Diplomarbeit

Regeln der WAM-Modellarchitektur

Oktober 2005

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Bettina Karstens

email: 0bkarste@informatik.uni-hamburg.de
Matrikelnr.: 5299997

Erstbetreuung: Prof. Dr.-Ing. Heinz Züllighoven
Zweitbetreuung: Prof. Dr. Horst Lichter

Betreuung:
Prof. Dr.-Ing. Heinz Züllighoven (Erstbetreuer)
Prof. Dr. Horst Lichter (Zweitbetreuer)

Prof. Dr.-Ing. Heinz Züllighoven

Arbeitsbereich Softwaretechnik (SWT)
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Prof. Dr. Horst Lichter

Software-Konstruktion
Lehr- und Forschungsgebiet Informatik
Rheinisch-Westfälische Technische Hochschule Aachen (RWTH Aachen)
Ahornstraße 55
52074 Aachen

Danksagung

Besonders danken möchte ich Petra Becker-Pechau, die mir mit zahlreichen Ideen und Anregungen zur Seite stand. Dank ebenfalls an Heinz Züllighoven, der vor allem in der Endphase schnelle und konstruktive Kritik lieferte. Carola Lilienthal danke ich für das freiwillig angebotene Korrekturlesen und die vielen Diskussionen zum Sotographen.

Viel wertvolles Feedback erhielt ich in den Treffen des Diplomarbeitsprojektes, mein Dank gilt allen Teilnehmern.

Das tolle Arbeitsklima im Arbeitsbereich SWT hat mir die Entscheidung leicht gemacht meine Arbeit in Stellingen zu verfassen. Vielen Dank an alle Mitarbeiter.

Bedanken möchte ich mich nicht zuletzt bei Christoph, der mich bei all meinen Entscheidungen unterstützt hat und meiner Familie, besonders meinem Bruder, der meine Arbeit mehrmals Korrektur gelesen hat.

Inhaltsverzeichnis

Kapitel 1 Einleitung..1
1.1 Motivation..1
1.2 Fragestellungen und Zielsetzung..1
1.3 Aufbau der Arbeit..2

Kapitel 2 Grundlagen..4
2.1 Der Sotograph..4
2.2 Architektur..7
2.3 Der WAM-Ansatz..8
2.4 Beispiel-Softwaresysteme..18
2.5 Stand der Kunst in der Architekturüberprüfung..20

Kapitel 3 Regeln der WAM-Modellarchitektur..24
3.1 Regelbeispiele..24
3.2 Regelterminologie..24
3.3 Quelle der Regeln..25
3.4 Kriterien für die Regelauswahl..25
3.5 Klassifizierung der Regeln..26
3.6 Aufbau der Regelsammlung..27
3.7 Überprüfbarkeit von Regeln..27
3.8 Allgemeine Erläuterungen zu den Regeln..30
3.9 Die Regeln..30
3.10 Zusammenfassung..38

Kapitel 4 Implementierung der Regeln..39
4.1 Identifikation von Elementen..39
4.2 Das Materialien-Problem..42
4.3 Überprüfung der WAM-Modellarchitektur mit dem ModelManager des Sotographen..46
4.4 Queries..47
4.5 Die WAM-Queries..52
4.6 Zusammenfassung..58

Kapitel 5 Analyse des Pausenplanersystems..59
5.1 Die Architektur des Pausenplanersystems..59
5.2 Ergebnisse der Analyse des Pausenplanersystems..63
5.3 Bewertung der Ergebnisse..74
5.4 Zusammenfassung..74

Kapitel 6 Untersuchung weiterer Systeme..75
6.1 Ergebnisse der Analyse des EMS..75
6.2 Ergebnisse der Analyse des Gepardsystems..82
6.3 Zusammenfassung..89

Kapitel 7 Auswertung..90

7.1 Bewertung der Queries der Gruppen WAM-Init, WAM-Elements und WAM-Inheritance 92
7.2 Bewertung der Queries, die Regelverstöße suchen 92
7.3 Schwächen und Verbesserungsmöglichkeiten einzelner Queries 94

Kapitel 8 Fazit und Ausblick 97
8.1 Bewertung .... **Fehler! Textmarke nicht definiert.**
8.2 Ausblick 98

Anhang A 100
Regeln der WAM-Modellarchitektur 100
A.1 Allgemeine WAM-Regeln 100
A.2 Regeln für Materialien 100
A.3 Regeln für Werkzeuge (allgemein) 102
A.4 Regeln für Funktionskomponenten 102
A.5 Regeln für Interaktionskomponenten 105
A.6 Regeln für Tool-Klassen 106
A.7 Regeln für GUIs 107
A.8 Regeln für Monotool-Klassen 108
A.9 Regeln für Automaten 110
A.10 Regeln für Services 110
A.11 Regeln für Fachwerte 111

Anhang B 114
WAM-Queries 114
B.1 WAM-init 114
B.2 WAM-Elements 115
B.3 WAM-Inheritance 116
B.4 WAM-Forbidden-Relationships 116
B.5 WAM-Enforced-Relationships 117
B.6 WAM-Single-Element-Rules 117
B.7 WAM-FollowUp 118

Anhang C 119
Ausschnitt aus dem ER-Modell des Sotographen 119

Literaturverzeichnis: 121

Abbildungsverzeichnis

Abbildung 2.1: Der MetricScope des Sotographen 5
Abbildung 2.2: Der XrefScope des Sotographen 6
Abbildung 2.3: Der GraphScope des Sotographen 7
Abbildung 2.4: Typische Beziehungen zwischen den WAM-Komponenten 11
Abbildung 2.5: Ein komplexes Werkzeug.. 12
Abbildung 2.6: Ein monolithisches Werkzeug 12
Abbildung 2.7: Komponentendiagramm mit monolithischem Werkzeug.. 13
Abbildung 2.8: Komponentendiagramm mit komplexem Werkzeug 13
Abbildung 2.9: Beziehungen zwischen gleichartigen Kontext- und Subwerkzeugen 14
Abbildung 2.10: Beziehungen zwischen verschiedenartigen Kontext- und Subwerkzeugen 15
Abbildung 2.11: Beziehungen zwischen Werkzeug, Material, Service, Automat und Fachwert.. 16
Abbildung 2.12: Elemente der Umgebung 17
Abbildung 2.13: Das Pausenplanersystem.. 19
Abbildung 2.14: Das EMS 20

Abbildung 3.1: Verbotene Beziehungen zwischen WAM-Elementen........ 37
Abbildung 3.2: Vorgeschriebene Beziehungen zwischen WAM-Elementen38

Abbildung 4.1: Thing-Hierarchie in JWAM 1.8, nur direkte Beziehungen zu Thing 43
Abbildung 4.2: Thing-Hierarchie in JWAM 1.8, nur Beziehungen zu ThingImpl 44
Abbildung 4.3: Thing-Hierarchie in JWAM 2 44
Abbildung 4.4: WAM-Schichten... 46
Abbildung 4.5: Der QueryDeveloper 48
Abbildung 4.6: ResultScope mit angewählter FollowUp-Query im Kontextmenü 50
Abbildung 4.7: Definition einer Query... 51

Abbildung 5.1: Die Hauptwerkzeuge im Pausenplanersystem 60
Abbildung 5.2: Materialien im Pausenplanersystem 60
Abbildung 5.3: Materialien und Werkzeuge im Pausenplanersystem 61
Abbildung 5.4: Werkzeuge, Services, Materialien und Automat 62
Abbildung 5.5: Druck-Werkzeug, Pausenplan und Interfaces 66
Abbildung 5.6: Service-Vererbungshierarchie im Pausenplaner 67
Abbildung 5.7: Großer Klassenzyklus im Pausenplanersystem 73
Abbildung 5.8: Drei kleine Zyklen entstanden aus dem großen Klassenzyklus im Pausenplanersystem 73

Abbildung C.0.1: Tabellen Symbols, Classes, Methods u.a................. 119
Abbildung C.0.2: Tabelle SymbolReferences u.a. 120

Tabellenverzeichnis

Tabelle 3.1: Überprüfbarkeit der WAM-Regeln 30

Tabelle 5.1: Allgemeine Daten des Pausenplanersystems 63
Tabelle 5.2: WAM-Elemente im Pausenplanersystem 64
Tabelle 5.3: Vererbungsfehler im Pausenplanersystem 66
Tabelle 5.4: Verstöße gegen Verbots-Beziehungsregeln im Pausenplanersystem 68
Tabelle 5.5: Verstöße gegen Gebots-Beziehungsregeln im Pausenplanersystem 70
Tabelle 5.6: Verstöße gegen Einzel-Element-Regeln im Pausenplanersystem 71
Tabelle 5.7: Zyklen im Pausenplanersystem 72

Tabelle 6.1: Allgemeine Daten des EMS 75
Tabelle 6.2: WAM-Elemente im EMS 76
Tabelle 6.3: Vererbungsfehler im EMS 77
Tabelle 6.4: Verstöße gegen Verbots-Beziehungsregeln im EMS 79
Tabelle 6.5: Verstöße gegen Gebots-Beziehungsregeln im EMS 80
Tabelle 6.6: Verstöße gegen Einzel-Element-Regeln im EMS 81
Tabelle 6.7: Allgemeine Daten des Gepardsystems 82
Tabelle 6.8: WAM-Elemente im Gepardsystem 83
Tabelle 6.9: Vererbungsfehler im Gepardsystem 84
Tabelle 6.10: Verstöße gegen Verbots-Beziehungsregeln im Gepardsystem 85
Tabelle 6.11: Verstöße gegen Gebots-Beziehungsregeln im Gepardsystem 85
Tabelle 6.12: Verstöße gegen Einzel-Element-Regeln im Gepardsystem. 87
Tabelle 6.13: Zyklen im Gepardsystem 87

Tabelle 7.1: Übersicht der Queryergebnisse; VBR: Verbots-Beziehungsregeln, GBR: Gebots-Beziehungsregeln, ER: Einzel-Element-Regeln 92
Tabelle 7.2: Falsch Positive der Verbots-Beziehungsregeln 93
Tabelle 7.3: Falsch Positive der Gebots-Beziehungsregeln 93
Tabelle 7.4: Falsch Positive der Einzel-Element-Regeln 93

Kapitel 1 Einleitung

1.1 Motivation

Die Qualität eines Softwaresystems hängt davon ab, wie stark das System Aspekte wie u.a. Änderbarkeit, Testbarkeit und Funktionalität erfüllt (vgl. [Bass98], S.73 ff.). Die Architektur der Software beeinflusst maßgeblich, wie gut diese Aspekte umgesetzt werden. Es ist also von Vorteil, wenn bei der Softwareentwicklung von vorneherein eine Architekturvorstellung besteht und diese möglichst genau in das System übernommen wird.

Architekturvorstellungen können in Form von Modellarchitekturen ausgedrückt werden. Modellarchitekturen beschreiben, welche Arten von Elementen in einem System vorkommen dürfen, welche Beziehungen zwischen ihnen erlaubt sind und geben Regeln vor, die für diese Elemente und Beziehungen gelten müssen. Eine Modellarchitektur kann als Vorlage für die Architektur eines Systems dienen, indem die speziellen Elemente dieses Systems nach dem Muster der allgemeinen Elemente aus der Modellarchitektur implementiert werden.

Bei der Entwicklung eines Systems ist es möglich, dass nicht alle Regeln für das System eingehalten werden. Selbst wenn die Regeln bekannt sind, kann es bei der Umsetzung passieren, dass der Entwickler Regeln vergisst, oder dass durch mangelndes Verständnis der Regeln gegen sie verstoßen wird. Der Architekt des Systems hat jedoch zurzeit keine Möglichkeit herauszufinden, ob das System die Regeln einhält. Regelverstöße von Hand im Quelltext zu finden wäre zu aufwändig und zu fehleranfällig. Eine Möglichkeit zur automatischen Überprüfung existiert zurzeit noch nicht. Eine Werkzeugunterstützung, um Regelverstöße zu finden, ist daher nötig.

1.2 Fragestellungen und Zielsetzung

Eine explizite Modellarchitektur für objektorientierte Systeme ist die WAM-Modellarchitektur. Sie wird in [Zül98] ausführlich beschrieben und wurde bereits in vielen Projekten eingesetzt. Sie ist daher ein relevantes Beispiel für eine Modellarchitektur. Die Hauptfragestellung, die in dieser Arbeit beantwortet werden soll, ist die Folgende: Lässt sich die WAM-Modellarchitektur mit Hilfe des Software-Tomograph (s. [Sotograph]), kurz „Sotograph“ genannt, automatisch überprüfen?

Zur Beantwortung dieser Frage müssen folgende Teilfragen geklärt werden:

- Welche WAM-Regeln gibt es?
- Wie können die WAM-Regeln kategorisiert werden?
- Welche Regeln sind automatisch überprüfbar?
- Wie können die WAM-Elemente im Quelltext gefunden werden?
- Wie können die Regeln mit dem Sotographen überprüft werden?

Die Ziele dieser Arbeit sind daher, die Regeln der WAM-Modellarchitektur in einer Regelsammlung explizit zu machen, sie zu klassifizieren und sie daraufhin zu untersuchen, ob sie automatisch überprüfbar sind. Weiterhin soll ein Teil der Regeln durch so genannte Queries im Sotographen überprüfbar gemacht werden, um die Fragestellungen der konstruktiven Umsetzung zu diskutieren.

Der Sotograph ist ein Softwareanalyse-Werkzeug, das Informationen aus dem Source- und Bytecode eines Systems extrahiert und diese in einer relationalen Datenbank ablegt. Sie bildet die Grundlage für die verschiedenen Untersuchungen, die mit dem Sotographen durchführbar sind. Darunter fallen u.a. die Metrikanalyse und die Strukturanalyse, um gezielt Beziehungen zwischen Artefakten genauer zu betrachten. Graphen und Tabellen visualisieren die Ergebnisse. Mit dem Sotographen kann ein System auch daraufhin geprüft werden, ob die vorgesehene Architektur eingehalten wird. Dies ist für Schichtenarchitekturen möglich. Andere Architekturen können nur kontrolliert werden, wenn sie nur wenige Regeln für die Beziehung zwischen Elementen vorgeben, da die Modellierung der Architektur im Sotographen sonst zu aufwändig ist. Die WAM-Modellarchitektur ist eine komplexe Architektur, die aus vielen Regeln besteht und nicht allein durch ein Schichtenmodell darstellbar ist. Sie kann also mit den vorgegebenen Funktionen des Sotographen nicht überprüft werden. Um die WAM-Modellarchitektur testen zu können, wird der Sotograph in dieser Arbeit um so genannte „Queries“ erweitert. Mit ihnen können Verstöße gegen die Regeln der WAM-Modellarchitektur mit geringem Aufwand gefunden werden, wie die Anwendung auf drei Beispielsysteme zeigen soll.

1.3 Aufbau der Arbeit

Kapitel 2 stellt zunächst den Sotographen und die in dieser Arbeit verwendeten Architekturbegriffe vor. Des Weiteren wird der WAM-Ansatz erläutert, der die WAM-Modellarchitektur enthält. In Abschnitt 2.4 werden drei Systeme vorgestellt, die nach der WAM-Modellarchitektur implementiert wurden. Ihre Architektur wird in späteren Kapiteln mit den vorgenommenen Erweiterungen des Sotographen überprüft. Im letzten Abschnitt wird die einschlägige wissenschaftliche Literatur zum Thema Architekturüberprüfung diskutiert.

In Kapitel 3 geht es um die Erstellung der Regelsammlung für die WAM-Modellarchitektur. Zunächst wird auf die Quellen für die Regeln der WAM-Modellarchitektur eingegangen. Weiterhin wird diskutiert, welche Kriterien bei der Auswahl der Regeln für die Regelsammlung eine Rolle spielen. Die Regeln werden klassifiziert und es wird untersucht, für welche Art von Regeln Verstöße mit Hilfe des Sotographen gefunden werden können. Im letzten Teil des dritten Kapitels werden die WAM-Regeln vorgestellt, deren Einhaltung mit dem Sotographen überprüft werden kann.

Verstöße gegen die Regeln der WAM-Modellarchitektur sollen mit Hilfe des Sotographen gefunden werden. Kapitel 4 stellt dar, wie dieses Ziel technisch umgesetzt wird. Es wird zunächst darauf eingegangen, welche Überlegungen die Identifizierung der WAM-Elemente erfordert und welche Probleme gelöst werden müssen. Abschnitt 4.3 diskutiert, warum die vorhandenen Möglichkeiten des Sotographen nicht ausreichen, um die WAM-Modellarchitektur zu überprüfen. Daraufhin wird das grundsätzliche Prinzip vorgestellt, wie Queries im Sotographen erstellt werden. Im Anschluss daran erfolgt eine ausführliche Erläuterung aller Queries, die für die Analyse von WAM-Systemen implementiert wurden.

Um zu klären, wie gut die Queries Verstöße gegen die WAM-Regeln finden, wird in Kapitel 5 das Pausenplanersystem untersucht. Das Pausenplanersystem ist eine Software-Anwendung, die mit dem Rahmenwerk „JWAM“ implementiert wurde. Die Vorgaben der WAM-Modellarchitektur wurden nur zum Teil umgesetzt, d.h. es gibt in diesem System einige Architekturverletzungen.

In Kapitel 6 werden zwei weitere WAM-Systeme untersucht. Dadurch sollen die Ergebnisse, die die Analyse des Pausenplaners für die Queries gebracht hat, bestätigt werden. Ein System ist das „Equipment Management System“ (EMS), das als Beispiel für die WAM-Modellarchitektur entwickelt wurde. Es ist daher gut geeignet, die Gültigkeit der erstellten Regeln zu überprüfen. Das zweite System wurde ebenfalls nach der WAM-Modellarchitektur für ein internationales Dienstleistungsunternehmen entwickelt. Aus Datenschutzgründen wurde das Projekt anonymisiert.

Aus der Untersuchung der Beispielsysteme ergeben sich nicht nur Aussagen über die untersuchte Software, sondern auch Aussagen über die Nützlichkeit und Zuverlässigkeit der Queries. Die Nützlichkeit und Zuverlässigkeit der Queries und ihre Verbesserungsmöglichkeiten werden in Kapitel 7 diskutiert.

Kapitel 8 bewertet die Ergebnisse dieser Arbeit. Es erfolgt des Weiteren ein Ausblick, der über die Verbesserungsmöglichkeiten für die einzelnen Queries hinausgeht.

Kapitel 2 Grundlagen

Dieses Kapitel gibt dem Leser zunächst einen Überblick über den Sotographen und die in dieser Arbeit verwendeten Architekturbegriffe. Abschnitt 2.3 stellt den WAM-Ansatz vor, wobei deutlich gemacht wird, welche Teile des Ansatzes für diese Arbeit wichtig sind. Es folgt eine Beschreibung der in dieser Arbeit untersuchten Beispielsysteme und in Abschnitt 2.5 die Diskussion der wissenschaftlichen Literatur zum Thema Architekturüberprüfung.

2.1 Der Sotograph

Der Software-Tomograph (Sotograph) (s. [Sotograph]) ist eine Software-Analyseumgebung, mit der die innere Struktur eines Softwaresystems untersucht werden kann. Der Sotograph analysiert zunächst den Quelltext und den Bytecode der Software und legt Informationen zu dem System in eine relationale Datenbank ab. Alle Analysen, die der Benutzer mit den verschiedenen Werkzeugen des Sotographen durchführen kann, basieren auf den Informationen in der Datenbank. Einige Werkzeuge erlauben auch die Manipulation dieser Daten. Im Weiteren werden die Hauptfunktionen und die wichtigsten Werkzeuge des Sotographen beschrieben.

Im Sotographen ist der Aufbau des Systems, d.h. die Zusammenhänge von Packages, Klassen und Dateien durch ein so genanntes logisches Modell repräsentiert. Mit dem *ModelManager* kann in dieses Modell eine zusätzliche Abstraktionsebene eingefügt werden, indem der Benutzer die Packages eines Systems Subsystemen zuordnet. Die Subsysteme können in einem oder mehreren Architekturmodellen in Schichten angeordnet werden. Diese Anordnung legt erlaubte und verbotene Beziehungen zwischen Subsystemen fest. Neben Schichtenarchitekturmodellen können „GraphModels" genutzt werden, um erlaubte und verbotene Beziehungen zu spezifizieren. Der Sotograph kann dann automatisch prüfen, an welchen Stellen im Quelltext diese Architekturmodelle verletzt werden. Die primäre Architekturdefinition bezieht sich auf Schichtenarchitekturen, andere komplexe Architekturen können mit dem Sotographen nicht ohne großen Aufwand dargestellt werden.

Der *MetricScope* (s. Abb. 2.1) stellt eine Reihe von Metriken zur Verfügung, um bestimmte Werte des Systems zu berechnen. Angefangen von der Anzahl der Codezeilen bis hin zu der Anzahl der Klassen, die sich in Zyklen befinden, können die verschiedensten Eigenschaften des Systems ermittelt werden. Es gibt drei verschiedene Metrikarten. Metriken können erweiterte SQL-Abfragen sein, die Informationen aus der Datenbank extrahieren. Die zweite Art von Metriken beruht auf einem regulären Ausdruck und listet für jede Quelltextdatei auf, wie oft dieser Ausdruck gefunden wurde. Der Wert der dritten Metrikart bezieht sich ebenfalls nur auf eine Quelltextdatei. Metriken dieser Art werden mit Hilfe von Java-Klassen implementiert, die ein Plug-in Interface des Sotographen implementieren. Die Java-Klassen berechnen den gesuchten Metrikwert für eine Quelltextdatei, wie z.B. die Schachtelungstiefe von if-else-Ausdrücken. Werte, die Beziehungen zwischen Artefakten zählen, können nur über SQL-Metriken ermittelt werden, andere Arten von Metriken können nur Werte für einzelne Quelltextdateien liefern. Der Sotograph erlaubt es, eigene Metriken und Plug-ins zu definieren, um Aspekte des Systems zu untersuchen, die von den vorhandenen Metriken nicht abgedeckt werden. Stößt der Benutzer die Berechnung einer oder mehrerer Metriken an, so werden die Ergebnisse in der Datenbank gespeichert und können bei Bedarf im MetricScope angezeigt werden.

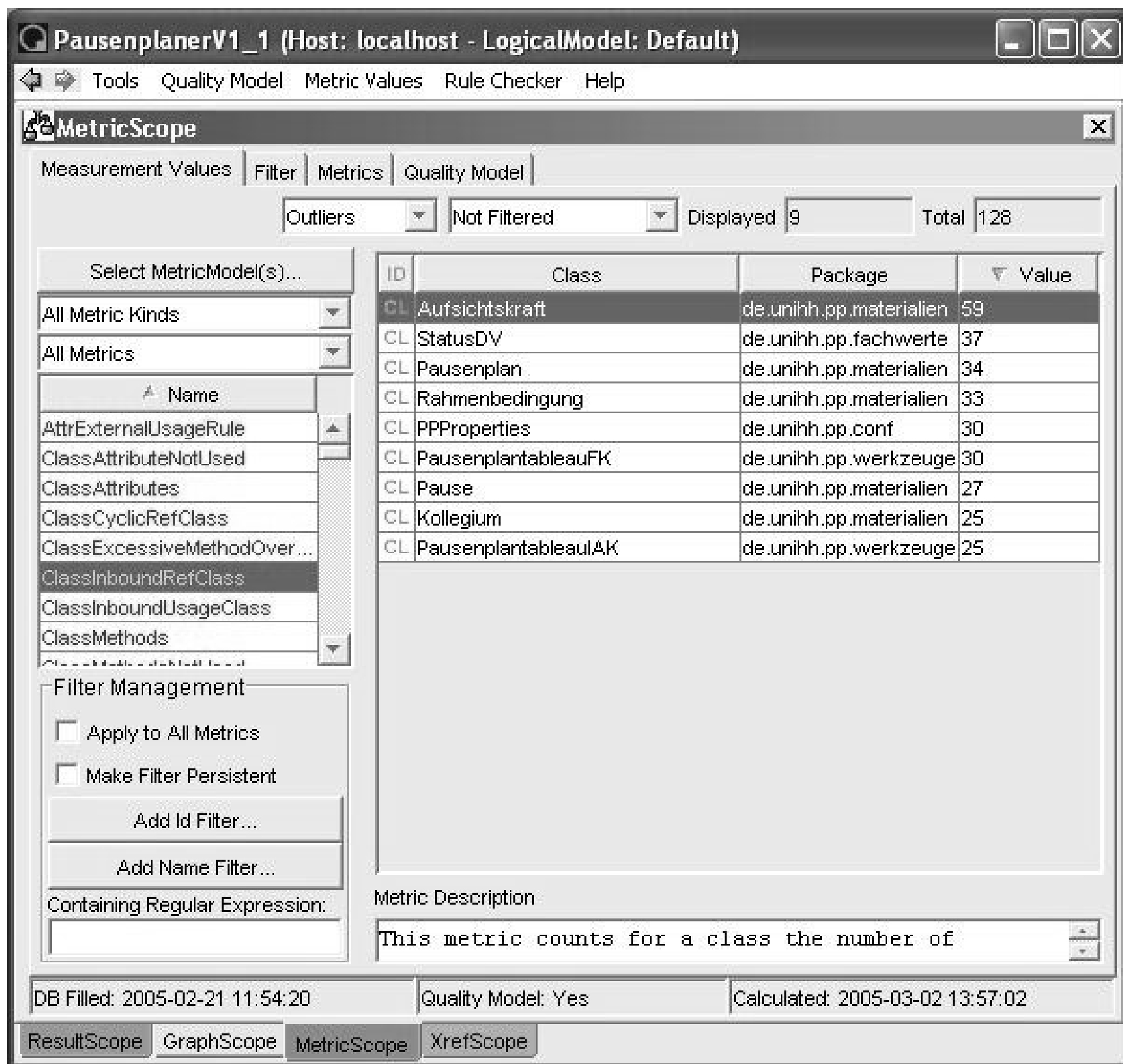

Abbildung 2.1: Der MetricScope des Sotographen

Ähnlich wie der MetricScope stellt auch der *QueryScope* vordefinierte Abfragen, „Queries“ genannt, zur Verfügung, um bestimmte Eigenschaften des Systems zu erkennen. Hier geht es zum Beispiel um unbenutzte Klassen, Flaschenhälse oder die Anzahl von Entwurfsmustern im Code. Queries müssen vom Benutzer einzeln ausgeführt werden. Die Ergebnisse speichert der Sotograph nicht wie bei Metriken in der Datenbank, sondern zeigt sie nur im ResultScope an. Der Benutzer kann eigene Queries im *QueryDeveloper* erstellen. Sie sind in Kapitel 4 genauer beschrieben.

