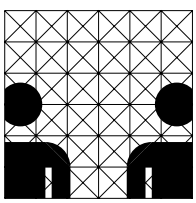
Henning Schwentner

Wider die enge Kopplung

Große Refactorings in einem Prozess zur Entflechtung von Anwendung und Rahmenwerk

Diplomarbeit

Department Informatik



Universität Hamburg

Henning Schwentner
Wider die enge Kopplung

Diplomarbeit

Wider die enge Kopplung

Große Refactorings in einem Prozess zur Entflechtung von Anwendung und Rahmenwerk

vorgelegt von
Jens Henning Schwentner
aus Hamburg

im Jahre 2008

Universität Hamburg

Jens Henning Schwentner
Vehrenkampstraße 1
22 527 Hamburg
Matr.Nr.: 522 42 88
henning.schwentner@hamburg.de

Erstbetreuer
Dr. Wolf-Gideon Bleek
Zentrum für Architektur und Gestaltung von IT-Systemen (AGIS)
Department Informatik
Universität Hamburg

Zweitbetreuer
Prof. Dr. Winfried Lamersdorf
Zentrum für Verteilte Informations- und Kommunikationssysteme (VIKS)
Department Informatik
Universität Hamburg

Für Lennart

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Definitionsverzeichnis

1 Einleitung

Ein Softwaresystem zu *entwerfen* heißt, es in Module zu dekomponieren (Brügge u. Dutoit 2004, S. 223). In traditionellen Vorgehensmodellen wie dem Wasserfallmodell wird der Entwicklungsprozess so modelliert, dass der Entwurf vor Beginn der Implementierung abgeschlossen sein muss (Royce 1987; Boehm 1979). Die Geschichte der Software-Entwicklung zeigt aber, dass es nur in der Theorie möglich ist, den perfekten Entwurf für ein Programm *vor* seiner Implementierung zu machen (Roock u. Lippert 2004, S. 76). Glücklicherweise ist das nicht nötig; denn der Entwurf eines Software-Systems kann *während* seiner Entwicklung verbessert werden. Dazu steht in modernen Vorgehensmodellen die Technik der *Refactorings* zur Verfügung. Ein Refactoring ist eine Änderung einer Software, bei der die Struktur (und damit der Entwurf) einer Software verbessert wird, ohne ihre Funktionalität zu verändern (Fowler 2000).

Ziel beim Entwurf eines Systems ist, dass seine Module möglichst *lose Kopplung* aneinander haben. Kopplung ist die Zahl der Abhängigkeiten zwischen zwei Modulen (Brügge u. Dutoit 2004, S. 230). Heutige Anwendungssoftware wird nicht alleinstehend entworfen, sondern so, dass sie bereits entwickelte Software – wie etwa ein Rahmenwerk – *wiederverwendet*. Das Prinzip der losen Kopplung ist auf die Software-Wiederverwendung übertragbar. Beim Entwurf muss dann die Frage beantwortet werden, welche Module überhaupt abhängig von einem bestimmten Rahmenwerk sein dürfen. Das Rahmenwerk wird durch die Wiederverwendung ein Modul des Entwurfs.

Auch ein Entwurf, der wiederverwendete Software enthält, kann verbesserungswürdig sein. Dies ist der Fall, wenn eine zu enge Kopplung zwischen Anwendung und Rahmenwerk besteht. Die Kopplung ist zu eng, wenn andere als die erlaubten Module abhängig von dem Rahmenwerk sind. Enge Kopplung an ein Rahmenwerk wirft verschiedene Problem auf; so ist es bei enger Kopplung z. B. schwierig oder sogar unmöglich, das Rahmenwerk durch ein anderes, das eine ähnliche Funktionalität bietet, zu ersetzen. Wir glauben, dass es für dieses Entwurfsproblem passende Refactorings gibt,

die eine Anwendung, die zu eng an ein Rahmenwerk gekoppelt ist, von ihm lösen können. Erstes Ziel der vorliegenden Arbeit ist es, solche Refactorings zu finden. Deshalb sollen zunächst die Probleme beschrieben werden, die durch Rahmenwerkskopplung entstehen und dann ein Lösungsansatz mit Refactoring gesucht werden. Dazu muss untersucht werden, ob und wenn ja, wie die Kopplung gelockert werden kann. Durch diesen Lockerungsprozess soll sich nichts an der Funktion der Software sondern nur an Ihrer Struktur ändern. Daher bietet es sich an, Refactorings zu verwenden und zu suchen.

Es gibt eine unüberschaubare Vielzahl von Rahmenwerken unterschiedlicher Typen für unterschiedliche Aufgaben. Wir wollen daher unsere Untersuchung an einem relevanten Beispielrahmenwerk beginnen und dann betrachten, wieweit die Ergebnisse dieser Untersuchung auf andere Rahmenwerke übertragbar sind. Im Internet-Zeitalter hat sich eine besondere Art Programm entwickelt: die Webanwendung. Viele Webanwendungen werden in der Programmiersprache Java mit einem Rahmenwerk aus der »Java Platform, Enterprise Edition« (JEE) entwickelt: dem sogenannten Servlet-Rahmenwerk. Dieses soll deshalb unser Beispielrahmenwerk sein. Wir werden also zunächst Kopplung an das Servlet-Rahmenwerk untersuchen.

Es stellt sich die Frage, warum wir uns für Rahmenwerkskopplung und Refactorings dagegen interessieren. Ein Entwurf steht nicht für sich allein, sondern gehört zu einem konkreten Programm. Entwurfsprobleme treten deshalb nicht im luftleeren Raum sondern in konkreten Systemen auf. Ein Entwurfsproblem ist auch nur dann relevant, wenn es ein System gibt, das dieses Problem hat. Ein Entwurfsproblem zu lösen, heißt deswegen immer auch, den Entwurf (mindestens) eines konkreten Programms zu verbessern. Das in dieser Arbeit untersuchte Entwurfsproblem der zu engen Kopplung an ein Rahmenwerk – konkreter das Servlet-Rahmenwerk – wurde in der Webanwendung JCommSy beobachtet. Aus dem Wunsch, JCommSy und Servlet-Rahmenwerk zu entflechten, d. h. die Kopplung zwischen beiden zu lockern, entstand diese Diplomarbeit.

JCommSy ist die Java-Ausprägung von CommSy, einer Anwendung, die Projekt-basiertes Lernen unterstützt. Ursprünglich wurde CommSy in PHP, später in Java neu implementiert. Die Entwicklung von CommSy befindet sich derzeit in einer Migrationsphase von PCommSy (der Ausprägung in PHP) zu JCommSy (der Ausprägung in Java). JCommSy wird von Forschern und Studenten im Zentrum AGIS an der Universität Hamburg entwickelt. Über die Jahre wuchs das System, und es entstand eine Architektur, in die sich Mängel einschlichen.

JCommSy verwendet das Servlet-Rahmenwerk aus der JEE. Dieses wird eingesetzt, um die technischen Aspekte der Oberfläche des Systems und der Interaktion mit dem Benutzer zu realisieren. Von den Architekturelementen des JCommSy sollten deshalb nur jene Elemente das Servlet-Rahmenwerk verwenden, welche die Oberflächen-Aspekte implementieren. Derzeit verwenden weitere Architekturelemente das Rahmenwerk, wodurch eine enge Kopplung daran entsteht. Dies hat verschiedene Nachteile; u. A. ist es schwierig, die an das Rahmenwerk gekoppelten Klassen zu testen und es ist fast unmöglich, zu einer anderen Oberflächentechnologie zu wechseln. Es ist wünschenswert, die Anwendung in einen Zustand zu überführen, in dem sie nur lose an das Servlet-Rahmenwerk gekoppelt ist. Wie dieser Zustand erreicht werden kann, ist vor Beginn der Arbeit nicht klar.

Zweites Ziel der Arbeit ist es, einen Prozess zu entwickeln, in dem JCommSy entsprechend umgebaut werden kann. JCommSy soll dabei als Beispiel dienen, an dem die entwickelten Refactorings ausprobiert werden.

Um die Architektur eines Systems zu ändern, müssen wir diese Architektur kennen. Deshalb soll die Architektur von JCommSy aufgearbeitet und präsentiert werden. Besonderes Augenmerk werden wir hierbei auf den in der vorangegangenen Arbeit von Hohmann (2007) eingeführten Architekturstil der *Seitenfragmente* legen.

Das erste Ziel (Finden von Refactorings gegen Rahmenwerkskopplung) erwächst aus dem zweiten (Entflechtung von JCommSy und Servlet-Rahmenwerk). Allgemeingültige Konzepte und Techniken sollen von dem konkreten Anwendungsfall JCommSy gelöst und in einer abstrahierten Form dargestellt werden, sodass sie später in anderen Projekten mit ähnlichen Problemen wiederverwendet werden können. Es stellt sich die Frage, inwieweit die Techniken übertragbar sind. Sie könnten übertragbar sein auf erstens: andere Anwendungen als JCommSy und zweitens: andere Rahmenwerke als das Servlet-Rahmenwerk. Entsprechend sollen am Ende der Arbeit zwei Ergebnisse stehen: ein JCommSy mit verbesserter Architektur und ein Katalog von übertragbaren Techniken.

Aufbau der Arbeit

In *Kapitel 2* werden wir zunächst die benötigten Begriffe einführen und definieren. Aus der Literatur übernehmen wir Entwurf, Kopplung, Wiederverwendung, Architektur, Tests, Refactoring. Weil JCommSy eine Weban-

wendung ist, gehen wir kurz auf die Besonderheiten dieses Programmtyps ein. Neue Begriffe sind Voraussetzung und Abhängigkeit, die wir benötigen, um zu beurteilen, inwieweit Kopplung an ein Rahmenwerk notwendig ist.

Um zu begründen, warum wir gegen Rahmenwerkskopplung vorgehen wollen, werden wir in *Kapitel 3* zunächst die Probleme untersuchen, die entstehen, wenn eine Anwendung zu eng an das Servlet-Rahmenwerk gekoppelt ist. Als zweites präsentieren wir einen Lösungsansatz für Software, die unter diesen Problemen leidet. Mit diesem Ansatz kann die Kopplung in zwei Schritten gelockert werden. Dazu müssen zunächst Vorbedingungen ausdrücklich gemacht werden und dann nach Möglichkeit in Abhängigkeiten umgewandelt werden.

In *Kapitel 4* wenden wir den Lösungsansatz an der Webanwendung JCommSy an. Dabei zeigt sich, dass JCommSy einen besonderen Architekturstil hat – die Seitenfragmentarchitektur – und dass der Prozess der Entflechtung bei JCommSy besonders aufwendig ist. Unser erster Lösungsansatz reicht nicht aus, weil die Kopplung in JCommSy in der Signatur überschriebener Methoden sitzt und die Signatur von überschriebenen Methoden schwer zu ändern ist.

In *Kapitel 5* betrachten wir das zweite Problemfeld: Das Ändern der Signatur von überschriebenen Methoden. Wenn man die Signatur einer überschriebenen Methode ändern will, dann muss man die Signatur sämtlicher überschreibender Methoden mitändern. Wir präsentieren einen Lösungsansatz, bei dem zunächst die überschriebene Methode und dann die überschreibenden Methoden einzeln angepasst werden können. Das Besondere an diesem Ansatz ist, dass das System jeweils zwischen zwei Schritten voll übersetzungs- und lauffähig ist.

In *Kapitel 6* kehren wir dann wieder zu JCommSy zurück und beschreiben den Prozess, in dem JCommSy und Servlet-Rahmenwerk entflochten werden.

In *Kapitel 7* präsentieren wir schließlich die Ergebnisse der Arbeit, d. h. inwieweit wir die gesteckten Ziele erreichen.

2 Grundlagen

In diesem Kapitel werden wir uns mit dem nötigen Handwerkszeug aus der Softwaretechnik versorgen, das wir brauchen, um unsere Untersuchungen durchzuführen.

Um zu begründen, warum es sich überhaupt lohnt, den Entwurf eines Softwaresystems zu verbessern, betrachen wir zunächst den Begriff Qualität. Entscheidend für eine hohe Qualität ist ein guter Entwurf; darum beleuchten wir die Prinzipien, die hinter einem guten Entwurf und dem damit verbundenen Begriff der Software-Architektur stehen. Unsere Untersuchung hat zum Ziel, die Kopplung an ein Rahmenwerk zu lockern; deshalb gehen wir auf die unterschiedlichen Arten der Software-Wiederverwendung ein. Für das Sicherstellen hoher Qualität ist das darauffolgend betrachtete Testen sehr wichtig. Weil wir den Entwurf unserer Anwendung im Nachhinein ändern wollen, schauen wir uns die dafür entwickelte Technik Refactoring an. In der vorliegenden Arbeit wird mit Refactorings die Signatur von Methoden geändert. Den Begriff der Signatur betrachten wir im Abschnitt über Prozessabstraktion. Weil das Rahmenwerk, von dem wir die Anwendung lösen wollen, ein Web-Rahmenwerk ist, betrachten wir die grundlegenden Konzepte von Webanwendungen. Für die über die Arbeit verteilten Diagramme erläutern wir die verwendete UML-Notation. Um beurteilen zu können, inwieweit Kopplung an ein Rahmenwerk notwendig oder überflüssig ist, führen wir die Begriffe Voraussetzung, Vorbedingung und Abhängigkeit ein. Abschließend geben wir eine Zusammenfassung verwandter Arbeiten.

2.1 Softwarequalität

Gegenstand der Wissenschaft *Softwaretechnik* ist die professionelle Entwicklung von großen Softwaresystemen, speziell Anwendungssoftware (Floyd u. Züllighoven 2002, S. 764). Der englische Name der Softwaretechnik – *software engineering* – hebt die Vorstellung hervor, dass der Softwareentwicklungsprozess ingenieurmäßig ablaufen soll. Ziel ist es, Software mit hoher Qualität

herzustellen. CommSy ist Anwendungssoftware, deshalb sollte es nach den Prinzipien der Softwaretechnik entwickelt werden. Der Begiff der Qualität wird in der Literatur folgendermaßen definiert:

Definition 2.1 (Qualität)
Qualität *ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen (DIN 55350 1995).*

Ein wesentlicher Faktor, der die Qualität einer Software bestimmt, ist also ihre Funktionalität. Die Qualität von Software hängt aber auch von *nicht-funktionalen Eigenschaften* ab, d. h. Eigenschaften, die nicht direkt mit dem zusammenhängen, was die Software leistet, sondern die ihr Verhalten zur Laufzeit sowie Struktur und Organisation von Quelltext und Dokumentation bestimmen (Sommerville 2001, S. 27). Wichtige nicht-funktionale Eigenschaften von Software sind Wartbarkeit, Anpassparkeit, Zuverlässigkeit und Robustheit (Claus u. Schwill 2006, S. 622). Ein entscheidender Faktor dafür ist ein guter Entwurf.

Boehm hebt besonders hervor, dass die Wartung von Software (engl.: *software maintenance*) Teil der wissenschaftlichen Disziplin »Softwaretechnik« ist (Boehm 1979). Das bedeutet, dass nicht nur während der Erstentwicklung, sondern auch im Laufe des Einsatzes einer Software ihre Qualität und damit ihr Entwurf verbessert werden kann und soll. Für diese nachträgliche Verbesserungen wurde die Technik des Refactoring eingeführt (Abschnitt 2.6).

2.2 Entwurfsprinzipien

Ein Software-System ist ein komplexes Gebilde, dass nicht auf einmal in allen Einzelheiten verstanden werden kann. Deshalb wird es *struktiert*, d. h. das System wird in *Module zerlegt* oder *dekomponiert*. Die Tätigkeit des Strukturierens nennen wir auch *Entwurf* (Brügge u. Dutoit 2004, S. 223). Der Begriff des Moduls ist hier ganz allgemein verwendet; er umfasst z. B. die Operation und den abstrakten Datentyp.

Brooks geht davon aus, dass es keinen Weg gibt, die Produktivität von Softwareentwicklern um eine Größenordnung zu verbessern, so wie es in der Hardwareentwicklung in den letzten Jahrzehnten mehrmals der Fall war (Brooks 1987). Trotzdem gibt es Entwurfskonzepte, die, wenn sie angewandt

werden, zu besserer Qualität der Software führen. Zu diesen gehören das Geheimnisprinzip, lose Kopplung und hohe Kohäsion (Winter 2005, S. 331 ff), die wir im Folgenden erläutern werden.

Das *Geheimnisprinzip* (engl. *information hiding*), erstmals beschrieben von Parnas (1971), besagt, dass zwischen zwei Modulen nur soviele Verbindungen bestehen dürfen wie unbedingt notwendig. »Die Verbindungen zwischen Modulen sind die Annahmen, die die Module übereinander machen«[1] (Parnas 1971, S. 339). Hierbei ist jedes Modul »charakterisiert durch sein Wissen über eine Entwurfsentscheidung, die es vor allen anderen versteckt«[2] (Parnas 1972, S. 1056). Parnas kommt zu dem Ergebnis, dass das Kriterium, nach denen man Softwaresysteme in Module aufteilen sollte, das Geheimnisprinzip ist (Parnas 1972). Jede Entwurfsentscheidung soll also in *genau einem* Modul gekapselt sein. Andere Module dürfen die Entscheidung nicht kennen.

In Systemen, die nach dem Geheimnisprinzip dekomponiert sind, haben die Module deshalb zwei Teile: eine *Schnittstelle* – die die anderen Module kennen dürfen – und eine *Implementierung* – die die zu versteckende Entwurfsentscheidung enthält und entsprechend die anderen Module nicht kennen dürfen – (Balzert 2000, S. 1050). Die Schnittstelle ist also eine Abstraktion von der Implementierung. Für die vorliegende Arbeit sind zwei Arten von Modulen in diesem Sinne wichtig, die in objektorientierten Systemen auftreten: Methoden (bieten Prozessabstraktion) und Klassen (bieten Datenabstraktion).

Kopplung ist die Zahl der Abhängigkeiten zwischen zwei Modulen (Brügge u. Dutoit 2004, S. 230). Kohäsion (engl. *cohesion*[3]oder *strengh*) ist die Zahl der Abhängigkeiten innerhalb eines Moduls (Brügge u. Dutoit 2004, S. 232). Ziel beim Entwurfs ist es, ein System zu erhalten, das eine geringe *Kopplung* (*coupling*, d. h. seine Module sind nur durch möglichst enge Schnittstellen mit einander verbunden, Balzert 1998, S. 474) und eine hohe *Kohäsion* (d. h. jedes Modul enthält nur Komponenten, die zueinander in Beziehung stehen und ähnliche Aufgaben erfüllen, Brügge u. Dutoit 2004, S. 232) hat (Balzert 2000, S. 1032). Die Zerlegung eines Systems nach dem Geheimnisprinzip fördert lose Kopplung und hohe Kohäsion. Ein so zerlegtes System hat von-

[1] Im Original: "The connections between modules are the assumpions which the modules make about each other."

[2] Im Original: "Every module [...] is chararacterized by its knowledge of a design decision which it hides from all others."

[3] Kohäsion und *cohesion* leiten sich von latein. *cohaerere* = zusammenhängen ab.

einander unabhängige Module. Dies wirkt sich vorteilhaft auf Lesbarkeit, Testbarkeit, Anpassbarkeit und Wartbarkeit und damit auf die Qualität der Module und des Gesamtsystems aus.

2.3 Software-Architektur

Wenn man ein Softwaresystem entwirft, entsteht eine Architektur. Wir definieren mit Bass, Clements, u. Kazman (2003):

Definition 2.2 (Software-Architektur)
Die Software-Architektur *eines Programms ist die Struktur oder die Strukturen des Systems, die besteht aus: Software-Elementen, den nach außen sichtbaren Eigenschaften dieser Elemente und den Beziehungen zwischen ihnen (Bass u. a. 2003, S. 3).*

Für die vorliegende Arbeit ist nur diese statische Sicht (Reussner u. Hasselbring 2006, S. 38) auf die Architektur wichtig.

Eine Menge von Klassen, die ein Element im Sinne von Definition 2.2 bilden, nennen wir ein *Architekturelement*. Architekturelemente können andere Architekturelemente enthalten. Die Architekturelemente lassen sich in Arten einteilen. Typische Beispiele für Architekturelementarten sind *Schicht* und *Subsystem*. Welche Art Architekturelemente ein System hat, hängt u. A. von seinem Architekturstil (auch: Architekturmuster, engl. *architectural pattern*, Bass u. a. 2003, S. 24 f) ab. Beispiel für einen häufig auftretetenden Architekturstil ist die Schichtenarchitektur (Lilienthal 2008).

Auch für Software-Architektur sind Geheimnisprinzip, Kopplung und Kohäsion maßgebliche Prinzipien (Floyd 2007). Architekturelemente sind Module (und bieten Datenabstraktion) in Sinne von Abschnitt 2.2.

2.4 Software-Wiederverwendung

Ein Faktor, der zur Qualität eines Softwaresystems beiträgt, ist die *Software-Wiederverwendung*. Software-Wiederverwendung ist der Prozess des Erzeugens von Software aus bereits bestehenden Software-Bausteinen anstatt einer Neuentwicklung von Grund auf (Krueger 1992, S. 131). Wiederverwendung kann die Entwicklungszeit (und damit den Preis) eines Softwaresystems verringern (Karlsson 1995, S. 8 ff).

Einer der Vorteile des objektorientierten Programmierens ist die gute Unterstützung für Wiederverwendung. Allerdings ist objektorientierte Software nicht automatisch leicht wiederzuverwenden. Damit dies leicht möglich ist, muss sie speziell für Wiederverwendung entworfen sein. (Johnson u. Foote 1988)

Es gibt unterschiedliche Arten von Software, die speziell für die Wiederverwendung entworfen wurde, so z. B. Bibliotheken, Rahmenwerke und Komponenten. Diese Unterscheidung ist für die vorliegende Arbeit nicht wichtig; weil der Begriff »wiederverwendete Software« etwas sperrig ist, sprechen wir im Laufe der Arbeit einfach von der im Beispielsystem JCommSy eingesetzten Variante wiederverwendeter Software, dem »Rahmenwerk«. Ein für die Arbeit wichtiges Rahmenwerk ist das Servlet-Rahmenwerk, das wir in Abschnitt 2.8.2 näher betrachten werden.

Verwendet ein System ein Rahmenwerk wieder, dann stellt sich die Frage, an welcher Stelle der Architektur das Rahmenwerk einzuordnen ist und welche Teile des Systems das Rahmenwerk verwenden dürfen. Aus dem Geheimnisprinzip und dem Ziel eine möglichst lose Kopplung zu erreichen folgt:

Architekturregel für die Verwendung von Rahmenwerken
Ein Rahmenwerk soll nicht von mehr als einem Architekturelement verwendet werden. ✍

Diese Regel gilt nur für eine sinnvolle Größe der Architekturelemente. Was als eine sinnvolle Größe betrachtet werden kann, hängt vom jeweiligen System, Architekturstil und Architekten ab. In einem System mit Schichtenarchitektur ist eine sinnvoll große Architekturelementart die Schicht.

2.5 Testen

Die Literatur ist sich einig, dass das Testen ein wichtiger Teil der Sicherung von Softwarequalität und dementsprechend des Softwareenwicklungsprozesses ist. (Balzert 2000; Zuser u. a. 2001, S. 193 und Floyd u. Oberquelle 2007, Folien 12.8 ff). Weniger Einigkeit besteht darüber, *was* Testen ist. Eine vielzitierte Definition hat Myers formuliert:

> Testen ist der Prozess des Ausführens eines Programms mit der Absicht, Fehler zu finden.[4] (Myers u. a. 2004, S. 6)

[4]Im Original: "Testing is the process of executing a program with the intent of finding

Andere Autoren legen den Schwerpunkt darauf, Korrektheit zu zeigen:

> Test ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteins relativ zu vorher festgelegten Anforderungen. (Denert 1991, S. 355)

Beide Aspekte – Fehlerfindung und Korrektheitsnachweis – spielen bei der Softwareerstellung eine wichtige Rolle. Einerseits kann ein Softwaresystem nur dann eine hohe Qualität haben, wenn es wenige (oder gar keine) Fehler enthält. Deshalb ist es wichtig, die vorhandenen Fehler zu finden. Andererseits kann ein gut formulierter Testfall einen entscheidenden Teil der Spezifikation des Systems ausdrücken. Wenn solch ein Testfall positiv abläuft, kann damit durchaus Korrektheit zu den im Testfall kodierten Anforderungen nachgewiesen werden. Dies ist insbesondere beim Ändern des Systems wichtig. Dazu definieren wir:

Definition 2.3 (Regressionstest)
Ein Test, der dazu dient, sicherzustellen, dass eine Änderung keine unbeabsichtigten Effekte hervorruft und dass die getestete Komponente ihre Anforderung weiterhin erfüllt, heißt Regressionstest *(engl.: regression test).*
(IEEE 610.12 1992, S. 61, Definition von regression testing)

Weitere für diese Arbeit wichtige Begriffe sind Komponententest und Akzeptanztest. Beide Arten können als Regressionstests dienen.

Definition 2.4 (Komponententest)
Ein Komponententest *(engl.: unit tests) ist ein Test, der Fehler durch isoliertes Testen einer individuellen Komponente mit einen Testfall finden soll.*
(Brügge u. Dutoit 2004, S. 449)

In einem objektorientierten System handelt es sich bei den getesteten Komponenten um Klassen. Als Faustregel gilt: Zu jeder Klasse gehört eine Testklasse.

Definition 2.5 (Akzeptanztest)
Ein Akzeptanztest *ist ein Test, der überprüft, ob das getestete Programm eine gegebene Anforderung erfüllt.*

Akzeptanztests beschreiben z. B. Anwendungsfälle (Stephens u. Rosenberg 2003, 242).

errors."

Um die Testfälle einfach, automatisch und häufig ausführen zu können, wird in JCommSy die Test-Software JUnit eingesetzt. Das einfache, automatische Ausführen von Testfällen ist z. B. für die Absicherung von Refactorings wichtig.

2.6 Refactoring

In Abschnitt 2.1 haben wir gesehen, dass es wichtig ist, die Qualität eines Systems hoch zu halten. Häufig verschlechtert sich der Entwurf und damit die Qualität eines Systems während es wächst. Fowler verwendet die Metapher, dass der Quelltext anfängt, »übel zu riechen«; er schreibt, der Code bekomme *smells*. Um solchen Verschlechterungen entgegen zu wirken, bietet es sich an, die Technik des *Refactoring* zu verwenden. (Fowler 2000)

Definition 2.6 (Refactoring)
Ein Refactoring *ist eine Änderung an der internen Struktur einer Software, um sie [die Struktur] leichter verständlich zu machen und einfacher zu verändern, ohne ihr [der Software] beobachtbares Verhalten zu ändern.*
(Fowler 2000, S. 41)

Bei dieser Definition zeigt sich wieder, dass es bei Qualität nicht nur um funktionale Eigenschaften geht. Wenn die Vorbedingungen eines Refactorings eingehalten werden, »zerbricht« es das Programm nicht (Opdyke 1992, S. 2). Es gibt für eine Reihe von Refactorings kleinschrittige Beschreibungen des Vorgehens (engl.: *mechanics*). Für einen Teil dieser Refactorings bieten moderne Entwicklungsumgebungen – wie etwa Eclipse (Daum 2006) oder NetBeans (Boudreau u. a. 2002) – Unterstützung und automatische Ausführung an.

Voraussetzung für Refactoring sind solide Tests (Fowler 2000, S. 83 ff). Hierbei handelt es sich natürlich um Regressionstests (Definition 2.3). Für die von Fowler beschriebenen kleinen Refactorings sind dies in erster Linie Komponententests (Definition 2.4).

Große Refactorings

Die einzelnen Refactorings setzen auf unterschiedlichen Granularitätsstufen an. Es gibt Refactorings, die auf Quelltext-Ebene im Kleinen wirken. Andere betreffen die Architektur eines Systems. Die Probleme, die sie lösen,

werden in *code smells* und *architecture smells* unterschieden (Roock u. Lippert 2004, S. 33). Gegen *architecture smells* helfen die *Großen Refactorings*, deren Definition wir von Roock u. Lippert übernehmen.

Definition 2.7 (Großes Refactoring)
Ein Großes Refactoring *hat folgende Eigenschaften:*

Dauer *Große Refactorings dauern länger als einen Tag.*

Team *Große Refactorings betreffen das ganze Projektteam.*

Unsicherheit *Große Refactorings können nicht vollständig durch elementare (sichere) Refactorings realisiert werden. Es sind zusätzliche (unsichere) Refactorings notwendig. (sic)*

(Roock u. Lippert 2004, S. 84)

Weil Große Refactorings nicht nur ein Modul sondern mehrere verändern, betreffen sie auf der Test-Seite nicht nur Komponenten- sondern auch Akzeptanztest. Um ein Großes Refactoring abzusichern, sind üblicherweise Akzeptanztests als Regressionstests nötig.

Definition 2.8 (Sicheres/Unsicheres Refactoring)
Ein sicheres Refactoring *kennzeichnet solche Refactorings, die durchgeführt werden können, ohne dabei die Gefahr einzugehen, das Verhalten des Systems zu verändern oder einen Fehler zu machen.*

Unter einem unsicheren Refactoring *verstehen wir solche Refactorings, für die keine schrittweise und überprüfte Anleitung zur Durchführung existiert. (sic)*
(Roock u. Lippert 2004, S. 85)

Große Refactorings betreffen nicht nur die Implementierung eines Moduls, sondern ändern auch Schnittstellen. Schnittstellen werden auch durch Prozessabstraktion gebildet.

2.7 Prozessabstraktion

Das Konzept *Operation* (bei Sebesta 2008 auch: *Unterprogramm*) wurde für *Prozessabstraktion* entwickelt. Das heißt: Die Definition einer Operation gibt einer Reihe von Anweisungen einen Namen, mit dem man sie ausführen kann. So wird von dem durch die Anweisungsreihe beschriebenen Prozess abstrahiert. Für den Aufruf der Operation muss nur der Name bekannt

sein (d. h. *was* sie tut), nicht jedoch die Implementierung (*wie* sie es tut). Ein anderer Name für Prozessabstraktion ist *prozedurale Abstraktion*. (Abelson u. Sussmann 1998, S. 26)

Die Schnittstelle einer Operation ist ihre *Signatur*. Um diesen Begriff formal definieren zu können, benötigen wir den Begriff des *Parameter-Profils*.

Definition 2.9 (Parameter-Profil)
Das Parameter-Profil *einer Operation besteht aus Anzahl, Reihenfolge und Typen ihrer Parameter. (Vgl. Sebesta 2008, S. 386)*

Angelehnt an Sebesta (2008, S. 386) sowie Gosling u. a. (1997, S. 153 f) definieren wir:

Definition 2.10 (Signatur)
Die Signatur *einer Operation besteht aus ihrem Namen, ihrem Parameter-Profil und den Namen der Parameter, ihrem Rückgabetyp, ihren Vor- und Nachbedingungen und den (geprüften) Ausnahmen, die bei ihrer Ausführung auftreten können.*

Für uns ist nur die objektorientierte Ausprägung des Konzeptes *Operation* wichtig, die *Methode*. In einer Klasse dürfen nicht zwei Methoden mit der gleichen Signatur definiert sein. Daher auch die Verwendung der Metapher »Signatur«, die für eine Operation wie die Unterschrift für einen Menschen einmalig ist.

2.8 Webanwendungen

Das *world wide web* basiert auf einer Anbieter-Nutzer-Architektur (engl. *client server architecture*). Während die ersten Webseiten statisch waren, hat sich inzwischen eine Vielzahl von dynamischen Anwendungen entwickelt – die *Webbasierten Anwendungen*. Webbasierte Anwendungen werden abhängig von ihrem Ausführungsort in zwei Gruppen aufgeteilt: in client-seitige und server-seitige Anwendungen (Turau 1999). Server-seitige Anwendungen werden auch als *Webanwendungen* oder *Webapplikationen* (engl. *web application*) bezeichnet. Wie wir später sehen werden, ist CommSy in die zweite Gruppe einzuteilen. Einen Server, auf dem eine Webanwendung ausgeführt wird, nennen wir *Anwendungsserver*.

Webanwendungen basieren auf dem Protokoll »Hypertext Transport Protocol«, kurz: HTTP (RFC 2616 1999) und sind geprägt vom *Anfrage-Antwort-*

Kontrollflussmodell. Dies rührt daher, dass ihre Oberfläche nicht wie die herkömmlicher Anwendungen durch eine reaktive grafische Benutzerschnittstelle (GUI) realisiert ist, sondern durch Hypertextseiten, die von einem speziellen Programm (dem Browser) interpretiert werden. Diese Seiten sind statisch; eine Kommunikation mit der eigentlichen Anwendung, die auf dem Server abläuft, findet nur statt, wenn der Benutzer eine neue *HTTP-Anfrage* (engl. *HTTP request*) nach einer neuen Seite stellt. Eine Anfrage wird typischerweise durch Auswählen einer Verknüpfung (engl. *link*) oder Drücken eines Knopfes gestellt.

Als Reaktion auf solche Anfrage sendet der Anwendungsserver eine *HTTP-Antwort* (engl. *HTTP response*), die die gewünschte Seite enthält.

Das Protokoll HTTP ist zustandslos. Viele Anwendungsserver bieten einen Mechanismus an, trotzdem Daten zwischen zwei Anfragen zu speichern: die *HTTP-Sitzung* (engl.: *HTTP session*). Wenn es im Zusammenhang unmissverständlich ist, sprechen wir auch kurz von *Anfrage*, *Antwort* und *Sitzung*.

2.8.1 Anfrageparameter und Operationsparameter

Eine Anfrage kann Parameter enthalten; um Verwechslungen mit den Parametern von Operationen zu vermeiden, sprechen wir in der Arbeit von *Anfrageparametern* bzw. *Operationsparametern* . Wird kurz von einem Parameter geschrieben, so ist damit immer ein Operationsparameter gemeint.

Anfrageparameter sind grundsätzlich Zeichenketten. Daraus folgt, dass Zahlenwerte zur Übertragung in einer Anfrage in Zeichenketten umgewandelt werden müssen. Zur Auswertung muss dann ein *parsing* der Zeichenketten durchgeführt werden. Auch Werte anderer Datentypen müssen in Zeichenkettenrepräsentation transportiert werden. Mit anderen Worten: Anfrageparameter sind ungetypt, ein Nachteil gegenüber den Operationsparametern einer getypten Programmiersprache wie Java.

2.8.2 Java EE

»Java Platform, Enterprise Edition« (kurz: Java EE oder JEE)[5] ist eine Spezifikation für eine Sammlung von Technologien für Server-seitige Anwendungen, die in der Programmiersprache Java geschrieben werden. Für Webanwendungen sind zwei Technologien aus dieser Spezifikation besonders relevant,

[5] früher »Java 2 Platform, Enterprise Edition« (kurz: J2EE)

die *Servlets* und die *java server pages* (JSP). Ein *Servlet* ist eine in Java programmierte Klasse, die HTTP-Anfragen bearbeitet und HTTP-Antworten darauf konstruiert und sendet (Jendrock u. a. 2006, S. 7). Eine JSP ist eine in der verwirrenderweise ebenfalls JSP genannten Skriptsprache definierte Datei. Die Sprache JSP kann als Erweiterung von HTML begriffen werden, die es ermöglicht, in einer HTML-Seite Java-Quelltext zu verwenden. (Turau u. a. 2004, S. 19 ff)

Für Servlets und JSPs (die zur Ausführung in Servlets übersetzt werden) muss eine spezielle Laufzeitumgebung zur Verfügung gestellt werden: der *container* (Turau u. a. 2004, S. 2). Die Referenzimplementierung für die Spezifikation für *servlet-container* ist Tomcat (Tomcat-Webseite 2008).

Java EE stellt für die Implementierung von Servlets das *Servlet-Rahmenwerk* zur Verfügung. In diesem ist eine Reihe von Klassen für die Erstellung von Servlets definiert. Zentral sind die Klassen HttpServlet, HttpServletRequest und HttpServletResponse sowie HttpSession. Um ein eigenes Servlet zu implementieren, definiert man eine Unterklasse von HttpServlet. Die Klassen HttpServletRequest und HttpServletResponse sind die JEE-Realisierungen der Konzepte HTTP-Anfrage und -Antwort. Die Klasse HttpSession implementiert die HTTP-Sitzung.

2.8.3 Webanwendungen und Qualität

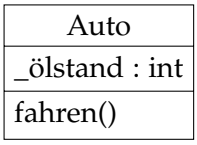
Eine hohe Qualität zu haben, ist für jede Art von Software wichtig. Nach Offutt (2002) ist hohe Qualität für Webanwendungen noch wichtiger als für herkömmliche Software, weil es für einen Anwender sehr leicht ist, zur Konkurrenz zu wechseln. (Sie ist ja »nur einen Klick entfernt«.)

