Baccalaureatsarbeit

Architekturvereinbarungen des JCommSys

Eingereicht von:

Arne Scharping

(Matrikelnummer.: 522314)

Betreuerin:

Prof. Dr. Christiane Floyd

Hamburg, 22. Dezember 2005

Universität Hamburg, Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Erklärung:

Hiermit versichere ich, dass ich diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt habe.

Hamburg, 22.12.2005

Arne Scharping

Halstenbeker Strasse 89a
22457 Hamburg
eMail: 1scharpi@informatik.uni-hamburg.de
Matrikelnummer: 5422314

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Hintergrund dieser Arbeit ist ein Lehreprojekt, dass im Wintersemester 2004/05 und im Sommersemester 2005 am Fachbereich Informatik der Universität Hamburg durchgeführt worden ist. Inhalt dieser Veranstaltung war die prototypische Migration einer Web-Anwendung von PHP nach Java. In diesem Projekt ist das JCommSy entstanden, das als Grundlage für die in dieser Arbeit durchgeführten Betrachtungen dient.

Die Entwicklung des JCommSys erfolgt in einem zyklisch-inkrementellen Prozesse. Ein Teil der Aktivitäten innerhalb dieses Prozesses dient der Entwicklung einer Soll-Architektur, die eine Vorgabe für die Implementierung ist. Im Laufe des Projektes wurde immer wieder festgestellt, dass vereinbarte Aspekte dieser Soll-Architektur in der Implementierung verletzt wurden. Solche Verletzungen vermindern die Verständlichkeit, Wartbarkeit und Erweiterbarkeit des Systems und begünstigen so eine Degenerierung der Architektur.

Diese Arbeit beschäftigt sich mit der Frage, wie sichergestellt werden kann, dass die Vorgaben der Soll-Architektur eines Softwaresystems auch tatsächlich in der Implementierung umgesetzt werden. Dazu wird zunächst die Entstehung des Problems betrachtet und eine Bestandsaufnahme der Lösungsmöglichkeiten durchgeführt. Weiter wird ein Lösungsansatz vorgestellt, bei dem wichtige Aspekte der Soll-Architektur in einer regelhaften Form, als so genannte *Architekturvereinbarungen* formuliert und in einem Katalog zusammengetragen werden. Dieser Katalog bildet die Basis einer toolgestützen Prüfung der Ist-Architektur auf Einhaltung der Vorgaben der Soll-Architektur.

Mit dem beschriebenen Ansatz schließt diese Arbeit an die Diplomarbeit von Bettina Karstens [Kar05] an. In dieser hat sie Architekturregeln für die WAM-Modellarchitektur formuliert und deren Prüfbarkeit untersucht hat.

Im Gegensatz zu den Architekturregeln der Modellarchitektur sind die Architekturvereinbarungen, die im Mittelpunkt dieser Arbeit stehen, für ein Projekt spezifisch. Sie entstehen und verändern sich mit den Aktivitäten zur Entwicklung der Soll-Architektur im Laufe des Projektes. Aus diesem Grund werden auch der Prozess der Identifizierung der Architekturvereinbarungen und Kriterien für die Entscheidung, ob ein Aspekt der Soll-Architektur in den Katalog aufgenommen werden sollte, betrachtet.

In dieser Arbeit wird der Einsatz verschiedener Werkzeuge zur Prüfung der Regeln beschrieben. Die Entwicklung eines allgemeinen Verfahrens zur Prüfung der Vereinbarungen und ein abschließender Vergleich der verwendeten Werkzeuge können in diesem Rahmen nicht durchgeführt werden.

2 Hintergrund: Migration des CommSy

2.1 Das WebMig-Projekt

Hintergrund der Arbeit ist das Lehreprojekt „Migration von Software-Systemen am Beispiel einer Web-Anwendung“, das im WS 2004/2005 und SS 2005 am Fachbereich Informatik der Universität Hamburg stattgefunden hat. Betreut wurde das Projekt von Martti Jeenike, Prof. Dr. Christiane Floyd und Dr. Wolf-Gideon Bleek. Die zentralen Inhalte des Projektes waren Methoden, Techniken und Werkzeuge zur Migration von Software.

Als Migration wird im Rahmen des Projektes der Wechsel der Programmiersprache eines Systems, unter Beibehaltung der Funktionalität, verstanden. Die Oberfläche und das für den Benutzer sichtbare Verhalten werden durch die Migration nicht verändert.

Als Gegenstand der Untersuchung diente das *CommSy* [Com], eine webbasierte Anwendung zur Unterstützung vernetzter Projektarbeit, die mit der Skriptsprache PHP geschrieben worden ist. Im Rahmen des Projektes sollte eine prototypische Migration des CommSys von PHP nach Java durchgeführt werden.

Mittlerweile ist das Lehreprojekt abgeschlossen und die Migration wird von wissenschaftlichen Mitarbeitern und studentischen Hilfskräften in einem mit Drittmitteln geförderten Projekt fortgeführt.

2.2 Das CommSy

Der Name CommSy steht für „Community System“. Das CommSy stellt für Projektgruppen eine Basis bereit, die sie bei ihrer Koordination und der Durchführung kooperativer Tätigkeiten unterstützt.

Das CommSy wird seit 1999 am Fachbereich Informatik der Universität Hamburg entwickelt. Erste Versionen wurde zunächst in Eigeninitiative von wissenschaftlichen Mitarbeitern und Studierenden geschrieben. Seit 2001 erfolgt die Weiterentwicklung im Rahmen verschiedener vom Bundesministerium für Bildung und Finanzen (BMFB) geförderter Projekte.

Im Jahr 2003 ist das CommSy als Open Source-Software auf Sourceforge[1] zur Verfügung gestellt worden In einer Kooperation mit dem Hamburger Informatik Technologie-Center e.V. (HITeC)[2] wird es aber auch kommerziell bereitgestellt.

Die wichtigste Nutzergruppe des CommSys sind Forschungs- und Lerngemeinschaften. Diesen stellt das CommSy geschützte (virtuelle) Projekträume zur Verfügung, in denen Sie eine Umgebung finden, die sie bei der Kooperation und Koordination unterstützt. Projekträume sind in so genannte Rubriken untergliedert, die zum Beispiel der Verwaltung von Materialien dienen oder die Vereinbarung und Visualisierung von Terminen ermöglichen. Durch Gemeinschaftsräume kann ein Rahmen für einen gruppenübergreifenden Austausch geschaffen werden.

2.3 Die Migration des CommSys

Nach einer Analyse der PHP-Implementierung des CommSys wurden zunächst zwei Strategien zur Migration verfolgt.

Die erste Strategie basierte auf dem Ansatz, eine weitgehend automatisierte Übersetzung des PHP-Codes in Java durchzuführen und in einem zweiten Schritt durch Refactorings das System auf

[1] http://sourceforge.net/
[2] http://www.hitec-hh.de/

Randbedingungen und Möglichkeiten der Sprache Java und des Applikationsservers abzustimmen. Nach einer kurzen Evaluation wurde dieser Ansatz eingestellt, da sich die Automatisierbarkeit einer Übersetzung des PHP-Codes in Java als zu gering erwies.

Im Weiteren wird nur noch die zweite Strategie eines zyklisch-inkrementellen Vorgehens betrachtet. Ein Bestandteil zyklisch-inkrementeller Vorgehensmodelle ist die frühzeitige Entwicklung von lauffähigen Prototypen und deren Evaluation unter realistischen Bedingungen. Daher ist eine Randbedingung der Migration die Integration der bereits migrierten und der „alten" Teilen des Systems zu einer lauffähigen Einheit. Die bereits migrierten Teile werden dabei als *JCommSy*, die noch in PHP geschriebene Teile als *PHP-CommSy* bezeichnet.

Bei einer zyklisch-inkrementellen Migration kann zwischen den Ansätzen, einer *vertikalen* und einer *horizontalen Migration*, unterschieden werden. Mit vertikaler Migration ist gemeint, dass schrittweise fachlichen Einheiten des Systems in Java implementiert werden und so das bestehende System langsam abgelöst wird. Bei einer horizontalen Migration werden mit der Zeit einzelne Schichten abgelöst.
Die Migration des CommSys folgt einer vertikalen Strategie, d.h. dass schrittweise die einzelnen Rubriken migriert werden. Begonnen wurde die Migration mit der Rubrik, die für die Verwaltung und die Koordinierung von gemeinsamen Terminen genutzt werden kann.

2.4 Entwicklung des JCommSys

verwendete Technologien

Im Fokus des Lehreprojektes, in dem die Entwicklung des JCommSys begonnen wurde, stand der Migrationsprozess. Daher wurde versucht, den Umfang verwendeter Technologien möglichst gering zu halten. Für die Steuerung und die Interaktion mit dem Benutzer (bzw. Browser) wurden Servlets, für die Darstellung JSPs und Tags des J2EE Rahmenwerkes verwendet. Auf den Einsatz von Enterprise Java Beans (EJBs) und auf weitere Rahmenwerke wie Struts oder Spring wurde verzichtet.
Das Datenbankmapping wird mit JDBC „von Hand", also ohne einen OR-Mapper wie JDO oder Hibernate durchgeführt. Ein Grund für diese Entscheidung war die Vorgabe, das Datenbankschema während der Migration nicht zu verändern. Diese Vorgabe hätte den Einsatz eines OR-Mappers zwar nicht ausgeschlossen, aber zumindest erschwert.
Für die Integration des JCommSys in das PHP-CommSy wurden verschiedene Technologien betrachtet, die bei der Überbrückung der Sprachbarriere helfen sollten. Besonders vielversprechend erschien eine prototypische Implementierung einer Java/PHP Bridge (siehe [JSR]), die sprachübergreifende Aufrufe ermöglicht. Aufgrund des geringen Reifegrades der zur Verfügung stehenden Implementierung, wurden sie aber nicht eingesetzt.
Technologisches Neuland wurde in dem Projekt nur mit der Verwendung von Java 5.0 betreten.

Integration in das PHP-System

Die Integration des JCommSys in das PHP-System wird dadurch stark vereinfacht, dass jede HTTP-Anfrage an das System entweder vom PHP-CommSy oder vom JCommSy bearbeitet wird. Dies geht einher mit der Strategie einer vertikalen Migration. Für das CommSy bedeutet dies, dass zunächst alle, die Terminrubrik betreffende Anfragen vom JCommSy abgearbeitet werden. Das PHP-CommSy wird nur dahingehend erweitert, dass Anfragen, die eine migrierte Rubrik betreffen, an das JCommSy delegiert werden.
Der Austausch von Daten zwischen beiden Systemen erfolgt über Parameter im HTTP-Request und über die Datenbank. Auch die Sessiondaten werden in der Datenbank abgelegt und stehen so beiden Systemen zur Verfügung.