Der *XrefScope* (s. Abb. 2.2) ermöglicht es, bestimmte Artefakte des Systems in den Mittelpunkt einer Analyse zu stellen. Zunächst wird ein Artefakt als Fokus gesetzt, im Weiteren kann die Art der Analyse festgelegt werden. Beispiele solcher Analysen sind: Welche Klassen enthält Package A? Welche Klassen erben von Klasse B? Welche Klassen werden von Klasse C benutzt?

Abgesehen von den Ergebnissen der Metriken werden Analyseergebnisse immer zuerst im *ResultScope* in einer Tabelle angezeigt.

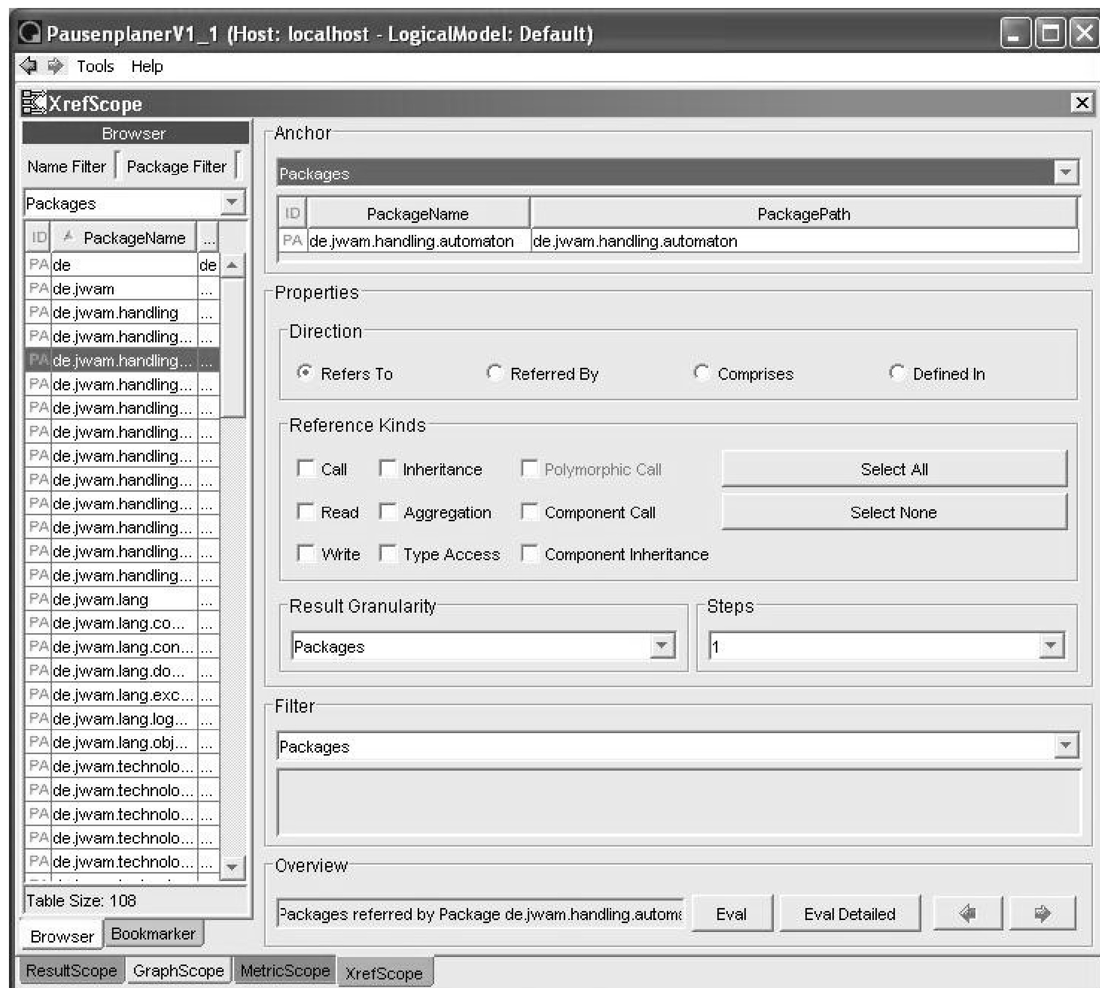

Abbildung 2.2: Der XrefScope des Sotographen

Artefakte, die in einer Tabelle dargestellt werden, kann der Sotograph auch als Graphik im GraphScope (s. Abb 2.3) anzeigen. Über einen Dialog kann der Benutzer auswählen, welche Beziehungen zwischen den Artefakten gezeichnet werden sollen. Der Benutzer kann sich bestimmte Ausschnitte aus einer Graphik, z.B. die Beziehung zwischen zwei Klassen des Graphen, auch wieder in Tabellenform ansehen.

Allen Werkzeugen gemein ist die Möglichkeit, eine Analyse auf verschiedenen Abstraktionsebenen durchzuführen. Um eine grobe Übersicht zu bekommen, sind Packages und Subsysteme gut geeignet. Um ein Problem zu verstehen ist es manchmal sinnvoll, es auf Klassen- oder sogar Methodenebene zu untersuchen. Die Möglichkeit direkt in den Quelltext zu springen, besteht in jedem Werkzeug des Sotographen.

In dieser Arbeit wurden hauptsächlich der QueryDeveloper und der QueryScope verwendet, um Queries zu erstellen und auszuführen. Die Anzeige der Ergebnisse erfolgte im ResultScope. Für die Zyklenanalyse wurde der MetricScope verwendet. Graphische Darstellungen von Beziehungen zwischen Klassen wurden mit dem GraphScope erstellt.

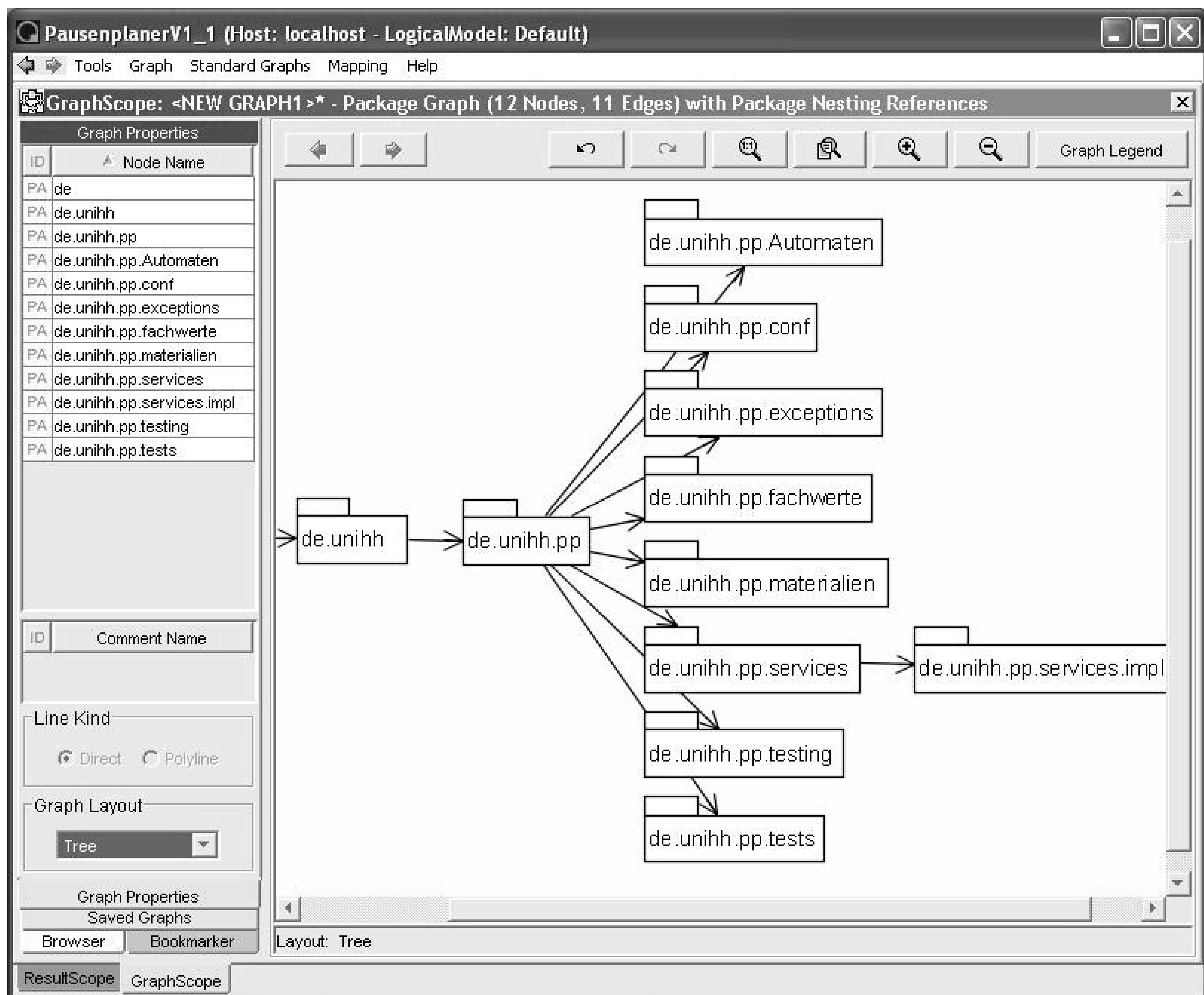

Abbildung 2.3: Der GraphScope des Sotographen

2.2 Architektur

Als *(Software-)Architektur* bezeichnet man die Architektur eines konkreten Softwaresystems. Eine Softwarearchitektur beschreibt den Aufbau des Systems, indem die einzelnen Elemente des Systems und ihre Beziehungen untereinander dargestellt werden. Eine Definition für Softwarearchitektur findet man in [Zül98]:
„Eine Softwarearchitektur bezeichnet die Modelle und die konkreten Komponenten eines Softwaresystems in ihrem statischen und dynamischen Zusammenspiel. Sie kann selbst als explizites Modell dargestellt werden. Eine Softwarearchitektur beschreibt ein konkretes System in seinem Anwendungskontext.“
Zur Architektur gehören neben den Komponenten und ihren Beziehungen auch die Schnittstellen der Komponenten, wie aus der Definition in [Bass98] deutlich wird:
„The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.“

Eine *Modellarchitektur* beschreibt einen Architekturstil (s. [Bass98], S. 25). Er ist eine Beschreibung von Elementarten und Beziehungen zwischen diesen Elementen. Ein Architekturstil kann laut [Bass98] als eine Menge von Einschränkungen gesehen werden, die eine Architektur einhalten muss, um ein Exemplar dieses Architekturstils zu sein. Eine Modellarchitektur gibt ebenfalls Regeln vor, die von Softwaresystemen eingehalten werden müssen, die nach ihr entworfen wurden. Eine Architektur, die

eine Modellarchitektur als Vorlage hat, kann man als „Exemplar“ dieser Modellarchitektur bezeichnen. Die Regeln der Modellarchitektur definieren Elemente, die in einem Exemplar dieser Modellarchitektur vorkommen dürfen und welche Beziehungen zwischen diesen Elementen erlaubt sind. Ein Unterschied zwischen einer Modellarchitektur und einem Architekturstil liegt darin, dass eine Modellarchitektur zusätzlich Richtlinien für das Design und die Qualität der Architektur umfasst. Bei der Konstruktion von Softwaresystemen ist es sinnvoll, Modellarchitekturen zugrunde zu legen, da sie durch ihre Regeln bereits ein Mindestmaß an Qualität sichern. Beispiele für Modellarchitekturen sind z.B. die Client-Server-Architektur oder die WAM-Modellarchitektur. Auch für den Begriff Modellarchitektur bzw. Architekturstil findet man Definitionen in [Zül98] und [Bass98]:
"Eine Modellarchitektur beschreibt die allgemeinen Prinzipien hinter einer Softwarearchitektur. Sie umfasst die grundlegenden Elemente, deren Verknüpfungen und die Regeln, die für eine Softwarearchitektur gelten. Eine Modellarchitektur gibt Anleitung bei der softwaretechnischen Realisierung eines Softwaresystems." [Zül98]

“An architectural style is a description of component types and a pattern of their runtime control and/or data transfer.“ [Bass98]

In dieser Arbeit bezieht sich der Begriff „Architekturregeln“ auf die Regeln der Modellarchitektur, die Elemente definieren und die Beziehungen zwischen diesen festlegen. Regeln, die Anleitung für einen guten Softwareentwurf geben, sind damit nicht gemeint. Architekturregeln speziell für die WAM-Modellarchitektur werden in dieser Arbeit als „Regeln der WAM-Modellarchitektur“, „WAM-Regeln“ oder auch nur als „Regeln“ bezeichnet, wenn der Zusammenhang eindeutig ist.

2.3 Der WAM-Ansatz

Dieser Abschnitt geht zunächst auf die Herkunft des WAM-Ansatzes und des Rahmenwerks JWAM (s. [JWAM]) ein. Es folgt ein Überblick über die für diese Arbeit relevanten Konzepte und Konstruktionsansätze des WAM-Ansatzes.

2.3.1 Der Ursprung des WAM-Ansatzes

Der WAM-Ansatz bietet Unterstützung für den Entwurf und die Konstruktion anwendungsorientierter Software und für das Vorgehen bei anwendungsorientierter Softwareentwicklung. Der WAM-Ansatz wurde im Arbeitsbereich SWT des Fachbereichs Informatik der Universität Hamburg entwickelt. Er wird sowohl in der Lehre als auch im spin-off Unternehmen der Universität, der C1 WPS GmbH, in vielen Projekten eingesetzt. Die Bezeichnung WAM steht für *Werkzeug, Automat* und *Material*. Diese Elemente sind neben den *Services* und *Fachwerten* die Hauptbestandteile der WAM-Modellarchitektur. Die Elemente Werkzeug, Automat und Material haben Metaphern aus dem Arbeitskontext der realen Welt als Ursprung. Mit diesen Elementen werden im Softwareentwurf die Gegenstände und Konzepte der Arbeitswelt realisiert. Durch diesen engen Zusammenhang zwischen den anwendungsfachlichen Begriffen und der Softwarearchitektur entstehen Vorteile für den Anwender und für den Entwickler: Die Anwender finden die Gegenstände ihrer Arbeit und ihre Fachsprache im Softwaresystem wieder und können ihre Arbeit in der gewohnten Weise organisieren. Die Entwickler haben es leichter Softwarekomponenten und Anwendungskonzepte

einander zuzuordnen und ihre gegenseitigen Abhängigkeiten zu erkennen (s. [Zül98], S. 5).

Das Rahmenwerk JWAM wurde von Mitarbeitern des Arbeitsbereichs SWT und Mitarbeitern der Firma C1 WPS entwickelt. Mit ihm können Systeme leichter nach der WAM-Modellarchitektur implementiert werden, da JWAM Oberklassen für die Elemente der WAM-Modellarchitektur zur Verfügung stellt. Während in vielen laufenden Projekten JWAM in der Version 1.8 eingesetzt wird, ist die Entwicklung von JWAM bereits bei Version 2 angelangt, die seit dem Jahr 2004 in Projekten verwendet wird.

2.3.2 Überblick über den WAM-Ansatz

Für die Entwicklung von Systemen nach dem WAM-Ansatz lassen sich drei Ebenen unterscheiden. Die erste Ebene bilden die WAM-Konzeptionsmuster, die die Grundkonzepte von WAM darstellen. Die Konzeptionsmuster helfen dem Entwickler, eine Vision des zu entwickelnden Systems zu entwerfen. Die Konzeptionsmuster beschreiben u.a. Werkzeuge, Materialien, Services und Automaten, s. Abschnitt 2.3.2.1. Konzeptionsmuster unterstützen die Modellierung des Anwendungsbereichs und sind meist nicht konkret genug, um aus ihnen softwaretechnische Regeln abzuleiten, die für ein WAM-System gelten sollen.

In der zweiten Ebene findet sich die WAM-Modellarchitektur. Sie gibt Anleitung, wie Systeme nach dem WAM-Ansatz realisiert werden. Als Konstruktionsanleitungen dienen Entwurfsmuster, die die Konzeptionsmuster der ersten Ebene konkretisieren. Durch die Modellarchitektur wird festgelegt, welche WAM-Elemente in einem WAM-System vorkommen, wie sie aufgebaut sind und welche Beziehungen es zwischen ihnen gibt. Die WAM-Entwurfsmuster erläutern z.B. den Aufbau komplexer Werkzeuge und die Beziehungen zwischen Werkzeugen, Materialien und anderen Elementen. Die WAM-Entwurfsmuster, die in Abschnitt 2.3.2.2 ausgeführt werden beschreiben den Teil der WAM-Modellarchitektur, der für diese Arbeit relevant ist und enthalten die Regeln, die in die Sammlung in Kapitel 3 aufgenommen wurden.

Die dritte Ebene behandelt die Implementierung der WAM-Modellarchitektur. In [Zül04] wird dafür das Rahmenwerk JWAM angeführt, das mit WAM-Entwurfsmustern implementiert wurde. JWAM bietet Unterstützung, um Systeme nach dem WAM-Ansatz zu entwickeln. In dieser Arbeit wird JWAM dazu genutzt, die Elemente der Modellarchitektur in einem WAM-System zu identifizieren.

In den folgenden Abschnitten werden die Konzepte der drei Ebenen zusammengefasst. Die Ausführungen beschränken sich auf die Teile des WAM-Ansatzes, die für diese Diplomarbeit relevant sind. Vollständige Beschreibungen finden sich in [Zül98] und [Zül04]

2.3.2.1 Konzeptionsmuster des WAM-Ansatzes

Werkzeuge sind ein zentraler Bestandteil des WAM-Ansatzes. Sie präsentieren Materialien und ermöglichen es, sie zu bearbeiten und zu sondieren. Ein Werkzeug ist durch seine anwendungsfachliche Funktionalität charakterisiert, die den Anwender bei wiederkehrenden Tätigkeiten unterstützt.

Ein weiteres zentrales Element des WAM-Ansatzes ist das *Material*. Materialien stehen für die Konzepte und Arbeitsgegenstände der realen Welt. Wesentlich ist, dass

die Umgangsformen von Materialien fachlich motiviert sind. Die Bearbeitung von Materialien steht für den Anwender im Mittelpunkt seiner Tätigkeit.

Ein *Service* bietet fachlich zusammenhängende Dienste an. Um sie zu realisieren benutzt der Service die Materialien, mit denen er umgeht und die er bereitstellt. Dienste werden häufig von Werkzeugen in Anspruch genommen. Services werden unabhängig davon implementiert, wie die Ergebnisse ihrer Dienstleistungen präsentiert werden.

Automaten übernehmen kleine Routinetätigkeiten. Im Gegensatz zu Werkzeugen geht beim Einsatz eines Automaten die Kontrolle vom Benutzer an den Automaten über, während eine Reihe von Handlungsschritten ausgeführt wird. Automaten arbeiten wie Werkzeuge auf Materialien, erfordern aber wenig Interaktion mit dem Benutzer.

Die *Arbeitsumgebung* ist der Ort, an dem Werkzeuge und Materialien bereit liegen. Sie hilft dem Anwender, die Arbeit zu organisieren und den Arbeitsplatz den eigenen Bedürfnissen anzupassen. Dabei ist die Sichtbarkeit und der Zugriff auf eine Arbeitsumgebung beschränkt, so dass Arbeitsgegenstände an ihrem Platz bleiben und eine konzeptionelle Einheit gegenüber dem Systemkontext gewahrt wird. Die Arbeitsumgebung bietet verschiedene Arten der Kooperation an, um die Zusammenarbeit mit Anderen zu erleichtern.

Behälter sind Materialien, die andere Materialien aufbewahren und anordnen. Die hier gemeinten fachlichen Behälter unterscheiden sich von rein technischen Behältern dadurch, dass ihre Umgangsformen fachlich motiviert sind. Dazu zählt, dass sie in der Lage sind, über die Sammlung fachliche Auskunft zu geben. Behälter zählen zwar zu den Materialien und können von Werkzeugen bearbeitet werden, nehmen aber eine Sonderstellung unter den Materialien ein, da sie andere Materialien organisieren und hierzu Operationen der Materialien aufrufen. Behälter können mit den in ihnen enthaltenen Materialien an verschiedene Orte transportiert werden.

Formulare sind spezielle Materialien. Sie enthalten hauptsächlich Felder, die ausgefüllt oder gelesen werden können. Dies passiert mit Hilfe generischer Lese- und Schreiboperationen. Die Felder eines Formulars bestehen aus Fachwerten (s. Abschnitt 2.3.2.2), um die Konsistenz des Formulars prüfen zu können.

Abbildung 2.4 zeigt ein Komponentendiagramm mit den typischen Beziehungen zwischen den Elementarten des WAM-Ansatzes. Abgesehen von Arbeitsumgebung und Werkzeug bestehen die Komponenten jeweils nur aus einer Klasse.

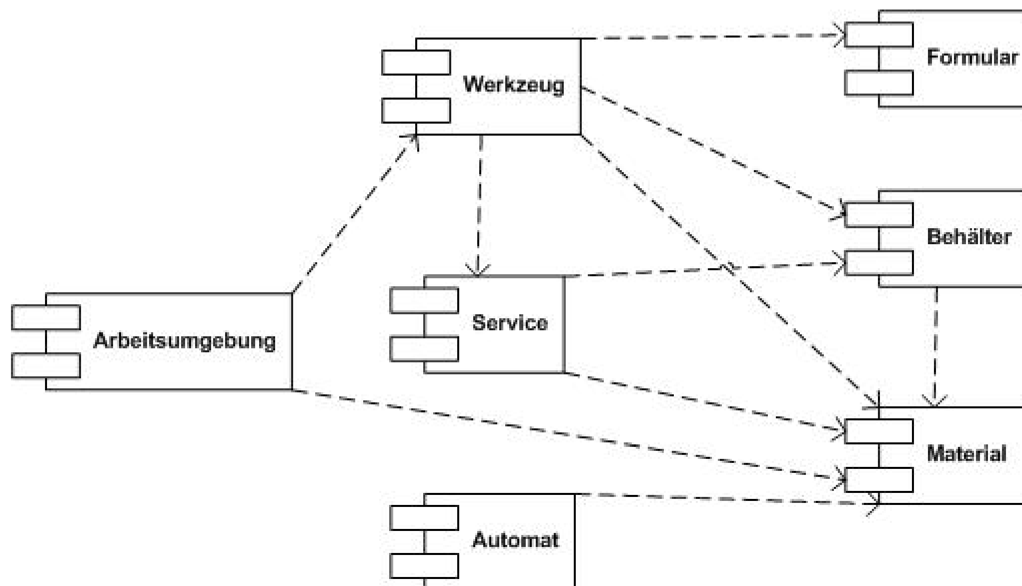

Abbildung 2.4: Typische Beziehungen zwischen den WAM-Komponenten

2.3.2.2 Entwurfsmuster des WAM-Ansatzes und die WAM-Modellarchitektur

Dieser Abschnitt stellt die Entwurfsmuster vor, die die Konzeptionsmuster des vorherigen Abschnittes präzisieren. Die dargestellten Entwurfsmuster beschreiben den Teil der WAM-Modellarchitektur, der für diese Arbeit relevant ist.