Es gibt Autoren, die fordern, dem Entwickeln von Webanwendungen eine eigene Disziplin zu widmen, das *web engineering* (Floyd u. Oberquelle 2007; Kappel u. a. 2004; Dumke u. a. 2003).

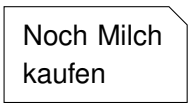
2.9 UML

In der vorliegenden Arbeit befinden sich Diagramme, um die Architektur und ihre Umbauten anschaulich zu machen. Wir geben hier eine kurze Legende der darin verwendeten Symbole und Pfeile. Wir verwenden grundsätzlich die *Unified Modeling Language* (UML) nach Booch u. a. (1999) mit einigen Anpassungen.

Um eine *Klasse* darzustellen, verwenden wir ein einfaches Rechteck, wenn keine Attribute und Operationen dargestellt werden sollen.



Wenn *Attribute* und *Operationen* sichtbar sein sollen, unterteilen wir das Rechteck. Wenn keine Zugriffsbeschränkungen mit +, # oder - angegeben werden, bedeutet dies: Operationen sind grundsätzlich öffentlich; Attribute sind grundsätzlich privat.
Abstrakte Klassen und Operationen werden kursiv gesetzt.



Notizen werden als Rechtecke mit Eselsohr oben rechts dargestellt.

Architekturelemente werden als Rechtecke mit runden Ecken dargestellt.

Die *Benutzungsbeziehung* stellen wir als durchgezogene Linie mit gefüllter Pfeilspitze dar.

Die *Ableitungsbeziehung* als durchgezogene Linie mit hohler Pfeilspitze.

Die *Implementierungsbeziehung* als gestrichelte Linie mit hohler Pfeilspitze.

Die *Kompositionsbeziehung* als durchgezogene Linie mit Raute auf der Seite des Kompositums.

Wenn wir ein *Refactoring* grafisch beschreiben, dann deuten wir dies mit dem Transformationspfeil an. Die Teile, die sich ändern, zeichnen wir dann in fetter Schrift aus.

«taglib»

Stereotypen sind in französischen Anführungszeichen dargestellt.

2.10 Voraussetzung, Vorbedingung, Abhängigkeit

Unser Ziel ist, die Kopplung an das Servlet-Rahmenwerk zu lockern. Um dies in einem konkreten System zu erreichen, muss zunächst die Frage beantwortet werden, ob es überhaupt möglich ist, die Kopplung zu lockern. Genauer gefragt bedeutet dies: Worin besteht die Kopplung des Systems an das Rahmenwerk? und: Ist diese Kopplung nötig oder nicht? Denn

wenn die Kopplung nötig ist, kann sie prinzipiell nicht gelöst werden. Um diese Fragen zu beantworten, führen wir in diesem Kapitel den Begriff der Voraussetzung ein, die entweder essentiell oder akzidentiell sein kann. Um eine Voraussetzung als akzidentiell zu erkennen, muss zunächst die Voraussetzung selbst erkannt werden; deshalb unterscheiden wir zwischen stillschweigenden und ausdrücklichen Voraussetzungen.

Die hier beschriebenen Begriffe sind bis auf den Begriff der *Vorbedingung* bisher so nicht in der Literatur beschrieben und wurden deshalb neu entwickelt.

Die *Voraussetzungen* einer Operation sind die Eigenschaften, die ihre Umwelt haben muss, damit die Operation fehlerfrei ausgeführt werden (und ihre Nachbedingungen garantieren) kann. Eine Voraussetzung kann sein: ein Modul, das existieren und bestimmte Dienstleistungen anbieten muss (genannt *Abhängigkeit*) oder eine logische Bedingung, die erfüllt sein muss (genannt *Vorbedingung*).

Merkmale von Software mit hoher Qualität sind, dass die Voraussetzungen ihrer Operationen ausdrücklich gemacht sind und dass sie keine (oder) wenige akzidentielle Voraussetzungen haben. Um zu erkennen, dass eine Voraussetzung akzidentiell ist, ist es sinnvoll, sie ausdrücklich zu machen. Das Entfernen von akzidentiellen Voraussetzungen hat, wie sich später zeigen wird, Einflüsse auf die Architektur eines Systems.

Weil wir die Begriffe *Voraussetzung*, *Abhängigkeit* und *Vorbedingung* und die Unterscheidungen *essentiell/akzidentiell* und *ausdrücklich/stillschweigend* im Laufe der Arbeit häufig benötigen, definieren wir sie im Folgenden exakt.

2.10.1 Abhängigkeit

Zunächst definieren wir den Begriff ganz allgemein für Module und Operationen.

Definition 2.11 (Abhängigkeit – allgemein)
Ein Modul X heißt abhängig *von einem Modul Y, wenn in X Elemente von Y selbst verwendet werden.*

Eine Operation foo() heißt abhängig *von einem Modul Y, wenn in foo() Elemente von Y verwendet werden.*

Y heißt dann auch Abhängigkeit *von X bzw. foo().*

Beispiel 2.1
In der Operation *bar()* wird eine Variable eines Typs, der im Modul *B* definiert ist, deklariert. Dann ist *bar()* von *B* abhängig. ✍

Beispiel 2.2
Im Modul *A* wird eine im Modul *B* definierte Operation aufgerufen. Dann ist *A* von *B* abhängig. ✍

Da unser Betrachtungsgegenstand JCommSy ein objektorientiertes System ist, definieren wir Abhängigkeit auch für Klasse und Methode. Der Begriff *Klasse* umfasst in dem hier verwendeten Sinne auch die in Java verwendeten Schnittstellen (engl. *interfaces*).

Definition 2.12 (Abhängigkeit – objektorientiert)
Eine Klasse *X heißt* abhängig *von einer Klasse Y, wenn gilt, X benutzt Y oder X erbt von Y.*

Eine Methode *m() heißt* abhängig *von einer Klasse Y, wenn die Definition von m() eine Variable (oder einen Parameter) vom Typ Y enthält oder verwendet.*

Y heißt dann auch Abhängigkeit von X bzw. m().

Beispiel 2.3
Die Klasse in Abbildung 2.1 B erbt von A und benutzt C. Daraus folgt: B hat

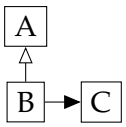


Abbildung 2.1: Beispiel 2.3: B ist von A und C abhängig

die Abhängigkeiten A und C. ✍

In einem UML-Klassendiagramm sind die Abhängigkeiten leicht an den Pfeilen zu erkennen. Eine Klasse ist abhängig von allen Klassen, zu denen Pfeile führen, gleich, ob es sich um Benutzungs- oder Vererbungspfeile handelt.

Man könnte eine Unterscheidung zwischen direkter und indirekter (durch Transitivität) Abhängigkeit einführen, sie ist allerdings für die Arbeit ohne Belang.

2.10.2 Vorbedingungen

Die zweite Art von Voraussetzung ist die Vorbedingung.

Definition 2.13 (Vorbedingung)
Eine logische Bedingung bed heißt Vorbedingung *einer Operation operation(), wenn bed erfüllt sein muss, damit operation() erfolgreich ausgeführt werden kann.*

Vorbedingungen können durch *Zusicherungen* ausgedrückt werden. Dies wird bei der Verwendung des Vertragsmodells (Meyer 1997, S. 331 ff) getan.

Zusicherungen wurden ursprünglich eingeführt, um die Korrektheit von Flussdiagrammen (Floyd 1967) und später für prozedurale Programme (mit dem Hoare-Kalkül, Hoare 1969) zu zeigen. Im Vertragsmodell werden sie verwendet, um zu überprüfen, ob in einem Programm zur Laufzeit zu einem gegeben Zeitpunkt eine bestimmte Bedingung erfüllt ist. Bertrand Meyer definiert:

Definition 2.14 (Zusicherung)
Eine Zusicherung *ist ein Ausdruck, der Entitäten des Programmes enthält und eine Eigenschaft beschreibt, die diese Entitäten in einem bestimmten Zustand der Programmausführung erfüllen sollen. (Meyer 1997, S. 336)*

Syntaktisch sind Zusicherungen Boolesche Ausdrücke. In Java werden sie mit Hilfe des Schlüsselwortes assert ausgedrückt.

Da Java das Vertragsmodell nicht unterstützt, wird in JCommSy ein typisches Hilfskonstrukt verwendet. Um zu beschreiben, dass eine Methode *operation()* die Vorbedingung ausdruck hat, wird dort geschrieben:

```
/**
 * @require ausdruck
 */
public void operation() {
    assert ausdruck : "Vorbedingung verletzt: ausdruck"
    // ...
}
```

So werden beide Aspekte der Vorbedingung programmiert: Beschreibung an der Schnittstelle und Überprüfung zu Laufzeit.

Um die vorliegende Arbeit nicht unnötig aufzublähen, lassen wir in Quelltext-Beispielen die assert-Anweisung weg und schreiben knapper:

```
/**
 * @require ausdruck
 */
public void operation() {
    // ...
}
```

In den Quelltextbeispielen in der Arbeit wird aber trotzdem davon ausgegangen, dass die Überprüfung auch zur Laufzeit stattfindet.

Neben Vorbedingungen werden in der vorliegenden Arbeit auch Nachbedingungen formuliert. Um Aussagen über den Rückgabewert einer Methode in Nachbedingungen machen zu können, verwenden wir den Ausdruck $result.

2.10.3 Voraussetzungen

Wir definieren exakt:

Definition 2.15 (Voraussetzung)
Die Voraussetzungen *einer Operation sind ihre Abhängigkeiten und ihre Vorbedingungen.*

2.10.4 Essentielle und akzidentielle Voraussetzungen

Die Voraussetzungen lassen sich danach einteilen, ob sie nötig sind oder nicht.

Definition 2.16 (Essentiell/Akzidentiell)
Eine Voraussetzung, die eine Operation hat, heißt essentiell, *wenn sie sie benötigt wird, um ihre Aufgabe zu erfüllen und sonst* akzidentiell.

Beispiel 2.4
Eine akzidentielle Abhängigkeit finden wir im folgenden Code:

```
public class A {
    public foo(B b, C c) {
        c.bar();
    }
}
```

Die Methode *foo()* ist akzidentiell abhängig von B. ✍

Ziel bei Entwurf und Implementierung von Operationen ist, dass sie möglichst wenige akzidentielle Voraussetzungen haben.

Welche Voraussetzungen ein Modul »benötigt«, ist nicht immer eindeutig; manchmal es hängt von der Situation und dem Betrachter ab. Innerhalb des Teams, das ein Softwaresystem entwickelt, sollte Einigkeit darüber bestehen, welche Voraussetzungen essentiell sind.

Beispiel 2.5
In dem Quelltext

```
public class X {
    public foo(Y y) {
        Z z = y.getZ();
        z.bar();
    }
}
```

ist nicht ohne Zusammenhang klar, ob *foo()* essentiell oder akzidentiell von Y abhängig ist. ✍

Einerseits wird y in *foo()* nicht selbst verwendet, sondern nur ein Z aus ihm ausgelesen. So könnte die Abhängigkeit der Methode *foo()* von Y als akzidentiell angesehen werden. Andererseits könnte es gerade die Aufgabe von *foo()* sein, aus einem Y ein Z auszulesen und dann mit diesem etwas zu tun. Insofern müsste die Abhängigkeit der Methode *foo()* von Y als essentiell betrachtet werden.

Wenn eine Methode einen Parameter hat, den sie nicht benötigt, sprechen wir auch von einem *akzidentiellen Parameter*. Akzidentielle Parameter sind relevant für überschriebene Methoden. Solche werden wir in Kapitel 5 betrachten.

2.10.5 Ausdrückliche und stillschweigende Voraussetzungen

Die Voraussetzungen können aufgeteilt werden in *ausdrückliche Voraussetzungen* und *stillschweigende Voraussetzungen*.

Definition 2.17 (Ausdrücklich/Stillschweigend)
Eine Voraussetzung heißt ausdrücklich, *wenn Sie im Programmcode direkt ausgedrückt wird und zur Übersetzungszeit bekannt ist, andernfalls* stillschweigend.

Beispiel 2.6 (Stillschweigende Voraussetzung)
Die Methode *foo()* in folgendem Code hat eine stillschweigende Voraussetzung.

```
public foo(Y y) {
    y.bar();
}
```

Die Variable y muss ein Objekt referenzieren; denn sonst kann an ihr die Methode *bar()* nicht aufgerufen werden. ✍

Beispiel 2.7 (Ausdrückliche Voraussetzung)
Wie Beispiel 2.6 mit ausdrücklicher Voraussetzung:

```
/** @require y != null */
public foo(Y y) {
    y.bar();
}
```

Die Voraussetzung von *foo()* wird nun als Vorbedingung ausgedrückt. ✍

Abhängigkeiten sind grundsätzlich ausdrücklich. Allerdings können Vorbedingungen unter bestimmten Umständen in Abhängigkeiten umgewandelt werden (dazu mehr in Abschnitt 3.3).

Stillschweigende Voraussetzungen entstehen z. B. durch Annahmen, die über ein verwendetes Rahmenwerk gemacht werden oder das Kontrollflussmodell, dass ein verwendetes Rahmenwerk festlegt.[6] Fehler, die durch stillschweigende Voraussetzung entstehen, führen erst zur Laufzeit zu Fehlern.

Ausdrückliche Voraussetzungen können maschinell gefunden werden, d. h. vom Übersetzer oder von Werkzeugen wie dem Duftoskop (Kubosch 2007) oder dem Sotographen (Bischofberger u. a. 2004). Stillschweigende Voraussetzungen sind schwieriger zu finden, sie können nur von Menschen verstanden werden.

Gute Programmiersprachen vereinfachen es dem Entwickler, Voraussetzungen ausdrücklich zu machen, und ermöglichen es ihm so, einen Teil der Fehlerfindung auf den Übersetzer abzuwälzen. Beispiele für programmiersprachliche Mittel für die Unterstützung des Ausdrücklichmachens von Voraussetzungen sind starke Typisierung und das Vertragsmodell.

2.11 Verwandte Arbeiten

In der von uns bearbeiteten Literatur wird der Zusammenhang von Kopplung und Software-Wiederverwendung betrachtet. Das Thema Refactoring

[6] Zum Beispiel wird durch die verwendete Oberflächentechnologie das Kontrollflussmodell festgelegt. Eine GUI-Oberfläche ist interaktiv und reaktiv, während bei einer Web-Oberfläche das Anfrage-Antwort-Kontrollflussmodell verwendet wird.

wird ebenfalls beleuchtet, auch in Hinblick auf die Verwendung von Rahmenwerken. Wir betrachten im Anschluss die Literatur genauer und werden sehen, dass sich dort keine Lösung für unser spezielles Problem der Entflechtung von Anwendung und Rahmenwerk findet. Es scheint, dass bisher nicht nichts zu diesem Thema veröffentlicht wurde.

2.11.1 Zu Wiederverwendung und Kopplung

Im Softwarehaus sd&m wurde *Quasar* entwickelt, ein System um Software mit hoher architektureller Qualität zu entwickeln. (Quasar = kurz für Qualitätssoftwarearchitektur)

> Quasar versucht, besonders wichtige Regeln und Mechanismen der Softwaretechnik verständlich zu beschreiben, zu präzisieren und zum Standard zu erklären. (Siedersleben 2004)

In Quasar wird u. A. der Begriff der Softwarekategorie (oder anschaulicher Software-Blutgruppe) verwendet. Softwarekategorien sind »Wissensgebiete, Themen, die in ein Stück Software einfließen« (Siedersleben 2004, S. 73). Die Softwarekategorie (oder die Softwarekategorien), zu der ein Stück Software gehört, gibt an, welches Wissen es enthält und wovon es abhängt (Siedersleben 2004, S. 77).

Siedersleben schlägt eine Unterteilung in (zumindest) die Kategorien 0, A, T und AT (analog zu den Blutgruppen 0, A, B, AB) vor. Software der Kategorie A befasst sich nur mit der Anwendung (also der Fachlichkeit). Software der Kategorie T ist technisch und kapselt Wissen über technische Vorgänge. Die Kategorie 0 beschreibt Software, die von allen Architekturelementen eines Systems verwendet wird. Zur Kategorie 0 gehören bspw. bei der Programmiersprache Java die Pakete `java.lang` und `java.util`. Software soll nicht zur Kategorie AT gehören, weil in ihr zwei Aspekte vermischt werden. Diese Regel ähnelt unser Architekturregel zur Verwendung von Rahmenwerken (Seite 9). Quasar präsentiert allerdings keine Lösung, wie man Software, die zur Kategorie AT gehört, in Module aufteilt, die entweder in Kategorie A oder Kategorie T gehört.

2.11.2 Zu Refactoring

Nachdem Opdyke (1992) Refactoring eingeführt und Fowler (2000) es katalogisiert hatte, haben viele weitere Autoren sich des Themas angenommen.

Roock u. Lippert (2004) haben Große Refactorings beschrieben. Kerievsky (2004) hat untersucht, mit welchen Refactorings man Entwurfsmuster einführen kann. Entwurfsmuster haben ganz allgemein lose Kopplung zum Ziel (Gamma u. a. 1996, S. 34 f). Wie man Rahmenwerke mit Refactoring ändern kann, dass sie generische Typen anbieten, zeigen Kieżun u. a. (2007). Was beachtet werden muss, wenn man Software mit Refactoring verändert, die das Vertragsmodell einsetzt, untersuchen Goldstein u. a. (2006). Balaban u. a. (2005) bieten einen Ansatz zur Unterstützung für das Refactoring von Software, die ein sich änderndes Rahmenwerk verwendet.

Keine dieser Arbeiten beantwortet jedoch die Frage, mit welchen Refactorings man Software und verwendetes Rahmenwerk entflechten kann, wenn diese zu eng gekoppelt sind.

2.11.3 Fazit

In der von uns bearbeiteten Literatur wird erkannt, dass zu enge Kopplung eines Programms an ein Rahmenwerk Probleme verursacht und sie deshalb von Anfang der Entwicklung des Programms an vermieden werden sollte. Es wurden Refactorings für ganz verschiedene Problemfelder publiziert. Anscheinend ist jedoch bis jetzt kein Ansatz zur nachträglichen Lockerung der Kopplung an ein Rahmenwerk veröffentlicht worden. Insbesondere wurden scheinbar bisher keine Refactorings gegen Rahmenwerkskopplung veröffentlicht.

2.12 Zusammenfassung

In diesem Kapitel haben wir die in der vorliegenden Arbeit verwendeten Begriffe aus der Literatur eingeführt.

Die Qualität von Software hängt von funktionalen und nicht-funktionalen Eigenschaften ab. Eine wichtige nicht-funktionale Eigenschaft einer Software ist ihr Entwurf. Entwurfsprinzipien, die zu hoher Qualität führen, sind das Geheimnisprinzip, lose Kopplung und hohe Kohäsion. Diese Prinzipien gelten auch für die Architektur einer Software und ihre Bestandteile, die Architekturelemente. Für wiederverwendete Software, d. h. Rahmenwerke stellten wir die Architekturregel zur Verwendung von Rahmenwerken auf. Diese Regel besagt, dass ein Rahmenwerk nicht von mehr als einem Architekturelement angemessener Granularität benutzt werden soll. Eine

Architekturelementart angemessener Granularität kann z. B. die Schicht sein.

Um hohe Qualiät eines Programms sicherzustellen ist es wichig, es zu testen. Wir unterschieden zwischen Komponententests, die ein einzelnes Modul isoliert testen und Akzeptanztests, die ein Systems als Ganzes testen. Regressionstests sind Tests, die dazu dienen, sicherzustellen, dass eine Anwendung auch nach Änderungen noch die gleiche Funktionalität bietet. Solche Änderungen können z. B. Refactorings sein, d. h. Änderungen, die die interne Struktur einer Anwendung verbesseren, ohne ihr äußeres Verhalten zu ändern. Ein Sonderfall von Refactorings sind die sogenannten Großen Refactorings, die der Verbesserung der Architektur dienen. Mit solchen Architekturverbesserungen kann die Änderung der Signatur von Methoden einhergehen, deshalb definierten wir die Begriffe Signatur und Parameter-Profil noch einmal genau.

Eine besondere Art der Anwendung ist die Webanwendung, die auf einem Server ausgeführt, aber deren Oberfläche in einem Browser auf einem Client angezeigt wird. Webanwendungen basieren auf dem Anfrage-Antwort-Kontrollflussmodell; die Anfragen können Parameter enthalten. Um diese Art Parameter und die Parameter von Methoden unterscheiden zu können, verwenden wir die Begriffe Anfrageparameter und Operationsparameter.

Zum Verständnis der Diagramme der vorliegenden Arbeit haben wir die verwendete, leicht angepasste UML-Notation erläutert.

Um die Arten von Kopplung und inwieweit sie notwendig ist, exakt untersuchen zu können, haben wir den Begriff der Voraussetzung und ihre Unterbegriffe Vorbedingung und Abhängigkeit definiert. Wir unterschieden Voraussetzungen danach, ob sie essentiell oder akzidentiell und ausdrücklich oder stillschweigend sind.

Zuletzt betrachteten wir verwandte Arbeiten. In der von uns betrachteten Literatur wird Rahmenwerkskopplung als problemverursachend beschrieben und Refactorings zur Lösung verschiedener Probleme beschrieben. Bis jetzt sind diese beiden Themen aber anscheinend noch nicht zusammengeführt worden. Wie es scheint, gibt es daher bis jetzt keinen Lösungsansatz für unser Problem.

3 Problemfeld A: Kopplung an das Servlet-Rahmenwerk

In diesem Kapitel beschäftigen wir uns mit der Kopplung an das Servlet-Rahmenwerk und wie man sie lösen kann. Um zu erläutern, warum die Kopplung möglichst lose sein soll, werden wir zunächst untersuchen, welche Probleme es aufwirft, Anwendungssoftware zu eng an das Servlet-Rahmenwerk zu koppeln. Dazu betrachten wir die typische Verwendung des Servlet-Rahmenwerks. Wir werden zeigen, dass eine enge Kopplung nicht nötig ist, da nur wenige essentielle Abhängigkeiten zum Servlet-Rahmenwerk bestehen. Als zweites präsentieren wir einen Ansatz, mit dem man die Verflechtung einer Anwendungssoftware mit dem Servlet-Rahmenwerk entwirren kann. Diesen Ansatz wollen wir dann in Kapitel 4 am Beispielsystem JCommSy anwenden.

3.1 Typische Verwendung des Servlet-Rahmenwerks

Wenn das Prinzip der Trennung von Interaktion und Funktion auf die Architektur einer Webanwendung (Abschnitt 2.8) mit Servlets (Abschnitt 2.8.2) angewandt wird, kann sie schematisch ganz grob wie in Abbildung 3.1 dargestellt werden. Die Servlets, die ja die Oberfläche der Webanwendung

Abbildung 3.1: Trennung von Präsentation und Funktion – allgemein und Servlet -Anwendung (Legende in Abschnitt 2.9)

realisieren, bilden ein Architekturelement (Abschnitt 2.3) gleichen Namens, das auf ein Architekturelement »Fachlogik« zugreift. Üblicherweise benutzt

dieses wiederum ein Architekturelement »Datenbank«; das ist für unsere Betrachtung jedoch zunächst ohne Belang.

Wendet man die Architekturregel zur Verwendung von Rahmenwerken (Seite 9) auf die Architektur aus Abbildung 3.1 an, so folgt daraus, dass das Servlet-Rahmenwerk nur von dem Architekturelement »Servlets« verwendet werden darf.

Die typische Verwendung des Servlet-Rahmenwerk (wie z. B. beschrieben in Wöhr 2004, S. 317 ff und Jendrock u. a. 2006, S. 99) ist folgende: Während in Java eine herkömmliche Anwendung durch Definition einer Methode *main()* erzeugt wird, werden Webanwendungen anders erzeugt. Um eine Webanwendung zu implementieren, programmiert man ein sogenanntes Servlet. Dazu wird eine Klasse von HttpServlet abgeleitet. Diese Klasse überschreibt eine (oder beide) der in der Oberklasse definierten Methoden *doGet()* und *doPost()* (die die beiden unterschiedlichen HTTP-Anfrageweisen GET und POST implementieren, Abschnitt 2.8 und RFC 2616 1999). Diese beiden Methoden haben das gleiche Parameter-Profil (Definition 2.9): sie erhalten einen Parameter vom Typ HttpServletRequest und einen vom Typ HttpServletResponse. Wird die entsprechende Web-Adresse aufgerufen, so startet der Anwendungsserver die Webanwendung, indem er eine dieser Methoden aufruft.

Die Methoden *doGet()* und *doPost()* müssen das gegebene Parameter-Profil haben; denn es wird vom Rahmenwerk vorgegeben. Oft werden Servlet-Anwendungen aber so entwickelt, dass *doGet()* und *doPost()* ihre Parameter direkt weitergeben. Wenn diese Weitergabe an andere Servlets geschieht, ist das kein Problem. Wenn die Parameter allerdings an die Fachlogik weitergegeben werden, dann wird auch sie an das Servlet-Rahmenwerk gekoppelt, weil auch sie dann die Typen HttpServletRequest und HttpServletResponse verwendet. Dies widerspricht der Architekturregel zur Verwendung von Rahmenwerken.

Zur Erinnerung: Kopplung ist die Zahl der Abhängigkeiten zwischen zwei Modulen (Abschnitt 2.2, Brügge u. Dutoit 2004, S. 230). Zwischen Klassen können nach Definition 2.12 zwei Arten von Abhängigkeiten bestehen: Vererbung und Benutzung. Vererbungsabhängigkeit ist ausgeprägt als Ableitung von einer Klasse oder Implementieren einer Schnittstelle. Benutzungsabhängigkeit ist ausgeprägt als Feld, lokale Variable oder Parameter einer Klasse oder Methode.

Einen Fall von Vererbungsabhängigkeit zwingt uns das Servlet-Rahmenwerk auf: jede Klasse, die ein Servlet implementiert, muss von HttpServlet

abgeleitet sein. Diese Abhängigkeit ist essentiell, weil die Aufgabe eines Servlet immer auch beinhaltet, ein Servlet zu sein.

Benutzungsabhängigkeit finden wir in den Methoden, die HttpServletRequest oder HttpServletResponse als Parameter haben. In Methoden in einer Servlet-Klasse kann diese Abhängigkeit essentiell sein. Im fachlichen Teil einer Anwendung ist Abhängigkeit von diesen Typen immer akzidentiell; denn diese Typen sind nur nötig, um die Benutzungsoberfläche der Anwendung zu realisieren.

3.2 Probleme durch Verwenden von HttpServletRequest und HttpServletResponse

Wir betrachten ein (vereinfachtes aber typisches) Beispiel einer fachlichen Methode, die HttpServletRequest verwendet. Wir verwenden ab hier das Vertragsmodell für Java in der für die Universität Hamburg typischen Weise, wie in Abschnitt 2.10.2 beschrieben.

Beispiel 3.1
Die Methode *createNewUser()* erzeugt ein Objekt vom Typ User; es handelt sich also um eine Fabrikmethode (Gamma u. a. 1996, S.131 ff).

```
/**
 * @require request != null
 */
public  User createNewUser(HttpServletRequest request) {
     String  name = request.getParameter("Username");
     return  new User(name);
}
```

Wir nehmen an, dass *createNewUser()* in der Klasse UserCreator definiert (Abbildung 3.2)ist. User sei eine nicht näher spezifizierte fachliche Klasse

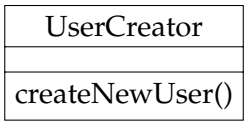


Abbildung 3.2: Beispiel 3.1: Die Klasse UserCreator

aus dem Architekturelement »Fachlogik«. ✍

Diese Methode verletzt die Architekturregel zur Verwendung von Rahmenwerken. An ihr können wir nun die daraus folgende Probleme erkennen:

- Technologiefestlegung (3.2.1)
- Wissen an der falschen Stelle (3.2.2)
- Schlechte Testbarkeit (3.2.3)
- Unklare Schnittstelle (3.2.4)
- Verlust der Typsicherheit (3.2.5)
- Zu breite Schnittstelle (3.2.6)
- Schreibzugriffe auf Parametern möglich (3.2.7)
- Globale Variablen (3.2.8)

Wir schauen uns genauer an, was die Probleme ausmacht und warum es Probleme sind.

3.2.1 Technologiefestlegung

Weil fachlicher und technischer Quelltext in der Methode *createNewUser()* aus Beispiel 3.1 gemischt werden, ist die Anwendung, zu der die Methode gehört, auf die Oberflächentechnologie festgelegt.

Unsere Methode *createNewUser()* aus Beispiel 3.1 verwendet die Klasse HttpServletRequest, die Teil des Servlet-Rahmenwerks ist. Sie verwendet sie jedoch nicht nur intern, sondern auch als Operationsparameter (Abschnitt 2.8.1), d. h. sie zeigt die Verwendung auch nach außen hin an, und zwingt so ihre Benutzer, sich an das Rahmenwerk zu koppeln. Die Methode *createNewUser()* hat HttpServletRequest als Abhängigkeit. Sie benötigt sie jedoch nicht um ihre Aufgabe zu erfüllen; also ist die Abhängigkeit zu HttpServletRequest akzidentiell (Definition 2.16).

Die Anwendung kann so nur mit dem Servlet-Rahmenwerk und nicht mit einer anderen Oberflächentechnologie verwendet werden, also z. B. nicht mit einer Rich-Client-Technologie, wie Swing (Horstmann u. Cornell 2005b, S. 439 ff) oder SWT (Daum 2006, S. 165 ff), aber auch noch nicht einmal mit einer anderen Web-Oberflächentechnologie wie Struts (Cekvenich u. Gehner 2004), Spring (Wolff 2007) oder JSF (Turau u. a. 2004, S. 180 ff). Wenn die verwendete Oberflächentechnologie geändert werden soll, müssen mehrere

Architekturelemente (hier neben dem Architekturelement »Servlets« auch das Architekturelement »Fachlogik«) angepasst werden. Technologie ändert sich meist in einem andern Rhythmus als Fachlichkeit, deswegen sollten Technologie und Fachlichkeit in unterschiedlichen Architekturelementen untergebracht werden (Siedersleben 2004, S. 90 f). Es kann also erforderlich sein, die gleiche Fachlogik mit einer neuen Oberflächentechnologie zu bedienen; das ist mit der aktuellen Architektur nicht möglich. Ziel muss es deshalb sein, dass der Austausch der Oberflächentechnologie durch ein einfaches Ersetzen eines Architekturelementes (hier des Architekturelements »Servlets«) durch ein anderes Architekturelement erfolgen kann.

3.2.2 Wissen an der falschen Stelle

Die Technologiefestlegung verursacht ein weiteres Problem: Die Methode *createNewUser()* aus Beispiel 3.1 muss wissen, unter welchem Namen der gewünschte Anfrageparameter (Abschnitt 2.8.1) in dem HttpServletRequest-Objekt gespeichert ist, hier unter dem Namen »Username«. Wenn sich der Name ändert oder ein Tippfehler einschleicht, kann dies nicht vom Übersetzer erkannt werden. Es gibt dann erst zur Laufzeit Rückkopplung über den Fehler. Umgekehrt formuliert kann man sagen, dass die Methode *createNewUser()* die stillschweigende Voraussetzung hat, dass unter dem Namen »Username« der gewünschte Anfrageparameter verfügbar ist.

Welchen Namen eine Anfrageparameter hat, ist eine Entwurfsentscheidung, die nur in einem Modul einer Webanwendung bekannt sein sollte, nämlich dem Architekturelement, das die Oberfläche (hier die Servlet-Oberfläche) realisiert. Die Methode *createNewUser()* kennt diese Entwurfsentscheidung allerdings auch; sie hat damit eine akzidentielle Voraussetzung.

3.2.3 Schlechte Testbarkeit

Das Testen unserer Beispielmethode ist unnötig schwierig. Eine Testmethode für *createNewUser()* aus Beispiel 3.1 bei Verwendung der Testsoftware JUnit könnte so aussehen, wie in dem folgenden Quelltext. Wir gehen hier davon aus, dass dem Leser die Verwendung von JUnit geläufig ist, sonst siehe Link (2005).

```
public void testCreateNewUser() {
    HttpServletRequest request;
    UserCreator c;
    //  ...
    request.setParameter("Username", "Hans");
    User u = c.createNewUser(request);
    assertEquals("Hans", u.getName());
}
```

Folgendes läuft in diesem Test ab: Zunächst wird an dem Objekt, auf das die Referenzvariable request verweist, der Anfrageparameter mit dem Namen »Username« auf die Belegung »Hans« gesetzt, dann wird mit unserer Methode *createNewUser()* aus Beispiel 3.1 ein neues User-Objekt erzeugt. Dazu wird ihr die Variable mit dem Namen request als Operationsparameter (Abschnitt 2.8.1) übergeben. Schließlich wird der eigentliche Test durchgeführt, nämlich die Überprüfung, ob der Name des neu erzeugten Benutzers wirklich »Hans« ist.

Der Test ist verständlich und scheint sinnvoll zu sein. Aber woher kommt das Objekt, auf das die Variable request verweist? Um die Methode *createNewUser()* zu testen, muss sie mit einem Operationsparameter des Typs HttpServletRequest aufgerufen werden. Exemplare vom Typ HttpServletRequest sind allerdings nicht leicht zu erzeugen, weil HttpServletRequest keine Klasse, sondern eine Schnittstelle (engl. *interface*) ist. HttpServletRequest ist im Servlet-Rahmenwerk enthalten; das Rahmenwerk bietet allerdings keine standardmäßige Implementierung an, auf die man von außerhalb des Rahmenwerkes zugreifen kann. Eine Anweisung wie

```
HttpServletRequest request = new HttpServletRequest()
```

führt deshalb zu einem Übersetzungsfehler.

Wir benötigen also eine eigene Implementierung von HttpServletRequest von der wir ein für den Test verwendbares Exemplar erzeugen können. Eine Möglichkeit wäre es, eine Mock-Implementierung (Mackinnon u. a. 2001) von HttpServletRequest selbst zu schreiben. Allerdings deklariert die Schnittstelle HttpServletRequest 25 Methoden, die alle implementiert werden müssten. Eine weitere Möglichkeit wäre die Verwendung einer Mock-Software wie EasyMock (EasyMock-Webseite 2008; Link 2005).

Das hier beschriebene Phänomen tritt häufig auf, weil fast alle Klassen, die das Servlet-Rahmenwerk verwenden, die Schnittstellen HttpServletRequest oder HttpServletResponse benutzen. Deshalb wird von der Apache Software Foundation das Test-Programm »Cactus« für diesen Zweck angeboten. Es handelt sich dabei um eine Erweiterung von JUnit zum Testen von Server-seitigem Code (Cactus-Webseite 2008; Mock-vs-Container 2008).

Die Testklasse wird bei der Verwendung von Cactus nicht wie sonst bei JUnit, von der Klasse TestCase aus dem Paket junit.framework abgeleitet, sondern von der Klasse ServletTestCase aus dem Paket org.apache.cactus. Diese von Cactus angebotene Klasse erbt selbst wiederum von TestCase. ServletTestCase enthält eine Exemplarvariable request mit geschützter (engl. *protected*) Sichtbarkeit, die vom Typ HttpServletRequestWrapper (ebenfalls aus org.apache.cactus) ist. Dieser Typ ist von HttpServletRequest abgeleitet und bietet genau die Funktionalität, die wir in unser Testmethode brauchen. Damit die Cactus-Tests erfolgreich laufen können, ist es erforderlich, dass ein Anwendungsserver (wie Tomcat, Tomcat-Webseite 2008) die zu testenden Klassen zur Verfügung stellt.

Ähnlich wie *createNewUser()* haben auch *createFragment()* und andere Methoden aus der Klasse FragmentOutfitter (Abbildung 4.10 auf Seite HttpServletRequest) Parameter vom Typ HttpServletRequest. Zusätzlich haben die meisten von ihnen einen Parameter vom Typ HttpServletResponse, für den das Geschriebene analog gilt.

3.2.4 Unklare Schnittstelle

Es ist nicht klar, welche essentiellen Voraussetzungen (d. h. Abhängigkeiten und Zusicherungen, Abschnitt 2.10.3) unsere Methode *createNewUser()* aus Beispiel 3.1 hat. Nur eine Voraussetzung ist ausdrücklich (Definition 2.17) nämlich die Abhängigkeit von HttpServletRequest, die Methode hat zusätzlich stillschweigende (Definition 2.17) Voraussetzungen.

Wir betrachten unsere Beispielmethode noch einmal genau. Ihre Signatur (Definition 2.10) ist

```
/**
 * @require request != null
 */
public  User createNewUser(HttpServletRequest request)
```

Diese Signatur erweckt den Eindruck, als würde die Methode das gesamte übergebene Objekt vom Typ HttpServletRequest benötigen; die Signatur gibt keinen Hinweis darauf, welche Daten die Methode konkret verwendet. Wenn wir uns nun aber die Implementierung der Methode anschauen:

```
public  User createNewUser(HttpServletRequest request) {
    String  name = request.getParameter("Username");
    return  new User(name);
}
```

Dann sehen wir, dass das, was sie eigentlich benötigt, um ihre Aufgabe (nämlich ein Objekt vom Typ User zu erzeugen) zu erfüllen (also ihre essentielle Abhängigkeit), ein Objekt vom Typ String ist. Die Abhängigkeit von HttpServletRequest ist akzidentiell.