Die Richtlinie, Anfragen der Terminrubrik vom JCommSy alleine abzuarbeiten, macht die vertikale Strategie zu einer idealisierten Vorstellung. Um die Anfragen vollständig verarbeiten zu können, muss das JCommSy Funktionalitäten anderer fachlicher Einheiten, also Basisfunktionalitäten oder Teile andere Rubriken implementieren.

Die wechselnden Zielsetzungen

Im Laufe des Lehreprojektes wurden zwei Iterationen durchgeführt. In der ersten Iterationen wurde ein Prototyp entwickelt, bei dem ein Java-Teil in das PHP-CommSy integriert ist. Dieser hat die Kernfunktionalität der Terminrubrik implementiert. In einer zweiten Iteration sollte die Terminrubrik vervollständigt und die Migration einer weitere Rubrik begonnen werden.

Die zweite Iteration endete mit dem Lehrprojekt, also zwei Semester nach Beginn der Migration. Da zu diesem Zeitpunkt eine Vielzahl kleinerer Features der Terminrubrik noch nicht implementiert waren, wurde die Migration weiterer Rubriken zurückgestellt und deren bereits begonnene Implementierungen rückgängig gemacht. Entsprechende Teile sind aus dem Code des JCommSys also wieder entfernt worden.

Seitdem werden die fehlenden Details der Terminrubrik implementiert und Änderungen, die sich im PHP-CommSy ergeben haben, ins JCommSy nachgezogen.

3 Problem: Verletzung der Soll-Architektur

Dieser Abschnitt stellt die Abweichung der Ist- von der Soll-Architektur, als Problemstellung dieser Arbeit, vor. Dazu wird zunächst die Verwendung einige zentrale Begriffe geklärt.

3.1 Begriffsklärung

Diese Arbeit beschäftigt sich mit der Architektur von Software-Systemen. Der zugrunde liegende Architekturbegriff wurde von Bass et al. (siehe [BCK05]) übernommen:

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

Diese Definition lässt offen, ob es sich um die Strukturen eines geplanten oder bereits implementierten Systems handelt. Für den Architekturbegriff ebenfalls unerheblich ist, ob eine Beschreibung der Strukturen beispielsweise in Form von Dokumenten oder nur durch Quellcode gegeben ist. Für diese Arbeit werden daher die Begriffe Ist-Architektur und Soll-Architektur definiert:

Als *Ist-Architektur* werden die Strukturen bezeichnet, die tatsächlich im Quellcode einer Software beschrieben werden. Im Gegensatz dazu bilden die Strukturen, deren Implementierung beabsichtigt ist, die *Soll-Architektur*. Die Soll-Architektur macht somit Vorgaben für die im Code beschriebenen Strukturen und kann als Vorlage für die Ist-Architektur verstanden werden.

Dem klassischen Wasserfall-Modell mit seinem Big Design Upfront liegt die Vorstellung zugrunde, das softwaretechnische Design (und damit die Soll-Architektur) könne am Anfang eines Projektes abschließend, vollständig, widerspruchsfrei und korrekt erstellt werden. Die zyklisch-inkrementellen Vorgehensmodelle verabschieden sich von dieser Vorstellung. Das Design einer Software wird im Laufe eines Projektes fortlaufend weiterentwickelt. Auch Soll-Architekturen sind jeweils nur für einen bestimmten Zeitraum gültig, bevor sie von einer überarbeiteten Version abgelöst werden. Für den Prozess der Implementierung, also die Weiterentwicklung der Ist-Architektur, dient dann die jeweils aktuelle Soll-Architektur als Vorgabe.

Die Entwicklung der Soll-Architektur geschieht in der Regel in Teams von Entwicklern oder Software-Architekten und unter dem Einfluss unterschiedlicher, teilweise auch gegensätzlicher Anforderungen. Die Soll-Architektur ist daher das Ergebnis eines Einigungs- und Abwägungsprozesses.

Häufig sind an der Entwicklung der Soll-Architektur andere Personen beteiligt, als an der folgenden Implementierung. In den seltensten Fällen sind alle Entwickler beteiligt. Damit die Soll-Architektur dennoch ihrem Zweck gerecht werden kann, müssen Dokumente erstellt werden, in denen die Soll-Architektur beschrieben ist. Hierfür stehen verschiedene Beschreibungssprachen wie zum Beispiel die Unified Modeling Language (UML) zur Verfügung. Der in dieser Arbeit vorgestellte Vorgabenkatalog (siehe Abschnitt 7) kann ebenfalls als Beschreibung der Soll-Architektur verstanden werden.

Auch für die Ist-Architektur können Beschreibungen erstellt werden. Dies kann beispielsweise im Rahmen der Analyse einer bestehenden Software geschehen oder um Unterschiede zwischen einer Ist- und einer Soll-Architektur zu betrachten. Da für diese Zwecke die Korrektheit der Beschreibung besonders wichtig ist, werden die Beschreibungen häufig mit Hilfe von Software-Werkzeugen aus dem Quellcode generiert.

3.2 Entstehung von Verletzungen

Weite Teile der Softwareentwicklung können als Anpassung an immer wieder neue Anforderungen verstanden werden. Viele dieser Anforderungen werden im Rahmen der sich entwickelnden Soll-Architektur erfüllt, es ergeben sich aber auch Verletzungen der Soll-Architektur. Abweichungen der Implementierung (bzw. der Ist-Architektur) von den Vorgaben der Soll-Architektur sind das zentrale Problem, dass in dieser Arbeit behandelt wird. In diesem Abschnitt wird beschrieben, wie es zu diesen Abweichungen kommt.

Refactorings

In einem zyklisch-iterativen Prozess wird nicht der Anspruch erhoben, dass die Soll-Architektur von vornherein alle Anforderungen erfüllt. Es ist daher ein normaler Bestandteil des Entwicklungsprozesses, dass sich bei der Erweiterung der Funktionalität einer Software neue Anforderungen ergeben, die mit der bestehenden Soll-Architektur nicht zu erfüllen sind. In einem solchen Fall kann es sinnvoll sein, in einem ersten Schritt eine Verbesserung der Struktur der Software unter Beibehaltung der Funktionalität, also ein Refactoring durchzuführen. Erst in einem zweiten Schritt wird die eigentliche Erweiterung der Funktionalität durchgeführt.

Der Ausgangspunkt für ein Refactoring ist eine Änderung der Soll-Architektur, die dann auf der Quellcodeebene nachvollzogen wird. Martin Fowler beschreibt in [Fow99] die Durchführung kleiner Refactorings, mit denen sehr elementare Veränderungen an objektorientierten Systemen durchgeführt werden können und gibt Hinweise und Hilfestellung zur Vermeidung von Fehlern und unerwünschten Seiteneffekten. Als Strategie beschreibt er die Zerlegung von Refactorings in elementare Schritte, die überschaubar sind und durch Unit-Tests abgesichert werden. Diese einfachen Refactorings werden heute durch die Entwicklungsumgebungen unterstützt und können relativ problemlos durchgeführt werden.

Schwieriger gestalten sich Refactorings, deren Umfang sich über mehrere Tage oder Wochen erstreckt. Stefan Roock und Martin Lippert beschäftigen sich in [RL04] ausführlich mit diesen *großen Refactorings*. Aufgrund des langen Zeitraumes über den sich große Refactorings erstrecken, werden sie häufig unterbrochen und nicht vollständig durchgeführt. Auch hier hilft die Strategie der Zerlegung in Einzelschritte. Auf diese Weise kann die Gefahr vermindert werden, dass durch die Unterbrechung von Refactorings Fehler entstehen. Auf der Architekturebene kann so eine Unterbrechung aber dazu führen, dass die Ist-Architektur zwischen dem ursprünglichen und dem angestrebten Zustand, also in einem inkonsistenten, nicht einheitlichen Zustand verbleibt.

Auch wenn die Absicht erhalten bleibt, das Refactoring vollständig durchzuführen, ist es manchmal schwierig zu prüfen, ob alle Änderungen vollständig umgesetzt wurden. Wenn Änderungen vergessen werden, muss dies, gemäß der Zerlegung von Refactorings in elementare Einzelschritte, nicht zu Compiler- oder Laufzeitfehlern führen. Dennoch entspricht auch dann die Ist-Architektur nicht der Soll-Architektur.

zeitweilige Duldung von Verletzungen der Soll-Architektur

Nicht in jedem Fall ist es sinnvoll, ein großes Refactoring durchzuführen, wenn Anforderungen mit der Soll-Architektur nicht vereinbar sind. Wenn strenge zeitliche Vorgaben ein umfangreiches Refactoring nicht erlauben, muss als pragmatische Lösung eine Verletzung der Soll-Architektur für eine bestimmte Zeit in Kauf genommen werden. Dabei besteht die Gefahr, dass die Verletzung in Vergessenheit gerät und das Refactoring auch dann nicht durchgeführt wird, wenn dies wieder möglich wäre.

wechselnde Anforderungen

Die Entwicklung der Anforderungen im Laufe der Zeit spielt nicht nur bei Refactorings, sondern auch bei der eigentlichen Erweiterung der Funktionalität einer Software eine Rolle. Die Priorisierung von Features ändert sich und bereits in Teilen implementierte Erweiterungen werden abgebrochen. Während eine derart dynamische Entwicklung auf Ebene der Soll-Architektur noch relativ leicht berücksichtigt werden kann, verbleiben im Quellcode Strukturen, die nicht mehr benötigt werden und sich nicht im Rahmen der Soll-Architektur bewegen.

individuelle Bilder der Soll-Architektur

Man kann davon ausgehen, dass Entwickler nicht mit allen Aspekten einer Soll-Architektur gleichermaßen vertraut sind. Jeder hat ein eigenes Bild der Soll-Architektur und dieses Bild ist in Bereichen, die dem Entwickler gut bekannt sind, detailierter und schärfer als in anderen Bereichen. Weiter kann das Bild der Soll-Architektur veraltet sein, denn nicht alle Entwickler sind an der Weiterentwicklung der Soll-Architektur gleichermaßen beteiligt. Das Wissen über Änderungen muss also immer wieder verbreitet werden. Dabei ist kaum zu gewährleisten, dass allen Entwicklern der neuste Stand der Soll-Architektur bekannt ist.
Die Folge eines falschen, ungenauen oder veralteten Bildes des Soll-Architektur ist eine im Quellcode beschriebene Ist-Architektur, welche die Vorgaben der Soll-Architektur verletzt. Ist- und Soll-Architektur sind dann nicht kongruent.