Materialien stellen an ihrer Schnittstelle fachliche Funktionalität zur Verfügung und kennen außer Fachwerten und anderen Materialien keine anderen Elemente der WAM-Modellarchitektur.

Behälter bewahren Materialien auf und legen ein Ordnungsprinzip für sie fest. Die Schnittstelle von Behältern wird durch die fachlichen Umgangsformen mit diesem Behälter bestimmt. Sie werden durch technische Behälter realisiert, aber nicht durch Spezialisierung von ihnen abgeleitet. Behälter haben eine Identität und häufig einen benutzerdefinierten Namen und verwalten ihre Elemente im Innern selbst. Behälter dürfen wie Materialien nur Materialien und Fachwerte kennen.

Services bieten an ihrer Schnittstelle Dienstleistungen an. Sie können Aufgaben an andere Services oder Automaten delegieren.

Automaten bearbeiten Materialien. Sie werden über Werkzeuge gestartet und dürfen Services benutzen, um ihre Aufgaben zu erfüllen.

Fachwerte erweitern die Menge der Standard-Datentypen und können von allen Elementen der WAM-Modellarchitektur benutzt werden. Sie folgen der Wertsemantik und können aus anderen Fachwerten zusammengesetzt werden. Außer anderen Fachwerten kennen sie keine Elemente der WAM-Modellarchitektur.

Formulare bestehen aus Formular-Elementen. Das einfachste Formular-Element ist das Formularfeld, das einen Fachwert enthält. Formulare werden nach dem Kompositum-Entwurfsmuster (Composite-pattern, s. [Gamma98]) strukturiert. Im Konstruktor eines Formulars werden alle Felder des Formulars angelegt. Sie können dann über die generischen Operationen der Formular-Klasse befüllt und ausgelesen werden.

Werkzeuge interagieren mit Automaten, Services und Materialien, um den Benutzer bei seinen Tätigkeiten zu unterstützen. Werkzeuge können als komplexe oder monolithische Werkzeuge konstruiert werden.

Komplexe Werkzeuge bestehen aus Tool-Klasse, Interaktionskomponente (IAK), Funktionskomponente (FK) und GUI (Graphical User Interface) (s. Abb. 2.5). Die GUI dient nur zur Erstellung der Widgets und der Gestaltung des Layouts. Die dynamischen Abläufe der Darstellung werden von der IAK gesteuert. Die FK stellt den funktionalen Kern des Werkzeugs dar. Sie kennt und bearbeitet das dargestellte Material. Zugriffe auf Services, andere Werkzeuge und Automaten finden nur durch die FK statt. Die Tool-Klasse dient als Abschluss des Werkzeugs gegenüber der Arbeitsumgebung. Alle Informationen, die das gesamte Werkzeug betreffen, sind in der Tool-Klasse angesiedelt. An dieser Stelle sei gesagt, dass sich der Begriff „Tool-Klasse“ in dieser Arbeit immer auf den Werkzeugabschluss und nie auf das ganze Werkzeug bezieht. Letzteres wird „Werkzeug“ genannt, nur bei der Implementierung der WAM-Regeln im Sotographen taucht der englische Begriff „tool“ stellvertretend für ein ganzes Werkzeug auf. Die gestrichelten Pfeile zwischen den Elementen eines komplexen Werkzeugs in Abbildung 2.5 stehen für eine lose Kopplung der Elemente, durchgezogene Pfeile für eine Benutzt-Beziehung.

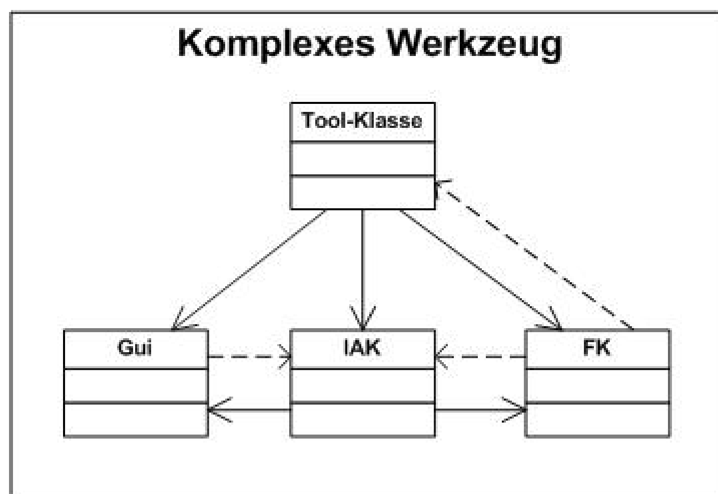

Abbildung 2.5: Ein komplexes Werkzeug

Monolithische Werkzeuge bestehen aus einer Monotool-Klasse und einer GUI (s. Abb. 2.6). Die Monotool-Klasse vereinigt die Funktionen von Tool-Klasse, IAK und FK.

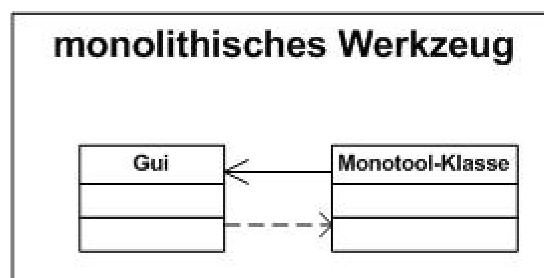

Abbildung 2.6: Ein monolithisches Werkzeug

Abbildungen 2.7 und 2.8 zeigen erneut das Komponentendiagramm aus Abschnitt 2.3.2.1. Hier wird das Werkzeug jedoch als monolithisches bzw. komplexes Werkzeug dargestellt.

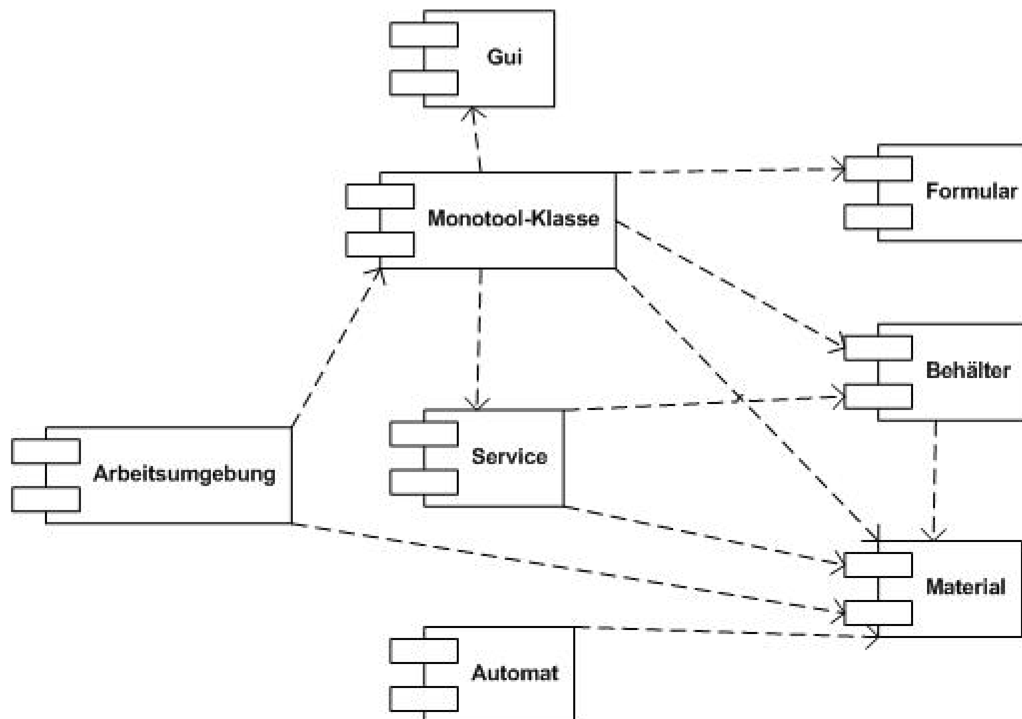

Abbildung 2.7: Komponentendiagramm mit monolithischem Werkzeug

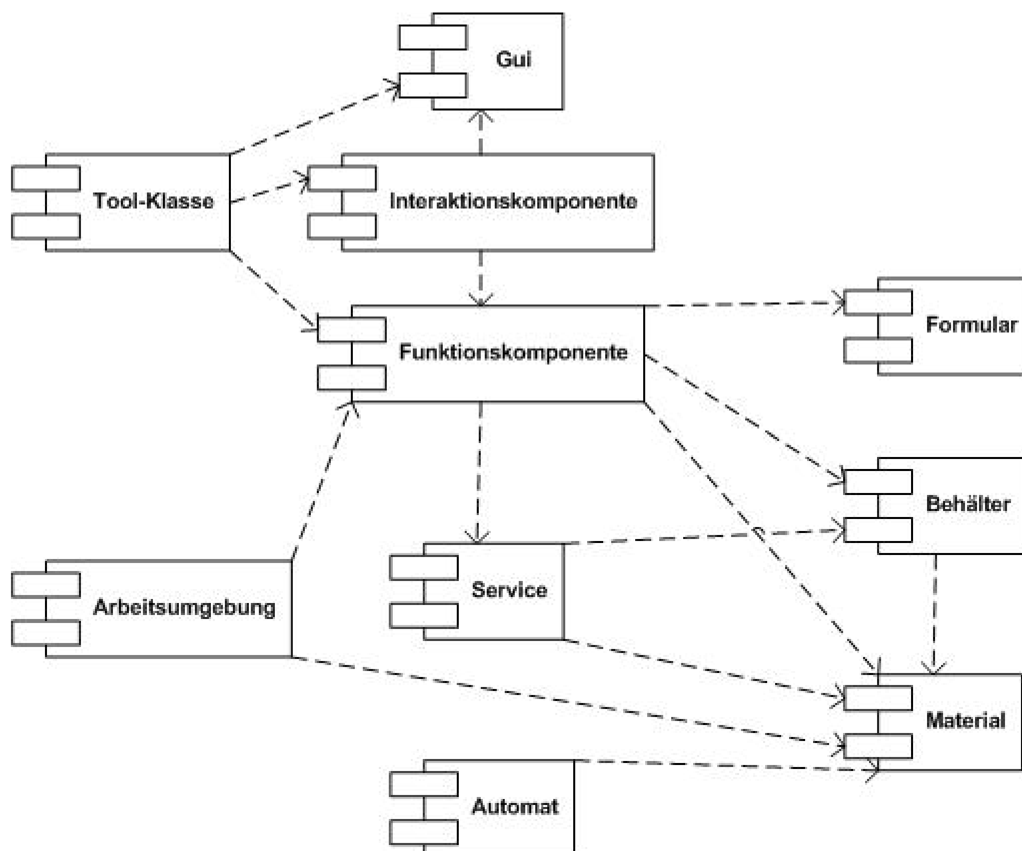

Abbildung 2.8: Komponentendiagramm mit komplexem Werkzeug

Werkzeuge können in andere Werkzeuge eingebettet werden. Zur Unterscheidung werden das eingebettete Werkzeug als „Subwerkzeug“ und das übergeordnete Werkzeug als „Kontextwerkzeug“ bezeichnet. Spricht man über einzelne Elemente eines Werkzeugs, so erhalten deren Namen ebenfalls das Präfix „Sub“ oder „Kontext“. Abbildung 2.9 zeigt, wie gleichartige Kontext- und Subwerkzeuge miteinander agieren, Abbildung 2.10 zeigt die Beziehungen zwischen Kontext- und Subwerkzeugen für komplexe und monolithische Werkzeuge.

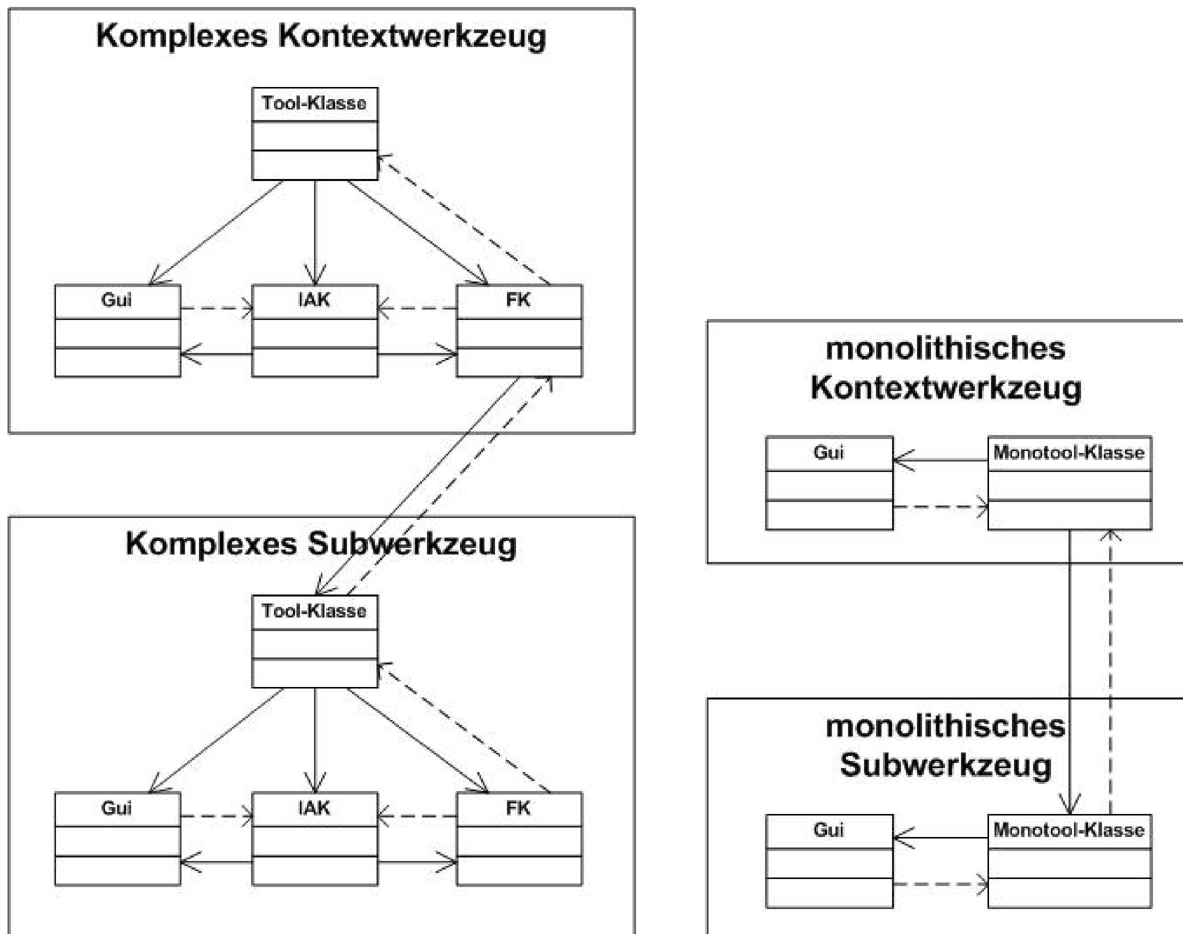

Abbildung 2.9: Beziehungen zwischen gleichartigen Kontext- und Subwerkzeugen

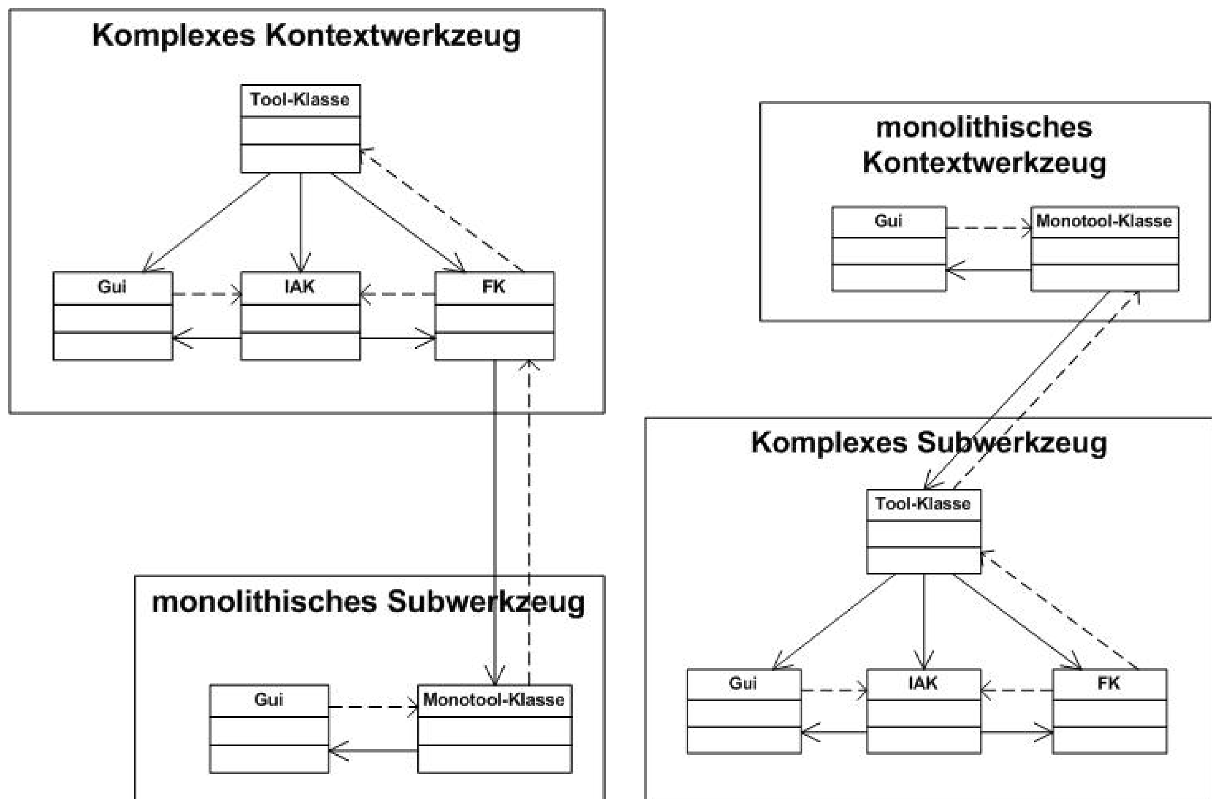

Abbildung 2.10: Beziehungen zwischen verschiedenartigen Kontext- und Subwerkzeugen

Um von der Handhabung eines Werkzeugs zu abstrahieren, werden *Interaktionstypen* (IATs) verwendet. Sie kapseln die verwendete Bibliothek für die graphische Benutzungsschnittstelle und verhindern so die Abhängigkeit der IAK von ihr. IATs werden von der IAK verwendet und sind über das Befehlsmuster (Command pattern, s. [Gamma98]) lose an sie gekoppelt. Systemereignisse werden von einem IAT in Programmereignisse umgewandelt. Informationen werden nur durch Fachwerte präsentiert und geliefert.

Abbildung 2.11 zeigt die Beziehungen der meisten der zuvor beschriebenen Elemente untereinander in einem Klassendiagramm. Aus Gründen der Übersichtlichkeit wurden die Pfeile für Rückkopplung, Reflexivität und Benutzt-Beziehungen zu Fachwerten weggelassen. Die lose Kopplung von Werkzeugelementen wurde in den vorherigen Abbildungen bereits dargestellt. Weitere lose Kopplungen können nur von Services zur FK oder zur Monotool-Klasse und von Automaten zur FK oder zur Monotool-Klasse bestehen. Jedes WAM-Element darf andere Elemente gleicher Art kennen, ein Pfeil zu sich selbst gehört daher an jedes WAM-Element. Fachwerte dürfen keine anderen WAM-Elemente benutzen und dürfen selbst von allen anderen WAM-Elementen benutzt werden. Von jedem WAM-Element hätte daher eine Benutzt-Beziehung zu Fachwerten ausgehen sollen.

Abbildung 2.11 zeigt, dass die WAM-Modellarchitektur eine Schichtenarchitektur enthält. Die oberste Schicht besteht aus den Werkzeugen, darunter liegen Services und Automaten. In der dritten Schicht folgen die Materialien, unter denen nur noch die Fachwerte liegen. Benutzt-Beziehungen über Schichtgrenzen hinweg dürfen nur nach unten bestehen. Die Schichten alleine können aber nicht alle Details der WAM-Modellarchitektur abbilden, wie vor allem die Beziehungen innerhalb der Werkzeugschicht deutlich machen.

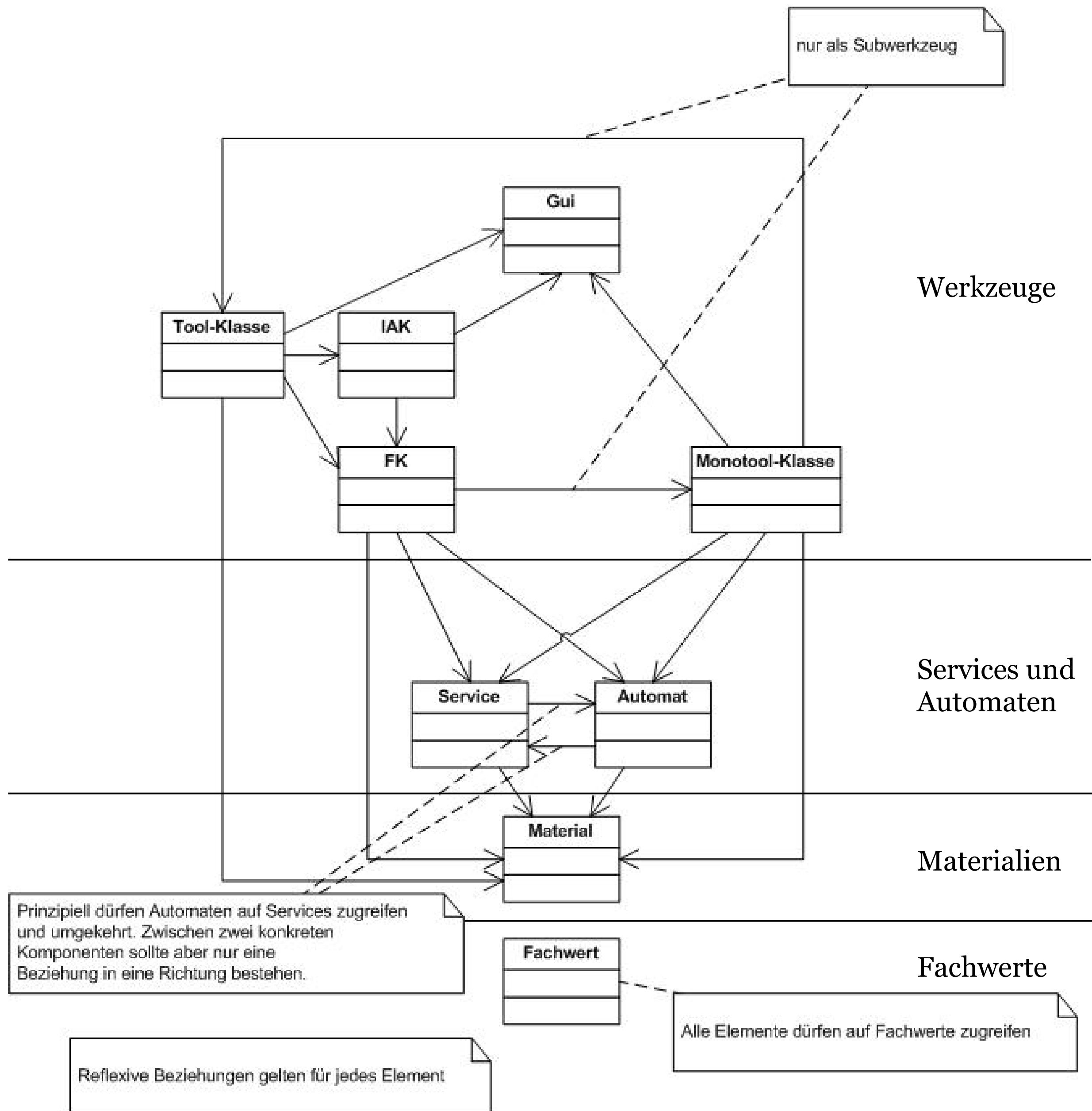

Abbildung 2.11: Beziehungen zwischen Werkzeug, Material, Service, Automat und Fachwert

Die *Arbeitsumgebung* ist der Kontext für alle in ihr enthaltenen Werkzeuge. Sie enthält den Ereignisverwalter, die Materialverwaltung und den Materialkoordinator. Im Folgenden werden diese Elemente beschrieben, ihr Zusammenspiel mit der Werkzeug-FK zeigt Abbildung 2.12 als Klassendiagramm.

