Diplomarbeit

Automatisierte Prüfung von Architekturregeln zur Entwicklungszeit

Januar 2008

Universität Hamburg
Fakultät MIN, Department Informatik
Zentrum für Architektur und Gestaltung von IT-Systemen

Arne Scharping

Halstenbeker Straße 89a
22457 Hamburg

Email: arne.scharping@informatik.uni-hamburg.de
Matrikelnr.: 5422314

Erstbetreuung: Prof. Dr.-Ing. Heinz Züllighoven
Zweitbetreuung: Prof. Dr. Claus Lewerentz

Betreuung

Prof. Dr.-Ing. Heinz Züllighoven (Erstbetreuer)

Prof. Dr. Claus Lewerentz (Zweitbetreuer)

Prof. Dr.-Ing. Heinz Züllighoven

Zentrum für Architektur und Gestaltung von IT-Systemen (AGIS)

Fakultät für Mathematik, Informatik und Naturwissenschaften (MIN), Department Informatik

Universität Hamburg

Prof. Dr. Claus Lewerentz

Lehrstuhl für Software-Systemtechnik

Institut für Informatik

Brandenburgische Technische Universität Cottbus

Danksagung

Mein besonderer Dank gilt Carola Lilienthal und Petra Becker-Pechau, für die zahlreichen Ideen und Verbesserungsvorschläge sowie für die Bereitschaft, so häufig und gelegentlich auch kurzfristig Entwürfe zu lesen.

Ich danke ebenfalls Prof. Dr. Claus Lewerentz und Prof. Dr.-Ing. Heinz Züllighoven für die Betreuung der Arbeit und die konstruktive Kritik, die ich von ihnen erhalten habe.

Dankbar bin ich auch den Studenten und Mitarbeitern am Arbeitsbereich SWT, die mich mit ihrem Interesse für meine Arbeit bestärkt und wertvolle Anregungen gegeben haben.

Nicht zuletzt möchte ich mich bei Gesche bedanken, die mich während der Zeit unterstützt hat sowie und Wiebke und Johannes, die meine Arbeit Korrektur gelesen haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1 Einleitung

Zentrale Eigenschaften der internen Qualität von Software, wie die Wartbarkeit, Erweiterbarkeit und Testbarkeit, werden in hohem Maße durch ihre Architektur bestimmt. (Bass et al. 2003, S. 30) Um diese Qualitätseigenschaften zu erreichen, verwendet man komplexe Architekturen, die eine Vielzahl von Regeln für die inneren Strukturen der Software beinhalten. Diese Regeln können allgemeine softwaretechnische Regeln, Regeln eines verwendeten Architekturstils oder projektspezifisch vereinbarte Regeln sein.

Während der Implementierung besteht die Gefahr, dass die Regeln im Code und damit in der Ist-Architektur verletzt werden. (Becker-Pechau et al. 2006; Bischofberger et al. 2004; Karstens 2005; Scharping 2005) Die Gründe für solche Architekturverletzungen sind vielschichtig:

- **Kommunikation, Veränderung des Teams:**

  Je größer Team und System, desto schwieriger ist es, die Soll-Architektur zu vermitteln (Bischofberger et al. 2004) und gerade neuen Entwicklern ist die Soll-Architektur oft nicht präsent, sodass sie unbewusst dagegen verstoßen. (Becker-Pechau et al. 2006)

  Auf der anderen Seite geht das Wissen über die Architektur eines Systems häufig verloren, wenn einzelne Personen das Team verlassen. Mit ihnen verschwindet bei mangelnder Kommunikation und Dokumentation das Verständnis für die bestehenden Lösungen.

- **Konzentration auf spezielle Probleme:**

  Auch wenn Entwicklern die Soll-Architektur vertraut ist, konzentrieren sie sich meist darauf spezielle Probleme zu lösen. Die Auswirkungen auf die Architektur werden dabei vernachlässigt. (Bischofberger et al. 2004)

- **Zeitdruck, Provisorien:**

  Neben diesen unbewussten Verletzungen gibt es auch solche, die als Provisorien bewusst in Kauf genommen werden müssen. Dies ist der Fall, wenn Zeitdruck dazu führt, dass weder eine Lösung im Rahmen der Soll-Architektur entwickelt werden kann, noch eine Anpassung der Soll-Architektur möglich ist. Dabei besteht die Gefahr, dass die Provisorien später vergessen werden, da nichts die Entwickler an sie erinnert, solange das Programm läuft. (Becker-Pechau et al. 2006; Bischofberger et al. 2004)

Diese Verletzungen der Soll-Architektur führen dazu, dass Ist- und Soll-Architektur auseinanderdriften und es schwieriger wird, das System zu verstehen, zu warten und zu erweitern. Es sollte daher regelmäßig geprüft werden, ob die Architekturregeln im Code eingehalten werden. Dies kann zwar durch manuelle Code-Reviews geschehen, bei dem Umfang heutiger Systeme bietet sich allerdings an, die Architekturüberprüfung durch den Einsatz von Werkzeugen zu automatisieren. (Becker-Pechau et al. 2006)

In dieser Diplomarbeit werden Konzepte für eine automatisierte Prüfung von Architekturregeln erarbeitet und prototypisch umgesetzt. Die Arbeit baut dabei auf den Ergebnissen auf, die in der Diplomarbeit von Karstens (Karstens 2005) und der Baccalaureatsarbeit von Scharping (Scharping 2005) vorgestellt wurden.

Karstens hat sich mit der Prüfung von Systemen beschäftigt, die nach dem *Werkzeug&Material-Ansatz* (kurz WAM-Ansatz) erstellt wurden. Der WAM-Ansatz (siehe Züllighoven 1998; Züllighoven 2005) wurde am Arbeitsbereich Softwaretechnik der Universität Hamburg entwickelt. Er gibt eine Anleitung für die Gestaltung von Softwaresystemen

und beschreibt eine iterative Vorgehensweise mit dem Ziel einer anwendungsorientierten Softwareentwicklung. Für diese Arbeit ist von besonderer Bedeutung, dass diese Vorgehensweise verknüpft ist mit allgemeinen Prinzipien für die Gestaltung von Softwarearchitekturen. Darüber hinaus beinhaltet der Ansatz eine genaue Vorstellung von den grundlegenden Architekturelementen und deren Verknüpfung. Mit diesen Elementen und den Regeln, die für die Elemente und ihre Verknüpfung gelten, beschreibt der WAM-Ansatz einen Architekturstil. Da der WAM-Ansatz auch die softwaretechnische Realisierung anleitet, ist er Teil einer Modellarchitektur. Karstens hat die Regeln der WAM-Modellarchitektur in einem Katalog zusammengestellt und formalisiert. Weiter hat sie ein Verfahren entwickelt, um mithilfe des Sotographen zu prüfen, ob Systeme, die nach dem WAM-Ansatz konstruiert wurden, die Architekturregeln einhalten. (Karstens 2005)

In einer Baccalaureatsarbeit (Scharping 2005), die dieser Arbeit vorangegangen ist, wurde der entwickelte Ansatz von Karstens auf projektspezifische Regeln ausgeweitet. Als konkretes Beispiel diente dabei der Prototyp des JCommSys, einer in Java geschriebene Webanwendung zur Unterstützung von Projektarbeit in Forschung und Lehre. Im Rahmen der Baccalaureatsarbeit wurde das JCommSy mithilfe unterschiedlicher Werkzeuge auf die Einhaltung der identifizierten Regeln überprüft.

Diese Diplomarbeit baut auf den beiden vorgestellten Arbeiten auf und entwickelt deren Konzepte zur Beschreibung und Prüfung von Architekturen weiter. Auf der konzeptionellen Ebene wurden dabei die folgenden Frage- bzw. Problemstellungen betrachtet:

- **Wie können Architekturregeln beschrieben werden?**

  Architekturregeln müssen so weit formalisiert werden, dass ein Werkzeug ihre Beschreibung für eine automatisierte Prüfung verwenden kann. Ziel ist eine Beschreibung, die leicht zu erstellen und zu pflegen ist.

- **Wie werden Architekturelemente bestimmt und mit dem Code in Beziehung gesetzt?**

  Architekturregeln beziehen sich auf Architekturelemente. Im Rahmen einer Prüfung der Regeln muss daher bestimmt werden, welche Architekturelemente es gibt und wie diese im Code repräsentiert sind. Dabei ist zu untersuchen, in welchen Fällen Architekturelemente im Code identifiziert werden können und wann eine externe Beschreibung nötig ist. Für den Fall einer externen Beschreibung ist ein Weg zu finden, um die Architekturelemente mit dem Code in Beziehung zu setzen.

  Sowohl bei der Identifikation von Elementen im Code als auch bei der der Abbildung von Code- zu Architekturelementen sollte eine hohe Flexibilität erreicht werden, um den spezifischen Bedürfnissen von Projekten mit unterschiedlichen Namenskonventionen oder Rahmenwerken gerecht werden zu können.

- **Wie ist bei der Prüfung zu verfahren? Wie ist mit Regelverletzungen umzugehen?**

  Für die eigentliche Prüfung der Regeln ist ein Verfahren zu entwickeln, dass eine direkte Rückmeldung zur Einhaltung von Architekturregeln ermöglicht. Mit direkter Rückmeldung ist neben der zeitlichen Nähe zur Änderung des Quellcodes auch eine räumliche Nähe gemeint. Architekturverletzungen sollen also an den verursachenden Stellen im Quelltext dargestellt werden. Schließlich soll bei der Darstellung der Architekturverletzung die Ursache der Verletzung erkennbar sein.

- **Wie ist mit Ausnahmen umzugehen?**

  Es ist anzunehmen, dass nicht in jedem Fall einer Regelverletzung diese auch behoben werden soll. Es kann begründete Ausnahmen geben, bei denen Regelverletzungen aus pragmatischen Gründen zugelassen werden. In solchen Fällen ist ein angemessener Weg für den Umgang mit Ausnahmen zu finden. Ein weiterer Grund für die Betrachtung von Ausnahmen ist, dass meistens nicht alle Regeln exakt beschrieben werden können. Die Behandlung als Ausnahmen kann ein pragmatischer Weg sein, um mit vermeintlichen Regelverletzungen umzugehen, die durch Mängel in der Regelbeschreibung entstehen. (vgl. Becker-Pechau et al. 2006; Karstens 2005)

In dieser Diplomarbeit wurden in einem iterativen Prozess Konzepte entwickelt, um Architekturregeln zu beschreiben und automatisch zu prüfen. Die konzeptionelle Arbeit wurde durch die Entwicklung eines Softwarewerkzeuges ergänzt. Die prototypische Umsetzung der entwickelten Konzepte ermöglicht, die zugrunde liegenden Annahmen zu überprüfen und Erfahrungen im Einsatz zu sammeln. Eine erste Version des Werkzeuges hat zunächst nur einen kleinen Satz von Regeln unterstützt. Basierend auf den mit dem Werkzeug gesammelten Erfahrungen konnten die Konzepte im Laufe der Arbeit verbessert werden. So wurden weiterer Regelarten in einem iterativen Prozess integriert und das Verfahren verfeinert.

Die Architekturen heutiger Softwaresysteme sind sehr vielfältig. Der Rahmen einer Diplomarbeit bietet daher nicht die Möglichkeit, die aufgeworfenen Fragen erschöpfend zu behandeln. Ziel der Diplomarbeit ist daher nicht, eine Lösung für die Prüfung aller denkbaren Architekturregeln zu finden. Vielmehr strebt die Arbeit lediglich die Behandlung von Regeln an, die bereits zusammengetragen und für die entsprechenden Systeme für relevant befunden wurden. Konkret beschränkt sich die Arbeit auf die beiden bereits erwähnten Regelkataloge: Der erste Katalog enthält die Regeln der WAM-Modellarchitektur, der zweite Katalog enthält die Regeln des JCommSys. So werden neben dem JCommSy auch WAM-Systeme als Beispielsysteme verwenden. An diesen wird das Werkzeug zur Regelprüfung erprobt und bewertet. Als Beispielsysteme für die WAM-Modellarchitektur dienen dabei zwei die beiden Systeme EMS und Pausenplaner, die auch Karstens untersucht hat. Die Verwendung dieser Systeme ermöglicht, die Ergebnisse der unterschiedlichen Prüfverfahren zu vergleichen. Dabei kann gezeigt werden, dass mit dem entwickelten Verfahren dieselben Architekturverletzungen gefunden werden. Schließlich dienen das JCommSy und der WAM-Ansatz in dieser Arbeit als Beispiele, um die dargestellten Konzepte zu veranschaulichen.

Sowohl für das JCommSy, als auch für WAM-Systeme sind zyklische Abhängigkeiten zwischen Architekturelementen verboten. Die Erkennung zyklischer Abhängigkeiten ist eigenständiges Thema und weitgehend unabhängig von den Fragestellungen dieser Arbeit. Daher wird die Prüfung von Regeln für zyklische Abhängigkeiten in dieser Arbeit nicht betrachtet.

Die Gliederung dieser Arbeit ist im Folgenden dargestellt:

Das Kapitel 2 dieser Arbeit dient der Einordnung der zentralen Begriffe. Neben dem Architekturbegriff wird verdeutlicht, was hier unter Architekturregeln verstanden wird. Mit den Architekturstilen wird auch die Grundlage für die Beschreibung von Architekturen und Architekturregeln vorgestellt. Weiter stellt das 2. Kapitel frühere Arbeiten, die sich mit dem Thema beschäftigen, vor und erläutert Anknüpfungspunkte an frühere Ansätze. In Kapitel 3 sind die entwickelten Konzepte zur Beschreibung und Prüfung von Architekturen beschrieben. Im 4. Kapitel sind die Details der Beschreibung der Regeln und der Benutzung des Werkzeuges dargestellt. Kapitel 5 gibt einen Einblick in die technische Umsetzung. Die Ergebnisse der Architekturüberprüfung mit dem Werkzeug sind in den Kapiteln 6 und 7 beschrieben. Die Arbeit schließt in Kapitel 8 mit einer Zusammenfassung und Bewertung der erreichten Ergebnisse sowie einem Ausblick auf weitere mögliche Untersuchungen.

Kapitel 2 Architektur und Architekturstile

Die Architektur von Softwaresystemen steht im Zentrum dieser Arbeit. Dieser Abschnitt erläutert das zugrunde liegende Verständnis von Softwarearchitektur und damit verknüpften Begriffen.

2.1 Architektur

Der hier verwendete Architekturbegriff orientiert sich an Bass, Clements und Kazman:

> *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. (Bass et al. 2003)*

Bei der Betrachtung von Architektur kommen die unterschiedlichen Anliegen der Projektbeteiligten zum Tragen. Aus ihren unterschiedlichen Standpunkten ergeben sich meist unterschiedliche Sichten eines Systems (Kruchten 1995). Eine Sicht repräsentiert eine Menge von Architekturelementen und die Beziehungen zwischen diesen Elementen.

Reussner und Hasselbring haben unterschiedliche Einteilungen in Sichten zusammengetragen. Sie unterscheiden dabei zwischen einer statischen, einer dynamischen und einer Verteilungs-Sicht. (Reussner und Hasselbring 2006)

In dieser Arbeit steht die statische Sicht der Architektur im Zentrum. Diese beschreibt die laufzeitunabhängige Zerlegung eines Systems in Elemente und die Abhängigkeiten zwischen diesen. In objektorientierten Systemen handelt es sich dabei also um Klassen und Architekturelemente, die von Mengen von Klassen repräsentiert werden (z. B. Subsysteme) sowie den Beziehungen untereinander. Zur Laufzeit existierende Objektgeflechte werden also nicht betrachtet.

2.1.1 Soll- und Ist-Architektur

Bei der Betrachtung der Architektur von Software ist zwischen der Soll- und der Ist-Architektur zu unterscheiden (vgl. Abbildung 1).

Die Soll-Architektur gibt vor, welche Strukturen innerhalb des Softwaresystems gewünscht sind. Sie wird von Software-Architekten oder Entwicklern erarbeitet und ist das Produkt einer Reihe von Entwurfsentscheidungen. Dem gegenüber ergibt sich die Ist-Architektur aus den tatsächlich im Code beschriebenen Strukturen. Die Soll-Architektur kann als eine Vorlage verstanden werden, die der Implementierung und damit der Ist-Architektur eine Menge von Regel, Vereinbarungen und Richtlinien vorgibt. Dies setzt aber nicht voraus, dass es eine vollständige Beschreibung der Soll-Architektur gibt. Häufig sind Teile der Soll-Architektur nur im Gedächtnis einzelner Entwickler festgehalten.

Um die Ist-Architektur eines Systems bestimmen zu können, reicht es nicht aus, den Quellcode des Systems zu betrachten. Da heutige Programmiersprachen die Architektur nur sehr eingeschränkt beschreiben können (vgl. Aldrich et al. 2002), ist über den Code hinaus eine Abbildung der Code- auf die Architekturelemente nötig. Diese Abbildung bildet zusammen mit dem Code eine Beschreibung der Ist-Architektur.

Dem klassischen Wasserfall-Modell mit einem Upfront Design liegt die Annahme zugrunde, man könne das softwaretechnische Design (und damit die Soll-Architektur) am Beginn eines Projektes vollständig und abschließend als Bestandteil einer Spezifikation erstellen. In einer späteren Phase implementiert man diese und überträgt die Soll-Architektur auf eine Ist-Architektur. Die zyklisch-inkrementellen Vorgehensmodelle verabschieden sich von dieser Vorstellung. Die Entwicklungen von Soll- und Ist-Architektur sind (wenn sie betrieben

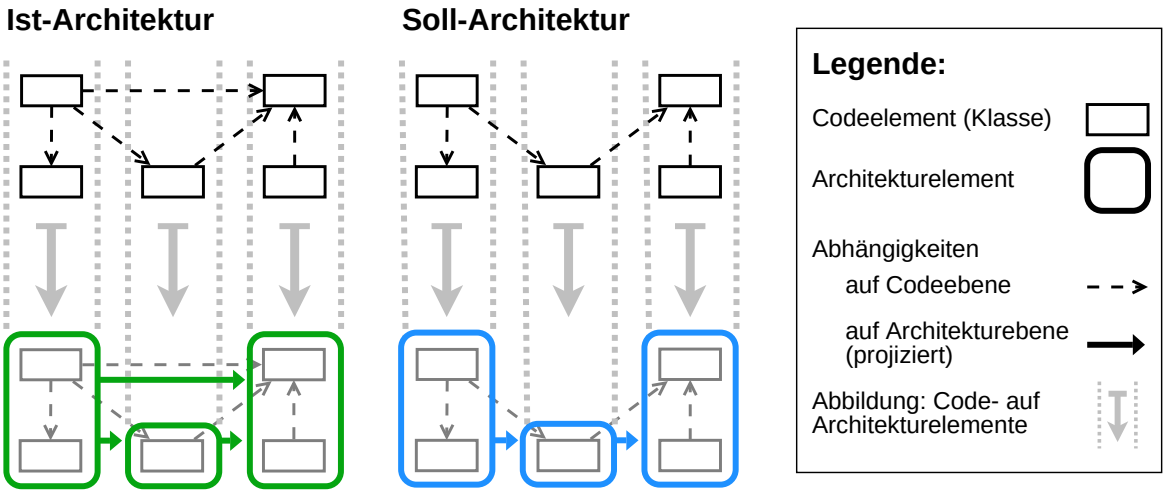

Abbildung 1: Ist- und Soll-Architektur

werden) eng miteinander verzahnt. Um neuen Anforderungen gerecht zu werden, passt man die Soll-Architektur ständig an und überträgt diese Anpassungen in Form von Refactorings und Erweiterungen auf den Code und damit auch auf die Ist-Architektur.

2.2 Architekturüberprüfung

Die in der Einleitung als Problem skizzierten Architekturverletzungen können als Abweichungen der Ist-Architektur von den Vorgaben der Soll-Architektur aufgefasst werden. Es gibt bereits verschiedene Ansätze der Architekturüberprüfung, die darauf abzielen, diese Abweichungen zu entdecken und einem Zerfall der Architektur vorzubeugen. Dieser Abschnitt stellt einige wichtige Ansätze kurz vor. Einen Überblick und Kriterien für die Bewertung geben auch (Becker-Pechau und Bennicke 2007) und (Knodel und Popescu 2007).

2.2.1 Pattern Lint

Bereits 1996 haben Sefika, Sane und Campbell ein Werkzeug vorgestellt, mit dem die Umsetzung einer Soll-Architektur in Teilen überprüft werden kann (Sefika et al. 1996). Das Werkzeug mit dem Namen *Pattern Lint* unterstützt einen hybriden Ansatz, der eine statische Codeanalyse mit einer interaktiven Betrachtung zur Laufzeit des untersuchten Systems kombiniert. Bei der Prüfung der Soll-Architektur unterscheiden die Autoren drei Abstraktionsebenen:

Heuristische Regeln (heuristic rules) repräsentieren abstrakte und subjektive Entwurfsrichtlinien, wie die Richtlinie Elemente mit hoher Kohäsion und geringer Kopplung zu erstellen oder das Geheimnisprinzip umzusetzen. Eine Prüfung solcher Regeln erfolgt interaktiv und basiert auf den Fähigkeiten von Pattern Lint Architekturen zu visualisieren.

Die konkreten Regeln (concrete rules oder low level rules) beschreiben konkrete Anforderungen, die sich direkt auf die Elemente des Codes beziehen. Eine solche Regel für das vorgestellte Beispielsystem ist, dass keine Vererbung über Subsystemgrenzen hinaus erlaubt ist. Eine Prüfung erfolgt mit Prolog-Klauseln, die Regelverletzungen auf einem Modell des Systems finden.

Die dritte Ebene stellen die Architekturregeln (architectural rules) dar, die sich auf die richtige Umsetzung von Entwurfsmustern und Architekturstilen beziehen. Um die Ausprägungen von Stilen und Mustern in einem System finden und dann prüfen zu können, werden ebenfalls Prolog-Klauseln eingesetzt.

Ein Problem besteht darin, Ausprägungen von Mustern und Stilen mit ihrer möglichen Vielfalt zuverlässig zu erkennen. Die Prologklauseln unterstützen zwar bestimmte Varianten zu berücksichtigen, dennoch werden damit nicht alle richtig erkannt.

2.2.2 Software Reflexion Models

Murphy und seine Kollegen beschreiben einen weiteren Ansatz, mit dem unter anderem verhindert werden soll, dass Architekturentwurf und Code auseinanderdriften (Murphy et al. 1995; Murphy et al. 2001). Er basiert darauf entstandene Abweichungen durch fehlende oder ungeplante Abhängigkeiten zwischen Architekturelementen zu erkennen und in sogenannten *Software Reflexion Models* zu visualisieren.

Bei dem Verfahren müssen die Entwickler zunächst die geplanten Module und die Abhängigkeiten zwischen ihnen in einem *High Level Model* beschreiben. Mithilfe eines Werkzeuges wird der Quelltext analysiert und die darin beschriebenen Strukturen im *Source Model* erfasst. High Level Model und Source Model befinden sich auf verschiedenen Abstraktionsebenen. Um sie miteinander in Beziehung setzen zu können, ist eine Abbildung (Mapping) der Elemente des Source Models auf die Elemente des High Level Models erforderlich. Mit dieser Abbildung können dann auch die Abhängigkeiten des Source Model auf Abhängigkeiten des High Level Model übertragen werden. Murphy et al. bezeichnen dies als *Lifting*.

Das Software Reflexion Model beschreibt Übereinstimmungen und Abweichungen dieser vorgefundenen Abhängigkeiten mit den geplanten Abhängigkeiten. Im Reflexion Model wird unterschieden zwischen vorhandenen Abhängigkeiten, die jedoch nicht geplant sind (divergences), fehlenden Abhängigkeiten (absences) und solchen, die der Planung entsprechen (convergences).

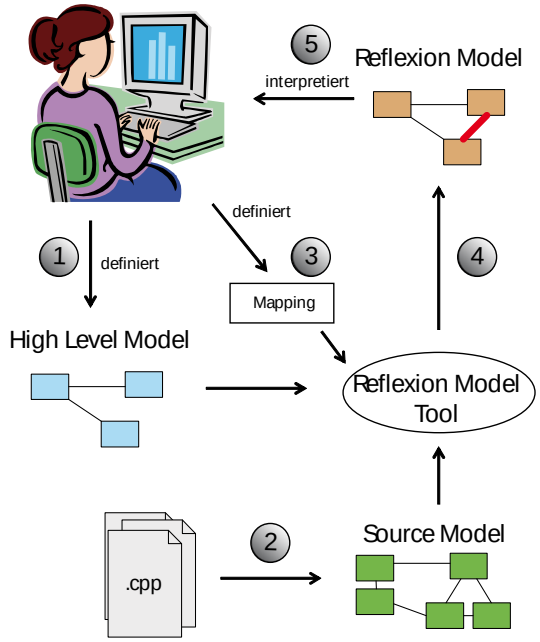

Abbildung 2: Software Reflexion Model (nach Murphy et al. 2001)

Das von Murphy und seinen Kollegen vorgestellte Mapping basiert auf der Grundlage von regulären Ausdrücken und muss manuell beschrieben werden. Christl, Koschke und Storey haben das manuelle Mapping weiterentwickelt und ein semi-automatisches Mapping vorgestellt, bei dem der Aufwand durch ein automatisches Clustering ähnlicher Dateien verringert wird. (Christl et al. 2005)

In der ursprünglichen Form unterstützt der Ansatz keine hierarchischen Modelle. Da dies für größere Systeme nicht angemessen ist, haben Koschke und Simon den Ansatz erweitert (Koschke und Simon 2003). In ihren Modellen können Module weitere Submodule haben. Die Beschreibung der erlaubten Abhängigkeiten zwischen Modulen wird dadurch vereinfacht, dass auch die Typen der Abhängigkeiten als hierarchisch modelliert werden und so die Abhängigkeiten auf unterschiedlichen Abstraktionsniveaus betrachtet werden können.

2.2.3 Metrikbasierte Architekturüberprüfung

Tvedt, Costa und Lindvall haben ein Verfahren vorgestellt, mit dem sie prüfen, ob die tatsächliche Architektur der geplanten Architektur entspricht (Tvedt et al. 2002). Das Verfahren basiert auf einer Metrik, die für jedes Modul die Anzahl der Module bestimmt, zu denen eine

Abhängigkeit besteht. Die Anzahl der geplanten Abhängigkeiten (also die erwarteten Werte) müssen manuell in einer Tabelle erfasst und den gemessenen Werten gegenübergestellt werden. Dieser Schritt ist nicht automatisiert, Abweichungen müssen also von Hand identifiziert werden. Bei der Beschreibung des Verfahrens bleibt offen, wie die Elemente des Codes mit den Modulen in Beziehung gesetzt werden.