3.3 Kongruenz von Ist- und Soll-Architektur als Qualitätsmerkmal

Wenn Vorgaben der Soll-Architektur im Quellcode verletzt werden, bedeutet dies häufig eine Verschlechterung der Ist-Architektur. Ein Beispiel für eine Verschlechterung ist eine erhöhte Kopplung von Komponenten. Die Folge ist eine geringere Verständlichkeit, Wiederverwendbarkeit und Erweiterbarkeit.

Unabhängig von der Qualität der Ist-Architektur ist aber auch die Kongruenz von Ist- und Soll-Architektur ein wichtiges Merkmal der inneren Qualität einer Software. Die Soll-Architektur ist Inhalt von Dokumentation und Kommunikation. Sie bildet so eine Basis für das individuelle, gedankliche Modell der Architektur eines Systems. Wenn die im Quellcode vorliegenden Strukturen nicht zu diesem Modell passen, ist die Verständlichkeit des Systems gemindert. Die Folgen sind weitere Verletzungen der Vorgaben der Soll-Architektur und eine Degenerierung der Architektur.

4 Bisherige Adressierung des Problems

4.1 Softwaretechnische Prinzipien

Die korrekte Umsetzung der Soll-Architektur im Quellcode zu erreichen, ist kein neu entdecktes Problem. Einige bereits im Entwicklungsprozess etablierten Prinzipien und Methoden fördern eine der Soll-Architektur entsprechende Implementierung und werden in diesem Abschnitt dargestellt.

Dokumentation

Eine gute Dokumentation der Soll-Architektur ist die Basis für eine erfolgreiche, also der Soll-Architektur entsprechende Implementierung. Eine umfangreiche Dokumentation der Soll-Architektur steht jedoch im Widerspruch zur Agilität des Entwicklungsprozesses. Im Entwicklungsalltag beschränkt man sich daher meist auf exemplarische Beschreibungen und einige wichtige Aspekte. Insbesondere die agilen Entwicklungsmethoden versuchen die Zahl und den Umfang von Dokumenten zu verringern, um den Aufwand zu senken, der mit der Erstellung und Aktualisierung der Dokumente verbunden ist.
Darüber hinaus kann auch eine vollständige Beschreibung in Dokumenten nicht gewährleisten, dass den Entwicklern alle Vorgaben der Soll-Architektur vertraut und auch präsent sind.

Geheimnisprinzip

Durch das Geheimnisprinzip als Bestandteil der Modularisierung wird erreicht, dass viele Änderungen in ihren Auswirkungen lokal beschränkt sind. Der Ausschnitt des Systems, den ein Entwickler bei einer Änderung verstehen muss, wird verringert. Damit sinkt die Zahl der zu berücksichtigenden Architekturvorgaben und die Gefahr einer Verletzung der Soll-Architektur.
Die Umsetzung des Geheimnisprinzipes impliziert die Beschreibung von Schnittstellen. Dafür stellt Java die Sprachelemente Interface, Klasse und Package zur Verfügung. Für Klassen kann eine Exportschnittstelle (Incomming Interface) explizit über Interfaces oder implizit über Sichbarkeitsmodifikatoren definiert werden. Klassen sind allerdings eher von niedriger Granularität und Kapseln daher nur einen Teil von Entwurfsentscheidungen. Die Verwendung von Packages ist in diesem Zusammenhang nur eingeschränkt möglich. Schnittstellen können nicht explizit beschrieben werden. Der Durchsetzung von Schnittstellen sind zudem bei der Schachtelung von Packages Grenzen gesetzt, denn die hierarchische Beziehung von Packages spielt bei der Verwendung der Sichtbarkeitmodifikatoren keine Rolle. So werden Packages nicht systematisch eingesetzt, um das Geheimnisprinzip auf der Ebene von Subsystemen oder Schichten umzusetzten. Auf dieser Ebene wird der Umfang zu berücksichtigenden Architekturvorgaben durch das Geheimnisprinzip daher kaum verringert.

Verwendung von Entwurfsmustern

Entwurfsmuster erfreuen sich großer Beliebtheit, seitdem sie in dem Buch von Gamma et Al. ([GHJV95]) beschrieben wurden. Sie sind so erfolgreich, weil sie zum einen Lösungen für immer wiederkehrende Probleme darstellen. Zum anderen erleichtern sie durch den Wiedererkennungseffekt das Verständnis von Software. Durch die Erkennung eines Enwurfsmusters werden Elementen (Objekten oder Klassen) bestimmte Rollen zugewiesen, die in bestimmten Beziehungen zueinander stehen. Damit werden Informationen über die Elemente transportiert, die nicht innerhalb der Programmiersprache ausgedrückt werden könnten. Die Wiedererkennung von Mustern fördert daher das Verständnis der Architektur und letztendlich die Kongruenz von Ist- und Soll-Architektur.

Verwendung von Modellarchitekturen

Eine Modellarchitektur macht eine Reihe von Vorgaben, für die Gestaltung der Architektur eines konkreten Systems. Mit der Entscheidung zur Verwendung einer Modellarchitektur, werden deren Vorgaben für das Projekt übernommen. Sofern den Entwicklern die Modellarchitektur vertraut ist, sind ihnen auch deren Vorgaben bekannt. Somit ist eine gute Basis für eine richte Umsetzung eines Grundgerüstes der Software geschaffen.
Leider ist auch dieses gemeinsame Wissen über die Vorgaben der Modellarchitektur noch kein Garant für deren Einhaltung. Siehe dazu siehe Abschnitt 4.2.

Collective Code-Ownership und Pair Programming

Wie die Verwendung einer Modellarchitektur, fördert die Praktizierung von collective code-ownerschip in Kombination mit pair-programming, die Verbreitung von Wissen über das System und dessen Soll-Architektur. Gleichzeitig kann collective code-ownership das Problem der Verletzung der Soll-Architektur auch verstärken, da Entwickler sich in Bereichen der Systems zurechtfinden müssen, die ihnen weniger vertraut sind.

4.2 Wissenschaftliche Arbeiten zur Prüfung der Vorgaben einer Soll-Architektur

Die im letzen Abschnitt aufgeführten Methoden und Prinzipien fördern die Übereinstimmung von Ist- und Soll-Architektur, sie reichen aber nicht aus, um diese in einem ausreichenden Maße zu gewährleisten. Diese Aussage basiert auf den Erfahrungen, die bei der Entwicklung des JCommSys gesammelt wurden, und deckt sich mit Erfahrungen anderer Autoren, die verschiedene Ansätze zur Lösung des Problems vorgestellt haben. In diesem Abschnitt werden einige dieser Ansätze vorgestellt.

Meyers, Duby und Reiss haben in [MDR93] einen Code-Orientierten Ansatz für Programme in C++ vorgestellt. Mit ihrer *C++ Constraint Expression Language* (CCEL) können Bedingungen für Design, Implementierung und Coding-Style ausgedrückt werden. Ein *constraint checker* erzeugt dann für ein Programm eine Liste von Verletzungen der contraints. Betrachtet werden bei diesem Ansatz allerdings ausschließlich Aspekte auf Code-Ebene, die sich beispielsweise aus der Speicherverwaltung von C++ ergeben. Die List der Contraint-Verletzungen kann helfen, einige problematische Stellen im Code zu finden.

Einen weiteren Ansatz haben Sefika, Sane, Campbell vorgestellt [SSC96]. Ihr *Pattern-Lint Tool* kombiniert eine logikbasierte statische Analyse und eine dynamische Analyse einer Software, um bei der Suche nach Abweichungen von ihrem *high-level design model* zu helfen. Bestandteil der Untersuchung kann die Einhaltung von Programmierrichtlinen, wie die Vermeidung von zyklischen Beziehungen zwischen Klassen sein. Auf der Ebene des Architekturmodells können Umsetzungen von Architekturstilen und Entwurfsmustern identifiziert und überprüft werden. Weiter kann die Kohäsion und Kopplung von Modulen betrachtet werden.

Fiutem und Antoniol beschreiben in [FA98] einen Ansatz, um die Übereinstimmung einer Implementierung mit einem gegebenen Design in OMT-Notation zu bestimmen. Das Verfahren bestimmt für C++-Programme aus dem Sourcecode die implementierten Klassen, sowie deren Attribute, Operationen und Beziehungen zueinander. Die Ergebnisse werden mit dem angegebenen Design verglichen und Abweichungen bestimmt.
Um bei leichten Abweichungen in der Benennung von Elementen noch den Bezug zwischen Spezifikation und Implementierung herstellen zu können, wird ein heuristisches Verfahren angewandt, dass eine *Distanz* zweier Elemente bestimmt.

Einen ganz ähnlicher Ansatz wird in [MNS01] vorgestellt. Murphy, Notkin und Sullivan beschrieben eine Technik, mit die Übereinstimmung von Design und Implementierung visualisiert werden kann. Dazu wird zunächst ein so genanntes *high-level model* erstellt. In diesem sind Module und deren Abhängigkeiten beschrieben. In einem zweiten Schritt wird ein Mapping zwischen Elementen der Implementierung und den Modulen erstellt. Grundlage für dieses Mapping kann die Position im Dateisystem oder die Benennung sein.
Toolgestützt wird ein *software reflexion model* generiert. In diesem sind Übereinstimmungen und Differenzen zwischen dem high-level model und der Implementierung beschrieben. Bei den Differenzen wird unterschieden zwischen ungewollten Abhängigkeiten, also Abhängigkeiten, die in der Implementierung, nicht aber im high-level model vorhanden sind und fehlenden Abhängigkeiten, die im high-lebel model, nicht aber in der Implementierung vorhanden sind.

Ein weiteres Verfahren zur Vermeidung einer Degenerierung von Software wird von Tvedt, Costa und Lindvall in [TCL02] beschrieben. Auch bei diesem Verfahren wird eine systematischen Analyse der Kopplung von Subsystemen durchgeführt. Zunächst werden angestrebte Eigenschaften der Implementierung als so genannte *design goals* formuliert. Mit Hilfe eines Software-Werkzeuges werden die Beziehungen zwischen Subsystemen, Packages und Klassen in der Implementierung bestimmt und tabellarisch zur Verfügung gestellt. In einem weiteren Schritt kann dann manuell geprüft werden, ob die ermittelten Eigenschaften des Systems gegen die Vorgeben verstoßen.