Der Ereignisverwalter wird von den Werkzeugen in der Umgebung beobachtet. Erhält der Ereignisverwalter von einem Werkzeug die Meldung, dass ein Material verändert wurde, verteilt er diese Nachricht an die ihn beobachtenden Werkzeuge, die sich für eine solche Materialänderung interessieren.

Die Materialverwaltung stellt auf Anfrage von Werkzeugen und Automaten Materialien zur Bearbeitung bereit. Für die Materialverwaltung werden ein Materialverwalter, ein Materialmagazin und ein Materialversorger benötigt. Der Materialverwalter

ist ein Automat, bei dem Materialien angefordert werden können. Er ist den Werkzeugen und Automaten der Umgebung bekannt. Er verwaltet die Identifikationen aller Materialien dieser Umgebung in einem Behälter, dem Materialmagazin. Der Materialverwalter fordert die Materialien von einem Automaten an, dem Materialversorger. Dieser beschafft die Materialien aus einem Persistenzmedium. Materialien werden über Materialsammlungen zwischen Materialverwalter und Materialversorger und zwischen Materialverwalter und Werkzeugen ausgetauscht. Materialsammlungen sind Behälter, die Materialen vom Typ „Verwaltbar" enthalten.

Die Administrationsumgebung stellt anderen Umgebungen Dienstleistungen zur Verfügung. In ihr wird der Materialkoordinator erzeugt. Er ist ein Automat, der den konkurrierenden Zugriff auf Materialien aus verschiedenen Arbeitsumgebungen regelt. Jede Umgebung erhält einen Materialkoordinator-Proxy. Wird ein Material beim Materialverwalter von einem Werkzeug angefordert, so fragt der Materialverwalter zunächst beim Proxy nach, ob das Material schon in einer anderen Arbeitsumgebung in Bearbeitung ist. Ist das Material bereits in Bearbeitung, so erhält das Werkzeug nur eine Kopie des Materials. Andernfalls erhält es das Original und der Materialverwalter wird beim Materialkoordinator als Halter des Originals registriert.

Abbildung 2.12 zeigt, dass der Ereignisverwalter mit dem Materialverwalter über einen Reaktionsmechanismus verbunden ist. Dadurch kann der Materialverwalter den Ereignisverwalter informieren, wenn ein Material in einer anderen Arbeitsumgebung verändert wurde. Der Materialverwalter erhält diese Information über den Materialkoordinator-Proxy.

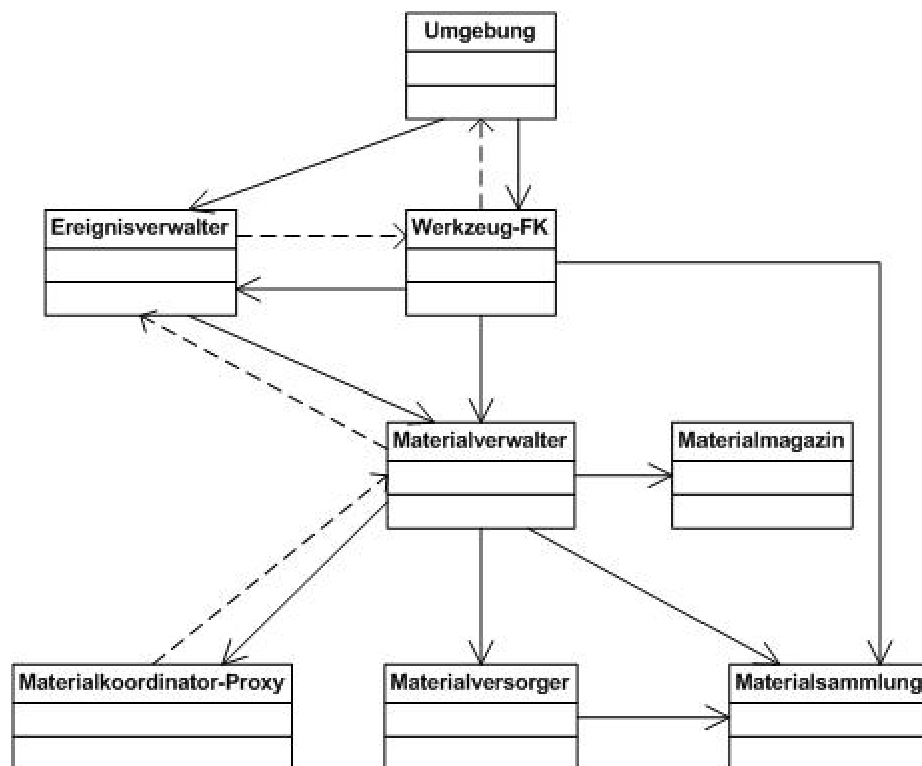

Abbildung 2.12: Elemente der Umgebung

Die in diesem Abschnitt dargestellten Entwurfsmuster beschreiben den Teil der WAM-Modellarchitektur, der für diese Arbeit relevant ist. Die Abbildungen haben Teile der WAM-Modellarchitektur anschaulich dargestellt. Aufgrund der Komplexität

der WAM-Modellarchitektur ist es nicht möglich, alle Elemente und ihre Beziehungen übersichtlich in einer Graphik darzustellen.

2.3.2.3 JWAM

Das JWAM Rahmenwerk (s.[JWAM]) bietet Unterstützung, um Systeme nach dem WAM-Ansatz zu konstruieren. Zur Implementierung des JWAM-Rahmenwerks wurden die Entwurfsmuster benutzt, die in Abschnitt 2.3.2.2 beschrieben wurden. Das Zusammenspiel zwischen verschiedenen Entwurfsmustern in JWAM ist in [Zül04], S.271ff. beschrieben.

Interfaces und Oberklassen für viele WAM-Elemente sind im Rahmenwerk vorhanden und implementieren ihre Grundfunktionen. Die konkreten WAM-Elemente aus dem zu konstruierenden System sollten die entsprechenden Interfaces implementieren bzw. von den entsprechenden Oberklassen erben. JWAM stellt des Weiteren eine Arbeitsumgebung zur Verfügung, die genutzt werden kann, um das Zusammenspiel von Werkzeugen und Materialien zu koordinieren.

2.4 Beispiel-Softwaresysteme

Die folgenden Abschnitte geben eine kurze Beschreibung der Beispielsysteme, die für diese Arbeit untersucht wurden. Zunächst wird das Pausenplanersystem vorgestellt, dass mit 235549 Codezeilen und 196 Klassen das kleinste der drei Systeme ist. Es folgt das EMS, das aus 31931 Codezeilen und 273 Klassen besteht. In Abschnitt 2.4.3 wird das Gepardsystem vorgestellt, das mit 113418 Codezeilen und 567 Klassen mit Abstand das größte der drei Systeme ist.

2.4.1 Das Pausenplanersystem

Das Pausenplanersystem (s. Abb. 2.13) wurde von Studenten im Rahmen des Studienprojektes „Objektorientierte Softwareentwicklung“ am Arbeitsbereich SWT des Fachbereichs Informatik der Universität Hamburg in den Jahren 2002 und 2003 entwickelt. Es handelt sich hierbei um ein kleines Softwaresystem, das die Planung von Pausenaufsichten unterstützt. Es wird in einer Schule für behinderte Kinder in der Nähe von Hamburg eingesetzt, in der die lückenlose Beaufsichtigung der Kinder in den Pausen sehr wichtig ist.

Das System sollte nach der WAM-Modellarchitektur entworfen werden und wurde mit dem Rahmenwerk JWAM umgesetzt. Aufgrund der mangelnden Erfahrung und fehlendem Wissen der Studenten um diese Architektur traten bei der Implementierung einige Fehler auf. Die Grundelemente der WAM-Modellarchitektur sind vorhanden, das System hält aber nicht alle Regeln der WAM-Modellarchitektur ein.

Einige Verletzungen der WAM-Modellarchitektur in diesem System waren im Vorfeld dieser Arbeit bereits bekannt. Das System war daher gut geeignet um zu zeigen, dass der Sotograph Verstöße gegen die Regeln der WAM-Modellarchitektur finden kann. Bei der Analyse des Systems mit dem Sotographen wurden die bekannten Verstöße, aber auch weitere Regelverletzungen gefunden.

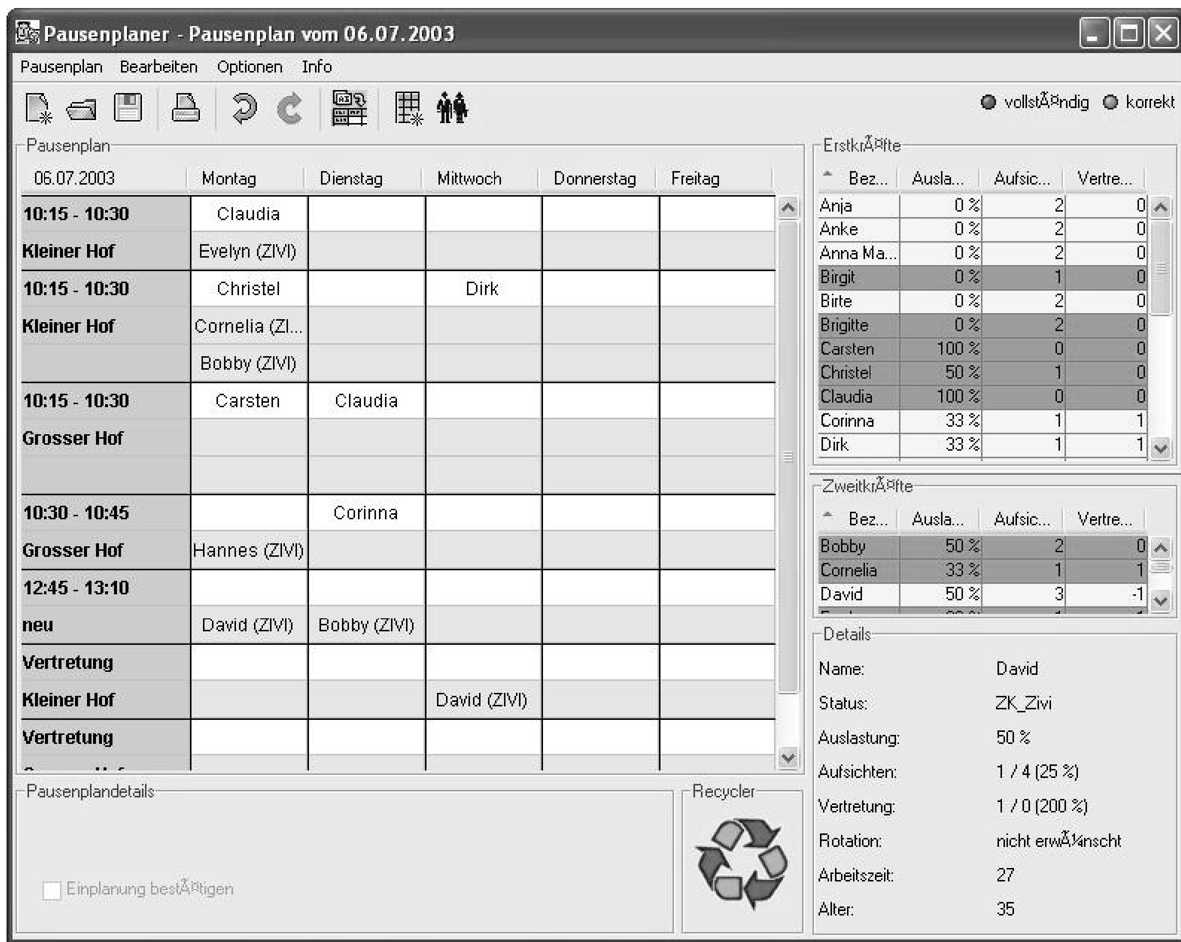

Abbildung 2.13: Das Pausenplanersystem

2.4.2 EMS

EMS steht für „Equipment Management System“ (s. [EMS] und Abb. 2.14). Mit diesem System können Personal, Räume und Geräte verwaltet werden. Mitarbeiter der C1 WPS und Studierende und Mitarbeiter der Universität Hamburg haben dieses kleine Softwaresystem als Beispiel für [Zül04] entwickelt. Das Buch erklärt, wie mit dem WAM-Ansatz objektorientierte Systeme entwickelt werden können. Das EMS verdeutlicht die Konstruktionsprinzipien des WAM-Ansatzes. Das EMS enthält die meisten Elemente, die in der WAM-Modellarchitektur vorgegeben sind, und hält alle Regeln dieser Modellarchitektur ein.

Da dieses System explizit als Beispiel-Anwendung für die WAM-Modellarchitektur entwickelt wurde, ist es gut geeignet, um die in dieser Arbeit gesammelten Regeln daran zu überprüfen. Bei der Überprüfung der Regeln sollten in diesem System keine Verstöße festgestellt werden können.

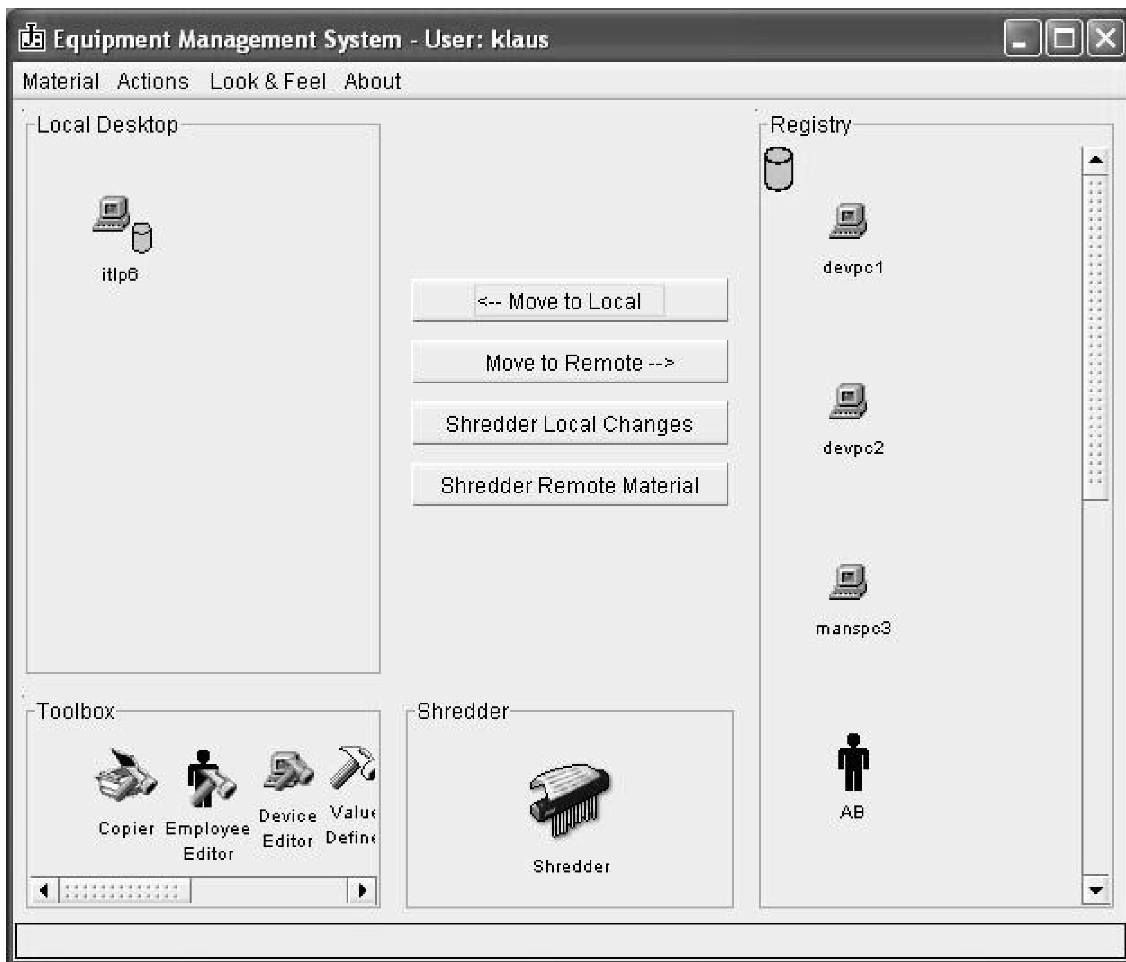

Abbildung 2.14: Das EMS

2.4.3 Das Gepardsystem

Die C1 WPS hat dieses System unter Benutzung des JWAM-Rahmenwerks in der Version 1.7 für einen internationalen Dienstleister entwickelt. Es handelt sich um ein Anlagebuchhaltungssystem, das buchhalterisch Geschäftsvorfälle in zwei Buchungskreisen nachvollzieht. Aus Datenschutzgründen wurden alle Daten dieses Projektes anonymisiert.

2.5 Stand der Kunst in der Architekturüberprüfung

Die Überprüfung von Softwarearchitekturen wird in mehreren Artikeln behandelt. Dabei liegt der Fokus meist auf der Untersuchung der Architektur eines bestimmten Systems. Architekturprüfungen, die mit wenig Aufwand auch auf andere Systeme angewendet werden können, finden sich kaum. Die Architektur eines Systems wird häufig durch die Angabe von Architekturelementen, meist „Subsysteme" genannt, und den erlaubten Beziehungen zwischen ihnen definiert. Die Art der Zuordnung von Klassen, Funktionen oder Packages zu Subsystemen variiert dabei stark. In einigen Artikeln wird auf diese Zuordnung gar nicht eingegangen, weil der Fokus auf der Architekturüberprüfung selber liegt. Die Prüfung der Beziehungen zwischen Elementen und die Darstellung der Ergebnisse erfolgt tabellarisch, in einigen Fällen auch graphisch.

Im Folgenden werden einige Ansätze beschrieben, die sich mit der Überprüfung von Architekturen beschäftigen.

In [Feijis98] wird ein relationaler Ansatz verfolgt, um Softwarearchitekturen zu analysieren. Beziehungen zwischen Funktionen werden durch Relationen dargestellt. Mit einer „lifting“ genannten Technik werden diese Beziehungen von Quelltextebene auf eine höhere Abstraktionsebene in Beziehungen zwischen Subsystemen projiziert. Verstöße gegen Regeln für die Beziehungen zwischen Subsystemen werden durch die Anwendung relationaler Algebra gefunden. Bei diesem Ansatz ist jedoch unklar, wie Funktionen Subsystemen zugeordnet werden.

Tvedt, Lindvall und Costa verfolgen in [Tvedt02] einen semi-automatischen Ansatz, um Architekturzerfall zu finden. Die Analyse basiert auf einer Metrik, die die Kopplung zwischen Komponenten berechnet. Die Ergebnisse dieser Metrik, Beziehungen zwischen Klassen, Packages und Subsystemen, werden in Tabellen dargestellt. Die Abweichungen und Verletzungen in Bezug auf die vorgesehene Architektur des Systems müssen von Hand identifiziert werden. Auch bei diesem Ansatz ist unklar, wie Klassen Subsystemen zugeordnet werden.

Murphy, Notkin und Sullivan beschreiben in [Murphy01], wie sie so genannte „Reflexion Models“ nutzen, um die Abweichungen von der vorgesehenen Architektur des Systems zu visualisieren. Der Entwickler spezifiziert zum Einen die Subsysteme und deren Beziehungen, das so genannte „high level model“, und zum Anderen die Zuordnung zwischen Klassen und Subsystemen. Die Zuordnung kann mit Hilfe relationaler Ausdrücke über die Verzeichnisstruktur oder auch über Dateinamen geschehen. Aus diesen Angaben wird ein Reflexion Model berechnet, das graphisch die Abweichungen zwischen dem System und der spezifizierten Architektur darstellt. Es werden Beziehungen dargestellt, die übereinstimmen, Beziehungen, die im high level model vorhanden sind, aber nicht im System und Beziehungen, die es im System gibt, aber nicht im high level model. Die Beziehungen können auch als Listen von Beziehungen zwischen Quelltextdateien angezeigt werden. Die Autoren sehen ihre Arbeit als nützlich an für Reengineering, Architekturprüfungen und das Verstehen von Systemen. Für den letzten Punkt wird jedoch ein rudimentäres Verständnis von den Subsystemen des Systems nötig sein, um überhaupt ein high level model erstellen zu können.

Guo, Atlee und Kazman, stellen in [Guo99] eine semi-automatische Methode vor, um Architekturen auf Basis von Entwurfsmustererkennung zu rekonstruieren. Die aus dem Quelltext extrahierten Informationen werden in einer relationalen Datenbank gespeichert. Um ein Entwurfsmuster in einem System zu finden, müssen zunächst Regeln aufgestellt werden, die das Entwurfsmuster charakterisieren. Die Regeln können auf Beziehungen zwischen Klassen basieren, aber auch auf Namenskonventionen für Klassen. Die Regeln werden als SQL-Queries implementiert. Gruppen von Software-Artefakten, die diese Regeln einhalten, werden als eine Instanz dieses Entwurfsmusters erkannt. Der Benutzer muss aus den Ergebnissen der Analyse die falsch Positiven und die falsch Negativen per Hand heraussuchen. Falsch Positive sind Entwurfsmusterinstanzen, die gar keine darstellen sollen. Falsch Negative sind Entwurfsmusterinstanzen, die nicht erkannt wurden. D.h. soll in einem System an einer Stelle ein Entwurfsmuster verwendet werden, so müssen die Grundregeln eingehalten werden, damit es entdeckt werden kann. Um fehlerhaft implementierte Entwurfsmuster zu erkennen, muss der Benutzer wissen, wo sich die Entwurfsmuster

im System befinden. Dann kann er die vorgesehenen Entwurfsmuster mit den gefundenen vergleichen.

Ein weiterer Ansatz, um Entwurfsmuster zu identifizieren, aber auch um sie zu überprüfen, wird in [Sefika96] vorgestellt. Statt SQL-Queries, wie im vorigen Ansatz, werden hier Prolog-Klauseln benutzt, um Entwurfsmusterinstanzen zu finden. Neben diesen Klauseln existieren weitere Klauseln, die Verletzungen der Regeln dieser Entwurfsmuster erkennen. Der Benutzer dieses Ansatzes wählt aus einer Liste das Entwurfsmuster aus, nach dem gesucht werden soll. Die Prolog-Klauseln werden ausgeführt und die gefundenen Entwurfsmuster graphisch dargestellt. Beziehungen zwischen Elementen des Systems werden schwarz oder grau dargestellt, je nachdem ob es sich um eine erlaubte oder eine verbotene Beziehung handelt. Verbotene Beziehungen können im Quelltext der betreffenden Klasse angezeigt werden. Ein Problem bei dieser Analyse ist, dass es für Entwurfsmuster viele verschiedene Möglichkeiten der Implementierung gibt. Die Regelbasis deckt zwar viele dieser Variationen ab, es kann aber nicht garantiert werden, dass die Muster immer erkannt werden. Ein Vorteil dieses Ansatzes ist, dass die erstellte Regelbasis auch auf andere Systeme angewendet werden kann. Der Ansatz beschränkt sich also nicht auf die Untersuchung eines speziellen Systems. Nachteile dieses Ansatzes sind, dass nicht überprüft wird, ob Entwurfsmuster dort implementiert wurden, wo sie vorgesehen waren. Stellen, an denen ein Entwurfsmuster vorgesehen war, grundlegende Regeln des Entwurfsmusters jedoch verletzt wurden, können mit diesem Ansatz nicht gefunden werden. Dies liegt daran, dass die Identifikation über einen Teil der Regeln des Entwurfsmusters geschieht, die daher nicht überprüft werden können.