2.2.4 ArchJava

Aldrich, Chambers und Notkin sehen als Kern des Problems, die fehlende Möglichkeit Architekturen in gängigen Programmiersprachen explizit zu beschreiben. Sie haben mit ArchJava, eine Spracherweiterung für Java vorgestellt, um diesem Problem entgegenzutreten (Aldrich et al. 2002).

Objekte werden als Komponenten (components) verstanden, die auf einer strukturierten Weise miteinander kommunizieren. Dazu wird für jede Komponente über die sogenannten Ports beschrieben, welche Dienstleistungen sie anbietet und welche sie benötigt. Die Verknüpfung zweier Komponenten (connection) wird ebenfalls explizit beschrieben. Durch die zusätzlichen Sprachbestandteile werden weitere Teile der Architektur im Code selbst beschrieben. Das erweiterte Typsystem sorgt dafür, dass diese konsistent sind, also dass Komponenten beispielsweise ausschließlich über die definierten Ports kommunizieren.

Im Gegensatz zu dieser Arbeit ist ArchJava durch eine dynamische Sicht auf die Architektur geprägt. Die Architekturelemente (Komponenten) existieren zur Laufzeit, sie haben also einen Lebenszyklus und auch die Verknüpfungen zwischen den Elementen können zur Laufzeit erstellt und verändert werden. ArchJava unterstützt zwar die Komposition von Komponenten aus Teilkomponenten, die Ebene der Objekte wird dabei aber nicht verlassen: Für die Modellierung von Subsystemen bietet ArchJava keine besondere Unterstützung.

2.2.5 Eclipse

Zum Teil kann man Subsysteme und deren Abhängigkeiten voneinander über die Projektstruktur von Entwicklungsumgebungen abbilden und durchsetzen. So kann man in Eclipse[1] Projekte verwenden, um ein System in Subsysteme aufzuteilen. Die Abhängigkeiten zwischen diesen Subsystemen kann man über *component access rules* beschreiben, sodass sie mithilfe des Compilers von Eclipse durchgesetzt werden.

Einen Schritt weiter gehen die Entwickler von Eclipse selbst: Eclipse basiert auf einer Plugin-Architektur auf der Grundlage des OSGi-Rahmenwerkes Equinox[2]. Im Gegensatz zu den Projekten sind die Plugins auch zur Laufzeit noch als Architekturelemente vorhanden und bieten die Möglichkeit, eingehende und ausgehende Abhängigkeiten differenziert zu beschreiben und durchzusetzen. Knodel und Popescu haben diesen Ansatz unter anderem mit den Reflexion Models verglichen (Knodel und Popescu 2007).

2.2.6 SonarJ

SonarJ von der Firma hello2morrow[3] ist ein kommerzielles Werkzeug zur Architekturüberprüfung. Das verwendete Metamodell für Architekturen erlaubt, eine (horizontale) Schichtung des Systems und eine vertikale Unterteilung (in *vertikale Schnitte*) zu beschreiben. Diese Einteilungen liegen quer zueinander und erzeugen in der Kombination die Subsysteme. Mit SonarJ können Abhängigkeiten zwischen Schichten, zwischen vertikalen Schnitten und

[1] www.eclipse.org

[2] www.eclipse.org/equinox/

[3] www.hello2morrow.com

zwischen Subsystemen modelliert und überprüft werden. Die Architektur kann man direkt in einer XML-Datei oder in einem grafischen Editor beschreiben. SonarJ kann eigenständig oder in Eclipse integriert genutzt werden. Bei der zweiten Variante liefert SonarJ eine direkte Rückmeldung zu Architekturverletzungen, wie sie auch in dieser Arbeit angestrebt wird.

2.2.7 Sotograph und Sotoarc

Der *Sotograph* wurde von der Firma Software Tomography[4] entwickelt. Er analysiert den Bytecode und erstellt eine Repräsentation der vorgefundenen Strukturen in einer relationalen Datenbank. Diese Datenbank ist die Grundlage für vielseitige Auswertungen. Der Sotograph bringt dazu eine Vielzahl von Metriken und Anfragen mit und kann die Ergebnisse auf unterschiedliche Weisen visualisieren.

Wie bei SonarJ kann mit dem Sotographen die Übereinstimmung mit einer Architekturbeschreibung geprüft werden. Für diese Beschreibung steht ein Metamodell zur Verfügung, das Schichten und Subsysteme vorsieht. In dem mit dem Sotographen verwandten *Sotoarc* können Schichten und Subsysteme auch beliebig verschachtelt werden. Über die Abbildung von Java-Packages auf die Architekturelemente können im Rahmen dieses Metamodells die Abhängigkeiten zwischen den Elementen analysiert und geprüft werden.

Um die Einschränkungen des Metamodells zu umgehen, können eigene SQL-Anfragen an die Datenbank gestellt werden. Dieser relativ aufwendige Ansatz ist in Abschnitt 2.4.2 genauer beschrieben.

Im Laufe der Erstellung dieser Diplomarbeit hat die Firma Software Tomography ein weiteres Werkzeug namens *Sotoarc Developer* vorgestellt. Dabei handelt es sich um ein Eclipse-Plugin, dass wie das hier entwickelte Werkzeug eine Architekturüberprüfung innerhalb der Entwicklungsumgebung durchführt und so eine direkte Rückmeldung ermöglicht. Sotoarc Developer ist im Vergleich zum Sotographen in seiner Funktionalität eingeschränkt. Es verwendet das Metamodell von Sotoarc und beschränkt sich darauf, die Umsetzung einer damit beschriebenen (Soll-)Architektur zu prüfen.

2.2.8 Software-Cockpit

Das *sd&m Software-Cockpit* hat die Firma sd&m[5] in Zusammenarbeit mit der Brandenburgischen Technischen Universität Cottbus entwickelt. Dabei handelt es sich um eine Integrationsumgebung, in der unterschiedliche Werkzeuge zusammengeführt werden, um Qualitätseigenschaften von Software während der Entwicklung zu überwachen. Wie die anderen Werkzeuge analysiert es den Quellcode und erstellt ein Modell des Systems. Werkzeuge, die beispielsweise die Testabdeckung ermitteln, Metriken berechnen oder Code-Konventionen prüfen, werden über Sensoren eingebunden. Eine Auswertungskomponente verarbeitet diese Daten auf der Grundlage eines Qualitätsmodells und generiert Qualitätsaussagen, die in dem Modell des Systems verankert sind. (Bennicke und Richter 2007)

In dem vom Software-Cockpit erstellten Modell eines Systems sind unterschiedliche Sichten repräsentiert. Neben einer logischen Sicht und einer Ablagesicht können Codeelemente in der Architektursicht zu größeren Elementen wie Subsystemen oder Schichten zusammengefasst werden. Im Gegensatz zu SonarJ und dem Sotographen gibt das Software-Cockpit für das Modell aber kein festes Metamodell vor. Vielmehr kann man dieses auf der Grundlage eines Meta-Metamodells anpassen, beispielsweise um die spezifischen Architekturelementarten des analysierten Systems abzubilden.

---

[4] www.software-tomography.com

[5] www.sd&m.de

Ionescu hat in seiner Diplomarbeit (Ionescu 2007) ein Analysekonzept auf der Grundlage des Software-Cockpit entwickelt, dass sich dessen flexibles Metamodell zunutze macht. Er hat eine Erweiterung des Software-Cockpits erstellt, mit der Regeln von Modellarchitekturen bzw. Architekturstilen (siehe Abschnitt 2.3) geprüft werden können. Dazu muss man ein Architektur-Metamodell mit den Element- und Beziehungsarten des Architekturstils erstellen. Damit das Werkzeug ein Exemplar des Architektur-Metamodells, also das Architekturmodell eines Systems erstellen kann, sind Instanziierungsvorschriften für die definierten Elementarten anzugeben. Diese beschreiben die Erstellung der Elemente im Architekturmodell auf Grundlage der Codeelemente (vgl. 3.2.3). Die Architekturregeln werden in einer eigenen Sprache über prädikatenlogische Ausdrücke formuliert.

In seiner Diplomarbeit hat Ionescu das Verfahren an WAM- und Quasar-Systeme erprobt und dabei die WAM-Regeln verwendet, die Karstens zusammengetragen hat und die auch in dieser Arbeit verwendet werden. Weiter hat Ionescu die gleichen WAM-Systeme untersucht, ein ausführlicher Vergleich der Ergebnisse der Arbeiten ist aufgrund der zeitlichen Überschneidung nicht möglich. Am Ende dieser Arbeit wird jedoch eine kurze Betrachtung der Unterschiede vorgenommen.

2.3 Architekturstile

Soll-Architekturen werden nicht in jedem Projekt von Grund auf neu entwickelt. Vielmehr setzt man meist bekannte Entwurfsmuster (Gamma et al. 1995) oder Architekturstile ein. In dieser Arbeit spielen Architekturstile eine besondere Rolle, da sie den Rahmen für die Formulierung und Prüfung von Architekturregeln bilden.

2.3.1 Terminologie

Die Begriffe *architectural pattern* und *architectural style* werden häufig für dasselbe Konzept verwendet (Bass et al. 2003; Shaw und Garlan 1996). Für diese Arbeit wurde die Bedeutung des Begriffes *Architekturstil* übernommen von Bass, Clements und Kazman:

> *„An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.*
> *A pattern can be thought of as a set of constraints on an architecture." (Bass et al. 2003)*

Verbreitete Architekturstile sind die Client-Server-Architektur und die Schichtenarchitektur. Eine Einteilung in fachliche Schnitte, wie sie beispielsweise in (hello2morrow 2005) beschrieben wird, kann als Architekturstil verstanden werden und auch die WAM-Modellarchitektur definiert einen Architekturstil.

Architekturstile werden verwendet, da sie für eine Problemstellung eine Lösung mit bekannten Qualitätseigenschaften anbieten. Wie Entwurfsmuster (Gamma et al. 1995), bringen sie ein Vokabular mit sich (Shaw und Garlan 1996) und erleichtern so dem Team, ein gemeinsames Bild von der Architektur zu entwickeln und über dieses zu sprechen. Im Vergleich zu Entwurfsmustern nehmen Architekturstile einen größeren Umfang im System ein, der Übergang ist allerdings fließend. Eine genaue Abgrenzung erscheint hier nicht nötig.

2.3.2 Stil und Ausprägung

Bei einer systematischen Betrachtung von Architekturstilen muss man zwischen dem Architekturstil und dessen Ausprägung innerhalb eines Systems unterscheiden. Ein Architekturstil ist im Allgemeinen unabhängig von einem konkreten System. Er gibt meist nur die Elementarten und Regeln für diese vor. In einer Architektur, die einen Stil umsetzt, wird der Stil dann ausgestaltet: Durch explizite Entwurfsentscheidungen oder die Erstellung entsprechender Codeelemente wird festgelegt, welche Arten von Architekturelementen es gibt und wie deren

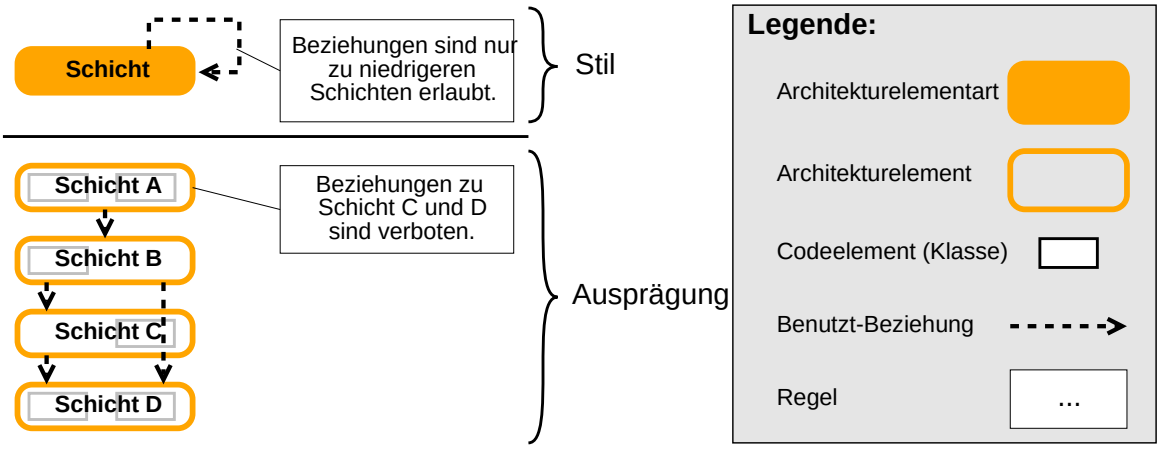

Abbildung 3: Architekturstil und Ausprägung

Beziehung zu den Elementen des Codes ist. Diese projektspezifische Ausgestaltung eines Architekturstils wird hier als Ausprägung des Stils bezeichnet. Die Ausprägung eines Architekturstils ist somit ein Bestandteil der Architektur eines Softwaresystems.

Der Unterschied zwischen Stil und Ausprägung wird im Folgenden am Beispiel der Schichtenarchitektur veranschaulicht. Der Architekturstil gibt als einzige Elementart die Schicht vor. Schichten sind übereinander angeordnet und Abhängigkeiten dürfen nur zu niedrigeren Schichten bestehen. Erst in der Ausprägung einer Schichtenarchitektur innerhalb eines Systems ist festgelegt, welche Schichten es gibt, wie diese angeordnet sind und welche Klassen einer Schicht zuzuordnen sind.

Diese Grenze zwischen Stil und Ausprägung ist jedoch nicht immer so klar zu erkennen: Die *Drei-Schichtenarchitektur*, wie sie unter anderem von Fowler beschrieben wird, kann selbst als Architekturstil aufgefasst werden (Fowler 2002). Dabei gibt der Stil neben der Elementart „Schicht“ auch die Menge der Elemente vor: die Schichten „Präsentation“, „Domäne“ und „Datenquelle“. Die Grenze zwischen Stil und Ausprägung verschwimmt ebenfalls bei Architekturstilen, die projektspezifisch sind, bei denen es also nur eine Ausprägung gibt.

Innerhalb eines Systems kann es mehrere Ausprägungen von Architekturstilen geben. Diese können unabhängig sein und sich auf ganz unterschiedliche Facetten der Architektur beziehen. Architekturstile können aber auch aufeinander Bezug nehmen (vgl. Abbildung 4). Eine Einteilung in fachliche Schnitte kann sich beispielsweise auf eine Schicht der Schichtenarchitektur beschränken oder quer durch alle Schichten verlaufen. Im JCommSy gibt es Ausprägungen von vier Architekturstilen, die teilweise ineinander verschachtelt sind (siehe Kapitel 6).

Da Architekturstile die Arten von Architekturelementen beschreiben, werden sie auch als Metamodelle für die Architektur bezeichnet (Becker-Pechau und Bennicke 2007). Da hier darauf Wert gelegt wird, dass in einer Architektur unterschiedliche Stile umgesetzt werden können, ist ein Stil als Metamodell lediglich für einen Ausschnitt (oder eine Struktur) der Architektur aufzufassen. Um dem fließenden Übergang zwischen Stil und Ausprägung Rechnung zu tragen, wird der Begriff Metamodell hier nicht weiter verwendet.

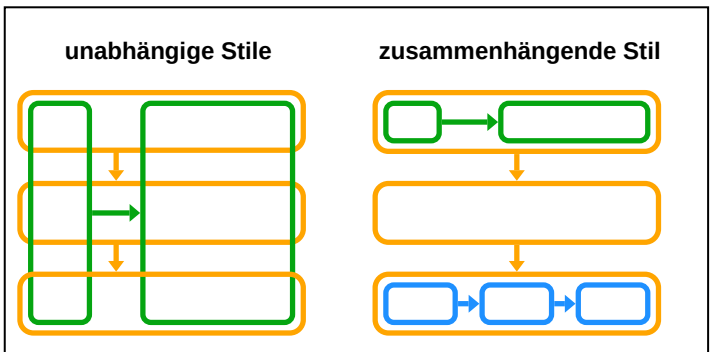

Abbildung 4: Beziehungen zwischen Stilen

2.3.3 Stile als Grundlage für die Architekturüberprüfung

Um zu überprüfen, ob die (Ist-)Architektur eines Systems die Vorgaben der Soll-Architektur einhält, müssen diese formal erfasst werden. Die Architekturstile eines Systems legen fest, welche Arten von Architekturelementen es gibt und liefern die Regeln für die Elemente und deren Beziehungen zueinander. Sie bilden damit den Rahmen, um die zu prüfenden Vorgaben einer Soll-Architektur formalisiert zu beschreiben. Um alle Regeln der vorgestellten Regelkataloge (WAM, JCommSy) in diesem Rahmen abbilden zu können, ist ein relativ weit gefasster Stilbegriff nötig. Für einen Architekturstil wird daher nicht gefordert, dass er ein wiederkehrendes Muster beschreibt. Vielmehr kann ein Architekturstil auch projektspezifische Vereinbarungen repräsentieren. Für das JCommSy wird auch die eher technische Einteilung des Codes in Test- und Produktivcode als Architekturstil aufgefasst.

Die Beschreibung und Prüfung von Architekturstilen bzw. ihrer Regeln wird in diesem und den nächsten Kapiteln am Beispiel der Schichtenarchitektur des JCommSys und des WAM-Architekturstils veranschaulicht.

2.3.4 Beispiel: Schichtenarchitektur

Die Schichtenarchitektur (vgl. Bass et al. 2003; Bäumer 1998; Fowler 2002; Shaw und Garlan 1996) ist einer der am häufigsten beschriebenen Architekturstile. Wie in vielen anderen Systemen wird auch im JCommSy eine Schichtenarchitektur eingesetzt.

Der Architekturstil beschreibt eine Einteilung eines Systems in Schichten, die übereinander angeordnet sind. In objektorientierten Systemen wird meist jede Klasse einer dieser Schichten zugeordnet. Der Architekturstil „Schichtenarchitektur“ sieht also lediglich eine Art von Architekturelementen vor: Alle Architekturelemente sind Schichten.

Der Stil legt nicht fest, welche Schichten es gibt und wie diese angeordnet sind. Dies ist Teil der projektspezifischen Ausprägung. Bei einer Schichtenarchitektur dürfen Schichten nur auf darunterliegende Schichten zugreifen. Bäumer unterscheidet dabei zwischen durchlässigen und undurchlässigen Schichtenarchitekturen. Bei undurchlässigen Schichtenarchitekturen ist der Zugriff auf die unmittelbar darunterliegende Schicht beschränkt (Bäumer 1998). Beziehungen zwischen Klassen derselben Schicht, also Beziehungen innerhalb einer Schicht sind im Allgemeinen erlaubt. Diese Regeln des Stils können in der systemspezifischen Ausprägung ergänzt werden: Im Falle einer strikten Schichtung ist der Zugriff auf unmittelbar darunter liegende Schichten beschränkt. In vielen Ausprägungen von Schichtenarchitekturen gibt es differenzierte Regeln für die Beziehungen zwischen Schichten. Dies gilt auch für die Schichtenarchitektur des JCommSys. Abbildung 5 visualisiert die Schichten des JCommSys und die erlauben Beziehungen zwischen ihnen. Als weitere Regel wird in der Ausprägung des JCommSys gefordert, dass alle Klassen (abgesehen von Testklassen) einer Schicht zugeordnet sein müssen. Eine genauere Beschreibung der Schichtenarchitektur des JCommSys folgt in Kapitel 6.

Durch die Definition von Schnittstellen kann der Zugriff auf Schichten eingeschränkt sein. Die Schnittstelle einer Schicht besteht aus einem Teil der Klassen dieser Schicht, sodass nur diese Klassen von darüberliegenden Schichten verwendet werden dürfen. Alle Klassen, die nicht der Schnittstelle angehören, dürfen nur innerhalb der Schicht verwendet werden. In der untersuchten Version des JCommSys wurden für die Schichtenarchitektur keine Schnittstellen definiert.

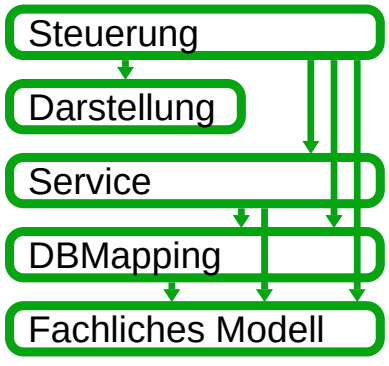

Abbildung 5:
Schichten des JCommSys

2.3.5 Beispiel: WAM-Architekturstil

Als zweites Beispiel wird der Architekturstil betrachtet, der durch die WAM-Modellarchitektur beschrieben wird.

Der WAM-Architekturstil beschreibt verschiedene Elemente, die aus Entwurfsmetaphern abgeleitet werden. Elementarten des Architekturstils sind zum Beispiel Werkzeuge, Materialien, fachliche und technische Dienstleister (Services) sowie Fachwerte. Im WAM-Ansatz beschreiben Konzeptions- und Entwurfsmuster die Gestaltung und das Zusammenspiel der sogenannten WAM-Elemente. Die mit diesen Mustern verknüpften Regeln hat Karstens zusammengetragen (Karstens 2005). Dieser Regelkatalog macht differenzierte Vorgaben dafür, wie beispielsweise die Schnittstelle von Elementen einer bestimmten Elementart zu gestalten ist oder in welchen Beziehungen verschiedenartige Elemente zueinanderstehen.

Anders als bei einer Schichtenarchitektur beziehen sich die Regeln dieses Stils nicht auf einzelne Elemente sondern nur auf Elementarten. Die Regeln sind somit weitgehend unabhängig von der konkreten Ausprägung des Stils. Da der Stil bereits sehr differenzierte Regeln beinhaltet, sind Anreicherungen der Regeln in bestimmten Ausprägungen eher die Ausnahme. In größeren Systemen wird der WAM-Stil allerdings mit anderen Stilen, wie einer fachlichen Einteilung des Systems, kombiniert.

Bei einer Software nach dem WAM-Ansatz können auch kleine Erweiterungen dazu führen, dass neue WAM-Elemente hinzugefügt werden. Bei einer Ausprägung des WAM-Stils ist die Menge der konkreten Elemente also weniger starr, als bei einer Schichtenarchitektur. Für eine Prüfung ist daher wichtig, dass die Menge der Architekturelemente eines Systems nicht durch eine externe Beschreibung der Architektur bestimmt wird. Vielmehr wird sie auf Grundlage der tatsächlich vorhandenen Klassen, die entsprechende Elemente repräsentieren, ermittelt.

2.3.6 Klassen- und Subsystemebene

Bei der Betrachtung der Beispiele fällt auf, dass sich die Architekturelemente der beiden Stile auf unterschiedlichen Ebenen befinden: Während sich die WAM-Elemente auf der Klassenebene befinden, sind Architekturelemente wie Schichten auf einer höheren Ebene einzuordnen. Diese Ebene wird hier als Subsystemebene bezeichnet (vgl. Lilienthal 2008). Die Unterscheidung zwischen Stilen der Klassenebene und der Subsystemebene basiert auf der Beziehung zwischen den Architekturelementen und den Klassen, also den Codeelementen. Die Stile können über ihre Architekturelemente diesen beiden Ebenen zugeordnet werden.

Ein Stil wird hier als Stil auf der Klassenebene bezeichnet, wenn seine Elemente von Klassen repräsentiert werden. Das bedeutet, dass jedes Architekturelement eine Klasse hat, die es in seiner Umgebung repräsentiert. Alle WAM-Elemente, wie zum Beispiel Werkzeuge und Materialien, haben eine Klasse, über die sie identifiziert werden können.

Ein Architekturelement auf der Subsystemebene wird nicht von einer einzelnen Klasse, sondern von einer Menge von Klassen repräsentiert. Damit sind Architekturelemente auf dieser Ebene im Allgemeinen deutlich größer als auf der Klassenebene. Dem Stilbegriff entsprechend ist in dieser Arbeit auch der Subsystembegriff relativ weit gefasst: Ein Subsystem ist eine Menge von Klassen, die in der Ausprägung eines Architekturstils eine Einheit bildet. Hier unterscheidet sich diese Arbeit von anderen, indem Subsysteme nicht den Schichten untergeordnet werden. Vielmehr werden Schichten, fachliche Schnitte oder auch der Testcode als Subsysteme oder als Architekturelement auf Subsystemebene bezeichnet. Wenn ein System mehrere Ausprägungen von Stilen auf Subsystemebene hat, so können auch diese unabhängig voneinander sein, sodass Subsysteme unterschiedlicher Stile „überlappen“ können (siehe Abbildung 4).

Im Allgemeinen sind die Elemente der Subsystemebene mit den Mitteln der Programmiersprache nicht beschreibbar. Die Elemente dieser Ebene werden primär außerhalb des Quellcodes beschrieben und in unterschiedlichem Umfang als Konventionen auf Packagestrukturen, Quellcode-Verzeichnisse oder Projekte der Entwicklungsumgebung abgebildet.

Zusammenfassend stellt sich die Beziehung zwischen Klassen und den Elementen der beiden Ebenen folgendermaßen dar:

- Auf der Klassenebene wird jedes Architekturelement von einer Klasse repräsentiert.
- Auf der Subsystemebene wird dagegen jedes Architekturelement durch eine Menge von Klassen repräsentiert.

Alle bisher betrachteten Architekturstile können einer dieser Ebenen zugeordnet werden.

Eine Unterscheidung der beiden Ebenen ist notwendig, da die Elemente auf den Ebenen grundverschiedene Eigenschaften aufweisen. Auf der Klassenebene werden die Eigenschaften durch die repräsentierende Klasse des Elementes bestimmt. Es handelt sich dabei also im Wesentlichen um die Eigenschaften einer Klasse. So entspricht die Schnittstelle eines WAM-Elementes (Klassenebene) der repräsentierenden Klasse. Dem gegenüber werden die Eigenschaften eines Subsystems durch die repräsentierende Menge von Klassen bestimmt. Die Schnittstelle einer Schicht (Subsystemebene) ist eine Teilmenge der zugehörigen Klassen. In Hinblick auf eine Prüfung von Regeln, die sich auf diese Eigenschaften beziehen, ist die Unterscheidung daher von großer Bedeutung.

2.3.7 Beziehungen zwischen Architekturelementen

Auch bei der Betrachtung der Beziehungen zwischen Architekturelementen ist es hilfreich, zwischen Elementen auf der Klassen- und der Subsystemebene zu unterscheiden (vgl. Lilienthal 2008).