Bettina Karstens hat in ihrer Diplomarbeit [Kar05] Regeln der WAM-Modellarchitektur (siehe [Zül98]) in einem Regelkatalog zusammengetragen. Weiter hat sie untersucht, ob sich die Regeln mit dem Sotographen, einem Werkzeug zur statischen Codeanalsyse (siehe Abschnitt 6.2), prüfen lassen.
Ihr Regelkatalog enthält 94 Regel, die sich auf einzelne Elemente oder deren Beziehung zueinander beziehen. Etwa die Hälfte der Regeln kann mit Hilfe des Sotographen geprüft werden. Die Funde von Regelverletzungen müssen allerdings interpretiert werden, um *echte Funde* von *unechten Funden* zu unterscheiden. Unechte Funde können zustande kommen durch akzeptierte Ausnahmen und Schwachstellen der Abbildung von Elementen des Objektmodells auf Elemente der Musterarchitektur.
Bettina Karstens hat verschiedene Systeme auf die Einhaltung der Regeln untersucht und die Bedeutung der einer Prüfung der Vorgaben für eine Architektur unterstrichen. In allen untersuchten Systemen, einschließlich einer Beispielimplementierung für die WAM-Modellarchitektur, wurden Verletzungen der Regeln gefunden.

5 Lösungsansatz: Prüfung von Vereinbarungen und Regeln

Der in dieser Arbeit verfolgte Ansatz besteht darin, ausgewählte Vorgaben der Soll-Architektur in einer einheitlichen, regelhaften Form zu formulieren und in einem *Katalog* zusammenzutragen. In einem zweiten Schritt wird die Implementierung toolgestützt auf Einhaltung der Vorgaben überprüft.

Dieses Vorgehen kann als Ausweitung des in Abschnitt 4.2 vorgestellten Ansatzes von Bettina Karstens auf projektspezifische Elemente der Architektur verstanden werden. Der Katalog enthält dementsprechend *softwaretechnische Regeln*, Regeln der Musterarchitektur (*Architekturregeln*) und projektspezifische Vereinbarungen der Soll-Architektur (*Architekturvereinbarungen*).

5.1 Architekturvereinbarungen und Architekturregeln

Sowohl die Regeln, als auch die Vereinbarungen machen Aussagen darüber, wie bestimmte Elemente der Architektur gestaltet sein oder in Beziehung zueinander stehen sollen. Die begriffliche Trennung soll unterstreichen, dass sie dennoch einen unterschiedlichen Charakter haben:
Architekturregeln (also Regel einer Musterarchitektur) und softwaretechniche Regeln sind losgelöst von konkreten Projekten und Softwaresystemen gültig. Sie sind zeitliche weitgehend invariant. Demgegenüber werden Architekturvereinbarungen für ein bestimmtes Softwaresystem formuliert. Da sich die Soll-Architektur eines Systems entwickelt, verändern sich auch die Architekturvereinbarungen eines Systems im Laufe der Zeit.

Da die Prüfung vieler softwaretechnischer Regeln bereits mit Werkzeugen, wie dem Sotographen (siehe Abschnitt 6.2) möglich ist und Architekturregeln in [Kar05] behandelt werden, liegt der Fokus dieser Arbeit auf den Architekturvereinbarungen.

5.2 Schärfung der Architektur

Mit der Entwicklung der Soll-Architektur werden Vorgaben für die Implementierung vereinbart. Es ist nahe liegend, dass die Formulierung von Architekturvereinbarungen und die Aufnahme in den Katalog zeitnah vorgenommen werden sollten. Von einer engen Verzahnung dieser Aktivitäten ist ein Effekt der *Schärfung der Soll-Architektur* zu erwarten.

Damit ist gemeint, dass Aussagen über die Architektur, vor dem Hintergrund einer möglichen Formulierung als Architekturvereinbarung, stärker hinterfragt werden. Beispielsweise stellen sich basierend auf der Aussage „Services werden von Servlets benutzt.“ die Fragen, ob Services ausschließlich von Servlets benutzt werden oder ob es andere Regelmäßigkeiten in der Beziehung von Servlets und Services gibt.
Die Betrachtung der Architektur wird also von der Ebene spezifischer Problemlösung zu einer allgemeineren Ebene verschoben und führt so letztendlich zu einer detaillierten, vielschichtigeren Soll-Architektur.

5.3 Der Katalog als eine Ergänzung der Dokumentation

Der Effekt der Schärfung der Soll-Architektur kann auch auf die Dokumentation übertragen werden. Die Beschreibung der Strukturen von Software ist in agilen Projekten meist nicht sehr umfangreich. Sie beschränkt sich auf exemplarische Beschreibungen, beispielsweise in Form von Klassendiagrammen mit einige zentrale Elementen. Ein Regel- und Vorgabenkatalog kann die vorhandene Dokumentation um Aspekte ergänzen, die sich auf einer anderen Ebene befinden und in den Diagrammen nur schwer ausgedrückt werden können.

5.4 Prüfung der Vereinbarungen

Mit der toolgestützten Prüfung von Regeln und Vereinbarungen, soll Entwicklern eine Möglichkeit zur Verfügung gestellt werden, eine Implementierung auf Einhaltung der Vorgaben der Soll-Architektur, also die Kongruenz von Ist- und Soll-Architektur, zu prüfen. Bei einem hohen Automatisierungsgrad ist eine Prüfung dieser Kongruenz, vergleichbar mit der Durchführung von Unit-Tests, im Anschluss an jede Änderung denkbar.
Verletzungen von Architekturvereinbarungen können unterschiedlich interpretiert werden: Eine Verletzung kann ein Hinweis darauf sein, dass Entwickler einen Aspekt der Soll-Architektur falsch verstanden oder vergessen hat. Indirekt können aber auch Mängel und Widersprüche innerhalb der Soll-Architektur aufgedeckt werden.

5.5 Kriterien für die Aufnahme in einen Katalog

Bei der Anfertigung des Kataloges stellt sich die Frage, welche Vorgaben der Soll-Architektur als Architekturvereinbarung formuliert, in den Katalog aufgenommen und geprüft werden sollen. Der Sinn einer Prüfung ist an die Verletzbarkeit der Vorgabe gekoppelt. Die Prüfung einer Vorgabe ist nicht sinnvoll, wenn diese nur an wenigen, festen Stellen im Code eingehalten oder verletzt werden kann. Für solche Vorgaben ist eine pragmatische Lösung, wie eine einmalige Umsetzung und gegebenenfalls die Wahrung der Umsetzung durch einem Kommentar im Code, einfacher. Dies soll an einem Beispiel aus dem JCommSy veranschaulicht werden:

Im JCommSy gibt es unterschiedliche Ausprägungen von Items, wie zum Beispiel Termine, Materialien und Anmerkungen. Als Bestandteil der Soll-Architektur wurde vereinbart, dass die Klassen aller konkreten Items eine ItemID haben müssen. Da die Zuordnung der ItemID in einer Basisklasse `Item` vorgesehen wurde, ist sie auf Codeebene lokal beschränkt. Der Aufwand für die Formulierung und Prüfung einer Architekturvorgabe scheint daher nicht gerechtfertigt.

Dem gegenüber steht die Vereinbarung, dass die Methode `setItemID()` der Klasse `Item` nur von Mappern verwendet werden soll. Diese Vereinbarung könnte an vielen, eventuell noch unbekannten Stellen im Code verletzt werden. Die Wahrung der Vereinbarung ist daher schwieriger und eine Prüfung gerechtfertigt. (Eine entsprechende Architekturvereinbarung ist in Abschnitt 7.4 formuliert.)

Als ein Kriterium für die Aufnahme einer Vereinbarung in den Katalog wurde in dieser Arbeit also die *nicht lokal beschränkte Verletzbarkeit* der Vereinbarung verwendet. Dieses Kriterium kann jedoch allenfalls als Richtlinie dienen, denn viele Vereinbarungen lassen eine unterschiedliche Interpretation zu, wie am bereits eingeführten Item-Beispiel verdeutlicht werden kann.
Die Vereinbarung, das die Klassen aller konkreten Items von der Basisklasse `Item` erben, kann unterschiedlich betrachtet werden. Wenn sie in einzelne Aussagen, wie „Die Klasse Appointment erbt von Item.“ und „Die Klasse Material erbt von Item.“, zerlegt wird, ist deren Prüfung gemäß dem Kriterium nicht sinnvoll. Verallgemeinert als „Die Klassen aller konkreten Items erben von der Klasse Item.“ wäre für die Aussage das Kriterium erfüllt.

Ein weiteres Kriterium ist die *Stabilität* der Vereinbarung. In einem iterativen Entwicklungsprozess ändern sich die Beziehungen zwischen Klassen häufig. So ist es nicht sinnvoll, jede vereinbarte Abhängigkeit zwischen Klassen als Architekturvereinbarungen in einen Katalog aufzunehmen, da der Aufwand für die Aktualisierung des Kataloges in einem zu ungünstigen Verhältnis zum erwarteten Nutzen steht.

Letztendlich kann auch die *Prüfbarkeit* als Kriterium für die Aufnahme in den Katalog verwendet werden. Von den Regel, die Bettina Karstens in den Katalog der WAM-Regeln aufgenommen hat, kann etwa die Hälfte nicht auf der Ebene einer Codeanalyse geprüft werden, weil beispielsweise eine fachliche Interpretation von Code-Elementen nötig ist.

Für das JCommSy wurden ausschließlich Architekturvereinbarungen formuliert, die geprüft werden können, da die Prüfung ein zentraler Bestandteil des verfolgten Ansatzes ist. Ein Beispiel für eine nicht prüfbare Vereinbarung wurde bereits erwähnt. Es kann nicht geprüft werden, ob die Klassen aller Items von einer Klasse `Item` erben, weil auf Codeebene nicht geklärt werden kann, welche Klassen Items repräsentieren.