Um dem Aufrufer einer Operation zu kommunizieren, welche Daten von ihm erwartet werden, gibt es u. A. den Mechanismus des *Operationsparameters* (Abschnitt 2.8.1 und Sebesta 2008, S. 387 ff). Dadurch, dass *createNewUser()* aber als Operationsparameter ein HttpServletRequest-Objekt übergeben wird, wird dieser Mechanismus ausgehebelt; denn die Klasse HttpServletRequest bietet über ihre Methode *getParameter()* die Möglichkeit, auf eine Fülle von Objekten – die Anfrageparameter – zuzugreifen. Somit hat *createNewUser()* zwar nur einen Operationsparameter, aber Zugriff auf sehr viele Daten.

Ein Benutzer von *createNewUser()* kann sich, wenn er nur ihre Signatur kennt, nicht erschließen, welche Daten die Methode benötigt, d. h. welche essentiellen Abhängigkeiten sie hat. Das heißt, dass er sie nicht bestimmungsgemäß verwenden kann, wenn er nicht zusätzlich zu ihrer Signatur ihre Implementierung kennt! Genau das läuft aber der Idee der Operation an sich zuwider, die ja gerade ist, von Implementierungsdetails zu abstrahieren und sie hinter einer Schnittstelle (ihrer Signatur) zu verstecken. (vgl. Abschnitt 2.7)

Die Methode *createNewUser()* hat also eine stillschweigende Voraussetzung: Unter dem Namen »Username« muss ein Anfrageparameter aus dem HttpServletRequest-Objekt abgerufen werden können. Diese Voraussetzung kann als Zusicherung, aber auch als Operationsparameter ausdrücklich gemacht werden (Abschnitt 3.3).

3.2.5 Verlust von Typsicherheit

Mit der unklaren Schnittstelle und der Verwendung von HttpServletRequest als Parameter-Typ hängt ein weiteres Problem zusammen, das in Abschnitt 2.8.1 angedeutet wurde. Die Ursache für dieses Problem ist, dass Anfrageparameter immer als Zeichenketten übertragen werden. In unser Methode *createNewUser()* aus Beispiel 3.1 wird folgender Code aufgerufen:

```
String name = request.getParameter("Username");
```

Das funktioniert gut, weil die Variable name eine Zeichenkette referenziert. Was geschieht aber, wenn wir beispielsweise eine ganze Zahl benötigen?

Die Methode *getParameter()* liefert ein Objekt vom Typ String zurück. Dies liegt darin begründet, dass die Anfrageparameter als Zeichenketten übergeben werden müssen, weil HTTP (Abschnitt 2.8) nur Zeichenketten überträgt. Wenn Zahlen transportiert werden müssen, dann geschieht dies in ihrer Zeichenketten-Repräsentation. Auf der Empfangsseite muss dann ein *parsing* durchgeführt werden, so wie in folgendem Beispielquelltext:

```
String yearString = request.getParameter("Year_of_birth");
int year = Integer.parseInt(yearString);
```

Das kann allerdings zu Laufzeitfehlern führen, die immer dann auftreten, wenn der entsprechende Anfrageparameter keine Zahl enthält. Es wäre also wünschenswert, korrekt getypte Operationsparameter zu übergeben.

Typenlose Sitzungsattribute

Ein weiteres Problem tritt zutage, wenn statt auf Anfrageparameter auf Attribute der HTTP-Sitzung (Abschnitt 2.8) zugegriffen wird. Wir definieren eine zweite Methode in Beispiel 3.2.

Beispiel 3.2
Die Methode *createNewUser2()* liest aus der HTTP-Sitzung.

```
public User createNewUser2(HttpServletRequest request) {
    HttpSession session = request.getSession();
    String name = (String) session.getAttribute("Username");
    return new User(name);
}
```

In *createNewUser2()* aus aus Beispiel 3.2 muss eine Typumwandlung zur Laufzeit (engl. *cast*) vorgenommen werden. Diese kann ebenfalls zu einem Laufzeitfehler führen. Es könnte z. B. durch einen Programmierfehler ein Objekt unter dem Namen »Username« abgespeichert werden, das einen anderen Typ als String hat, wie z. B. in:

```
Date d = new Date();
session.setAttribute("Username", date);
```

Wird zur Laufzeit *createNewUser2()* aus Beispiel 3.2 nach diesem Code aufgerufen, wird eine ClassCastException geworfen.

Es zeigt sich, dass *createNewUser2()* aus Beispiel 3.2 zwei stillschweigende Voraussetzungen hat, die ebenfalls nicht an der Signatur sichtbar sind. Erstens muss unter dem Namen »Username« ein Sitzungsattribut aus dem HttpServletRequest-Objekt abgerufen werden können. Voraussetzung dafür ist, dass dieses Sitzungsattribut existieren muss. Zweitens muss das Sitzungsattribut mit dem Namen »Username« vom Typ String sein.

Diese Voraussetzungen können ebenfalls durch Vorbedingungen bzw. als Parameter ausdrücklich gemacht werden. Dazu mehr in Abschnitt 3.3.

3.2.6 Zu breite Schnittstelle

Wie oben gezeigt, hat die Methode *createNewUser()* aus Beispiel 3.1 eine akzidentielle Abhängigkeit. Sie bekommt nicht den Parameter übergeben, den sie benötigt, um ihre Arbeit zu machen, d. h. einen String, sondern einen HttpServletRequest. Dies ist nicht nur problematisch, weil die Signatur von *createNewUser()* so unklar ist, sondern auch, weil so im Körper von *createNewUser()* auf mehr Daten zugegriffen werden kann als nötig. Dies fördert das Programmieren von Fehlern. Außerdem erleichtert die zu breite Schnittstelle das Brechen des Gesetzes der Demeter (Lieberherr u. Holland 1989; Basili u. a. 1996).

Solch breite Schnittstelle erhöht die Kopplung und ist ein Verstoß gegen das Geheimnisprinzip (Abschnitt 2.2); sie wird von der Literatur allgemein abgelehnt. (Ludewig u. Lichter 2007, S. 388; Brügge u. Dutoit 2004, S. 229f; Zuser u. a. 2001, S. 124)

3.2.7 Schreibzugriffe auf Parameter möglich

Ein besonderer Nachteil der zu breiten Schnittstelle ist, dass das übergebene Objekt vom Typ HttpServletRequest an seiner Schnittstelle nicht nur sondierende, sondern auch zustandsverändernde Operationen anbietet. So z. B. *setAttribute()*.

Java bietet leider kein Sprachmittel, um den Aufruf von zustandsverändernden Operationen an übergebenen Objekten zu verbieten; in C++ steht hierfür das Schlüsselwort const zur Verfügung (Stroustrup 1998, S. 155 ff).

Die Implementierung von *createNewUser()* aus Beispiel 3.2 könnte folgendermaßen geändert werden, ohne dass aufrufender Quelltext angepasst werden müsste.

```
public  User createNewUser(HttpServletRequest request) {
    request.setAttribute("URL", null);

    String  name = (String) request.getParameter("Username");
    return  new User(name);
}
```

Der Zustand des Objektes, das von request referenziert wird, wird nun verändert, und das eventuell ohne dass die Entwickler von aufrufendem Quelltext Nachricht darüber erhalten. Das ist nicht das Verhalten, das man von einer Methode namens *createNewUser()* erwarten würde und hat den unerwünschten Seiteneffekt, dass das Objekt, das von der Variable request referenziert wird, durch den Aufruf von *createNewUser()* verändert wird.

Die Methode *createNewUser()* braucht nur lesenden Zugriff auf ihre Parameter und keinen schreibenden. Deshalb sollte ihre Signatur entsprechend entworfen sein.

Es gibt natürlich auch Methoden, die schreibenden Zugriff auf Ihre Parameter benötigen. Wenn innerhalb der Fachlogik in Exemplare von Typen aus dem Servlet-Rahmenwerk geschrieben wird, so ist auch das problematisch. In diesem Fall besteht eine Abhängigkeit zu den entsprechenden Typen. Diese Abhängigkeiten sind akzidentiell, weil die Fachlogik, um ihre Aufgabe zu erfüllen, keine Abhängigkeiten zur Oberflächentechnologie benötigt.

3.2.8 Globale Variablen

Webanwendungen sind durch das Anfrage-Antwort-Kontrollflussmodell geprägt. Als Abstraktion einer HTTP-Anfrage (Abschnitt 2.8) existiert in einer Webanwendung üblicherweise genau ein Exemplar einer Klasse. Bei Anwendungen, die das Servlet-Rahmenwerk einsetzen, ist dies die Klasse HttpServletRequest. Weil bei einem Lauf des Programms üblicherweise nur eine Anfrage bearbeitet wird, existiert nur ein Exemplar von HttpServletRequest. Wird dies Exemplar an viele Methoden weitergereicht, so wirkt es wie eine Art Halde für globale Variablen.

Wenn die Anfrageparameter und Sitzungsattribute innerhalb eines Durchlaufes zusätzlich noch geändert werden (Abschnitt 3.2.7), geht leicht die Übersicht verloren.

3.2.9 Bewertung

Wir haben die Methode *createNewUser()* aus Beispiel 3.1 stellvertretend für alle Methoden betrachtet, die Parameter von Typen aus dem Servlet-Rahmenwerk haben, aber Teil der Fachlogik sind. Die Aussagen, die über *createNewUser()* bezüglich Rahmenwerkskopplung gemacht werden, gelten auch für die anderen Methoden. Die Methode *createNewUser()* ist von HttpServletRequest abhängig. Diese Abhängigkeit (Definition 2.12) ist akzidentiell (Definition 2.16). Die essentiellen (Definition 2.16) Voraussetzungen (Definition 2.15) von *createNewUser()* sind nicht ausdrücklich (Definition 2.17) sondern stillschweigend (Definition 2.17). Weiterhin hat *createNewUser()* stillschweigende akzidentielle Voraussetzungen. In diesem Abschnitt haben wir eine Reihe von Problemen gesehen, die gelöst oder verbessert werden können, indem die stillschweigenden Voraussetzungen ausdrücklich gemacht und die akzidentiellen Voraussetzungen beseitigt werden. Wie das geht, untersuchen wir in Abschnitt 3.3.

Zunächst wollen wir jedoch die Übertragbarkeit auf andere Rahmenwerke betrachten. Wir haben bis jetzt die Probleme betrachtet, die durch Kopplung an das Servlet-Rahmenwerk entstehen. Nun wollen wir schauen, ob diese Probleme auch bei Kopplung an andere Rahmenwerken auftreten. Dazu gehen wir auf jedes Problem einzeln ein:

Technologiefestlegung Dieser Punkt ist auf alle Rahmenwerke übertragbar, die eine Technologie repräsentieren. Immer wenn ein ein Rahmenwerk vom einem Architekturelement benutzt wird, wird dieses Element

auf das Rahmenwerk und die von ihm implementierte Technologie festgelegt.

Wissen an der falschen Stelle Das oben beschriebene Problem, dass die Fachlogik die Namen der Anfrageparameter kennen muss, ist typisch für das Servlet-Rahmenwerk. Es gibt aber auch andere Rahmenwerke, die Klassen mit einer Methode haben, die den Namen oder Schlüssel eines Attributes, Parameters oder ähnlichem erwartet. So eine Methode hat eine Signatur nach dem Muster:

```
public  Object get(Object schluessel)
```

Anwendungen, die diese Rahmenwerke verwenden, können die Probleme haben, die durch Wissen an der falschen Stelle ausgelöst werden. Ein Beispiel ist die Klasse Map aus dem *Java Collection Framework*, mit ihrer Methode *get()* (Horstmann u. Cornell 2005a, S. 149 ff).

Schlechte Testbarkeit Der Grund für die hier dargestellte schlechte Testbarkeit ist, dass die getestete Methode ein Objekt eines Typs erwartet, der nicht leicht zu instantiieren ist – hier HttpServletRequest. Ähnliche Probleme haben deswegen auch Anwendungen, die zu eng an andere Rahmenwerke gekoppelt sind, die solche schwer instantiierbbaren Typen haben. Beispiel für so ein Rahmenwerk ist EJB (Ihns u. a. 2007). Schwer instantiierbare Typen sind z. B. jene, die nur Konstruktoren mit vielen Parametern haben oder jene, die nur als Schnittstelle ohne Standardimplementierung angeboten werden.

Unklare Schnittstelle Das konkrete Problem ist typisch für das Servlet-Rahmenwerk, aber Rahmenwerke mit Methoden nach dem Muster aus »Wissen an der falschen Stelle« können die gleichen prinzipiellen Probleme haben.

Verlust der Typsicherheit Dieses Problem gilt allgemein für Methoden nach dem Muster aus »Wissen an der falschen Stelle«. Ein Beispiel ist wieder das *Java Collection Framework*; mit der Einführung von Typparametern in Java 5 (Horstmann u. Cornell 2005b, S. 885 ff) wurde das Problem in diesem Rahmenwerk allerdings entschärft.

Zu breite Schnittstelle Die zu breite Schnittstelle ist ein Problem, das nicht nur durch Rahmenwerkskopplung ausgelöst wird. Es tritt immer dann

auf, wenn eine Operation akzidentielle Parameter hat. Rahmenwerke wie das Servlet-Rahmenwerk, die Typen anbieten, die eine große Zahl von Daten kapseln, laden dazu ein, Operationen mit zu breiter Schnittstelle zu definieren. Fowler (2000, S. 74 f) nennt solche Typen »Datenklumpen«.

Schreibzugriffe auf Parametern möglich Dieses Problem tritt immer dann auf, wenn Operationen Parameter von Typen haben, die zustandsverändernde Operationen anbieten. Eine Maßnahme dagegen ist es, immer Kopien zu übergeben; wenn an diesen etwas geändert wird, so können dadurch keine unerwünschten Seiteneffekte ausgelöst werden. Eine andere Maßnahme besteht darin, zu einem veränderlichen Typ einen unveränderlichen (engl. *immutable*) Typen mit gleicher Schnittstelle anzubieten, und wenn möglich unveränderliche Exemplare zu übergeben. Dieser zweite Ansatz wird vom *Java Collection Framework* verfolgt. Dessen Klasse Collections bietet z. B. die Methode *unmodifiableCollection()* für diesen Zweck an (Horstmann u. Cornell 2005a, S. 165 ff).

Globale Variablen Dieses Problem tritt auf, wenn Rahmenwerke Typen definieren, von denen üblicherweise zur Laufzeit nur ein Exemplar existiert.

3.3 Lösungsansatz: Akzidentielle Abhängigkeiten beseitigen

Wenn wir die Kopplung einer Methode an das Servlet-Rahmenwerk lösen wollen, dann müssen wir alle Abhängigkeiten zu Typen aus dem Servlet-Rahmenwerk lösen. Dies ist nur möglich, wenn diese Abhängigkeiten akzidentiell sind; andernfalls kann die Kopplung nicht gelöst werden. Damit Voraussetzungen einer Methode als akzidentiell erkannt werden können, ist es sinnvoll, zunächst alle Voraussetzungen ausdrücklich zu machen. Denn erst, wenn die Voraussetzungen einer Methode bekannt sind, können sie daraufhin untersucht werden, ob sie essentiell oder akzidentiell sind.

Wir untersuchen deshalb zunächst, wie stillschweigende Voraussetzungen zu ausdrücklichen gemacht werden können. Dazu müssen sie im Quelltext als Vorbedingungen formuliert werden. Dann untersuchen wir, wie wir

akzidentielle Voraussetzungen beseitigen können. Dazu werden Vorbedingungen in Abhängigkeiten umgewandelt.

Betrachten wir noch einmal unsere Beispielmethode *createNewUser2()* aus Beispiel 3.2, die über die HTTP-Anfrage auf die HTTP-Sitzung zugreift:

```
/**
 * @require request != null
 */
public User createNewUser2(HttpServletRequest request) {
    HttpSession session = request.getSession();
    String name = (String) session.getAttribute("Username");
    return new User(name);
}
```

Wie wir uns erinnern, hat diese Methode zwei stillschweigende Voraussetzungen. Erstens: Unter dem Namen »Username« muss ein Sitzungsattribut aus dem HttpServletRequest-Objekt geholt werden können. Zweitens: Das Sitzungsattribut mit dem Namen »Username« muss vom Typ String sein.

3.3.1 Voraussetzungen ausdrücklich machen

Um die bisher stillschweigenden Voraussetzungen ausdrücklich zu machen, bietet sich die intensive Verwendung des *Vertragsmodells* (engl. *design by contract*, Abschnitt 2.10.2 und Meyer 1997, S. 331 ff) an. Dazu erweitern wir die Signatur von *createNewUser2()* um Vorbedingungen, die die beiden Voraussetzungen ausdrücken:

```
/**
 * @require request != null
 * @require request.getSession()
 *      .getAttribute("Username") != null
 * @require request.getSession()
 *      .getAttribute("Username") instanceof String
 */
public User createNewUser2(HttpServletRequest request)
```

Dieser Schritt ist verwandt mit dem Refactoring »Zusicherung einführen« (Fowler 2000, S. 273 ff). Er geht über dieses hinaus, weil er auch die Schnittstelle der geänderten Methode betrifft. Nach unser Definition 2.10 sind die

Vorbedingungen Teil der Signatur. Wir untersuchen, ob sich die Probleme gebessert haben:

Technologiefestlegung Keine Besserung. Nach wie vor wird HttpServletRequest als Parametertyp verwendet. Die Methode *createNewUser2()* ist nach wie vor nur mit dem Servlet-Rahmenwerk zu verwenden.

Wissen an der falschen Stelle Keine Besserung. Die Methode *createNewUser2()* hat immer noch Wissen über die Namen der Sitzungsattribute.

Schlechte Testbarkeit Keine Besserung. Weil *createNewUser2()* einen Parameter vom Typ HttpServletRequest hat (und außerdem abhängig vom Typ HttpSession ist), kann sie nur in einem Servlet-Container getestet werden.

Unklare Schnittstelle Leichte Besserung. Jetzt haben wir die Voraussetzungen der Methode an ihrer Schnittstelle sichtbar gemacht. Die Signatur ist also für einen menschlichen Leser klarer. Es können aber immer noch Laufzeitfehler auftreten, weil der Übersetzer die Voraussetzungen nicht kennt. Die Laufzeitfehler, die jetzt auftreten können, sind allerdings klarer. Sie zeigen direkt an, dass eine Vorbedingung verletzt wurde.

Leider unterstützt Java nicht von Haus aus das Vertragsmodell; aber Vor- und Nachbedingungen werden auch in Sprachen mit Vertragsmodell erst zur Laufzeit geprüft. Die hier gemachten Aussagen gelten also auch analog z. B. für Eiffel (Meyer 1992).

Verlust der Typsicherheit Die Typsicherheit ist etwas verbessert, weil jetzt an der Schnittstelle der benötigte Typ (String) bekanntgemacht wird. Das Werfen einer ClassCastException wird verhindert. Dafür wird die Vorbedingung erst zur Laufzeit geprüft, und es kann dann ein AssertionError geworfen werden.

Auch nach diesem Schritt des Ausdrücklichmachens von Voraussetzungen können also immer noch Laufzeitfehler auftreten.

Zu breite Schnittstelle Keine Besserung. Über den Parameter request kann immer noch auf mehr Daten als nötig zugegriffen werden.

Schreibzugriffe auf Parametern möglich Keine Besserung.

Globale Variablen Keine Besserung.

Die Methode *createNewUser2()* aus Beispiel 3.2 greift nur lesend auf das ihr als Parameter übergebene Objekt vom Typ HttpServletRequest zu. Wenn die Methode, die vom Servlet-Rahmenwerk entflochten werden soll, auch schreibend auf Objekte von Typen aus dem Servlet-Rahmenwerk zugreift, dann sollten in diesem ersten Schritt entsprechende Nachbedingungen formuliert werden. Wir geben ein kurzes Beispiel.

Beispiel 3.3
Die Methode *saveUser()*.

```
/**
 * @require req != null
 * @require user != null
 */
public void saveUsername(HttpServletRequest req, User user) {
    HttpSession session = req.getSession();
    session.setAttribute ("username", user.getName());
}
```

Die Methode speichert einen ihr übergebene User in die ihr übergebene HTTP-Anfrage. ✍

Die Methode *saveUser()* aus Beispiel 3.3 könnte mit ausdrücklichen Nachbedingungen so verbessert werden:

```
/**
 * @require request != null
 * @require user != null
 * @ensure req.getSession()
 *      .getAttribute("user") != null
 * @ensure req.getSession()
 *      .getAttribute("user") instanceof User
 */
public void saveUsername(HttpServletRequest req, User user)
```

Die Nachbedingungen drücken nun die Veränderung der Sitzung aus, die der Aufruf der Methode *saveUser()* an ihr vornimmt.

3.3.2 Vorbedingungen in Abhängigkeiten umwandeln

Als zweiten Schritt werden wir die ausdrücklich gemachten Vorbedingungen nun in Abhängigkeiten umwandeln. Dadurch werden Fehler früher erkannt; während sie vorher erst zur Laufzeit klar wurden, können sie jetzt schon zur Übersetzungszeit erkannt werden. Dazu muss die Signatur (Definition 2.10) der Methode *createNewUser2()* aus Beispiel 3.2 geändert und entsprechend ihre Implementierung angepasst werden. Statt als Vorbedingungen formulieren wir nun die essentiellen Voraussetzungen als Abhängigkeit (konkret in Form eines Parameters). Dabei lösen wir auch die akzidentielle Abhängigkeit zu HttpServletRequest.

```
/**
 * @require name != null
 */
public  User createNewUser2(String name)
    return  new User(name);
}
```

Nun haben wir eine Version von *createNewUser2()*, die einen Übersetzerfehler verursacht, wenn sie aufgerufen wird, ohne dass ihr ein String-Parameter übergeben wird. Der von *createNewUser2()* benötigte Anfrageparameter muss nun an der Stelle des Systems ausgelesen werden, an der *createNewUser2()* aufegrufen wird. Wir untersuchen wieder, ob sich die Probleme aus Abschnitt 3.2 gebessert haben:

Technologiefestlegung Die Methode *createNewUser2()* verwendet nun keine Typen des Servlet-Rahmenwerks mehr. Sie hat nur noch einen Parameter vom Typ String. Also kann sie technologieunabhängig verwendet werden, z. B. in einer Anwendung mit einer Swing-Oberfläche.

Wissen an der falschen Stelle Da die Methode *createNewUser2()* jetzt nicht mehr die Klasse HttpSession verwendet, muss sie auch kein Wissen mehr über die Namen der Sitzungsattribute haben. Sollte sich der Name eines Sitzungsattributes ändern, so bleibt *createNewUser2()* mit dieser Implementierung davon unberührt.

Schlechte Testbarkeit Die neue Implementierung von *createNewUser2()* ist leichter zu testen. Jetzt können wir folgenden Test definieren:

```
public void testCreateNewUser2() {
    User u = c.createNewUser("Hans");
    assertEquals("Hans", u.getName());
}
```

Diese Implementierung ist viel einfacher als die in Abschnitt 3.2.3.

Außerdem kann dieser Test in der normalen JUnit-Umgebung ablaufen und braucht nicht die spezielle schwergewichtige Unterstützung von Cactus.

Unklare Schnittstelle Die Signatur von *createNewUser2()* sagt nun deutlich aus, welche Daten die Methode benötigt, d. h. ihre essentiellen Voraussetzungen sind ausdrücklich und an der Signatur sichtbar. Dies kann sowohl ein menschlicher Leser verstehen als auch ein Programm, als insbesondere der Übersetzer erkennen.

Verlust der Typsicherheit Die Signatur von *createNewUser2()* zeigt nun, welche Typen die ihr übergebenen Daten haben müssen. Auch dies kann sowohl ein menschlicher Leser verstehen als auch ein Programm erkennen.

Zu breite Schnittstelle Die Signatur von *createNewUser2()* hat nun genau die richtige Breite. Das liegt daran, dass jetzt die essentiellen Voraussetzungen die ausdrücklichen sind und umgekehrt. Die Methode kann nun nicht mehr durch Unfälle oder Programmierfehler auf eigentlich unbenötigte (nur akzidentiell verfügbare) Daten zugreifen.

Schreibzugriffe auf Parametern möglich Der Methode *createNewUser2()* werden in der neuen Implementierung keine Parameter mehr übergeben, deren Zustand sie ändern könnte. Also kann sie auch keine Schreibzugriffe auf solchen mehr ausführen.

Globale Variablen Diese Implementierungsvariante gibt *createNewUser2()* Zugriff nur noch auf ein String-Objekt und nicht mehr auf ein HttpServletRequest-Objekt. Die Methode greift dadurch nicht mehr auf dasselbe Objekt zu, auf das auch alle anderen Methoden zugreifen (unter der Annahme, dass die Zeichenkette mit dem Benutzernamen von weniger Methoden benötigt wird). Wenn auch die anderen Methoden

entsprechend umgebaut werden, dann kann das HttpServletRequest-Objekt nicht mehr wie eine Halde für globale Variablen wirken.

Die Methode *createNewUser2()* aus Beispiel 3.2 greift nur lesend auf das ihr als Parameter übergebene Objekt vom Typ HttpServletRequest zu; deshalb konnte ihre Vorbedingung in einen Parameter umgewandelt werden. Wenn die vom Servlet-Rahmenwerk zu entflechtende Methode auch schreibend zugreift und nur ein Datum am HttpServletRequest-Objekt setzt, dann kann dieses Datum als Rückgabewert zurückgegeben werden.

Wenn mehrere Felder am HttpServletRequest-Objekt gesetzt werden, dann kann ein Rückgabe-Parameterobjekt eingeführt werden, in dem die unterschiedlichen Daten gespeichert werden. So wie beim Lesen sichergestellt werden muss, dass die Parameter für die Methode nun an einer anderen Stelle im System aus der HTTP-Anfrage ausgelesen werden, muss beim Schreiben sichergestellt werden, dass nach dem Aufruf der umgestellten Methode der oder die Rückgabedaten in die HTTP-Anfrage geschrieben werden.

3.3.3 Verwendung eines Parameterobjektes

Eine weitere Möglichkeit zum Lösen der Abhängigkeiten aus Kapitel 3 ist das Ersetzen des HttpServletRequest-Parameters durch ein speziell für diesen Zweck eingeführtes *Parameterobjekt* . Ein Parameterobjekt repräsentiert eine Gruppe von Parametern, die häufig gemeinsam übergeben werden (Fowler 2000, S. 303). Ein Beispiel für eine Parameterobjekt-Klasse kann man in Abbildung 3.3 betrachten. Diese Klasse bietet über die Methode *getUserName()*

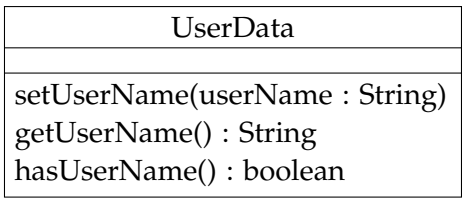


Abbildung 3.3: Die Klasse UserData

Zugriff auf den Benutzernamen, der mit Aufruf der Methode *setUserName()* gesetzt wird. Die Methode *hasUserName()* ist eine Hilfsmethode, um Vor- und Nachbedingungen formulieren zu können. Die Klasse könnte wie in dem folgenden Quelltext implementiert sein.

```
public class UserData {

    private String _userName;

    /**
     * @require userName != null
     * @ensure hasUserName()
     */
    public void setUserName(String userName) {
        _userName = userName;
    }

    /**
     * @require hasUserName()
     * @ensure $result != null
     */
    public String getUserName() {
        return _userName;
    }

    public boolean hasUserName() {
        return _userName != null;
    }

}
```

Dieses Beispiel ist unrealistisch klein. Eine Parameterobjekt-Klasse wird üblicherweise nur dann eingeführt, wenn nicht wie hier nur ein Datum, sondern viele Datenelemente gekapselt werden sollen.

Die Methode *createNewUser2()* wird für die Verwendung von UserData entsprechend angepasst:

```
/**
 * @require data != null
 * @require data.hasUserName()
 */
public User createNewUser2(UserData data) {
    return new User(data.getUserName());
}
```

Bei dieser dritten Implementierung der Methode *createNewUser2()* bekommt sie als Parameter kein HttpServletRequest-Objekt übergeben, sondern ein Objekt vom Typ UserData. Wir untersuchen wieder, welchen Einfluss diese Implementierung auf die Probleme hat:

Technologiefestlegung Auch bei dieser Lösung ist die Methode *createNewUser2()* technologieneutral, weil sie keine Typen aus dem Servlet-Rahmenwerk verwendet.

Wissen an der falschen Stelle Die Methode *createNewUser2()* muss so kein Wissen über die Namen von Anfrageparametern (noch darüber, dass es überhaupt Anfrageparameter gibt) haben.

Schlechte Testbarkeit Diese Implementierung von *createNewUser2()* ist leichter zu testen als die aus 3.2.3. Wir definieren folgenden Test:

```
public void testCreateNewUser2() {
    UserData data = new UserDate();
    data.setUsername("Hans");
    User u = c.createNewUser(data);
    assertEquals("Hans", u.getName());
}
```

Dieser Test kann in der normalen JUnit-Umgebung ohne die Unterstützung von Cactus ablaufen.

Der Test ist nicht ganz so einfach wie der aus Abschnitt 3.3.2; denn er muss ein Objekt vom Typ UserData erzeugen und konfigurieren.

Unklare Schnittstelle Die Schnittstelle von *createNewUser2()* ist so klarer als in Abschnitt 3.2.3. Durch die Vorbedingungen ist klar ausgedrückt, welche Daten im Parameterobjekt zur Verfügung stehen müssen. Allerdings besteht auch Zugriff auf die möglichen anderen Daten des Parameterobjektes UserParam, deshalb kann man die Abhängigkeit von *createNewUser2()* zu UserData als akzidentiell ansehen und deshalb ist die Schnittstelle weniger klar als in Abschnitt 3.3.2.

Verlust der Typsicherheit Die Parameterobjekt-Klasse bietet für jedes von der Methode benötigte Datum eine eigene Zugriffsmethode, die als Rückgabetyp den Typ des jeweiligen Datums hat. (In unserem Beispiel

benötigt *createNewUser2()* nur ein Datum und die Parameterobjekt-Klasse kapselt auch nur eines. Erst wenn die entsprechende Methode viele Daten benötigt, macht das Parameterobjekt richtig Sinn.) So kann Typsicherheit in dieser Methode garantiert werden, da sie keinen *cast* mehr ausführen muss.

Um aus den Anfrageparametern ein Parameterobjekt zu füllen, muss natürlich weiterhin ein *cast* ausgeführt werden. Dies soll dann aber in einer Stelle im System geschehen, die Wissen darüber hat, dass es sich um eine Webanwendung handelt, z. B. in einer Servlet-Klasse.

Zu breite Schnittstelle Die Methode *createNewUser2()* hat so keinen Zugriff auf die HTTP-Anfrage, über die potentiell sehr viele Daten zugreifbar sind. Die Schnittstelle ist allerdings breiter als in 3.3.2, weil nicht nur das essentielle Datum »Benutzername«, sondern ein Parameterobjekt mit potentiell vielen unbenötigten Daten übergeben wird.

Schreibzugriffe auf Parametern möglich Die Methode *createNewUser2()* kann in dieser dritten Form nicht den Zustand der HTTP-Anfrage ändern. Sie kann allerdings auf zustandsverändernde Methoden des Parameterobjektes zugreifen.

Will man diesen Zugriff ausschließen, so gibt es erstens die Möglichkeit, die Parameterobjekt-Klasse so zu definieren, dass ihr Zustand nur über den Konstruktor geändert werden kann. Die zweite Möglichkeit besteht darin, die Parameterobjekt-Klasse so zu definieren, dass ihre zustandsändernden Methoden nur in einem Initialisierungszustand aufgerufen werden dürfen und andernfalls Ausnahmen werfen.

Globale Variablen Die Methode *createNewUser2()* hat in dieser Implementierung keinen Zugriff auf das (üblicherweise einzige) Exemplar des Typs HttpServletRequest. Von dem Typ UserData können auf einfache Weise beliebig viele Exemplare erzeugt werden, sodass üblicherweise nicht nur ein einziges Exemplar während des gesamten Ausführung des Programmes existiert.

Wenn allerdings zur Laufzeit eines Programms nur ein Exemplar vom Typ, der durch die Parameterobjekt-Klasse definiert ist, erzeugt und an alle Methoden weitergereicht wird, so kann auch dieses Exemplar

wie eine Halde für globale Variablen wirken. (Und genau das werden wir beim späteren Umbau des Systems JCommSy tun.)

Schreibt die zu entkoppelnde Methode in eines der ihr übergebenen Objekte, so kann bei der Umstellung auf ein Parameterobjekt auch dies umgestellt werden. Wo die Methode vorher in das HttpServletRequest-Objekt geschrieben hat, kann sie nun in das Parameterobjekt schreiben. Dazu muss die Parameterobjekt-Klasse entsprechende Setze-Methoden anbieten. Die Methode *saveUser()* aus Beispiel 3.3 könnte so geändert werden:

```
/**
 * @require data != null
 * @require user != null
 * @ensure data.hasUsername()
 */
public void saveUsername(UserData data, User user) {
    data.setUsername(user.getName());
}
```

Wie bei der Lösung aus Abschnitt 3.3.2 muss sichergestellt werden, dass nach dem Aufruf der umgestellten Methode der oder die Rückgabewerte in die HTTP-Anfrage geschrieben werden. Dazu mehr in Abschnitt 5.4.1.

3.3.4 Bewertung

Wir fassen die Ergebnisse dieses Abschnitts zur Tabelle 3.1 zusammen. Wie sich zeigt, ist grundsätzlich die Variante aus Abschnitt 3.3.2 (Alle Voraussetzungen soweit wie möglich als Abhängigkeiten und insbesondere als Parameter auszudrücken) empfehlenswert.

Wann sollte also die Variante »Verwendung eines Parameterobjektes« (Abschnitt 3.3.3) verwendet werden? Dies ist z. B. sinnvoll, wenn man die Signatur einer überschriebenen Methode ändern möchte, deren verschiedene überschreibende Methoden unterschiedliche Daten aus der HTTP-Anfrage oder -Sitzung verwenden. Da die Signatur von überschreibenden Methoden die gleiche sein muss wie die der überschriebenen, bietet sich in solch einem Fall die Verwendung eines Parameterobjektes an. (Eben solche Änderung müssen wir zur Entflechtung von JCommSy durchführen, Abschnitt 4.5.)

Tabelle 3.1: Vergleich der verschiedenen Maßnahmen gegen Rahmenwerkskopplung

	Ausdrückliche Voraussetzungen (3.3.1)	Abhängigkeiten statt Vorbedingungen (3.3.2)	Parameterobjekt (3.3.3)
Technologiefestlegung	-	++	++
Wissen an falscher Stelle	-	++	++
Schlechte Testbarkeit	-	++	+
Unklare Schnittstelle	+	++	+
Verlust Typsicherheit	+	++	++
Zu breite Schnittstelle	-	++	+
Schreibzugriffe auf Parametern möglich	-	++	+
Globale Variablen	-	++	+

Legende: - keine Verbesserung, + Verbesserung, ++ deutliche Verbesserung

3.4 Zusammenfassung

Dieses Kapitel hatte Kopplung an das Servlet-Rahmenwerk zum Thema und wie man sie lösen kann.

Wir haben zunächst die typische Verwendung des Rahmenwerks betrachtet. Um eine Webanwendung zu erzeugen, wird eine Klasse von der Rahmenwerksklasse HttpServlet abgeleitet und eine oder beide der Methoden *doGet()* und *doPost()* überschrieben. Diese beiden Methoden haben Parameter der Typen HttpServletRequest und HttpServletResponse. Wenn die Webanwendung eine Architektur nach dem Prinzip der Trennung von Interaktion und Funktion hat, dann befindet sich das Servlet – d. h. die von HttpServlet abgeleitet Klasse – in einem Architekturelement, das die Oberfläche repräsentiert (und benutzt die Fachlogik). Nach der Architekturregel zur Verwendung von Rahmenwerken darf nur das Oberflächenarchitekturelement auf das Servlet-Rahmenwerk zugreifen. Wenn aus der Fachlogik auf das Rahmenwerk zugriffen wird, wird diese Regel verletzt. Das ist der Fall, wenn Exemplare der Typen HttpServletRequest und HttpServletResponse von einem Servlet an eine aufgerufene Methode aus der Fachlogik weitergeben werden.