Auf der Klassenebene gibt es zwischen den Elementen Benutzt-, Enthaltenseins- und Vererbungs-Beziehungen. Im Quelltext können Benutzt-Beziehungen verschiedene Formen annehmen: Dazu gehören beispielsweise Aufrufe von Operationen, Zugriffe auf Konstanten und die Verwendung definierter Typen für Variablen oder Typumwandelungen.

Die Enthaltenseins-Beziehung ermöglicht, zusammengesetzte Architekturelemente zu beschreiben. Ein Beispiel für die Konstruktion zusammengesetzter Architekturelemente auf der Klassenebene liefert die Werkzeugkonstruktion des WAM-Architekturstils. Werkzeuge sind meist aus verschiedenen Elementen (Werkzeugkomponenten) zusammengesetzt. Jedes Werkzeug hat eine Werkzeugklasse, die es nach außen repräsentiert. Einfache Werkzeuge (MonoTools) haben eine Oberflächenkomponente (GUI). Umfangreichere Werkzeuge (hier kurz: Tools) besitzen darüber hinaus eine Interaktionskomponente (IAK) und eine Funktionskomponente (FK). Einem Werkzeug sind also verschiedenartige Elemente als Bestandteile zugeordnet. Unabhängig davon kann man zusammengesetzte Werkzeuge realisieren, indem man Sub-Werkzeuge verwendet.

Enthaltenseins-Beziehungen könnten zwar mit dem Sprachmittel der inneren Klassen ausgedrückt werden, meistens realisiert man sie jedoch mit den gleichen Mitteln wie die Benutzt-Beziehungen. Im Quelltext ist damit allerdings schwer zu unterscheiden, ob ein Werkzeug ein anderes benutzt oder ob es sich um ein Sub-Werkzeug handelt.

Auch auf der Subsystemebene gibt es Enthaltenseins-Beziehungen, über die Subsysteme geschachtelt werden können. So werden in Schichtenarchitekturen Schichten teilweise wieder in Sub-Schichten untergliedert. Letztlich kann auch die Teilung einer Schicht in einen internen und einen öffentlichen Bereich (Schnittstelle) als eine solche Untergliederung aufgefasst werden. Im Gegensatz zu einem Sub-Werkzeug oder einer Werkzeugkomponente kann eine

Schicht allerdings nur in einer übergeordneten Schicht enthalten sein. Die Bedeutung derartiger Hierarchiebildungen heben Becker-Pechau und Bennicke hervor (Becker-Pechau und Bennicke 2007).

Benutzt- und Vererbungs-Beziehungen auf der Subsystemebene entstehen durch die Beziehungen zwischen den Klassen, die den Elementen zugeordnet sind. Diese Beziehungen werden auf die Subsysteme *projiziert*. (Lilienthal bezeichnet die Projektion der Beziehungen als *Aggregation*, Murphy et al. als *Lifting* (Lilienthal 2008; Murphy et al. 1995). Eine genauere Darstellung der Projektion erfolgt in Abschnitt 3.2.5.)

2.4 Architekturregeln

Architekturregeln formulieren Anforderungen an die Ausprägungen von Architekturstilen und damit an die Architektur von Systemen. Jede Architekturregel ist einem Architekturstil zugeordnet, der die Architekturelemente beschreibt, auf die sich die Regel bezieht. Somit werden Architekturregeln in dieser Arbeit als Bestandteile von Architekturstilen verstanden.

Karstens hat 94 Regeln des WAM-Architekturstils beschrieben (Karstens 2005). Für das JCommSy konnten 14 Regeln identifiziert werden, die verschiedenen Stilen zuzuordnen sind (Scharping 2005). Während die WAM-Modellarchitektur einen relativ gefestigten und gleichzeitig differenzierten Architekturstil beschreibt, befand sich die Architektur des JCommSys während der Identifikation der Regeln noch in einem frühen Entwicklungsstadium. Dabei konnte beobachtet werden, dass durch diese Form der Beschäftigung mit der Architektur diese immer wieder hinterfragt wurde und so ein Effekt der „Schärfung der (Soll-)Architektur“ eintrat. Durch die Auseinandersetzung mit der Soll-Architektur wurde diese vereinheitlicht und gefestigt.

2.4.1 Regelarten

Bereits in diesen überschaubaren Regelkatalogen ist eine große Vielfalt von Regeln zu finden. Für die Regeln des JCommSys wurde nach der Identifizierung die folgende Kategorisierung vorgenommen: Es gibt projektspezifische Regeln, Regeln von Architekturstilen und allgemeine softwaretechnische Regeln. Allgemeine softwaretechnische Regeln, wie ein Verbot von zyklischen Beziehungen zwischen Klassen, werden in dieser Arbeit nicht weiter betrachtet. Viele Softwarewerkzeuge wie der Sotograph können bereits Systeme auf Einhaltung dieser Regeln überprüfen. Der bei der Kategorisierung verwendete Stilbegriff ist nicht so weit gefasst wie in dieser Arbeit. Hier können Architekturstile auch projektspezifisch sein. Die projektspezifischen Regeln des JCommSys werden daher (projektspezifischen) Stilen zugeordnet.

Karstens hat für die WAM-Regeln eine andere Klassifikation vorgenommen, die hier aufgegriffen und erweitert wird. Karstens unterscheidet zwischen Regeln, die sich auf die Gestaltung einzelner Elemente beziehen (Elementregeln) und Regeln, die Beziehungen zwischen Elementen einschränken (Verbotsregeln) oder fordern (Gebotsregeln).

Elementregeln

Die Elementregeln der betrachteten Regelkataloge beschreiben Anforderungen an die Schnittstelle von Elementen oder an den Aufbau von Elementen. Dabei handelt es sich ausschließlich um Regeln auf der Klassenebene.

Beispiele für Regeln für die Gestaltung der Schnittstelle (WAM):

Materialien haben nicht nur Getter und Setter.

Fachwerte haben keinen öffentlichen Konstruktor.

Beispiel einer Regel für den Aufbau von Elementen (WAM):

Werkzeuge sollten grundsätzlich in Interaktions- und Funktionskomponenten aufgeteilt werden.

Verbotsregeln

Verbotsregeln verbieten alle oder ausgewählte Abhängigkeiten zwischen Elementen oder Elementarten. Die Regelkataloge enthalten Verbotsregeln auf beiden Ebenen.

Auf der Subsystemebene werden zum Beispiel die Beziehungen zwischen den Schichten des JCommSys über Verbotsregeln eingeschränkt. Auch die Beziehungen zwischen WAM-Elementen sind durch Regeln eingeschränkt. Beispiele für diese Verbotsregeln auf Klassenebene sind:

Die Item-Schicht kennt nicht die Serviceschicht.

Materialien kennen keine Werkzeuge.

Fachwerte kennen keine Automaten.

In diesen Beispielen verbieten die Architekturregeln bestimmte Abhängigkeiten zwischen Architekturelementen (Schichten) bzw. Architekturelementarten (Material, Werkzeug, …). Regeln können jedoch auch für ausgewählte Abhängigkeiten gelten. Für beide Ebenen ist hier das gesamte Spektrum von Abhängigkeiten zwischen Klassen denkbar (Deklaration von Variablen eines bestimmten Typs, Aufruf von Methoden, Vererbung usw.).

Auf der Subsystemebene kann das Verbot von Abhängigkeiten auch für bestimmte Klassen gelten bzw. ausgenommen sein. Auf diese Weise können Schnittstellen als Teil einer Regel formuliert werden.

Gebotsregeln

Mit Gebotsregeln können Abhängigkeiten zwischen Architekturelementen gefordert werden. Dieser Regeltyp scheint nur für die Klassenebene relevant zu sein. Dabei wird in den meisten Fällen eine bestimmte Abhängigkeit gefordert.

Gebotsregeln stehen in einem engen Zusammenhang mit der Konstruktion von zusammengesetzten Architekturelementen. Die überwiegende Zahl der identifizierten Gebotsregeln betrifft das Zusammenspiel der verschiedenen Komponenten bei der Werkzeugkonstruktion. Beispiel:

Die IAK kennt ihre FK unter ihrer vollen Schnittstelle.

Zuordnungsregel

Diese bisher vorgestellten Regelarten entsprechen der von Karstens vorgestellten Klassifikation. Darüber hinaus wurde für das JCommSy eine Regel identifiziert, die sich nicht in dieses Schema einordnen lässt. Die Regel 2.1 besagt, dass jede Klasse des Produktivcodes einer Schicht zugeordnet ist. Es gibt also *Zuordnungsregeln*, die die Zuordnung von Codeelementen zu Architekturelementen betreffen.

Bei Betrachtung der untersuchten Architekturstile scheint es eine allgemeingültige Zuordnungsregel zu geben: Eine Klasse kann innerhalb einer Stilausprägung nicht zu unterschiedlichen Architekturelementen gehören. So kann beispielsweise keine Klasse mehreren Schichten angehören. Hier ist einschränkend anzumerken, dass nicht ausgeschlossen werden kann, dass bei der Betrachtung weiterer Architekturstile eine Klasse mehrere Rollen einnimmt. Daher kann die Zuordnung zu mehreren Elementen sinnvoll sein. Dies wird hier jedoch nicht weiter betrachtet.

In der folgenden Aufstellung sind die bisher bekannten Regelarten zusammengefasst:

Elementregeln: Regeln für die Gestaltung der Elemente einer Art

Verbotsregeln: Regeln, die bestimmte Beziehungen zwischen bestimmten Elementarten verbieten.

Gebotsregeln: Regeln, die bestimmte Beziehungen zwischen bestimmten Elementarten fordern.

Zuordnungsregeln: Regeln, die die Zuordnung von Codeelementen zu Architekturelementen betreffen.

Im Rahmen dieser Arbeit kann nicht ausgeschlossen werden, dass es weitere Regelarten gibt. So ist beispielsweise denkbar, dass es Architekturstile gibt, in denen die Anwesenheit bestimmter Architekturelemente reglementiert ist. Da die gegebenen Kataloge solche Regeln jedoch nicht enthalten, werden sie hier nicht weiter betrachtet.

2.4.2 Prüfung von Architekturregeln

Die Formulierung der Vorgaben einer Soll-Architektur bzw. der Vorgaben von Architekturstilen in Regelform geschieht hier mit dem Ziel einer automatisierten Prüfung dieser Regeln. Dieser Ansatz wurde unter anderem von Karstens in ihrer Diplomarbeit angewandt (vgl. Becker-Pechau et al. 2006; Karstens 2005). Karstens hat die Regeln des WAM-Architekturstils (bzw. der WAM-Modellarchitektur) zusammengetragen und mithilfe des Sotographen überprüft. Da die Regeln des WAM-Architekturstils nicht innerhalb des Architektur-Metamodells des Sotographen beschrieben werden können, hat sie die Regeln über direkte Anfragen (Queries) an die Datenbank des Sotographen geprüft. Der Architekturstil ist auf der Klassenebene einzuordnen, daher hat Karstens zunächst Datenbankanfragen formuliert, über die Klassen identifiziert werden, die WAM-Elemente repräsentieren. Die entdeckten WAM-Elemente werden in neuen Tabellen abgelegt. Basierend auf diesen Tabellen konnte sie die WAM-Regeln wiederum als Datenbankanfragen formulieren. Die Ergebnisse dieser Anfragen können als Regelverletzungen interpretiert werden.

Auf diese Weise konnte Karstens 45 von 94 WAM-Regel prüfen. Für die übrigen Regeln lassen sich verschieden Gründe ausmachen, die eine automatisierte Prüfung mit dem Sotographen verhindern:

Nicht alle Regeln können über eine statische Codeanalyse geprüft werden. Davon sind Regeln betroffen, die das Verhalten von Elementen betreffen und solche, die sich nicht nur auf die Struktur, sondern auch auf die Semantik beziehen.

Einige Regeln konnten nicht geprüft werden, da der Sotograph die Analyse von Methodenrümpfen nur begrenzt ermöglicht und es zu aufwendig erschien, externe Bibliotheken einzubeziehen. Schließlich wurden nur strikte Regeln geprüft. Regeln, die eher den Charakter einer Richtlinie mit geringer Verbindlichkeit haben, wurden nicht berücksichtigt.

Karstens hat drei Beispielsysteme, die nach dem WAM-Ansatz konstruiert wurden, auf Einhaltung der Regeln untersucht. Dabei hat sie Regelverletzungen in allen Systemen gefunden. Bei dem Verfahren ist zu beachten, dass die Ergebnisse der Anfragen interpretiert werden müssen. Die Identifikation der WAM-Elemente über die von Karstens formulierten Anfragen setzt die Verwendung des JWAM-Rahmenwerkes voraus, welches gemeinsame Basisklassen für die meisten WAM-Elemente mit sich bringt. Auch mit diesem Rahmenwerk gelingt die Identifikation nicht fehlerfrei und es kommt zu so genannten „unechten Funden“, also vermeintlichen Architekturverletzungen, die unter anderem auf Schwächen der Datenbankanfragen zurückzuführen sind.

Neben unechten Funden ist die Prüfung von Architekturregeln mit weiteren Schwächen verbunden. Die Formulierung von Architekturregeln über SQL-Statements ist relativ aufwendig. Der Sotograph ist nicht in die Entwicklungsumgebung eingebunden, sodass Entwickler keine direkte Rückmeldung über die Einhaltung der Regeln erhalten. Der Umgang mit dem Sotographen erfordert Expertenwissen. In der Regel können nur wenige Entwickler prüfen, ob Architekturregeln in einem System eingehalten werden. Der Sotoarc Developer bietet zwar eine direkte Rückmeldung und ist einfach zu bedienen, er verwendet aber keine Datenbank, die genutzt werden könnte, um Regeln zu prüfen, die (wie die WAM-Regeln) nicht mit dem verwendeten Metamodell beschrieben werden können.

Die Regeln, die für das JCommSy zusammengetragen wurden (Scharping 2005), sind verschiedenen Architekturstilen zuzuordnen. Neben den bereits vorgestellten Regeln der Schichtenarchitektur gibt es solche, die sich auf vertikale Unterteilungen des Systems beziehen oder die Beziehung zwischen Test- und Produktivcode beschreiben. Um die Regeln des JCommSys zu prüfen, wurden in der Baccalaureatsarbeit verschiedene Werkzeuge eingesetzt und ausgewertet. Neben dem Sotographen kamen SonarJ und JDepend zum Einsatz. SonarJ und JDepend sind in der Verwendung einfacher als der Sotograph, konnten aber nur für einen kleinen Teil der identifizierten Architekturregeln verwendet werden. Dabei war es nur selten möglich, die Regeln direkt zu prüfen. Für die meisten Regeln musste ein individueller Weg gefunden werden, um die entsprechende Vorgabe mit dem Metamodell des Werkzeuges auszudrücken.

Kapitel 3 Konzeptionelle Umsetzung

Dieses Kapitel beschreibt, wie die Architekturüberprüfung in dieser Arbeit konzeptionell umgesetzt wird.

3.1 Überblick

Bei der hier entwickelten Architekturüberprüfung stehen die Architekturstile eines Systems mit ihren Elementen und Regeln im Zentrum. Das entwickelte Werkzeug prüft, ob die Stile richtig umgesetzt wurden, also ob die Architekturelemente die Regeln des jeweiligen Stils einhalten. Verstöße gegen diese Regeln meldet es als Architekturverletzungen. Dieser Abschnitt gibt im weiteren Verlauf einen Überblick, wie eine solche Architekturüberprüfung abläuft und welche Voraussetzungen dafür zu schaffen sind. Das Grundprinzip der Prüfung ist ergänzend in Abbildung 6 skizziert.

Das Werkzeug führt eine statische Codeanalyse durch: Es analysiert das System, ohne dabei Programmteile auszuführen. Stattdessen untersucht es den Quelltext des Systems und trägt die darin beschriebenen Strukturen in einem Modell zusammen, das hier als *Systemmodell* bezeichnet wird. In dieses Modell nimmt es Informationen zu allen im Quelltext definierten Typen (Klassen und Interfaces) und den Abhängigkeiten zwischen ihnen auf. Welche Eigenschaften der Typen und Abhängigkeiten in das Systemmodell aufgenommen werden, orientiert sich an der Frage, ob diese benötigt werden, um die Codeelemente in der Architektur einzuordnen oder um die Architekturregeln zu prüfen. Das Systemmodell repräsentiert die im Quelltext beschriebenen Strukturen. Eine genaue Beschreibung des Systemmodells und seiner Bestandteile liefert Kapitel 5.

Zusätzlich zum Quelltext benötigt das Werkzeug eine *Architekturbeschreibung*, um die Architektur eines Systems überprüfen zu können. Diese muss in einer XML-Datei bereitgestellt werden. Sie beschreibt die Architekturstile des Systems und stellt den Bezug zwischen deren Architekturelementen und den Elementen des Codes her.

Mit der Architekturbeschreibung erstellt das Werkzeug auf der Grundlage des Systemmodells ein Modell, das die Ist-Architektur des untersuchten Systems repräsentiert. Dieses *Architekturmodell* ergänzt das Systemmodell um die Ebene der Architekturstile mit deren Architekturelementen. Anhand des Architekturmodells überprüft das Werkzeug, ob die Architekturregeln der einzelnen Stile eingehalten werden.

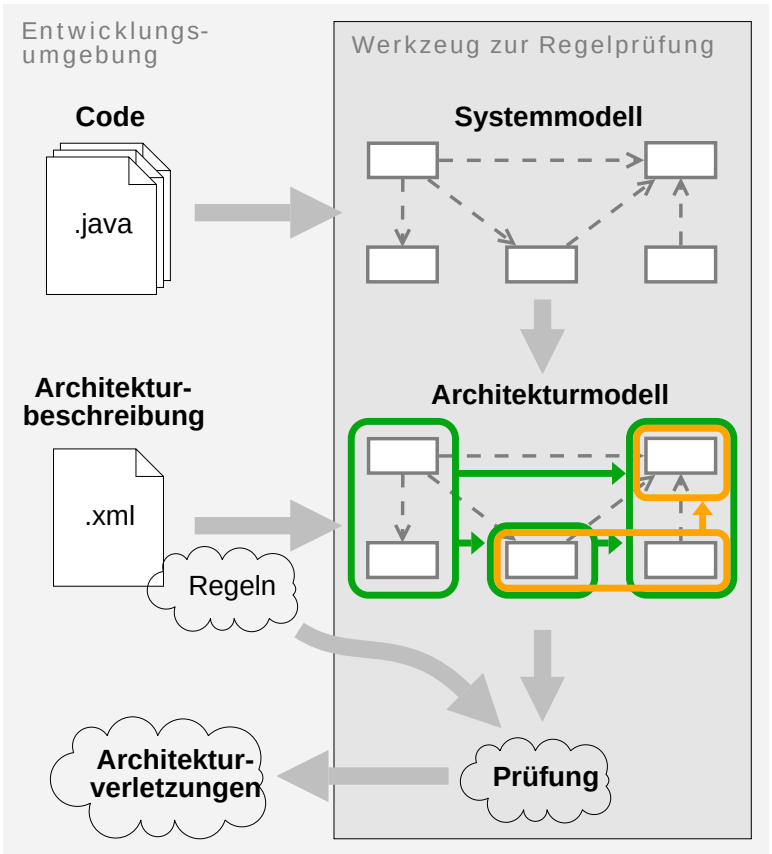

Abbildung 6: Grundprinzip der Architekturüberprüfung

3.2 Erstellung des Architekturmodells

Das Architekturmodell ist nach Architekturstilen gegliedert und enthält deren Elemente und die Abhängigkeiten zwischen ihnen. Es baut auf dem Systemmodell auf und wird somit zum Teil durch den Quellcode bestimmt. Die zweite Grundlage für das Architekturmodell ist die Architekturbeschreibung. Diese ist nötig, da die Ebene der Architekturstile und deren Elemente im Quellcode nicht ausreichend beschrieben sind. Die Architekturbeschreibung ergänzt den Quellcode also um diese (Architektur-)Ebene und stellt die Verbindung zu den Elementen des Codes her. Ausgehend von Quellcode und Architekturbeschreibung erstellt das Werkzeug mit dem Architekturmodell eine Repräsentation der Ist-Architektur.

Die Architekturbeschreibung eines Systems wird von den Entwicklern erstellt und ist so weit formalisiert, dass das entwickelte Werkzeug sie nutzen kann, um das Architekturmodell zu generieren und die Umsetzung der Architekturstile zu überprüfen. Die Beschreibung hat nicht den Anspruch eine vollständige Dokumentation der Architektur zu sein. Bei ihrer Entwicklung stand die Überprüfung der Architekturstile im Vordergrund. Wichtig war dabei, dass die Beschreibung der Architekturstile an projektspezifische Eigenarten und Ausprägungen angepasst werden kann, um die Zahl „unechter Funde“ von Architekturverletzungen gegenüber einer Prüfung mit dem Sotographen (vgl. 2.4.2) zu vermindern und so die Qualität der Ergebnisse zu verbessern. Schließlich sollte die Architekturbeschreibung leicht zu pflegen sein.

In diesem Kapitel sind lediglich die entwickelten Konzepte der Architekturbeschreibung dargestellt. Das detaillierte Format wird in Kapitel 5 erläutert.

3.2.1 Verschmelzung von Stil und Ausprägung

Die Architekturbeschreibung eines Systems beschreibt sowohl die Architekturstile, die in einem System umgesetzt wurden, als auch deren systemspezifische Ausprägung (vgl. 2.3.2). Dabei sind Stil und Ausprägung nicht voneinander getrennt. Durch die Verschmelzung von Stil und Ausprägung wird erreicht, dass die Architekturbeschreibung relativ einfach strukturiert ist und nur eine einzige Beschreibung zu pflegen ist. Eventuelle Schwierigkeiten, die durch den teilweise fließenden Übergang zwischen Stil und Ausprägung entstehen, werden auf diese Weise umgangen. Aus Benutzersicht wird die Beschreibung damit einfacher. Ein Nachteil wird allerdings sichtbar, wenn mehrere Projekte den gleichen Architekturstil umsetzen und man zwar die Beschreibung des Architekturstils, nicht jedoch dessen Ausprägung wiederverwenden möchte. Die Wiederverwendung von Stilbeschreibungen erfolgt in der derzeitigen Umsetzung über Copy&Paste. Im Rahmen dieser Arbeit erscheint diese Einschränkung als pragmatisch und tragbar. (Die Architekturregeln sind aus der Architekturbeschreibung teilweise rausgelöst und können leichter wiederverwendet werden. Mehr dazu im Abschnitt 4.2). Für die weitere Beschreibung ist die Unterscheidung von Stil und Ausprägung meist unbedeutend. Zur Vereinfachung wird daher im Folgenden der Begriff Architekturstil sowohl für den Stil, als auch dessen Ausprägung verwendet.

3.2.2 Ebenen und Elementarten der Stile

Bei den Architekturstilen wird zwischen Stilen auf der Klassenebene und Stilen auf der Subsystemebene unterschieden (vgl. 2.3.6). In der bisherigen Umsetzung können Stile beschrieben werden, die auf der Subsystemebene nur eine Art von Elementen haben. Dies ist bei den betrachteten Architekturstilen und Beispielsystemen ausreichend. So hat der Stil „Schichtenarchitektur“ nur Elemente der Elementart „Schicht“. Bei Stilen auf der Klassenebene wird eine Menge von Elementarten aufgezählt. Beispielsweise hat der WAM-Architekturstil unter anderem die Elementarten Werkzeug, Material und Fachwert.

3.2.3 Bestimmung der Elemente

Im Architekturmodell soll für jedes Architekturelement eine Repräsentation angelegt werden. Dafür ist dem Werkzeug bekannt zu machen, welche Architekturelemente (der unterschiedlichen Elementarten) in dem System vorhanden sind. Hier sind zwei Varianten zu unterscheiden:

Variante 1 - Identifikation der Elemente im Quelltext:
Voraussetzung ist, dass die Elemente eine Repräsentation im Quelltext haben und diese über bestimmte Merkmale identifiziert werden kann. Für diese Variante ist in der Architekturbeschreibung anzugeben, wie die Elemente einer Art im Code erkannt werden können.

Variante 2 - Aufzählen der Elemente:
Der Stil gibt einen festen Satz von Architekturelementen vor oder die Elemente können dem Code nicht oder nur schwer entnommen werden. Die Elemente sind in diesem Fall in der Architekturbeschreibung aufzuzählen.

Architekturstile auf der Klassenebene werden durch eine Klasse im Quelltext repräsentiert (siehe 2.3.6). Meist ist es möglich Klassen, die ein Architekturelement einer bestimmten Art repräsentieren, anhand verschiedener Kriterien zu erkennen. Um alle Klassen zu finden, die Elemente einer Art repräsentieren, können unterschiedliche Eigenschaften, wie Namenskonventionen, Vererbung, Packagezugehörigkeit und Java Annotationen betrachtet werden. Bei der Verwendung des JWAM-Rahmenwerkes können viele WAM-Elemente im Code auch daran erkannt werden, dass sie von bestimmten Basisklassen erben.

Architekturelemente auf der Subsystemebene können mit den Mitteln der Programmiersprache meist nicht beschrieben werden. Daher haben sie oft keine eindeutige Repräsentation im Quellcode, über die sie zu identifizieren sind. Um diese Elemente dennoch dem Werkzeug bekannt zu machen, müssen sie in der Beschreibung des Soll-Modells aufgezählt werden. Ein Beispiel für diese Variante ist die Schichtenarchitektur des JCommSys. Die Schichten können dem Code nicht entnommen werden. Zu einem gewissen Grad spiegeln sich die Schichten zwar in der Packagestruktur wieder, diese Übereinstimmung reicht jedoch nicht aus, um über die Packages die Schichten zu identifizieren.

Diese Variante ist nicht grundsätzlich auf Architekturstile der Subsystemebene beschränkt. Es ist denkbar, dass in Fällen, in denen die Menge der Architekturelemente festgelegt oder eine Identifikation im Code schwierig ist, die Aufzählung von Elementen auch auf der Klassenebene Vorteile bietet.

Es ist auch denkbar, Elemente auf der Subsystemebene im Code zu identifizieren. Als Kriterien könnten dabei Packages oder Quelltextverzeichnisse verwendet werden. Da die Elementmengen auf dieser Ebene meist überschaubar sind und sich selten ändern, wird dieser Fall hier nicht weiter betrachtet. Stattdessen wird hier immer die Variante 2 angewandt.