6 Verwendete Werkzeuge

6.1 SonarJ

Das Werkzeug SonarJ (siehe [Son]) der Firma Hello2Morrow ermöglicht die Überprüfung von Software-Architekturen auf einem hohen Abstraktionslevel. Dazu muss zunächst ein logisches Architekturmodell erstellt werden, dessen Einhaltung dann geprüft werden kann.
In einem Architekturmodell werde Subsysteme beschrieben, die in zwei Dimensionen angeordnet werden können. In der ersten Dimension werden die Subsysteme einzelnen Schichten, in der zweiten Dimension einzelnen „Scheiben“ (engl. vertical-slices) zugewiesen werden. Ein weiterer Bestandteil des Architekturmodells ist die Beschreibung von Abhängigkeiten zwischen den Elementen. Ausgedrückt werden können Beziehungen zwischen Schichten, zwischen Scheiben und zwischen Subsysteme innerhalb einer Schicht. Weiter können Schnittstellen für Subsysteme definiert werden. Die Beschreibung des Architekturmodells kann in einer XML-Datei oder über grafisches Werkzeug erstellt werden.

Damit SonarJ den Code auf Einhaltung dieses Architekturmodells prüfen kann, muss eine Abbildung des physikalische, auf das logische Modell beschrieben werden. Diese Abbildung erfolgt über die Zuweisung von Packages zu Subsysteme.

SonarJ kann zwar auch als eigenständige Anwendung laufen, eine besondere Stärke ist jedoch die Integration in die Entwicklungsumgebung. Durch sie gibt SonarJ den Entwicklern eine unmittelbare Rückmeldung über Verletzungen des beschriebenen Architekturmodells. SonarJ listet Verletzung auf und markiert die entsprechende Orte im Quellcode.

6.2 Sotograph

Der Sotograph (siehe [Sot]) ist ein kommerzielles Analysewerkzeug der Software-Tompgraphy GmbH. Der Name Sotograph ist eine Zusammensetzung aus Software und Tomograph und spiegelt die Anlehnung an die Grundidee der medizinischen Computer-Tomographen wider, d.h. er extrahiert zunächst so viele Daten über das untersuchte System wie möglich, bevor die eigentliche Analyse durchgeführt wird.

Zunächst gewinnt der Sotograph aus dem Bytecode Informationen über Methoden, deren Aufrufe, Klassen, Dateien und Packages und legt diese in einer relationalen Datenbank ab. Darüber hinaus können vom Benutzer Subsysteme und Schichten als höhere Abstraktionsebenen in einem Architekturmodell definiert werden.

Für Beziehungen zwischen Subsystem und zwischen Schichten können in dem Modell Restriktionen beschrieben werden. Der Sotograph kann, auf Basis der gewonnen Informationen, Verletzungen dieses Modells finden und anzeigen. Weiter liefern metrikbasierte Analysen Hinweise auf Architekturprobleme und die Verletzung softwaretechnischer Regeln. So können beispielsweise Klasse, Packages und Subsystemen gefunden werden, die in zyklischen Beziehungen zueinander stehen.

Für eine kontinuierliche Überwachung bietet der Sotograph eine Unterstützung durch die so genannte Trend-Analyse. Diese ermöglicht die Untersuchung verschiedener Versionen eines Systems und die Betrachtung von zeitlichen Entwicklungen.

Der Sotograph kann eingesetzt werden, um Lösungen für erkannte Probleme zu evaluieren. Mit der so genannten „What-If-Analyse“ können Änderungen, wie das Verschieben einer Klasse in ein anderes Package, simuliert werden. Die Änderungen werden dazu innerhalb der Datenbank durchgeführt und der simulierte Zustand kann auf neue Probleme hin untersucht werden.

6.3 JDepend

Mit JDepend (siehe [JDea]) können Abhängigkeiten zwischen Packages analysiert werden. JDepend bietet zur Anzeige der Abhängigkeiten eine Oberfläche, die auch als Eclipse-Plug-in verfügbar ist (siehe [JDeb]). Über eine Programmierschnittstelle können in JUnit-Tests Prüfungen der Abhängigkeiten formuliert werden.
Die Abhängigkeiten zwischen Klassen werden von JDepend über die Betrachtung von Import-Statements ermittelt. Da dieses Verfahren nicht zuverlässig ist und eine Betrachtung auf Klassenebene nicht möglich ist, wird dieses Tool in dieser Arbeit nicht weiter verwendet.

7 Architekturvereinbarungen des JCommSys

In diesem Abschnitt wird zunächst der Identifizierungsprozess für die Vereinbarungen des JCommSys dargestellt. Es werden die identifizierten Architekturvereinbarungen mit ihrem Hintergrund vorgestellt und die Prüfung der Vereinbarunge, sowie deren Ergebnisse beschrieben.

7.1 Identifizierung der Architekturvereinbarungen

Ein Element des hier beschriebenen Ansatzes ist die Formulierung von Architekturvereinbarungen während der Aktivitäten zur Entwicklung der Soll-Architektur. In der Zeit der Entstehung dieser Arbeit wurde die Soll-Architektur des JCommSys jedoch nur in geringem Maße weiterentwickelt und nur ein Teil der Architekturvereinbarungen ist dabei formuliert worden. Stattdessen wurden die meisten der hier vorgestellten Vereinbarungen nachträglich identifiziert. Damit ist gemeint, dass sie Ergebnis einer gezielten Analyse der Soll-Architektur sind.

Einige Ausschnitte der Soll-Architektur des JCommSy sind in Dokumenten beschrieben. Diese Dokumente bilden eine Grundlage für die Analyse. Andere Aspekte wurden nicht in Dokumenten festgehalten. Zum Teil waren Elemente der Soll-Architektur durch die Teilnahme an deren Entwicklung bekannt oder konnten in Gesprächen mit Entwicklern zusammengetragen werden.

7.2 Vereinbarungen zu den Schichten

Schichten des JCommSys

Auf einem hohen Abstraktionsniveau werden die Elemente der Soll-Architektur in Bereiche gegliedert, die sich über die gesamte fachliche Breite erstrecken. Diese Bereiche werden als Schichten bezeichnet. Die Schichten und deren Beziehung zueinander sind in Abbildung 1 dargestellt.

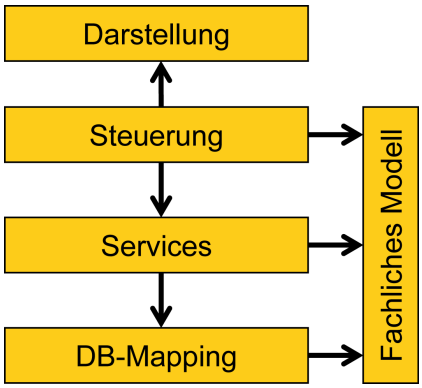

Abbildung 1: Die Schichten des JCommSy

Der Abbildung ist zu entnehmen, dass das JCommSy nicht um eine strikte Schichtenarchitektur handelt, bei der die Verwendung auf Elemente der nächsten Schicht beschränkt ist. Vielmehr handelt es sich um eine nicht strikte Schichtenarchitektur, bei der der Zugriff auf mehrere, niedrigere Schichten möglich ist. Dieser Zugriff ist allerdings zusätzlichen Einschränkungen unterworfen.

Im Folgenden werden die einzelnen Schichten vorgestellt.

Steuerung: Diese Schicht regelt das Verhalten der Anwendung und die Interaktion mit dem Benutzer. Hier befindet sich das CommSyServlet, dass den zentralen Einstiegspunkt der Web-Anwendung darstellt. Das CommSy-Servlet delegiert die Anfragen an spezialisierte Servlets, wie

das AppointmentServlet, in denen die eigentliche Funktionalität der Rubriken des CommSy implementiert ist.

Darstellung: Die Darstellung erfolgt mit Hilfe von Java Server Pages (JSPs), die Bestandteil der Darstellungsschicht sind. Die Servlets leiten die Anfragen nach der inhaltlichen Verarbeitung an die JSPs weiter, damit diese die Ergebnisse darstellen können. Die für die Darstellung relevanten Informationen werden als Java-Beans an die JSPs übergeben.

Fachliches Modell: In dieser Schicht sind die fachlichen Einheiten des Anwendungsbereiches wie Termine, Materialien, Anmerkungen und Personen modelliert, die Gegenstand der Arbeit mit dem CommSy sind. Diese Elemente werden im CommSy Items genannt. Neben den Items sind auch die Fachwerte Bestandteil dieser Schicht.

Services: Unterhalb der Steuerungsschicht sind die Services in einer Schicht zusammengefasst. Zum aktuellen Entwicklungsstand, ist die Aufgabe der meisten Services bestimmte Items zu verwalten und der Steuerungsschicht zur Verfügung zu stellen.

DB-Mapping: Die Items werden in einer relationalen Datenbank abgelegt. Das objekt-relationale Mapping wird nicht in den Services, sondern in dieser Schicht in den Mapper-Klassen implementiert.

Architekturvereinbarungen

Die zwei Kernpunkte der Schichtenarchitektur des JCommSys können als Architekturvereinbarungen formuliert werden:

Vereinbarung 1.1: Es gibt die Schichten Steuerung, Darstellung, fachliches Modell, Services und DB-Mapping. Jede Klasse ist einer dieser Schichten zugeordnet.

Vereinbarung 1.2: Nur die folgenden Abhängigkeiten zwischen Schichten sind erlaubt:
- Steuerung benutzt Darstellung, fachliches Modell und Services
- Services benutzt fachliches Modell und DB-Mapping
- DB-Mapping benutzt fachliches Modell

Prüfung

Die Prüfung dieser Vereinbarungen kann mit dem Sotographen auf unterschiedliche Arten durchgeführt werden. Es können Queries, also Anfragen an die Datenbank, formuliert werden, die als Ergebnis Verletzungen der Vereinbarungen liefern. Diese Methode wird für die Vereinbarungen in Abschnitt 7.4 verwendet.
Für die Einhaltung der vereinbarten Schichtenarchiktur kann im Sotographen ein Architekturmodell mit Schichten und deren Abhängigkeiten definiert werden. In einem zweiten Schritt wird die Implementierung auf Einhaltung dieses Modelles geprüft. Diese Methode wird auch von SonarJ unterstützt. Im Folgenden wird die Prüfung mit beiden Tools beschrieben.

Den Tools liegt die Annahme zugrunde, dass Schichten in Subsysteme untergliedert sind und diesen wiederum Packages zugeordnet werden. Da in der Soll-Architektur des JCommSys zunächst keine Subsysteme vorgesehen sind, wird im Architekturmodell jeder Schicht genau ein Subsystem zugewiesen.
Die Schichtenarchitektur spiegelt sich in der Packagestruktur (siehe Abbildung 2) des JCommSys wider. Zu jeder Schicht existieren ein oder mehrere Packages, die auch in der Beschreibung des Architekturmodells der Schicht (genauer: dem Subsystem) zugewiesen werden.