Wenn eine Methode der Fachlogik einen Operationsparameter vom Typ HttpServletRequest hat, kann sie über die Methode *getParameter()* auf die Anfrageparameter zugreifen. Solche Methode ist auf die Oberflächentechnologie Servlet-Rahmenwerk festgelegt und nicht mit anderen Oberflächentechnologien verwendbar. Sie hat Wissen über die Namen der Anfrageparameter, das besser an anderer Stelle im System aufgehoben wäre. Die Methode ist schwierig zu testen, weil sie für den Test in einem Anwendungsserver ausgeführt werden muss. Ihre Schnittstelle ist unklar und zu breit, weil sie einerseits stillschweigende und andererseits akzidentielle Voraussetzungen hat. Durch die Art und Weise, wie die Methoden HttpServletRequest.*getParameter()* und HttpSession.*getAttribute()* entworfen und implementiert sind, verliert unsere Methode die Vorzüge der Typsicherheit. Die Methode hat schreibenden Zugriff auf ihren Operationsparameter und kann deshalb unerwünschte Seiteneffekte – wie das Ändern der Anfrageparameter – auslösen. Dadurch, dass während eines Programmdurchlaufes einer Webanwendung üblicherweise nur genau ein Exemplar von HttpServletRequest existiert, kann dieses Exemplar leicht zu einer Halde für globale Variablen werden, wenn es an zuviele Methoden weitergereicht wird. Für die Schnittstellen HttpServletResponse und HttpSession gilt ähnliches wie für HttpServletRequest.

Die Ergebnisse sind übertragbar auf andere Rahmenwerke als Servlet-Rahmenwerk: Diese Probleme treten in unterschiedlicher Stärke auch bei zu enger Kopplung an andere Rahmenwerke auf. Sie werden davon begleitet, dass akzidentielle Voraussetzungen existieren und die essentiellen Voraussetzungen nicht ausdrücklich gemacht – also stillschweigend – sind.

Für Software, die unter diesen Problemen leidet, haben wir Verfahren vorgeschlagen, die Kopplung zu lösen. Wir haben drei Schritte beschrieben, die darauf basieren, akzidentielle Voraussetzungen zu beseitigen und stillschweigende Voraussetzungen ausdrücklich zu machen: »Voraussetzungen als Vorbedingungen ausdrücklich machen«, »Vorbedingungen in Parameter umwandeln« und »Einführen eines Parameterobjektes«. Die erste Variante ist als erster Schritt auf dem Weg zu einer der anderen Möglichkeiten zu sehen; die zweite und dritte Variante sind Alternativen.

Grundsätzlich ist es sinnvoll, die Variante »Voraussetzungen als Parameter ausdrücken« zu verwenden; in Verbindung mit überschriebenen Methoden bietet sich die »Verwendung eines Parameterobjektes« an.

4 Problemgegenstand: die Architektur von JCommSy

Nachdem wir nun die Probleme, die bei der Kopplung an das Servlet-Rahmenwerk allgemein auftreten können, untersucht und eine Lösung für sie vorgeschlagen haben, wollen wir in diesem Kapitel an einem konkreten Beispiel untersuchen, ob die Lösung die gewünschte Wirkung hat. Wir versuchen, sie an dem Programm anzuwenden, dessen Entwurf die Frage aufgeworfen hat, wie man Kopplung an das Servlet-Rahmenwerk lockern kann: der Webanwendung JCommSy.

In Kapitel 3 haben wir ganz allgemein untersucht, welche Probleme in Anwendungen auftreten können, die zu eng an das Servlet-Rahmenwerk gekoppelt sind. Es wurde eine Lösung zur Entflechtung vorgeschlagen. Dieses Kapitel stellt nun eine Anwendung in den Fokus, die an zu enger Kopplung an das Servlet-Rahmenwerk leidet: das Programm JCommSy. Wir wollen an ihr ausprobieren, ob die von uns vorgeschlagene Entflechtungsstrategie an einem konkreten Beispiel funktioniert. JCommSy wurde nicht zufällig als Beispiel ausgewählt. Vielmehr entstand aus dem Wunsch, den Entwurf von JCommSy zu verbessern diese Arbeit. Ein Ziel dieser Arbeit ist es deshalb, JCommSy von der zu engen Kopplung zu befreien.

Nach einer kurzen Einführung in die Anwendung und ihre Benutzung gehen wir intensiv auf die Architektur von JCommSy ein und betrachten insbesondere die Teile der Architektur, die das Servlet-Rahmenwerk verwenden.

Bei der Untersuchung wird sich zeigen, dass die Entflechtung von JCommSy vom Servlet-Rahmenwerk schwieriger ist, als zunächst angenommen, weil die Anwendung unter einem weiteren Problem leidet. Dieses werden wir dann in Kapitel 5 genauer betrachten.

4.1 Die Anwendung JCommSy

CommSy ist eine Web-basierte Anwendung (Abschnitt 2.8), die speziell für die Bedürfnisse von Projekt-basiertem Lernen entworfen wurde (Janneck u. Bleek 2002). Es wurde an der Universität Hamburg im Rahmen des vom Bundesministeriums für Forschung und Entwicklung geförderten Forschungsprojektes WissPro[1] ursprünglich für die Unterstützung der Lehre im Informatikstudium entwickelt (Bleek u. Jackewitz 2004). Heute hat sich sein Einsatzrahmen vergrößert. CommSy wird an verschiedenen Departments der Universität Hamburg (u. A. Erziehungswissenschaften, Informatik) und an hamburger Schulen eingesetzt. Desweiteren wird es nicht mehr nur in der Lehre, sondern auch z. B. in Unternehmen verwendet. Es dient als Forschungsobjekt einerseits für E-Learning-Forschung andererseits für softwaretechnische Themen.

CommSy wurde ursprünglich in PHP implementiert. Aus verschiedenen Gründen (die in Jeenicke 2005 erläutert werden) wurde beschlossen, das System in Java neu zu entwickeln (also eine *Migration* durchzuführen). Zur besseren Unterscheidbarkeit wird die neue Verkörperung *JCommSy*, die alte *PCommSy* genannt. JCommSy soll die gleiche Funktionalität und das gleiche Aussehen haben wie PCommSy. Für die Benutzer soll der Übergang also unmerklich vonstatten gehen.

Da die Migration während des Betriebes und der Weiterentwicklung von PCommSy stattfindet, arbeiten derzeit mehrere Entwickler parallel an beiden Implementierungen. Stück für Stück sollen Teile des Produktivsystems von PCommSy zu JCommSy wandern.

Das Gesamt-Projekt CommSy wird hauptsächlich von Studenten und Forschern des Departments Informatik vorangetrieben. Diese sind Mitglieder unterschiedlicher Arbeitsbereiche, arbeiten in unterschiedlichen Forschungsprojekten und forschen in unterschiedlichen Disziplinen. Das Projekt hat also eine hohe *organizational distribution* (Gumm 2006), die zu unterschiedlichen Herausforderungen führt. So ist es z. T. für die Mitglieder des Projekt-Teams schwierig, Zusagen zu machen, weil ihre jeweiligen Arbeitsbereiche unterschiedliche Ziele haben (Gumm 2006). Weil CommSy heute produktiv eingesetzt wird, widersprechen sich manchmal die Ziele der Betreiber (das System muss stabil laufen) und die Ziele der Forscher (neue Technologien und Konzepte sollen ausprobiert werden).

[1] dazu siehe WissPro-Webseite 2007

4.2 Benutzung

Der Begriff »CommSy« wird in mehreren Bedeutungen verwendet. Einmal wird damit die CommSy-Software gemeint. Zum anderen wird mit »einem CommSy« – d. h. einem *community system* – eine konkrete Installation der Software bezeichnet. Das CommSy der Universität Hamburg ist z. B. unter `http://www.commsy.uni-hamburg.de/` (Uni-CommSy 2007) zu erreichen. Dieses enthält die verschiedenen *Portale* der Universität. Das Portal ist die gröbste Organisationseinheit in einem CommSy. Das CommSy der Universität Hamburg enthält z. B. »EduCommSy«, das Portal des Departments Erziehungswissenschaft und »MIN-CommSy«, das Portal der Fakultät für Mathematik, Informatik und Naturwissenschaften. (CommSy-Handbuch 2005)

Ein solches *community system* wird über eine Webschnittstelle, also einen *thin client* bedient; derzeit existiert kein *rich client* als grafische Benutzeroberfläche (engl. *graphical user interface*, GUI). Dieses *community system* besteht hautpsächlich aus mehreren virtuellen *Räumen*, die wiederum sogenanntes *Rubriken* enthalten. In einem *Gemeinschaftsraum* werden mehrere Räume zusammengefasst. Ein Raum kann Teil mehrerer Gemeinschaftsräume sein. Ein Portal enthält wiederum mehrere Räume und Gemeinschaftsräume.

Zugriff auf das CommSy haben angemeldete *Personen*. Jede Person hat eine *Kennung*, die für jeweils ein CommSy gilt. Personen können die Mitgliedschaft in einem Raum beantragen und können dann vom *Verwalter* des Raumes als *Mitglied* freigeschaltet werden. Desweiteren können sie sich in *Gruppen* organisieren.

Die Benutzer können in *Diskussionen* miteinander kommunizieren. Außerdem können sie *Materialien* der Raumöffentlichkeit zur Verfügung stellen. Materialien können Texte sein oder auch Dateien, die in das CommSy hochgeladen werden können.

Für viele der o. g. Begriffe gibt es innerhalb eines Raumes Rubriken. Typische Rubriken sind z. B. »Ankündigungen«, »Diskussionen« und »Materialien«. Welche Rubriken in einem Raum zur Verfügung stehen, kann vom Verwalter des Raumes bestimmt werden. Die für den jeweiligen Raum freigeschalteten Benutzer können Einträge in den Rubriken hinzufügen und verändern. Eine Sonderrolle spielt die Rubrik »Home«, die aktuelle Daten aus mehreren anderen Rubriken zusammengefasst darstellt.

Die Benutzeroberfläche beider Verkörperungen von CommSy sind gleich. Gegenstand der Untersuchungen dieser Arbeit ist nur JCommSy, dessen Architektur wir uns nun genauer anschauen.

4.3 Architektur von JCommSy

Das Programm JCommSy wird innerhalb einer für Webanwendungen typischen Teilung auf drei Prozesse ausgeführt, wie in Abbildung 4.1 zu sehen. Der Benutzer von JCommSy greift mit seinem Browser auf den Anwen-

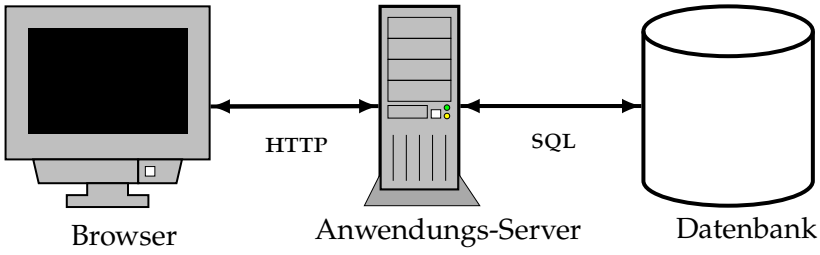


Abbildung 4.1: Teilung auf drei Prozesse

dungsserver zu, der wiederum Daten aus einer Datenbank liest und in sie schreibt. Der Großteil des Quelltextes von JCommSy ist in Java programmiert und läuft auf der Ebene des Anwendungsservers ab. Hinzu kommen noch kleine Teile Code in Java-Script, die im *browser* des Benutzers ablaufen.[2] Die Datenbank wird nur zum Speichern von Daten verwandt; sie enthält keine Anwendungssoftware wie *stored procedures* (Date 2004) oder ähnliches. Der Datenbankprozess kann auf einem anderen Rechner als der Anwendungsserverprozess ausgeführt werden. (Der Prozess des Browsers natürlich auch.)

Aus dem Baukasten der JEE (Abschnitt 2.8.2) werden Elemente des *presentation layer* (die Technologien Servlet und JSP, 2.8.2) verwendet, jedoch nicht die Technologie »Enterprise Java Beans (EJB)« (Ihns u. a. 2007). Für die Ausführung wird daher ein *servlet-container*, wie etwa Tomcat (Tomcat-Webseite 2008), aber kein EJB-*container*, wie JBoss (Rupp 2005), benötigt. Für den Zugriff auf die Datenbank wird die Software Hibernate (Beeger u. a. 2007) eingesetzt. Dadurch wird von der konkreten Implementierung der Datenbank abstrahiert. CommSy kann Datenbanken verschiedener Hersteller verwenden; typischerweise (so auch im Entwicklungsprojekt) wird MySQL (Dyer 2005) benutzt.

[2] Diese Teile sind prinzipiell unabhängig von der verwendeten Verkörperung von CommSy (also PCommSy oder JCommSy). Es ist geplant, diese in eine Art Rahmenwerk auszugliedern, damit sie von beiden Verkörperungen verwendet werden können. In Zukunft wird dieser Teil anwachsen, weil vermehrt AJAX und DHTML verwendet werden sollen, um die Benutzerschnittstelle zu verbessern.

Durch manuelle und maschinelle Analyse wurde die Architektur von JCommSy eingehend untersucht. Die maschinelle Analyse wurde mit dem Sotographen (Bischofberger u. a. 2004) durchgeführt. Im Detail stellt die Architektur sich wie in Abbildung 4.2 dar. Gezeigt werden in dieser Abbildung

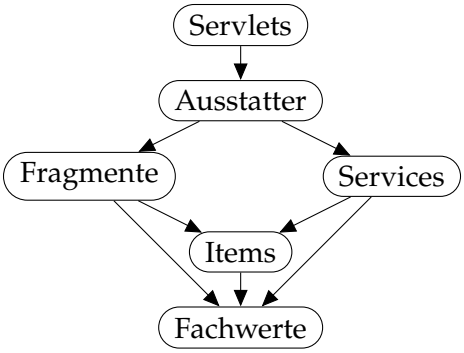


Abbildung 4.2: Die Architektur von JCommSy (Legende in Abschnitt 2.9)

nur Module, die direkt zu JCommSy gehören, nicht jedoch die verwendeten Rahmenwerke. Diese Architektur kann als Schichtenarchitektur aufgefasst werden.

Diese Architektur kann man auf die schematische Architektur zur Trennung von Präsentation und Funktion in Abbildung 3.1 aus Kapitel 3 abbilden. Das Architekturelement »Servlets« aus Abbildung 4.2 entspricht dann »Oberfläche« aus Abbildung 3.1, die restlichen Architekturelemente aus Abbildung 4.2 entsprechen »Fachlogik«. Die Fachlogik aus Abbildung 4.2 speichert Daten in einer Datenbank, die in diesem Diagramm weggelassen wurde.

4.3.1 Die Seitenfragment-Architektur

Eine Besonderheit von JCommSy ist die sogenannte Seitenfragment-Architektur, die im Zuge der vorangegangenen Diplomarbeit[3] von Hohmann (2007) entwickelt wurde. Bei der Seitenfragment-Architektur handelt es sich um einen Architekturstil nach Lilienthal (2008, S. 29) und Reussner u. Hasselbring (2006).

[3]Dort wird sie als »Web Components Architektur« (sic.) bezeichnet.

Um die Seitenfragment-Architektur verstehen zu können, muss man die Geschichte der Architektur von JCommSy und der Entstehung der Seitenfragmente kennen. Die Oberfläche der Anwendung wird wie die jeder Webanwendung von HTML-Seiten dargestellt. Ursprünglich gab es in JCommSy wenige JSPs, von denen jede jeweils mit einer HTML-Seite korrespondierte.

Dieser Ansatz hatte verschiedene Nachteile, u. A. musste Code für Elemente, die auf mehreren (oder sogar allen) Seiten verwendet wurden, in allen diesen Seiten vorgehalten werden. Hohmann führte eine Modularisierung dieser HTML-Seiten ein – die Seitenfragment-Architektur. Die Idee dahinter ist, dass unterschiedliche Seiten aus den gleichen Modulen zusammengesetzt werden. Hohmanns Lösung enthält zwei wesentliche Elemente: *Seitenfragmente* und *Ausstatter*. Die Seitenfragmente realisieren die Module, aus denen eine Oberflächenseite aufgebaut wird. Die Ausstatter versorgen die Seitenfragmente mit den für die Anzeige notwendigen Daten. Beide Konzepte werden wir im Anschluss genau vorstellen.

Seitenfragmente

Die Benutzeroberfläche von JCommSy wird von HTML-Seiten gebildet. Hohmann führte eine Modularisierung dieser HTML-Seiten ein – die Seitenfragment-Architektur. Die Module, aus denen eine Seite zusammen gesetzt wird, heißen heute *Seitenfragmente* oder kurz *Fragmente*. In Hohmanns Arbeit werden dafür noch die Begriffe *Web Component*, *Seiten-Komponente* und *Komponente* verwendet.

Ein Seitenfragment kann selbst wieder aus mehreren Seitenfragmenten zusammengesetzt sein; Hohmann hat sie nach dem Entwurfsmuster *Kompositum* (engl. *composite*, Gamma u. a. 1996, S. 239) konstruiert. Die Seitenfragmente einer Seite bilden also einen Baum, dessen Wurzel die ganze Seite ist (auch die ganze Seite ist also als ein Fragment modelliert). Dieser Baum stellt die *Kompositionshierarchie* der Seitenfragmente dar.

Wir schauen uns beispielhaft die Rubrik »Ankündigungen« an, die in Abbildung 4.3 gezeigt ist. Für Entwicklungszwecke kann man die Seitenfragmente der angezeigten HTML-Seiten von JCommSy mit einem Rahmen umgeben lassen. Dieses Verhalten kann über die Konfigurationsdatei von JCommSy eingestellt werden. Dadurch wird sichtbar, welche Teile einer Seite zu welchem Seitenfragment gehören. An den schwarzen Rahmen in Abbildung 4.3 sehen wir, dass es ein Seitenfragment »Root« gibt. »Root«

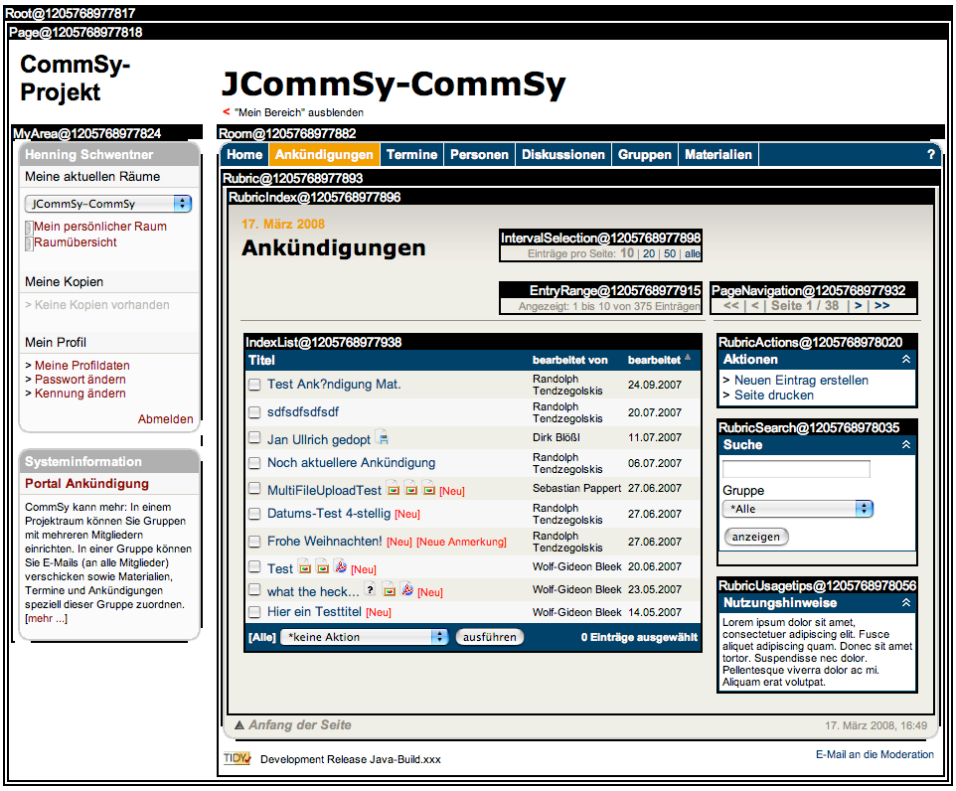


Abbildung 4.3: Bildschirmfoto JCommSy mit umrandeten Seitenfragmenten

enthält das Seitenfragment »Page«, das wiederum einerseits Text (u. A. den Titel des aktuellen Raumes, hier: »JCommSy-CommSy«) und andererseits weitere Seitenfragmente enthält. Diese Seitenfragmente enthalten wieder entweder Text oder andere Seitenfragmente. Den Baum der zu diesem Beispiel passenden Kompositionshierarchie sieht man in Abbildung 4.4. Für die anderen Seiten der Rubrik »Ankündigungen« – wie die Seite, mit der eine neue Ankündigung erzeugt werden kann – existieren ähnliche Kompositionsbäume. Auch die Seiten der anderen Rubriken sind nach diesem Muster aufgebaut.

Um ein tieferes Verständnis der Seitenfragmente zu erlangen, werden wir im Folgenden näher betrachten, wie sie implementiert sind. Die Implementierung eines Seitenfragments besteht aus zwei Teilen: einer JSP und einer Klasse, die von der Oberklasse Fragment abgeleitet ist. Die Klasse, die das

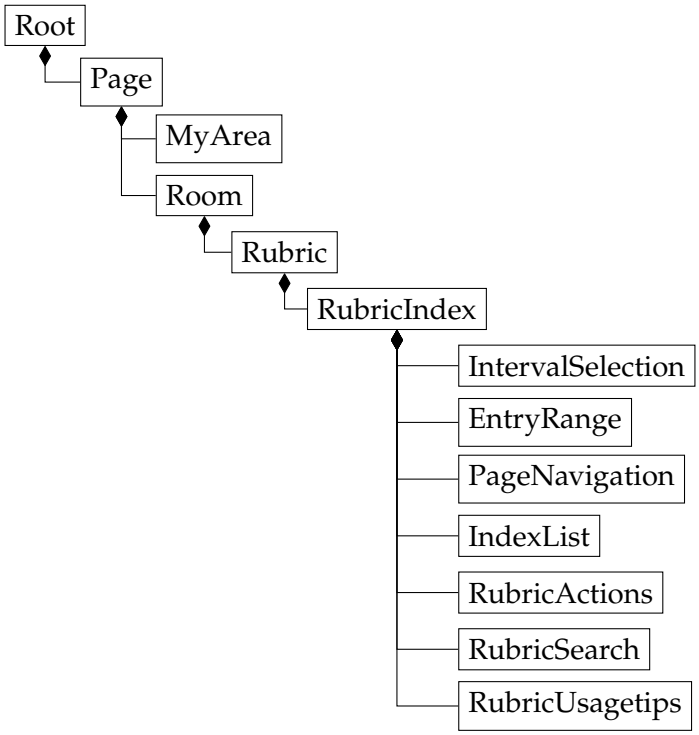


Abbildung 4.4: Kompositionshierarchie der Seitenfragmente aus Abbildung 4.3

Seitenfragment implementiert, ist eine *java bean*[4].

Die Architektur eines einzelnen Seitenfragmente sieht schematisch betrachtet aus wie in Abbildung 4.5. Eine JSP stellt das Fragment dar; dazu benutzt es die Daten, die es aus der zugehörigen Seitenfragment-Klasse erhält. Der Zugriff von der JSP auf die Seitenfragmentklasse, die ja eine *java bean* ist, erfolgt über eine *custom tag library* (kurz *taglib*) (Turau u. a. 2004,

[4] Eine *java bean*, kurz *bean*, ist eine in Java geschriebene Klasse mit folgenden Eigenschaften:

- Sie hat einen Standard-Konstruktor, d. h. einen Konstruktor, der keine Parameter hat.
- Sie bietet an ihrer Schnittstelle sog. Eigenschaften (engl. *properties*) an.
- Für den Zugriff auf Eigenschaften gilt folgende Namenskonvention: Die Eigenschaft mit dem Namen *Eigenschaft* kann mit *getEigenschaft()* ausgelesen und mit *setEigenschaft()* gesetzt werden.

In der Java-Welt werden *beans* zu unterschiedlichen Zwecken eingesetzt. Zum Beispiel sind alle Widgets von Swing *java beans*.

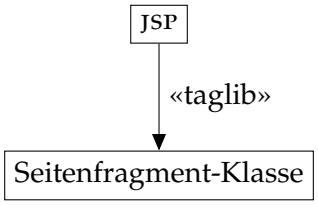


Abbildung 4.5: Architektur eines Seitenfragments – allgemein

S. 107 ff). Um das Architekturdetail leichter verständlich zu machen, schauen wir es uns konkret am Beispiel des Seitenfragments *announcement detail* in Abbildung 4.6 an. Dieses Seitenfragment hat die Aufgabe, die Details einer

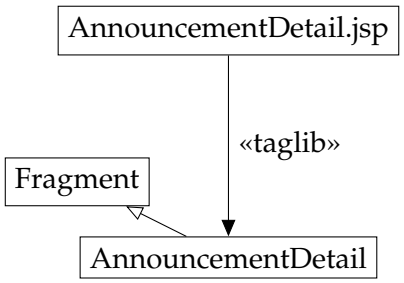


Abbildung 4.6: Architektur eines Seitenfragments – am Beispiel »AnnouncementDetail«

»Ankündigung«, engl. *announcement*, anzuzeigen. »Ankündigungen« ist eine der Rubriken (Abschnitt 4.2) von CommSy.

Die Interaktion ist nach dem MVC-Muster (Reenskaug 1979, in Webanwendungen auch Model-2, Jendrock u. a. 2006, S. 138, genannt) aufgebaut. Bei diesem Entwurfsmuster interagieren drei Einheiten: Modell (engl. *model*), Präsentation (engl. *view*) und Steuerung (engl. *controller*) (Reenskaug 1979). Bei JCommSy entspricht das Fragment dem Modell, die JSP der Präsentation und Servlet und Ausstatter der Steuerung.

Die Klassen, die als Implementierung der Seitenfragmente dienen, sind alle von der abstrakten Klasse FragmentOutfitter abgeleitet. Die Vererbungshierarchie der Seitenfragmentklassen ist in Abbildung 4.7 dargestellt.

Was die Seitenfragmente betrifft, müssen wir also zwei Hierarchien unterscheiden; zum ersten die Kompositionshierarchie zwischen den Seitenfragmenten (Abbildung 4.4), zum zweiten die Vererbungshierarchie der Seitenfragment-Klassen (Abbildung 4.7).

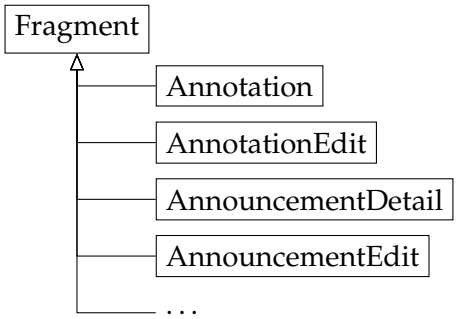


Abbildung 4.7: Vererbungshierarchie der Seitenfragment-Klassen

Ausstatter

Weiterhin gehört zu jedem Fragment wiederum mindestens ein sog. *Ausstatter* (engl. *outfitter*[5]) der das Fragment mit den zur Anzeige nötigen Daten ausstattet. Ein Ausstatter kann aus Unterausstattern bestehen, denen nicht unbedingt ein Fragment zugeordnet sein muss. Wie die Fragmente sind die Ausstatter nach dem Kompositum-Muster entworfen.

Die Oberklasse für alle Ausstatter, d. h. die Klasse FragmentOutfitter, enthält Methoden, um die beiden parallel existierenden Komposita oder Bäume aufzubauen: den Baum der Fragmente und den Baum der zugehörigen Ausstatter. Diese Methoden heißen *buildCommandTree()* und *buildFragmentTree()* und rufen sich jeweils selbst rekursiv auf. Die Methode *buildOutfitterTree()* muss immer vor *buildFragmentTree()* aufgerufen werden, damit erst alle Ausstatter erzeugt werden und die später zu erzeugenden Fragmente mit Daten ausstatten können. Erst wenn alle Ausstatter erzeugt wurden, können mit *buildFragmentTree()* alle Fragmente rekursiv erzeugt werden. Derzeit werden diese Methoden nur von der Klasse FragmentServlet benutzt. Diese Klasse stößt das Aufbauen der Fragmente an, damit sie als Ergebnis als HTML-Seite angezeigt werden können.

Alle Ausstatterklassen sind von der Klasse FragmentOutfitter abgeleitet (deshalb wird sie im Folgenden auch als die Ausstatter*ober*klasse bezeichnet). Die Vererbungshierarchie der Ausstatterklassen ist in Abbildung 4.8 darge-

[5]Der ursprüngliche und bei Hohmann (2007) verwendete Name »Befehl« (engl. *command*) soll an das gleichnamige Entwurfsmuster (Gamma u. a. 1996, S. 273) erinnern, dem sie ursprünglich wohl entsprachen. Heute haben die Module, die in JCommSy Ausstatter heißen, jedoch eine andere Funktion als die Befehle bei Gamma u. a. (1996).

stellt. Es gibt derzeit 81 Ausstatterklassen, deshalb sind in dieser Abbildung

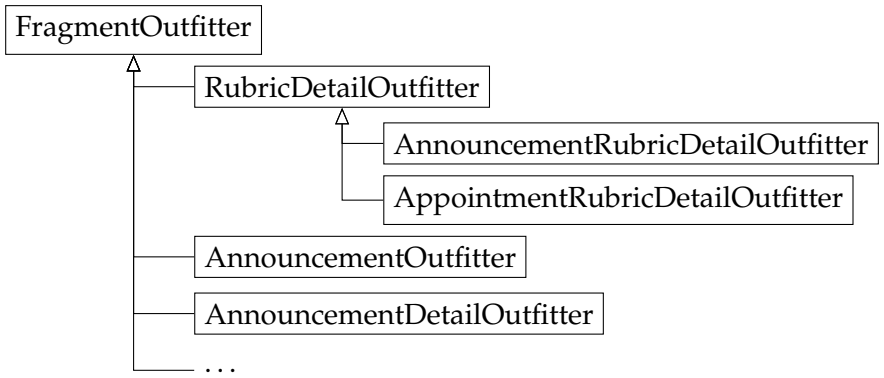


Abbildung 4.8: Vererbungshierarchie der Ausstatter-Klassen

nur einige Beispiele genannt. Wie man sieht, gibt es Ausstatterunterklassen, die selbst wiederum Unterklassen haben.

Auch bei den Ausstattern existieren also zwei Hierarchien, nämlich Kompositionshierarchie und Vererbungshierarchie.

Zusammenhang von Ausstattern und Fragmenten

Die Vererbungshierarchien von Fragmenten und Ausstattern laufen parallel. Zu einem Fragment mit dem Namen X gehört immer ein Ausstatter mit dem Namen XAusstatter. Umgekehrt gilt das nicht, da die Ausstatter feiner granuliert sind als die Fragmente. Es gibt eine Reihe von Ausstattern, denen kein Fragment entspricht, so z. B. Ausstatter, die Daten vorbereiten, die von mehreren Fragmenten gebraucht werden. Hier wird die Gemeinsamkeit dann in eine eigene Ausstatterklasse ausgelagert. Beispiel sind die Fragmente »AnnouncementDetail« und »AnnouncementEdit«, die beide neben anderen vom Ausstatter AnnouncementOutfitter mit Daten ausgestattet werden. Achtung: Hier wird keine Vererbung sondern Komposition verwendet! Das heißt, dass die Unterausstatter eines Ausstatters nicht seine Unterklassen sind, so wie die Unterfragmente eines Fragmentes auch nicht dessen Unterklassen sind.

4.4 Verzahnung von CommSy und *servlet*-Rahmenwerk

Bisher haben wir die Architektur von CommSy untersucht, ohne die verwendeten Rahmenwerke beachten. Da wir JCommSy vom Servlet-Rahmenwerk entkoppeln wollen, untersuchen wir nun, welche Beziehungen zu dieser externen Software bestehen. Wir erweitern die Architekturdarstellung aus Abbildung 4.2 um das Servlet-Rahmenwerk und erhalten Abbildung 4.9. Wie

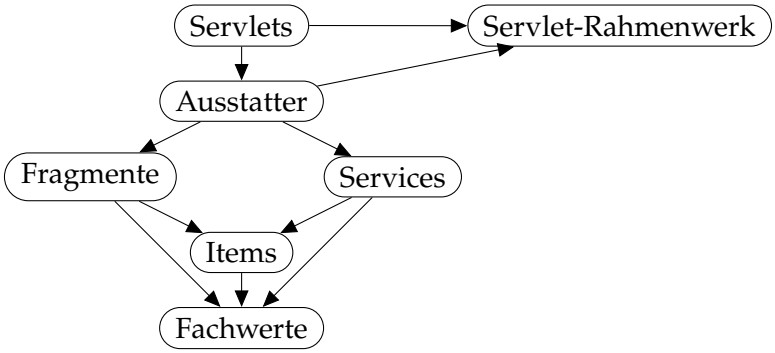


Abbildung 4.9: Die Architektur von JCommSy mit Servlet-Rahmenwerk

man in dieser Abbildung sieht, sind verschiedene Architekturelemente vom Servlet-Rahmenwerk abhängig, neben dem Architekturelement »Servlets«, das Teil der »Oberfläche« aus Abbildung 3.1 ist, auch das Architekturelement »Ausstatter«, das in die »Fachlogik« aus Abbildung 3.1 gehört. In Abschnitt 3.1 wurde festgestellt, dass das Servlet-Rahmenwerk nur aus dem Architekturelement »Oberfläche« heraus verwendet werden darf. Diese zweite Abhängigkeit der »Ausstatter« vom Servlet-Rahmenwerk ist also eine Architekturverletzung. Um zu verstehen, wie die Verletzung entstanden ist, betrachten wir die Architekturelemente »Servlets« und »Ausstatter« näher.

4.4.1 Architekturelement »Servlets«

Wie man aus dem Namen schließen kann, enthält das Architekturelement »Servlets« Klassen, die Servlets (Abschnitt 2.8.2) implementieren. Dies Klassen werden auch *Servlet-Klassen* genannt.

Dieses Architekturelement definiert eine Klasse AbstractCommsyServlet, von der die Servlet-Klassen des CommSy abgeleitet sind. Die Klasse Ab-

stractCommsyServlet erbt von HttpServlet[6]. AbstractCommsyServlet benutzt u. A. die Klassen HttpServletRequest und HttpServletResponse[7] in der in Abschnitt 3.1 beschriebenen Weise. Die Klasse überschreibt die Methoden *doGet()* und *doPost()*. Beide Methoden sind so implementiert, dass sie die abstrakte Einschubmethode *doExecute()* aufrufen, die infolgedessen in allen konkreten Unterklassen implementiert werden muss. Die Methode *doExecute()* hat Parameter der Typen HttpServletRequest und HttpServletResponse; deshalb sind alle Servlet-Klassen von JCommSy von diesen beiden Typen abhängig.

4.4.2 Architekturelement »Ausstatter«

Wenn man die Abhängigkeiten des Architekturelements »Ausstatter« vom Servlet-Rahmenwerk auf die Klassenebene herunterbricht, sieht man, dass zu dem Rahmenwerk keine Vererbungs-, sondern nur Benutzungsbeziehungen bestehen. Weiter zeigt sich, dass alle Ausstatterklassen (d. h. FragmentOutfitter und von ihr abgeleitete Klassen, vgl. Abschnitt 4.3.1) die Klassen HttpServletRequest und HttpServletResponse und einige Ausstatterklassen zusätzlich die Klasse HttpSession (Abschnitt 2.8.2) benutzen. Das sind die einzigen Klassen des Servlet-Rahmenwerks, die vom Architekturelement »Ausstatter« verwendet werden. In Abbildung 4.10 sieht man die abstrakte Klasse FragmentOutfitter mit einer fiktiven konkreten Unterklasse. Im System JCommSy hat FragmentOutfitter 80 Unterklassen.

Eine nähere Untersuchung zeigt, dass die Ausstatteroberklasse FragmentOutfitter ihre Abhängigkeit von HttpServletRequest und HttpServletResponse an ihre Unterklassen vererbt. FragmentOutfitter deklariert eine Reihe von abstrakten Einschubmethoden[8]. Diese Einschubmethoden müssen von den von FragmentOutfitter abgeleiteten Klassen implementiert werden und machen sie so ebenfalls von den Klassen aus javax.servlet.http abhängig. Es handelt sich um die Methoden *createOutfitters()*, *getFragmentClass()*, *createFragment()* und *updateFragment*, die von den Schablonenmethoden *buildOutfitterTree()* und *buildFragmentTree()* aufgerufen werden. Alle diese Methoden erwarten als Parameter je ein Objekt der Typen HttpServletRequest und HttpServletResponse.