Drei der in diesem Abschnitt vorgestellten Artikel, [Feijis98], [Tvedt02] und [Murphy01], unterscheiden sich von dem Ansatz dieser Diplomarbeit vor allem dadurch, dass sich die Architekturüberprüfung auf die Verknüpfungen zwischen Subsystemen bezieht. Es werden keine Regeln für die Relationen zwischen einzelnen Architekturelementen aufgestellt und kontrolliert, wie es in dieser Diplomarbeit der Fall ist. In den drei genannten Artikeln wird jeweils eine Architekturüberprüfung für ein spezifisches System vorgestellt. In [Feijis98] und [Murphy01] können die Abweichungen von der vorgesehenen Architektur automatisch erkannt werden, in [Tvedt02] ist dies nur per Hand möglich. Alle Ansätze dieser Artikel könnten genutzt werden, um einen Teil der Verletzungen von WAM-Regeln zu finden. Verstöße gegen Regeln, die sich nur auf ein Element beziehen oder Regeln, die eine Verbindung zwischen zwei bestimmten Elementen vorschreiben, könnten nicht überprüft werden. Es wäre jedoch möglich, einen Großteil der Verletzungen zu entdecken, die sich auf Relationen zwischen Elementarten beziehen. Dazu müssten alle Elemente einer Art in ein Subsystem gruppiert werden. Dann könnten die Verbindungen zwischen diesen Subsystemen überprüft werden. Die vorgestellten Ansätze erlauben es aber nicht, Elemente über Vererbungsbeziehungen zu identifizieren, wie es in dieser Diplomarbeit geschieht. In [Murphy01] wird erwähnt, dass relationale Ausdrücke verwendet werden, um Subsysteme zu erstellen. Mit diesem Ansatz könnte man Elemente der WAM-Modellarchitektur über ihre Klassennamen identifizieren. Auf diese Art können jedoch nicht alle WAM-Elemente erkannt werden und es treten mehr Fehler auf als bei der Identifizierung über Vererbungsbeziehungen.

Während die Architekturüberprüfungen, die in [Feijis98], [Tvedt02] und [Murphy01] dargestellt werden, auf ein bestimmtes System angepasst sind, stellen [Sefika96] und [Guo99] einen Ansatz vor, der ohne großen Aufwand auch für andere Systeme verwendet werden kann, wie auch der Ansatz dieser Diplomarbeit. Allerdings handelt es

sich in den Artikeln um die Erkennung und Überprüfung von Entwurfsmustern und nicht um die Erkennung und Kontrolle von Elementen und Beziehungen einer Modellarchitektur. In [Guo99] werden die Elemente der Entwurfsmuster über Regeln identifiziert, die als SQL-Queries implementiert sind. Diese Regeln können sich auch auf Namenskonventionen beziehen. Der Ansatz, Elemente über Namenskonventionen zu erkennen wird z.T. auch in dieser Diplomarbeit benutzt. Eine weitere Ähnlichkeit zu dieser Diplomarbeit besteht in der Nutzung von SQL um Regeln zu implementieren. Der Ansatz in [Sefika96] hat wenig Ähnlichkeit mit dem Ansatz dieser Arbeit. Beiden Ansätzen gemein ist, dass zwischen der Identifikation von Entwurfsmuster und dem Finden von Verletzungen unterschieden wird und dass sowohl die Entwurfsmuster als auch die Verletzungen automatisch erkannt werden können.

Architekturelemente in einem System werden in dieser Diplomarbeit darüber identifiziert, dass die Klassen, die die Architekturelemente repräsentieren, von bestimmten Klassen des Rahmenwerks JWAM erben. Ein solcher Ansatz, Architekturelemente über Vererbungsbeziehungen zu finden, wird in keinem der Artikel verfolgt.

Keiner der in diesem Abschnitt vorgestellten Ansätze könnte alle Verstöße gegen Regeln der WAM-Modellarchitektur finden, die mit den in dieser Arbeit vorgestellten Queries gefunden werden können. Nur WAM-Regeln, die sich auf Verbindungen zwischen Elementen beziehen, könnten, z.T. mit großem Aufwand, mit diesen Ansätzen überprüft werden. Verstöße gegen Regeln, die nur für ein einzelnes Element gelten, könnten mit keinem dieser Ansätze entdeckt werden.

Kapitel 3 Regeln der WAM-Modellarchitektur

Um ein System nach der WAM-Modellarchitektur zu entwickeln, ist es notwendig, die Regeln der WAM-Modellarchitektur zu kennen. In diesem Kapitel geht es daher um die Erstellung einer Regelsammlung für die WAM-Modellarchitektur. Dabei werden u.a. folgende für diese Arbeit wichtige Fragestellungen beantwortet: Welche WAM-Regeln gibt es? Wie können die WAM-Regeln klassifiziert werden? Welche Regeln sind automatisch überprüfbar?

Abschnitt 3.1 stellt einige Regeln der WAM-Modellarchitektur vor. Diese sollen in den nachfolgenden Abschnitten als Beispiele dienen. Die Terminologie für die Regeln wird in Abschnitt 3.2 dargestellt. Um die Quellen und die Auswahl der Regeln geht es in den Abschnitten 3.3 und 3.4. Die Klassifizierung der Regeln spielt eine große Rolle für die Benutzbarkeit und damit Nützlichkeit der Regelsammlung und wird in Abschnitt 3.5 erläutert. Es folgen einige Anmerkungen zu den Regeln und dem verwendeten Schema in der Regelsammlung. Abschnitt 3.7 diskutiert, welche Regeln automatisch überprüfbar sind, worauf ein Ausschnitt aus der Regelsammlung folgt. Er enthält alle Regeln, die der Sotograph kontrollieren kann. Die vollständige Regelsammlung ist in Anhang A zu finden. Sie enthält alle Regeln, d.h. sowohl die, die der Sotograph überprüfen kann als auch die, die er nicht überprüfen kann.

3.1 Regelbeispiele

Die folgenden Regeln sind ein kleiner Ausschnitt der Regeln der WAM-Modellarchitektur. Sie werden im Laufe der folgenden Abschnitte als Beispiele für WAM-Regeln herangezogen, um die Schwierigkeiten und Besonderheiten bei der Auswahl und der Klassifizierung der Regeln zu verdeutlichen.

„Materialien dürfen keine FKs kennen“
„Die FK bearbeitet das Material“
„Zyklische Strukturen sollen vermieden werden“
„Kontext-IAKs kennen die volle Schnittstelle ihrer Sub-IAKs“
„Die Beziehung zwischen Kontext- und Sub-IAKs ist nicht immer vonnöten“
„Die Aufgaben der FK werden in einer Klasse gekapselt“
„Es muss möglich sein, zu fragen, ob ein Fachwert nur eine endliche Anzahl von Werten annehmen kann“
„Die Tool-Klasse erzeugt die FK“
„Werkzeuge sollten die Materialien, die sie bearbeiten unter einem Aspekt kennen“
„Die IAK hat im Regelfall keinen direkten Zugriff auf das Material. In einigen Fällen (z.B. um komplexe tabellarische Materialien zu handhaben), darf die IAK Lesezugriff auf ein Material haben“

3.2 Regelterminologie

In dieser Arbeit werden für unterschiedliche Arten von Regeln drei Begriffe verwendet, die in diesem Abschnitt kurz erläutert werden.

„Beziehungsregeln“ geben an, ob ein Element ein anderes Element kennen darf oder nicht und wie diese Beziehung gegebenenfalls aussieht. Ist eine Relation zwischen zwei Elementen vorgeschrieben, so heißt die Regel „Gebots-Beziehungsregel“. Ist eine Verbindung zwischen zwei Elementen verboten, so wird die Regel „Verbots-Beziehungsregel“ genannt.

Regeln, die nur für ein Element gelten, werden „Einzel-Element-Regeln“ genannt. Diese Regeln beziehen sich meist auf die Schnittstelle oder den Aufbau eines Elements.

3.3 Quelle der Regeln

Die WAM-Modellarchitektur ist in [Zül98] und der englischen Neuauflage [Zül04] beschrieben. Wo sich die Bücher entsprechen, wird nur auf [Zül98] verwiesen. [Zül98] ist die Hauptquelle für die Regeln der WAM-Modellarchitektur, die in dieser Arbeit gesammelt wurden. Viele der Regeln, die in der Auflistung in Kapitel 3.9 und in Anhang A zu finden sind, sind explizit in [Zül98] oder [Zül04] angegeben und haben in der Sammlung als Quellverweis [Zül98] oder [Zül04] samt Seitenzahl.

Alle anderen Regeln haben als Quellverweis „WAM-Architekten“. Diese Regeln wurden hauptsächlich aus der Erfahrung der Autorin dieser Arbeit mit der WAM-Modellarchitektur erstellt. Einige dieser Regeln sind implizit in [Zül98] zu finden. So z.B. die Regel *„Materialien dürfen keine FKs kennen“*. Sie kann aus den Regeln *„Die FK bearbeitet das Material“* und *„Zyklische Strukturen sollen vermieden werden“* abgeleitet werden. Um sicherzugehen, dass die aus Projekterfahrung erstellten Regeln für alle WAM-Systeme gelten und die abgeleiteten Regeln auch in der Praxis angewendet werden, wurden sie von erfahrenen WAM-Architekten überprüft. Diese waren z.T. auch an der Entwicklung des WAM-Ansatzes beteiligt. Aus der Diskussion mit den WAM-Architekten haben sich zusätzliche Regeln ergeben.

Im WAM-Ansatz lassen sich die Regeln auf der Ebene der Entwurfsmuster (s. Kapitel 2.3.2) einordnen. Die Regeln geben wie die Entwurfsmuster konkrete Konstruktionsanleitungen, um Systeme nach dem WAM-Ansatz zu entwickeln.

3.4 Kriterien für die Regelauswahl

Diese Arbeit beschränkt sich auf einen Teil der Regeln der WAM-Modellarchitektur. Es werden nur Regeln berücksichtigt, die sich auf die wichtigsten und am häufigsten benutzten WAM-Elemente beziehen. Dieses sind Materialien, Werkzeuge, Services, Automaten und Fachwerte. Im Folgenden wird begründet, warum Regeln für einige Elemente der WAM-Modellarchitektur nicht erfasst wurden.

[Zül98] beschreibt Regeln für die Entwurfsmetapher Arbeitsumgebung, die in der Auflistung nicht berücksichtigt wurden. Die Arbeitsumgebung hat u.a. die Aufgabe, das Zusammenspiel von Werkzeugen und Materialien zu koordinieren und ist daher ein elementarer Bestandteil eines WAM-Systems. Regeln für sie wurden jedoch nicht in die Sammlung aufgenommen. Ein Grund dafür liegt darin, dass die meisten WAM-Systeme mit dem Rahmenwerk JWAM entwickelt werden, das die Arbeitsumgebung bereits implementiert. Die Arbeitsumgebung besteht nach [Zül98] aus mehreren Elementen. Diese werden in der Praxis nicht unbedingt alle gebraucht oder sind durch andere Komponenten realisiert. So wird z.B. die Funktionalität des Materialverwalters durch Services abgedeckt. Ein Materialkoordinator wird nur im Mehrbenutzerbetrieb benötigt und auch diese Funktionalität wird in der Praxis von Services übernommen.

Ein weiteres Element der WAM-Modellarchitektur, für das es in der Regelsammlung keine expliziten Regeln gibt, ist der Behälter. Behälter sind Materialien. Alle Regeln für Materialien gelten daher auch für Behälter. Über die Schnittstelle von Behältern lässt sich kaum eine Aussage treffen, die als Regel verwertbar wäre, da *„die Schnittstelle einer fachlichen Behälterklasse wesentlich durch die fachlichen Umgangsformen mit diesem Behälter bestimmt wird."* [Zül98], S. 305. Diese fachlichen Umgangsformen sind von System zu System unterschiedlich. Behälter werden in der Praxis selten in Projekten eingesetzt. Stattdessen werden meist Services verwendet, die neben anderen Funktionen das Konzept der Behälter realisieren.

Es gibt weitere Regeln für die Modellarchitektur, die sich in [Zül98] oder [Zül04], aber nicht in der Regelsammlung finden lassen. Darunter fallen z.B. die Regeln für Formulare oder Interaktionstypen. Diese Elemente werden in der Praxis nicht sehr häufig oder gar nicht mehr eingesetzt. Die Regeln hätten daher die Regelsammlung vergrößert und unübersichtlich gemacht, ohne einen nennenswerten Nutzen zu bringen.

3.5 Klassifizierung der Regeln

Bei der Klassifizierung der Regeln gibt es viele Kategorisierungsmöglichkeiten. Im Folgenden wird zunächst auf verschiedene Klassifizierungsmöglichkeiten eingegangen, die verworfen wurden und dann beschrieben, welche Klassifizierung für die Regelsammlung dieser Arbeit gewählt wurde.

3.5.1 Verworfene Klassifizierungsmöglichkeiten

Es wäre möglich, die Regeln einzuteilen in Regeln, die Empfehlungen aussprechen, und Regeln, die strikt sind. Hierbei stellt sich die Frage ob Regeln, die Ausnahmen zulassen strikt sind oder nicht. Z.B. ist es in Ausnahmefällen erlaubt, dass eine IAK ein Material kennt. Die Einteilung nach „strikt" und „nicht strikt" hat außerdem den Nachteil, dass sie Regeln trennt, die inhaltlich zusammengehören, wie z.B. bei den Regeln: *„Kontext-IAKs kennen die volle Schnittstelle ihrer Sub-IAKs"* und *„Die Beziehung zwischen Kontext- und Sub-IAKs ist nicht immer vonnöten"*.

Eine andere Möglichkeit die Regeln aufzuteilen wäre, sie nach Verbotsregeln und Erlaubnisregeln zu ordnen. Die Einteilung wäre aber oft nur von der Formulierung abhängig. Die Regel *„Die Aufgaben der FK werden in einer Klasse gekapselt"* kann z.B. auch als *„Die Aufgaben der FK sollen nicht auf mehrere Klassen verteilt werden"* formuliert werden.

Weiterhin könnten die Regeln danach klassifiziert werden, ob sie mit dem Sotographen überprüft werden können oder nicht. Dies wäre aber keine Kategorie die von den Regeln, sondern von den Möglichkeiten des Sotographen abhängt. Da sich die Möglichkeiten des Sotographen ändern können, ist auch diese Einteilung nicht sinnvoll.

3.5.2 Gewählte Klassifizierung

Die Regeln werden in dieser Arbeit zunächst unterteilt nach den Elementtypen, auf die sie sich beziehen. D.h. es gibt Regeln für Automaten, Regeln für Werkzeuge usw. Daneben gibt es noch eine allgemeine Regel, die für alle Elemente gilt. Die Regeln für

jedes Element sind weiterhin unterteilt in Einzel-Element-Regeln, Verbots-Beziehungsregeln und Gebots-Beziehungsregeln. Bei Relationen zwischen Elementen handelt es sich immer um gerichtete Beziehungen, die nur von einem Element ausgehen können. Die Regeln für solche Beziehungen finden sich immer bei dem Element, von dem die Verbindung ausgeht oder nicht ausgehen darf.

Diese Klassifizierung ist sinnvoll, da die Einteilung nach Elementen die Suche nach Regeln erleichtert. Ist ein Architekt bei der Erstellung eines Architekturmodells in Zweifel wie die Schnittstelle eines Elements aussehen muss, so braucht er nur bei den Regeln für das entsprechende Element zu suchen. Ist er bei der Beziehung zwischen zwei Elementen in Zweifel, so muss er wiederum nur an einer Stelle nach einer Regel suchen, nämlich bei den Regeln für das Element von dem die Verbindung ausgeht.

Da sich die Regeln von Modellarchitekturen auf die Elemente und ihre Relationen untereinander beziehen, ist die Einteilung der Regeln nach Elementen und weiter nach den verschiedenen Beziehungsregeln und Einzel-Element-Regeln nicht nur sinnvoll, sondern ergibt sich aus der Definition für Softwarearchitekturen.

3.6 Aufbau der Regelsammlung

Die Regelsammlung listet die Regeln unter den entsprechenden Kategorien nach folgendem Schema auf:

Regel: …

> **Quelle:** …
> **Anmerkungen:** …
> **Überprüfbarkeit/Query:** …

Zu jeder Regel gibt es eine Quellenangabe. Die Quellenangabe besteht entweder aus [Zül98] oder [Zül04] mit Seitenzahl oder „WAM-Architekten“. Es gibt nicht zu jeder Regel Anmerkungen. Gibt es sie, so wird dort auf Ausnahmen zu dieser Regel hingewiesen oder es werden weiterführende Erklärungen gegeben, die für das Verständnis dieser Regel wichtig sind. Ob Regeln eingehalten werden, wird im Sotographen durch so genannte „Queries“ überprüft, die von der Autorin dieser Arbeit erstellt wurden. Sie werden in Kapitel 4.5 genauer beschrieben. Kann eine Regeln mit dem Sotographen überprüft werden, so verweist „Query“ auf die entsprechende Implementierung im Sotographen. In den Namen der Queries tauchen statt der deutschen Bezeichnungen FK, IAK, Fachwert und Automat die englischen Bezeichnungen FP (functional part), IP (interaction part), DV (domain value) und automaton auf. Bei Regeln, für die es keine Query gibt, wird unter „Überprüfbarkeit“ erläutert, warum für diese Regel nicht nach Verstößen gesucht wird. Es wird entweder dargestellt, warum es nicht sinnvoll schien, die Regel zu prüfen oder warum es technisch nicht möglich war, sie zu prüfen.

3.7 Überprüfbarkeit von Regeln

Dieser Abschnitt geht auf die Frage ein, welche Regeln automatisch überprüfbar sind. Oft sind die Gründe, weshalb eine Regel überprüfbar ist, von Regel zu Regel verschieden. Deshalb wird hier nur auf Gründe eingegangen wird, die auf ganze Gruppen von Regeln zutreffen. Details dazu, ob kontrolliert werden kann, ob ein System eine Regel einhält, finden sich in der Regelsammlung (s. Anhang A) bei jeder Regel.

Die meisten WAM-Regeln, deren Einhaltung geprüft werden kann, sind Verbots-Beziehungsregeln. Dies liegt daran, dass diese Regeln strikt sind und die beteiligten Elemente dieser Regeln meist identifizierbar sind. Weiterhin ist eine Verletzung solcher Regeln eindeutig definiert. Dürfen sich zwei Elemente nicht kennen, so ist keinerlei Beziehung, sei es Methodenaufruf, Aggregation oder Ähnliches, zwischen ihnen erlaubt. Ein Beispiel für eine Verbots-Beziehungsregel ist die Regel *„Materialien dürfen keine FKs kennen“*. Es gibt jedoch Regeln, bei denen der Aufwand für die Prüfung den Nutzen übersteigt. Verstöße gegen die Regel *„Fachwerte dürfen keine Events kennen“* könnten prinzipiell automatisch gefunden werden. Will man die Verletzungen mit dem Sotographen entdecken, so müsste die Eventklasse als Library mit eingelesen werden, damit sie dem Sotographen bekannt ist. Dieses wäre für die Prüfung einer Regel ein sehr hoher Aufwand.

Einige Gebots-Beziehungsregeln sind überprüfbar. Sie sind jedoch in der Regel schwieriger zu kontrollieren, da die Regeln meist für zwei bestimmte Elemente gelten. Eine IAK soll z.B. nicht irgendeine GUI, sondern die GUI des Werkzeugs kennen, zu der die IAK gehört. Dadurch ist die Identifizierung der Elemente schwieriger, für die eine Regel gilt.

Verstöße gegen Regeln, die sich mit dynamischen Aspekten von Beziehungen beschäftigen, sind im Regelfall mit einer statischen Analyse nicht auffindbar. Ein Beispiel für eine solche Regel ist *„Die FK benachrichtigt die IAK durch das Ereignismuster (oder Beobachtermuster) über Änderungen an der FK“*. Es gibt jedoch auch dynamische Regeln, die überprüft werden können. Darunter fällt z.B. die Regel *„Die Tool-Klasse erzeugt die FK“*. Hier kann getestet werden, ob die Tool-Klasse einen Konstruktor der zugehörigen FK aufruft.

Weiterhin gibt es einige Regeln, deren konkrete Ausprägung von einer fachlichen Interpretation abhängt. Diese Regeln beziehen sich nicht nur auf die Struktur des WAM-Systems, sondern auch auf die Semantik. Ein Beispiel für solch eine Regel ist die folgende: „*Die Aufgaben der FK werden in einer Klasse gekapselt*“. Um Verstöße gegen diese Regel zu finden, muss bekannt sein, welches die Aufgaben einer FK sind. Dies kann von FK zu FK verschieden sein und daher nicht in Form einer konkreten Regel ausgedrückt, sondern nur im Einzelfall entschieden werden.

Andere Regeln sind zwar eindeutig, jedoch ist die Umsetzung nicht immer gleich. So weist die Regel „*Es muss möglich sein zu fragen, ob ein Fachwert nur eine endliche Anzahl von Werten annehmen kann*“, inhaltlich keine Unklarheiten auf, die Implementierungen könnten jedoch verschiedener Art sein. Es bleibt nur die Möglichkeit über den Methodennamen auf die Semantik der Methode zu schließen. In JWAM gibt es Namenskonventionen für einige Fachwert-Methoden. Eine automatisierte Überprüfung, ob jeder Fachwert eine Methode dieses Namens hat, ist jedoch nur bedingt sinnvoll, da sich Namenskonventionen ändern können und die Überprüfung dann wertlos wäre. Solche Überprüfungen sollten also nur vorgenommen werden, wenn eine Konvention vorliegt, die sich mit großer Wahrscheinlichkeit nicht ändern wird. Beispiele hierfür sind die Benennung von „get“- und „set“-Methoden in Java oder die Bezeichnungen „IAK“ und „FK“ in JWAM.

Neben mehreren Regeln dieser Art für Fachwerte gibt es weitere Regeln, für die Verstöße nicht festgestellt werden können, weil die Erkennung der Artefakte, um die es geht, nicht möglich ist. So kann z.B. technisch nicht zwischen FKs und Sub-FKs un-

terschieden werden, da eine Unterscheidung nur zur Laufzeit gegeben ist und sich oft nur auf Objekte bezieht und nicht unbedingt auf Klassen.

Regeln, die nur Vorgaben machen, aber nicht strikt sind, eignen sich verständlicherweise nicht für die Überprüfung. Alle Ergebnisse der Kontrolle müsste der Benutzer von Hand auswerten, nur um vielleicht festzustellen, dass keine echte Verletzung vorliegt. Die Regel *„Werkzeuge sollten die Materialien, die sie bearbeiten, unter einem Aspekt kennen“* ist solch eine Regel. Sie macht eine Vorgabe, muss aber nicht zwingend eingehalten werden.

Dahingegen kann bei Regeln, die Ausnahmen definieren, überprüft werden, ob sie eingehalten werden. Sie unterscheiden sich von den Regeln, die Vorgaben machen, darin, dass die Ausnahmen definiert sind. Dies ist z.B. bei dieser Regel der Fall: *„Die IAK hat im Regelfall keinen direkten Zugriff auf das Material. In einigen Fällen (z.B. um komplexe tabellarische Materialien zu handhaben), darf die IAK Lesezugriff auf ein Material haben“*. Ob eine gefundene Verletzung einer Regel eine Ausnahme ist, muss der Benutzer von Hand prüfen. Der Benutzer kann aber davon ausgehen, dass der Großteil der Ergebnisse echte Verstöße gegen die Regel sind.

Einige Regeln wären prinzipiell automatisch prüfbar, können aber mit den Möglichkeiten des Sotographen nicht überprüft werden. Darunter fällt z.B. die Regel *„Die innere Konsistenz eines Materials wird über das Vertragsmodell durch Zusicherung an den verändernden Operationen einer Klasse abgesichert“*. Um Verstöße gegen diese Regel zu finden, müsste untersucht werden, ob in den Methoden eines Materials „asserts“ verwendet werden. D.h. es müsste im Quelltext nach dem Schlüsselwort „assert“ gesucht werden. Da mit dem Sotographen nicht auf Methodenrümpfe zugegriffen werden kann, kann er diese Regel nicht überprüfen.

Zusammenfassend kann über das automatische Finden von Regelverletzungen gesagt werden, dass Verbots-Beziehungsregeln oft überprüfbar sind. Verstöße gegen Gebots-Beziehungsregeln sind nicht so oft erkennbar. Am seltensten sind jedoch Verletzungen von Regeln auffindbar, die sich auf Schnittstellen beziehen. Nicht entdeckt werden kann, ob Regeln befolgt werden, die sich auf dynamische Aspekte beziehen, Regeln, die fachlich interpretiert werden müssen, und Regeln, deren Umsetzung im Code unterschiedlich sein kann. Hierbei handelt es sich um die allgemeine Möglichkeit der automatischen Überprüfbarkeit. Bezieht man die Möglichkeiten des Sotographen in die Überlegungen mit ein, so kann festgestellt werden, dass auch Regeln, deren Umsetzung nur innerhalb von Methodenrümpfen zu finden ist, nicht kontrolliert werden können.