Es gibt allerdings auch Packages, die sich keiner Schicht zugeordnet werden können. Das Package `de.unihamburg.informatik.jcommsy.migration` enthält eine Reihe von Elementen, die für den integrierten Betrieb von JCommsy und PHP-Commsy (siehe Abschnitt 2.4), also nur während der

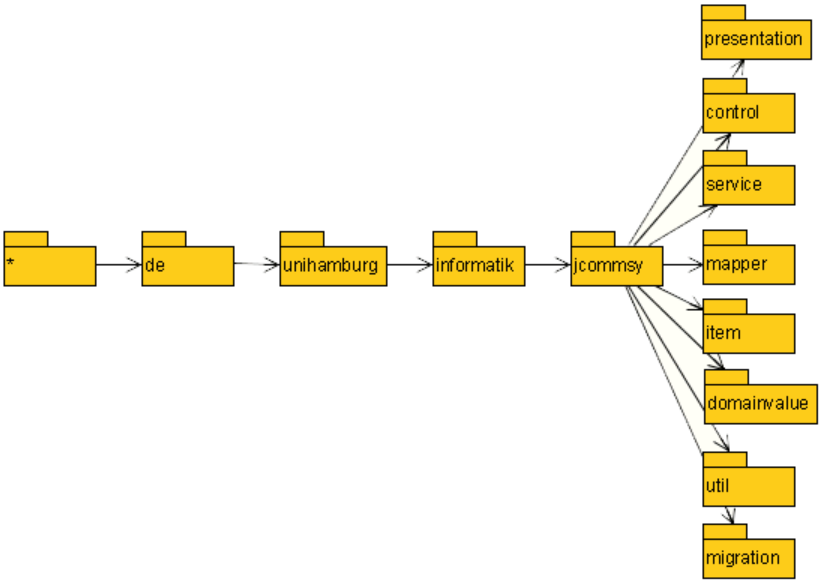

Abbildung 2: Die Packagestruktur des JCommSys

Migration nötig sind.

Auch dieser Teil des Systems ist unter Berücksichtigung Schichtenarchitektur entwickelt worden. Bei den Klassen innerhalb dieses Packages handelt es sich beispielsweise um Services und Mapper, die Schichten zugeordnet werden können.
Im Sotographen ist diese Zuordnung über den Mechanismus der What-If-Analyse möglich. Das heißt, in der Datenbank des Sotographen wird eine Aufteilung des Packages in einzelne Packages durchgeführt. SonarJ bietet keinen solchen Mechanismus. Damit die Elemente bei einer Prüfung berücksicht werden können, müsste also ein Refactoring durchgeführt werden.
(Die Gründe, die zur Erstellung des `migration`-Packages geführt haben, sind in Abschnitt 7.3 dargestellt.)

Auch das Package `de.unihamburg.informatik.jcommsy.util` kann keiner Schicht zugeordnet werden. Im Gegensatz zum `migration`-Package gilt dies jedoch auch für die vier Klassen in dem Package. Die Klassen `CommsyException`, `ErrorContainer`, `GetItemsResult` und `LinkFactory` verletzen daher die Vereinbarung 1.1.

Die Klassen wurden zur die Erfüllung bestimmter Anforderungen erstellt. Nach der Entdeckung der Architekturverletzungen ist zu klären, ob es einen Weg gibt, diese Anforderungen im Rahmen der Soll-Architektur und den damit verbundenen Vereinbarungen zu erfüllen, oder ob die Soll-Architektur dazu geändert werden muss. Eine solche Änderung könnte beispielsweise die Einführung einer gemeinsam genutzten Schicht sein.

Verletzungen der ersten Architekturvereinbarung wurden also dadurch entdeckt, dass bestimmte Elemente keiner Schicht zugeordnet werden konnten. Die verwendeten Werkzeuge unterstützen den Entwickler dabei, durch die Auflistung dieser nicht zugeordneten Elemente.

Die Prüfung der Einschränkungen für die Abhängigkeiten zwischen Schichten (Vereinbarung 1.2) ergab insgesamt zwölf Verletztungen. Gezählt wurden dabei Paare von Klassen, deren Beziehung gegen die Vereinbarungen verstößt.

Abhängige Schichten	Anzahl	Erklärung
Darstellung ⟶ Fachliches Modell	9	Beans, die Items oder Domainvalues enthalten
Darstellung ⟶ DB-Mapping	1	JSP, die den Zustand der Datenbankverbindung erfragt
DB-Mapping ⟶ Services	2	Mapper, die einen falschen Logger verwenden

Tabelle 1: Verletzungen der Vereinbarung zu Abhängigkeiten von Schichten

7.3 Vereinbarungen zur vertikalen Unterteilung

Vertikale Unterteilung des JCommSy

Orthogonal zu den eher technischen Schichten kann in Systemen eine vertikale Unterteilung vorgenommen werden, die durch eine fachliche Gliederung des Anwendungsbereiches bestimmt ist. Eine solche Unterteilung ist auch die Basis für die Abgrenzung der Teilziele bei der vertikalen Migration des CommSys (siehe Abschnitt 2.3).

Die Basis für die fachliche Unterteilung des CommSys bilden die Rubriken. Das JCommSy implementiert bisher die Terminrubrik, kleine Teile andere Rubriken und gemeinsame Basisfunktionalitäten. Auf der Ebene der Architektur spielt die Unterteilung in Rubriken bisher keine sehr große Rolle. Es wurden keine Vereinbarungen getroffen, die Abhängigkeiten zwischen Elementen unterschiedlicher Rubriken regeln. Dieser Aspekt wird erst mit der Migration einer zweiten Rubrik an Bedeutung gewinnen.

Vertikale Unterteilungen sind nicht nur fachlich motiviert. Häufig gibt es auch technische Unterteilungen des Systems, die sich über mehrere Schichten erstrecken. So gibt es im JCommSy die Gruppe von Elementen, die nur für die Migration eine Bedeutung haben. Diese Elemente sind zuständig für die Integration in das PHP-CommSy, und damit vor allem für den Austausch der Session. Nach der Migration werden sie aus dem System wieder entfernt. Daher ist eine klare Abgrenzung dieser Elemente wichtig und begründet die vertikale Unterteilung in die Bereiche `JCommSy` und `Migration`.

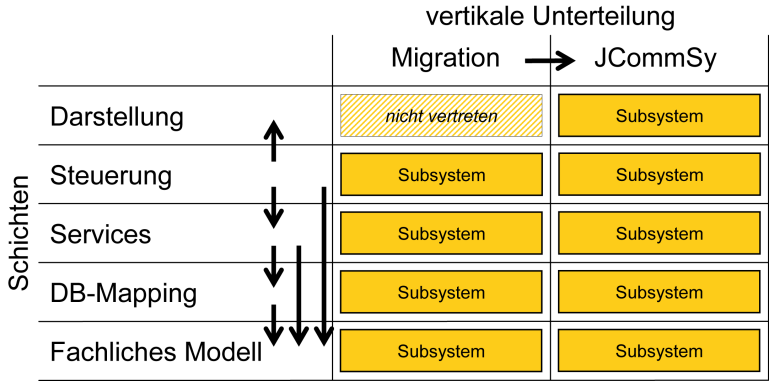

Abbildung 3: Vertikale und horizontale Unterteilung des JCommSys

Architekturvereinbarungen

Vereinbarung 2.1: Das JCommSy-System ist vertikal in die Bereiche `JCommSy` und `Migration` gegliedert. Jede Klasse ist einem dieser Bereiche zugeordnet.

Vereinbarung 2.2: Elemente des Bereiches `JCommSy` dürfen nicht von Elementen des Bereiches `Migration` abhängen.

Vertikale Unterteilung im Quellcode

Auf Codeebene verkörpert das Package `de.unihamburg.informatik.jcommsy.migration` die Abgrenzung der Migrationselemente. In diesem Package sind alle Klassen des Bereiches `Migration` enthalten.
Das Package wurde eingeführt, nachdem ungewollte Abhängigkeiten zu den Migrationselementen entstanden waren. Durch die Einführung des Packages und die Nutzung der Sichbarkeitsmodifikatioren wird der Zugriff auf diese Klassen unterbunden. Lediglich die Klasse `MigrationFilter` kann ausserhalb des `migration`-Packages verwendet werden, da dem Application-Server der Zugriff auf diese Klasse möglich sein muss.

Durch Einführung des `migration`-Packages ist die Einhaltung der Vereinbarungen 2.1 automatisch gewährleistet. Die Zugehörigkeit zum `Migration`-Bereich ergibt sich aus der Zugehörigkeit zu dem `migration`-Package. Alle übrigen Klassen sind der dem `JCommSy`-Bereich zugeordnet. Auch die Einhaltung der Vereinbarung 2.2 ist (abgesehen von der `MigrationFilter`-Klasse) sichergestellt.

Einige Nachteile der verwendeten Lösung wurde bereits in Abschnitt 7.2 erwähnt. Das `migration`-Package widerspricht dem Ansatz, die Packagestruktur an den Schichten auszurichten. Die Lösung, das Package zur Durchsetzung der vertikalen Unterteilung zu verwenden, erlaubt es nicht, dass Package weiter zu untergliedern, da die Sichtbarkeitsmodifikatoren die Schachtelung von Packages nicht unterstützen.

Das `migration`-Package ist ein Komprimiss zwischen unterschiedlichen Interessen. Dieser Gegensatz könnte durch eine Prüfung der Vereinbarungen mit Hilfe der bereits vorgestellten Werkzeuge aufgehoben werden. Die Packagestruktur könnte dann beispielsweise auf erste Ebene die Schichten und auf zweiter Ebene die vertikale Trennung repräsentieren.

Prüfung

Für die Prüfung der Vereinbarungen mit SonarJ muss die Beschreibung des Architekturmodells um die *vertical-slices* `Migration` und `JCommSy` ergänzt werden. Innerhalb der Schichten ergeben sich auf diese Weise jeweils zwei Subsysteme, wie in Abbildung 3 dargestellt ist. Diesen Subsystemen müssen dann die Packages zugewiesen werden. Für die Prüfung mit SonarJ, muss die Packagestruktur daher sowohl die horizontale Teilung des Systems (in Schichten), als auch die vertikale Teilung wider spiegeln.