[6] Die Klasse HttpServlet stammt aus dem Package javax.servlet

[7] Die drei Klassen HttpServletRequest, HttpServletResponse und HttpSession stammen aus dem Package javax.servlet.http

[8] Die Einschubmethode ist Teil des Entwurfsmusters »Schablonenmethode«, engl. *template method*, Gamma u. a. (1996, S. 131)

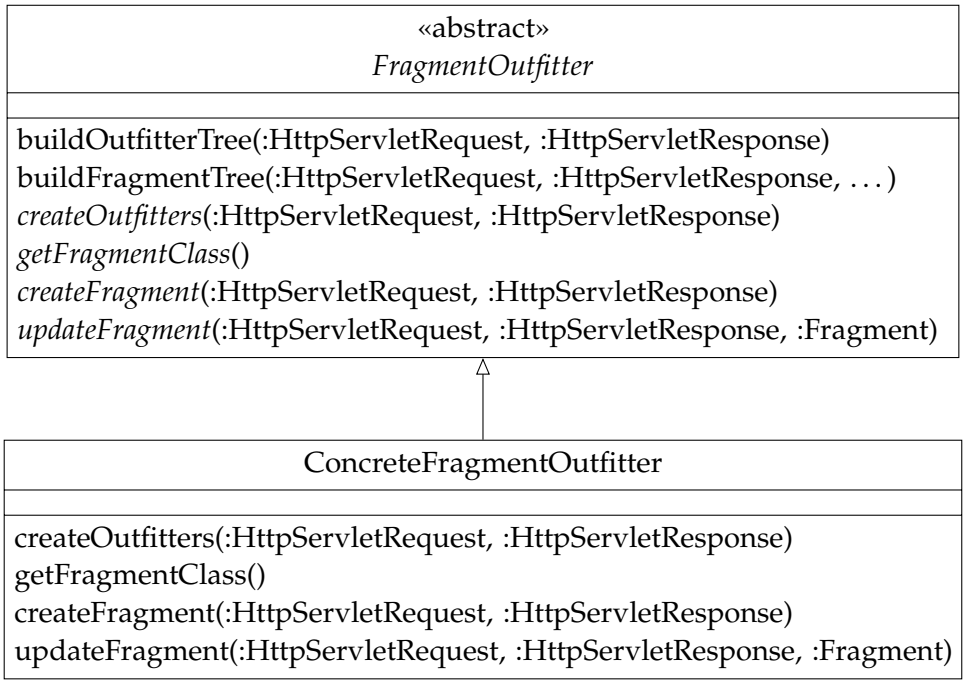


Abbildung 4.10: Abstrakte Klasse FragmentOutfitter mit konkreter Unterklasse

4.5 Versuch einer Entflechtung

Um die Architekturverletzung zu beseitigen, müssen alle Abhängigkeiten des Architekturelements »Ausstatter« zum Servlet-Rahmenwerk beseitigt werden. In einem ersten Versuch verwenden wir dazu den Lösungsansatz aus Abschnitt 3.3. Wir beginnen mit einer fiktiven Unterklasse UserFragmentOutfitter. Sie enthält – wie alle Ausstatter – drei Methoden, die vom Servlet-Rahmenwerk abhängen, nämlich *createOutfitters()*, *createFragment()* und *updateFragment()*. Nehmen wir an, die Methode *updateFragment()* wäre folgendermaßen implementiert:

```
/**
 * @require req != null
 * @require frag != null
 */
public void updateFragment(
        HttpServletRequest req,
```

```
            HttpServletResponse resp,
            Fragment frag) {
        String name = req.getParameter("Username")
        frag.setUsername(user);
}
```

Wenden wir auf diese Methode nun den Lösungsschritt »Voraussetzungen ausdrücklich machen« (Abschnitt 3.3.1) an, so kommen wir zu folgender Signatur für *updateFragment()*:

```
/**
 * @require req != null
 * @require req.getParameter("Username") != null
 * @require frag != null
 */
public void updateFragment(
            HttpServletRequest req,
            HttpServletResponse resp,
            Fragment frag)
```

Nach dem Liskovschen Substitutionsprinzip (Liskov u. Wing 1993) dürfen Unterklassen Vorbedingungen nur abschwächen aber nicht verschärfen; deshalb muss die neu eingeführte Vorbedingung auch Teil der Signatur der Methode *updateFragment()* in der Oberklasse FragmentOutfitter werden. Die anderen von FragmentOutfitter abgeleiteten Klassen müssen die Vorbedingung nicht einführen. Wenn sie sie nicht einführen, so entspricht das einer Abschwächung der Vorbedingung.

Wenn wir nun den zweiten Schritt aus »Vorbedingungen in Abhängigkeiten umwandeln« (Abschnitt 3.3.2) anwenden wollen, kommen wir zu folgender Implementierung von *updateFragment()* in UserFragmentOutfitter:

```
/**
 * @require name != null
 * @require frag != null
 */
public void updateFragment(String name, Fragment frag) {
    frag.setUser(name);
}
```

Das zugehörige Klassendiagramm würde sich wie in Abbildung 4.11 ändern. Wenn wir versuchen, das Programm in dieser Form zu übersetzen, kommt

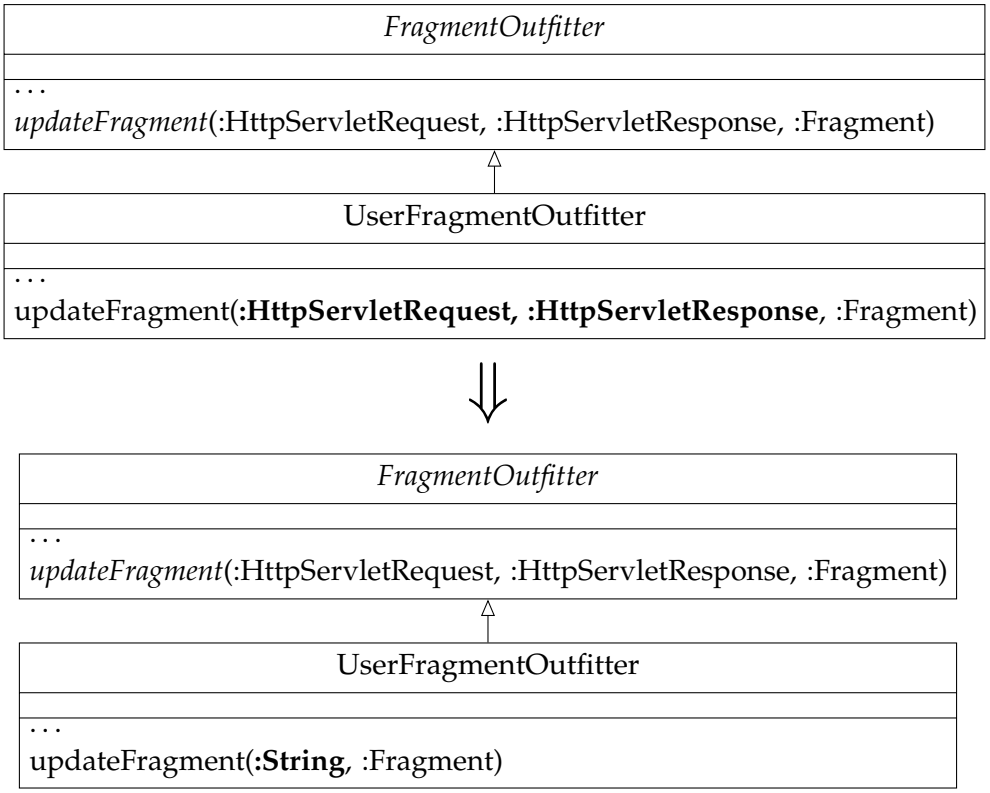


Abbildung 4.11: *updateFragment()* mit String-Parameter

es zu einem Fehler. Dadurch dass, die Signatur von *updateFragment()* in der Unterklasse UserFragmentOutfitter aber nicht in der Oberklasse FragmentOutfitter geändert wurde, überschreibt die Methode in der Unterklasse nicht mehr die in der Oberklasse. Wir müssen also auch die Signatur von *updateFragment()* in der Oberklasse FragmentOutfitter ändern. In der Folge muss auch die Signatur von *updateFragment()* in allen 79 anderen Unterklassen von FragmentOutfitter geändert werden. Hier steckt nun ein Problem; denn die Signatur in folgendem Quelltext:

```
/**
 * @require name != null
 * @require frag !=  null
 */
public  void  updateFragment(String name, Fragment frag)
```

wird nicht in allen anderen Ausstatterunterklassen passen. Zu einer für alle Implementierungen von *updateFragment()* passenden Signatur kommen wir mit dem Lösungsschritt »Verwendung eines Parameterobjektes« (Abschnitt 3.3.3). Damit würde *updateFragment()* in UserFragmentOutfitter so angepasst:

```
/**
 * @require name != null
 * @require params != null
 * @require params.hasName()
 */
public  void  updateFragment(ParamBean params, Fragment frag) {
     String  name = params.getName();
     frag.setUser(name);
}
```

Diese Signatur könnte auch in der Oberklasse FragmentOutfitter und ihren anderen Unterklassen verwendet werden. Diese Änderung würde sich dann wie in Abbildung 4.12 darstellen. Dieser Lösungsweg scheint gangbar; die Parameterobjekt-Klasse ParamBean muss dann so implementiert werden, dass sie alle Daten enthält, die die verschiedenen Implementierungen von *updateFragment()* verwenden.

Es zeigt sich allerdings ein ganz anderes Problem: damit die Änderung durchgeführt werden kann, muss die Signatur von *updateFragment()* in allen 81 Klassen, die diese Methode definieren *in einem Schritt* geändert werden. Schlimmer noch: Es bleibt nicht bei der Änderung der Signaturen; sondern es müssen auch noch die jeweils passenden Implementierungen angepasst werden. Offensichtlich handelt es sich bei der gewünschten Entflechtung um ein Großes Refactoring (Definition 2.7). Es stellt sich die Frage, ob dieses Große Refactoring in kleine Schritte aufgeteilt werden kann, oder ob wirklich alle 81 Klassen auf einmal geändert werden müssen. Diese Frage können wir nicht *ad hoc* beantworten; deshalb werden wir das Problem der Änderung

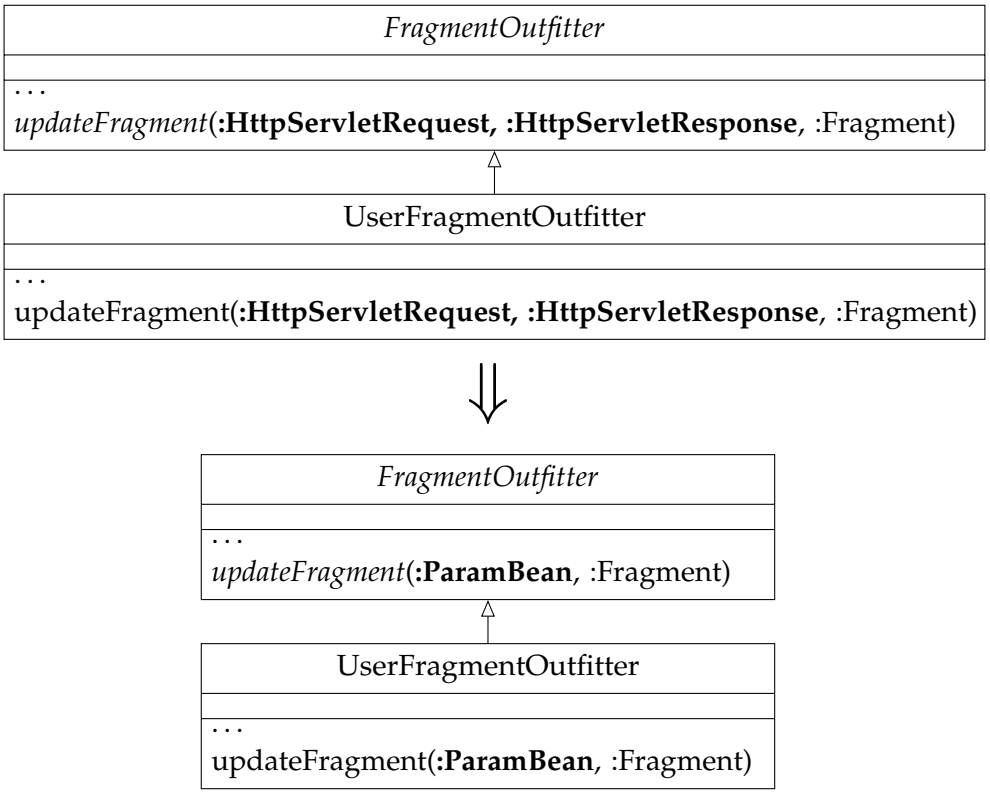


Abbildung 4.12: *updateFragment()* mit ParamBean-Parameter

einer vielfach überschriebenen Methode im nächsten Kapitel isoliert und unabhängig vom Servlet-Rahmenwerk betrachten.

4.6 Zusammenfassung

In diesem Kapitel haben wir JCommSy untersucht, ein Softwaresystem, das unter der in Kapitel 3 beschriebenen zu engen Verflechtung mit dem Servlet-Rahmenwerk und den daraus folgenden Problemen leidet. Um die Probleme zu beseitigen, versuchten wir, JCommSy und Servlet-Rahmenwerk mit dem Lösungsansatz aus Kapitel 3 zu entflechten.

CommSy ist eine ursprünglich in PHP implementierte Webanwendung für Projekt-basiertes Lernen. Im Rahmen eines Migrationsprojektes wird es in Java neu implementiert (und dann JCommSy genannt).

JCommSy hat eine Schichtenarchitektur und verwendet das von JEE angebotene Rahmenwerk für Webanwendungen mit Servlets. Eine Besonderheit ist die Seitenfragment-Architektur (engl. *web component architecture*), bei der einzelne Teile einer HTML-Seite durch sogenannte Seitenfragmente realisiert werden. Die Implementierung eines Seitenfragment besteht aus einer JSP und einer Java-Klasse (genannt Seitenfragment-Klasse), die eine *java bean* ist. Ein Seitenfragment kann weitere Seitenfragmente enthalten; die Fragmente einer HTML-Seite bilden einen Kompositionsbaum. Ein Seitenfragment wird durch sogenannte Ausstatter ausgerüstet und aufgebaut. Ein Ausstatter kann Unterausstatter haben. Zu jedem Seitenfragment gehört mindestens ein Ausstatter. Alle Ausstatter enthalten Methoden, die Abhängigkeiten von HttpServletRequest und anderen Typen des Servlet-Rahmenwerks haben. Diese Abhängigkeiten sind akzidentiell.

Die derzeitige Implementierung von JCommSy ist also unnötig eng an das Rahmenwerk für Servlets gekoppelt. Wir haben deshalb versucht, die Lösungsschritte aus Kapitel 3 anzuwenden, zunächst »Vorbedingungen ausdrücklich machen« und dann »Parameterobjekt einführen«. Dabei stießen wir auf ein weiteres Problem: Um JCommSy und Servlet-Rahmenwerk zu entflechten, müssen wir die Signatur mehrerer vielfach (jeweils genau achtzigmal) überschriebener Methoden ändern. Die Änderung der Signatur einer überschriebenen Methode ist deshalb problematisch, weil in einem Schritt die überschriebene Methode und die überschreibenden Methoden angepasst werden müssen.

Im nächsten Kapitel wollen wir untersuchen, ob es eine Möglichkeit gibt, diesen Schritt in mehrere kleine und einfach handhabbare Unterschritte zu zerschlagen. Diese Betrachtung soll zunächst unabhängig vom Servlet-Rahmenwerk erfolgen. In Kapitel 6 wollen wir dann zu JCommSy zurückkehren und die hoffentlich erarbeitete allgemeine Lösung am konkreten Beispiel ausprobieren.

5 Problemfeld B: Ändern der Signatur überschriebener Methoden

In diesem Kapitel werden wir ein Problemfeld betrachten, das auf den ersten Blick nicht direkt mit Rahmenwerkskopplung (Kapitel 3) zu tun hat, das aber die Entflechtung von unserem Problemgegenstand JCommSy und dem Servlet-Rahmenwerk (Kapitel 4) schwierig macht. Es handelt sich hierbei um die Schwierigkeiten bei der Änderung der Signatur einer überschriebenen Methode. Solche Änderung ist schwierig, weil nicht nur eine, sondern mehrere Methodendefinitionen angepasst werden müssen.

Wir werden zunächst die unterschiedlichen Arten untersuchen, nach denen die Änderung von Quelltext eingeteilt werden kann. Wir gehen genau auf die Änderung der Signatur von überschriebenen Methoden ein und beleuchten in JCommSy autretende Sonderfälle. Schließlich präsentieren wir eine Lösung, die diese Änderung einfach zu handhaben macht. Diese Lösung soll dann in Kapitel 6 verwendet werden, um JCommSy zu entkoppeln.

Im Abschnitt »Übertragbarkeit« werden wir erläutern, warum wir vermuten, dass die in diesem Kapitel beschriebenen Schwierigkeiten typisch für die Kopplung an eine bestimmte Art von Rahmenwerken ist.

5.1 Arten der Änderung

Das Ändern von Quelltext durch Refactoring (Definition 2.6) kann in drei Arten eingeteilt werden.[1] Fowler erwähnt die Arten in Kapitel 2.5.2 seines Buches »Refactoring« (Fowler 2000, S. 53 ff), geht jedoch dann nicht weiter auf sie ein. Da die Methoden in JCommSy, die für diese Arbeit geändert wer-

[1] Grundsätzlich können alle Änderungen von Quelltext nach diesen Arten unterschieden werden, also auch solche, die keine Refactorings sind. Wir interessieren uns hier jedoch besonders für Refactorings.

den müssen, überschrieben sind, wollen wir die Arten genauer untersuchen. Die von Fowler unterschiedenen Arten der Änderung eines Moduls sind:

1. Änderung seiner Implementierung (Abschnitt 5.1.1)
2. Änderung seiner *öffentlichen* Schnittstelle (Abschnitt 5.1.2)
3. Änderung seiner *veröffentlichten* Schnittstelle (Abschnitt 5.1.3)

In einem objektorientierten System handelt es sich bei diesen Modulen üblicherweise um Klassen oder Methoden.

Die Eigenschaften einer Änderung kann man am zugehörigen Klassendiagramm verfolgen. Bei einer Implementierungsveränderung muss nur eine Klasse angepasst werden, bei einer Schnittstellenveränderung jedoch mehrere (so es Benutzer der Schnittstelle gibt).

Die Schnittstelle einer Klasse wird u. A. durch die Schnittstellen der in ihr definierten Methoden bestimmt. Die Schnittstelle einer Methode ist ihre Signatur (Definition 2.10). Die Änderungen der Signatur einer Methode können unterschieden werden nach:

1. Änderung der Signatur von *nicht-überschriebenen*[2] Methoden
2. Änderung der Signatur von *überschriebenen* Methoden

Diese Einteilung ist unabhängig davon, ob die geänderte Methode Teil einer veröffentlichten oder öffentlichen Schnittstelle ist.

Wir betrachten im Folgenden kurz die unterschiedlichen Varianten und gehen dann intensiv auf die Änderungen der *überschriebenen* Methoden ein, weil diese das Lockern der Kopplung von JCommSy an das Servlet-Rahmenwerk erschweren.

5.1.1 Änderungen einer Implementierung

Änderungen an der Implementierung eines Moduls, die seine Schnittstelle nicht betreffen, sind einfach und meist gefahrlos durchzuführen. Einfach, weil sie üblicherweise nur eine Datei betreffen und nur diese Datei nach der Änderung neu übersetzt werden muss. Gefahrlos, weil für die meisten

[2] Weil es kürzer ist, schreiben wir statt »eine Methode, die keine Methode überschreibt und nicht überschrieben wird« knapper »eine nicht-überschriebene Methode« und statt »eine Methode, die eine Methode überschreibt und/oder überschrieben wird« knapper »eine überschriebene Methode«

dieser eher kleinen Änderungen sichere Refactorings (Abschnitt 2.8) mit schrittweisen Durchführungsanweisungen existieren.

Falls der zu dem Modul gehörige Komponententest (Definition 2.4) ein Glass-Box-Test (Ludewig u. Lichter 2007, S. 480) ist, wird auch dieser von der Änderung betroffen.

Beispiel 5.1
Umbenennen einer Exemplarvariable (wie in Abbildung 5.1). ✍

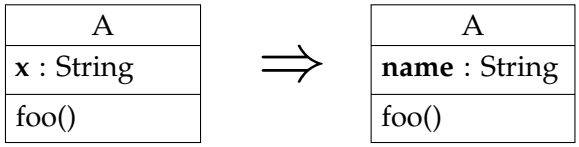


Abbildung 5.1: Beispiel 5.1: Umbenennen einer Exemplarvariable (Legende in Abschnitt 2.9)

Beispiel 5.2
Einführen einer lokalen Variable. ✍

Wir bezeichnen solche Änderungen auch als *(Klassen-)lokal*. In Abbildung 5.1 sieht man weshalb: Um das Refactoring durchzuführen, muss nur eine Klasse (hier die Klasse A) geändert werden. Da sich die Schnittstelle von A nicht ändert, müssen Klassen, die A benutzen, nicht angepasst und auch nicht neu übersetzt werden. Klassenlokale Änderungen erhalten sowohl Quelltext- als auch Binärkompatibilität (Gosling u. a. 1997, S. 229) der geänderten Klasse.

5.1.2 Änderungen einer öffentlichen Schnittstelle

Die öffentliche Schnittstelle eines Moduls zu ändern, bedeutet, die Signatur einer seiner Operationen zu ändern. Die öffentliche Schnittstelle einer Klasse zu ändern, bedeutet, die Signatur einer ihrer öffentlichen Methoden zu ändern.

Die Signatur einer Methode zu verändern ist schwierig, weil die Änderung nicht nur ihre *Definition*, sondern auch die *Aufrufe* der Methode betrifft. Wenn es sich um eine öffentliche Methode handelt, befinden sich üblicherweise Aufrufe in anderen Klassen als der, in der sie definiert ist. Dann betrifft die Änderung mehr als eine Klasse.

Beispiel 5.3
Die Klasse A definiert die Methode *foo()*. *foo()* hat den Parameter x1 vom Typ X.

Änderung: Der Parameter x1 wird nicht mehr benötigt. Aus *foo()* soll der Parameter deshalb entfernt werden. Hierzu bietet sich das Refactoring »Parameter entfernen« (Fowler 2000, S. 283 f) an. Diese Änderung ist grafisch dargestellt in Abbildung 5.2. ✍

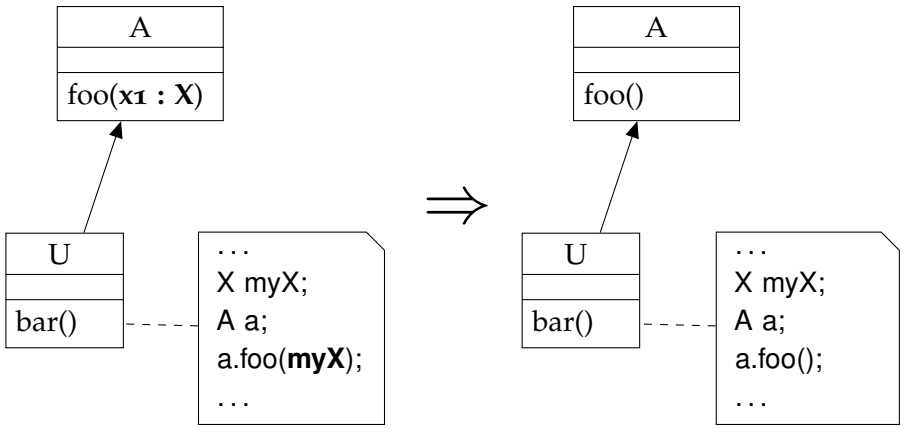


Abbildung 5.2: Beispiel 5.3: Entfernen des Parameters x1

Wie man in Abbildung 5.2 sieht, wird *foo()* von der Methode *bar()* der Klasse U aufgerufen. Wird jetzt der Parameter x1 entfernt, so muss neben der Definition von *foo()* auch der Aufruf von *foo()* in *bar()* angepasst werden.

Solche Änderung nennen wir auch *klassenübergreifend*, weil mehrere Klassen geändert werden müssen. Klassenübergreifende Änderungen zerstören Quelltext- und Binärkompatibilität der geänderten Klasse.

5.1.3 Änderungen einer veröffentlichten Schnittstelle

Eine Schnittstelle heißt *veröffentlicht*, wenn sie von Quelltext aufgerufen wird, auf den der ändernde Programmierer keinen Zugriff hat. Das heißt üblicherweise, dass sie auch in einem anderen Programm als dem, in dem sie definiert wird, verwendet wird. Sie befindet sich dann meist in einem Rahmenwerk.

Damit die eine veröffentlichte Schnittstelle verwendenden Programme auch nach einer Änderung dieser Schnittstelle funktionieren, dürfen nur

folgende Änderungen an ihr durchgeführt werden:

- Hinzufügen von neuen Operationen
- Änderungen an bestehenden Operationen nur in der Art, dass Vorbedingungen abgeschwächt oder Nachbedingungen verschärft werden.

Sollen andere Änderungen durchgeführt werden, so kann dies durch Hinzufügen einer neuen Operation und Umleiten der alten zu ändernden Operation an die neue Operation geschehen. In Java bietet sich zusätzlich die Verwendung der Annotation @Deprecated an.

Solche Änderung heißt nicht nur klassen- sondern auch *programmübergreifend*. Programmübergreifende Änderungen zerstören Quelltext- und Binärkompatibilität der geänderten Klasse und des geänderten Rahmenwerks.

5.2 Problem: Änderung der Signatur von überschriebenen Methoden

Bisher haben wir nur die Änderung von nicht-überschriebenen Methoden (wie die Methode *foo()* in Beispiel 5.3) betrachtet.

Ist die zu ändernde Methode überschrieben (oder überschreibt sie), so ist die Änderung ihrer Signatur noch schwieriger. Das liegt daran, dass neben den (polymorphen) Aufrufen dieser Methode nicht nur eine, sondern *mehrere Methoden-Definitionen* angepasst werden müssen. Es müssen die Signaturen aller die Methode überschreibenden oder von ihr überschriebenen Methoden angepasst und alle entsprechenden Definitionen geändert werden.

Beispiel 5.4
Die Klasse C aus Abbildung 5.3 erbt von der Klasse B, die die Schnittstelle A implementiert. Die drei Typen A, B und C definieren alle eine polymorphe Methode *foo()* mit der gleichen Signatur. Mit dieser Signatur erhält *foo()* in allen Fällen einen Parameter vom Typ X. Die Definition von *foo()* in C überschreibt *foo()* in B, die wiederum *foo()* in A überschreibt. Die Klasse U benutzt die Klasse C, indem ihre Methode *bar()* die Methode *foo()* aufruft.

Änderung: Die Signatur der Methode B.*foo()* soll um einen Parameter vom Typ Y erweitert werden. ✍

Um die Änderung aus Beispiel 5.4 durchzuführen, müssen die Definition von *foo()* in B und der Aufruf von *foo()* in U.*bar()* angepasst werden. Weil

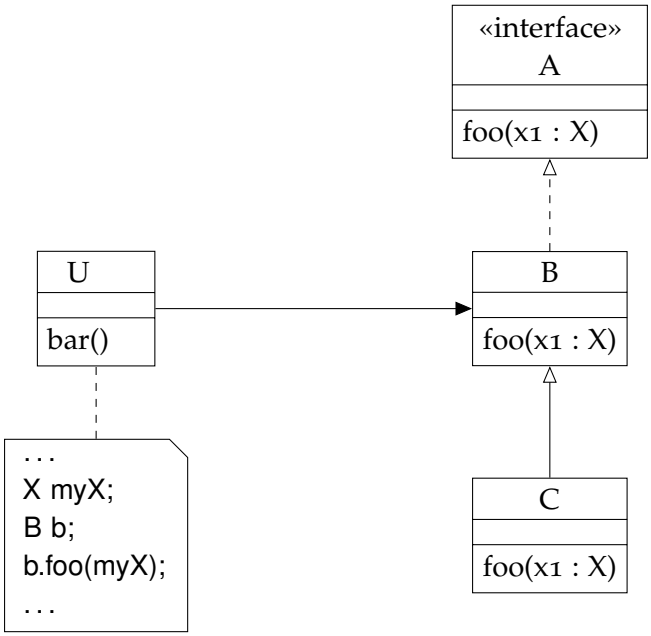


Abbildung 5.3: Beispiel 5.4: Die polymorphe Methode *foo()*

foo() polymorph ist, müssen zusätzlich auch noch die Definitionen von *foo()* in A und C angepasst werden.

Wir betrachten also eine klassen- oder programmübergreifende Änderung, die sowohl horizontal (d. h. auf der Ebene von Benutzungsbeziehungen) als auch vertikal (d. h. auf der Ebene von Vererbungsbeziehungen) wirkt.

5.2.1 Änderung von Signatur und Implementierung

Das Ändern der Signatur einer Methode ist normalerweise kein Selbstzweck. Meist folgt dem Ändern der Signatur ein Ändern der Implementierung der Methode. Bei nicht-überschriebenen Methoden ist das einfach möglich. Hier muss nur *eine* Implementierung angepasst werden. Handelt es sich um eine überschriebene Methode, so steigt der Aufwand, da *mehrere* Implementierungen geändert werden müssen.

Beispiel 5.5
Die Klasse A aus Abbildung 5.4 hat die drei Unterklassen B, C und D. Jede der Unterklassen überschreibt die in der Oberklasse definierte Methode *foo()* mit einer eigenen Implementierung.

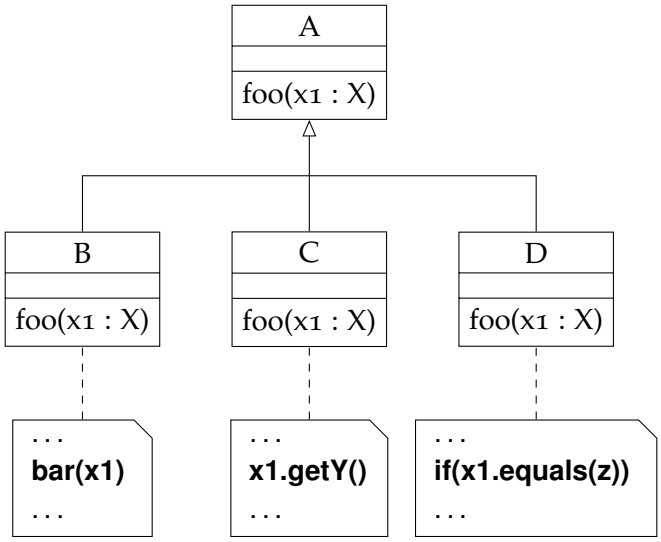


Abbildung 5.4: Beispiel 5.5: Drei Implementierungen von *foo()*

Änderung: Der Parameter `x1` soll durch einen anderen Parameter ersetzt werden. ✍

In Beispiel 5.5 müssen vier Implementierungen angepasst werden. Erben viele Klassen von der Oberklasse, in der die Schablonenmethode implementiert ist, wird es schnell unübersichtlich. Es empfiehlt sich, den Zyklus »Erst Signatur ändern, dann Implementierung ändern« beizubehalten. Allerdings sollte nicht in der Reihenfolge

1. Erst alle Signaturen ändern
2. dann alle Implementierungen ändern

vorgegangen werden, weil so das System erst wieder lauffähig wäre, nachdem *alle* Änderungen gemacht wurden. Besser ist die Reihenfolge:

1. Signatur erster überschreibender Methode ändern
2. Implementierung dieser ersten Methode ändern
3. Signatur zweiter überschreibender Methode ändern
4. Implementierung dieser zweiten Methode ändern
5. usw.

Denn diese Schritte können so ausgeführt werden, dass nach jeweils der Änderung einer Methode (Signatur und Implementierung) das System wieder lauffähig ist. Dazu mehr mehr in Abschnitt 5.3.

5.2.2 Schablonen- und Einschubmethoden

Eine typische Anwendung des Vererbungsmechanismus ist das Entwurfsmuster »Schablonenmethode« (engl. *template method*), wie in Abbildung 5.5 zu sehen und in Gamma u. a. (1996, S. 366 ff) beschrieben. Die Struktur

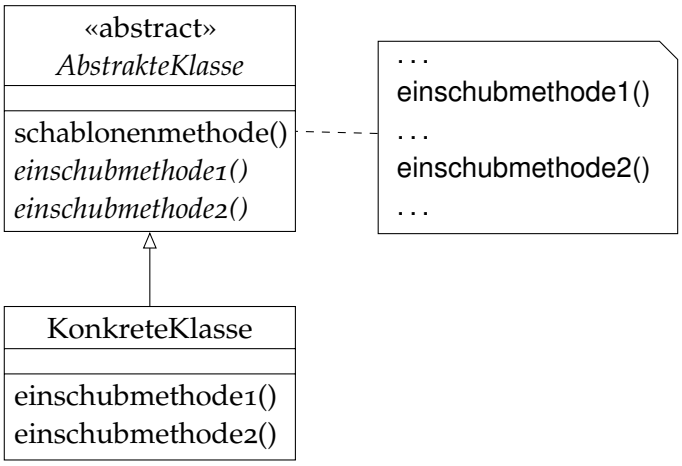


Abbildung 5.5: Das Entwurfsmuster »Schablonenmethode«

eines Algorithmus wird dabei schemenhaft in der sogenannten *Schablonenmethode* definiert. An den Stellen des Algorithmus, die variabel sein sollen, werden sogenannte *Einschubmethoden* aufgerufen. Eine Einschubmethode wird in derselben Klasse wie die Schablonenmethode definiert, allerdings abstrakt, d. h. ohne Implementierung. Konkrete Implementierungen der Einschubmethoden werden erst in einer Unterklasse definiert.

Soll die Signatur einer Einschubmethode geändert werden, so gilt das in Abschnitt 5.2.1 Geschriebene. Zwei Punkte können allerdings die Änderung erleichtern. Erstens ist die Einschubmethode in der Oberklasse abstrakt definiert, dadurch muss eine Methodendefinition weniger als bei einer konkreten Methode angepasst werden. Zweitens wird eine Einschubmethode üblicherweise nur an genau einer Stelle – nämlich der Schablonenmethode – aufgerufen, dann muss nur ein Methodenaufruf angepasst werden. Trotz-

dem müssen mehrere Definitionen der Methode geändert werden und das verursacht den größten Anteil der Änderung.

5.2.3 Schablonen- und Einschubmethode mit gleichen Abhängigkeiten

Wenn die Schablonenmethode die Daten, die ihr übergeben werden, an die Einschubmethode weiterleitet, dann haben Schablonen- und Einschubmethode das gleiche Parameter-Profil (Definition 2.9) und damit zumindest zum Teil auch die gleichen Abhängigkeiten (ein Beispiel siehe Abbildung 5.6). Genau dies ist der Fall in unser Beispielanwendung JCommSy; dort haben

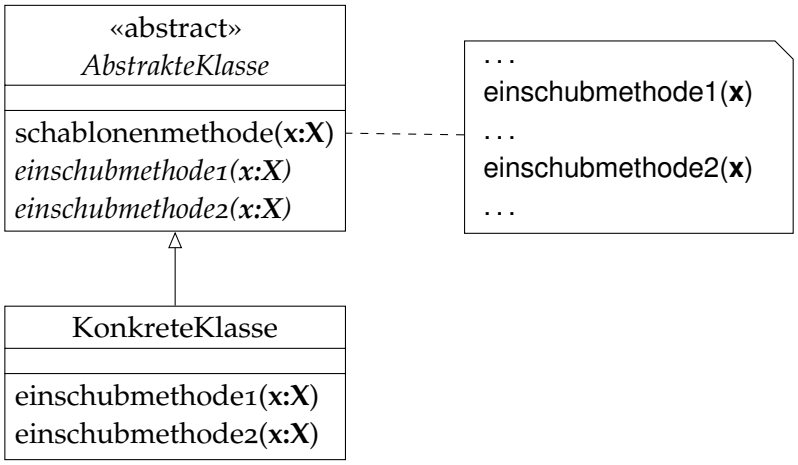


Abbildung 5.6: Schablonen- und Einschubmethoden mit gleichen Abhängigkeiten

die Ausstatterklassen Schablonen- und Einschubmethoden, deren Parameter-Profil die Typen HttpServletRequest und HttpServletResponse enthält.

Soll die Signatur solcher Schablonenmethode geändert werden, so muss die Signatur der Einschubmethoden entsprechend angepasst werden. Der Sinn einer Einschubmethode ist aber gerade, dass sie in verschiedenen Klassen unterschiedlich definiert wird. Die Änderung der Signatur einer Einschubmethode zieht also üblicherweise viele Änderungen in den abgeleiteten Klassen nach sich.

Es kann auch vorkommen, dass die Schablonenmethode nur einen Teil ihrer Parameter weiterleitet. Dann betrifft eine Signaturänderung auch die

Einschubmethoden, wenn sie die weitergeleiteten Parameter ändert.