Tabelle 3.1 zeigt eine Übersicht über die Überprüfbarkeit der WAM-Regeln. Der Großteil der Regeln kann unterschieden werden in prüfbare und nicht prüfbare Regeln. Es gibt jedoch Sonderfälle: Für einige Regeln ist es nicht sinnvoll, nach Verletzungen zu suchen, da sie nur Vorgaben machen. Bei anderen Regeln ist es zu aufwändig, Verstöße zu finden oder sie werden indirekt durch die Überprüfung einer anderen Regel gefunden.

Die Tabelle zeigt, dass es keine Verbots-Beziehungsregeln gibt, die nicht kontrolliert werden können. Es gibt etwas mehr nicht überprüfbare Gebotsbeziehungsregeln als prüfbare. Von den Einzel-Element-Regeln können nur sehr wenige auf Verletzungen untersucht werden. Insgesamt können für über die Hälfte der Regeln, deren Prüfung sinnvoll ist, automatisch Verstöße gefunden werden.

	Prüfbar	Nicht prüfbar	Prüfung nicht sinnvoll	Prüfung zu aufwändig	Indirekt geprüft	Gesamt
Einzel-Element-Regeln	6	19	1	0	0	26
Verbots-Beziehungsregeln	27	0	0	4	2	33
Gebots-Beziehungsregeln	12	16	6	0	1	35
Gesamt	45	35	7	4	3	94

Tabelle 3.1: Überprüfbarkeit der WAM-Regeln

3.8 Allgemeine Erläuterungen zu den Regeln

Werden in den Regeln die Elemente „FK“ (Funktionskomponente), „IAK“ (Interaktionskomponente) und „Tool-Klasse“ erwähnt, so sind nur die Elemente gemeint, die jeweils aus einer Klasse bestehen. Monotool-Klassen sind damit nicht gemeint.

Sind in den Regeln Kontext-FKs oder Sub-FKs erwähnt, so können dies auch Kontext-Monotool-Klassen bzw. Sub-Monotool-Klassen sein.

3.9 Die Regeln

In diesem Abschnitt sind alle Regeln aufgelistet, für die mit Hilfe von Queries im Sotographen überprüft werden kann, ob sie eingehalten werden. Die Gliederung entspricht der Klassifizierung, die in Kapitel 3.5.2 beschrieben wurde.

Die Beziehungsregeln der folgenden Abschnitte werden in Kapitel 3.9.12 in zwei Abbildungen zusammengefasst.

3.9.1 Allgemeine WAM-Regeln

Regel: *Zyklische Strukturen sollen vermieden werden. Müssen Strukturen wechselseitig verknüpft werden, so sollte in einer Richtung eine lose Kopplung verwendet werden.*

Quelle: [Zül98], S. 344
Überprüfbarkeit: Diese Regel ist mit den Metriken „ClassCyclicRefClass“ und „PckgCyclicRefPckg“ des Sotographen überprüfbar, die nach Zyklen zwischen Klassen bzw. Packages suchen. Diese Metriken gehören zur Grundausstattung des Sotographen. Zyklen zwischen Gruppen von Packages, so genannten „Subsystemen“, können mit der Query „SubsysCyclicRefSubsys“ gefunden werden. Hierzu müssen zunächst die Subsysteme definiert werden. Es wurde noch nicht untersucht, ob es Subsystemmodelle gibt, die für alle WAM-Systeme gelten. Für jedes WAM-System müsste daher ein eigenes Subsystemmodell erstellt werden. Aus diesem Grund werden Subsystemzyklen bei der Analyse der Beispielsysteme nicht berücksichtigt.

3.9.2 Regeln für Materialien

Einzel-Element-Regeln

Regel: *Materialien sollten nicht nur generische Operationen definieren.*

Quelle: [Zül98], S. 170
Anmerkungen: Materialien, die nur generische Operationen definieren, degenerieren dadurch zu einem Wertebehälter und die fachliche Funktionalität wandert in das Werkzeug.
Query: „Materials Shouldn't Have "get"- And "set"-Methods Only"

Verbots-Beziehungsregeln

Regel: *Materialien dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Der Begriff „Werkzeuge" umfasst sowohl komplexe als auch monolithische Werkzeuge. Materialien dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen.
Query: „Materials Shouldn't Know Tools"

Regel: *Materialien dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: „Materials Shouldn't Know Services"

Regel: *Materialien dürfen keine Automaten kennen.*

Quelle: WAM-Architekten
Query: „Materials Shouldn't Know Automatons"

3.9.3 Regeln für Werkzeuge (allgemein)

Einzel-Element-Regeln

Regel: *Ein Werkzeug sollte grundsätzlich in Funktionskomponente (FK) und Interaktionskomponente (IAK) aufgeteilt werden.*

Quelle: [Zül98], S. 236
Anmerkungen: Zu jeder FK sollte es eine IAK geben und zu jeder IAK eine FK. Es gibt aber auch Konstruktionsansätze, bei denen ein Subwerkzeug nur aus FK besteht und die Interaktion durch die Kontext-IAK implementiert wird. Diese Regel gilt nicht für monolithische Werkzeuge, bei denen die Funktionen von FK und IAK in einer Klasse vereinigt werden.
Query: „Complex Tools Consist Of GUI, FP, IP and Tool-class"

Regel: *Jedes eigenständige Werkzeug soll gegenüber dem Kontext durch eine Tool-Klasse abgeschlossen sein.*

Quelle:[Zül98], 276/289/290
Anmerkungen: D.h. Subwerkzeuge müssen nicht unbedingt eine Tool-Klasse besitzen. In der Praxis haben aber auch Subwerkzeuge immer eine Tool-Klasse, da Subwerkzeuge in JWAM nur über die Tool-Klasse gestartet werden können. Hier unterscheidet sich also die Implementierung von JWAM von der WAM-Modellarchitektur wie sie in [Zül98] beschrieben ist. Die Regel bezieht sich nur auf Werkzeuge, die aus FK und IAK zusammengesetzt sind.
Query: „Complex Tools Consist of GUI, FP, IP and Tool-class"

3.9.4 Regeln für Funktionskomponenten

Einzel-Element-Regeln

Regel: *Die FK bietet über eine materialunabhängige Schnittstelle Informationen zur Präsentation des Materials für die IAK an.*

Quelle: [Zül98], S. 240
Query: „FPs Shouldn't Return Materials"

Verbots-Beziehungsregeln

Regel: *Die FK soll möglichst keinerlei Kenntnis von ihrer IAK haben.*

Quelle: [Zül98], S. 236
Anmerkungen: Auch wenn in der Regel „möglichst" steht, so ist es in der Praxis immer der Fall, dass die FK ihre IAK nicht direkt kennt.
Query: „FPs Shouldn't Know IPs"

Regel: *FKs dürfen keine GUIs kennen.*

Quelle: WAM-Architekten
Query: „FPs Shouldn't Know GUIs"

Regel: *FKs dürfen keine Tool-Klassen kennen.*

Quelle: WAM-Architekten
Anmerkungen: Eine Ausnahme ist die Tool-Klasse eines Subwerkzeugs. In JWAM 2 kennt eine FK ihre Toolklasse unter der Tool-Schnittstelle. Dies verstößt gegen die Regel. Auf Nachfrage bei dem für den„ToolConstruction"-Teil von JWAM zuständigen Architekten stellte sich diese Implementierung als Fehler heraus, der möglichst bald behoben werden sollte. Eine FK sollte ihre eigene Tool-Klasse höchstens als Request-Handler kennen.
Query: „FPs Shouldn't Know Tool-classes".

Gebots-Beziehungsregeln

Regel: *Die FK bearbeitet das Material.*

Quelle: [Zül98], S.236
Anmerkungen: Die Bearbeitung von Materialien ist der Hauptzweck von Werkzeugen. Daneben rufen sie evtl. noch Automaten und/oder Services.
Query: „FPs Should Know Materials"

3.9.5 Regeln für Interaktionskomponenten

Verbots-Beziehungsregeln

Regel: *Die IAK hat im Regelfall keinen direkten Zugriff auf das Material. In einigen Fällen (z.B. um komplexe tabellarische Materialien zu handhaben), darf die IAK Lesezugriff auf ein Material haben.*

Quelle: [Zül98], S. 240 und [Zül04], S. 226
Anmerkungen: Ursprünglich kommt die Regel aus [Zül98], die Ausnahme kommt aus [Zül04]
Query: „IPs Shouldn't Know Materials".

Regel: *IAKs dürfen keine Tool-Klasse kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Tool-classes"

Regel: *IAKs dürfen keine Monotool-Klasse kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Monotool-classes"

Regel: *IAKs dürfen keine Automaten kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Automatons"

Regel: *IAKs dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Services"

Gebots-Beziehungsregeln

Regel: *Die IAK steuert die Präsentation an der Oberfläche.*

Quelle: [Zül98], S. 236
Anmerkungen: Mit „Oberfläche" ist hier die GUI gemeint. D.h. die IAK kennt die GUI unter ihrer vollen Schnittstelle und ruft Methoden an der GUI.
Query: „IPs Should Know Their GUI"

Regel: *Die IAK kennt ihre FK unter ihrer vollen Schnittstelle.*

Quelle: [Zül98], S. 246/247
Query: „IPs Should Know Their FP"

3.9.6 Regeln für Tool-Klassen

Verbots-Beziehungsregeln

Regel: *Tool-Klassen dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: "Tool-classes Shouldn't Know Services"

Regel: *Tool-Klassen dürfen keine Automaten kennen.*

Quelle: WAM-Architekten
Query: "Tool-classes Shouldn't Know Automatons"

Gebots-Beziehungsregeln

Regel: *Die Tool-Klasse erzeugt die FK.*

Quelle: [Zül98], S. 290
Anmerkungen: Die Regel bezieht sich ursprünglich nur auf Kontext-FKs, in der Regel haben aber auch Subwerkzeuge eine Tool-Klasse, die die IAK und FK erzeugt.
Query: „The Tool-class Should Initialize The FP"

Regel: *Die Tool-Klasse erzeugt die IAK.*

Quelle: [Zül98], S. 290
Anmerkungen: Die Regel bezieht sich ursprünglich nur auf Kontext-IAKs, in der Regel haben aber auch Subwerkzeuge eine Tool-Klasse, die die IAK und FK erzeugt.
Query: „The Tool-class Should Initialize The IP"

Regel: *Die Tool-Klasse erzeugt die GUI.*

Quelle: WAM-Architekten
Query: „The Tool-class Should Initialize The GUI"

3.9.7 Regeln für GUIs

Verbots-Beziehungsregeln

Regel: *GUIs dürfen keine Materialien kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Materials"

Regel: *GUIs dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Services"

Regel: *GUIs dürfen keine Automaten kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Automatons"

Regel: *GUIs dürfen keine FKs kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Tools"

Regel: *GUIs dürfen keine Tool-Klassen kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Tools".

Regel: *GUIs dürfen keine Monotool-Klassen kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Tools".

Regel: *Die GUI sollte ihre IAK nicht direkt kennen.*

Quelle: WAM-Architekten
Anmerkungen: Die Kopplung von der GUI zur IAK ist in Java mit dem Beobachtermuster lösbar (Listener). Die Kopplung ist aber vom Toolkit abhängig. In WAM gibt es dazu auch eine Lösung mit Präsentationsformen, die das Befehlsmuster verwendet.
Query: „GUIs Shouldn't Know Tools". Es kann allerdings nur geprüft werden, ob die GUI ihre IAK direkt kennt. Eine indirekte Kopplung ist nicht erkennbar. Bei einer Kopplung über das Befehlsmuster könnte nur geprüft werden, ob die IAK Befehle kennt und ob es Befehle gibt, die die GUI kennen. Damit wäre aber nicht sichergestellt, dass es eine Kopplung gibt.

3.9.8 Regeln für Monotool-Klassen

Verbots-Beziehungsregeln

Regel: *Monotool-Klassen dürfen keine IAKs kennen.*

Quelle: WAM-Architekten
Query: „Monotool-classes Shouldn't Know IPs"

Regel: *Monotool-Klassen dürfen keine Tool-Klassen kennen.*

Quelle: WAM-Architekten
Anmerkungen: Ausnahmen sind Tool-Klassen von Subwerkzeugen.
Query: „Monotool-classes Shouldn't Know Tool-classes"

Gebots-Beziehungsregeln

Regel: *Die Monotool-Klasse steuert die Präsentation an der Oberfläche.*

Quelle: WAM-Architekten
Anmerkungen: Mit „Oberfläche“ ist hier die GUI gemeint. D.h. die Monotool-Klasse kennt die GUI unter ihrer vollen Schnittstelle und ruft Methoden an der GUI.
Query: „Monotool-classes Should Know Their GUI“

Regel: *Monotool-Klassen erzeugen sich ihre GUI selbst.*

Quelle: WAM-Architekten
Anmerkungen: Da monolithische Werkzeuge keine Tool-Klasse besitzen, müssen sie ihre GUI selber erzeugen.
Query: „The Monotool-class Should Initialize The GUI“

Regel: *Die Monotool-Klasse bearbeitet das Material.*

Quelle: WAM-Architekten
Erläuterungen: Die Bearbeitung von Materialien ist der Hauptzweck von Werkzeugen. Daneben rufen sie evtl. noch Automaten und/oder Services.
Query: „Monotool-classes Should Know Materials“

3.9.9 Regeln für Automaten

Verbots-Beziehungsregeln

Regel: *Automaten dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Mit dem Begriff „Werkzeuge“ sind sowohl komplexe als auch monolithische Werkzeuge gemeint, Automaten dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen. Nur in bestimmten Fällen dürfen Automaten FKs kennen. Diese Ausnahmen sind in [Zül98] S.197/198 beschrieben. Sie sind in der Praxis allerdings noch nicht aufgetreten.
Query: „Automatons Shouldn’t Know Tools“

Gebots-Beziehungsregeln

Regel: *Ein Automat arbeitet wie ein Werkzeug auf Materialien.*

Quelle: [Zül98] S. 195
Query: „Automatons Should Know Materials“

3.9.10 Regeln für Services

Verbots-Beziehungsregeln

Regel: *Services dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Mit dem Begriff „Werkzeuge“ sind sowohl komplexe als auch monolithische Werkzeuge gemeint, Services dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen.
Query: „Services Shouldn’t Know Tools“

Gebots-Beziehungsregeln

Regel: *Services gehen mit Materialien um.*

Quelle: [Zül04], S. 265
Query: „Services Should Know Materials“

3.9.11 Regeln für Fachwerte

Einzel-Element-Regeln

Regel: *Der Konstruktor der Fachwert-Klasse darf nicht öffentlich sein.*

Quelle: [Zül98], S. 319
Query: „DVs Shouldn't Have a Public Constructor"

Regel: *Fachwert-Klassen bieten keine Operationen an, die es erlauben, das Fachwert-Objekt zu verändern.*

Quelle: [Zül98], S. 319/320
Anmerkungen: D.h. ein Fachwert darf keine Set-Methoden haben.
Query: „DVs Shouldn't Have "set"-Methods". Ob gegen diese Regel verstoßen wird, kann nur festgestellt werden, wenn verändernde Operationen den String "set" im Namen haben. Da diese Bezeichnung meistens eingehalten wird, macht es Sinn, mit einer Query zu überprüfen, ob diese Regel eingehalten wird.

Verbots-Beziehungsregeln

Regel: *Fachwerte dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Mit dem Begriff „Werkzeuge“ sind sowohl komplexe als auch monolithische Werkzeuge gemeint, Fachwerte dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen.
Query: „DVs Shouldn't Know Tools“

Regel: *Fachwerte dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: „DVs Shouldn't Know Services“

Regel: *Fachwerte dürfen keine Materialien kennen.*

Quelle: WAM-Architekten
Query: „DVs Shouldn't Know Materials“

Regel: *Fachwerte dürfen keine Automaten kennen.*

Quelle: WAM-Architekten
Query: „DVs Shouldn't Know Automatons“

3.9.12 Abbildungen für Beziehungsregeln

Die folgenden Abbildungen zeigen die Beziehungen, die zwischen WAM-Elementen gelten müssen.

In Abbildung 3.1 sind alle Relationen zwischen WAM-Elementen dargestellt, die verboten sind. Aus Gründen der Übersichtlichkeit wurden ein Teil der Pfeile weggelassen, diese verbotenen Beziehungen sind als Anmerkungen in der Graphik zu finden.

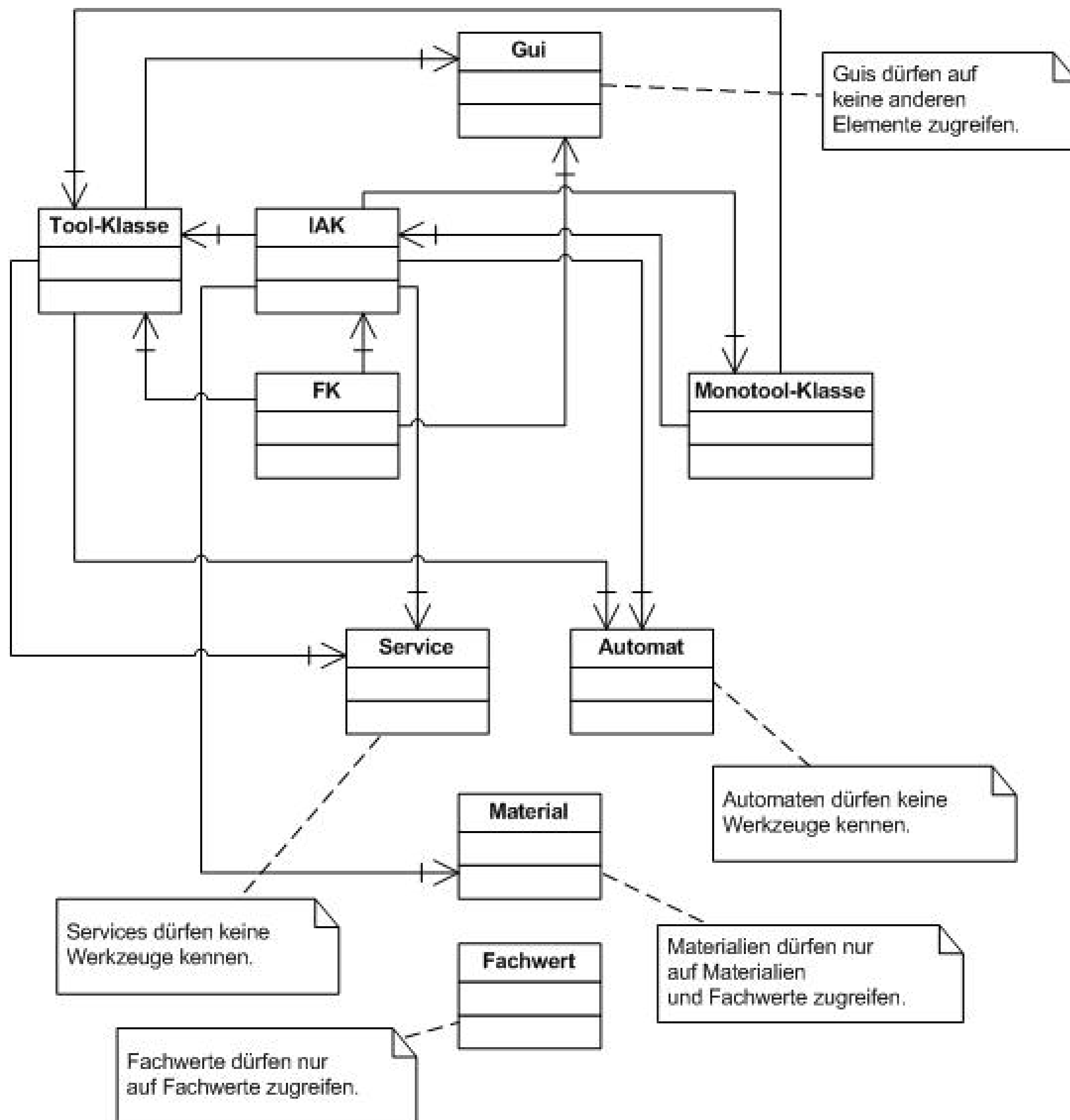

Abbildung 3.1: Verbotene Beziehungen zwischen WAM-Elementen

Abbildung 3.2 bildet die Verbindungen zwischen WAM-Elementen ab, die in einem WAM-System vorhanden sein sollten. Es handelt sich dabei im Regelfall um Benutzt-Beziehungen, die durch Regeln vorgeschrieben werden. So arbeitet ein Service im Normalfall auf Materialien und sollte sie daher kennen und benutzen. Eine Ausnahme bilden die Beziehungen von der Tool-Klasse zu GUI, IAK und FK. Die Regeln schreiben lediglich vor, dass die Tool-Klasse die genannten WAM-Elemente erzeugt, eine weitere Benutzt-Beziehung muss jedoch nicht vorhanden sein.

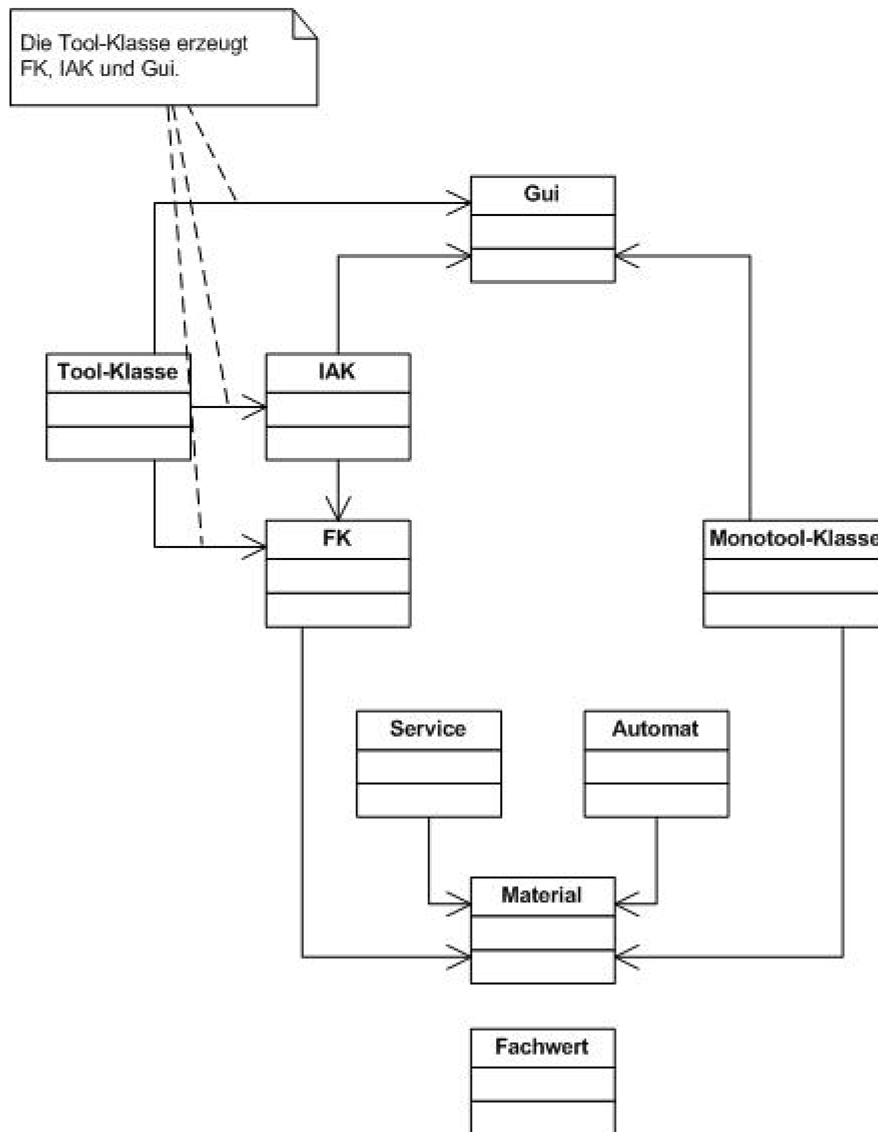

Abbildung 3.2: Vorgeschriebene Beziehungen zwischen WAM-Elementen

3.10 Zusammenfassung

In diesem Kapitel wurden die Regeln der WAM-Modellarchitektur selektiert und klassifiziert. Diese Regeln wurden in einer Regelsammlung zusammengestellt, die in Anhang A zu finden ist.