Der Sotograph besitzt zwar kein explizites Konzept für eine vertikale Unterteilung des Systems, eine Prüfung ist dennoch möglich. Dazu muss ein *Graph Archtecture Model* statt eines *Layered Architecture Model* definiert werden. Dadurch verzichtet man auf die Beschreibung von Schichten, kann aber komplizierte Beziehungesgeflechte zwischen Subsystemen modellieren. Die dabei beschrieben Subsysteme sind dieselben, wie bei der Verwendung von SonarJ.

Da die Vereinbarungen durch die Packagestruktur gewahrt werden, konnten bei der Prüfung keine Verletzungen dieser Vereinbarungen gefunden.

7.4 Vereinbarungen zu einzelnen Klassen und Methoden

Dieser Abschnitt beschreibt Vereinbarungen auf der Ebene von Klassen und Methoden. Diese ergeben sich teilweise aus dem beabsichtigen Zusammenspiel von Schichten oder der Klassen innerhalb einer Schicht. Ein Beispiel ist das Zusammenspiel von Services, Mappern und Items.

Services, Mapper und Items

Items repräsentieren die fachlichen Einheiten, die Gegenstand der Arbeit mit dem CommSy sind. Sie werden von Services verwaltet. Items werden in einer relationalen Datenbank abgelegt. Das eigentliche Speichern und Laden, sowie das damit verbundene objekt-relationale Mapping wird von den Services an die Mapper-Klassen delegiert.
Ein Item das zum ersten Mal gespeichert wird, bekommt vom Mapper eine systemweit eindeutigen Identifier zugewiesen. Da sich diese `ItemID` nicht mehr ändert, wurde die folgende Architekturvereinbarung getroffen:

Vereinbarung 3.1: Die Methode `Item.setItemID(ItemID iid)` darf nur von Elementen der Schicht `Mapper` aufgerufen werden.

Steuerung und Darstellung

Das Zusammenspiel der Steuerungs- und Darstellungsschicht wird durch die Verarbeitung von HTTP-Anfragen charakterisiert. Die Steuerungsschicht nimmt Anfragen entgegen und führt, unter Verwendung der darunter liegenden Schichten, die fachliche Verarbeitung durch. Anschließend werden „Java Beans"-Objekte erstellt, in denen die für die Anzeige relevanten Daten abgelegt werden. Diese Objekte werden dann als Parameter an die Darstellungssschicht übergeben und dort verwendet, um die HTML-Seiten mit Inhalt zu füllen. Der Lebenszyklus der Bean-Objekte endet mit der Abarbeitung einer Anfrage. Der Parameter-Charakter des Bean-Objekte begründet die folgende Architekturvereinbarung:

Vereinbarung 3.2: Klassen der Darstellungsschicht, deren Name auf `Bean` endet, werden als Beans bezeichnet. Der Aufruf von Set-Methoden an Beans ist nur innerhalb der Steuerungsschicht erlaubt.

Konstruktoren und Methoden für Testklassen

Bei der Erstellung von JUnit-Tests ist es gelegentlich notwendig die zu testende Klasse um bestimmte Konstruktoren oder Methoden zu erweitern, die ausschließlich Testzwecken dienen. Diese Konstruktoren bzw. Methoden werden üblicherweise mit einem Kommentar markiert, um eine Verwendung durch reguläre Klassen zu verhindern. Es erscheint darüber hinaus sinnvoll, diesen Aspekt bei der Formulierung und Prüfung von Architekturvereinbarung zu berücksichtigen.

Vereinbarung 3.3: Die folgenden Konstruktoren und Methoden dürfen nur von Testklassen verwendet werden:
- `AppointmentService(AnnotationMapper, AppointmentMapper)`
- `CommsySessionService(CommsySessionMapper)`
- `CampusService(CampusMapper)`
- `GroupService(GroupMapper)`
- `LinkItemService(LinkMapper, TypeResolver)`
- `MaterialService(MaterialMapper)`
- `ModifierService(ModifierMapper)`
- `DBConnection.unsetConnection()`

ServiceRegistry und Services

Die ServiceRegistry verwaltet die Services. Sie ist als Singleton (siehe [GHJV95]) implementiert und kann so relativ einfach an verschieden Stellen im System genutzt werden, um eine Referenz auf einen konkreten Service zu erhalten. Die ServiceRegistry ist auch dafür zuständig, von jedem

Service ein Exemplar zu erstellen und gewährleistet, dass von jedem Service genau ein Exemplar existiert.

Vereinbarung 3.4: Exemplare der Services dürfen nur von Testklassen und der ServiceRegistry erzeugt werden.

Prüfung

Die Prüfung der in diesem Absatz beschriebenen Vereinbarungen ist nicht über die Prüfung eines im Sotographen oder in SonarJ beschriebenen Architekturmodells möglich. Der Sotograph bietet allerdings einen anderen Weg, um eine Analyse von Methodenaufrufen durchzuführen. Für die Prüfung der Vereinbarungen können im Sotographen eigene Queries beschrieben werden. Dabei handelt es sich um SQL-Anfragen an die Datenbank, in welcher der Sotograph die Daten abgelegt hat, die er über das untersuchte System erhobenen hat. Die Query zur Vereinbarung 3.2 ist in Anhang A beschrieben.

Die in diesem Absatz beschriebene Vereinbarungen werden im JCommSy eingehalten.

7.5 Ausblick auf weitere Architekturvereinbarungen

Die letzten Abschnitte haben gezeigt, dass bei der Analyse der Soll-Architektur einige Architekturvereinbarungen identifiziert und formuliert werden konnten. Bei der Analyse hat sich allerdings herausgestellt, dass für viele Bereiche der Architektur, noch keine Vereinbarungen getroffen wurden. Die Soll-Architektur macht in diesen Bereichen also nur sehr lockere Vorgaben. Dies ist dadurch zu erklären, dass die „Vervollständigung der Termin-Rubrik“, also die Migration aller Features der Rubrik, eine höhere Priorität hatte, also die Entwicklung der Soll-Architektur.

In diesem Abschnitt werden Aspekte der Architektur beschrieben, bei denen eine Schärfung der Architektur und die Formulierung weiterer Architekturvereinbarungen zu erwarten ist oder sich bereits abzeichnen.

Die Rubriken des CommSys zeichnen sich dadurch aus, dass sie stark und teilweise wechselseitig vernetzt sind. Mit der Migration weiterer Rubriken steigt daher der Bedarf an einer fachlichen, vertikalen Unterteilung des Systems auf Architekturebenen. Es ist zu klären, welche Elemente Basisfunktionalität implementieren und welche Elementen zu bestimmten Rubriken gehören. Weiter sind Formen der Interaktion zwischen diesen Teilen zu entwickeln und festzulegen. Diese Vereinbarungen müssen insbesondere für die Schichten Steuerung und Services getroffen werden, in denen sich die Vernetzung zwischen fachlich unterschiedlichen Bereichen bereits abzeichnet.

Mit der Migration weiterer Rubriken steigt die Zahl der Elemente innerhalb der Packages. Diese müssen unterteilt werden. Im Gegensatz zum untersuchten Entwicklungsstand sind dann die Abhängigkeiten zwischen Packages nicht mehr durch die Vereinbarungen zu den Schichten geregelt. Üblicherweise werden für die Packages innerhalb einer Schicht Vereinbarungen getroffen, die festlegen, wie die Klassen auf gleicher Ebene und auf unterschiedlichen Ebenen der Packagehierarchie miteinander in Beziehung stehen dürfen.

Bei einer steigenden Zahl von Klassen in den Schichten kann es sinnvoll sein, für einzelne Schichten Schnittstellen zu vereinbaren und deren Einhaltung zu prüfen. Dies ist von Bedeutung, sobald für die Schichten nicht mehr erkennbar ist, welche Elemente von einer anderen Schicht zu verwenden sind, und welche Elemente nur für die Nutzung innerhalb der Schicht vorgesehen sind.

Einige Anzeichen, für den Bedarf an einer Schärfung der Soll-Architektur sind bereits bei dem untersuchten Entwicklungsstand zu entdecken. Ein Beispiel für solche Anzeichen sind unterschiedliche Lösungen, die an verschiedenen Stellen für sehr ähnliche Probleme verwendet werden. So gibt es unterschiedliche Varianten für die Initialisierung der Mapper-Klassen.

8 Architekturegeln des JCommSy

Der in dieser Arbeit verfolgte Lösungsansatz sieht vor, neben den projektspezifischen Architekturvereinbarungen auch softwaretechnische Regeln und Regeln einer verwendeten Modellarchitektur in den Katalog aufzunehmen und in die Prüfung einzubeziehen.

8.1 Regeln der WAM-Modellarchitektur

Das JCommSy wurde zwar nicht vollständig nach dem Werkzeug-Material-Ansatz (siehe [Zül98]) entwickelt, einige Elemente der WAM-Modellarchitektur wurde jedoch verwendet. Die im JCommSy verwendeten Items folgen der Material-Metapher und auch die Services und Fachwerte entsprechen denen in WAM.

Aus dem von Bettina Karstens (siehe Abschnitt 4.2) zusammengestellten Regelkatalog gelten die Regeln, die sich nur auf diese Elemente beziehen auch für das JCommSy. In diesem Abschnitt werden die Regeln dargestellt, für die sich eine Prüfung mit Hilfe des Sotographen als möglich erwiesen hat.

WAM-Regel 1:	Services gehen mit Materialien um.
WAM-Regel 2:	Fachwert-Klassen bieten keine Operationen an, die es erlauben, das Fachwert-Objekt zu verändern.
WAM-Regel 3:	Fachwerte dürfen nur auf Fachwerte zugreifen.
WAM-Regel 4:	Materialien dürfen nur auf Fachwerte und Materialien zugreifen.

Prüfung

Im JCommSy wird ausschließlich die zweite WAM-Regel verletzt. Die Klassen `AppointmentFilterPattern`, `GroupFilterPattern`, `MaterialFilterPattern`, `TopicFilterPattern` und `RequestContext` wurden ursprünglich nicht als Fachwerte modelliert. Mittlerweile werden sie aber als solche angesehen und entsprechend verwendet. Die Schnittstellen der Klassen enthalten allerdings noch Methoden, die erlauben, die Objekte zu verändern und stellen daher eine Verletzung der Regel dar.