In JCommSy kommt Schablonen- und Einschubmethoden mit gleichem Parameter-Profil in der Implementierung der Seitenfragment-Ausstatter (siehe Abschnitt 4.3.1) vor.

5.3 Lösung: Wie man die Signatur einer vielfach überschriebenen Methode ändern kann

Wie wir im vorangehenden Abschnitt 5.2 gezeigt haben, ist das Ändern der Signatur einer vielfach überschriebenen Methode nicht einfach möglich. Als naive Lösung bietet es sich an, alle Methodendefinitionen und -Aufrufe, d. h. die der überschriebenen und die der überschreibenden Methoden, in einem großen Schritt anzupassen. Dieses Vorgehen hat einen großen Nachteil: Das System ist erst wieder übersetzbar und lauffähig, wenn die Änderung als Ganzes durchgeführt und abgeschlossen wurde. Je nachdem, wieviele Klassen von der Änderung betroffen sind, ist dieser Nachteil nicht akzeptabel. Deshalb betrachten wir in diesem Kapitel, wie das gleiche Ziel mit einem Verfahren erreicht werden kann, dass aus kleinen Schritten besteht. Jeweils nach einem Schritt ist die Anwendung übersetzbar und lauffähig.

Dazu wird für eine Hierarchie von sich überschreibenden Methoden eine parallele Hierarchie von Methoden mit der gewünschten Signatur aufgebaut und schrittweise von den Methoden mit der alten Signatur an die Methoden mit der neuen Signatur umgeleitet. Hierzu ist es nötig, die Oberklasse, die die überschriebene Methode definiert, so vorzubereiten, dass für die Dauer dieses Großen Refactoring (Definition 2.7) Unterklassen mit Methoden alter Signatur und Unterklassen mit Methoden neuer Signatur parallel existieren können. Wie das erreicht werden kann, zeigen wir an folgendem Beispiel.

Beispiel 5.6
Von der abstrakten Klasse A aus Abbildung 5.7 sind die konkreten Klassen B1, B2, . . . , B100 abgeleitet. Alle Bx implementieren die Methode *foo()* aus A. *foo()* erwartet derzeit einen Parameter vom Typ X. A hat eine triviale Implementierung:

```
public abstract class A {
    public abstract void foo(X x);
}
```

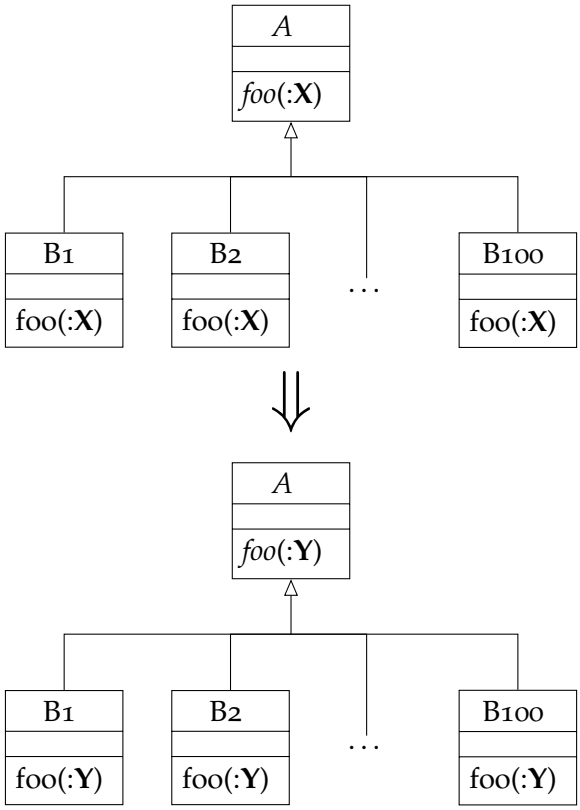


Abbildung 5.7: Beispiel 5.6: Änderung der Signatur von *foo()*

Es ist gewünscht, dass *foo()* in Zukunft statt des Parameters vom Typ X einen Parameter vom Typ Y akzeptiert. ✍

Um Beispiel 5.6 umzustellen, haben wir folgende Idee: Zunächst wird in der Oberklasse A eine neue Methode *foo_NEU()* definiert, die die gewünschte Signatur hat und am Ende *foo()* ersetzen soll. Die Methode *foo()* mit der alten Signatur wird so umgestellt, dass sie an *foo_NEU()* weiterleitet. Dann werden nacheinander die Bx so umgestellt, dass sie statt *foo()* die Methode *foo_NEU()* definieren. Das Programm ist nach jedem dieser Schritte übersetzbar und lauffähig. Wenn alle Bx umgestellt sind und alle Aufrufe von *foo()* auf *foo_NEU()* umgebogen wurden, wird *foo* in A gelöscht. Schließlich wird *foo_NEU()* mit dem Refactoring »Methode umbenennen« in *foo()* umbenannt. Wir werden dieses Verfahrens nun an Beispiel 5.6 im Detail entwickeln.

5.3.1 Vorbereitung – Einführen von foo_NEU()

Zunächst wird in der Oberklasse A eine neue Methode *foo_NEU()* definiert, die langfristig *foo()* ersetzen soll (Abbildung 5.8). Sie erhält das für *foo()*

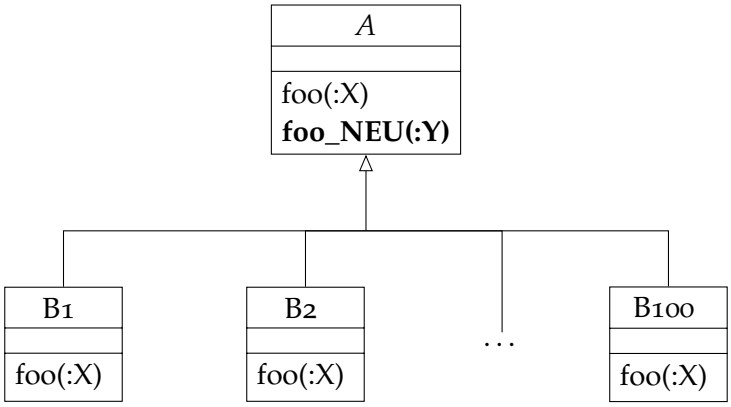


Abbildung 5.8: Schritt 1 – Einführen von *foo_NEU()*

gewünschte neue Parameter-Profil (Definition 2.9), hier also einen Parameter vom Typ Y. Die Definition von *foo()* in A erhält eine Implementierung, die an *foo_NEU()* weiterleitet. Die Methode *foo()* ist nun also nicht mehr abstrakt, die Klasse A bleibt es weiterhin. Die Methode *foo_NEU()* muss schon in der Oberklasse A und nicht erst in den Unterklassen definiert werden, damit A.*foo()* sie aufrufen kann. Es bietet sich an, *foo_NEU()* als abstrakt zu definieren; denn in A soll sie ja keine Implementierung haben, sondern nur in den Unterklassen Bx. Würden wir allerdings *foo_NEU()* abstrakt definieren, müssten wir die Methode sofort in allen konkreten Unterklassen implementieren. Aber da wir schrittweise, d. h. immer nur eine Unterklasse zur Zeit, vorgehen wollen, verbietet es sich, *foo_NEU()* abstrakt zu definieren. Stattdessen definieren wir *foo_NEU()* so, dass sie immer eine Ausnahme wirft. Es wird dann eine Ausnahme vom Typ UnsupportedOperationException geworfen. Dieser Typ wurde für solche Fälle entwickelt, in denen eine Operation keine Implementierung haben soll, aber aus Übersetzungsgründen haben muss.

Damit *foo()* an *foo_NEU()* weiterleiten kann, muss *foo()* ein Exemplar vom Typ X an *foo_NEU()* übergeben. Woher das zu übergebende Exemplar vom Typ X stammt, hängt von der jeweiligen Situation ab. Es kann z. B. sein, dass das gewünschte X-Exemplar aus dem Y-Exemplar extrahiert werden kann.

Eine andere Möglichkeit ist, dass ein Objekt vom Typ X neu erzeugt und mit Daten aus dem Parameter vom Typ Y gefüllt wird. Diese Erzeugung kann z. B. in einer Fabrikmethode (Gamma u. a. 1996, S.131 ff) erfolgen, manchmal existiert für diese eine eigene Klasse (siehe auch Abschnitt 5.4.1). Als neue Implementierung von A ergibt sich:

```
public abstract class A {
    public void foo(X x) {
        Y y = ...;
        foo_NEU(y);
    }
    public void foo_NEU(Y y) {
        throw new UnsupportedOperationException();
    }
}
```

Damit ist die Oberklasse A vorbereitet und wir sind soweit, dass wir die Unterklassen einzeln anpassen können. Die Umstellung der Unterklassen folgt jeweils demselben Muster. Wie dieses Muster aussieht, betrachten wir am Beispiel der Umstellung der Unterklasse B1.

5.3.2 Durchführung – Umstellen der Unterklassen

Die Unterklassen werden nacheinander umgestellt. Zunächst die Klasse B1 wie in Abbildung 5.9 zu sehen. Wie diese Umstellung genau aussieht,

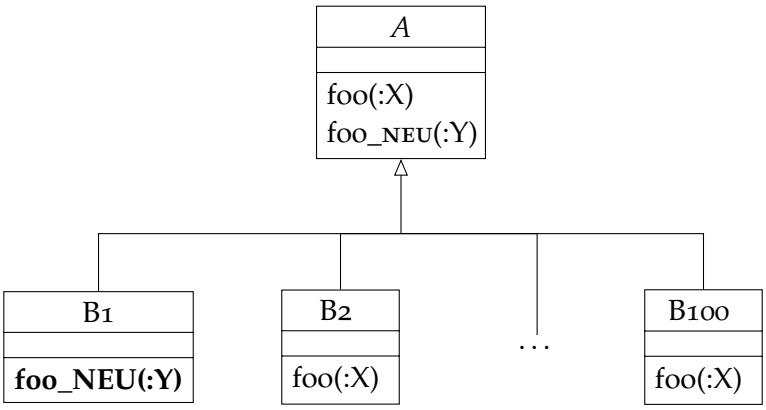


Abbildung 5.9: Schritt 2: Umstellen der ersten Unterklasse

kommt auf die jeweilige Situation an. Erster Aspekt ist, wie die Signatur geändert werden soll. In unserem Beispiel 5.6 wird der Parameter vom Typ X durch einen Parameter vom Typ Y ersetzt. Der zweite Aspekt ist, wie die Implementierung der Methode davon betroffen ist. In Beispiel 5.6 müssten etwa die Daten, die bisher aus dem Parameter vom Typ X gewonnen wurden nun aus dem Parameter vom Typ Y gewonnen werden. Nach dieser Umstellung ist der Quelltext übersetzbar und das System weiterhin lauffähig. Es zeigt das gleiche Verhalten wie vor Beginn des Umbaus, also ist der Umbau ein Refactoring (Definition 2.6). Die Implementierung der Oberklasse A bleibt bei diesem Schritt unverändert; nur die Klasse B1 ändert sich.

Schritt für Schritt wird dann Klasse für Klasse nach diesem Muster umgestellt, bis alle Unterklassen ihre Definitionen von *foo()* duch Definitionen von *foo_NEU()* ersetzt haben (Abbildung 5.10). Das System bleibt – jeweils nach

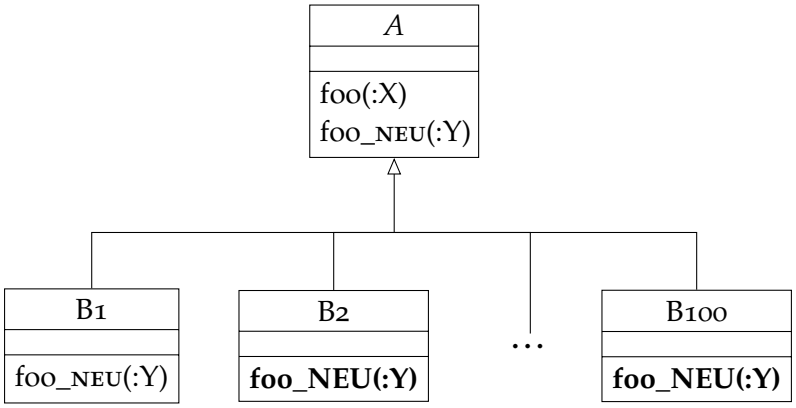


Abbildung 5.10: Schritt 3: Umstellen der weiteren Unterklassen

der Umstellung einer Unterklasse – lauffähig. Technisch gesehen ist also auch jeder dieser Einzelschritte ein Refactoring im Sinne von Definition 2.6. Jeder dieser einzelnen Schritte betrifft jeweils nur eine Klasse und nur diese Klasse muss neu übersetzt werden; insbesondere bleibt die Oberklasse A unverändert.

5.3.3 Durchführung – Umbiegen der Aufrufe

Nachdem alle *Definitionen* von *foo()* umgestellt wurden (und temporär *foo_NEU()* heißen), müssen nun noch die *Aufrufe* von *foo()* auf *foo_NEU()*

umgebogen und damit die neue Signatur verwendet werden. Die Erzeugung eines Objektes vom Typ Y kann erfolgen wie bei der Weiterleitung von *foo()* an *foo_NEU()* (Abschnitt 5.3.1).

Dieser Schritt betrifft potentiell mehrere Klassen, je nachdem in wievielen Klassen die Methode mit zu ändernder Signatur aufgerufen wird. In Beispiel 5.6 ist nicht angegeben, welche Klassen die Methode *foo()* aufrufen.

5.3.4 Nachbereitung

Sind alle Aufrufe von *foo()* auf *foo_NEU()* umgebogen, so kann die Methode *foo()* auch aus der Oberklasse A entfernt werden (Abbildung 5.11). Dann

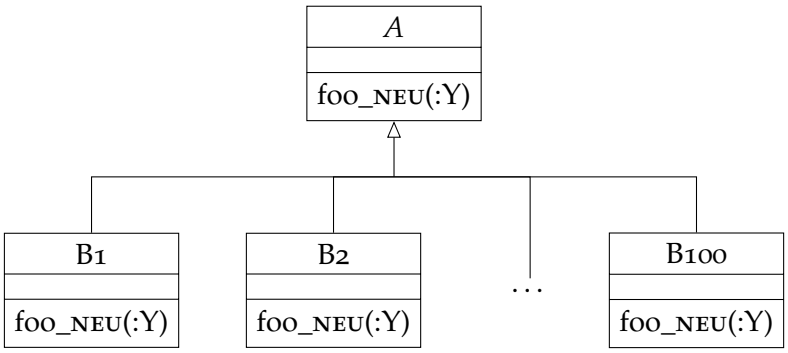


Abbildung 5.11: Nachbereitung

kann auch *foo_NEU()* in A abstrakt gemacht werden. Als neue Implementierung von A ergibt sich:

```
public abstract class A {
    public abstract void foo_NEU(Y y);
}
```

Als letztes wird das Refactoring »Methode umbenennen« (Fowler 2000, S. 279) verwendet, um die Methode *foo_NEU()* in allen Definitionen und Aufrufen in *foo()* umzubenennen. Wir haben nun den gewünschten Zustand erreicht (Abbildung 5.12). Die endgültige Implementierung von A ist:

```
public abstract class A {
    public abstract void foo(Y y);
}
```

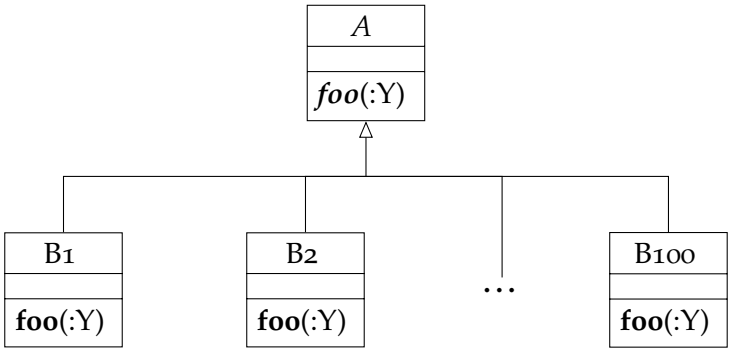


Abbildung 5.12: Endzustand

5.3.5 Fazit

Die gewünschte Änderung konnte in kleinen Schritten durchgeführt werden, zwischen denen Übersetzbarkeit und Lauffähigkeit des Programms gewahrt wurden. Wird die Aktion als Ganzes betrachtet, so sieht es aus, als hätte sich lediglich die Signatur von *foo()* geändert. Die temporäre Methode *foo_NEU* existiert nur innerhalb der Schritte.

Aus dem obigen kleinschrittigen Vorgehen lässt sich eine Beschreibung für ein sicheres Refactoring (Definition 2.8) ableiten. Dieses ist in Abschnitt 7.2.3 beschrieben.

5.4 Die Kombination der beiden Problemfelder

In Software, die überschriebene (im Sinne von Abschnitt 5.1) Methoden hat, in deren Signaturen Typen aus dem Servlet-Rahmenwerk verwendet werden, treten die Probleme aus diesem Kapitel mit denen aus Kapitel 3 in Kombination auf. In solchem Fall können die jeweils beschriebenen Lösungen (Abschnitte 3.3 und 5.3) ebenfalls in Kombination verwendet werden, allerdings muss beachtet werden, dass hier noch eine Einschränkung gilt. Die Signaturen sämtlicher Definitionen einer polymorphen Methode müssen gleich sein. Wir untersuchen an einem Beispiel, was geschieht, wenn diese Einschränkung nicht beachtet wird.

Beispiel 5.7
Von der Schnittstelle IUserCreator sind zwei implementierende Klassen abgeleitet, FreeUserCreator und FamilyYearUserCreator(Abbildung 5.13). IUser-

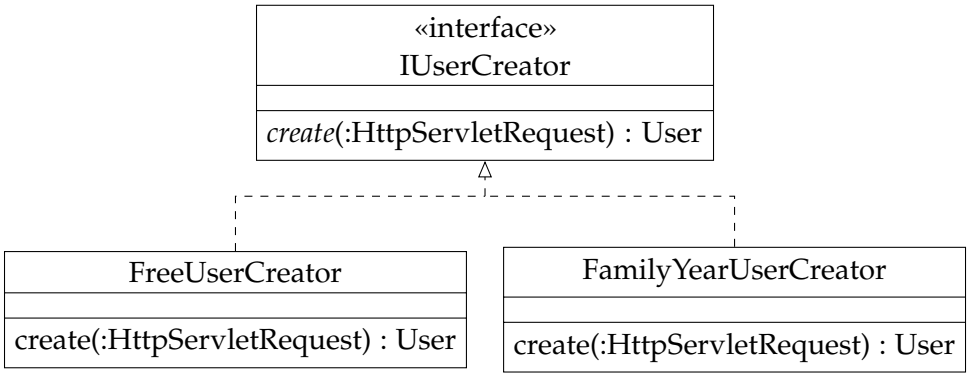


Abbildung 5.13: Beispiel 5.7: Implementierungshierarchie von IUserCreator

Creator definiert die Methode *create()*, die einen Parameter vom Typ HttpServletRequest verlangt. ✍

Jede (konkrete) Klasse, die die Schnittstelle IUserCreator aus Beispiel 5.7 implementiert, muss auch die Methode *create()* implementieren. Das bedeutet, dass jede implementierende Klasse eine Methode mit einem Parameter vom Typ HttpServletRequest enthalten muss. Dadurch wird nicht nur IUserCreator von HttpServletRequest (und damit vom Servlet-Rahmenwerk) abhängig gemacht, sondern auch jede der IUserCreator implementierenden Klassen. Wir sagen, die Abhängigkeit wird vererbt.

Mit den Abhängigkeiten werden auch die Probleme aus Abschnitt 3.2 vererbt, d. h. FreeUserCreator und FamilyYearUserCreator sind genauso technologieabhängig, schlecht testbar usw. wie IUserCreator. Um die Architektur zu verbessern bietet es sich also an, die Abhängigkeiten zu lösen, indem akzidentielle Abhängigkeiten wegfallen und möglichst nur essentielle Abhängigkeiten bestehen bleiben (Abschnitt 3.3). Da hierbei die Schnittstelle einer polymorphen Methode geändert werden muss, bekommen wir es zusätzlich mit den Problemen aus Abschnitt 5.2 zu tun.

Wir betrachten die beiden Implementierungen der Schnittstelle IUserCreator aus Abbildung 5.13.

```
public class FreeUserCreator implements IUserCreator{
    public User createNewUser(HttpServletRequest request) {
        String name = request.getParameter("Username");
        return new User(name);
    }
}
```

```
public class FamilyYearUserCreator implements IUserCreator {
    public User createNewUser(HttpServletRequest request) {
        String name = request.getParameter("Family_Name");
        String yearString = request.getParameter("Year_Of_Birth");
        int yearOfBirth = Integer.parseInt(yearString);
        return new User(name, yearOfBirth);
    }
}
```

FreeUserCreator erzeugt einen User mit Benutzernamen aus dem Anfrageparameter (Abschnitt 2.8.1) »Username«, FamilyYearUserCreator erzeugt einen User mit einem Benutzernamen, der sich aus zwei anderen Anfrageparametern zusammensetzt. Für die Erledigung der Aufgaben der beiden Implementierungen von *createNewUser()* wird das als Parameter übergebene Objekt der Klasse HttpServletRequest nicht selbst benötigt, sondern Daten, die aus diesem Objekt ausgelesen werden können. Die Abhängigkeit der Klassen FreeUserCreator und FamilyYearUserCreator von HttpServletRequest ist also akzidentiell (Definition 2.16).

Um die akzidentielle Abhängigkeit zu lösen, wenden wir die Techniken aus Abschnitt 3.3 an. Wir machen zunächst die von *createNewUser()* aus FreeUserCreator verwendeten Parameter ausdrücklich (Definition 2.17), wie in Abschnitt 3.3.2 beschrieben. Dabei ändert sich der Quelltext wie folgt.

```
public class FreeUserCreator implements IUserCreator{
    public User createNewUser(String name) {
        return new User(name);
    }
}
```

Leider kommt es nach dieser Änderung zu einem Übersetzerfehler. Dies geschieht, weil wir die Signatur nur einer Methodendefinition geändert haben. Für den Übersetzer ist nicht klar, dass *createNewUser(**String**)* aus FreeUserCreator die Methode *createNewUser(**HttpRequest**)* aus IUserCreator überschreiben soll, da sie eine andere Signatur hat. (Technisch gesehen handelt es sich hier nicht um *überschreiben*, sondern um *überladen*. Ein Überschreiben findet in Java nur statt, wenn zwei Methoden die gleiche Signatur haben, Gosling u. a. 1997, S.161)

Wir müssen also die Signatur von *createNewUser(String)* auch in der Schnittstelle IUserCreator und der anderen diese implementierende Klasse (nämlich FamilyYearUserCreator) ändern. Einen Versuch sehen wir in Abbildung 5.14. Auch dieser Implementierungsversuch scheitert, da *cre-*

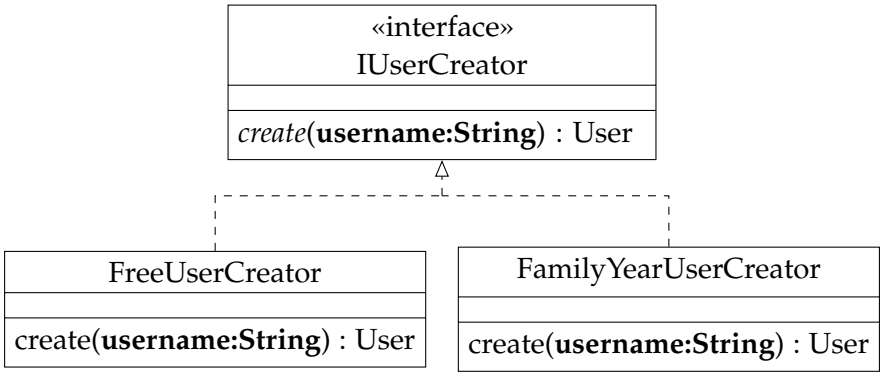


Abbildung 5.14: Implementierungshierarchie von IUserCreator mit expliziten Parametern

ateNewUser() in FamilyYearUserCreator andere essentielle Abhängigkeiten hat, als in FreeUserCreator. Die Implementierung von *createNewUser()* in FreeUserCreator benötigt einen String der einen Namen repräsentiert; die Implementierung von *createNewUser()* in FamilyYearUserCreator benötigt einen String, der einen Familiennamen repräsentiert und einen int der ein Geburtsjahr repräsentiert. Wenn *createNewUser()* in FamilyYearUserCreator als einzigen Parameter einen String erhält, der einen Namen repräsentiert, dann sind ihre essentiellen Voraussetzungen nicht erfüllt.

Als Lösung bietet sich die Technik aus Abschnitt 3.3.3 an, die Einführung und Verwendung eines Parameterobjektes. Die Parameterobjekt-Klasse heiße in unserem Beispiel UserParams. Die Signaturen der Methodendefinitionen ändern sich dann wie in Abbildung 5.15. Die Klasse UserParams muss dann Gib-Methoden (engl. *getter*) für alle Parameter haben, die die unterschiedlichen Definitionen von *createNewUser()* als essentielle Abhängigkeiten haben. Sie wird deshalb implementiert wie in Abbildung 5.16. Die Implementierungen der von IUserCreator abgeleiteten Klassen werden so angepasst, dass sie statt auf einen Parameter vom Typ HttpServletRequest auf einen Parameter vom Typ UserParams zugreifen. Sie rufen statt HttpServletRequest.*getParameter()* die jeweils passende Methode aus UserParams. Es ergibt sich die folgende Implementierung:

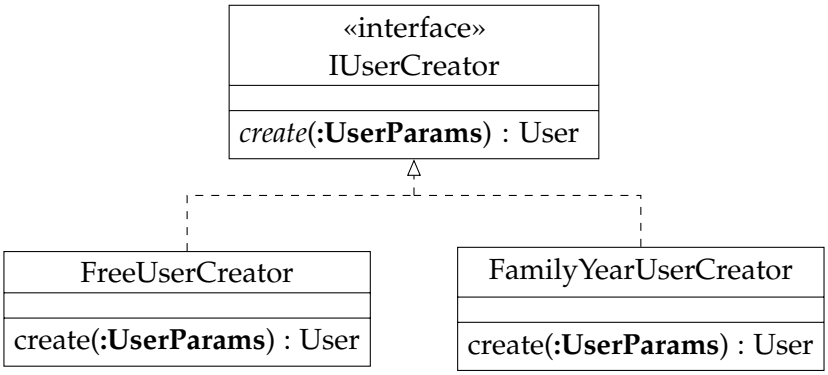


Abbildung 5.15: Implementierungshierarchie von IUserCreator mit Parameterobjekt

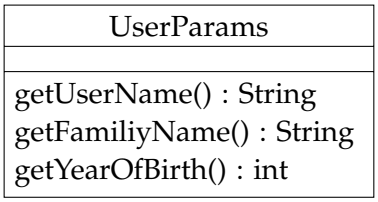


Abbildung 5.16: Die Parameterobjekt-Klasse UserParams

```
public class FreeUserCreator implements IUserCreator{
    public User createNewUser(UserParams params) {
        String name = params.getUserName();
        return new User(name);
    }
}

public class FamilyYearUserCreator implements IUserCreator {
    public User createNewUser(UserParams params) {
        String name = params.getFamiliyName();
        int yearOfBirth = params.getYearOfBirth();
        return new User(name + yearOfBirth);
    }
}
```

Nun haben wir ein System von Klassen, die nach wie vor in einer Vererbungshierarchie stehen, aber nicht mehr vom Servlet-Rahmenwerk abhängen. Sind nun auch nur noch essentielle Abhängigkeiten vorhanden? Nein, denn die Parameterobjekte vom Typ UserParams enthalten mehr Daten, als die jeweiligen Definitionen von *createNewUser()* benötigen.

Wir können aber klarer machen, welche essentiellen Abhängigkeiten die Methoden haben. Dazu folgen wir der Technik aus Abschnitt 3.3.1 und machen die Vorbedingungen ausdrücklich. Dazu erweitern wir die Klasse UserParams um sondierende Methoden, die angeben, ob der jeweils gewünschte Parameter von dem entsprechenden Objekt zurückgegeben werden kann. Dann können wir schreiben:

```
public class FreeUserCreator implements IUserCreator {
    /**
     * @require params.hasUserName()
     */
    public User createNewUser(UserParams params) {
        // Implementierung bleibt gleich
    }
}

public class FamilyYearUserCreator implements IUserCreator {
    /**
     * @require params.hasFamiliyName()
     * @require params.hasYearOfBirth()
     */
    public User createNewUser(UserParams params) {
        // Implementierung bleibt gleich
    }
}
```

Es bietet sich also an, eine Kombination der Techniken »Parameterobjekt verwenden« (Abschnitt 3.3.3) und »Sicheres Ändern der Signatur einer vielfach überschriebenen Methode« (Abschnitt 5.3) einzusetzen.

5.4.1 Erstellen des Parameterobjektes

Nachdem nun die Schnittstelle IUserCreator und die von ihr abgeleiteten Klassen keine Abhängigkeiten von HttpServletRequest mehr haben, muss

an den Stellen im Quelltext, an denen Methode *createNewUser()* aufgerufen wird, statt einem HttpServletRequest-Objekt ein UserParams-Objekt übergeben werden. Dazu muss das UserParams-Objekt mit Anfrageparametern (Abschnitt 2.8.1) aus dem HttpServletRequest-Objekt befüllt werden. Das Wissen darüber, welcher Anfrageparameter zu welchem Eintrag im Parameterobjekt passt, sollte an genau einer Stelle im Quelltext hinterlegt sein. Es bietet sich daher an, eine Parameterobjekt-Fabrik zu entwickelt, die diese Aufgabe übernimmt. Solche Fabrik hat typischerweise einen Konstruktor, der ein Objekt vom Typ HttpServletRequest erwartet und eine Methode *create()* die ein Parameterobjekt zurückliefert. Für unser Parameterobjekt aus Abbildung 5.16 könnte solche Fabrik folgendermaßen implementiert sein:

```
public class UserParamFactory {
    private HttpServletRequest _req;

    /**
     * @require req != null
     * @require isParsableInt(req.getParameter("Year_Of_Birth"))
     */
    public UserParamCreator(HttpServletRequest req) {
        _req = req;
    }

    /**
     * @ensure $result != null
     */
    public UserParams create() {
        UserParams result = new UserParams();
        result.setUsername(_req.getParameter("Username"));
        result.setFamily(_req.getParameter("Family_Name"));
        int yob = Integer.parseInt(
            _req.getParameter("Year_Of_Birth"));
        result.setYearOfBirth(yob);
        return result;
    }
}
```

Die Fabrik kann dann an den Stellen des Systems verwendet werden, an denen eine Umwandlung von HttpServletRequest in UserParams notwendig ist. Typischerweise ist dies in einem Servlet der Fall.

Die oben definierte Fabrik ist natürlich an das Servlet-Rahmenwerk gekoppelt. Soll die Oberflächentechnologie ausgetauscht werden, so muss eine zur neuen Technologie passende (und von dieser Technologie abhängige) Fabrik definiert werden. Manchmal ist es dann sinnvoll, eine Schnittstelle zu definieren, die von den konkreten Fabrikklassen implementiert wird. Dies könnte beispielsweise wie in Abbildung 5.17 erfolgen. In dieser Abbildung

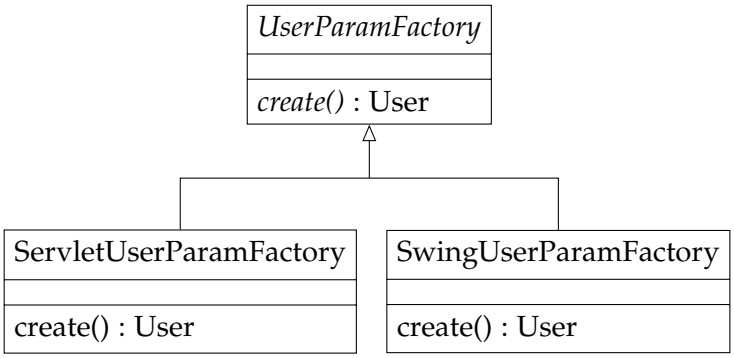


Abbildung 5.17: UserParamFactory und abgeleitete Klassen

ist neben der obigen Fabrik ein Klasse SwingUserParamFactory, die statt mit der Oberflächentechnlogie Servlets mit der Oberflächentechnologie Swing arbeitet.

5.5 Übertragbarkeit

So wie wir das zweite Problemfeld hier betrachtet haben, präsentiert es sich als unabhängig vom ersten. Es scheint, dass es in der Entwicklung von allen möglichen Systemen der Fall sein könnte, dass es eines Tages gewünscht ist, die Signatur einer überschriebenen Methode zu ändern. Die oben entwickelte Technik funktioniert gut, wenn eine Methode nur ein- oder zweimal überschrieben wurde. Ihren vollen Nutzen entfaltet sie aber erst dann, wenn die Signatur einer zehn-, zwanzig- oder hundertfach überschriebenen Methode geändert werden soll. Es stellt sich hierbei die Frage, in welchen Systemen solche Methoden auftreten. Wir vermuten, dass solche Methoden

häufig in Systemen auftreten, in denen Klassen existieren, von denen zur Laufzeit nur ein Exemplar existiert, ähnlich Singletons (Gamma u. a. 1996, S. 157), die aber als Parameter übergeben werden müssen. Insbesondere, wenn sie zugreifbare Daten enthalten, also wie Halden für globale Variablen wirken (Abschnitt 3.2.8). Um diese Vermutung zu stützen, reicht leider die Zeit im Rahmen einer Diplomarbeit nicht aus.

Wenn diese Vermutung richtig wäre, so wäre das in diesem Kapitel beschriebene Problemfeld typisch für Anwendungen, die das Servlet-Rahmenwerk verwenden. Dieses Rahmenwerk enthält ja mit HttpServletRequest, HttpServletResponse und HttpSession genau solche Klassen, von denen zur Laufzeit nur ein Exemplar existiert. Sie wäre weiterhin typisch für Rahmenwerke, die ähnliche Strukturen haben. Beispiele könnten Datenbank-Rahmenwerke sein, die so verwendet werden, dass es nur eine Datenbank-Verbindung gibt und diese in einer Klasse implementiert ist, von der es dann zur Laufzeit nur ein Exemplar gäbe. Wird der Standard Java Data Objects (JDO) verwendet, ist die Klasse javax.jdo.PersistenceManager (Jordan u. Russell 2003, S. 11) ein Kandidat für solche Art der Verwendung.

5.6 Zusammenfassung

In diesem Kapitel haben wir untersucht, warum das Ändern der Signatur von überschriebenen Methoden aufwendig ist und wie es sich auf eine sichere Art durchführen lässt. Wir haben dies Problem zunächst isoliert und dann in Kombination mit der Kopplung an das Servlet-Rahmenwerk beleuchtet.

Wir begannen mit einer Betrachtung, in welche Arten sich Refactorings einteilen lassen. Fowler (2000) folgend unterschieden wir zwischen Änderungen der Implementierung, der öffentlichen Schnittstelle und der veröffentlichten Schnittstelle. Die Änderungen der Signatur einer Methode unterschieden wir zwischen der Änderung einer überschriebenen oder nicht-überschriebenen Methode. Soll die Signatur einer nicht-überschriebenen Methode geändert werden, so müssen eine Methodendefinition und mehrere Methodenaufrufe geändert werden. Soll die Signatur einer überschriebenen Methode geändert werden, müssen zusätzlich mehrere Methodendefinitionen angepasst werden, deshalb sind solche Änderungen aufwendiger.

Um diese aufwendige Änderung in sicheren (im Sinne von Definition 2.8), kleinen Schritten durchführen zu können, haben wir ein Verfahren entwi-

ckelt und beschrieben. Dabei wird zunächst in der Oberklasse eine neue Methode definiert, die die gewünschte Signatur hat. Die Methode mit der alten Signatur wird so umgestellt, dass sie an die neue Methode weiterleitet. Jetzt werden nacheinander die Unterklassen so umgestellt, dass sie statt der Methode mit der alten Signatur die Methode mit der neuen Signatur implementieren. Das System ist nach jedem Schritt übersetzbar und lauffähig. Wenn alle Unterklassen umgestellt sind und alle Aufrufe auf die Methode mit der neuen Signatur umgebogen wurden, wird die Methode mit der alten Signatur aus der Oberklasse gelöscht und der Name der Methode mit der neuen Signatur wird auf den Namen der Methode mit der alten Signatur geändert.