Im Allgemeinen ist es vorzuziehen, die Elemente im Code zu identifizierten, sofern dies möglich ist. Dies entspricht dem Single-Source-Prinzip und verringert damit den Wartungsaufwand und das Risiko, Elemente zu vergessen. Eine Aufzählung von Elementen kann dennoch hilfreich sein, wenn Regeln sich auf bestimmte Elemente (nicht auf Elementarten) beziehen und dazu eine Benennung der Elemente im Rahmen einer Aufzählung vorgenommen wird. Dies ist beispielsweise bei der Schichtenarchitektur der Fall.

Aufgrund der Anforderungen, die sich aus den gegebenen Regelkatalogen ergeben, unterstützt der entwickelte Prototyp die Varianten nicht in vollem Umfang. In der bisherigen Umsetzung müssen die Architekturelemente auf der Subsystemebene aufgezählt werden. Auf der Klassenebene wird nur die Identifikation der Elemente unterstützt.

3.2.4 Zuordnung von Code- zu Architekturelementen

Die Architekturbeschreibung definiert die Elementarten und gibt an, welche Elemente es gibt bzw. wie die Menge der Elemente ermittelt werden kann. Darüber hinaus wird auch der Bezug der Architekturelemente zu den Elementen des Codes hergestellt. Die Codeelemente (des Systemmodells) werden also auf die Architekturelemente (des Architekturmodells) abgebildet. Auf der Klassenebene ist diese Abbildung durch die Identifikation der Elemente im Code implizit. Eine Klasse, die ein Architekturelement identifiziert, ist diesem naturgemäß zugeordnet. Da die Elemente auf der Subsystemebene jedoch aufgezählt werden, ist zusätzlich anzugeben, welche Klassen den einzelnen Elementen zuzuordnen sind. Auch hier können wieder unterschiedliche Kriterien verwendet werden, um die entsprechenden Klassen auszuwählen.

Sowohl bei der Identifikation von Elementen auf der Klassenebene, als auch bei der Zuordnung von Code- zu (aufgezählten) Architekturelementen auf der Subsystemebene, werden Codeelemente anhand von Kriterien ausgewählt und auf Architekturelemente abgebildet. Abbildung 7 veranschaulicht dies.

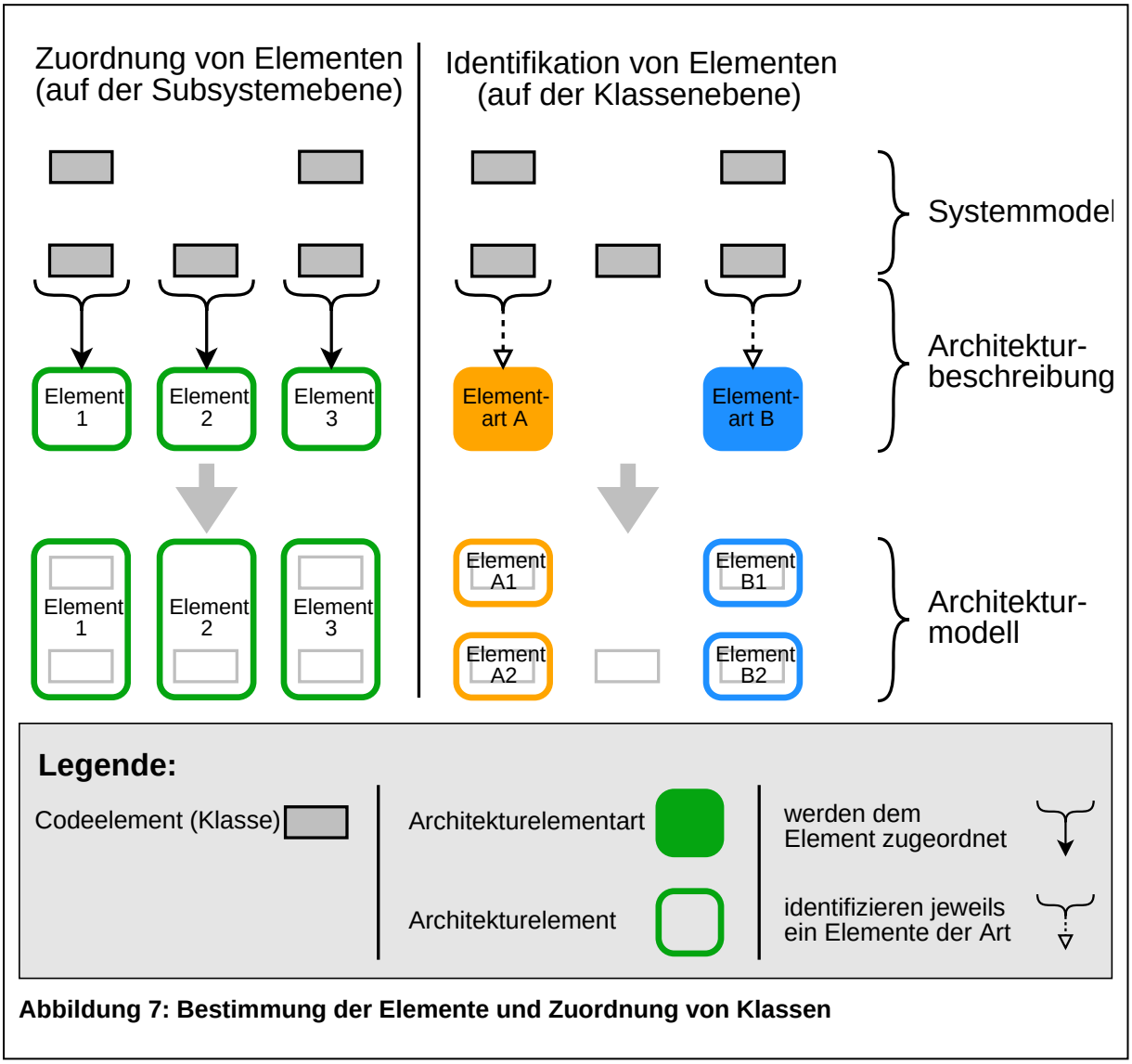

Abbildung 7: Bestimmung der Elemente und Zuordnung von Klassen

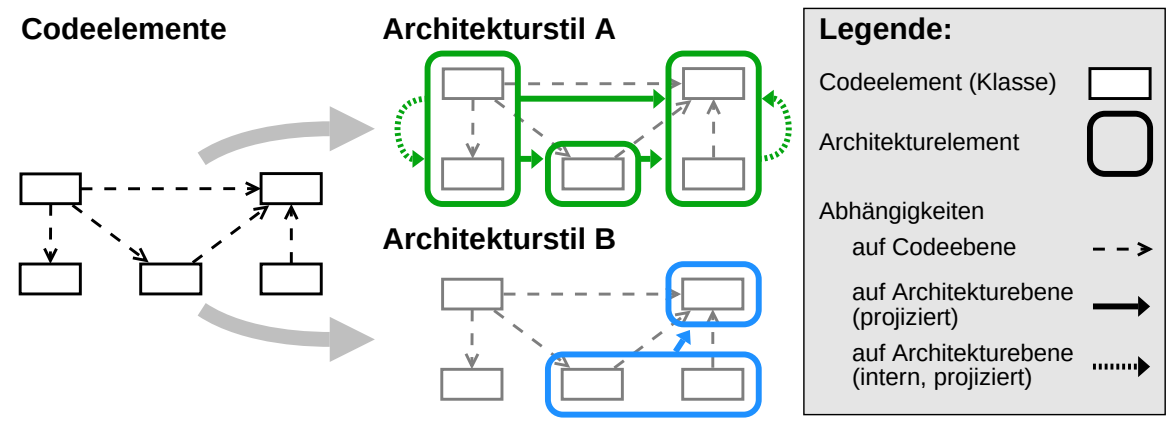

Abbildung 8: Projektion der Abhängigkeiten

3.2.5 Projektion von Abhängigkeiten

Die Abbildung von Codeelementen auf Architekturelemente wird auch für Abhängigkeiten zwischen den Elementen nachvollzogen. Abhängigkeiten zwischen den Elementen des Systemmodells werden also auf Abhängigkeiten zwischen Architekturelementen im Architekturmodell projiziert (siehe Abbildung 8). Eine Abhängigkeit zwischen zwei Klassen kann dabei auf eine Abhängigkeit zwischen Architekturelementen, wie zum Beispiel zwei Schichten, abgebildet werden. Wenn die beiden Klassen demselben Architekturelement zugeordnet sind, kann allerdings auch eine Abhängigkeit innerhalb eines Architekturelements entstehen. Die Projektion entspricht dem Vorgang, der im Zusammenhang mit Software Reflexion Models (siehe 2.2.2) auch als Lifting bezeichnet wird.

3.2.6 Zusammengesetzte Elemente

Zusammengesetzte Elemente konnten im Rahmen dieser Arbeit nur am Rande behandelt werden. Die bisherige Umsetzung ermöglicht, zusammengesetzte Elemente sowohl auf der Klassen- als auch auf der Subsystemebene zu modellieren. Dabei orientiert sie sich aber stark an den untersuchten Architekturstilen.

Elemente auf der Subsystemebene können Sub-Elemente besitzen, die zusammen mit ihnen aufgezählt werden. Damit ist ein Sub-Element eindeutig dem übergeordneten Element zugeordnet. Auf diese Weise können Elemente verfeinert und beispielsweise Sub-Schichten beschrieben werden. Ein Sub-Element repräsentiert eine Teilmenge der Klassen des übergeordneten Elementes. Die Auswahl der Klassen für diese Teilmenge erfolgt wie auf der obersten Ebene anhand von Kriterien.

Um zusammengesetzte Elemente auf der Klassenebene zu modellieren, können beliebige Elemente miteinander verknüpft werden. So ist es möglich, Elemente anderen Elementen als Bestandteil zuzuordnen. Weiter können benannte Verknüpfungen zwischen Elementen eingerichtet werden. In WAM-Systemen können damit einem Werkzeug seine Subwerkzeuge und Komponenten (IAK, FK und GUI) zugeordnet werden.

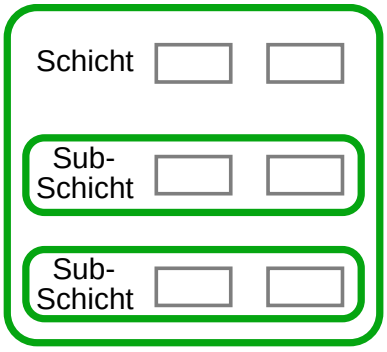

Abbildung 9:
Schicht mit Sub-Schichten

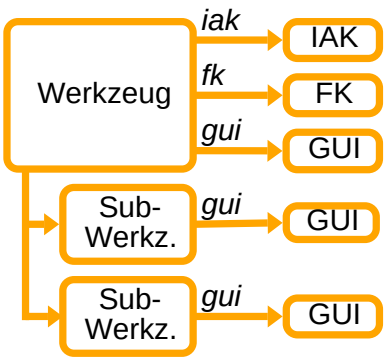

Abbildung 10:
Werkzeug mit Sub-Werkzeugen

Bei der Verknüpfung der Elemente ist zu beachten, dass diese auf der Klassenebene nicht aufgezählt, sondern im Code identifiziert werden. Die Elemente können daher nicht in der Architekturbeschreibung miteinander verknüpft werden. Vielmehr erfolgt einer automatisierte Zuordnung auf der Grundlage von Kriterien nach der Identifikation der Elemente. Dies ist im Abschnitt 4.4.2 genauer dargestellt.

3.3 Architekturregeln

Konzeptionell sind die Architekturregeln Teil der Architekturbeschreibung. Als Bestandteil eines Architekturstils sind sie in der XML-Datei aufgelistet. Inhaltlich werden die Architekturregeln von den Benutzern des Werkzeuges allerdings algorithmisch in Java beschrieben. Für jede Regel gibt es eine Java-Klasse, die von der Umgebung aufgerufen wird, um bestimmte Elemente des Architekturmodells zu prüfen. Sie analysiert die übergebenen Elemente und meldet ggf. Architekturverletzungen. In dieser Arbeit wurden auch Alternativen zur Beschreibung der Regeln in Java betrachtet. In Kapitel 5 sind diese Alternativen und der Aufbau der Regelklassen genauer dargestellt. Abhängig von der Ebene des Stils werden bisher unterschiedliche Regelarten unterstützt:

Auf der Subsystemebene ist es möglich, die Abhängigkeiten zwischen bestimmten Elementen über Verbotsregeln einzuschränken. Die untersuchten Architekturstile auf dieser Ebene haben keine Element- und Gebotsregeln. Auf eine Unterstützung dieser Regeln auf der Subsystemebene wurde daher verzichtet. Es ist allerdings davon auszugehen, dass sie keine konzeptionellen Schwierigkeiten bereiten würde.

Auch auf der Klassenebene können Verbotsregeln beschrieben werden. Diese beziehen sich im Allgemeinen auf Abhängigkeiten zwischen Elementen bestimmter Arten. Es ist allerdings auch möglich, Abhängigkeiten zwischen bestimmten Elementen zu betrachten. Darüber hinaus können Regeln beschrieben werden, die Vorgaben für die Ausgestaltung einzelner Architekturelemente prüfen. Da diese Regeln auch die ein- und ausgehenden Abhängigkeiten analysieren können, ist es möglich, über Elementregeln auch Verbots- und Gebotsregeln abzubilden.

Zuordnungsregeln können bisher nicht geprüft werden. Das Werkzeug bietet in diesem Bereich allerdings eine Unterstützung, in dem es an der Oberfläche darstellt, welchem Element eine Klasse zugeordnet ist und welche Klassen innerhalb eines Architekturstils nicht zugeordnet wurden. Näheres zu dieser Darstellung folgt im Abschnitt 3.4.

3.3.1 Ausnahmen

Fast immer, wenn eine Architektur bestimmten Regeln unterworfen wird, gibt es Ausnahmen. Eine Ausnahme ist schränkt eine Regel ein und führt dazu, dass sie für bestimmte Elemente oder Beziehungen nicht gilt. Ausnahmen können auch als Randbedingung der Regel aufgefasst werden. So verbietet eine WAM-Regeln Interaktionskomponenten den direkten Zugriff auf Materialien. Ausgenommen sind Fälle, in denen komplexe tabellarische Materialien gehandhabt werden müssen. In solchen Fällen darf die Interaktionskomponente Lesezugriff auf ein Material haben. Um eine solche Ausnahme zu berücksichtigen, kann man entweder die Randbedingung mit der Regel beschreiben oder, falls dies nicht möglich ist, einzelnen Interaktionskomponenten den Zugriff auf ein entsprechendes Material explizit erlauben.

3.3.2 Nicht strikte Regeln

Es gibt Regeln, die nicht strikt sind, also eher den Charakter von Richtlinien haben (Karstens 2005). Für diese Regeln ist im Einzelfall abzuwägen, ob sie geprüft werden sollen. Um der unterschiedlichen Gewichtung von Regeln Rechnung zu tragen, kann man Regelverletzungen als Fehler, Warnung oder Information melden. Karstens hat einige Regeln als nicht strikt be-

zeichnet, weil sie nur unter bestimmten Bedingungen gelten. In solchen Fällen kann eine genauere Beschreibung der Regel, die diese Bedingungen bzw. die Ausnahmen berücksichtigt eine bessere Strategie sein.

3.4 Einbindung in die Entwicklungsumgebung

Das Werkzeug wurde mit dem Ziel entwickelt, eine direkte Rückmeldung zur Einhaltung von Architekturverletzungen geben zu können, also Architekturverletzungen direkt im Quelltext unmittelbar nach dessen Änderung anzeigen zu können. Um dies zu erreichen, ist das Werkzeug zur Architekturüberprüfung als Plugin in die verbreitete Entwicklungsumgebung Eclipse integriert.

Wenn eine Architekturbeschreibung vorliegt, prüft dieses Plugin kontinuierlich, ob die darin formulierten Regeln eingehalten werden. Die Prüfung muss dafür nicht manuell angestoßen werden. Entdeckte Architekturverletzungen werden an der verursachenden Stelle im Code visualisiert (siehe Abschnitt 4.3). Dies funktioniert auch, wenn betroffene Java-Dateien nicht fehlerfrei übersetzt werden konnten.

Mit dem Architekturmodell wird ein Modell der Architektur eines Systems erstellt. Dieses beinhaltet deutlich mehr Informationen zu den umgesetzten Architekturstilen als in Entwicklungsumgebungen normalerweise dargestellt wird. Durch die Einbindung in die Entwicklungsumgebung kann, unabhän-gig von der automatisierten Architekturüberprüfung, von diesem Modell während der Entwicklung profitiert werden.

Im Rahmen dieser Arbeit wurde mit der *Architektur View* (siehe Abbildung 11) eine Oberflächenkomponente erstellt, die wesentliche Teile des Architekturmodells visuali-siert. Dargestellt sind die Stile mit ihren Elementarten und Elementen sowie die zugeordneten Klassen. Damit hilft, sie die Architekturbeschreibung zu erstellen. Andererseits bietet sie einen nützlichen Zugang zur Architektur des Systems.

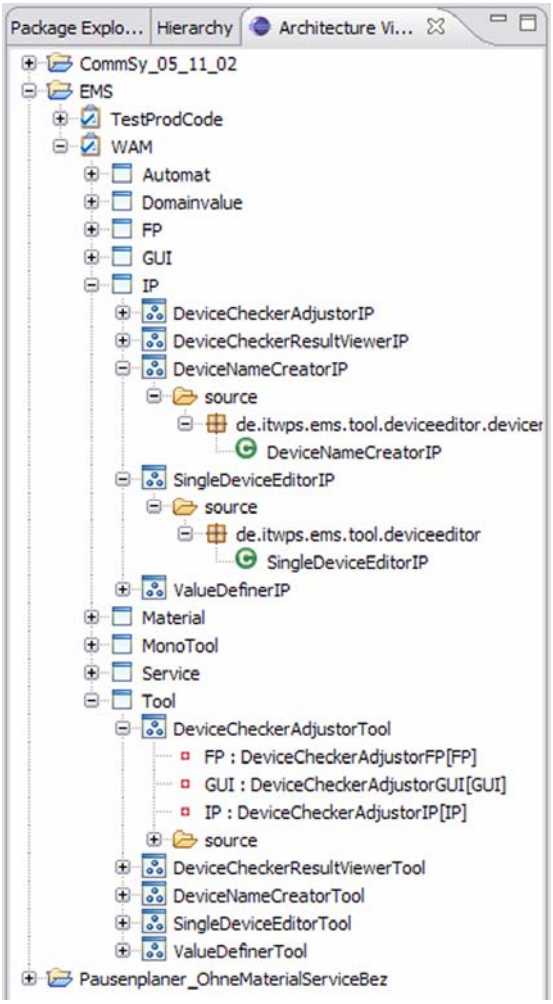

Abbildung 11: Architecture View

Kapitel 4 Benutzung

In diesem Kapitel sind die Aspekte des Werkzeuges beschrieben, die über die konzeptionelle Umsetzung hinausgehen, aber für die Benutzung relevant sind.

4.1 Architekturbeschreibung

Die Architekturbeschreibung wird in einer XML-Datei angelegt. Dabei beschreiben die Benutzer des Werkzeuges deklarativ die Architekturstile des untersuchten Systems. XML bietet dabei als Beschreibungssprache einige Vorteile. Entwickler und Architekten sind mit dem XML-Format im Allgemeinen vertraut. Beschreibungen in XML sind sowohl für Personen, als auch für ein Software-Werkzeug relativ leicht interpretierbar. Für die Architekturbeschreibung wurde im Rahmen der Arbeit eine XML-Schemadatei entwickelt. In dieser ist die Struktur der XML-Datei definiert, sodass ein eingesetzter XML-Editor die Entwickler beim Beschreiben der Architekturstile unterstützt.

Die XML-Datei muss im Projektverzeichnis abgelegt werden. Dort wird sie von dem Werkzeug eingelesen und in eine Objektstruktur überführt. Der Aufbau der Architekturbeschreibung wird im Folgenden anhand des JCommSys mit dessen Schichtenarchitektur und am Beispiel des nach dem WAM-Ansatz entwickelten Pausenplaners verdeutlicht.

Im Zentrum der Architekturbeschreibung eines Systems stehen dessen Architekturstile. In der XML-Datei werden sie nebeneinander aufgelistet:

```
<architecture-description
   xmlns="http://www.arnescharping.de/ArchitectureDescription"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.arnescharping.de/ArchitectureModel
   ArchitectureDescription.xsd" classpath="../RulesProject"
>
   <class-style name="WAM"> … </class-style>
   <style-subsyslevel name="Fachliche_Schnitte"> … </subsys-style>
</architecture-description>
```

Jeder Stil erhält dabei einen eindeutigen Namen, der verwendet wird, um Regelverletzungen zu beschreiben und um Bezüge zwischen Stilen herstellen zu können. Weiter muss für jeden Stil kenntlich gemacht werden, ob es sich um einen Stil auf der Klassen- oder auf der Subsystemebene handelt. Schließlich gibt ein Attribut `classpath` das Verzeichnis an, in dem die class-Dateien mit den Architekturregeln liegen.

4.1.1 Elementarten

Ein Architekturstil definiert Arten von Architekturelementen. Jede dieser Elementarten hat einen Namen, der innerhalb des Stils eindeutig ist. WAM definiert als Elementarten unter anderem Werkzeuge, Materialien und Fachwerte.

```
<class-style name=”WAM”>
   <elements>
      <type name=”Tool”> … </type>
      <type name=”Material”> … </type>
      <type name=”Domainvalue”> … </type>
      …
   </elements>
   …
</class-style>
```

Im Gegensatz dazu wird bei einer Schichtenarchitektur nicht zwischen verschiedenen Elementarten unterschieden: Alle Architekturelemente sind Schichten.

```
<subsys-style name=”Schichtenarchitektur”>
   <elements type=“Schicht“>
      …
   </elements>
   …
</subsys-style>
```

4.1.2 Elemente

Die Informationen zur Bestimmung der Elemente werden innerhalb des Tags der Elementart hinterlegt. Auf der Klassenebene werden Kriterien für die Identifikation der Elemente benötigt. Für jede Elementart ist ein Kriterium anzugeben, mit dem die Klassen bestimmt werden können, die Elemente dieser Art repräsentieren. Auf der Subsystemebene werden die Elemente aufgezählt. Dabei ist für jedes Element über Kriterien anzugeben, welche Klassen ihm zuzuordnen sind. (vgl. Absatz 3.2.3 und 3.2.4)

Sowohl auf der Klassenebene als auch auf der Subsystemebene sind also Klassen anhand von Kriterien auszuwählen. Dies geschieht über sogenannte *Filter* (vgl. Abbildung 12). Filter beschreiben die Kriterien zur Auswahl von Klassen und beziehen sich dabei auf deren Eigenschaften, wie beispielsweise die Zugehörigkeit zu einem Package. Filter können aber auch andere Filter miteinander logisch verknüpfen.

Bei dem WAM-Beispiel (Klassenebene) können die meisten Elemente über die Verwendung von Basisklassen aus dem WAM-Rahmenwerk (JWAM) identifiziert werden. So erbt jeder Fachwert (engl.: Domainvalue) von der Klasse `DomainValueImpl`. Für die Identifikation dieser Elemente wird daher innerhalb des Tags der Elementart ein entsprechender Filter angegeben:

```
<type name=”Domainvalue”>
   <inheritance-filter name="DomainValueImpl"/>
</type>
```

Da die Schichtenarchitektur ein Stil auf der Subsystemebene ist, werden bei ihr alle Elemente aufgezählt.

```
<type name=”Schicht”>
   <element name=“Präsentation“> … </element>
   <element name=“Steuerung“> … </element>
   <element name=“Service“> … </element>
   <element name=“Mapper“> … </element>
   <element name=“Item“> … </element>
   <element name=“Domainvalue“> … </element>
</type>
```

Für jede Schicht ist anzugeben, welche Klassen dem Element zuzuordnen sind. Bei dem JCommSy-Beispiel kann beispielsweise die Auswahl der Klassen der Service-Schicht über einen Package-Filtern erfolgen:

```
<element name=”Service”>
   <package-filter pattern= "de.jcommsy.service..." />
</element>
```

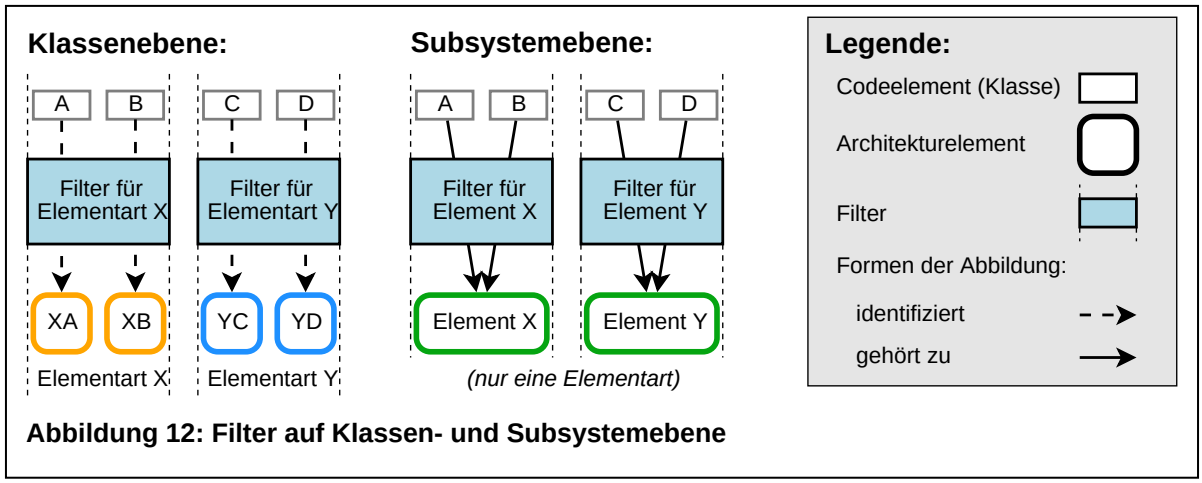

Abbildung 12: Filter auf Klassen- und Subsystemebene

Filter dienen also der Auswahl von Klassen in zwei verschiedenen Bereichen. Auf der Klassenebene beschreiben sie im Rahmen der Identifikation von Architekturelementen, welche Klassen jeweils ein Architekturelement einer bestimmten Elementart repräsentieren. Auf der Subsystemebene dienen sie der Zuordnung von Code- zu Architekturelementen. Jedes Architekturelement hat einen Filter, der angibt, welche Klassen dem Element zuzuordnen sind.

