalität, sowie die gefundenen Features und die Änderungen, die sie an den Web Services vornehmen, genauer vorgestellt. Dabei wird gezeigt, dass Features querschneiden und meist mehrere Web Services gleichzeitig betreffen. Drei Features betreffen sechs Web Services, zwei betreffen fünf und ein Feature betrifft vier Web Services (vgl. Abb. 4.9).

Sie können sich auf das Interface und den Quell-Code eines Web Services auswirken oder nur auf den Quell-Code. Dabei verändern vier Features die Interfaces von Web Services und somit auch ihre Methoden und zwei Features ändern oder fügen nur Quell-Code hinzu (vgl. Abb. 4.9).

Implementierung: Die Implementierung der Web Services beschränkt sich auf ein Minimum. Es geht nicht unbedingt darum, was die Web Services bei einem Aufruf zurückliefern (keine Datenbankanbindung), sondern um das Zusammenspiel von Web Services und Features:

- Welche Featues sind vorhanden?
- Welche Änderungen machen sie?
- Wie viele Web Services betreffen sie?

Die Stellen, an denen die Features Änderungen vornehmen, sind mit Kommentaren versehen. Dadurch lassen sich die Änderungen, die von den Features verursacht werden, leichter wiederfinden.

Mit Hilfe von Web Service Clients wird eine Testumgebung erstellt. Sie besteht aus insgesamt 23 Java-Dateien und 54 *JavaServer Pages (JSP)*. JSP ist definiert als:

> „JavaServer Pages (JSP) technology enables Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of Web-based applications that are platform independent. JSP technology separates the user interface from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content.“[6]

[6] Quelle: http://java.sun.com/products/jsp/overview.html

Kapitel 5

Zusammenfassung und Ausblick

In dieser Bachelorarbeit wird auf der Grundlage einer Service-orientierten Architektur eine Feature-orientierte Analyse durchgeführt. Die SOA ist einer der modernsten Trends der IT und insbesondere im Bereich des Management sehr beliebt, da sie sich häufig an Geschäftsprozessen orientiert. Die drei wichtigsten Konzepte der SOA sind *Virtualisierung/Sprachenunabhängigkeit*, *Verteilung* und *Entkopplung*. Zur technischen Realisierung dieser Software Architektur werden Web Services verwendet.

Web Services basieren auf modernen und offenen Standards und bieten Dienste an. Sie besitzen eine klar definierte Schnittstelle und tauschen, über das XML-basierte Protokoll SOAP, Daten aus. Zum Senden von Nachrichten kann ein beliebiges Transportprotokoll verwendet werden. Am häufigsten tritt die Kombination SOAP zusammen mit HTTP auf.

Mit Hilfe eines WSDL-Dokuments, in dem Informationen über die angebotenen Funktionen, Daten, Datentypen und Austauschprotokolle eines Web Services gespeichert sind, können angebundene Anwendungen erkennen, welche Operationen, sowie deren Parameter und Rückgabewerte, von außen zugänglich sind.

Um Web Services zu veröffentlichen, wird von Dienstanbietern der standardisierte Verzeichnisdienst UDDI benutzt. Dieser Verzeichnisdienst wird ebenfalls von Dienstkonsumenten verwendet, um einen für ihre Bedürfnisse passenden Web Service zu finden.

Einer der größten Vorteile einer SOA ist, dass die Services und die sie benutzenden Anwendungen in grundverschiedenen Programmiersprachen und Plattformen implementiert sein können, da die Services unabhängig von Middleware, Hardware und Betriebssystemen sind. SOAs bringen jedoch auch noch einige Probleme mit sich, was z.B. die Modularität oder Variabilität von Services anbelangt.

Es gibt nämlich Funktionalitäten, die man nicht so einfach in einem einzelnen Service kapseln kann. Diese Funktionalitäten werden durch *Features* repräsentiert und sind oft über mehrere Services gleichzeitig verstreut. Formal definiert ist ein Feature eine semantisch zusammenhängende Einheit, die einer Anforderung an ein Programm entspricht, zur Spezifikation von Programmen dient und eine Konfigurationsmöglichkeit bietet.

Man spricht auch von einem Querschneiden der Features. In der vorgestellten Fallstudie lässt sich dies z.B. an dem Feature *Discount* sehr gut verdeutlichen. Die Implementierung dieses Features betrifft alle Web Services, d.h. es werden Änderungen an den Interfaces der Web Services vorgenommen bzw. Quelltext

hinzugefügt. Diese mangelnde *Feature-Kohäsion* führt zu dem sog. *Feature Traceability Problem*.

Die Feature-orientierte Programmierung ist in der Lage, diese Probleme zu lösen, indem jedes Feature durch ein *Feature-Modul* implementiert wird. Dadurch wird eine Trennung und Modularisierung von Features erreicht und die Komposition von Features vereinfacht. Hierfür müssen die gefundenen Features mit Hilfe von Dekompositionstools herausgelöst werden. Deshalb werden die Änderungen, die von den Features vorgenommen werden, im Quelltext der Web Services annotiert, damit man sie später leichter wiederfinden und den Features zuordnen kann.