Es wurde festgestellt, dass es verschiedene Arten von Regeln gibt, von denen einige automatisch kontrollierbar sind, während andere dies nicht sind. Die WAM-Regeln, für die automatisch Verletzungen gefunden werden können, wurden als Ausschnitt der Regelsammlung in Kapitel 3.9 vorgestellt. Damit wurde die Grundlage für eine automatische Überprüfung der Regeln durch den Sotographen geschaffen. Als Nächstes müssen nun die Möglichkeiten der praktischen Umsetzung im Sotographen untersucht werden.

Kapitel 4 Implementierung der Regeln

Nachdem im letzten Kapitel die Erstellung der Regelsammlung für die WAM-Modellarchitektur im Vordergrund stand, geht es in diesem Kapitel darum, die Regeln überprüfbar zu machen. Abschnitt 4.1 diskutiert zunächst, wie WAM-Elemente im Quelltext identifiziert werden können. Danach wird darauf eingegangen, wie der Sotograph Verstöße gegen Regeln finden kann. Dazu wird erläutert, wie Queries im Sotographen erstellt werden. Anschließend werden die Queries vorgestellt, die implementiert wurden, um zu testen, ob die Regeln der WAM-Modellarchitektur befolgt werden.

Die in Kapitel 3.9 aufgeführten Regeln sind strikt. Entwickler können sich aber dafür entscheiden, nicht alle Regeln bei der Entwicklung eines Systems einzuhalten. Die in diesem Kapitel vorgestellten Queries suchen nach Verstößen gegen alle Regeln, die in Kapitel 3.9 aufgelistet wurden. Die Interpretation der Ergebnisse ist allerdings den Entwicklern überlassen. Wurde die Entscheidung getroffen, eine Regel nicht zu befolgen, so muss der Entwickler die Ergebnisse der Query, die nach Verstößen gegen die Regel sucht, entsprechend interpretieren. Ihre Ergebnisse dokumentieren dann nur die Abweichung von der WAM-Modellarchitektur.

4.1 Identifikation von Elementen

Um mit Queries zu überprüfen, ob die Regeln der WAM-Modellarchitektur eingehalten werden, ist es notwendig, die Elemente der WAM-Modellarchitektur im Quelltext zu finden. Hierfür gibt es mehrere Möglichkeiten, die im folgenden Abschnitt erläutert werden. Danach wird die gewählte Identifizierungsart begründet und erläutert, wie die Elemente mit Hilfe von JWAM erkannt werden können.

4.1.1 Möglichkeiten der Identifikation

WAM-Elemente könnten über ihr Package gefunden werden. Dazu müssten WAM-Elemente in Packages liegen, deren Namen einer festgelegten Namenskonvention folgen. Beispielsweise müssten alle Automaten in einem Package liegen, dessen Name den String „automat“ enthält. Diese Lösung ist theoretisch möglich. Da es solche Packagestrukturen jedoch nur in den seltensten Fällen gibt, ist sie nicht praktikabel.

Eine weitere Möglichkeit, Elemente zu identifizieren, besteht darin, sie anhand ihres Klassennamens einer Elementart zuzuordnen. Abgesehen von Materialien gibt es für jedes der WAM-Elemente, für die es Regeln in der in Anhang A aufgeführten Regelsammlung gibt, eine Namenskonvention. Z.B. sollten alle Klassennamen von Interaktionskomponenten (engl. interaction part) ein „IAK“ oder ein „IP“ im Namen haben, vorzugsweise am Ende des Namens. Da diese Namenskonventionen in den meisten WAM-Systemen gut eingehalten werden, ist dies eine Möglichkeit, WAM-Elemente zu finden, die auch praktisch umsetzbar ist. Probleme gibt es allerdings mit Tool-Klassen und Monotool-Klassen. Beide haben meist nur ein „Tool“ im Namen, nur selten sind die Monotools mit „Monotool“ gekennzeichnet. Verletzungen der Regeln für diese Elemente könnten also in den meisten Systemen nicht entdeckt werden. Da sie aber als Werkzeugelemente erkannt werden können, könnte zumindest überprüft werden, ob Regeln wie *„Fachwerte dürfen keine Werkzeuge kennen“* befolgt werden. Nur Materialien können über ihren Namen überhaupt nicht gefunden werden. Auf die verschiedenen Möglichkeiten, Materialien zu finden, wird in Kapitel 4.2 eingegangen.

Als Drittes gibt es die Möglichkeit, das JWAM-Rahmenwerk für die Identifizierung der Elemente einzubeziehen. Das Rahmenwerk stellt Interfaces und Klassen bereit, die genutzt werden können und sollen, um WAM-Elemente zu implementieren. Für die meisten Elemente gibt es JWAM-Klassen, in denen die Grundfunktionen dieser Elemente umgesetzt sind. Werden diese nicht verwendet, so müssen zumindest Interfaces implementiert werden, um eine Klasse als ein bestimmtes WAM-Element zu markieren. Über diese Vererbungs- und Implementierungsbeziehungen kann eine Klasse als ein WAM-Element erkannt werden. Materialien können allerdings mit dieser Methode nicht identifiziert werden. Außerdem können GUIs nicht in allen JWAM-Versionen gefunden werden.

Eine andere Idee besteht darin, den Sotographen so zu erweitern, dass der Benutzer die WAM-Elemente manuell identifiziert bevor das System eingelesen wird. Dies hätte mehrere Vorteile: Zum Einen würden alle WAM-Elemente eindeutig erkannt, zum Anderen könnten WAM-Elemente auch in Systemen gefunden werden, die kein JWAM und keine Namenskonventionen verwenden. Der dritte Vorteil besteht darin, dass das Rahmenwerk JWAM nicht als library in den Sotographen eingelesen werden muss, wie es bei der Erkennung über Vererbungsbeziehungen zu JWAM der Fall ist. Einer der Nachteile dieser Lösung ist, dass Systeme nur mit Hilfe eines Entwicklers untersucht werden könnten, da dieser die Elemente markieren müsste. Es wäre aber sinnvoll eine Analyse auch ohne Entwickler durchführen zu können. Der zweite Nachteil dieser Lösung ist, dass sie für den Benutzer sehr zeitaufwändig ist, da jedes WAM-Element einzeln zugeordnet werden müsste. Die praktische Umsetzung dieser Idee ist im Moment nicht praktikabel, da es im Sotographen noch kein Konzept gibt, um Klassen Architekturelementen zuzuordnen. Die Erweiterung des Sotographen daraufhin wäre nicht unerheblich und könnte nur von den Entwicklern des Sotographen geleistet werden. Es wäre zudem eine Unterstützung notwendig, damit die WAM-Elemente nicht in jeder neuen Version des Systems erneut zugeordnet werden müssten. Dieser Aufwand, zusammen mit den Nachteilen, die diese Art der Zuordnung mit sich bringt, macht die Umsetzung dieser Idee nicht sinnvoll.

4.1.2 Begründung der gewählten Identifikation

In dieser Arbeit werden die Vererbungsbeziehungen zu den Klassen und Interfaces aus dem JWAM-Rahmenwerk genutzt, um WAM-Elemente in einem System zu identifizieren. Wird das Rahmenwerk verwendet, um ein WAM-System zu implementieren, so kann davon ausgegangen werden, dass die vorgesehenen Vererbungsbeziehungen für alle Elemente eingehalten werden und die Elemente darüber erkannt werden können.

Elemente über Vererbungsbeziehungen zu finden ist zuverlässiger als die Erkennung über Namenskonventionen. Letztere ist unzuverlässig, wenn es Klassen gibt, die zufällig einen String im Namen haben, der eine Elementart bezeichnet. So gibt es z.B. im Pausenplanersystem, das in Kapitel 5 untersucht wird, eine Klasse `IPausenplanService`[1]. Das Kürzel „IP“ steht im WAM-Kontext für eine Interaktionskomponente, „Service“ für einen Service. Die Klasse würde also als Service, fälschlich aber auch als Interaktionskomponente identifiziert werden. Über ihre Vererbungsbeziehung würde sie eindeutig als Service gekennzeichnet sein, da sie das Interface `Ser-`

[1] Klassennamen, Elemente der Beispielsysteme und Quelltextauszüge werden in dieser Schriftart dargestellt.

`viceProvider` implementiert. Ein weiterer Grund, die Vererbungsbeziehungen zur Identifizierung zu nutzen, ist, dass so auch Tool-Klassen und Monotool-Klassen voneinander unterschieden werden können, da die Vererbungsbeziehungen eindeutig sind, die Namen jedoch nicht. Der Nachteil dieser Lösung ist, dass Materialien nicht über ihre Vererbungsbeziehungen allein, und GUIs in JWAM 2 nicht über Vererbung erkannt werden können. Um GUIs zu finden muss daher auf Namenskonventionen zurückgegriffen werden. Materialien werden über eine Mischung aus Vererbungsbeziehungen und Namenskonventionen entdeckt.

4.1.3 Verwendung von JWAM zur Identifikation

Wie in Abschnitt 2.3.2 beschrieben, ist JWAM ein Beispiel für die dritte Ebene des WAM-Ansatzes, der Implementierungsebene. Diese Ebene der Modellarchitektur wird benötigt, um die Regeln der 2. Ebene zu überprüfen.

Zurzeit wird in einigen Projekten JWAM 2 eingesetzt, viele Projekte arbeiten mit JWAM 1.8. Aus diesem Grund sollten die Queries sowohl Systeme untersuchen können, die JWAM 1.8 verwenden als auch Systeme, die JWAM 2 verwenden. Bei der Identifikation der Elemente mussten daher die Unterschiede zwischen Version 1.8 und 2 des JWAM-Rahmenwerks berücksichtigt werden.

Im Folgenden werden die Vererbungsbeziehungen aufgeführt, anhand derer die WAM-Elemente erkannt werden können. Dabei sind Klassen oder Interfaces angegeben, die in der Vererbungshierarchie eines Elements auftauchen müssen, und Klassen oder Interfaces, die nicht vorhanden sein dürfen. Letzteres ist notwendig, um einige Elemente, z.B. Tool-Klassen und FKs, voneinander abzugrenzen. Nur die Hälfte der betrachteten Elemente haben in JWAM 1.8 und in JWAM 2 die gleichen Vererbungsbeziehungen. Die anderen Elemente müssen bei der Benutzung verschiedener JWAM-Versionen über unterschiedliche Vererbungsbeziehungen gefunden werden. Zu den Vererbungsbeziehungen sind daher ihre Geltungsbereiche angegeben.

Vererbungsbeziehungen (gelten für JWAM 1.8 und JWAM 2):

- IAKs implementieren das Interface `ToolInteraction`.
- FKs implementieren das Interface `ToolFunctionality` und nicht das Interface `Tool`.
- Automaten erben von der Klasse `Automaton`.
- Fachwerte implementieren das Interface `DomainValue`.

Vererbungsbeziehungen (gelten nur für JWAM 1.8):

- Monotool-Klassen implementieren das Interface `ToolFunctionality` und das Interface `Tool`.
- Tool-Klassen implementieren das Interface `Tool` und implementieren nicht das Interface `ToolFunctionality`.
- GUIs implementieren das Interface `ToolView`.
- Services implementieren das Interface `ServiceProvider`.

Vererbungsbeziehungen (gelten nur für JWAM 2):

- Monotool-Klassen erben von der abstrakten Klasse `AbstractToolMono`.

- Tool-Klassen implementieren das Interface `Tool`, implementieren nicht das Interface `ToolFunctionality` und erben nicht von der abstrakten Klasse `AbstractToolMono`.
- Services implementieren das Interface `Service`.

Die Queries, die IAKs, FKs, Automaten und Fachwerte identifizieren, mussten nur in einer Form implementiert werden, da die Vererbungsbeziehungen in JWAM 1.8 und JWAM 2 gleich sind. Die Queries für Monotool-Klassen und Services hingegen wurden jeweils für JWAM 1.8 und für JWAM 2 implementiert, da die Vererbungsbeziehungen in den beiden JWAM-Versionen unterschiedlich sind. Die Queries für Tool-Klassen wurden nur einmal implementiert, da die Vererbungsbeziehung in JWAM 2 eine Erweiterung der Vererbungsbeziehung in JWAM 1.8 ist. Die Vererbungsbeziehung in JWAM 2 identifiziert daher Tool-Klassen in Systemen, die JWAM 1.8 benutzen, und in Systemen, die JWAM 2 benutzen. GUIs konnten nur in Systemen, die JWAM 1.8 benutzen über Vererbungsbeziehungen identifiziert werden.

Um die Beschreibungen der Queries nicht unnötig kompliziert zu machen, wird in den folgenden Kapiteln nur so viel wie nötig auf die unterschiedlichen Queries für JWAM 1.8 und JWAM 2 eingegangen. Die Unterschiede zwischen den JWAM-Versionen spielen nur bei der Identifikation von WAM-Elementen eine Rolle. Für die Überprüfung einer Regel sind die Unterschiede irrelevant.

Um ein Element zu identifizieren, ist es nicht wichtig, ob die Klassen in einer direkten Vererbungs- oder Implementierungsbeziehung stehen. Die Superklassen oder Interfaces müssen nur in der Hierarchie der Superklassen der Elemente auftauchen. Der Sotograph unterscheidet nicht, ob von einer Klasse geerbt wird, oder ein Interface implementiert wird.

Für JWAM 1.8 gibt es eine Vererbungsbeziehung, die GUIs eindeutig identifiziert. In JWAM 2 gibt es jedoch keine Superklasse und kein Interface für GUIs. Sie sollen bei Benutzung von JWAM 2 nur noch von der Klasse `JPanel` aus dem JDK (Java Development Kit) erben. Es wäre möglich, GUIs über diese Vererbungsbeziehung zu erkennen. Allerdings müsste hierzu das JDK oder zumindest die Klasse `JPanel` als Library in den Sotographen eingelesen werden, da der Sotograph nur die Klassen und Beziehungen berücksichtigt, die er als Bytecode vorliegen hat. Um die GUIs ohne diesen Extraaufwand zu finden, wurden sie in dieser Arbeit darüber erkannt, dass sie den String „GUI“ im Namen haben. Dies ist eine gebräuchliche Konvention, die in den Beispielsystemen eingehalten wurde. Die Überlegungen zum „Materialien-Problem“ im nächsten Abschnitt sind übertragbar auf das „GUI-Problem“ bei der Benutzung von JWAM 2.

4.2 Das Materialien-Problem

In der Liste der Vererbungsbeziehungen im vorigen Abschnitt tauchen alle wichtigen WAM-Elemente bis auf Materialien auf. Dies liegt daran, dass Materialien über einfache Vererbungsbeziehungen nicht eindeutig zu erkennen sind. Materialien sollen in JWAM das Interface `Thing` implementieren. Über diese Vererbungsbeziehung können sie aber nicht einwandfrei identifiziert werden, da auch viele andere Elemente in JWAM das Interface `Thing` implementieren. Da es für Materialien auch keine Namenskonventionen wie z.B. für GUIs gibt, können sie auch darüber nicht gefunden werden. Da das Material eines der zentralen Elemente der WAM-Modellarchitektur

ist, sollten die vorhandenen Regeln für Materialien aber trotzdem überprüft werden können. Im Weiteren werden verschiedene Lösungsansätze für dieses Problem betrachtet.

Eine Lösungsmöglichkeit wäre, nach dem Ausschlussprinzip vorzugehen. Dabei werden Materialien über ihre Vererbungsbeziehung zu `Thing` erkannt. Über zusätzliche Vererbungsbeziehungen werden die Klassen ausgeschlossen, die keine Materialien sein können. Z.B. sind Klassen, die die Interfaces `Thing`, und `Tool` implementieren, keine Materialien, sondern Werkzeugklassen. Materialien würden also darüber identifiziert, dass sie das Interface `Thing`, aber z.B. nicht das Interface `Tool` implementieren. Neben `Tool` müssten viele andere Interfaces ausgeschlossen werden, die in der Vererbungshierarchie unter `Thing` liegen. Es gibt allerdings sehr viele Subtypen von `Thing`, wie die Abbildungen 4.1, 4.2 und 4.3 zeigen. Neben der Schwierigkeit, dass sehr viele Vererbungsbeziehungen ausgeschlossen werden müssten, ist das größere Problem, dass es Subtypen von `Thing` gibt, die von Materialien, aber auch von anderen WAM-Elementen implementiert werden. Ein Beispiel dafür ist das Interface `SelfDescribingThing` aus JWAM 2, das im Beispielsystem EMS von Materialien und von Werkzeugen implementiert wird. Hier zeigt sich, dass es Interfaces gibt, die nicht zweifelsfrei Materialien oder „Nicht-Materialien" zugeordnet werden können. Eine Klasse, die nur `Thing` und `SelfDescribingThing` implementiert, könnte also nicht eindeutig als Material oder „Nicht-Material" erkannt werden. Aus den Überlegungen dieses Abschnittes ergibt sich, dass Materialien mit den zur Zeit benutzten JWAM-Versionen nicht allein über Vererbungsbeziehungen identifiziert werden können.

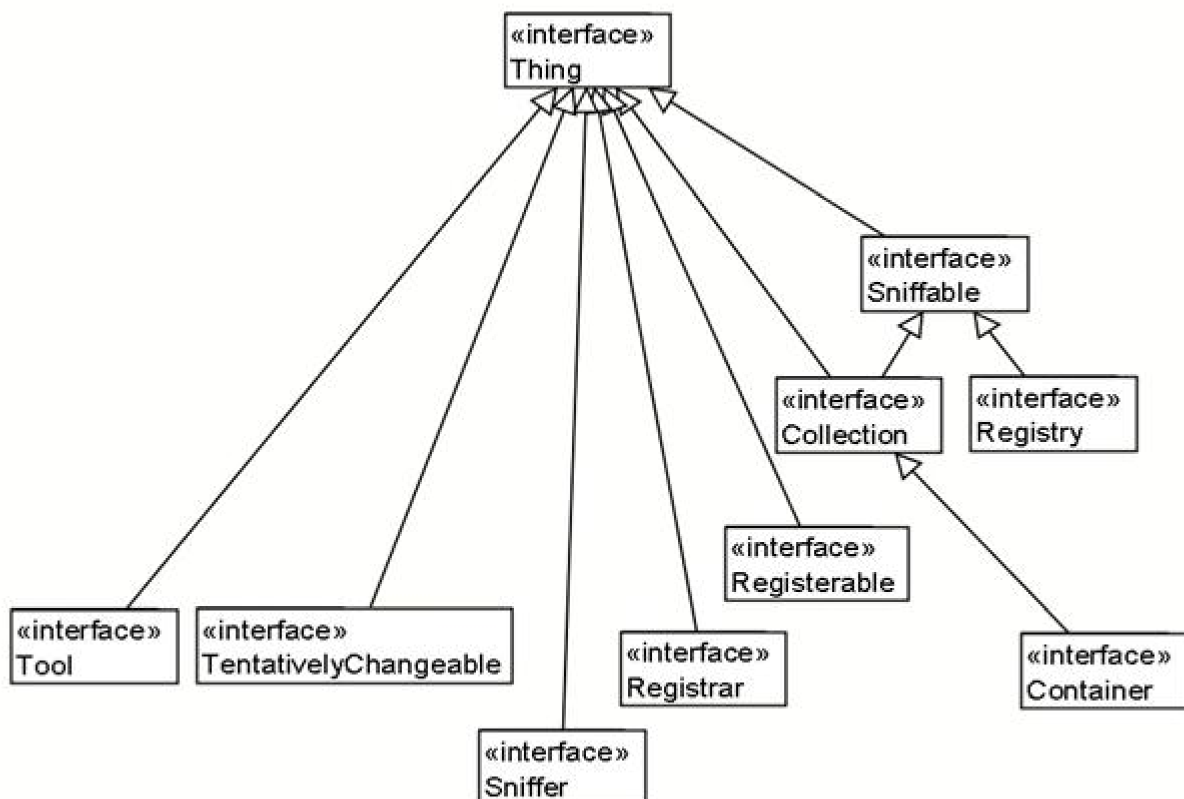

Abbildung 4.1: `Thing`-Hierarchie in JWAM 1.8, nur direkte Beziehungen zu `Thing`

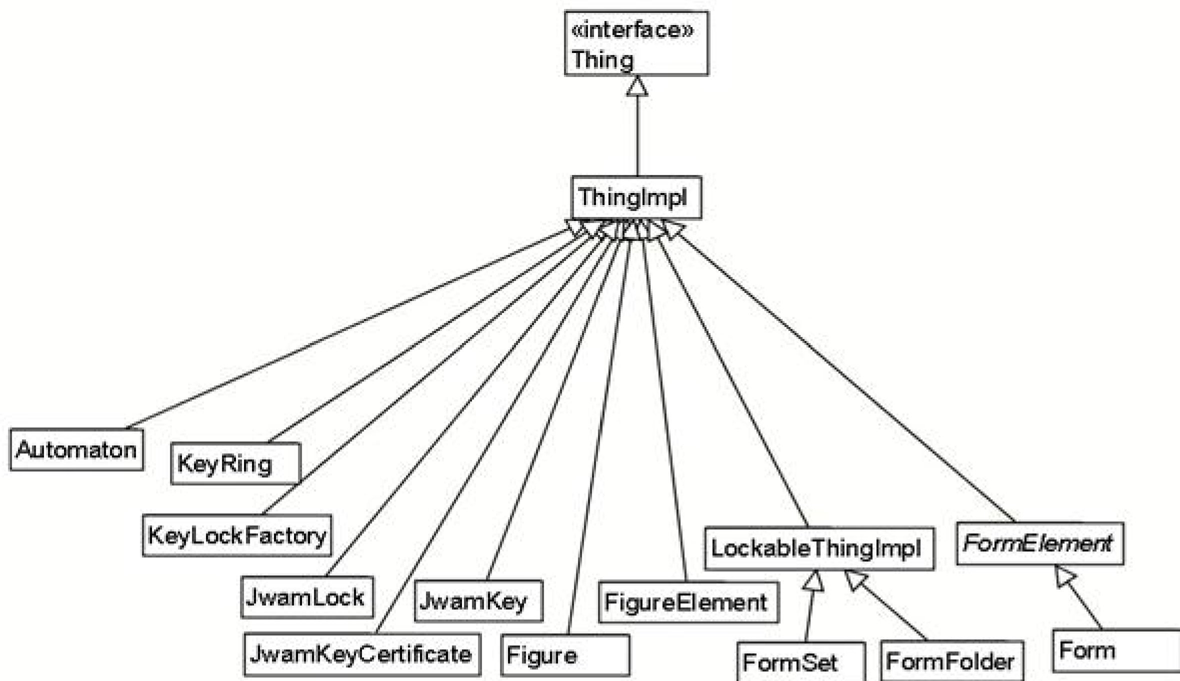

Abbildung 4.2: `Thing`-Hierarchie in JWAM 1.8, nur Beziehungen zu `ThingImpl`

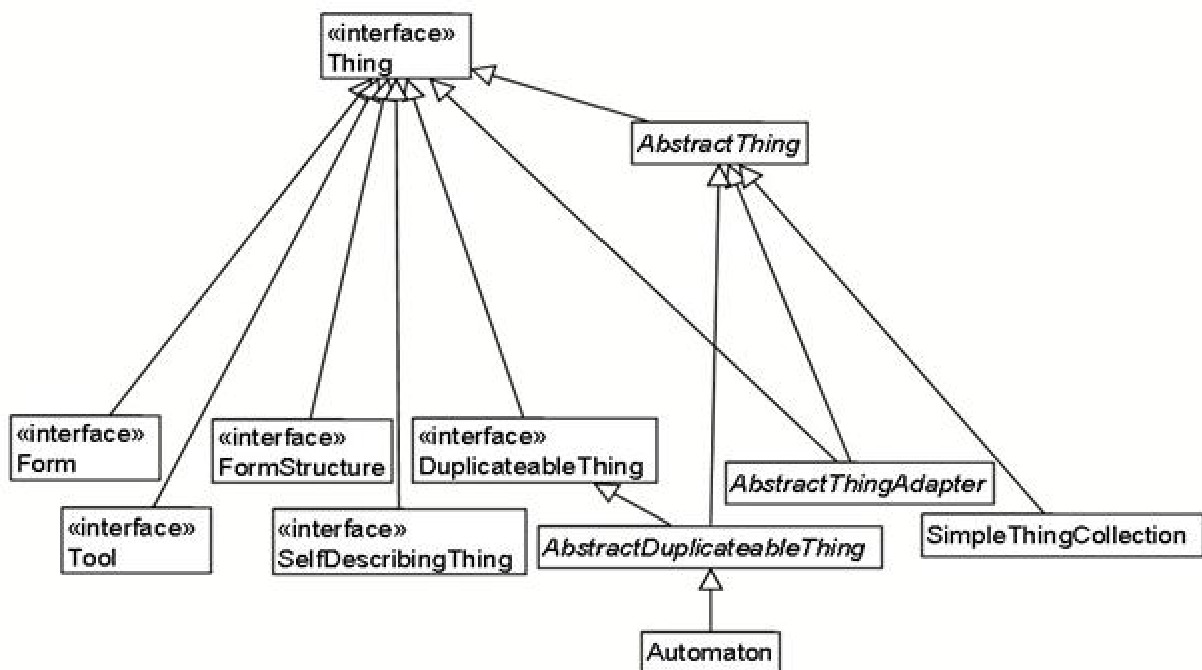

Abbildung 4.3: `Thing`-Hierarchie in JWAM 2

Um Materialien zu identifizieren, sind weitere Möglichkeiten denkbar. Eine Query, die alle Materialien in einer Tabelle zusammenstellt, könnte vom Benutzer als Eingabe das Package verlangen, unter dem alle Materialien liegen. Zusätzlich könnte geprüft werden, ob eine Vererbungsbeziehung zu `Thing` vorliegt. Es treten aber Probleme auf, wenn nicht alle Materialien unter einem Package liegen. Jedes Eingabefeld für Queries im Sotographen muss explizit codiert werden. Eine flexible Lösung, bei der eine beliebige Anzahl von Packages angegeben werden kann, ist also nicht möglich. Die Lösung, ein Package anzugeben, funktioniert auch nicht, wenn in diesem Package nicht nur Materialien liegen, sondern auch noch andere Klassen. Diese wür-

den dann, sofern sie das Interface `Thing` implementieren, auch als Materialien behandelt. Diese Lösung des Materialien-Problems, die nicht sehr viel Aufwand für den Benutzer bedeuten würde, funktioniert also in vielen Fällen in der Praxis nicht. Es sei denn der Benutzer passt die Packagestruktur des Systems auf diese Möglichkeit hin an, wodurch er einen großen Mehraufwand hätte.