Neben der Packagezugehörigkeit und den verwendeten Basisklassen können weitere Eigenschaften von Klassen über Filter verwendet werden. In der bisherigen Umsetzung können die folgenden Eigenschaften als Filterkriterien berücksichtigt werden:

Name der Klasse	(`name-filter`)
Packagezugehörigkeit	(`package-filter`)
Basisklassen / Interfaces	(`inheritance-filter`)
Verzeichnis des Quellcodes	(`sourcefolder-filter`)
verwendete Annotationen	(`annotation-filter`)
Zugehörigkeit zu einem Element eines anderen Stils	(`element-filter`)
Ist die Klasse abstrakt?	(`concreteClass-filter`)
Ist die Klasse eine innere Klasse?	(`toplevelClass-filter`)

Die Filterkriterien können nicht nur einzeln verwendet, sondern auch miteinander kombiniert werden. Filter können als Konjunktion (`and`) und Disjunktion (`or`) verknüpft, dabei geschachtelt und negiert (`not`) werden. Ein Element-Filter kann beispielsweise genutzt werden, um die Auswahl der Klassen einer Schicht auf den Produktivcode zu beschränken.

```
<element name="Darstellung">
   <and>
      <or>
         <package-filter pattern="de.jcommsy.presentation..."[6] />
         <package-filter pattern="org.apache.jsp..." />
      </or>
      <element-filter style="Test_Prod_Code" element="Produktivcode" />
   </and>
</element>
```

[6] Der Name des Packages ist hier verkürzt dargestellt.

Bei der Abbildung von Code- auf Architekturelemente durch Filter sind zwei Sonderfälle zu betrachten:

1. Innerhalb eines Stils erfüllt eine Klasse die Filterkriterien mehrerer Elementarten bzw. Elemente.
2. Innerhalb eines Stils erfüllt eine Klasse die Filterkriterien keiner Elementart bzw. keines Elements.

Ob diese Fälle als Fehler anzusehen sind, hängt vom Architekturstil und dessen Zuordnungsregeln ab und steht in engem Zusammenhang mit den Zuordnungsregeln. Da für alle untersuchten Fälle gilt, dass keine Klasse mehreren Elementen eines Stils zugeordnet werden kann, behandelt das Werkzeug den ersten Fall als Fehler.

4.1.3 Filter für Stile

Filter werden nicht nur auf der Ebene von Architekturelementen, sondern auch auf der Ebene von Architekturstilen eingesetzt. Dadurch wird die Beschreibung von Architekturstilen erleichtert, die sich nur auf einen Teil des Systems beziehen. Hier ist als Filterkriterium die Zugehörigkeit zu einem Element eines anderen Stils von besonderer Bedeutung. Damit können Beziehungen zwischen Architekturstilen realisiert werden. Vor allem die Verschachtelung von Architekturstilen ist hier bedeutend (vgl. Abbildung 4: Beziehungen zwischen Stilen). Die Schichtenarchitektur des CommSys gilt beispielsweise nur für den Produktivcode. Der Architekturstil „Schichtenarchitektur“ hat daher als (stil-)*globalen Filter* einen Element-Filter, der dafür sorgt, dass nur Klassen betrachtet werden, die im Stil Test-Prod-Code zum Element Produktivcode gehören:

```
<subsys-style name=”Schichtenarchitektur”>
   <elements type=“Schicht“>
      …
   </elements>
   …
   <global-filter>
      <element-filter style="Test_Prod_Code" element="Produktivcode"/>
   </global-filter>
</subsys-style>
```

4.2 Beschreibung der Regeln

Architekturregeln werden als Bestandteile von Architekturstilen in der Architekturbeschreibung neben den Architekturelementen eines Stils aufgeführt.

```
<x-style name=”…”>
   <elements>
      …
   </elements>
   <rules>
      …
   </rules>
   <global-filter> … </global-filter>
</x-style>
```

Die bisherige prototypische Umsetzung unterstützt Element- und Gebotsregeln auf der Klassenebene sowie Verbotsregeln auf beiden Ebenen. Bei der Beschreibung der Regeln wurden im Laufe der Arbeit verschiedene Ansätze verfolgt, die im Folgenden vorgestellt und

gegeneinander abgewogen werden. Diese Ansätze spiegeln auch das iterative Vorgehen wieder, bei dem nach und nach weiteren Arten von Regeln betrachtet wurden.

4.2.1 Codierung der Regeln im Werkzeug

In einem ersten Prototyp wurde lediglich eine Regelart unterstützt, mit der Beziehungen zwischen bestimmten Elementen auf der Subsystemebene erlaubt oder verboten werden konnten. Hierfür wurde eine Grundform einer Regel vom Werkzeug vorgegeben, die über Parameter in der XML-Datei für verschiedene Architekturstile ausgestaltet werden muss. Für diese Ausgestaltung erwartet die als Whitelist-Rule bezeichnete Regel als Parameter eine Liste mit allen erlaubten Abhängigkeiten zwischen Elementen. Um beispielsweise die zentralen Regeln einer Schichtenarchitektur mit der Whitelist-Rule zu beschreiben, müssen alle erlaubten Beziehungen zwischen den Schichten aufgelistet werden.

```
<subsys-style name=”Schichtenarchitektur”>
   <elements type=“Schicht“>
      <element name=“Präsentation“>…/<element>
      <element name=“Steuerung“>…/<element>
      <element name=“Service“>…/<element>
      <element name=“Mapper“>…/<element>
      <element name=“Item“>…/<element>
      <element name=“Domainvalue“>…/<element>
      …
   </elements>
   <rules>
      <whitelist-rule>
         <dependency from=”Steuerung” to=”Service”/>
         <dependency from=”Steuerung” to=”Mapper”/>
         <dependency from=”Service” to=”Mapper”/>
         …
      </whitelist-rule>
   </rules>
   …
</subsys-style>
```

Dieser Ansatz ermöglicht eine gut lesbare Beschreibung und Einbindung der Regeln. Über die Parametrisierung können verschiedene Ausprägungen der Regeln formuliert werden. So kann mit der vorgestellten Grundform eine Schichtenarchitektur ebenso, wie die vertikale Unterteilung eines Systems beschrieben und geprüft werden.

Durch die Codierung der Regel innerhalb des Werkzeuges ist der Ansatz jedoch nicht sehr flexibel. Sobald eine Regel nicht mit den vom Werkzeug vorgesehen Grundformen und ihren Parametern ausgedrückt werden kann, ist eine Anpassung des Werkzeuges nötig.

4.2.2 Prädikatenlogische Ausdrücke

Ziel der zweiten Iteration war die Prüfung von Elementregeln auf der Klassenebene: Dabei lag der Fokus auf Regeln, die Anforderungen an die Schnittstellen der Elemente formulieren.

Als Beispiele wurden zwei WAM-Regeln ausgewählt:

Regel 3.2: Fachwerte haben keinen öffentlichen Konstruktor.

Regel 3.3: Materialien haben nicht nur Getter und Setter.

Um möglichst viele Regeln beschreiben zu können, wurde der Ansatz einer generischen Elementregel verfolgt. Die einfache Parametrisierung der vorangegangenen Iteration, für eine

solche Regel jedoch nicht ausreichend. Es wurde daher in Erwägung gezogen, beispielsweise die Anforderungen an eine Schnittstelle deklarativ in einer prädikatenlogischen Sprache innerhalb der XML-Datei zu beschreiben.

Die vorgestellten Regeln könnten in einer solchen Sprache folgendermaßen formuliert werden:

Regel 3.2: Fachwerte haben keinen öffentlichen Konstruktor.

```
<element-rule type=”Fachwert” name=”HatKeinenÖffentlKonstruktor”>
   FORALL Constructor c IN element.constructors : NOT (c.isPublic)
</element-rule>
```

Regel 3.3: Materialien haben nicht nur Getter und Setter.

```
<element-rule type=”Material” name=”HatNichtNurGetterUndSetter”>
   EXISTS Method m IN element.methods : NOT(m.isGetter OR m.isSetter)
</element-rule>
```

Eine solche Beschreibung der Regeln ist übersichtlich und leicht verständlich. Die Bedeutung der Regeln ist bereits der XML-Datei zu entnehmen.

Voraussetzung für diesen Ansatz ist allerdings eine prädikatenlogische Sprache, die von Entwicklern zu erlernen ist und die vom Plugin interpretiert werden muss. Ein weiterer Aspekt ist, dass die vorgestellten Ausdrücke eine passende Faktenbasis voraussetzen. So wurde für die Beispiele angenommen, dass für die Architekturelemente auf Klassenebene die Methoden und Konstruktoren über einfache Attribute zugreifbar sind und dass diese wiederum über alle nötigen Eigenschaften (`isSetter`, `isPublic`, etc.) verfügen. Die Flexibilität des Ansatzes und die Lesbarkeit der Ausdrücke werden durch die vom Werkzeug gegebene Faktenbasis begrenzt.

Für den Anspruch dieser Arbeit, die Prüfung festgelegter Regelkataloge zu unterstützen, wäre diese Einschränkung zwar hinnehmbar, der vorgestellte Ansatz wurde jedoch nach einer Abwägung des Aufwandes und der damit erreichbaren Flexibilität nicht weiter verfolgt.

4.2.3 Regelbeschreibung mit Java

Seit der zweiten Iteration werden die Architekturregeln in Java-Klassen beschrieben. Anders als beim ersten Ansatz sind die Regeln nicht Bestandteil des Werkzeuges, sodass sie von den Benutzern (also Entwicklern und Architekten) erstellt und verändert werden können. Mit Java wird eine mächtige Sprache verwendet, die den meisten Benutzern vertraut ist.

Die Regeln werden in Java nicht deklarativ sondern imperativ beschrieben. Jede Regelklasse implementiert eine Methode mit einem Prüfalgorithmus. Beispielsweise wird der Prüfmethode bei Elementregeln für jedes Element der entsprechenden Elementart die Repräsentation des Elementes aus dem Architekturmodell in einem Methodenaufruf als Parameter übergeben. Die Methode überprüft das Element und meldet gegebenenfalls die entdeckten Architekturverletzungen. Die Lesbarkeit der Regeln ist damit etwas schlechter als bei einer deklarativen Beschreibung, bei einer entsprechenden Unterstützung durch ein API kann dieser negative Effekt jedoch vermindert werden.

Im Folgenden sind die Java-Klassen, zur Beschreibung und Prüfung der Beispielregeln aufgeführt:

Regel 3.3: Materialien haben nicht nur Getter und Setter.

```
public class NotOnlyGetterAndSetter extends AbstractClasslevelElementRule
{
   public void check(ClasslevelElement element)
   {
      ClassInfo clazz = element.getIdentifyingClass();
      List<MethodInfo> methods = clazz.getMethods();
      for (MethodInfo method : methods)
      {
         if (!method.isGetter() && !method.isSetter())
            return; // passende Methode gefunden
                    // => Regel erfüllt
      }
      createRuleViolation(me);
   }
}
```

Regel 2: Fachwerte haben keinen öffentlichen Konstruktor.

```
public class NoPublicConstructor extends AbstractClasslevelElementRule
{
   public void check(ClasslevelElement element)
   {
      ClassInfo clazz = element.getIdentifyingClass();
      List<ConstructorInfo> constructors : clazz.getConstructors();
      for (ConstructorInfo constructor : constructors)
      {
         if (!constructor.getVisibility() == Visibility.PUBLIC)
            createRuleViolation(constructor);
      }
   }
}
```

Um den Bezug zwischen Regel und Architekturstil herzustellen, muss die Regelklasse in die XML-Datei eingebunden werden. Dabei wird auch festgelegt, auf welche Elemente die Regel anzuwenden ist.

```
<element-rule type=”Material” class=”NotOnlyGetterAndSetter”/>
<element-rule type=”Domainvalue” class=”NoPublicConstructor”/>
```

Die Java-Klassen mit den Regeln werden in der vertrauten Umgebung entwickelt und zusammen mit der Architekturbeschreibung während der Prüfung geladen. Wenn die Soll-Architektur geändert wird, können so die Stile und Regeln im normalen Entwicklungsprozess angepasst werden. Allerdings sollten die Regeln sollten in einem eigenen Eclipse-Projekt abgelegt werden, da die Regelklassen eine Reihe spezieller Bibliotheken im Classpath benötigen.

Das API für die Java-Regeln wird von dem Werkzeug vorgegeben. Die Regelklassen haben direkten Zugriff auf die Elemente des Architekturmodells. In den Beispielen bietet das Architekturmodell passende sondierende Methoden für die Prüfung der Regeln. Anders als bei dem prädikatenlogischen Ansatz gibt es hier jedoch keinen Bruch in der Sprache. Falls benötigte Methoden zur Sondierung der Elemente fehlen, können Benutzer diese in Hilfsmethoden oder -klassen nachimplementieren. Da die Regelklassen in der Umgebung des Eclipse-Plugins ausgeführt werden, steht ihnen diese Umgebung vollständig zur Verfügung. Sie können neben

dem Architektur- und Systemmodell auch die von Eclipse zur Verfügung gestellten Mechanismen der Codeanalyse verwenden. So können die Regelklassen zu einem `ClassInfo`-Objekt des Systemmodells den Quellcode auf Dateiebene betrachten oder ein entsprechendes JavaModel-Objekt oder eine AST-Repräsentation erstellen. Die Regeln haben damit dieselben Möglichkeiten zur Codeanalyse, wie das Werkzeug selbst.

Die Klassen mit den Regeln und die Hilfsklassen zur Codeanalyse können als Bibliotheken für unterschiedliche Projekte verwendet und wiederverwendet werden. Um Regeln an projektspezifische Bedürfnisse anpassen und sie damit leichter wiederverwenden zu können, ist es auch bei diesem Ansatz möglich, Regeln über die XML-Datei zu parametrisieren. Dazu können die Regelklassen zusätzliche Attribute und XML-Tags auswerten. Auf diese Weise wurde auch die vorgestellte Whitelist-Rule nachgebaut. Ebenso ist es denkbar eine Regelklasse zu schreiben, die als Parameter die prädikatenlogischen Ausdrücke des zweiten Ansatzes entgegennimmt und auswertet.

Im Gegensatz zu einer deklarativen Regelbeschreibung ist es mit den in Java geschriebenen Regeln leicht möglich, genaue Hinweise auf die Ursache einer Regelverletzung zu geben. So wird in der zweiten Beispielregel eine Architekturverletzung für jeden öffentlichen Konstruktor eines Fachwertes erstellt. Dazu wird der Methode `createRuleViolation` das zugehörige `ConstructorInfo`-Objekt als Parameter übergeben und so als Ort der Verletzung interpretiert.

4.3 Anzeige von Regelverletzungen

Gefundene Architekturverletzungen werden Entwicklern über Marker signalisiert. Mit Markern können in Eclipse Meldungen wie Compilerfehler und -warnungen angezeigt werden, die an eine Ressource (ein Projekt oder eine Datei) gebunden sind. Marker einer Datei können darüber hinaus in der Datei eine Position haben und werden in den Editoren entsprechend visualisiert (siehe Abbildung 13).

Eclipse listet die Marker aller Projekte in der *Problems View* auf, über welche die entsprechenden Stellen im Code leicht erreicht werden können (siehe Abbildung 14).

Die Marker von Eclipse sind abgestuft in Fehler, Warnung und Information. In der derzeitigen Umsetzung werden Architekturverletzungen als Warnungen angezeigt, es wäre allerdings auch möglich dies von der entsprechenden Regel abhängig zu machen.

Über die Positionierung der Marker im Quellcode ist es möglich, Entwicklern relativ genaue Hinweise über die Ursachen von Architekturverletzungen zu geben. Gerade nicht erlaubte Abhängigkeiten sind so auf einzelne Statements im Code zurückzuführen. Dafür kann die Regelklasse die Position des Markers festlegen.

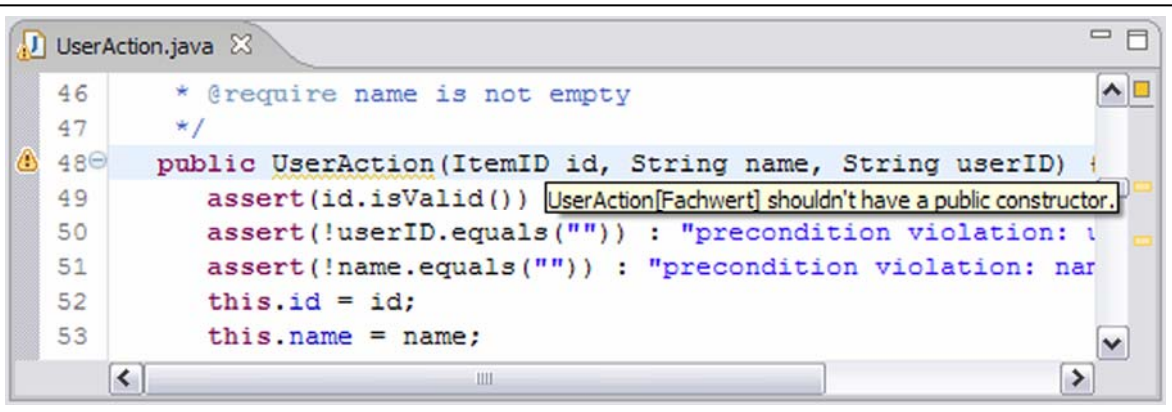

Abbildung 13: Anzeige von Architekturverletzungen im Editor

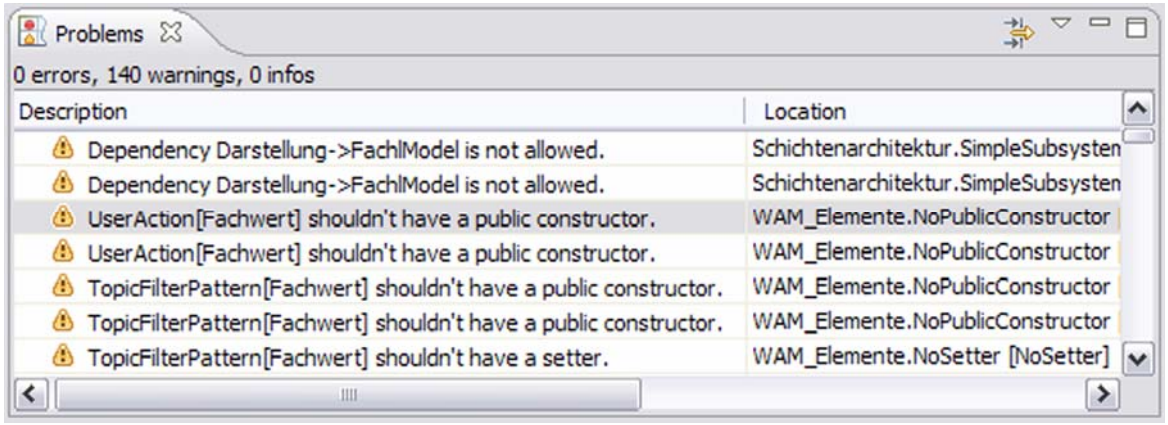

Abbildung 14: Auflistung von Architekturverletzungen in der Problems-View

4.4 Zusammengesetzte Architekturelemente

4.4.1 Subsystemebene

Die Elemente eines Stils auf der Subsystemebene werden in der Architekturbeschreibung aufgezählt. Dabei beschreiben Filter die Menge der Klassen, die den einzelnen Elementen zuzuordnen sind. Sub-Elemente werden weitgehend analog unterhalb der Elemente beschrieben.

```
<element>
   <element name="Service">
      <package-filter pattern="de.example.service..."/>
      <element name="Interface">
         <package-filter pattern="de.example.service"/>
      </element>
      <element name="Internal">
         <package-filter pattern="de.example.service.internal"/>
      </element>
   </element>
   <element name="Tool">
      <package-filter pattern="de.example.client..."/>
   </element>
</elements>
```

Der Filter des Sub-Elementes wird dabei nur auf Klassen angewandt, die bereits den Filter des übergeordneten Elementes passiert haben. Um auf Sub-Elemente in Regeln oder Element-Filtern Bezug zu nehmen, kann eine einfache Pfad-Notation verwendet werden.

```
<rules>
   <dependency-rule from="Tool" to="Service.Internal"
                    class="NoDependencyAllowed"/>
</rules>
```

4.4.2 Klassenebene

Da die Elemente auf der Klassenebene nicht aufgezählt, sondern im Code identifiziert werden, erfolgt die Zuordnung von Elementen über eine Initialisierungsklasse. Diese wird von den Benutzern wie eine Regelklasse bereitgestellt und für eine Elementart in der Architekturbeschreibung eingebunden.

```
<elements>
   <type name=“Tool“ className=“ToolInitializer“>…</type>
   <type name=“MonoTool“ className=“ToolInitializer“>…</type>
   <type name=“GUI“>…</type>
   <type name=”FP“>…</type>
   <type name=”IP“>…</type>
   …
</elements>
```

Um die Regeln prüfen zu können, müssen alle Elemente initialisiert sein. Die Initialisierung der Elemente erfolgt daher unmittelbar, nachdem die Elemente identifiziert wurden. Die Initialisierungsklasse wird dabei für jedes Element aufgerufen, sucht die zugehörigen Elemente im Architekturmodell und richtet entsprechende Verknüpfungen ein. Um die zugehörigen Elemente zu ermitteln, kann die Initialisierungsklasse unterschiedliche Merkmale betrachten, wie zum Beispiel:

- Namenskonventionen
- Packagezugehörigkeit
- Angaben in Annotationen
- bestehende Abhängigkeiten (z. B. Felddeklarationen)

Welches dieser Merkmal geeignet ist, hängt vom Projekt ab. Um bei der Untersuchung der WAM-Systeme die Werkzeugkomponenten und Sub-Werkzeuge zuverlässig zu bestimmen, wurden zusätzliche Annotationen für die Werkzeugklassen eingeführt und von der Klasse `ToolInitializer` ausgewertet.

Kapitel 5 Technische Umsetzung

Das in dieser Arbeit entwickelte Werkzeug ist in die für Java verbreitete Entwicklungsumgebung Eclipse integriert.

5.1 Eclipse

Eclipse basiert auf einer Plugin-Architektur nach dem OSGi-Standard. (Im OSGi-Sprachgebrauch werden die Plugins als Bundles bezeichnet). Das Eclipse Projekt besteht aus den folgenden Teilprojekten:

- Equinox: Implementierung eines OSGi-Rahmenwerkes
- Platform: Rahmenwerk für eine Eclipse-Anwendung unter anderem mit Oberflächenbibliotheken und einer workspace-basierten Ressourcenverwaltung
- Java Developement Tools (JDT): Oberfläche und Infrastruktur für die Java-Entwicklung
- Plugin Developement Environment: Oberfläche und Infrastruktur zur Entwicklung von Eclipse-Plugins

Diese Bestandteile bilden den Kern der Eclipse Entwicklungsumgebung. Darüber hinaus bilden sie die Grundlage für eine Vielzahl weiterer Projekte, in denen Erweiterungen für die Entwicklungsumgebung, eigenständige Anwendungen oder Anwendungsrahmenwerke entstehen.

Das in meiner Arbeit entwickelte Werkzeug ist als Eclipse-Plugin konstruiert. Durch die Plugin-Architektur bietet Eclipse auf allen Ebenen Anknüpfungspunkte für Erweiterungen. Durch diese kann man einerseits neue Funktionalität in die Entwicklungsumgebung integrieren, andererseits ist es möglich, dabei die Infrastruktur von Eclipse zu nutzen. So bietet Eclipse mit den JDT eine breite Unterstützung bei der Analyse des Quellcodes. (Mehr dazu im Abschnitt „Erstellung des Systemmodells“) Ein weiterer Vorteil einer Implementierung des Werkzeuges als Eclipse-Plugin ist, dass dieses wiederum offen für Erweiterungen ist. So wurde die Architecture View (siehe 3.4) als ein weiteres Plugin entwickelt, das auf dem Plugin zur Architekturüberprüfung aufsetzt.

5.2 Erstellung eines Architekturmodells

Um die Architekturregeln zu prüfen, erstellt das Werkzeug das Systemmodell und darauf basierend das Architekturmodell. Die Modelle sollen eine effiziente Prüfung der Regeln ermöglichen und werden daher als Objektstruktur im Speicher vorgehalten.

Das Architekturmodell wird in zwei logischen Schritten erstellt. Zunächst werden die Informationen zu allen Typen (Klassen und Interfaces) und den Abhängigkeiten zwischen diesen aus dem Code extrahiert und in das Systemmodell aufgenommen. In einem zweiten Schritt wird dieses um Informationen zu Architekturstilen und -elementen angereichert und so in das Architekturmodell überführt.

Das Systemmodell enthält Informationen zu den Typen eines Systems und zu den Abhängigkeiten zwischen diesen Typen. Ob eine Eigenschaft in das Modell aufgenommen wird, richtet sich danach, ob sie für die Zuordnung der Typen zu Architekturelementen, für die Identifikation von Architekturelementen oder für die Prüfung der Regeln von Bedeutung ist.

In der derzeitigen Umsetzung werden zu den im Code definierten Typen die folgenden Eigenschaften ermittelt:

- Name und Package
- Dateiname, (Quellcode-)Verzeichnis und (Eclipse-)Projekt
- Basisklasse und implementierte Interfaces
- Signaturen der Methoden (und Felder)
- Annotationen

Neben den Typ-Informationen werden auch die Abhängigkeiten zwischen den Typen bestimmt. Zu jeder Abhängigkeit werden folgende Informationen erhoben:

- Typ, von dem die Abhängigkeit ausgeht
- Typ, zu dem die Abhängigkeit besteht
- Art der Abhängigkeit:
  Import, Vererbung (Basisklasse oder Interface), Typecast, Felddeklaration, Variablendeklaration, Rückgabetyp, Feldzugriff, Methodenaufruf, Instanziierung, Übergabe als Argument
- ggf. Methode, zu der die Abhängigkeit besteht
- Code-Abschnitt, in dem die Abhängigkeit formuliert wird
  (z. B. eine Variablendeklaration oder ein Methodenaufruf)

Diese Informationen werden aus dem Quellcode gewonnen. Eclipse stellt hier mit den Java Developement Tools (JDT) eine mächtige Infrastruktur bereit. Eine statische Quellcodeanalyse kann damit auf der Textebene, über einen Syntaxbaum (AST) oder auf der Ebene des sogenannten *Java-Models* erfolgen. Das Java-Model ist eine von Eclipse generierte Objektstruktur, die Informationen für eine „äußere Sicht“ auf Typen enthält.

Das Java-Model ähnelt dem Systemmodell, da es vergleichbare Informationen zu den Typen des Systems enthält. Es kann das Systemmodell jedoch nicht ersetzen, da das Java-Model auf die Betrachtung einzelner Typen ausgelegt ist. Somit wird es auch von Eclipse nicht für das gesamte System vorgehalten, sondern kann bei Bedarf für einen einzelnen Typ erzeugt werden. Darüber hinaus enthält das Java-Model nicht alle benötigten Informationen. So sind die Methodenrümpfe (und damit ein Teil der ausgehenden Abhängigkeiten) sowie die Annotationen nicht Bestandteil des Java-Models. Aus diesem Grund wurde für das Systemmodell eine eigene Objektstruktur definiert.