Wir haben das Problemfeld in Kombination mit der Kopplung an das Servlet-Rahmenwerk betrachtet, weil in JCommSy Typen aus dem Rahmenwerk in der Signatur von überschriebenen Methoden verwendet wird und diese Verwendung beseitigt werden soll. Hierfür lässt sich eine Kombination der Lösungen aus diesem und Kapitel 3 verwenden. Dabei muss beachtet werden, dass die überschreibenden Methoden die gleiche Signatur behalten müssen, wie die überschriebene Methode. Aus diesem Grund empfiehlt sich die Verwendung eines Parameterobjektes.

Wir vermuten, dass das hier beschriebene Phänomen einer Methode, die oft überschrieben wird, häufig mit Rahmenwerken auftritt, die Klassen haben, von denen zur Laufzeit nur ein Exemplar existiert. Wäre diese Vermutung richtig, dann ließen sich das beschriebene Verfahren zur Änderung von überschriebenen Methoden zur Entflechtung von Anwendungssoftware mit solchen Rahmenwerken verwenden.

6 Die Entflechtung von JCommSy und Servlet-Rahmenwerk

In den vorangegangenen Kapiteln untersuchten wir die Probleme von zu enger Kopplung an das Servlet-Rahmenwerk und schlugen einen Ansatz zur Entflechtung vor (Kapitel 3). Wir wendeten den Ansatz an der Webanwendung JCommSy an und stellten fest, dass er bei diesem System nicht ausreicht, weil die Kopplung hier in der Signatur von überschriebenen Methoden steckt (Kapitel 4). In Kapitel 5 haben wir das Ändern der Signatur einer überschriebenen Methode zunächst isoliert und dann kombiniert mit Rahmenwerkskopplung studiert und ein sicheres und kleinschrittiges Verfahren für das Ändern solcher Methoden vorgeschlagen. Nun wollen wir zum System »JCommSy« zurückkehren und untersuchen, ob es sich mit der Kombination der beiden vorgeschlagenen Verfahren vom Servlet-Rahmenwerk lösen lässt.

6.1 Rückblick und neuer Lösungsansatz

Zur Erinnerung: In Kapitel 4 stellten wir fest, dass JCommSy zu eng an das Servlet-Rahmenwerk gekoppelt ist. Konkret handelt es sich hierbei um das Architekturelement der »Ausstatter«. Das System soll deshalb umgebaut werden. Ziel des Umbau ist das vollständige Loslösen des Architekturelements »Ausstatter« von dem Rahmenwerk. Bei diesem Umbau handelt es sich um ein *Großes Refactoring* (Definition 2.7).

In Abbildung 4.10 auf Seite 66 wurde die abstrakte Klasse FragmentOutfitter mit ihren Einschubmethoden gezeigt. Von FragmentOutfitter sind 80 Unterklassen abgeleitet, die alle die Einschubmethoden implementieren. Drei dieser Einschubmethoden haben Parameter der Typen HttpServletRequest und HttpServletResponse, d. h. sind abhängig vom Servlet-Rahmenwerk. Es handelt sich um die Methoden *createOutfitters()*, *createFragment()* und *updateFragment()*. Unser erster Enflechtungsversuch (Abschnitt 4.5), der nur die Lösungsschritte gegen Rahmenwerkskopplung aus Abschnitt 3.3

verwendete, scheiterte, weil die Rahmenwerkskopplung in der Signatur der Einschubmethoden steckt.

Um diese Abhängigkeit zu lösen, wenden wir deshalb in diesem Kapitel die in Abschnitt 5.4 vorgeschlagene Kombination der Techniken »Parameterobjekt verwenden« (Abschnitt 3.3) und »Sicheres Ändern der Signatur einer vielfach überschriebenen Methode« (Abschnitt 5.3) an. Die drei Methoden sollen so geändert werden, dass ihr Parameter-Profil statt Parametern der Typen HttpServletRequest und HttpServletResponse einen Parameter vom Typ ParameterBean enthält (Abbildung 6.1). Wir folgen dem Muster

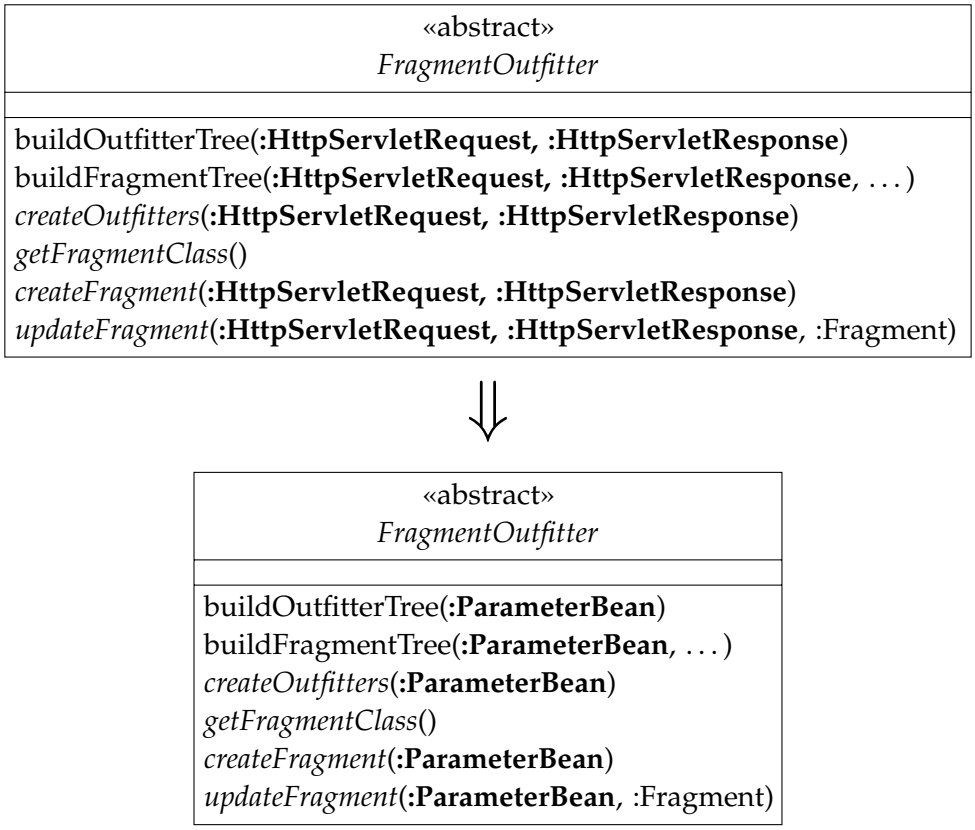


Abbildung 6.1: Änderung der Klasse FragmentOutfitter (Legende in Abschnitt 2.9)

aus Abschnitt 5.3, um die Übersetzungs- und Lauffähigkeit des Systems während des Refactorings aufrechtzuhalten. Das heißt, dass wir in kleinen Schritten vorgehen werden. Zunächst wird die Oberklasse FragmentOutfitter

vorbereitet (Abschnitt 6.2), dann jeweils eine Unterklasse zur Zeit umgebaut. Wir betrachten hier beispielhaft nur eine Unterklasse (Abschnitt 6.3). Schließlich muss die Oberklasse nachbereitet werden (Abschnitt 6.4). Da wir die Signatur von drei vielfach überschriebenen Methoden ändern wollen, die in denselben Klassen definiert sind, führen wir die Einzelschritte jeweils für die drei Methoden gemeinsam aus.

6.2 Vorbereitung

Wir beginnen mit dem Einführen von Methoden mit der neu gewünschten Signatur in die Oberklasse FragmentOutfitter (vgl. Abschnitt 5.3.1) und erhalten dann eine Klasse wie in Abbildung 6.2. Als Suffix für die Methoden,

«abstract» *FragmentOutfitter*
buildOutfitterTree(:HttpServletRequest, :HttpServletResponse) buildFragmentTree(:HttpServletRequest, :HttpServletResponse, . . .) createOutfitters(:HttpServletRequest, :HttpServletResponse) **createOutfitters_NOJEE(:ParameterBean)** *getFragmentClass()* createFragment(:HttpServletRequest, :HttpServletResponse) **createFragment_NOJEE(:ParameterBean)** updateFragment(:HttpServletRequest, :HttpServletResponse, :Fragment) **updateFragment_NOJEE(:ParameterBean, :Fragment)**

Abbildung 6.2: Einführen neuer Methoden mit gewünschter Signatur

die die neue Signatur haben, verwenden wir statt »_NEU« hier »_NOJEE« um anzudeuten, dass keine Abhängigkeit mehr von JEE gewünscht ist. Als Implementierung ergibt sich z. B. für *createOutfitters()*:

```
public abstract class FragmentOutfitter {
    //  ...
    @Deprecated
    protected void  createOutfitters (
            HttpServletRequest req, HttpServletResponse resp)
            throws CommsyException {
```

```
        ParamBean parameters =
            new ParamBeanCreator(req).create();
        createOutfitters_NOJEE(parameters);
    }
    protected void createOutfitters_NOJEE(
            ParamBean parameters)
            throws CommsyException {
        throw new UnsupportedOperationException(
            "createOutfitters_NOJEE() should be overwritten.");
    }
}
```

Die Klausel throws CommsyException wird angegeben, da überschreibende Methoden Ausnahmen des Typs CommsyException werfen können. Die hier verwendete Klasse ParamBeanCreator ist eine Fabrik, die dem in Abschnitt 5.4.1 beschriebenen Muster folgt. Wir implementieren zunächst nur einen Rahmen:

```
public class ParamBeanCreator {
    private HttpServletRequest _req;

    /**
     * @require req != null
     */
    public ParamBeanCreator(HttpServletRequest req) {
        _req = req;
    }

    /**
     * @ensure $result != null
     */
    public ParamBean create() {
        ParamBean parameters = new ParameterBean();
        return parameters;
    }
}
```

So erzeugt die Methode *create()* aus ParameterBeanCreator ein leeres ParameterBean-Objekt, in das noch keine Daten gefüllt werden. Welche Daten aus

der Variable _req in das ParameterBean-Objekt übertragen werden müssen, zeigt sich erst im nächsten Schritt.

Damit die Klassen FragmentOutfitter und ParamBeanCreator übersetzt werden können, benötigen wir auch eine Implementierung der Parameterobjekt-Klasse. Sie ist zunächst leer:

```
public class ParamBean {
}
```

Welche Felder und Zugriffsmethoden diese Klasse erhält, wird sich ebenfalls erst im nächsten Schritt zeigen. Parameterobjekt-Klasse und Parameterobjekt-Erzeuger-Klasse werden sich parallel entwickeln.

Nun ist die Vorbereitung der Ober- und Hilfsklassen abgeschlossen und wir können uns an die Anpassung der Unterklassen machen. Als Beispiel verwenden wir die Klasse AnnotationActionsOutfitter.

6.3 Beispielhafte Umstellung der Unterklasse AnnotationActionsOutfitter

Als erstes wird der zu der Klasse gehörige Komponententest (Definition 2.4) umgestellt (bzw. wenn, wie in diesem Fall, bisher kein Test existierte, erstellt). Dort wird statt der HttpServletRequest- und HttpServletResponse-Objekte nun ein Objekt vom Typ ParameterBean vorbereitet und an die zu testende Methode übergeben.

Dann soll die erste Methode angepasst werden. Wir betrachten aus Platzgründen nur die Methode *updateFragment()* und aus ihr auch nur die Ausschnitte, die für das Refactoring relevant sind.[1] Die bisherige Implementierung von *updateFragment()* unter Verwendung von HttpServletRequest und HttpServletResponse sieht folgendermaßen aus:

```
protected void updateFragment(HttpServletRequest req,
        HttpServletResponse resp, Fragment frag) {
    //  ...
    ItemID refItemID = ItemID.selectID(req.getParameter("ref_iid" ));
    User user = (User) req.getSession(). getAttribute ("user");
    //  ...
}
```

[1] Außerdem sind diese Ausschnitte vereinfacht dargestellt, um das Wesentliche nicht in für diese Abhandlung unwichtigen Quelltextzeilen untergehen zu lassen.

Die Methode liest also den Anfrageparameter (Abschnitt 2.8.1) »ref_iid« und das Sitzungsattribut »user« aus. Nebenbei bemerkt: In dieser Methode treten beide Arten des Typsicherheitsverlusts aus Abschnitt 3.2.5 auf: sowohl *parsing* als auch *cast* sind notwendig.

Damit wir *updateFragment()* in AnnotationActionsOutfitter durch *updateFragment_NOJEE* ersetzen können, müssen wir die Parameterobjekt-Klasse so anpassen, dass sie die von *updateFragment()* benötigten Daten bereitstellt. Wir zeigen hier nur den für das Sitzungsattribut »user« zuständigen Teil, die Handhabung des Anfrageparameters »ref_iid« verläuft nach dem gleichen Muster:

```
public class ParamBean {
    private User _user;

    public boolean hasUser() {
        return _user != null;
    }

    /**
     * @require user != null
     * @ensure hasUser()
     */
    public void setUser(User user) {
        _user = user;
    }

    /**
     * @require hasUser()
     * @ensure $result != null
     */
    public User getUser() {
        return _user;
    }
}
```

Für einen Anfrageparameter oder ein Sitzungsattribut werden also jeweils ein Datenfeld und drei Methoden in ParamBean eingeführt. Die Klasse ParamBean hat allerdings kein Wissen über Anfrageparameter oder Sitzungsattribute und ihre Exemplare können auch mit Daten befüllt werden,

die nicht aus Anfrageparametern oder Sitzungsattributen stammen.

Damit die Parameterobjekte auch wie gewünscht mit den Werten aus den passenden Anfrageparametern oder Sitzungsattributen befüllt werden können, passen wir die Methode *create()* in der Fabrik an:

```
public class ParamBeanCreator {
    // ...
    public ParameterBean create() {
        ParameterBean parameters = new ParameterBean();
        parameters.setUser(
            (User) _req.getSession().getAttribute("user"));
        parameters.setREF_IID(_req.getParameter("ref_iid"));
        return parameters;
    }
}
```

Die Methode *create()* liest also die Anfrageparameter und Sitzungsattribute aus und schreibt sie in die entsprechenden Datenfelder des Parameterobjektes. Wir haben dieses Beispiel gewählt, weil es das Auslesen sowohl eines Anfrageparameters als auch eines Sitzungsattributes zeigt.

Nun haben wir die Hilfsklassen in die benötigte Form gebracht und können uns schließlich der eigentlich zu ändernden Unterklasse AnnotationActionsOutfitter zuwenden. Wir nennen die Methode *updateFragment()* in *updateFragment_NOJEE()* um und passen sie an. Es ergibt sich die Implementierung:

```
protected void updateFragment_NOJEE(
        ParameterBean pb, Fragment frag) {
    // ...
    refItemID = pb.getRefItemID();
    User user = pb.getUser();
    // ...
}
```

Statt aus der Anfrage oder der Sitzung werden die benötigten Daten nun also aus dem Parameterobjekt ausgelesen.

Nachdem die Methode *updateFragment()* umgestellt wurde, führen wir die gleichen Schritte für die anderen vom Servlet-Rahmenwerk abhängigen Einschubmethoden aus. Als Ergebnis ist die Klasse AnnotationActionsOutfitter von ihren Beziehungen zum Servlet-Rahmenwerk befreit.

Nach AnnotationActionsOutfitter müssen noch die 79 anderen Unterklassen von FragmentOutfitter umgestellt werden. Bei diesen kann es auch vorkommen, dass nicht nur lesend – wie in der behandelten Implementierung von *updateFragment()* – sondern auch schreibend auf die Parameter der Typen HttpServletRequest und HttpServletResponse zugegriffen wird. In solchen Fällen kann das Parameterobjekt gewissermaßen umgekehrt verwendet werden. Das heißt in der entsprechenden Implementierung von *updateFragment()* wird in das Parameterobjekt geschrieben statt gelesen.

Es muss dann sichergestellt werden, dass an den Stellen im Quelltext, an denen die Methode aufgerufen wird, direkt nach diesem Aufruf die Daten aus dem Parameterobjekt ausgelesen und in das HttpServletRequest- bzw. HttpServletResponse-Objekt geschrieben werden. Es bietet sich an, dieses Speichern in HTTP-Anfrage und -Antwort in eine eigene Methode in der Parameterobjekt-Fabrik zu kapseln. Diese Methode könnte z. B. *save()* heißen und ist komplementär zu *create()* zu sehen. Als Beispiel zeigen wir die Implementierung von *save()*, nachdem das Speichern der ItemID des letzten besuchten Raumes in sie verschoben wurde:

```
public class ParamBeanCreator {
    // ...
    public void save(ParamBean parameters) {
        if (parameters.hasLastRoomID()) {
            _req.getSession().setAttribute(
                "lastRoom",
                parameters.getRoom().getItemID());
        }
    }
}
```

Wie wir lesen, wird hier nicht direkt in das HttpServletRequest-Objekt geschrieben, sondern in das über dieses verfügbare HttpSession-Objekt, das die HTTP-Sitzung repräsentiert. Das ItemID-Objekt wird unter dem Namen »lastRoom« als Sitzungsattribut gespeichert. Diese Speicherung wird nur vorgenommen, wenn der letzte Raum vorher im Parameterobjekt gesichert wurde.

Mit den vorgestellten Schritten sollten alle Ausstatterunterklassen umzustellen sein. Je nachdem, ob sie lesend oder schreibend auf die Objekte der Typen aus dem Servlet-Rahmenwerk zugreifen, kann das Parameterobjekt als Speicher für eingehende oder ausgehende Daten verwendet werden.

6.4 Nachbereitung

Nachdem alle Unterklassen von FragmentOutfitter umgestellt sind, befinden sich in ihnen allen Implementierungen der drei Einschubmethoden *createOutfitters_NOJEE(), createFragment_NOJEE()* und *updateFragment_NOJEE()*. Nun müssen zunächst die Aufrufe der drei Einschubmethoden *createOutfitters(), createFragment()* und *updateFragment_NOJEE()* umgestellt werden. Es soll jetzt jeweils die »_NOJEE«-Variante aufgerufen werden. Da die Einschubmethoden jeweils nur an einer Stelle aufgerufen werden (es sind eben Einschubmethoden), ist dies einfach. Bisher wurden HttpServletRequest- und HttpServletResponse-Objekte weitergegeben:

```
public final void buildOutfitterTree (
        HttpServletRequest req, HttpServletResponse resp)
        throws CommsyException {
    createOutfitters (req, resp);
    // ...
}
```

Nun lassen wir uns ein ParamBean-Objekt von der Parameterobjekt -Fabrik erzeugen. Dann wird die »_NOJEE«-Variante aufgerufen und schließlich werden die gewünschten Daten aus dem Parameterobjekt in HTTP-Anfrage und -Sitzung gespeichert:

```
public final void buildOutfitterTree (
        HttpServletRequest req, HttpServletResponse resp)
        throws CommsyException {
    ParamBean parameters =
        new ParamBeanCreator(req).create();
    createOutfitters _NOJEE(parameters);
    new ParamBeanCreator(req).save();
    // ...
}
```

Nun können wir das Refactoring »Methode umbenennen« (Fowler 2000, S. 279 ff) benutzen, um *updateFragment_NOJEE()* wieder in *updateFragment()* umzubenennen. Auf die gleiche Weise werden die beiden anderen Einschubmethoden von FragmentOutfitter behandelt.

Dann können wir auch die Methode *buildOutfitterTree()* von ihrer akzidentiellen Abhängigkeit von HttpServletRequest und HttpServletResponse

befreien. Hierzu verschieben wir den Einsatz der Parameterobjekt-Fabrik in der Aufrufhierarchie nach oben. Er erfolgt dann nicht mehr in FragmentOutfitter, sondern in der einzigen Klasse, die *buildOutfitterTree()* aufruft, nämlich FragmentServlet. Entsprechend wird dann dort der Aufruf von *buildOutfitterTree()* von Aufrufen der beiden Methoden *create()* und *save()* aus der Parameterobjekt-Fabrik umgeben. Die Klasse FragmentServlet befindet sich im Architekturelement »Servlets«; sie hat essentielle Abhängigkeiten zum Servlet-Rahmenwerk.

Jetzt sind wir am Ziel angelangt; alle Klassen aus dem Architekturelement »Ausstatter« verwenden nun keine Klassen mehr aus dem Servlet-Rahmenwerk und sind damit auch nicht mehr abhängig von ihm.

6.5 Zusammenfassung

In diesem Kapitel wendeten wir die in Kapitel 5 vorgeschlagene Kombination der Lösungsansätze gegen Rahmenwerkskopplung in der Signatur von überschriebenen Methoden an unserem Beispielsystem JCommSy an. Es konnte dadurch in weiten Teilen von der Abhängigkeit zum Servlet-Rahmenwerk befreit werden.

Zunächst blickten wir darauf zurück, dass JCommSy nicht allein mit dem Verfahren aus Kapitel 3 entkoppelt werden konnte. Deshalb wendeten wir nun die Kombination von »Parameterobjekt verwenden« mit »Sicheres Ändern der Signatur einer vielfach überschriebenen Methode« an. Wir begannen das Große Refactoring mit der Vorbereitung der Ausstatteroberklasse FragmentOutfitter. Diese definiert drei Methoden, die alle die Typen HttpServletRequest und HttpServletResponse verwendeten. Um das Vorgehen zu beschleunigen, bearbeiteten wir in jedem Schritt die drei Methoden parallel. Wir zeigten beispielhaft die Umstellung der Unterklasse AnnotationActionOutfitter. Dabei betrachteten wir die unterschiedlichen Lösungen für einerseits lesenden und andererseits schreibenden Zugriff auf die Typen aus dem Servlet-Rahmenwerk. Die anderen Unterklassen von FragmentOutfitter wurden nach dem gleichen Muster umgestellt. Schließlich zeigten wir die nachbereitende Umstellung der Oberklasse FragmentOutfitter, die damit vollständig von Abhängigkeiten zum Servlet-Rahmenwerk befreit wurde.

7 Ergebnis

Die vorliegende Arbeit verfolgte zwei Ziele. Erstens sollten Refactorings gefunden werden, mit denen die Kopplung an ein Rahmenwerk gelockert werden kann. Der Schwerpunkt sollte auf das Servlet-Rahmenwerk gelegt werden. Zweitens sollten die gefunden Refactorings an der Webanwendung JCommSy angewendet werden und der dadurch angestoßene Transformationsprozess zu einer saubereren Architektur dieses System führen. Damit sollte die Praxistauglichkeit der Refactorings gezeigt werden. In diesem Kapitel werden wir darstellen, inwieweit diese Ziele erreicht wurden.

In Kapitel 3 haben wir ein Verfahren zum Entflechten von Anwendung und Rahmenwerk vorgeschlagen. Dieses Verfahren wendeten wir in Kapitel 4 an JCommSy an. Es zeigte sich, dass es für das Entflechten von JCommSy und Servlet-Rahmenwerk noch nicht ausreichend war, weil die Rahmenwerkskopplung in den Signaturen von überschriebenen Methoden steckte. In Kapitel 5 verbesserten wir daher das Entflechtungsverfahren durch eine Strategie zum Ändern solcher Signaturen. Das verbesserte Verfahren konnte in Kapitel 6 genutzt werden, um Anwendung und Rahmenwerk zu entflechten.

Die Refactorings konnten in die von Fowler (2000) vorgeschlagene Form gebracht und in kleinen Schritten beschrieben werden. Den Katalog dieser Beschreibungen präsentieren wir in Abschnitt 7.2. Vorher beleuchten wir in Abschnitt 7.1 jedoch, inwieweit die Entflechtung von JCommSy erfolgreich war.

Ein interessantes Nebenergebnis ist, dass es sich auszahlte, nicht nur in der Theorie nach Refactorings zu suchen, sondern sie auch in der Praxis anzuwenden. Erst dadurch, dass die *a priori* entwickelten Refactorings an einem konkreten Beispiel ausprobiert wurden, konnte das zweite Problemfeld ins Bewusstsein rücken. Die Lösung hat sich also iterativ entwickelt. Das entwickelte Refactoring »Ändere die Signatur einer überschriebenen Methode« ist auch in Zusammenhängen ohne Rahmenwerkskopplung einsetzbar.

7.1 Entflechtung JCommSy vom Servlet-Rahmenwerk

Nach eingehender Analyse der Architektur der Anwendung JCommSy konnte ein Plan für ein Großes Refactoring (Definition 2.7) des Systems entworfen werden. Mithilfe dieses Planes konnte der Entflechtungsprozess angefangen und in entscheidenden Teilen durchgeführt werden.

Das System JCommSy hat 410 Klassen mit insgesamt ca. 40 000 Programmzeilen (engl. *lines of code*). Vor dem Umbau hatten 96 dieser Klassen Abhängigkeiten (Definition 2.12) von Klassen aus dem Servlet-Rahmenwerk. Diese abhängigen Klassen umfassten insgesamt ca. 15 000 Programmzeilen. Die Untersuchung ergab, dass 85 Klassen, die ca. 12 000 Programmzeilen umfassten, akzidentiell abhängig waren. Hierbei handelte es sich um die Ausstatterklassen (Abschnitt 4.3.1) und einige Hilfsklassen. Von diesen konnten im Laufe der Arbeit 80 von ihren akzidentiellen Abhängigkeiten befreit werden, für die verbleibenden 5 mit ca. 1700 Zeilen reichte die Zeit nicht aus. Diese Daten sind noch einmal kompakt in Tabelle 7.1 dargestellt. Die

Tabelle 7.1: Kopplung JCommSy an Servlet-Rahmenwerk

	Programmzeilen	Klassen
Insgesamt	$\approx$ 40k	415
abhängig (vorher)	$\approx$ 15k	101
akzidentiell abhängig (vorher)	$\approx$ 12k	80
essentiell abhängig	$\approx$ 3k	21
abhängig (nachher)	$\approx 4,7$k	26
akzidentiell abhängig (nachher)	$\approx 1,7$k	5

akzidentiellen Abhängigkeiten zum Servlet-Rahmenwerk konnten in weiten Teilen gelöst werden. Vor der Entflechtung befanden sich fast ein Drittel der Quelltextzeilen in Klassen, die vom Servlet-Rahmenwerk abhängig waren. Danach befinden sich nur noch etwas mehr als ein Neuntel der Quelltextzeilen in abhängigen Klassen. Sobald alle akzidentiellen Klassen befreit sind, wird es weniger als ein Zehntel sein.

Die aus dem Umbau erwachsenen Erkenntnisse konnten in eine Form gebracht werden, in der sie auch für andere Projekte in anderen Umfeldern verwendbar sind: einen Katalog von Refactorings, der sich in Abschnitt 7.2 befindet.

Dadurch, dass das Umbauverfahren auf eine einfache Art (eben als Refactoring) beschrieben werden konnte, konnten für den verbleibenden Teil des Umbaus leicht Anweisungen formuliert werden, wie die weiteren Teile des Systems umgebaut werden sollen. Das Projekt, in dem JCommSy entwickelt wird, setzt das Vorgehensmodell »Extreme Programming« (Beck u. Andres 2005) ein, entsprechend wurden die Anweisungen als *story cards* formuliert.

7.1.1 Verbesserungen

Die Anwendung JCommSy hatte die Probleme, die in Abschnitt 3.2 beschrieben werden. Da die Technik »Parameterobjekt verwenden« (Abschnitt 3.3.3) eingesetzt wurde, konnten diese Probleme gelindert werden.

Daraus ergibt sich eine Reihe von Verbesserungen. So ist es nun leichter möglich, die Oberflächentechnologie von JCommSy auszutauschen. Die Ausstatterklassen (Abschnitt 4.3.1) haben nun kein Wissen mehr über die Namen von Anfrageparametern (Definition in Abschnitt 2.8.1). Die Ausstatterklassen werden nun durch herkömmliche JUnit-Testklassen getestet und benötigen keine Cactus-Tests mehr. Die Schnittstelle der geänderten Methoden der Ausstatter sind klarer geworden; sie erwarten nun keine HttpServletRequest- und HttpServletResponse-Objekte mehr, sondern jeweils ein Exemplar der für diesen Zweck eingeführten Parameterobjekt-Klasse ParamBean. Zusätzlich wird durch Vorbedingungen bei jeder Methode ausgedrückt, welche Felder des ParamBean-Objektes jeweils gesetzt sein müssen. Die benötigten Daten werden von der Klasse ParamBean streng typisiert zur Verfügung gestellt und es muss nicht wie vorher eventuell *parsing* oder *casting* ausgeführt werden. Die Schnittstelle der Methoden ist nun schmaler.

Allerdings sind folgende Probleme geblieben: Die Ausstatter können zustandsverändernden Operationen an dem jeweiligen ParamBean-Objekt aufrufen. Weil zu einem Programmdurchlauf nur ein Exemplar der Klasse ParamBean existiert und dieses Exemplar an alle Ausstatter weitergeben wird, wirkt dieses nun wie eine Halde für globale Variablen. Wie auch diese Probleme gelöst werden können beschreibt Abschnitt 3.3.

7.1.2 Fazit

Das Ziel JCommSy vom Servlet-Rahmenwerk zu entkoppeln, d. h. akzidentielle Abhängigkeiten zu Typen aus dem Servlet-Rahmenwerk zu entfernen

wurde fast vollständig erreicht. Wichtig ist, dass das grundlegenden Verfahren entwickelt und ausprobiert wurde. Die verbleibenden Schritte sind bekannt und einfach auszuführen. Ausschließlich aus Zeitgründen war es nicht möglich, sie im Rahmen dieser Arbeit auszuführen.

7.2 Refactoring-Katalog

Martin Fowler hat in seinem Buch »Refactoring« (Fowler 2000) einen Katalog von Refactorings erstellt und dabei auch ein Standardformat zum Beschreiben von Refactorings definiert (Fowler 2000, S. 99). Die Bestandteile dieser Beschreibung sind Name (engl. *name*), Zusammenfassung (*summary*), Motivation (*motivation*), Vorgehensweise (*mechanics*) und eventuell Beispiele (*examples*). Die im Laufe der Arbeit neu entdeckten Refactorings werden im Folgenden in diesem Format beschrieben. Beispiele lassen wir hier weg, da sie in den vorangegangenen Kapiteln beschrieben wurden.

Folgende Refactorings haben wir beschrieben:

- Ersetze technischen Datenklumpen durch Parameterobjekt (7.2.1)
- Führe Parameterobjekt-Erzeuger ein (7.2.2)
- Ändere die Signatur einer überschriebenen Methode (7.2.3)

7.2.1 Ersetze technischen Datenklumpen durch Parameterobjekt

Für eine Klasse, die von der Abhängigkeit zu einem technischen Datenklumpen (Fowler 2000, S. 74) befreit werden soll, wird eine Parameterobjekt-Klasse erzeugt. Für jedes Datum, das die Klasse aus dem Datenklumpen verwendet, wird in die Parameterobjekt-Klasse ein Attribut mit zugehörigen Zugriffsmethoden eingeführt. In der Klasse werden Datenklumpen-Objekte durch Parameterobjekte ersetzt. Methoden, die bisher Daten aus dem Datenklumpen verwendeten, werden so geändert, dass sie die entsprechende Zugriffsmethode der Parameterobjekt-Klasse aufrufen.

Der die Klasse verwendende Code muss so umgestellt werden, dass er nun statt Datenklumpen-Objekten Parameterobjekte übergibt. Für das Erzeugen eines Parameterobjektes aus einem Datenklumpen-Objekt bietet sich die Einführung eines Parameterobjekt-Erzeugers an (Abschnitt 7.2.2).

Motivation

Der fachliche Teil einer Anwendung soll nicht von Technologien abhängig sein.

Es gibt Rahmenwerke, die Datenklumpen bieten, um Parameter zu transportieren. Dies ist typisch für Anwendungen nach dem Anfrage-Antwort-Kontrollflussmodell. Solche Datenklumpen tragen daher oft Namen wie Request oder RequestContext.

Wenn ein technischer Datenklumpen in fachlichem Code verwendet wird, so wird dieser eng an das Rahmenwerk gekoppelt, aus dem der Datenklumpen stammt. Diese enge Kopplung ist unerwünscht, weil sie dazu führt, dass der Anwendungscode so nicht mit einer alternativen Technologie verwendet werden kann.

Die Kopplung an den Datenklumpen kann durch Einführen eines Parameterobjektes gelockert werden. Eine Parameterobjekt-Klasse kann sich später leicht zu einer Klasse mit fachlicher Verhaltensweise weiterentwickeln.

Vorgehensweise

1. Untersuche die Methode, aus deren Signatur der Datenklumpen entfernt werden soll. Für jedes Datum des Datenklumpen, dass in der Methode verwendet wird, erzeuge ein Feld mit dazugehöriger Gib- und Setze-Methode (engl. *getter* bzw. *setter*) in der Parameterobjekt-Klasse.

2. Kopiere die Methode und ändere die Signatur der Kopie so, dass sie statt eines Datenklumpen ein Parameterobjekt als Parameter erhält. Falls in einer Programmiersprache gearbeitet wird, die kein Überladen unterstützt, muss die neue Methode einen anderen Namen erhalten als die alte.

3. Ersetze die Zugriffe auf den Datenklumpen durch Aufrufe der Gib-Methoden des Parameterobjektes.

4. Ändere die alte Methode so, dass sie an die neue weiterleitet.

5. Wenn möglich: Markiere die alte Methode als veraltet (in Java z. B. mit @Deprecated).

6. Ändere alle Aufrufe der alten in Aufrufe der neuen Methode um.

7. Lösche die alte Methode.
8. Falls die neue Methode einen anderen Namen hat, als die alte: Benenne die neue Methode auf den Namen der alten um. Dazu bietet sich das Refactoring »Methode umbenennen« (Fowler 2000, S. 279 ff) an.

7.2.2 Führe Parameterobjekt-Erzeuger ein

Für eine Parameterobjekt-Klasse wird eine passende Erzeuger-Klasse entwickelt, die die passenden Daten für jedes Datenfeld des Parameterobjektes aus dem zugehörigen technischen Datenklumpen ausliest.

Motivation

In einer Anwendung, in der ein technischer Datenklumpen durch ein Parameterobjekt ersetzt wird, muss es eine Möglichkeit geben, ein Parameterobjekt aus einem Exemplar der Datenklumpenklasse zu erzeugen. Dazu bietet es sich an, eine Fabrikmethode (Gamma u. a. 1996, S.131 ff) zu entwickeln und diese in einer eigenen Klasse zu kapseln. Solche Klasse wird als Parameterobjekt-Erzeuger oder Parameterobjekt-Fabrik bezeichnet. Dieses Refactoring funktioniert nur in Kombination mit dem Refactoring »Ersetze technischen Datenklumpen durch Parameterobjekt« aus Abschnitt 7.2.1.

Vorgehensweise

1. Erzeuge die Parameterobjekt-Erzeuger-Klasse.
2. Gib ihr einen Konstruktor, der ein Exemplar der Klasse, die den technischen Datenklumpen repräsentiert, übergeben bekommt. Lasse den Konstruktor den Parameter in eine Exemplarvariable speichern.
3. Schreibe eine Fabrikmethode, die ein Exemplar der Parameterobjekt-Klasse zurückgibt.
4. Setze jedes Datenfeld des Parameterobjektes mithilfe der passenden Setze-Methode. Die benötigten Daten können über die Zugriffsmethoden an der Exemplarvariable, die den Datenklumpen speichert, ausgelesen werden. Wie das zu geschehen hat, findet sich im Quelltext, der bisher den technischen Datenklumpen verwendet hat und hinterher ein Parameterobjekt verwenden soll. Kopiere die entsprechenden Quelltextzeilen in die Fabrikmethode.

5. Wenn das zugehörige Refactoring »Ersetze technischen Datenklumpen durch Parameterobjekt« durchgeführt wurde, dann erzeuge überall dort wo es nötig ist, einen Parameterobjekt-Erzeuger, dem der technische Datenklumpen übergeben wird. Rufe an dem eben erzeugten Exemplar des Parameterobjekt-Erzeugers die Fabrikmethode zum Erzeugen des Parameterobjektes auf und gib es an die umgestellten Methoden statt des technischen Datenklumpens weiter.

7.2.3 Ändere die Signatur einer überschriebenen Methode

Um die Signatur einer überschriebenen Methode zu ändern, wird zunächst in der Oberklasse, d. h. in der Klasse, in der die überschriebene Methode definiert wird, eine neue Methode mit der gewünschten Signatur eingeführt. Die alte Methode wird so umgebaut, dass sie an die neue Methode umleitet. Nun können Schritt für Schritt die Unterklassen, d. h. die Klassen, in denen die überschreibenden Methoden definiert werden, umgestellt werden. Zum Schluss wird der aufrufende Quelltext so geändert, dass er die neue Methode verwendet. Die alte Methode kann nun gelöscht werden.

Das Refactoring basiert auf der Polymorphie. Durch sie wird sichergestellt, dass je nachdem, ob eine Unterklasse schon umgestellt wurde oder nicht, die passende Variante der Methode, d. h. die mit der alten oder die mit der neuen Signatur, aufgerufen wird.