8.2 Softwaretechnische Regeln

Für objektorientierte Systeme im Allgemeinen und im Speziellen für die jeweiligen Programmiersprachen gibt es softwaretechnische Regeln, die ebenfalls in den Katalog mit aufgenommen werden können. Für das JCommSy wurde zwei Regeln als Beispiel ausgewählt. Mit Sicherheit können weitere softwaretechnische Regeln gefunden werden.

SWT-Regel 1:	Es gibt keine zyklischen Beziehungen zwischen Packages.
SWT-Regel 2:	Es gibt keine zyklischen Beziehungen zwischen Klassen.

Prüfung

Bei der Prüfung auf zyklische Beziehungen wurde ein Klassenzyklus zwischen Services und der ServiceRegistry entdeckt (siehe Abbildung 4). Die ServiceRegistry kennt alle Services. Der Zyklus entsteht, da ein Service sich an die ServiceRegistry wendet, um eine Referenz auf einen anderen Service zu erhalten.

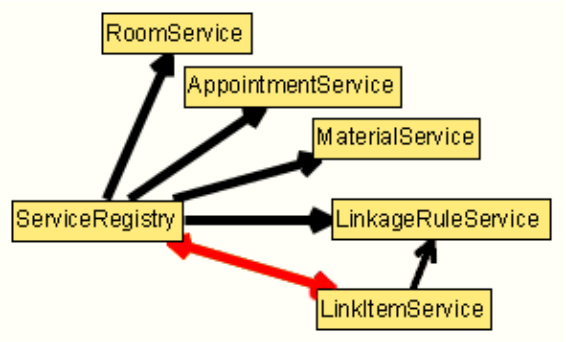

Abbildung 4: Klassenzyklus auf der Ebene der Services

Einen weiteren Klassenzyklus bilden die Klassen `CommsyServlet` und `LinkFactory`. Dieser Zyklus ist gleichzeitig auf der Grund für den einzigen Packagezyklus im JCommSy. An diesem sind das `util`-Package und das `control`-Package beteiligt.

9 Zusammenfassung und Ausblick

Diese Arbeit beschäftigt sich mit dem Problem, dass bei der Implementierung von Software Vorgaben der Soll-Architektur verletzt werden und sich so die innere Qualität verschlechtert. Nach einer Beschreibung dieses Problems und einer Bestandsaufnahme zu dem Thema wurde ein Lösungsansatz vorgestellt. Dieser Ansatz sieht vor, ausgewählte Aspekte der Soll-Architektur als Architekturvereinbarungen in einer einheitlichen Weise zu formulieren, in einem Katalog zusammenzutragen und toolgestützt die Ist-Architektur auf Einhaltung der Vereinbarungen zu überprüfen.

Der beschriebene Ansatz wurde für das JCommSy angewandt. Dabei ist gezeigt worden, dass Aspekte der Soll-Architektur des JCommSys identifiziert werden können, für die eine Formulierung als Architekturvereinbarungen möglich und sinnvoll ist. Die formulierten Vereinbarungen betreffen die Schichtenarchitektur, die vertikalen Unterteilung und einzelne Methoden des JCommSys. Dabei kann allerdings nicht gewährleistet werden, dass der erarbeitete Katalog vollständig ist.

Durch die Prüfung der Architekturvereinbarungen konnten Verletzungen der Soll-Architektur aufgedeckt und explizit gemacht werden. Die Betrachtung des `migration`-Packages hat deutlich gemacht, dass auch in einem so kleinen System wie dem JCommSy ein Bedarf besteht, Einteilungen des Systems durchzusetzen. Weiter ist davon auszugehen, das die Bedeutung einer klaren Gliederung des Systems mit dem Fortschreiten der Migration des CommSys zunimmt.

Als Einschränkung bei der Zusammenstellung des Kataloges hat sich der Entwicklungsstand des JCommSys erwiesen. Da bisher nur eine Rubrik migriert wurde, ist die Soll-Architektur in weiten Teilen noch unbestimmt. Sie macht in vielen Bereichen keine oder nur sehr grobe Vorgaben. Diese Unbestimmtheit der Soll-Architektur hat die Grundlage für die Formulierung von Architekturvereinbarungen verkleinert und somit deren Anzahl verringert.

Bei der Erstellung der Arbeit konnte der Effekt einer Schärfung der Soll-Architektur im Rahmen der Identifizierung und Formulierung von Architekturvereinbarungen deutlich beobachtet werden. Die „Suche nach Vereinbarungen“ führt zu einer veränderten Perspektive bei der der Betrachtung der Soll-Architektur. Eine wirkliche Weiterentwicklung der Soll-Architektur war im Rahmen dieser Arbeit zwar nicht möglich, bei den beteiligten Entwicklern hat sich das Verständnis der Soll-Architektur und das Bewusstsein für die unbestimmten Bereiche aber spürbar verbessert.

Die Prüfung der Architekturvereinbarungen muss noch weiterentwickelt werden. Für die Prüfung wurden sehr unterschiedliche Verfahren angewandt, die von der Beschreibung eines Architekturmodells bis zur Formulierung von SQL-Queries reichten. Zwischen Formulierung und Prüfung entsteht ein Bruch, der zu überwinden ist, um den Ansatz praxistauglich zu machen. Bisher fehlt ein Verfahren, dass leichtgewichtig ist und die Mächtigkeit des Sotographen mit dem direkten Feedback von SonarJ verbindet.

Schließlich ist festzuhalten, dass weitere Erfahrungen bei der Anwendung des Ansatzes gesammelt werden müssen. Die empirische Basis ist noch zu gering, um die Frage zu klären, ob die Prüfung von projektspezifischen Architekturvereinbarungen tatsächlich im Entwicklungsalltag eine Verbesserung der inneren Qualität einer Software bewirkt.

A Beispiel für eine Query

In diesem Anhang werden Auszüge aus dem Quelltext einer Query aufgelistet. Diese kann im Sotographen dazu verwendet werden die Einhaltung einer Vereinbarung zu Prüfen.

Die aufgelistete Query prüft die Vereinbarung 3.2 aus dem Abschnitt 7.4, die hier noch einmal wiedergegeben wird:

Vereinbarung 3.2: Klassen der Darstellungsschicht, deren Name auf `Bean` endet, werden als Beans bezeichnet. Der Aufruf von Set-Methoden an Beans ist nur innerhalb der Steuerungsschicht erlaubt.

Aus der Quelltext sind zwei Teile dargestellt. Der erste Teil dient vorbereitend der Erkennung der relevanten Klassen.

```
DROP TABLE IF EXISTS PresentationClasses;
DROP TABLE IF EXISTS BeanClasses;

CREATE TABLE PresentationClasses
SELECT DISTINCT
  bean.*
FROM
  Classes= bean,
  SubSystemPackages= beans_package_to_subsystem,
  SetSubsystems= presentation_subsys
WHERE
  bean.packageid = beans_package_to_subsystem.packageid AND
  beans_package_to_subsystem.subsystemid = presentation_subsys.subsystemid AND
  presentation_subsys.name = 'presentation';

CREATE TABLE BeanClasses
SELECT DISTINCT
  bean.*
FROM
  PresentationClasses= bean
WHERE
  bean.name LIKE 'Bean';
```

Im zweiten Teil führt zu einer Auflistung von Methodenaufrufen, die gegen die Vereinbarung verstoßen.

```
CREATE TABLE BeanClasses
SELECT DISTINCT
  bean.*
FROM
  PresentationClasses= bean
WHERE
  bean.name LIKE '\%Bean';

SELECT
  caller.name, called_bean.name, setter_method.name
FROM
  BeanClasses= called_bean,
  Methods= setter_method,
  MethodReferencesFlat= method_call,
  PresentationClasses= caller
```

```
WHERE
  setter_method.classid = called_bean.symbolid AND
  setter_method.name LIKE 'set\%' AND
  method_call.refedSymbolId = setter_method.symbolId AND
  method_call.refingClassId = caller.symbolId AND
  NOT (called_bean.symbolid = caller.symbolId);
```

Literatur

[BCK05] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice*. Addison-Wesley, 2005

[BKL04] Bischofberger, Walter ; Kühl, Jan ; Löffler, Silvio: Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In: Oquendo, Flávio (Hrsg.) ; Warboys, Brian (Hrsg.) ; Morrison, Ronald (Hrsg.): *EWSA* Bd. 3047, Springer, 2004, S. 1–9

[Com] *CommSy*. `http://www.commsy.de`

[FA98] Fiutem, R. ; Antoniol, G.: Identifying Design-Code Inconsistencies in Object-Oriented Software: a Case Study. In: *International Conference on Software Maintenance (ICSM)* (1998), S. 94–102

[Fow99] Fowler, Martin: *Refactorings*. Addison-Wesley, 1999

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[JDea] *JDepend*. `http://www.clarkware.com/software/JDepend.html`

[JDeb] *JDepend4Eclipse*. `http://andrei.gmxhome.de/jdepend4eclipse`

[JSR] *Java Specification Request 223: Scripting for the JavaTM Platform*. `http://jcp.org/en/jsr/detail?id=223`

[Kar05] Karstens, Bettina: *Regeln der WAM-Modellarchitektur*. Diplomarbeit, 2005. – Universtät Hamburg

[MDR93] Meyers, Scott ; Duby, Carolyn K. ; Reiss, Steven P.: Constraining the Structure and Style of Object-Oriented Programs. Providence, RI, USA : Brown University, 1993. – Forschungsbericht

[MNS01] Murphy, Gail C. ; Notkin, David ; Sullivan, Kevin J.: Software Refexion Models: Bridging the Gap between Design and Implementation. In: *IEEE Transactions on Software Engineering* 27 (2001), April, Nr. 4, S. 364–380

[RL04] Roock, Stefan ; Lippert, Martin: *Refactorings in großen Softwareprojekten*. dpunkt.verlag, 2004

[Son] *SonarJ*. `http://www.hello2morrow.de`

[Sot] *Sotograph*. `http://www.sotograph.com`

[SSC96] Sefika, Mohlalefi ; Sane, Aamod ; Campbell, Roy H.: Monitoring Compliance of a Software System with its High-Level Design Models. In: *Proceedings of the International Conference of Software Engineering*, 1996, S. 387–396

[TCL02] Tvedt, Roseanne T. ; Costa, Patricia ; Lindvall, Mikael: Does the Code Match the Design? A Process for Architecture Evaluation. In: *Proceedings of the International Conference of Software Maintainance (ICSM)*, 2002, S. 393–401

[Zül98] Züllighoven, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. dpunkt.verlag, 1998