Die Erstellung eines Systemmodells ist wiederum in zwei Phasen untergliedert. In der ersten Phase betrachtet das Werkzeug die Java-Dateien isoliert. Zu jeder Datei generiert es mithilfe der JDT ein Java-Model-Element und einen AST. Aus diesen entnimmt es die benötigten Informationen zu den definierten Typen sowie deren Abhängigkeiten zu anderen Typen und überträgt sie in das Systemmodell. Von den Typen, zu denen die Abhängigkeiten bestehen, sind in dabei zunächst nur die Namen bekannt. In einer zweiten Phase wird versucht, die Abhängigkeiten *aufzulösen*, also über den Namen das Objekt zu ermitteln, das den entsprechenden Typ im Systemmodell repräsentiert. Das Systemmodell wird auch erstellt, wenn der Java-Compiler nicht alle Klassen fehlerfrei übersetzten kann und auch fehlerbehaftete Klassen werden in das Modell aufgenommen. Einerseits kann das Werkzeug damit Architekturverletzungen auch in Dateien mit Compiler-Fehlern anzeigen, andererseits fließen die Fehler auch in das Systemmodell ein, was bei der Benutzung zu beachten ist. Eine häufige Folge von fehlerhaftem Code ist, dass Abhängigkeiten nicht aufgelöst werden können.

In das Systemmodell werden nur Typen aufgenommen, die als Quelltextdateien im Classpath liegen und daher vom Eclipse-Compiler übersetzt werden. Typen aus externen Bibliotheken sind also nicht repräsentiert. Dies hat zur Folge, dass auch Abhängigkeiten zu solchen Klassen nicht aufgelöst werden. Die Regeln können allerdings auch nicht aufgelöste Abhängigkeiten berücksichtigen.

In einem nächsten Schritt erstellt das Werkzeug auf Grundlage des Systemmodells und der Architekturbeschreibung das Architekturmodell. Wie das Systemmodell ist das Architekturmodell eine Objektstruktur. In ihm werden die Architekturstile mit den ermittelten Elementen und den Abhängigkeiten zwischen ihnen aufgenommen. Die Architekturelemente und Abhängigkeiten des Architekturmodells sind mit den zugeordneten Elementen des Systemmodells verknüpft, sodass es für die Regelklassen einen fließenden Übergang zwischen Architektur- und Systemmodell gibt.

5.3 Einbindung in den Build-Prozess

Ein zentrales Ziel bei der Entwicklung des Werkzeuges ist, möglichst direkt über die Einhaltung von Architekturregeln informieren zu können. Neben der Anzeige der Verletzungen im Code gehört dazu, dass die Verletzungen unmittelbar bei der Bearbeitung des Quellcodes angezeigt werden. Um dieses Ziel zu erreichen, wurde das Werkzeug in den Übersetzungsprozess von Eclipse eingebunden.

5.3.1 Builder

Die zentralen Einheiten des Übersetzungsprozesses von Eclipse sind die *Builder*. Builder leiten automatisch aus Ressourcen neue Arbeitsergebnisse in Form von neuen oder veränderten Dateien ab. Der Java-Builder sorgt zum Beispiel dafür, dass neue oder veränderte Java-Dateien compiliert werden und so immer aktuelle Class-Dateien vorliegen. Um ihre Aufgabe zu erfüllen, werden Builder von der Umgebung informiert, wenn sich eine Menge von Ressourcen verändert hat.

Das Plugin zur Architekturüberprüfung erweitert den Übersetzungsprozess mit einem eigenen Builder und nutzt damit den Mechanismus, um über den Bedarf einer Prüfung informiert zu werden. Die Architekturüberprüfung wird immer dann angestoßen, wenn eine Java-Datei gespeichert wurde, und beschränkt sich auf die Teile des Systems, die von der Veränderung betroffen sind. Dies ist essenziell, da die Erstellung des Systemmodells relativ viel Zeit in Anspruch nimmt.

Als Alternative zu einem Builder bietet Eclipse die Möglichkeit, über einen Beobachtungsmechanismus am Java-Build-Prozess teilzunehmen. Für einen Beobachter (Listener) sind jedoch alle Ressourcen für Änderungen gesperrt. Dies wäre für die Architekturüberprüfung problematisch, da von einem Beobachter gefundene Architekturverletzungen durch die Sperrung nicht als Marker an Dateien angehängt werden könnten.

Sobald der von dem Plugin zur Verfügung gestellte Builder für ein Projekt aktiviert wurde, wird er immer nach dem Java-Builder angestoßen und der Build-Prozess (im Folgenden kurz „Build“) durchgeführt. In Eclipse wird zwischen zwei Varianten des Builds unterschieden: Neben dem *vollständigen Build* (full build) gibt es den *inkrementellen Build* (incremental build). Ein inkrementeller Build wird angestoßen, wenn veränderte Ressourcen gespeichert werden. Die Umgebung ermittelt dabei, welche Ressourcen neu erzeugt, verändert oder gelöscht wurden und stellt dem Builder für die Verarbeitung ein entsprechendes Delta-Objekt zur Verfügung. Ein vollständiger Build wird beim ersten Build durchgeführt und wenn der Benutzer dies über den „clean“-Befehl ausdrücklich auslöst. Bei einem vollständigen Build werden ggf. alle bis dahin vorhandenen Ergebnisse verworfen und anschließend alle Ressourcen des Projektes verarbeitet.

5.3.2 Vollständige Prüfung

Der Builder für die Architekturüberprüfung verwirft bei einem vollständigen Build das Architektur- und das Systemmodell und alle Daten zu Architekturverletzungen. Danach wird das gesamte Projekt nach dem in Kapitel 3 vorgestellten Grundprinzip überprüft:

Zunächst wird die XML-Datei mit der Architekturbeschreibung neu eingelesen und in eine Objektstruktur überführt. Mit der Architekturbeschreibung werden auch die angegebenen Regelklassen geladen. In einem zweiten Schritt erstellt das Werkzeug das Systemmodell mit den im Code beschriebenen Typen und überführt es mithilfe der Architekturbeschreibung in ein Architekturmodell. Anschließend werden für jeden Stil die Regeln auf alle betroffenen Architekturelemente und Abhängigkeiten angewandt. Dazu erstellt das Werkzeug Exemplare von den Regelklassen und ruft die Prüfmethoden mit Elementen des Architekturmodells auf. Dabei entdeckte Verletzungen werden gesammelt und als Marker dargestellt.

5.3.3 Inkrementelle Prüfung

Ein inkrementeller Build wird jedes Mal angestoßen, wenn eine oder mehrere Dateien verändert wurden. Da dies sehr häufig vorkommt, ist die Prüfung auf die Teile des Systems beschränkt, die von der Änderung betroffen sein könnten:

- Die Architekturbeschreibung und die Regelklassen müssen nur neu geladen werden, wenn die entsprechenden Dateien seit dem letzten Build verändert wurden.
- Im System- und Architekturmodell werden die Teile aktualisiert, die von der Veränderung betroffen sind.
- Bei einer Änderung der Architekturbeschreibung wird das Architekturmodell neu erstellt und vollständig geprüft. Das Systemmodell muss dabei nicht erneuert werden. Falls die Architekturbeschreibung unverändert ist, reicht es aus, die Teile des Architekturmodells zu prüfen, die von der Änderung betroffen sein könnten.

Von der Änderung eines Elementes können neben dem Element selbst auch jene betroffen sein, die mit ihm in Beziehung stehen. Da die Regeln implizite Abhängigkeiten zwischen Elementen herstellen können, ist nicht immer zweifelsfrei festzustellen, ob eine Prüfung notwendig ist. In der bisherigen Umsetzung werden daher mehr Regelprüfungen durchgeführt, als nötig wären.

Um das Systemmodell zu erstellen, analysiert das entwickelte Werkzeug den Quellcode des Systems und nicht, wie der Sotograph, den Bytecode. Damit kann es auch Klassen prüfen, die nicht fehlerfrei übersetzt werden konnten. Damit können Architekturverletzungen früher entdeckt werden. Die Überprüfung von fehlerhaftem Code erfordert jedoch auch einen defensiven Umgang mit dem Systemmodell, in dem sich die Fehler widerspiegeln. So ist eine häufige Folge, dass Abhängigkeiten nicht aufgelöst werden können.

Kapitel 6 Überprüfung des JCommSys

6.1 JCommSy

In der Architektur des JCommSys wurden verschiedene Architekturstile umgesetzt. Teilweise nehmen diese Stile Bezug aufeinander. Die Stile und ihre Verschachtelung sind in Abbildung 15 grob dargestellt. Im Folgenden ist vorgestellt, wie die Stile mit dem entwickelten Ansatz beschrieben und geprüft wurden.

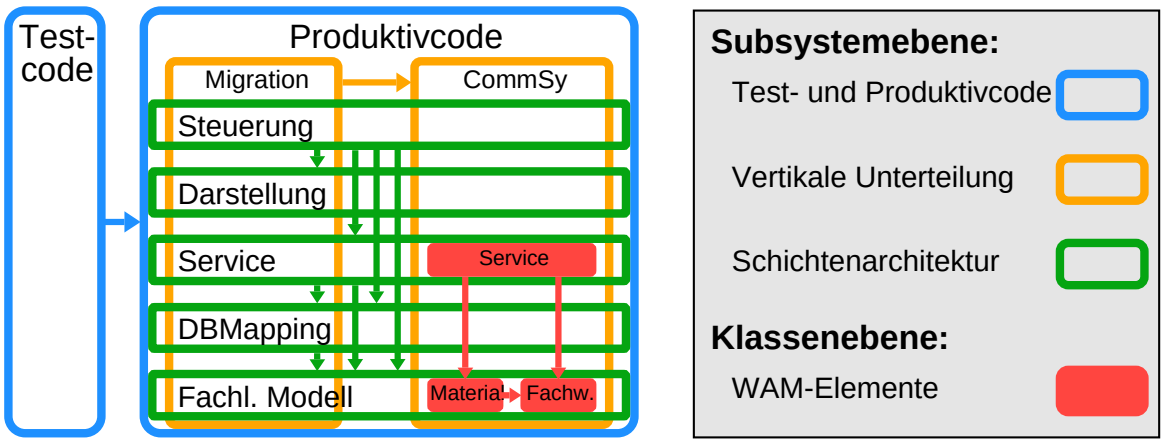

Abbildung 15: Architekturstile des JCommSys

6.2 Architekturstil „Test- und Produktivcode“

Wie in vielen anderen Systemen ist im JCommSy der Testcode von solchem, der im Produktivbetrieb verwendet wird (Produktivcode), getrennt. Diese Einteilung des Systems wird hier als Architekturstil „Test- und Produktivcode“ beschrieben. Ziel ist, zu verhindern, dass im Produktivcode Abhängigkeiten zum Testcode entstehen und dieser im Produktivbetrieb benötigt oder sogar ausgeführt wird.

Ebene: Subsystemebene
Elemente: Testcode, Produktivcode
Zuordnung: Quelltextverzeichnisse
Regeln: Verbotsregeln: 2 (geprüft: 2)

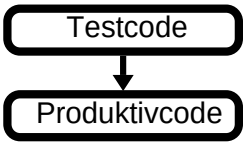

Abbildung 16: JCommSy Test- und Produktivcode

6.2.1 Elemente

Im JCommSy-Projekt befinden sich Testklassen in einem eigenen Quelltextverzeichnis. Die Zuordnung von Klassen zu Elementen kann daher über dieses Verzeichnis erfolgen.

```
<elements>
   <element name="Produktivcode">
      <or>
         <sourcefolder-filter folder="JavaSource" />
         <sourcefolder-filter folder="JSPSource" />
      </or>
   </element>
   <element name="Testcode"> … </element>
</elements>
```

6.2.2 Regeln

Für diesen Stil wurden zwei Regeln geprüft und dabei eine Verletzung entdeckt.

Regel 1.1:	***Produktivcode darf keinen Testcode nutzen.***
Prüfung:	Für die Prüfung wird die Regelklasse `SimpleSubsystemlevelDepRule` verwendet. Wie bei der in 4.2.1 vorgestellten Whitelist-Rule werden bei dieser Regel alle erlaubten Abhängigkeiten aufgezählt. In diesem Fall sind nur Abhängigkeiten von Test- zu Produktivcode erlaubt. `<dependency-rule className="SimpleSubsystemlevelDepRule" allowInnerDependencies="all" > <dependency from="Testcode" to="Produktivcode"/> </dependency-rule>` Über das Attribut `allowInnerDependencies` wird angegeben, dass Abhängigkeiten innerhalb der Elemente von der Regel nicht eingeschränkt werden.
Verletzungen:	Diese Regel wird im JCommSy nicht verletzt.

In den Klassen des Produktivcodes gibt es einzelne Methoden, die ausschließlich für Tests vorgesehen sind. Ursprünglich war dies im CommSy im Kommentar der Methode beschrieben. Für die Prüfung der Regel wurden diese Methoden mit einer zusätzlichen Annotation versehen. Genaugenommen gehören diese Methoden zum Testcode. Da die Zuordnung zu Architekturelementen auf der Klassenebene erfolgt, kann dies hier nicht abgebildet werden. Stattdessen wird über die folgende Regel wird sichergestellt, dass die Methoden nur von Testcode aufgerufen werden.

Regel 1.2:	***Methoden mit der Annotation @Test dürfen nur von Testcode oder von anderen Methoden mit der Annotation verwendet werden.***
Prüfung:	Die Regelklasse `DontUseTestMethods` prüft genau diese Regel. In der XML-Datei wird angegeben, dass nur Abhängigkeiten innerhalb des Produktivcodes zu betrachten sind. `<dependency-rule from="Produktivcode" to="Produktivcode" className="DontUseTestMethods"/>`
Verletzungen:	Es gibt eine Verletzung dieser Regel.

6.3 Architekturstil „Schichtenarchitektur“

Die Schichtenarchitektur des JCommSys wurde bereits in Abschnitt 2.3.4 vorgestellt.

Ebene:	Subsystemebene
Elemente:	Steuerung, Darstellung, Service, DB-Mapping, Fachliches Modell Beans (Sub-Schicht von Darstellung)
Zuordnung:	Packages und Klassennamen nur aus Produktivcode (vgl. Stil „Test- und Produktivcode“)
Regeln:	Zuordnungsregeln: 1 (geprüft: 0) Verbotsregeln: 3 (geprüft: 3)

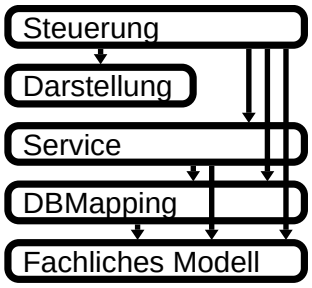

Abbildung 17: JCommSy Schichten

6.3.1 Elemente

Die Schichten werden aufgezählt, die Zuordnung erfolgt über Package- und Name-Filter. So gehören beispielsweise zur Darstellungs-Schicht die Klassen des `presentation`-Packages und die übersetzten JSPs (Package: `org.apache.jsp…`). Die Schicht hat eine Sub-Schicht Beans, der die Klassen über eine Namenskonvention zugeordnet werden. Dabei werden nur Klassen der Darstellungs-Schicht betrachtet.

```
<element name="Darstellung">
   <or>
      <package-filter pattern="de.jcommsy.presentation..." />
      <package-filter pattern="org.apache.jsp..." />
   </or>
   <element name="Beans">
      <name-filter pattern="*Bean"/>
   </element>
</element>
…
```

Die Schichtenarchitektur beschränkt sich auf den Produktivcode des JCommSys. Dazu wird für den Stil ein Element-Filter angegeben.

```
<global-filter>
   <element-filter style="Test_Prod_Code" element="Produktivcode" />
</global-filter>
```

6.3.2 Regeln

Die ersten beiden Regeln betreffen den Kern der Schichtung. Sie beziehen sich auf die Zuordnung der Klassen zu Schichten und geben vor, welche Schichten aufeinander zugreifen dürfen.

Regel 2.1:	***Jede Klasse ist einer der Schichten zugeordnet.***
Prüfung:	Dies ist eine Zuordnungsregel und kann bisher nicht geprüft werden.
Verletzungen:	Die Regel wird von vier Klassen verletzt: `CommsyException`, `ErrorContainer`, `GetItemsResult` und `LinkFactory`

Regel 2.2:	***Nur die folgenden Beziehungen zwischen den Schichten sind erlaubt:*** ***Steuerung nutzt Darstellung, Service und Fachliches Modell*** ***Services nutzt Fachliches Modell und DB-Mapping*** ***DB-Mapping nutzt Fachliches Modell.***

Prüfung:	Für die Prüfung wird die Regelklasse `SimpleSubsystemlevelDepRule` verwendet. Alle erlaubten Abhängigkeiten zwischen Schichten sind aufgezählt.
	`<dependency-rule className="SimpleSubsystemlevelDepRule" allowInnerDependencies="all" >`
	`<dependency from="Steuerung" to="Darstellung" />`
	`<dependency from="Steuerung" to="Service" />`
	`<dependency from="Steuerung" to="FachlModel" />`
	`<dependency from="Service" to="DBMapping" />`
	`<dependency from="Service" to="FachlModel" />`
	`<dependency from="DBMapping" to="FachlModel" />`
	`</dependency-rule>`
	Über das Attribut `allowInnerDependencies` wird angegeben, dass Abhängigkeiten innerhalb der Elemente von dieser Regel nicht eingeschränkt werden.

Verletzungen:	Es gibt zwölf Paare von Klassen, zwischen denen eine unerlaubte Abhängigkeit besteht. von der Darstellung zum Fachlichen Modell: 9 von der Darstellung zum DB-Mapping: 1 vom DB-Mapping zu den Services: 2 Alle diese Abhängigkeiten wurden gefunden.

Die dritte und vierte Regel betreffen Details in der Benutzung einzelner Klassen oder Methoden innerhalb der Schichten. Da sie dabei in engem Bezug zur Schichtenarchitektur stehen, sind sie diesem Stil zugeordnet.

Regel 2.3:	***Die Methode Item.setItemID(ItemID) darf nur aus der Schicht Mapper aufgerufen werden.*** Die Regel wurde in Hinblick auf eine flexiblere Prüfung neu formuliert: ***Methoden mit der Annotation @MapperOnly dürfen nur aus der Schicht Mapper aufgerufen werden.***

Prüfung:	Die Prüfung erfolgt über die Regelklasse `DontUseMapperMethods`.
	`<dependency-rule to="FachlModel"`
	`className="DontUseMapperMethods" />`

Verletzungen:	Diese Regel wird nicht verletzt.

Regel 2.4:	***Setter an Beans dürfen nur aus der Schicht Steuerung aufgerufen werden***. (Beans sind Klassen der Sub-Schicht `Darstellung.Beans`)
Prüfung:	Über Regel 2.2 wird sichergestellt, dass Beans nur aus den Schichten „Steuerung“ und „Darstellung“ verwendet werden. Über die Regelklasse `DontUseSetter` wird darüber hinaus geprüft, ob Setter von Beans aus der Darstellungsschicht aufgerufen werden. `<dependency-rule from="Darstellung" to="Darstellung.Beans" className="DontUseSetter" />`
Verletzungen:	Diese Regel wird nicht verletzt.

Das Werkzeug hat alle bekannten Verletzungen der drei geprüften Regeln entdeckt. Im Fall der nicht geprüften Zuordnungsregel bietet das Werkzeug zumindest eine Unterstützung für eine manuelle Überprüfung.

6.4 Architekturstil „Vertikale Unterteilung“

Das JCommSy ist ein Migrationsprojekt, bei dem das in PHP geschriebene CommSy nach Java migriert wird. Um die Migration schrittweise durchführen zu können, läuft der bereits migrierte Teil (JCommSy) zusammen mit dem PHP-CommSy in einem System. Für diesen integrierten Betrieb ist zusätzlicher Migrationscode erforderlich. Mit der „Vertikalen Unterteilung“ wird der Code des JCommSys von diesem Migrationscode getrennt. Da dieser nach der Migration entfernt werden soll, sind Abhängigkeiten aus dem CommSy-Code zum Migrationscode nicht erlaubt.

(Im JCommSy-Projekt wurde beabsichtigt, zu einem späteren Zeitpunkt auch eine fachliche Unterteilung vorzunehmen, und diese in die vertikale Unterteilung aufzunehmen.)

Ebene:	Subsystemebene
Elemente:	Migrationscode CommSy-Code
Zuordnung:	Packages (Annotationen) nur Klassen aus dem Produktivcode (vgl. Stil „Test- und Produktivcode“)
Regeln:	Zuordnungsregeln: 1 (geprüft: 0) Verbotsregeln: 1 (geprüft: 1)

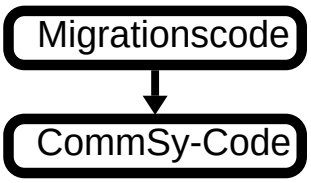

Abbildung 18: JCommSy Vertikale Unterteilung

6.4.1 Elemente

Im JCommSy befindet sich der Migrationscode in einem speziellen Package. Über einen Package-Filter können die Klassen des Migrationscodes daher leicht zugeordnet werden. Die Auswahl wird dabei durch einen Stil-Filter beschränkt, der nur Klassen des Produktivcodes akzeptiert.

6.4.2 Regeln

Die Regeln dieses Stils verhindern, dass im CommSy-Code Abhängigkeiten zum Migrationscode entstehen, da dieser nach der Migration entfernt werden soll.

Regel 3.1:	***Jede Klasse ist einem der Elemente zugeordnet.***
Prüfung:	Diese Regel wird nicht geprüft. (vgl. Regel 2.1)
Verletzungen:	Diese Regel wird nicht verletzt.

Regel 3.2:	***CommSy-Code darf nicht auf Migrations-Code zugreifen.***
Prüfung:	Die Prüfung erfolgt analog zu Regel 2.2 der Schichtenarchitektur.
Verletzungen:	Diese Regel wird nicht verletzt.

Bei diesem Stil wurden im JCommSy Mittel der Programmiersprache genutzt, um Architekturverletzungen zu verhindern. Dazu wurde ein Package `migration` gebildet, in dem alle Klassen des Migrationscodes enthalten sind. Eine unbeabsichtigte Verwendung von Klassen aus diesem Package wird über Sichtbarkeitsmodifikatoren verhindert. Damit wurde allerdings eine Abweichung von der Packagestruktur in Kauf genommen. Die Klassen des Migrationscodes können nicht nach Schichten auf Packages verteilt werden. Mit dem hier vorgestellten Werkzeug wäre eine bessere Lösung möglich. So könnte man die Migrationsklassen über Annotationen markieren und den Packages der Schichten zuordnen.

6.5 Architekturstil „WAM-Elemente“

Im JCommSy wird ein Ausschnitt der WAM-Modellarchitektur verwendet. Es gibt Services, Materialien und Fachwerte. Für diese Elemente wurden auch einige WAM-Regel übernommen.

Ebene: Klassenebene
Elemente: Services, Materialien (im JCommSy als Items bezeichnet), Fachwerte
Zuordnung: Zuordnung zu Schichten, Packages nur aus dem Produktivcode (vgl. Stil „Test- und Produktivcode“)
Regeln: Elementregeln: 2 (geprüft: 2)
Verbotsregeln: 1 (geprüft: 1)

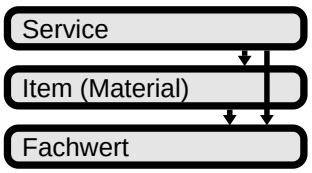

Abbildung 19: JCommSy WAM-Elemente

6.5.1 Elemente

Die WAM-Elemente sind bestimmten Schichten des JCommSys zugeordnet (siehe Abbildung 16). Die Services befinden sich in der Service-Schicht, Materialien und Fachwerte bilden zusammen die Schicht des fachlichen Modells. Um Services zu identifizieren, kann daher ein Element-Filter verwendet werden, der die Klassen der entsprechenden Schicht auswählt. Um Materialien und Fachwerte zu unterscheiden, werden Package-Filter verwendet.

```
<type name="Service">
   <element-filter style="Schichten" element="Service"/>
</type>
<type name="Item">
   <package-filter pattern="de.jcommsy.item" />
</type> …
```

6.5.2 Regeln

Für die WAM-Elemente des JCommSys gilt eine Auswahl der WAM-Regeln. Diese Auswahl wird durch die verwendeten Elementarten beschränkt. Weiter gelten für die Konstruktion der Fachwerte des JCommSys weniger strenge Vorgaben, als in typischen WAM-Systemen. Daher wurden nur einige Regeln für diese übernommen.

Regel 4.1:	***Services gehen mit Materialien um.***
	(Diese Regel ist auch Teil der WAM-Regel 2.2 in Abschnitt 7.1.5.)

Prüfung:	Die Regel `HasDepToElement` betrachtet, alle ausgehenden Abhängigkeiten von Elementen einer bestimmten Art und prüft, ob eine Abhängigkeit zu einem Element einer anderen Elementart existiert. So wird mit der folgenden Parametrisierung sichergestellt, dass alle Services eine Abhängigkeit zu einem Item haben.
	`<element-rule elementType="Service"`
	`  className="HasDepToElement"`
	`  toElementType="Item" />`

Verletzungen:	In der Baccalaureatsarbeit wurden keine Verletzungen dieser Regel entdeckt. Hier konnten vier Services entdeckt werden, die gegen die Regel verstoßen: `CommSySessionService, LinkageRuleService, ModifierService, ServiceRegistry`

Regel 4.2:	***Fachwert-Klassen bieten keine Operationen an, die es erlauben, das Fachwert-Objekt zu verändern.***
	Da mit einer statischen Codeanalyse nicht zuverlässig festgestellt werden kann, ob eine Methode den fachlichen Zustand eines Objektes verändert, wird diese Regel in einer entschärften Variante geprüft: ***Fachwerte haben keine Setter.*** (Diese Regel entspricht der WAM-Regel 4.3 in Abschnitt 6.5.2.)

Prüfung:	Die Regel `NoSetter` verbietet Setter an Elementen und wird auf alle Fachwerte angewandt.
	`<element-rule elementType="Fachwert"`
	`  className="NoSetter"/>`
	Ob eine Methode von der Regel als Setter behandelt wird, hängt ausschließlich von der Signatur ab. Zwar besteht die Möglichkeit, die Methodenrümpfe zu analysieren, aufgrund des damit verbundenen Aufwandes, wurde darauf aber verzichtet.