Motivation

Es kann vorkommen, dass eine polymorphe Methode sehr häufig überschrieben wird. Oft handelt es sich hierbei um eine Einschubmethode, die nur einmal aufgerufen wird. Einschubmethoden sind abstrakt. Wir betrachten der Einfachheit halber zunächst diesen Sonderfall, dass die zu ändernde Methode in der Oberklasse abstrakt definiert ist.

Wenn die Signatur einer solchen Methode geändert werden soll, müssen auch die Signaturen aller Methoden angepasst werden, die sie überschreiben. Bei einer großen Zahl von Überschreibungen kann dies nicht einfach und sicher in einem Schritt geschehen. Dieses Refactoring bietet einen Mechanismus, um eine Situation herzustellen, in der Schritt für Schritt, Methode für Methode die einzelnen Definitionen umgestellt werden. Das System bleibt zwischen der Umstellung von zwei Methoden stabil. Es gibt Zeitpunkte, zu denen Methodendefinitionen sowohl mit der alten, als auch mit der neuen

Signatur parallel existieren. Das System ist auch zu diesen Zeitpunkten lauffähig.

Vorgehensweise

1. Erzeuge eine Methode in der Oberklasse mit der gewünschten neuen Signatur, die so heißt, wie die zu ändernde Methode mit der Zeichenkette »_NEU« angehängt.
2. Lasse die neue Methode eine Ausnahme (in Java typischerweise UnsupportedOperationException) werfen. Dies erfolgt, um sicherzustellen, dass die Methode nicht aufgerufen wird. Falls sie doch versehentlich aufgerufen wird, gibt es zur Laufzeit Rückkopplung über diesen Fehler.
3. Gib der bisher abstrakten alten Methode eine Implementierung, in der sie an die neue Methode weiterleitet. Bisher wird sie ja in jeder Unterklasse überschrieben und deshalb sollte diese Implementierung niemals aufgerufen werden.
4. Markiere die alte Methode als veraltet (in Java mit @Deprecated).
5. Führe an der ersten Unterklasse das gewünschte Signatur-Refactoring durch. Dann an der zweiten usw.
6. Wenn alle Unterklassen geändert sind, ersetze die Aufrufe der alten Methode durch Aufrufe der neuen Methode.
7. Lösche die alte Methode.
8. Benutze das Refactoring »Methode umbenennen«, um der neuen Methode den Namen der alten zu geben.

Allgemeiner Fall: Ist die alte Methode nicht abstrakt, wird ihre Implementierung in die neue Methode kopiert. Die neue Methode darf dann natürlich keine Ausnahme werfen. Die alte Methode soll weiterhin an die neue umleiten.

7.2.4 Fazit

Die präsentierten Refactorings sind in einer möglichst abstrakten Form beschrieben. So sind sie in unterschiedlichen Zusammenhängen zu verwenden. Die Beschreibungen sind unabhängig von der konkreten Anwendung, vom konkreten Rahmenwerk und von der konkreten Programmiersprache.

Die Refactorings sind zwar gegen Rahmenwerkskopplung entworfen, lassen sich aber auch verwenden, ohne dass ein Rahmenwerk benutzt wird. Die beiden Refactorings »Ersetze technischen Datenklumpen durch Parameterobjekt« und »Führe Parameterobjekt-Erzeuger ein« können immer dann verwendet werden, wenn ein technischer Datenklumpen in einem fachlichen Teil einer Anwendung benutzt wird. Das Refactoring »Ändere die Signatur einer überschriebenen Methode« kann immer dann verwendet werden, wenn die Signatur überschriebener Methoden verändert werden soll.

7.3 Zusammenfassung

Dieses Kapitel untersuchte, inwieweit die Arbeit die gestecken Ziele erreicht hat. Das erste Ziel war die Beschreibung von Refactorings gegen Rahmenwerkskopplung. Dieses ziel konnte erreicht werden; wir stellten die erarbeiteten Refactorings in diesem Kapitel noch einmal zu einem Katalog von Beschreibungen nach dem Format von Fowler (2000) zusammen. Dieser Katalog enthält die Refactorings »Ersetze technischen Datenklumpen durch Parameterobjekt«, »Führe Parameterobjekt-Erzeuger ein« und »Ändere die Signatur einer überschriebenen Methode«. Die Beschreibungen der Refactorings abstrahieren von konkreter Anwendung, konkretem Rahmenwerk und konkreter Programmiersprache.

Als zweites Ziel sollten die entwickelten Refactorings an JCommSy angewendet werden und dieses Programm dadurch eine verbesserte Struktur erhalten. Auch dieses Ziel konnte zu einem großen Teil erreicht werden. Die Zahl der Programmzeilen in vom Servlet-Rahmenwerk akzidentiell abhängigen Klassen konnte im Rahmen dieser Arbeit von vorher ca. 12 000 auf ca. 1700 gesenkt werden. Was zu tun bleibt, um die verbleibenden Teile von der akzidentiellen Abhängigkeit zu befreien, konnte auf *story cards* formuliert werden. Die verbliebenen Schritte sind einfach auszuführen, weil das Verfahren bekannt ist.

8 Schlussbemerkungen

8.1 Zusammenfassung der Arbeit

In dieser Arbeit haben wir untersucht, welche Auswirkungen zu enge Kopplung einer Anwendung an ein Rahmenwerk haben und wie man diese Kopplung lockern kann. Wir haben die erarbeiteten Lösungen beispielhaft an der Webanwendung JCommSy angewendet.

Weil es eine unüberschaubare Vielzahl von Rahmenwerken gibt, haben wir uns zunächst als relevantes Beispielrahmenwerk das Servlet-Rahmenwerk der JEE gewählt. Wir haben das Beispielrahmenwerk untersucht und dann die Übertragbarkeit der Ergebnisse auf andere Rahmenwerke diskutiert. Wir haben acht aus der zu engen Kopplung resultierende Probleme erkannt und beleuchtet. Wir wollen hier exemplarisch nur zwei nennen. Erstens: Die Anwendung wird bei zu enger Kopplung an ein Rahmenwerk auf die Technologie (beim Servlet-Rahmenwerk die der Oberfläche) festgelegt. Zweitens: Die Klassen der Anwendung, die Elemente des Rahmenwerks verwenden, sind unnötig schwer zu testen. Diese und die weiteren sechs Probleme treten in unterschiedlicher Stärke auch bei der Verwendung anderer Rahmenwerke auf. Sie werden davon begleitet, dass einerseits akzidentielle Voraussetzungen existieren und andererseits essentielle Voraussetzungen nicht ausdrücklich – also stillschweigend – sind. Alle diese Probleme können durch die Lockerung der Kopplung an das Rahmenwerk gelöst oder zumindest abgeschwächt werden. Für diese Lockerung haben wir deshalb drei Schritte vorgeschlagen, die darauf basieren, akzidentielle Voraussetzungen zu beseitigen und stillschweigende Voraussetzungen ausdrücklich zu machen: »Voraussetzungen als Vorbedingungen ausdrücklich machen«, »Vorbedingungen in Parameter umwandeln« und »Einführen eines Parameterobjektes«. Grundsätzlich ist es sinnvoll, die Variante »Voraussetzungen als Parameter ausdrücken« zu verwenden; in Verbindung mit überschriebenen Methoden bietet sich die Verwendung eines Parameterobjektes an. (Kapitel 3)

Die Lösung der drei Lockerungsschritte wendeten wir an der Weban-

wendung JCommSy an, die unter den oben beschriebenen Problemen litt. JCommSy hat den Architekturstil »Seitenfragmentarchitektur«, bei dem es eine Klasse mit vielen (in JCommSy 80) Unterklassen gibt. Die Kopplung an das Servlet-Rahmenwerk steckte in JCommSy in der Signatur von Methoden, die in der Oberklasse abstrakt definiert und in allen Unterklassen überschrieben wurde. Unser Lösungsansatz reichte für diesen Fall noch nicht aus. (Kapitel 4)

Deshalb beschäftigten wir uns zunächst isoliert mit dem Ändern der Signatur von überschriebenen Methoden. Wir entwickelten sichere Refactorings, die im Rahmen eines Großen Refactoring angewandt werden können. Diese Refactorings sind auch in anderen Zusammenhängen verwendbar. Dann untersuchten wir die Kombination der Probleme von Rahmenwerkskopplung und von Signaturänderungen. Es zeigte sich, dass diese durch eine Kombination der Lösungen gegen Rahmenwerkskopplung und für das Ändern von überschriebenen Methoden beseitigt werden können. Dazu setzt man die Verwendung eines Parameterobjektes und die Refactorings zum Ändern der Signatur von überschriebenen Methoden ein. Wir vermuten, dass dieses zweite Problemfeld häufig durch das erste Problemfeld ausgelöst wird und entsprechend die Kombination häufig auftritt. (Kapitel 5)

Um JCommSy und Servlet-Rahmenwerk schließlich zu entflechten, wendeten wir die Kombination der Lösungsansätze an. JCommSy konnte dadurch in weiten Teilen von der Abhängigkeit zum Servlet-Rahmenwerk befreit werden. Die konkrete Umstellung beschrieben wir am Beispiel einer typischen Klasse. Die Umstellung umfasst sowohl lesenden als auch schreibenden Zugriff auf Typen aus dem Servlet-Rahmenwerk. (Kapitel 6)

Die Arbeit hatte zwei Ziele. Das erste Ziel ist, sichere Refactorings gegen Rahmenwerkskopplung zu beschreiben. Wir präsentierten die drei Refactorings »Ersetze technischen Datenklumpen durch Parameterobjekt«, »Führe Parameterobjekt-Erzeuger ein« und »Ändere die Signatur einer überschriebenen Methode«. Das zweite Ziel ist, unsere Beispielanwendung JCommSy von der engen Kopplung an das Servlet-Rahmenwerk zu befreien. Von den 85 akzidentiell vom Servlet-Rahmenwerk abhängigen Klassen des Systems JCommSy konnten 80 von ihrer akzidentiellen Abhängigkeit befreit werden. Die noch verbleibenden Arbeiten konnten so beschrieben werden, dass sie mit den in der Arbeit entwickelten Refactorings ausgeführt werden können. (Kapitel 7)

8.2 Fazit

Die gesteckten Ziele wurden erreicht. Wir haben Refactorings gegen Rahmenwerkskopplung gefunden und JCommSy konnte vom Servlet-Rahmenwerk entflochten werden. Nebenbei haben wir ein unerwartetes Problem intensiv betrachtet und eine gute Lösung finden können: wie man die Signatur von überschriebenen Methoden sicher und schrittweise ändern kann.

Die verwendeten Refactorings waren gut geeignet, um JCommSy zu entflechten. Wir kamen mit dem Umbau zügig voran. Leider war die schiere Menge des Quelltextes, der angepasst werden musste, zu groß, um sie im Rahmen einer Diplomarbeit komplett durchzuführen. Der Teil der Refactorings, die von der verwendeteten Entwicklungsumgebung unterstützt werden, konnte automatisch ausgeführt werden. Zu diesen gehören Hilfs-Refactorings, wie »Methode umbenennen«. Unsere neu entwickelten Refactorings haben leider noch keine automatische Unterstützung, und so mussten wir sie von Hand ausführen.

Letztendlich konnte ein wesentlich größerer Teil von JCommSy im Rahmen der vorliegenden Arbeit bewältigt werden, als wir ursprünglich angenommen hatten. Dies ist dem Umstand geschuldet, dass die Refactorings ihre Aufgabe sehr gut erfüllen.

8.3 Ausblick

In dieser Arbeit konnten wir aus Zeitgründen nur die Auswirkungen enger Kopplung an *ein* Rahmenwerk – das Servlet-Rahmenwerk – betrachten. Verschiedene Ergebnisse sind auf andere Rahmenwerke übertragbar. Es wäre wünschenswert, diese Ergebnisse an *anderen* Rahmenwerken auszuprobieren, um festzustellen, ob dort noch weitere Probleme entstehen und inwieweit sie mit den von uns vorgeschlagenen Lösungen beseitigt werden können. Hierbei sollten sowohl weitere Web-Rahmenwerke (wie etwa Spring oder JSF) als auch anderen Oberflächen-Rahmenwerke (wie Swing oder SWT), als auch Rahmenwerke ganz anderen Typs (wie etwa für die Datenbankanbindung, z. B. Hibernate oder JPA) untersucht werden.

Wir konnten unsere Lösung nur an einer Anwendung – der Webanwendung JCommSy – ausprobieren. Eine interessantes Projekt wäre es, die entstandenen Refactorings an weiteren zu eng an ein Rahmenwerk gekoppelten Anwendungen auszuprobieren. Hier könnten sowohl Anwendungen

untersucht werden, die an das Servlet-Rahmenwerk gekoppelt sind, als auch solche, die an andere Rahmenwerke gekoppelt sind.

Das Problemfeld der Änderung der Signatur überschriebener Methoden wurde erst im Laufe der Arbeit in den Fokus gerückt. Zunächst sah es so aus, als sei es nur zufällig in Kombination mit der Rahmenwerkskopplung aufgetreten. Wir vermuten allerdings, dass diese Kombination häufig ist und dass dieses zweite Problemfeld oft als Konsequenz der Rahmenwerkskopplung vorkommt. Um diese Vermutung zu stützen, müssten weitere Systeme mit enger Rahmenwerkskopplung untersucht werden.

Für viele Refactorings gibt es automatisierte Unterstützung durch Werkzeuge wie Eclipse oder Netbeans. Es ist denkbar, solche Unterstützung auch für die hier vorgestellten Refactorings zu entwickeln.

Die entwickelten Verfahren wurden bei dem Beispielsystem JCommSy mit Erfolg angewendet und haben ihre Praxistauglichkeit bewiesen. Wir sind deshalb sicher, dass sie auch in ähnlichen Situationen erfolgreich angewandt werden kann.

Danksagung

Die Prüfungsordnung erfordert es – eine Diplomarbeit muss selbständig und ohne fremde Hilfe angefertigt werden. Das ist eigentlich ein Widerspruch in sich selbst; denn eine Abschlussarbeit kann nicht ohne die Unterstützung von ganz unterschiedlichen Menschen geschrieben werden. Diesen Menschen möchte ich meinen Dank aussprechen.

Wolf-Gideon Bleek und Winfried Lamersdorf danke ich dafür, dass sie die Begutachtung meiner Arbeit übernommen haben. Wolf-Gideon hat mich intensiv betreut und die Arbeit im rohen, halb-fertigen und fast-ganz-fertigen Zustand gelesen und mir in zahlreichen Diskussionen wervolle Ratschläge gegeben. Er beherrscht die Kunst, Kritik so zu formulieren, dass der Schreiber sie als Ansporn und nicht als niederschmetternd empfinden kann. Auch Alexander Pokahr hat mich gut betreut, und ich danke ihm für das Korrekturlesen der Arbeit und die zahlreichen Verbesserungsvorschläge.

Meinen Kommilitionen Arne Scharping, Christof Kubosch und Sebastian Middeke danke ich für die Korrekturen und die guten Ideen. Sie waren wirkliche Mit-Kämpfer und die Gespräche mit ihnen über unsere Diplomarbeiten und Gott und die Welt haben mir sehr geholfen.

Zuletzt und doch vor allen anderen möchte ich meiner kleinen Familie danken, die sich während der mit dem Schreiben der Diplomarbeit vergangenen Monate entscheidend vergrößert hat. Anja und Lennart, Ihr habt mir die nötige Kraft gegeben. Ihr macht mir jeden Tag die größte Freude. Danke!

Literaturverzeichnis

Abelson u. Sussmann 1998 Abelson, Harold ; Sussmann, Gerald Jay mit Julie S.: *Struktur und Interpretation von Computerprogrammen : eine Informatik-Einführung*. 3. überarb. Auflage. Berlin : Springer, 1998. – ISBN 3–540–63898–9

Balaban u. a. 2005 Balaban, Ittai ; Tip, Frank ; Fuhrer, Robert: Refactoring support for class library migration. In: Johnson, Ralph (Hrsg.) ; Gabriel, Richard P. (Hrsg.): *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–031–0, S. 265–279

Balzert 1998 Balzert, Helmut: *Lehrbuch der Software-Technik*. Bd. 2: *Software-Managment, Software-Qualitätssicherung, Unternehmensmodellierung*. 1. Auflage. Heidelberg : Spektrum, 1998. – ISBN 3–8274–0065–1

Balzert 2000 Balzert, Helmut: *Lehrbuch der Software-Technik*. Bd. 1: *Software-Entwicklung*. 2. Auflage. Heidelberg : Spektrum, 2000. – ISBN 3–8274–0480–0. – 1. Nachdruck 2001

Basili u. a. 1996 Basili, Victor R. ; Briand, Lionel C. ; Melo, Walcélio L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. In: *IEEE Transactions on Software Engineering* 22 (1996), October, Nr. 10, S. 751–761. http://dx.doi.org/10.1109/32.544352. – DOI 10.1109/32.544352. – ISSN 0098–5589

Bass u. a. 2003 Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice*. 2. Auflage. Boston, MA, USA : Addison-Wesley Professional, 2003 (SEI Series in Software Engineering). – ISBN 9780321154958

Beck u. Andres 2005 Beck, Kent ; Andres, Cynthia: *Extreme Programming Explained: Embrace Change*. 2. Auflage. Upper Saddle River, NJ, USA : Addison-Wesley, 2005. – ISBN 0–321–27865–8

Beeger u. a. 2007 Beeger, Robert F. ; Haase, Arno ; Roock, Stefan ; Sanitz, Sebastian: *Hibernate – Persistenz in Java-Systemen mit Hibernate un der Java Persistence API*. 2. überarbeitete und erweiterte Auflage. Heidelberg : dpunkt, 2007. – ISBN 978–3–89864–447–1

Bischofberger u. a. 2004 Bischofberger, Walter ; Kühl, Jan ; Löffler, Silvio: Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In: Oquendo, Flavio (Hrsg.) ; Warboys, Brian (Hrsg.) ; Morrison, Ron (Hrsg.): *Software Architecture – First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004. Proceedings* Bd. 3047. Berlin / Heidelberg : Springer, 2004 (Lecture Notes in Computer Science). – ISBN 978–3–540–22000–8, S. 1–9

Bleek u. Jackewitz 2004 Bleek, Wolf-Gideon ; Jackewitz, Iver: Providing an E-Learning Platform in a University Context – Balancing the Organisational Frame for Application Service Providing. In: *Hawaii International Conference on System Sciences* Bd. 03. Los Alamitos, CA, USA : IEEE Computer Society, 2004. – ISSN 1530–1605, S. 30067c

Boehm 1979 Boehm, Barry W.: Software engineering. In: Yourdon, Edward N. (Hrsg.): *Classics in software engineering*. New York, NY, USA : Yourdon Press, 1979. – ISBN 0–917072–14–6, S. 323–361

Booch u. a. 1999 Booch, Grady ; Rumbaugh, Jim ; Jacobson, Ivar: *Das UML-Benutzerhandbuch*. 2. Auflage. München : Addison-Wesley, 1999 (Professionelle Softwareentwicklung). – ISBN 3–8237–1486–0

Boudreau u. a. 2002 Boudreau, Tim ; Glick, Jesse ; Greene, Simeone ; Vaughn, Spurlin ; Woehr, Jack: *NetBeans : the definitive guide*. Sebastopol, CA, USA : O'Reilly, 2002. – ISBN 978–0596002800

Brooks 1987 Brooks, Frederick Phillips J.: No Silver Bullet: Essence and Accidents of Software Engineering. In: *Computer* 20 (1987), April, Nr. 4

Brügge u. Dutoit 2004 Brügge, Bernd ; Dutoit, Allen H.: *Object-Oriented Software Engineering Using UML, Patterns, and Java*. 2. Auflage. Upper Saddle River, NJ, USA : Pearson Prentice Hall, 2004. – ISBN 0–13–191179–1

Cactus-Webseite 2008 *Die Webseite der Testsoftware Cactus*. `http://jakarta.apache.org/cactus/`, Abruf: 2008-02-18

Cekvenich u. Gehner 2004 Cekvenich, Vik ; Gehner, Wolfgang: *Struts – Best Practices*. Heidelberg : dpunkt, 2004. – ISBN 3–89864–284–4

Claus u. Schwill 2006 Claus, Volker (Hrsg.) ; Schwill, Andreas (Hrsg.): *Duden Informatik*. Mannheim : Dudenverlag, 2006. – ISBN 978–3–411–0523–9

CommSy-Handbuch 2005 CommSy-Team (Hrsg.): *CommSy BenutzerInnenhandbuch Kontext: Hochschule*. Hamburg: CommSy-Team, 2005. `http://www.cli84.kunden.minispace.de/commsyweb/uploads/Software/commsy%_nutzungshandbuch.pdf`, Abruf: 2007-09-21

Date 2004 Date, Chris J.: *An introduction to database systems*. 8. Boston, MA, USA : Addison-Wesley, 2004. – ISBN 0–321–18956–6

Daum 2006 Daum, Berthold: *Java-Entwicklung mit Eclipse 3.2 – Anwendungen, Plugins und Rich Clients*. 4. überarbeitete und erweiterte Auflage. Heidelberg : dpunkt, 2006. – ISBN 3–89864–426–X

Denert 1991 Denert, Ernst: *Software-Engineering : methodische Projektabwicklung*. Berlin : Springer, 1991. – ISBN 3–540–53404–0

DIN 55350 1995 Norm DIN 55350-11 1995. *Begriffe zu Qualitätsmanagement und Statistik - Teil 111: Begriffe des Qualitätsmanagements*

Dumke u. a. 2003 Dumke, Reiner ; Lother, Mathias ; Cornelius, Wille ; Zbrog, Fritz: *Web Engineering*. München : Pearson Studium, 2003. – ISBN 3–8273–7080–9

Dyer 2005 Dyer, Russell J. T.: *MySQL in a Nutshell*. Sebastopol, CA, USA : O'Reilly, 2005. – ISBN 0–596–00789–2

EasyMock-Webseite 2008 *Die Webseite der Testsoftware EasyMock*. `http://www.easymock.org/`, Abruf: 2008-02-18

Floyd 2007 Floyd, Christiane: Architekturzentrierte Softwaretechnik. In: Bleek, Wolf-Gideon (Hrsg.) ; Raasch, Jörg (Hrsg.) ; Züllighoven, Heinz (Hrsg.): *Software Engineering 2007*. Bonn : Köllen Druck+Verlag, 2007. – ISBN 978–3–88579–199–7, S. 19–22

Floyd u. Oberquelle 2007 Floyd, Christiane ; Oberquelle, Horst: Softwaretechnik und Software-Ergonomie / Universität Hamburg. 2007/2008. – Vorlesungsskript

Floyd u. Züllighoven 2002 Floyd, Christiane ; Züllighoven, Heinz: Softwaretechnik. In: Pomberger, Gustav (Hrsg.) ; Rechenberg, Peter (Hrsg.): *Informatik Handbuch*. 3., aktualisierte und erweiterte Auflage. München : Hanser, 2002. – ISBN 3–446–21824–4, S. 763 – 790

Floyd 1967 Floyd, Robert W.: Assigning Meaning to Programs. In: *Mathematical Aspects of Computer Science* Volume 19 of Proceedings of Symposia on Applied Mathematics (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/readlings/Floyd.pdf

Fowler 2000 Fowler, Martin: *Refactoring: Wie Sie das Design vorhandener Software verbessern*. München : Addison-Wesley, 2000. – ISBN 978–0201485677

Gamma u. a. 1996 Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster*. München : Addison-Wesley, 1996. – ISBN 0–201–63361–2

Goldstein u. a. 2006 Goldstein, Maayan ; Feldman, Yishai A. ; Tyszberowicz, Shmuel: Refactoring with Contracts. In: Chao, Joseph (Hrsg.) ; Cohn, Mike (Hrsg.) ; Maurer, Frank (Hrsg.) ; Sharp, Helen (Hrsg.) ; Shore, James (Hrsg.) ; IEEE (Veranst.): *Proceedings of AGILE 2006 Conference (AGILE'06)* IEEE, 2006. – ISBN 0–7695–2562–8, S. 10 ff

Gosling u. a. 1997 Gosling, James ; Joy, Bill ; Steele, Guy: *Java – Die Sprachspezifikation*. 1. Bonn : Addison-Wesley-Longman, 1997 (Java Series). – ISBN 3–8273–1038–5

Gumm 2006 Gumm, Dorina C.: Distribution Dimensions in Software Development Projects: A Taxonomy. In: *IEEE Software* 23 (2006), Nr. 5, S. 45–51. http://dx.doi.org/10.1109/MS.2006.122. – DOI 10.1109/MS.2006.122. – ISSN 0740–7459

Hoare 1969 Hoare, C. A. R.: An axiomatic basis for computer programming. In: *Communications of the ACM* 12 (1969), Oktober, Nr. 10, S. 576–580. http://dx.doi.org/10.1145/363235.363259. – DOI 10.1145/363235.363259. – ISSN 0001–0782

Hohmann 2007 Hohmann, Christoph: *Softwarearchitektur von J2EE Web Applikationen mit Web Components am Beispiel von AJAX*, Universität Hamburg, Diplomarbeit, Januar 2007. `http://swt-www.informatik.uni-hamburg.de/publications/papers/Dipl/diplomarbeit_christoph_hohmann.pdf`

Horstmann u. Cornell 2005a Horstmann, Cay S. ; Cornell, Gary: *Core Java*. Bd. 2: *Expertenwissen*. München : Addison-Wesley, 2005. – ISBN 3–8273–2244–8

Horstmann u. Cornell 2005b Horstmann, Cay S. ; Cornell, Gary: *Core Java*. Bd. 1: *Grundlagen*. München : Addison-Wesley, 2005. – ISBN 3–8273–2216–2

IEEE 610.12 1992 Norm IEEE 610.12 1992. *IEEE Standard Glossary of Software Engineering Terminology*

Ihns u. a. 2007 Ihns, Oliver ; Harbeck, Dierk ; Heldt, Stefan M. ; Koschek, Holger: EJB 3 *professionell – Grundlagen- und Expertenwissen zu Enterprise JavaBeans 3 für Einsteiger, Umsteiger und Fortgeschrittene*. Heidelberg : dpunkt, 2007 (iX-Edition). – ISBN 978–3–89864–431–0

Janneck u. Bleek 2002 Janneck, Michael ; Bleek, Wolf-Gideon: Project-based Learning with CommSy. In: Stahl, Garry (Hrsg.): *Proceedings of the Conference on Computer Supported Cooperative Learning*, 2002, 509 – 510

Jeenicke 2005 Jeenicke, Martti: Architecture-Centric Software Migration for the Evolution of Web-based Systems. In: *Workshop on Architecture-Centric Evolution (ACE 2005), ECOOP 2005*. Glasgow, July 2005. – Nur online

Jendrock u. a. 2006 Jendrock, Eric ; Ball, Jennifer ; Carson, Debbie ; Evans, Ian ; Fordin, Scott ; Haase, Kim: *The Java EE 5 Tutorial*. 3. Auflage. Upper Saddle River, NJ : Addison-Wesley, 2006 `http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`. – ISBN 0–321–49029–0

Johnson u. Foote 1988 Johnson, Ralph E. ; Foote, Brian: Designing Reusable Classes. In: *Journal of Object-Oriented Programming* 1 (1988), June/July, Nr. 2, S. 22–35

Jordan u. Russell 2003 Jordan, David ; Russell, Craig: *Java Data Objecs*. Sebastopol, CA, USA : O'Reilly, 2003. – ISBN 0–596–00276–9

Kappel u. a. 2004 Kappel, Gerti (Hrsg.) ; Pröll, Birgit (Hrsg.) ; Reich, Siegfried (Hrsg.) ; Retschitzegger, Werner (Hrsg.): *Web-Engineering – Systematische Entwicklung von Web-Anwendungen*. 1. Auflage. Heidelberg : dpunkt, 2004. – ISBN 3–89864–234–8

Karlsson 1995 Karlsson, Even-André (Hrsg.): *Software Reuse: A Holistic Approach*. Chichester, England : John Wiley & Sons, 1995. – ISBN 0 471 95489 6

Kerievsky 2004 Kerievsky, Joshua: *Refactoring to Patterns*. Addison-Wesley, 2004. – ISBN 978–0321213358

Kieżun u. a. 2007 Kieżun, Adam ; Ernst, Michael D. ; Tip, Frank ; Fuhrer, Robert M.: Refactoring for Parameterizing Java Classes. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE)* IEEE, 2007. – ISBN 0–7695–2828–7, S. 437–446

Krueger 1992 Krueger, Charles W.: Software Reuse. In: *ACM Computing Surveys* 24 (1992), June, Nr. 2, S. 131–183. http://dx.doi.org/10.1145/130844.130856. – DOI 10.1145/130844.130856. – ISSN 0360–0300

Kubosch 2007 Kubosch, Christof: *Einsatz von Softwareinstrumenten zur Detektion von Smells und Verletzungen von softwaremaßbasierten Regeln im Rahmen eines kontinuierlichen Software-Entwicklungsprozesses am Beispiel von JCommSy*, Universität Hamburg, Diplomarbeit, November 2007

Lieberherr u. Holland 1989 Lieberherr, Karl J. ; Holland, Ian M.: Assuring Good Style for Object-Oriented Programs. In: *IEEE Softw.* 6 (1989), Nr. 5, S. 38–48. http://dx.doi.org/10.1109/52.35588. – DOI 10.1109/52.35588. – ISSN 0740–7459

Lilienthal 2008 Lilienthal, Carola: *Komplexität von Softwarearchitekturen – Stile und Strategien*. Hamburg, Universität Hamburg, Diss., 2008

Link 2005 Link, Johannes: *Softwaretests mit JUnit*. 2., überarb. u. erw. Auflage. Heidelberg : Dpunkt Verlag, 2005. – ISBN 978–3898643252

Liskov u. Wing 1993 Liskov, Barbara ; Wing, Jeannette M.: Family Values: A Behavioral Notion of Subtyping / Carnegie Mellon University. Pittsburgh, PA, USA, 1993. – Forschungsbericht

Ludewig u. Lichter 2007 Ludewig, Jochen ; Lichter, Horst: *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken*. 1. Auflage. Heidelberg : Dpunkt Verlag, 2007. – ISBN 3–89864–268–2

Mackinnon u. a. 2001 Mackinnon, T. ; Freeman, S. ; Craig, P.: Endo-Testing: Unit Testing with Mock Objects. In: Succi, Giancarlo (Hrsg.) ; Marchesi, Michele (Hrsg.): *Extreme Programming Examined*. Boston : Addison-Wesley, 2001. – ISBN 0–201–71040–4, 287–301

Meyer 1992 Meyer, Bertrand: *Eiffel: the language*. New York : Prentice-Hall, 1992. – ISBN 0–13–247925–7

Meyer 1997 Meyer, Bertrand: *Object-oriented software construction*. 2. Auflage. Upper Saddle River, NJ, USA : Prentice-Hall, 1997. – ISBN 0–13–629155–4

Mock-vs-Container 2008 *The Apache Software Foundation: Mock Objects vs In-Container testing*. `http://jakarta.apache.org/cactus/mock_vs_cactus.html`, Abruf: 2008-02-18

Myers u. a. 2004 Myers, Glenford J. ; Badgett, Tom ; Thomas, Todd M. ; Sandler, Corey: *The Art of Software Testing*. 2. Auflage. Hoboken, NJ, USA : John Wiley & Sons, 2004. – ISBN 0–471–46912–2. – Revised and updated by Tom Badgett and Todd Thomas, with Corey Sandler

Offutt 2002 Offutt, Jeff: Quality attributes of Web software applications. In: *Software, IEEE* 19 (2002), March/April, Nr. 2, 25–32. `http://dx.doi.org/10.1109/52.991329`. – DOI 10.1109/52.991329. – ISSN 0740–7459S

Opdyke 1992 Opdyke, William F.: *Refactoring Object-Oriented Frameworks*. Urbana-Champaign, IL, USA, University of Illinois, Diss., 1992. `http://citeseer.ist.psu.edu/opdyke92refactoring.html`

Parnas 1971 Parnas, David L.: Information Distribution Aspects of Design Methodology. In: Freiman, C. V. (Hrsg.) ; Griffith, John E. (Hrsg.) ; Rosenfeld, J. L. (Hrsg.) ; IFIP (Veranst.): *Information Processing 71,*

Proceedings of IFIP Congress 71, Ljubljana, Yugoslavia, August 23-28, 1971 Bd. 1 - Foundations and Systems. North-Holland, 1971. – ISBN 0–7204–2063–6, S. 339–344

Parnas 1972 Parnas, David L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), Nr. 12, S. 1053–1058. http://dx.doi.org/10.1145/361598.361623. – DOI 10.1145/361598.361623. – ISSN 0001–0782

Reenskaug 1979 Reenskaug, Trygve: Models – Views – Controllers / Xerox PARC. Version: December 1979. http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf, Abruf: 2007-11-20. 1979. – Technical note

Reussner u. Hasselbring 2006 Reussner, Ralf (Hrsg.) ; Hasselbring, Wilhelm (Hrsg.): *Handbuch der Software-Architektur*. Heidelberg : Dpunkt Verlag, 2006. – ISBN 978–3898643726

RFC 2616 1999 Norm RFC 2616 1999. *Hypertext Transfer Protocol – HTTP/1.1*

Roock u. Lippert 2004 Roock, Stefan ; Lippert, Martin: *Refactorings in großen Softwareprojekten. Komplexe Restrukturierungen erfolgreich durchführen*. Heidelberg : Dpunkt Verlag, 2004. – ISBN 978–3898642071

Royce 1987 Royce, Winston W.: Managing the development of large software systems: concepts and techniques. In: *ICSE '87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1987. – ISBN 0–89791–216–0, 328–338

Rupp 2005 Rupp, Heiko W.: *JBoss – Server-Handbuch für J2EE-Entwickler und Administratoren*. Heidelberg : dpunkt, 2005. – ISBN 3–89864–318–2

Sebesta 2008 Sebesta, Robert W.: *Concepts of Programming Languages*. 8. Auflage. Boston : Pearson Addison-Wesley, 2008. – ISBN 978–0–321–50968–0

Siedersleben 2004 Siedersleben, Johannes: *Moderne Software-Architektur. Umsichtig planen, robust bauen mit Quasar*. Heidelberg : Dpunkt Verlag, 2004. – ISBN 978–3898642927

Sommerville 2001 Sommerville, Ian: *Software Engineering*. 6. Auflage. München : Pearson, 2001. – ISBN 3–8273–7001–9

Stephens u. Rosenberg 2003 Stephens, Matt ; Rosenberg, Doug: *Extreme Programming Refactored: The Case Against XP*. Berkeley, CA, USA : Apress, 2003. – ISBN 1–59059–096–1

Stroustrup 1998 Stroustrup, Bjarne: *Die C++-Programmiersprache*. 3., aktualisierte und erweiterte Auflage. Bonn : Addison-Wesley-Longman, 1998 (Professionelle Programmierung). – ISBN 3–8273–1296–5

Tomcat-Webseite 2008 *Die Webseite des Anwendungsservers Tomcat*. `http://tomcat.apache.org/`, Abruf: 2008-03-17

Turau 1999 Turau, Volker: Techniken zur Realisierung Web-basierter Anwendungen. In: *Informatik-Spektrum* 22 (1999), Februar, Nr. 1, S. 3–12. `http://dx.doi.org/10.1007/s002870050119`. – DOI 10.1007/s002870050119. – ISSN 1432–122X

Turau u. a. 2004 Turau, Volker ; Saleck, Krister ; Lenz, Christopher: *Webbasierte Anwendungen entwickeln mit* jsp 2. 2., vollständig überarbeitete Auflage. Heidelberg : dpunkt, 2004. – ISBN 3–89864–235–6. – Die erste Auflage erschient unter dem Titel »Java Server Pages und J2EE«

Uni-CommSy 2007 *Das CommSy der Universität Hamburg*. `http://www.commsy.uni-hamburg.de/`, Abruf: 2007-09-21

Winter 2005 Winter, Mario: *Methodische objektorientierte Softwareentwicklung*. 1. Auflage. Heidelberg : dpunkt, 2005. – ISBN 3–89864–273–9

WissPro-Webseite 2007 *Die Webseite des Forschungsprojektes WissPro*. `http://www.wisspro.de/`, Abruf: 2007-09-17

Wolff 2007 Wolff, Eberhard: *Spring 2 – Framework für die Java-Entwicklung*. 2., aktuaisierte und erweiterte Auflage. Heidelberg : dpunkt, 2007 (iX-Edition). – ISBN 978–3–89864–465–5

Wöhr 2004 Wöhr, Heiko: *Web-Technologien: Konzepte – Programmiermodelle – Architekturen*. Heidelberg : dpunkt, 2004. – ISBN 3–89864–247–X

Zuser u. a. 2001 Zuser, Wolfgang ; Biffl, Stefan ; Grechenig, Thomas ; Köhle, Monika: *Software engineering : mit UML und dem Unified Process*. München : Pearson Studium, 2001. – ISBN 3–8273–7027–2

Erklärungen

Selbständigkeit

Ich versichere, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Bibliothekseinstellung

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.