Zur Implementierung von Feature-Modulen können Tools, wie z.B. *AHEAD Tool Suite*[1] oder *FeatureHouse*[2] verwendet werden. Diese Tools müssen jedoch mit HTML-Code umgehen können. Für die genannten Tools wird zusätzlich ein *Featue-Modell* benötigt, das erst noch aus den gefunden Features erstellt werden muss. Feature-Modelle sind hierarchisch angeordnete Mengen von Features, die die Beziehungen zwischen Eltern-Features und Kind-Features darstellen.

Das Zusammenspiel von Services und Features wird anhand einer Fallstudie untersucht. Zunächst wird nach bereits vorhandenen Service-orientierten Architekturen im Internet gesucht, mit dem Ziel diese Feature-orientiert zu analysieren. Diese Analyse ist aber nicht möglich, da kein Zugang zur Implementierung der Services besteht. Daher wird eine SOA selbst entwickelt.

In einem ersten Schritt werden diverse Reiseportale analysiert:

- Welche Services werden angeboten?
- Lassen sich Features finden und herauskristallisieren?
- Betreffen die Features mehrere Services?
- Kann man bestimmte Features auch weglassen?
- Betreffen die Features die Implementierung oder das Interface der Services oder beides?
- Gibt es Features, die sich gegenseitig ausschließen?
- Wie erfolgen die Aufrufe zwischen den Services?

Im zweiten Schritt werden dann sieben konkrete Services, die in fast allen Reiseportalen enthalten sind, ausgewählt und auf der Grundlage dieser Services wird das Flugbuchung-Szenario erstellt. Für die Implementierung der Services wird die Java NetBeans 6.5 IDE verwendet. Es wird auch eine Testumgebung implementiert, mit der die Methoden der Services genutzt werden können. Danach werden die Features, die sich aus diesem System herauskristallisieren lassen, analysiert und im Quelltext der Services bzw. der Testumgebung markiert.

[1] http://www.cs.utexas.edu/users/schwartz/ATS.html
[2] http://www.fosd.de/featurehouse

Es werden anhand des erstellten Flugbuchung-Szenarios Statistiken erhoben, um den Einfluss der Features auf die Services zu quantifizieren. Dabei wird festgestellt, dass bereits in einem kleinen System wie diesem, Features querschneiden und mehrere Services gleichzeitig betreffen. Es kann auch festgehalten werden, dass die Services meist von mehreren Features betroffen sind.

Die Annotation der Änderungen durch die Features hilft dann, in einem weiteren Schritt, beim Herauslösen der Features mit Hilfe der oben genannten Dekompositionstools. Die damit erreichte Modularität erlaubt eine Erstellung von Varianten anhand einer *Feature-Selektion*. Dabei wird aus allen vorhandenen Features eine Menge von Features ausgewählt, die das Programm bildet. Eine solche Programmerstellung soll dann mit Tools und per Knopfdruck möglich sein.

Zusätzlich müssen auch noch weitere Fallstudien durchgeführt werden. Diese Arbeit ist nur ein erster Schritt in diese Richtung und in der Zukunft kann man aus den Erfahrungen dieser Arbeit ein Konzept entwickeln für die Softwareentwicklung, auf Basis von Services und Features.

Kapitel 6

Enthaltene Software

Diese Arbeit beinhaltet eine CD-ROM mit folgendem Inhalt:

- Quelltext aller Web Services
- Quelltext aller Web Service Clients (Testumgebung)
- Digitale Version dieser Ausarbeitung im PDF-Format

Kapitel 7

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbsständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

_____

Daniel Rolf

Literaturverzeichnis

[AE03] Andreas Eberhart, Stefan F.: *Web Services: Grundlagen und praktische Umsetzung mit J2EE und .NET*. München, Wien : Carl Hanser Verlag, 2003. – ISBN 3–446–22530–7

[AKL08] Apel, Sven ; Kaestner, Christian ; Lengauer, Christian: Research challenges in the tension between features and services. In: *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–029–6, S. 53–58

[AL08] Apel, Sven ; Lengauer, Christian: Superimposition: A Language-Independent Approach to Software Composition. In: Pautasso, Cesare (Hrsg.) ; Tanter, Éric (Hrsg.): *Software Composition* Bd. 4954, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–78788–4, 20–35

[ALMK08] Apel, Sven ; Lengauer, Christian ; Möller, Bernhard ; Kästner, Christian: An Algebra for Features and Feature Composition. In: Meseguer, José (Hrsg.) ; Rosu, Grigore (Hrsg.): *AMAST* Bd. 5140, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–79979–5, 36–50

[Ape08] Apel, Sven: *Moderne Programmier Paradigmen*. Vorlesung Universität Passau - Fakultät für Informatik und Mathematik, 2007/2008

[Bat05] Batory, Don: Feature Models, Grammars, and Propositional Formulas / The University of Texas at Austin, Department of Computer Sciences. Version: April 11 2005. ftp://ftp.cs.utexas.edu/pub/techreports/tr05-14.pdf. 2005 (CS-TR-05-14). – Forschungsbericht. – Fri, 28 Sep 107 13:03:38 GMT

[Bat06] Batory, Don S.: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In: Lämmel, Ralf (Hrsg.) ; Saraiva, João (Hrsg.) ; Visser, Joost (Hrsg.): *GTTSE* Bd. 4143, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–45778–X, 3–35