Verletzungen:	Die Regel wird erfolgreich geprüft. Die fünf Fachwerte, die diese Regel verletzen, wurden entdeckt: `AppointmentFilterPattern`, `GroupFilterPattern`, `MaterialFilterPattern`, `TopicFilterPattern` und `RequestContext`

Regel 4.3:	***Fachwerte dürfen nur auf Fachwerte zugreifen.*** ***Materialien dürfen nur auf Fachwerte und Materialien zugreifen.*** (Diese Regel ist auch Teil der WAM-Regel 1.1 in Abschnitt 7.1.4.)
Prüfung:	Prüfung über `SimpleClasslevelDepRule` (entspricht der `SimpleSubsystemlevelDepRule` auf der Klassenebene)

```
<dependency-rule className="SimpleClasslevelDepRule">
   <dependency from="Service" to="Item"/>
   <dependency from="Service" to="Fachwert"/>
   <dependency from="Item" to="Fachwert"/>
</dependency-rule>
```

Verletzungen:	Diese Regel wird nicht verletzt.

6.6 Nicht zugeordnete Regel

Die folgende Regel kann nicht einem einzigen Stil zugeordnet werden, da sie die Elemente verschiedener Stile miteinander in Beziehung setzt.

Regel 5.1:	***Exemplare der Services dürfen nur von Testklassen und der Klasse ServiceRegistry erzeugt werden.***
Prüfung:	Architekturregeln müssen in der Architekturbeschreibung einem Stil zugeordnet werden. Die Regelklassen haben dennoch Zugriff auf das gesamte Architektur- und Systemmodell. Daher kann auch diese Regel geprüft werden. Die Zuordnung zu einem Stil bestimmt, bei welchen Änderungen eine Regelprüfung angestoßen wird und welche Elemente dabei zur Prüfung übergeben werden. Aus diesem Grund kann die Regel nicht dem WAM-Stil zugeordnet werden, da die Regel dann nur bei Änderungen von WAM-Elementen involviert werden würde. Stattdessen ist die Regel der „Schichtenarchitektur“ zugeordnet, da auf diese Weise die Prüfung am Besten auf relevante Abhängigkeiten beschränkt werden kann: Die Schichtenarchitektur beschränkt sich auf den Produktivcode (vgl. Stil-Filter der Schichtenarchitektur), die Regeln des Stils werden also nur für die Prüfung von Produktivcode aufgerufen. Da alle Services (WAM-Elemente) in der Service-Schicht liegen, muss die Regel nur Abhängigkeiten zu Klassen in dieser Schicht betrachten. Die Regelklasse `DontCreateServices` prüft diese Abhängigkeiten und verbiete alle Objekterzeugungen (Art der Abhängigkeit). Die `ServiceRegisty` wird von der Regel als Ausnahme behandelt.

```
<dependency-rule to="Service"
                 className="DontCreateServices"
                 exclude="ServiceRegistry" />
```

Verletzungen:	Diese Regel wird nicht verletzt.

6.7 Zusammenfassung

Es ist gelungen, die vier zum Teil verschachtelten Stile des JCommSys zu beschreiben und ihre Umsetzung zu prüfen. Dabei konnte die berücksichtigt werden, dass die Elemente der Stile auf sehr unterschiedliche Weise im Code repräsentiert sind. Gegenüber der Architekturüberprüfung mit bestehenden Werkzeugen (vgl. Scharping 2005) konnte damit eine deutliche Verbesserung erreicht werden. Die Ergebnisse der Prüfung sind in der folgenden Tabelle 1 zusammengefasst.

Regeln:	Verletzungen:	Ergebnis der Prüfung:
9	17	erfolgreich geprüft
1	5	abgeschwächt geprüft (Verhalten von Fachwerten)
2	2	nicht geprüft (Verbot von Zyklen)
2	4	nicht geprüft (Zuordnungsregeln)

Tabelle 1: Ergebnisse der Prüfung des JCommSys

Mit 10 Regeln konnten die meisten Architekturregeln des JCommSys beschrieben und geprüft werden. Bei einer Regel musste eine Beschränkung durch das Prinzip einer statischen Codeanalyse in Kauf genommen werden. Regel 4.2 ist nur in abgeschwächter Form prüfbar, da sie sich auf das Verhalten von Methoden bezieht. Die Verletzungen der geprüften Regeln wurden mit dem Prototyp jedoch ausnahmslos entdeckt.

Eine Regel konnte nicht eindeutig einem Stil zugeordnet werde, da sie Elemente unterschiedlicher Stile miteinander in Beziehung setzt. Aufgrund der flexiblen Regelbeschreibung war es dennoch möglich, auch diese Regel zu prüfen.

Die Erkennung von zyklischen Strukturen wurde in dieser Arbeit nicht betrachtet. Die entsprechenden Regeln konnten daher nicht geprüft werden.

Im JCommSy gibt es zwei Zuordnungsregeln, die jeweils für ihren Stil fordern, dass jede Klasse einem Element zugeordnet sein muss. Der Prototyp ermöglicht zwar bisher keine direkte Prüfung dieser Regeln, allerdings werden nicht zugeordnete Klassen in der *Architekture View* aufgelistet und sind daher leicht erkennbar.

Kapitel 7 Überprüfung der WAM-Systeme

7.1 Architekturstil „WAM“

In dieser Arbeit wurden zwei WAM-Systeme auf Einhaltung der Regeln des WAM-Architekturstils untersucht. Bei den Beispielsystemen handelt es sich um den Pausenplaner und das EMS. Karstens hat diese beiden Systeme ebenfalls untersucht (Karstens 2005), sodass ihre Ergebnisse mit den hier gewonnen verglichen werden können.

7.1.1 Elemente

Der WAM-Architekturstil ist ein Stil auf der Klassenebene mit neun verschiedenen Elementarten (vgl. Abbildung 20):

- Komplexes Werkzeug (Tool)
- Einfaches Werkzeug (MonoTool)
- Funktionskomponente (FK)
- Interaktionskomponente (IAK)
- Oberflächenkomponente (GUI)
- Service
- Automat
- Material
- Fachwert (Domainvalue)

In den untersuchten Systemen können die WAM-Elemente weitgehend über verwendete Basisklassen und Interfaces aus dem JWAM-Rahmenwerk identifiziert werden. Materialien werden anhand der Basisklasse und des Packages identifiziert. Im Pausenplaner haben auch die Services keine gemeinsame Basisklasse und werden über das Package identifiziert. Oberflächenkomponenten können am leichtesten anhand des Namens identifiziert werden.

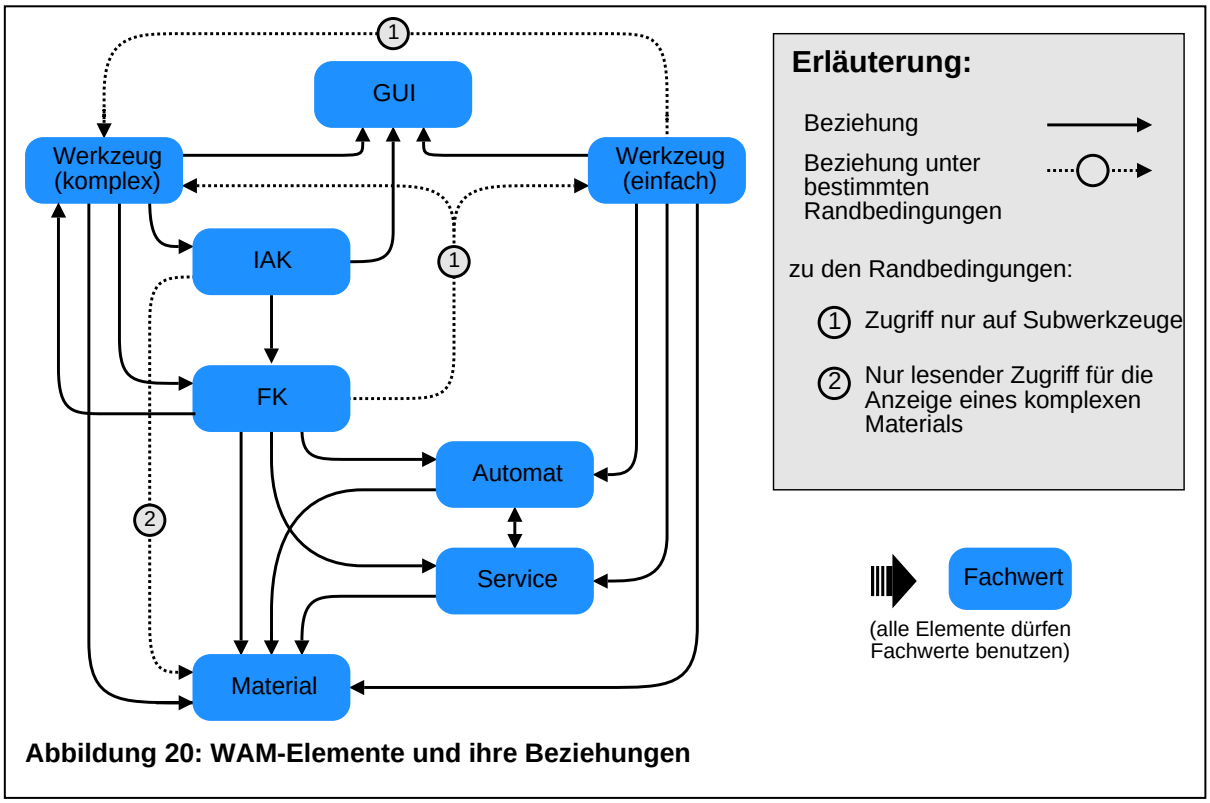

Abbildung 20: WAM-Elemente und ihre Beziehungen

```
<elements>
   <type name="Domainvalue">
      <inheritance-filter name="DomainValue"/>
   </type>
   <type name="Material">
      <package-filter pattern="de.itwps.ems.material..." />
   </type>
   <type name="GUI">
      <name-filter pattern="*GUI"/>
   </type>
   …
</elements>
```

Bei der Identifikation der WAM-Elemente werden Testklassen über einen Filter auf Stilebene ausgeschlossen.

7.1.2 Zuordnung von Werkzeugkomponenten

Bei den Werkzeugen wird zwischen einfachen Werkzeugen (MonoTools) und komplexen Werkzeugen (Tools) unterschieden. Beide haben eine repräsentierende Klasse und weitere Architekturelemente, die ihnen als Bestandteil zuzuordnen sind. Ein MonoTool sollte eine GUI, ein Tool sollte eine FK, eine IAK und eine GUI haben. Die Zuordnung dieser Komponenten wird von einer (Element-)Initialisierungsklasse durchgeführt.

```
<type name="Tool" className="ToolInitializer"> … </type>
<type name="MonoTool" className="ToolInitializer"> … </type>
```

In den untersuchten Systemen liegen die Komponenten eines Werkzeuges meist in einem Package und sind ähnlich benannt. Es gibt jedoch eine Vielzahl von Abweichungen. Eine automatische Zuordnung der Werkzeugkomponenten wird dadurch erschwert, dass einige Komponenten von mehreren Werkzeugen verwendet werden. Daher wurde eine Annotation für die Werkzeugklassen eingeführt, über die Werkzeugbestandteile explizit zugeordnet werden können.

```
@Tool(fp=DruckFK3.class, ip=DruckIAK3.class, gui=DruckGui3.class)
public class DruckTool3 extends ToolFpIpImpl {   … }
```

Die Initialisierungsklasse wertet die Annotationen aus und hinterlegt die Verknüpfungen im Architekturmodell als Eigenschaften der Werkzeugelemente. Obwohl die Annotation eingeführt wurde, erscheint die Identifizierung von Werkzeugen über die Basisklasse sinnvoll. Auf diese Weise können fehlende Annotationen erkannt werden.

7.1.3 Zuordnung von Subwerkzeugen

Werkzeuge können als Subwerkzeuge in andere Werkzeuge eingebettet werden. Dies wird im Architekturmodell über eine Eltern-Kind-Beziehung zwischen den Werkzeugelementen abgebildet. Dass ein Werkzeug Subwerkzeuge hat, ist im Code nur mit großem Aufwand zu erkennen. Daher wurde eine Annotation eingeführt, die zu einem Werkzeug die Subwerkzeuge angibt.

```
@ParentTool(subtools = { DeviceEditorTool.class, ValueDefinerTool.class })
public class WorkbenchTool extends AbstractToolMono {…}
```

Die im Beispiel verwendete Annotation `ParentTool` wird bei der Initialisierung des Werkzeuges `WorkbenchTool` ausgewertet. Dabei ermittelt die Initialisierungsklasse die Architekturelemente zu den Klassen `DeviceEditorTool` und `ValueDefinerTool` und ordnet diese dem `WorkbenchTool` als Subwerkzeuge zu.

7.1.4 Verbotsregeln

Der WAM-Ansatz beschreibt detailliert, welche Elemente miteinander in Beziehung stehen dürfen (siehe Abbildung 20). Die ersten drei Verbotsregeln verbieten Beziehungen zwischen WAM-Elementen. Die vierte Regel beschränkt Abhängigkeiten zu Elementen externer Bibliotheken.

Regel 1.1:	***Diese Regel fasst eine Reihe von Regeln mit dem gleichen Schema zu einer Regel zusammen:*** ***Tools kennen keine Services und Automaten.*** ***MonoTools kennen keine IAKs und Tools.*** ***FKs kennen keine IAKs und GUIs.*** ***IAKs kennen keine Tools, Automaten und Services.*** ***GUIs kennen keine Materialien, Automaten, Services, Tools, MonoTools, IAKs und FKs.*** ***Automaten kennen keine Tools, IAKs und FKs.*** ***Services kennen keine Tools, IAKs und FKs.*** ***Materialien kennen keine Tools, IAKs, FKs, Services und Automaten.*** ***Fachwerte kennen keine Tools, IAKs, FKs, Services, Automaten und Materialien.***

Prüfung:	Die Prüfung erfolgt über die `SimpleClasslevelDepRule.` Wie die `WhitelistRule` auf der Subsystemebene basiert diese Regel darauf, dass in einer Parameterliste alle Paare von Elementarten aufgezählt werden, zwischen denen eine Abhängigkeit erlaubt ist.
	`<dependency-rule className="SimpleClasslevelDepRule">`
	`  <dependency from="Tool" to="FK"/>`
	`  <dependency from="Tool" to="IAK"/>`
	`  …`
	`</dependency-rule>`
	Regeln, die zwar ein vergleichbares Schema haben (Regel 1.2 und 1.3), aufgrund von Ausnahmen oder Randbedingungen aber eine genauere Betrachtung erfordern, werden nicht mit der `SimpleClasslevelDepRule` geprüft. Die entsprechenden Abhängigkeiten werden vielmehr von dieser explizit erlaubt und dann von anderen Regeln eingeschränkt.

Ausnahmen:	Im EMS gibt es zwei Ausnahmen. Die entsprechenden Klassenpaare werden der Regel als Parameter übergeben.
	`<except from="DeviceCheckerAdjustorTool"`
	`  to="DeviceCheckerAutomaton"/>`
	`<except from="DeviceCheckerResultViewerTool"`
	` to="DeviceCheckerAutomaton"/>

Verletzungen:	Unter Berücksichtigung der Ausnahmen wird diese Regel im EMS nicht verletzt. Im Pausenplaner gibt es neun Klassenpaare, mit insgesamt 60 Beziehungen, die gegen diese Regeln verstoßen. Alle diese Architekturverletzungen wurden entdeckt.

Regel 1.2:	***IAKs kennen keine Materialien.***
	Für diese Regel gilt die Einschränkung, dass eine IAK für die Anzeige komplexer Materialien lesenden Zugriff auf das eigene Material erhalten kann.
Prüfung:	Die Regel wird über die Klasse IpReadsMaterial geprüft.
	`<dependency-rule from="IP" to="Material" className="IpReadsMaterial"/>`
	Diese Klasse verbietet grundsätzlich alle Abhängigkeiten von IAKs zu Materialien. Einzelne Paare von IAK und Material können als Ausnahmen angegeben werden.
	Da mit einer statischen Codeanalyse nicht ermittelt werden kann, ob eine Methode den fachlichen Zustand eines Objektes verändert, kann die Regel nicht prüfen, ob die IAK nur lesend auf das Material zugreift. Die Regel könnte allerdings den Code nach Anzeichen für einen schreibenden Zugriff durchsuchen. Bisher ist dies jedoch nicht implementiert.
Verletzungen:	Im EMS wird diese Regel von der IAK `DeviceCheckerResultViewerIP` verletzt. Der Zugriff auf Konstanten der Klasse `DeviceCheckerResult` wird von der Regel erkannt, der Zugriff auf die innere Klasse `DeviceChekerResult.Entry` jedoch nicht. Das liegt daran, dass die innere Klasse in der bisherigen Umsetzung nicht als Teil des Elementes angesehen wird. Im Pausenplaner gibt es 13 Klassenpaare, mit insgesamt 290 Beziehungen, die gegen diese Regeln verstoßen. Alle diese Architekturverletzungen wurden entdeckt.

Regel 1.3:	***FKs und MonoTools kennen keine Tools.***
	Für diese Regel gilt die Einschränkung, dass bei der Konstruktion zusammengesetzter Werkzeuge eine FK bzw. eine MonoTool-Klasse Zugriff auf die Werkzeugklasse eines Subwerkzeuges erhält.
Prüfung:	Diese Regel wird über die Klasse `DependencyOnlyToSubtools` geprüft. Sie verbietet Abhängigkeiten von FKs und MonoTools zu Tools. Ausgenommen sind Abhängigkeiten zu Subtools.
	`<dependency-rule from="FP" to="Tool" className="DependencyOnlyToSubtools"/>` `<dependency-rule from="MonoTool" to="Tool" className="DependencyOnlyToSubtools"/>`
Verletzungen:	Im EMS gibt es eine FK und drei MonoTools mit einer Beziehung zu Werkzeugklassen. In allen Fällen handelt es sich um Abhängigkeiten zu Subtools. Da dies in Annotationen angegeben ist, werden die Beziehungen richtigerweise nicht als Architekturverletzung gemeldet. Im Pausenplaner gibt es eine FK mit einer Beziehung zu einer Werkzeugklasse. Auch dabei handelt es sich um ein Subtool und nicht um eine Architekturverletzung. Gegenüber der Prüfung mit dem Sotographen konnte hier durch die Zuordnung von Subtools eine Verbesserung erzielt werden.

Regel 1.4:	***Fachwerte kennen keine Events und keine Requests.*** ***Materialien kennen keine Events und keine Requests.***

Prüfung:	Die Regel `NoDependencyTo` betrachtet alle ausgehenden Abhängigkeiten der Elemente einer Art und stellt sicher, dass keine Abhängigkeiten zu angegebenen Klassen oder Packages bestehen. Um diese Packages oder Klassen anzugeben, können `Package-` oder `Name-Filter` verwendet werden.
	Hier wird die Regel verwendet, um Abhängigkeiten von Fachwerten zum Package `java.awt.event` (sowie allen Subpackes) und zu den JWAM-Klassen `Event` und `Request` zu verbieten.
	`<element-rule elementType="Domainvalue" className="NoDependencyTo">`
	`<package-filter pattern="java.awt.event..."/>`
	`<name-filter pattern="de.jwam.*.observation.Event"/>`
	`<name-filter pattern="de.jwam.*.request.Request"/>`
	`</element-rule>`
	(Die Beschreibung der Regel für Materialien erfolgt analog.)
	Da Events und Requests keine WAM-Elemente sind, kann eine Abhängigkeit zu diesen Klassen nicht auf eine Abhängigkeit zwischen Architekturelementen abgebildet werden. Daher können solche Abhängigkeiten nicht als `dependency-rule` geprüft werden. Stattdessen ist die Regel `NoDependencyTo` als eine `element-rule` implementiert, die alle ausgehenden Abhängigkeiten auf der Codeebene prüft.
	Um diese Regel zu prüfen, müssen die Abhängigkeiten zu externen Klassen in das Systemmodell aufgenommen werden. Dies ist zwar standardmäßig nicht der Fall, kann aber aktiviert werden. Karstens hat diese Regel nicht geprüft, da für eine Analyse mit dem Sotographen nicht nur die Abhängigkeiten zu den Bibliotheken, sondern die gesamten Bibliotheken in die Analyse einbezogen werden müssten. Dies würde einen hohen Ressourcenbedarf verursachen.

Verletzungen:	Diese Regel wird im EMS und im Pausenplaner nicht verletzt.

7.1.5 Gebotsregeln

Die folgenden Gebotsregeln gelten nicht für abstrakte Klassen oder Interfaces. Da die Regelklassen dies berücksichtigen, konnten einige unechte Funde des Sotographen vermieden werden.

Regel 2.1:	***Tools kennen und erzeugen ihre FK, IAK und GUI.*** ***IAKs kennen ihre GUI und ihre FK.*** ***MonoTools kennen und erzeugen ihre GUI.***

Prüfung:	Eine Prüfung erfolgt über die Klassen `ToolConstruction` und `MonoToolConstruction`. Diese Klassen betrachten die zugeordneten Komponenten und die geforderten Beziehungen zwischen ihnen.
	`<element-rule elementType="Tool" className="ToolConstruction"/>`
	`<element-rule elementType="MonoTool" className="MonoToolConstruction"/>`
	IAKs sind Bestandteile von Tools. Die Regel für IAKs wird daher zusammen mit der Regel für Tools von der Klasse `ToolConstruction` geprüft.

Verletzungen:	Diese Regeln werden im EMS von elf Monotools verletzt. Im Pausenplaner gibt es keine Verletzungen dieser Regeln. Bei der Untersuchung mit dem Sotographen waren Funde gemeldet worden, da Werkzeugkomponenten, aufgrund von fehlenden Namenskonventionen, nicht richtig zugeordnet wurden. Da hier die Komponenten über Annotationen zugeordnet werden, tritt dieses Problem nicht auf. Es sollte allerdings in Erwägung gezogen werden, die Einhaltung der Namenskonventionen innerhalb der Regelklassen zu prüfen.

Regel 2.2:	***MonoTools, FKs, Automaten und Services gehen mit Materialien um.***

Prüfung:	Diese Regel wird über die Klasse `HasDepToElement` geprüft. Diese betrachtet, alle ausgehenden Abhängigkeiten von Elementen einer bestimmten Art und prüft, ob eine Abhängigkeit zu einem Element einer anderen gegeben Elementart existiert.
	`<element-rule elementType="MonoTool" className="HasDepToElement" toElementType="Material">`
	`  <except name="ToolboxTool"/>`
	`</element-rule>`

Verletzungen:	Im EMS und im Pausenplaner gibt es keine Verletzungen dieser Regel. Durch die Angabe von Ausnahmen und die Beschränkung auf konkrete Klassen konnten fälschliche Meldungen von Architekturverletzungen (unechte Funde) vermieden werden.

7.1.6 Elementregeln

Regel 3.1:	***Tools bestehen aus FK, IAK und GUI.***
Prüfung:	Diese Regel wird zusammen mit der Regel 2.1 über die Klasse `ToolConstruction` geprüft. Die Klasse prüft auch, ob alle Komponenten (FK, IAK und GUI) zugeordnet wurden.
Verletzungen:	Im EMS und im Pausenplaner wird diese Regel nicht verletzt.

Regel 3.2:	***Fachwerte haben keinen öffentlichen Konstruktor.***
Prüfung:	Die Regel wird durch die Klasse NoPublicConstructor geprüft. `<element-rule elementType="Domainvalue" className="NoPublicConstructor"/>`
Verletzungen:	Im EMS und im Pausenplaner wird diese Regel nicht verletzt.

Regel 3.3:	***Fachwert-Klassen bieten keine Operationen an, die es erlauben, das Fachwert-Objekt zu verändern.*** Da mit einer statischen Codeanalyse nicht zuverlässig festgestellt werden kann, ob eine Methode den fachlichen Zustand eines Objektes verändert, wird diese Regel in einer entschärften Variante geprüft: ***Fachwerte haben keine Setter.***
Prüfung:	Die Regel `NoSetter` verbietet Setter an Elementen und wird auf alle Fachwerte angewandt. Ob eine Methode von der Regel als Setter behandelt wird, hängt dabei ausschließlich von der Signatur ab. Es besteht zwar die Möglichkeit, die Methodenrümpfe zu analysieren. Aufgrund des damit verbundenen Aufwandes wurde darauf aber verzichtet. `<element-rule elementType="Domainvalue" className="NoSetter"/>`
Verletzungen:	Im EMS wird diese Regel nicht verletzt. Im Pausenplaner wird diese Regel von fünf Fachwerten verletzt.

Regel 3.4:	***FKs geben keine Materialien zurück.***
Prüfung:	Diese Regel wird von der Klasse `DontReturnMaterial` geprüft. `<element-rule elementType="FP" className="DontReturnMaterial"/>`
Verletzungen:	Im EMS gibt es eine Verletzung der Regel, die auch erkannt wurde. Im Pausenplaner wird diese Regel von fünf Funktionskomponenten verletzt.

Regel 3.5:	***Materialien haben nicht nur Getter und Setter.***
Prüfung:	Die Regel wird von der Klasse `NotOnlyGetterAndSetter` geprüft.
	`<element-rule elementType="Material" className="NotOnlyGetterAndSetter"/>`
	Wie bei Regel 3.3 erfolgt die Erkennung von Settern und Gettern ausschließlich aufgrund einer Betrachtung der Signatur der Methoden.
Verletzungen:	Im EMS wird diese Regel von fünf Materialien verletzt. Alle Verletzungen wurden erkannt.
	Im Pausenplaner wird diese Regel nicht verletzt.

7.2 Zusammenfassung

Es ist gelungen, in einer Architekturbeschreibung für zwei WAM-Systeme die für eine Prüfung wesentlichen Elemente des Stils und seiner Ausprägungen zu beschreiben. Für die Identifikation der WAM-Elemente wurden die Basisklassen aus dem JWAM-Rahmenwerk verwendet. Es ist jedoch davon auszugehen, dass das Konzept der Filter eine Prüfung auch von Systemen ermöglicht, die das Rahmenwerk nicht verwenden.

Regeln:[7]	**Verletzungen:**		**Ergebnis der Prüfung:**
	EMS	PP	
43	18	78	erfolgreich geprüft, eine Verletzung unentdeckt
1	0	5	abgeschwächt geprüft (Verhalten von Fachwerten)
1	0	3	nicht geprüft (Verbot von Zyklen)
4	0	0	zusätzlich geprüft (Abhängigkeit zu externen Bibliotheken)

Tabelle 2: Ergebnisse der Prüfung der WAM-Systeme

In dieser Arbeit ist gelungen, die WAM-Regeln in dem angestrebten Umfang zu prüfen. Dabei musste Regel 3.3 (wie bei der Prüfung mit dem Sotographen) abgeschwächt werden, da sie sich auf das fachliche Verhalten von Methoden bezieht.

Vier zusätzliche Regeln konnten berücksichtigt werden, die sich auf Abhängigkeiten zu großen externen Bibliotheken beziehen und deren Prüfung mit dem Sotographen daher zu aufwendig gewesen wäre. Dies ist möglich, da die Abhängigkeiten zu diesen Bibliotheken in das Systemmodell aufgenommen werden können, ohne die gesamte Bibliothek analysieren zu müssen.

Die Erkennung von zyklischen Strukturen wurde in dieser Arbeit nicht betrachtet. Die entsprechenden Regeln konnten daher nicht geprüft werden.

Bei der Überprüfung des EMS ist eine Architekturverletzung unentdeckt geblieben. Eine Abhängigkeit zu einem Material wurde nicht als erkannt, da auf eine innere Klasse zugegriffen wurde und diese bisher nicht als Teil des Materials behandelt wird. Um diesen Fehler zu beheben, müssten die Enthaltenseins-Beziehungen zwischen innerer und äußerer Klasse in das Systemmodell aufgenommen werden. Alle weiteren Verletzungen der geprüften Regeln wurden mit dem Werkzeug entdeckt.

[7] Die angegebenen Zahlen beziehen sich auf die Regel von Karstens. In meiner Arbeit wurden teilweise mehrere Regeln zu einer zusammengefasst.

Über die Zuordnung von Werkzeugkomponenten und Sub-Werkzeugen wurde die Voraussetzung geschaffen, um verschiedene Randbedingungen von Regeln bei der Prüfung berücksichtigen zu können. Damit konnte eine Vielzahl unechter Funde vermieden werden. Für die Zuordnung wurden bei der Untersuchung der Beispielsysteme Annotationen verwendet. Ohne Annotationen wäre sie allerdings sehr viel schwieriger.

Alle Ausnahmen von Regeln konnten angemessen berücksichtigt werden, sodass auch unechte Funde, die auf Ausnahmen zurückzuführen sind, vermieden werden konnten.

Kapitel 8 Fazit und Ausblick

8.1 Fazit

Ziel dieser Arbeit war, ein Verfahren zu entwickeln, mit dem geprüft werden kann, ob die Ist-Architektur eines Systems den Vorgaben der entworfenen Soll-Architektur entspricht. Bei dem dazu gewählten Ansatz erfolgt die Prüfung der Architekturregeln auf der Grundlage einer Architekturbeschreibung, die den Quellcode um Informationen zu den Architekturstilen ergänzt. Eine wichtige Zielsetzung war, die Regeln unmittelbar bei der Bearbeitung des Quellcodes zu prüfen, sodass Entwickler eine direkte Rückmeldung zur Umsetzung der Architekturstile erhalten. Um dieses Ziel zu erreichen, wurde die Architekturüberprüfung in die Entwicklungsumgebung integriert.

Die in der Arbeit entwickelten Konzepte wurden prototypisch in einem Softwarewerkzeug umgesetzt und an drei Beispielsystemen erprobt. Dabei hat sich gezeigt, dass die entwickelten Konzepte geeignet sind, die Architekturstile des JCommSys und die Umsetzung der WAM-Modellarchitektur im Pausenplaner-System und im EMS zu überprüfen. Ein Vergleich mit den Ergebnissen früherer Architekturüberprüfungen hat bestätigt, dass die in den Systemen vorhandenen Architekturverletzungen tatsächlich entdeckt wurden.

8.1.1 Auswertung

Die mit der Prüfung verbundene Problemstellung wurde eingangs mit vier Fragen umrissen. Anhand dieser Fragen wird im Folgenden diskutiert, wie gut die Konzepte bei den einzelnen Problemstellungen greifen.

- **Wie können Architekturregeln beschrieben werden?**

  Im Verlaufe dieser Diplomarbeit wurden unterschiedliche Arten der Beschreibung von Architekturregeln betrachtet. Schließlich wurden Java-Klassen einsetzt, um die Regeln zu formulieren. Dies hat sich bei der Untersuchung der Beispielsysteme bewährt. Die Sprache bietet die benötigte Ausdrucksmächtigkeit und ist gleichzeitig den potenziellen Benutzern des Werkzeuges vertraut.

  Die Architekturbeschreibung eines Systems bestimmt, ob alle relevanten Aspekte der Architektur im Architekturmodell repräsentiert sind. Die Regelklassen setzen auf diesem Modell auf. Wie leicht Regeln formuliert bzw. geprüft werden können, hängt davon ab, ob die Architektur des untersuchten Systems in der Architekturbeschreibung adäquat modelliert werden konnte. Falls das zugrunde liegende Schema für Architekturstile nicht ausreicht, um die Architektur des Systems zu modellieren, kommt ein besonderer Vorteil der Verwendung von Java für die Regelbeschreibung zum Tragen. Die Regelklassen können auf die verschiedenen Repräsentationen des Quellcodes (Systemmodell, Eclipse Java Model, AST, Dateiebene) ausweichen. So wird bei der Beschreibung der Regeln ein hohes Maß an Flexibilität erreicht.

  In Java werden die Regeln algorithmisch beschrieben. Für die Lesbarkeit und Verständlichkeit ist dies eher als Nachteil zu werten, da die eigentliche Bedeutung einer Regel immer mit Implementierungsdetails vermengt ist. Wie stark dieser Effekt ist, hängt von der Schnittstelle (API) ab, die den Regeln zur Verfügung steht, um das Architekturmodell zu analysieren.

  In der XML-Datei der Architekturbeschreibung wird deklarativ beschrieben, für welche Elemente eine Regelklasse gilt. Dieser deklarative Anteil kann über die Parametrisierung von Regeln erhöht werden. Damit ergibt sich ein Gestaltungsspielraum, der hier nur ansatzweise ausgereizt werden konnte. Über die Parametrisierung ist es

möglich, generische Regeln wie die `SimpleXlevelDepRule` zu entwickeln und diese in Bibliotheken zur Verfügung zustellen.

- **Wie werden Architekturelemente bestimmt und mit dem Code in Beziehung gesetzt?**

  In dieser Arbeit wurden zwei Verfahren angewandt, um Architekturelemente mit den Elementen des Codes in Beziehung zu setzen. Auf der Klassenebene werden die Elemente im Code identifiziert, auf der Subsystemebene werden die Elemente aufgezählt. Eine Unterstützung der beiden Verfahren auf beiden Ebenen war vor dem Hintergrund der untersuchten Architekturstile nicht notwendig und aufgrund des hohen Implementationsaufwandes wurde darauf verzichtet.

  Sowohl bei der Identifikation als auch bei der Aufzählung der Elemente (bzw. bei der folgenden Zuordnung von Klassen) sind Mengen von Klassen anhand bestimmter Kriterien auszuwählen. Mit dem Konzept der Filter wurde dafür eine Möglichkeit gefunden, die den spezifischen Eigenarten von Softwareprojekten Rechnung trägt und sich bei der Prüfung der Beispielsysteme bewährt hat. Damit wurde gegenüber anderen Werkzeugen zur Architekturüberprüfung bereits eine deutliches Maß an Flexibilität gewonnen.

- **Wie ist bei der Prüfung zu verfahren? Wie ist mit Regelverletzungen umzugehen?**

  Das hier entwickelte Verfahren zur Architekturüberprüfung basiert wie der Sotograph auf einer statischen Codeanalyse. Die Prüfbarkeit von Architekturregeln, die das Verhalten oder die Semantik von Methoden betreffen, ist damit gleichermaßen eingeschränkt.

  Diese Arbeit hat gezeigt, dass es möglich ist, die Architekturüberprüfung in den Übersetzungsprozess der Entwicklungsumgebung einzubinden. Damit ist es gelungen, die Architekturregeln unmittelbar bei der Bearbeitung des Quellcodes zu prüfen und auftretende Verletzungen direkt im Quellcode anzuzeigen. Sobald für ein System eine Architekturbeschreibung vorhanden ist, kann die Prüfung von jedem Entwickler durchgeführt werden. Architekturverletzungen werden so angezeigt, dass ihre Ursache leicht erkennbar ist. In den Regelklassen kann man beeinflussen, wie ausführlich die Erläuterungen sind, die zu den Architekturverletzungen gegeben werden. Auf diese Weise kann die Architekturüberprüfung auch von unerfahrenen Entwicklern genutzt werden. Lediglich die Entwicklung der Architekturbeschreibung und der Regeln erfordert einen höheren Einarbeitungsaufwand. Es ist jedoch davon auszugehen, dass diese Tätigkeit in den meisten Projekten nur von wenigen Personen wahrgenommen wird.

  Für Benutzer unterscheidet sich die Architekturüberprüfung mit diesem Werkzeug deutlich vom Sotographen. Eine automatisch angestoßene Überprüfung kann mit dem Sotographen beispielsweise über die Kopplung mit einem Integrationsserver realisiert werden. Damit erfolgt die Rückmeldung allerdings deutlich später, als bei dem hier vorgestellten Ansatz. Da die Ergebnisse der Regelprüfung mit dem Sotographen interpretiert werden müssen (Karstens 2005), erfordert der Umgang damit viel Erfahrung. (Mit dem Sotoarc Developer existiert zwar seit Kurzem ein Werkzeug, dass eine automatische Architekturüberprüfung mit direkter Rückmeldung ermöglicht, aufgrund des festen Metamodells und der fehlenden Datenbank können damit die WAM-Regeln jedoch nicht geprüft werden.)

  Bei der Entwicklung des Werkzeuges konnte die umfangreiche Infrastruktur genutzt werden, die Eclipse zur Verfügung stellt. Die verwendeten Marker scheinen ein ge-

eignetes Mittel zu sein, um Architekturverletzungen anzuzeigen. Sie bieten einerseits eine genaue Lokalisierung der Verletzung, andererseits können die Architekturverletzungen eines Systems in der Problems-View in einer Listendarstellung überblickt und nach Kriterien gefiltert werden.

Im Gegensatz zum Sotographen analysiert der vorgestellte Prototyp den Quellcode eines Systems. Dies ist vorteilhaft, da einige Abhängigkeiten im Bytecode nicht mehr nachvollzogen werden können. Der Compiler entfernt beispielsweise Abhängigkeiten zu Konstanten. Damit sind durch diese Abhängigkeiten verursachte Architekturverletzungen im Sotographen nicht zu entdecken.

- **Wie ist mit Ausnahmen umzugehen?**

  Viele Ausnahmen können vermieden werden, indem Randbedingungen innerhalb der Regeln genau beschrieben werden. Ist dies nicht möglich, können die einzelnen Elemente oder Abhängigkeiten als Ausnahmen berücksichtigt werden. Dies kann innerhalb der Regelklasse, über die Parametrisierung oder über Annotationen erfolgen.

Im Rahmen dieser Arbeit wurden Konzept und Werkzeug in Hinblick auf eine Prüfung der Regeln des JCommSys und der WAM-Modellarchitektur entwickelt. Dabei wurde versucht, von diesen konkreten Regeln zu abstrahieren und Lösungen zu entwickeln, die auch auf andere Architekturen anwendbar sind. Um zu beurteilen, ob dies gelungen ist, müssten weitere Systeme geprüft werden.

Ziel dieser Diplomarbeit ist letztendlich ein Werkzeug, das kontinuierlich, den Entwicklungsprozess begleitend, eingesetzt werden kann. Der entwickelte Prototyp wurde bisher nur genutzt, um Momentaufnahmen der Beispielsysteme zu analysieren. Das Verhalten während der Entwicklung konnte nur exemplarisch getestet werden. Für eine abschließende Bewertung der vorgestellten Konzepte ist es daher notwendig, die Konzepte im Entwicklungsalltag zu erproben. Dabei sollte untersucht werden, welche Rollenverteilung im Umgang mit dem Werkzeug sinnvoll ist und wie das Werkzeug bei einer Weiterentwicklung der Soll-Architektur (zum Beispiel bei Refactorings) eingesetzt werden kann.

Das vorgestellte Verfahren sollte hinsichtlich der Skalierbarkeit weiter untersucht werden. So hält der Prototyp das Architekturmodell eines untersuchten Systems ständig im Speicher vor, um eine effiziente Prüfung zu ermöglichen. Bei den Beispielsystemen war dies zwar unproblematisch, gegebenenfalls sollten jedoch Alternativen, wie eine leichtgewichtige Datenbanklösung, in Erwägung gezogen werden.

8.1.2 Vergleich mit der Regelprüfung im Software-Cockpit

In dieser Arbeit wurde ein Verfahren vorgestellt, mit dem Architekturregeln beschrieben und ihre Einhaltung in einem System geprüft werden können. Es stellt damit eine Alternative zu der von Karstens entwickelten Regelprüfung mit dem Sotographen dar. Auf die Gemeinsamkeiten und Unterschiede wurde bereits an verschiedenen Stellen eingegangen.

Im Folgenden wird ein kurzer Vergleich des hier vorgestellten Ansatzes mit der von Ionescu entwickelten Prüfung im Software-Cockpit (vgl. Ionescu 2007 und Abschnitt 2.2.8) vorgenommen. Eine umfassendere Gegenüberstellung der beiden Ansätze war aufgrund der zeitlichen Überschneidung der beiden Arbeiten nicht möglich.

Beide Ansätze umfassen eine Beschreibungssprache für Architekturstile. Ionescu verwendet dazu das im Software-Cockpit umgesetzte Konzept der Metamodellierung, dass der Beschreibung von Architekturstilen und Modellierung kaum Grenzen setzt. So unterstützt die Modellierungssprache die Beschreibung von Vererbungsbeziehungen zwischen Elementarten und die explizite Modellierung von Beziehungsarten. Beides ist in dieser Form in der hier vorgestellten Architekturbeschreibung nicht möglich. Diese orientiert sich stattdessen stärker

an den Bedürfnissen der untersuchten Architekturstile und soll genutzt werden können, um diese und vergleichbare Stile intuitiv zu beschreiben. In diesem Zusammenhang ist auch die Unterscheidung der Klassen- und der Subsystemebene zu sehen, die in dem Ansatz von Ionescu nicht enthalten ist.

Mit der Metamodellierung ist eine scharfe Trennung des Architekturstils (Metamodell-Ebene) von seiner Ausprägung (Modell-Ebene) verbunden. Der Stil definiert die Architekturelement- und Beziehungsarten, die Architektur die konkreten Elemente und Beziehungen. Hier werden Stil und Ausprägung aus pragmatischen Gründen gemeinsam beschrieben.

Die Regelbeschreibung erfolgt hier algorithmisch in Java. Ionescu verwendet eine eigene prädikatenlogische Sprache. Die Vor- und Nachteile dieser Ansätze wurden im Verlauf meiner Arbeit abgewogen.

Schließlich verfolgen beide Ansätze unterschiedliche Strategien zur Einbindung in den Entwicklungsprozess. Die Einbindung in das Software-Cockpit bettet die Architekturüberprüfung in eine regelmäßig durchgeführte Qualitätsprüfung ein. Das hier verfolgte Ziel, die Regeln bereits bei der Bearbeitung des Quellcodes zu prüfen, wird damit nicht erreicht.

8.2 Ausblick

Im Verlaufe dieser Arbeit musste an verschiedenen Stellen festgestellt werden, dass eine abschließende Behandlung in dem gegebenen Rahmen einer Diplomarbeit nicht möglich ist. Im Folgenden wird ein Überblick über die Bereiche gegeben, in denen eine Weiterentwicklung der Konzepte oder des Prototypes sinnvoll erscheint.

8.2.1 Architekturbeschreibung

Die entwickelte Architekturbeschreibung ergänzt den Quellcode eines Systems um Informationen zu dessen Architektur. Sie erlaubt, die umgesetzten Architekturstile direkt zu beschreiben. Der Sotograph stellt eine vergleichbare Beschreibungssprache nur für Ausprägungen von Schichten- und Grapharchitekturen zur Verfügung (Bischofberger et al. 2004). Andere Architekturstile müssen aufwendig über Datenbankanfragen abgebildet werden. Somit schafft die hier entwickelte Architekturbeschreibung einen Mehrwert für die Dokumentation des Systems. Das Architekturmodell, das hier auf der Grundlage der Beschreibung generiert wird, kann nicht nur für eine Architekturüberprüfung genutzt werden. Vielmehr bietet es die Möglichkeit innerhalb der Entwicklungsumgebung, im Sinne einer architekturzentrierten Softwareentwicklung, einen neuen Zugang zur Architektur des Systems zu schaffen. Die Architecture View ist ein erster Schritt in diese Richtung.

In der bisherigen Umsetzung können Elemente auf der Subsystemebene aufgezählt und auf der Klassenebene identifiziert werden. Im Sinne einer Vereinheitlichung oder zur Untersuchung bisher nicht betrachteter Architekturstile, könnte es sinnvoll sein, das Werkzeug entsprechend zu erweitern.

Zusammengesetzte Architekturelemente auf der Klassenebene, wie sie bei der Werkzeugkonstruktion (WAM) auftreten, konnten nicht in ihrer gesamten Vielfalt betrachtet werden. Mithilfe der Initialisierungsklassen war es zwar auch möglich, die Enthaltenseins-Beziehungen im Architekturmodell abzubilden und die darauf beruhenden Regeln zu prüfen, wünschenswert wäre jedoch, diese Beziehungen in der XML-Datei zu beschreiben. Bisher sind sie dort nicht sichtbar.

Die Menge der Filter könnte noch ergänzt und die Konfigurierbarkeit der bestehenden Filter verbessert werden. So können Attribute von Annotationen bisher nicht berücksichtigt werden. Um möglichst große Flexibilität zu erreichen, sollten Benutzer des Werkzeuges eigene Filterklassen wie die Regelklassen erstellen können.

XML wurde für die Architekturbeschreibung verwendet, da es ein leicht verständliches Format ist, das gleichzeitig leicht maschinell interpretiert und auf syntaktische Korrektheit geprüft werden kann. Die Lesbarkeit der XML-Dateien nimmt mit steigendem Umfang jedoch schnell ab, sodass bereits bei der Untersuchung der Beispielsysteme ein Editor hilfreich gewesen wäre. Ein Editor für die Architekturbeschreibung könnte ihre Erstellung und Pflege deutlich erleichtern.

8.2.2 Architekturregeln

In der bisherigen Umsetzung können keine Zuordnungsregeln geprüft werden. Eine entsprechende Unterstützung sollte implementiert werden.

Über die Parametrisierung der Regeln wird ein Teil ihrer Semantik von der algorithmischen Beschreibung in Java in eine deklarative Beschreibung in der XML-Datei verlagert. Durch den deklarativen Charakter wird dabei die Verständlichkeit erhöht. Es ist daher erstrebenswert eine Regelklasse zu entwickeln, die den Ansatz der prädikatenlogischen Ausdrücke (vgl. Absatz 4.2.2) aufgreift und solche Ausdrücke als Parameter entgegennimmt und auswertet.

In der bisherigen Umsetzung werden Ausnahmen ausschließlich als individuelle Bestandteile einzelner Architekturegeln betrachtet. Um dennoch eine einheitliche Behandlung von Ausnahmen zu erreichen, sollten Konventionen für ihre Beschreibung etabliert werden. Letztendlich wäre es wünschenswert, eine standardisierte Ausnahmebehandlung in das Werkzeug zu integrieren, die in Einzelfällen von Regeln überschrieben oder ergänzt werden kann.

Es sollte ein Verfahren entwickelt werden, um die Regelklassen zu testen. Dies ist schwierig, da die Klassen auf der Eclipse-Umgebung aufsetzen, und diese somit in Unit-Tests zur Verfügung stehen muss. Ein weiteres Problem besteht darin, dass für den Test einer Regel zum Teil umfangreiche Testkonstellationen aufgebaut werden müssen, um die vielen Varianten (z. B. durch Innere Klassen und Vererbung) zu berücksichtigen. Um Fehler in Regelklassen zu finden, wäre es weiter hilfreich, Regeln einfacher debuggen zu können. Dies ist bisher nur mit hohem Aufwand möglich.

8.2.3 Prüfung der Regeln

Bei Klassen, mit regelwidrigen Abhängigkeiten zu anderen Klassen, kann die Anzeige der Architekturverletzungen auf der Ebene von Statements, zu einer Vielzahl einzelner Verletzungen führen. Hier sollte die Option geschaffen werden, gleichartige Verletzungen innerhalb einer Klasse zusammenzufassen.

Das entwickelte Werkzeug sollte um eine optionale Analyse von Bytecode erweitert werden. Damit könnten auch Bibliotheken und Systembestandteile in die Analyse einbezogen werden, die nicht im Quellcode vorliegen. Da der genutzte Eclipse-Parser auch für Bytecode geeignet ist, wäre eine entsprechende Erweiterung mit mäßigem Aufwand möglich.

Hinsichtlich der Skalierbarkeit gibt es Optimierungspotenzial im Bereich der inkrementellen Regelprüfung. Dabei wird versucht, nur die Teile des Architekturmodells zu prüfen, die von einer Änderung betroffen sind. Aufgrund der umfassenden Analysemöglichkeiten innerhalb der Regelbeschreibung ist schwierig zu ermitteln, in welchen Elementen durch eine Codeänderung Regelverletzungen entstehen können. In der bisherigen Umsetzung werden daher mehr Elemente als nötig geprüft. Hier sollten weitere Mechanismen entwickelt werden, um unnötige Prüfungen zu vermindern. So sollte erwogen werden, für einzelne Regeln anzugeben, bei welchen Änderungen eine Prüfung nötig ist.

Literatur

(Aldrich et al. 2002) Aldrich, J., Chambers, C. und Notkin, D. *ArchJava: connecting software architecture to implementation*. Proceedings of the 24th International Conference on Software Engineering. Orlando, Florida, ACM Press. 2002.

(Bass et al. 2003) Bass, L., Clements, P. und Kazman, R. *Software Architecture in Practices*, Addison-Wesley Longman Publishing Co., Inc. 2003.

(Bäumer 1998) Bäumer, D. *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme* (Dissertationsschrift). Fachbereich Informatik, Universität Hamburg. 1998.

(Becker-Pechau und Bennicke 2007) Becker-Pechau, P. und Bennicke, M. *Concepts of Modelling Architectural Module Views for Consistency Checks*. 2007.

(Becker-Pechau et al. 2006) Becker-Pechau, P., Karstens, B. und Lilienthal, C. *Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln*. SE 2006, Bonn, Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI). 2006.

(Bennicke und Richter 2007) Bennicke, M. und Richter, J.-P. *Architecture of a Generic Software Control Center*. Workshop on Measurement-based Cockpits for Distributed Software and Systems Engineering Projects, München, Deutschland. 2007.

(Bischofberger et al. 2004) Bischofberger, W. R., Kühl, J. und Löffler, S. *Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking.* Software Architecture, First European Workshop, EWSA 2004, Proceedings, St Andrews, UK, Springer. 2004.

(Christl et al. 2005) Christl, A., Koschke, R. und Storey, M.-A. *Equipping the Reflexion Method with Automated Clustering*. Proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society. 2005.

(Fowler 2002) Fowler, M. *Patterns of Enterprise Applications Architecture*, Addison Weasley. 2002.

(Gamma et al. 1995) Gamma, E., Helm, R., Johnson, R. und Vlissides, J. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc. 1995.

(hello2morrow 2005) hello2morrow. *SonarJ White Paper*. 2005.

(Ionescu 2007) Ionescu, I. *Architekturkonsistenzprüfungen auf Basis von Modellarchitekturen* (Diplomarbeit). Institut für Informatik, Brandenburgische Technische Universität Cottbus. 2007.

(Karstens 2005) Karstens, B. *Regeln der WAM-Modellarchitektur* (Diplomarbeit). Fachbereich Informatik, Universität Hamburg. 2005.

(Knodel und Popescu 2007) Knodel, J. und Popescu, D. *A Comparison of Static Architecture Compliance Checking Approaches*. Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society. 2007.

(Koschke und Simon 2003) Koschke, R. und Simon, D. *Hierarchical Reflexion Models*. Proceedings of the 10th Working Conference on Reverse Engineering, IEEE Computer Society. 2003.

(Kruchten 1995) Kruchten, P. *The 4+1 View Model of Architecture*. IEEE Softw. 12(6): 42-50. *1995.*

(Lilienthal 2008) Lilienthal, C. *Komplexität von Softwarearchitekturen - Stile und Strategien* (Dissertation). Fachbereich Informatik, Universität Hamburg. 2008.

(Murphy et al. 1995) Murphy, G. C., Notkin, D. und Sullivan, K. *Software reflexion models: bridging the gap between source and high-level models*. Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering. Washington, D.C., United States, ACM Press. 1995.

(Murphy et al. 2001) Murphy, G. C., Notkin, D. und Sullivan, K. J. *Software Reflexion Models: Bridging the Gap between Design and Implementation*. IEEE Transactions on Software Engineering 27(4): 364-380. *2001*.

(Reussner und Hasselbring 2006) Reussner, R. und Hasselbring, W. *Handbuch der Software-Architektur*, dpunkt.Verlag. 2006.

(Scharping 2005) Scharping, A. *Architekturvereinbarungen des JCommSys* (Baccalaureatsarbeit). Fachbereich Informatik, Universität Hamburg. 2005.

(Sefika et al. 1996) Sefika, M., Sane, A. und Campbell, R. H. *Monitoring compliance of a software system with its high-level design models* Proceedings of the 18th international conference on Software engineering Berlin, Germany IEEE Computer Society**:** 387-396 1996

(Shaw und Garlan 1996) Shaw, M. und Garlan, D. *Software Architecture - Perspectives on an Emerging Discipline*. New Jersey, Prentice-Hall, Inc. 1996.

(Tvedt et al. 2002) Tvedt, R. T., Costa, P. und Lindvall, M. *Does the code match the design? A process for architecture evaluation*. Software Maintenance, 2002. Proceedings. International Conference on. 2002.

(Züllighoven 1998) Züllighoven, H. *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material Ansatz*, dpunkt Verlag. 1998.

(Züllighoven 2005) Züllighoven, H. *Object-Oriented Construction Handbook*, dpunkt Verlag. 2005.

Erklärung:

Ich versichere, dass ich die vorstehende Arbeit selbstständig 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.

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

Hamburg, den 11. Januar 2008

Arne Scharping

Halstenbeker Straße 89a

22457 Hamburg

Email: arne.scharping@informatik.uni-hamburg.de

Matrikelnr: 5422314