Eine andere Lösung wäre, die What-If Funktion des Sotographen zu nutzen. Mit ihr kann der Benutzer u.a. Klassen in andere Packages verschieben, Packages erstellen oder umbenennen. Diese Funktion ist vorgesehen, um z.B. festzustellen, ob sich Zyklen zwischen Packages bereits auflösen, wenn eine Klasse verschoben wird. Dies kann mit der What-If-Analyse im Sotographen simuliert werden. Die Verschiebung wirkt sich nicht auf den Quelltext aus, sondern nur auf das logische Modell des Systems, das der Sotograph in der Datenbank speichert. Es wäre möglich, alle Packages umzubenennen, in denen Materialien liegen, so dass die Packages den String „material" im Namen haben. Es könnte auch ein neues Package mit „material" im Namen angelegt werden und alle Materialien dorthin verschoben werden. Diese Lösung hat den Nachteil, dass der Entwickler das logische Modell selber anpassen muss. Ein weiterer Nachteil dieser Lösung macht sich bemerkbar, wenn der Sotograph das Modell, das er vom System hat, auf Zyklen überprüft. Ändert man dieses Modell durch eine What-If-Analyse, so kann es sein, dass nicht mehr alle Zyklen bestehen, da sie durch das Verschieben von Materialien behoben worden sind. D.h. man müsste die Zyklenanalyse vor der What-If-Analyse vornehmen, oder die Änderungen wieder rückgängig machen nachdem überprüft wurde, ob die Materialien-Regeln eingehalten werden. Diese Lösung wäre umsetzbar, sie würde aber bei einigen Systemen sehr viel Aufwand für den Benutzer bedeuten.

Die Verwendung von Namenskonventionen ist in WAM-Systemen üblich, es gibt jedoch keine Namenskonvention für Materialien. Gäbe es ein eindeutiges Namenskürzel für jedes WAM-Element, so könnten alle WAM-Elemente über ihre Namen identifiziert werden. Die Verwendung des Rahmenwerks zur Erkennung der Elemente wäre dann nicht mehr nötig. Allerdings würden eindeutige Namenskonventionen evtl. zu sehr langen Klassennamen führen. Bei zu kurzen Namenskonventionen wäre die Gefahr groß, dass sie ungewollt in anderen Klassennamen auftauchen.

Die eleganteste Lösung wäre es, wenn es in JWAM auch für Materialien eine Vererbungsbeziehung gäbe, die sie von den anderen Elementen unterscheiden würde, z.B. ein Markerinterface `Material`. Solche Markerinterfaces, die nur einen Typ, aber keine Schnittstelle definieren, gibt es bereits. So ist z.B. das Interface `Service` in JWAM 2 ein reines Markerinterface. Mit der Einführung eines Markerinterfaces `Material` in JWAM könnten Materialien ohne Mehraufwand für den Benutzer eindeutig identifiziert werden, auch wenn sie über mehrere Packages verstreut liegen. Markerinterfaces könnten jedoch nicht nur für das Auffinden von Materialien Vorteile bringen: Ein Markerinterface `GUI` in JWAM würde die Erkennung der GUIs sicherer machen als über Namenskonventionen. Markerinterfaces könnten auch in WAM-Systemen eingesetzt werden, die weder JWAM noch Namenskonventionen verwenden. Der geringe Mehraufwand, Markerinterfaces im System zu benutzen würde aufgewogen durch die Möglichkeit, die WAM-Modellarchitektur überprüfen zu können. Markerinterfaces nachträglich in Systeme einzuführen, würde großen Aufwand mit sich bringen. Sie sollten aber bei der Entwicklung neuer Systeme eingeführt werden.

Auch wenn es für das Problem, Materialien zu finden unter den vorhandenen Rahmenbedingungen keine optimale Lösung gibt, so musste für die Erstellung der Queries eine Methode der Materialienidentifikation gewählt werden. In den Projekten, an denen die Autorin dieser Arbeit bereits mitgearbeitet hat, gab es die Konvention, dass Materialien in einem Package mit dem Namen „material“ lagen. Deshalb werden alle Queries, die überprüfen, ob die Regeln für Materialien eingehalten werden, auf allen Klassen ausgeführt die das Interface `Thing` implementieren und die in Packages liegen, die den String „material“ oder „Material“ im Namen haben. Diese Lösung ist zwar nicht befriedigend, funktioniert aber bei den Systemen, die in dieser Arbeit untersucht wurden. Im günstigsten Fall hat der Anwender der Queries keine Extraarbeit. Im ungünstigen Fall müsste er die What-If-Analyse verwenden, um die Packages, die Materialien enthalten umzubenennen, oder die Materialien zu verschieben. Mit dieser Lösung und durch mögliche Mehrarbeit durch den Anwender können alle Materialien identifiziert werden. Es wird also eine Mischung aus Identifikation über Vererbung und Namenskonvention verwendet.

4.3 Überprüfung der WAM-Modellarchitektur mit dem ModelManager des Sotographen

Wie bereits in Kapitel 2.1 erwähnt, können mit dem Sotographen Schichtenarchitekturen definiert und überprüft werden. Zunächst können Packages Subsystemen zugeordnet werden. Einzelne Klassen Subsystemen zuzuweisen ist nicht möglich. Subsysteme wiederum können in Schichten angeordnet werden. Die Beziehungen zwischen Schichten sind folgendermaßen festgelegt: Beziehungen einer unteren Schicht zu einer oberen Schicht sind generell verboten. Auf wie viele der unteren Schichten Elemente einer oberen Schicht zugreifen dürfen, kann der Benutzer festlegen. Ebenfalls festgelegt werden kann, ob Beziehungen zwischen Subsystemen innerhalb einer Schicht erlaubt sind.

Wenn Werkzeuge, Materialien, Services, Automaten und Fachwerte in verschiedenen Packages liegen, wie es meist der Fall ist, können diese Packages einzelne Subsysteme bilden. Diese Subsysteme können mit dem Sotographen in Schichten angeordnet werden, wie es in Abbildung 4.4 dargestellt ist. Der Sotograph kann diese Schichtenarchitektur überprüfen und die Verletzungen anzeigen.

Abbildung 4.4: WAM-Schichten

Subsysteme müssen für jedes System neu definiert werden, die Schichtenarchitektur könnte für verschiedene Systeme wieder verwendet werden. Automat- und Serviceschicht müssten aber je nach Projekt angepasst werden, da Services auch Automaten benutzen dürfen, solange es nicht zu einer gegenseitigen Benutzt-Beziehung kommt.

Sollte es in einem Projekt Automaten geben, die Services benutzen, aber auch Services die Automaten benutzen, müssten diese beiden Elementarten in einer Schicht angeordnet werden.

Mit dieser Schichtenarchitektur wird aber nur ein Teil der Regeln der WAM-Modellarchitektur berücksichtigt. Zu den Werkzeugen gehören auch GUIs und IAKs. Nach dieser Schichtenarchitektur dürften sie auf Automaten, Services und Materialien zugreifen. Die WAM-Regeln verbieten dies aber. Auch die Regeln für die Beziehungen zwischen FK, IAK, Tool-Klasse und GUI eines komplexen Werkzeugs deckt diese Schichtenarchitektur nicht ab. Um die Beziehungen zwischen diesen Werkzeugkomponenten mit den vorhandenen Mitteln des Sotographen zu modellieren, wäre großer Aufwand erforderlich.

Da GUIs, IAKs, FKs und Tool-Klassen, die zu einem Werkzeug gehören, meist in einem Package liegen, können sie nicht verschiedenen Subsystem zugeordnet werden. Beziehungen zwischen ihnen können daher auch nicht mit einer Architekturspezifikation festgelegt werden. Um GUIs, IAKs, FKs und Tool-Klassen verschiedenen Subsystemen zuzuweisen, müssten sie zunächst in verschiedene Packages verschoben werden. Dann könnten über eine Schichten- oder Grapharchitektur die Beziehungen zwischen ihnen festgelegt werden. Der Aufwand, um die einzelnen Elementklassen in verschiedene Packages zu verschieben, ist jedoch vor allem für große Systeme so hoch, dass eine Architekturüberprüfung nicht mit vertretbarem Aufwand durchgeführt werden kann. Hinzu kommt, dass mit einer solchen Architekturüberprüfung durch den Sotographen nur die Verbots-Beziehungsregeln der WAM-Modellarchitektur überprüft werden können. Verletzungen von Gebots-Beziehungsregeln und Einzel-Element-Regeln können so nicht festgestellt werden.

4.4 Queries

In den folgenden Abschnitten wird dargestellt, wie Queries im Sotographen implementiert werden. Abschnitt 4.4.1 beschreibt den Inhalt der Datenbank, die die Grundlage für die Queries bildet. Dadurch werden auch die prinzipiellen Möglichkeiten, die mit Queries gegeben sind, verdeutlicht. In Abschnitt 4.4.2 wird die Sprache TQL erläutert, mit der Queries implementiert werden. Eine spezielle Art von Queries sind FollowUp-Queries. Sie werden in Abschnitt 4.4.3 behandelt. Zum Abschluß werden in Abschnitt 4.4.4 einige Querybeispiele vorgestellt.

Queries werden im QueryDeveloper (s. Abbildung 4.5) erstellt. Neben dem Quelltext der Query werden hier u.a. alle Tabellen der Datenbank angezeigt und ihre Felder beschrieben. Aus dem QueryDeveloper heraus lassen sich Queries speichern, ausführen oder als XML-Dateien ex- und importieren.

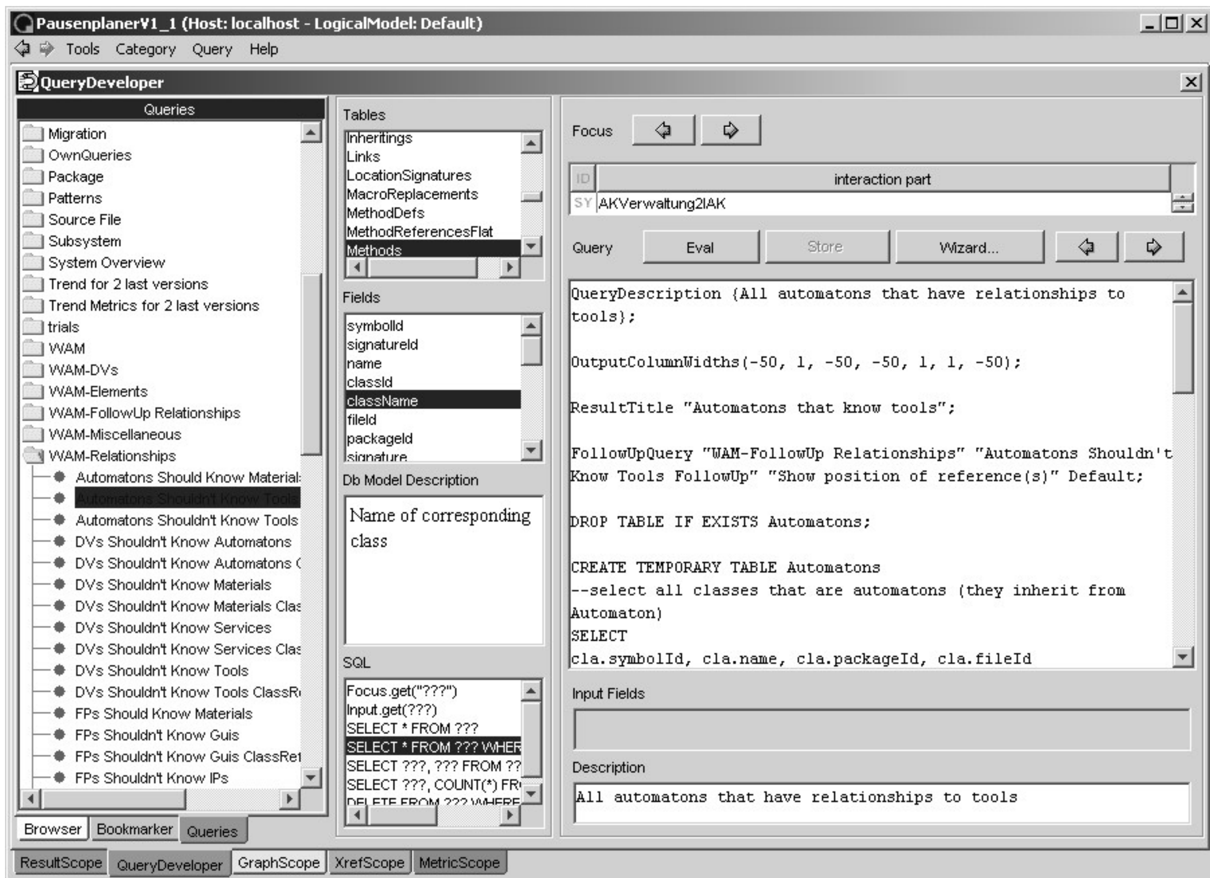

Abbildung 4.5: Der QueryDeveloper

4.4.1 Datenbank des Sotographen

Queries sind Abfragen auf der Datenbank des Sotographen. Der Sotograph hat die Datenbank mit allen Strukturinformationen des untersuchenden Systems gefüllt. Diese Strukturinformationen bestehen aus allen Artefakten des Systems und ihren Beziehungen untereinander. Artefakte des Systems sind Attribute, Methoden, Klassen, Dateien und Packages. Beziehungen sind Vererbung, Aufruf, Assoziation, Typverwendung, Attributzugriffe und Zugehörigkeit. Unter Zugehörigkeit fällt z.B. in welchem Package eine Klasse liegt. Auf welchem ER-Modell die Tabellen des Sotographen beruhen, kann in der Dokumentation des Sotographen nachgelesen werden. Ein Ausschnitt aus dem ER-Modell (Entity Relationship-Modell) des Sotographen ist in Anhang C abgebildet. Mit Queries können nur Sachverhalte überprüft werden, die mit den Informationen aus der Datenbank ermittelt werden können. Die niedrigste Detailebene sind daher Attribute und Methoden einer Klasse. Bei Letzteren kann nur auf die Signatur zugegriffen werden, aber nicht auf den Methodenrumpf.

4.4.2 TQL

Queries werden in TQL, einer Erweiterung von SQL geschrieben und bestehen im Kern aus einer oder mehreren SQL-Abfragen. Die vollständige EBNF zu TQL kann im Benutzerhandbuch des Sotographen nachgelesen werden.

Die Erweiterungen in TQL dienen u.a. dazu, die Query zu dokumentieren, indem neben Kommentaren im Quelltext der Query Beschreibungen und Tooltips definiert werden können. Die Ausgabe des Ergebnisses einer Query erfolgt in einer Tabelle im

ResultScope des Sotographen. Mit TQL kann die Darstellung dieser Tabelle beeinflusst werden. Weiterhin können verschiedene Arten von Eingabefeldern definiert werden. Der Benutzer muss die Eingabefelder ausfüllen bevor er die Query startet, damit die Inhalte zugreifbar sind während die Query ausgeführt wird. TQL erlaubt die Anpassung der SQL-Abfrage auf den Inhalt der Eingabefelder und den Inhalt von Tabellen, indem mit If-Else-Konstrukten gearbeitet werden kann. Ein weiterer Vorteil von TQL ist die Möglichkeit, für eine Query einen Fokus zu definieren. Dieser Fokus ist eine Tabellenzeile, auf deren Werte innerhalb der Query zugegriffen werden kann. Der Benutzer muss den Fokus angeben, bevor die Query ausgeführt wird.

Die EBNF[2] der TQL beschreibt den Aufbau einer Query wie folgt:

```
Query :   { QueryDescription }{ InputField }
          { InputDescription | OutputDefinition | ResultTitle }
          ExtendedSQL
```

Die `QueryDescription` kann aus einer textuellen Beschreibung der Query und einem Tooltip bestehen.

Im Bereich `InputField` können Integer-, Boolean-, String- oder Listeneingabefelder definiert werden.

Die `InputDescription` besteht aus einer Beschreibung eines Eingabefelds.

Die `OutputDefiniton` legt das Aussehen und Verhalten der Ergebnistabelle fest. In diesem Abschnitt können auch eine oder mehrere FollowUp-Queries (s. Abschnitt 4.4.3) definiert werden.

Der Titel der Ergebnistabelle wird unter `ResultTitle` festgelegt.

`ExtendedSQL` besteht aus folgenden Elementen:

```
ExtendedSQL:   { BddQuery | MacroLines | ExtendedSQLLines }
```

Mit einer `BddQuery` wird eine Anfrage ausgedrückt, die von einer speziellen engine, dem „CrocoPat“ berechnet wird. Da diese Art von Anfragen in dieser Arbeit nicht benutzt wird, wird nicht weiter auf sie eingegangen.

Mit `MacroLines` können entweder Kommentare oder if-else-Konstrukte eingefügt werden. Sie enthalten neben booleschen Ausdrücken und den „if“-, „else“- und „elseif“-Schlüsselworten nur `ExtendedSQLLines`.

`ExtendedSQLLines` enthalten neben dem Zugriff auf Eingabe- oder Fokusfelder nur SQL-Statements.

4.4.3 FollowUp-Queries

[2] Verwendete EBNF:
[]: 0 oder 1 Vorkommen
{}: 0, 1 oder mehrfaches Vorkommen
|: oder

FollowUp-Queries können ein Ergebnis durch weitere Details genauer erläutern, oder ausgehend von einem Ergebnis weitere Informationen liefern. Beispielsweise gibt es eine Query, die Materialien sucht, die nur „get“- und/oder „set“-Methoden haben. In der Ergebnistabelle werden diese Materialien aufgelistet. Mit Hilfe der FollowUp-Query können in einer weiteren Tabelle alle Methoden eines dieser Materialien angezeigt werden.

Um eine FollowUp-Query zu nutzen, muss der Benutzer in der Ergebnistabelle einer Query eine Zeile selektieren und im Kontextmenü die FollowUp-Query ausführen (s. Abbildung 4.6). Das Ergebnis der FollowUp-Query wird dann im ResultScope angezeigt. Dies ist allerdings nur möglich, wenn für diese Query eine FollowUp-Query definiert ist.

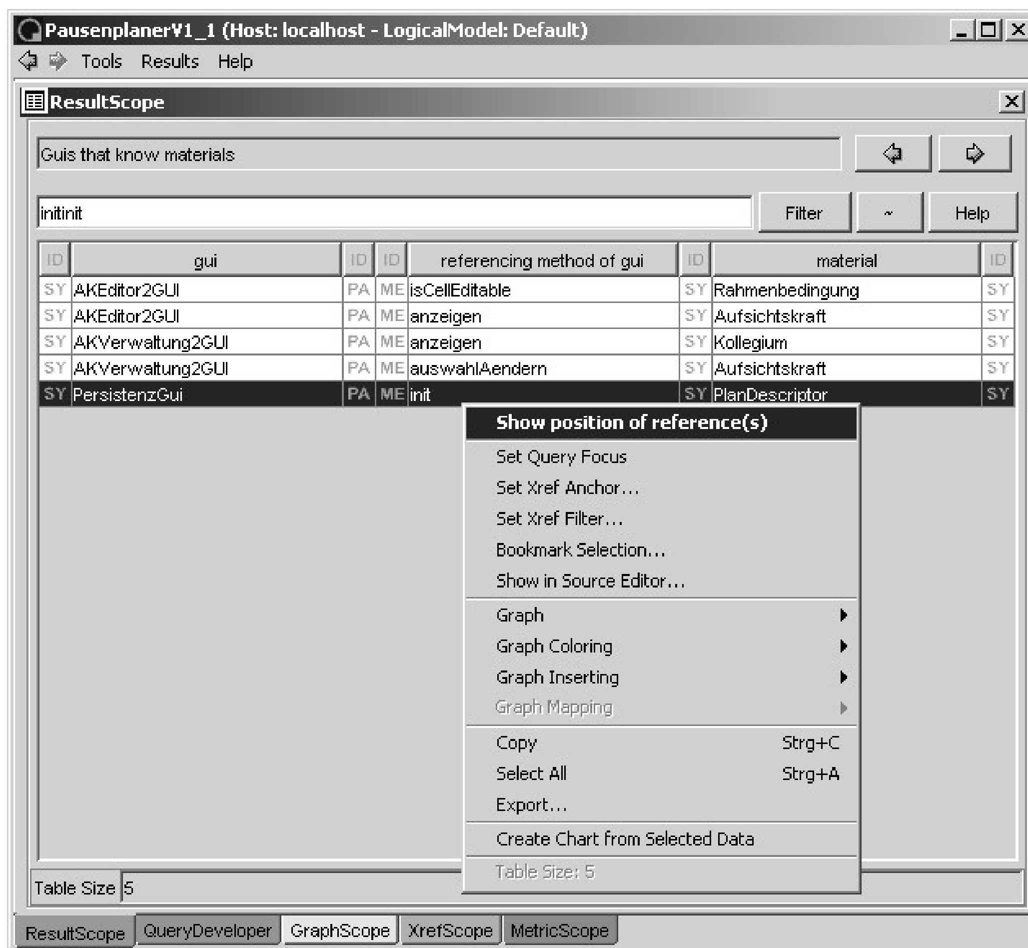

Abbildung 4.6: ResultScope mit angewählter FollowUp-Query im Kontextmenü

Soll nach Ausführung einer Query eine FollowUp-Query zur Verfügung stehen, so muss im Quelltext der Query angegeben werden, wie die FollowUp-Query heißt und in welchem Ordner sie sich befindet. In der FollowUp-Query wird ein Fokus definiert, der auf die Inhalte der Ergebnistabelle der ersten Query ausgerichtet ist. D.h. die FollowUp-Query greift auf die Inhalte einer Zeile der Ergebnistabelle der ersten Query zu. Mit der Auswahl einer Zeile im ResultScope und der Ausführung der FollowUp-Query aus dem Kontextmenü heraus, setzt der Benutzer diesen Fokus. Abgesehen von der Ausrichtung auf den Fokus unterscheidet sich die Implementierung einer FollowUp-Query nicht von der Implementierung anderer Queries.

4.4.4 Querybeispiele

In Abbildung 4.7 ist eine sehr einfache Query im QueryDeveloper dargestellt. Der Quelltext der Query beginnt mit der Querybeschreibung. Nach der Querybeschreibung folgen eine Definition eines Eingabefeldes und die dazugehörige Beschreibung. Mit `OutputColumnWidths` werden die Spaltenbreiten der Ergebnistabelle definiert. Ihr Titel wird mit `ResultTitle` festgelegt.

Nach den Anweisungen für die Ergebnistabelle folgt eine SQL-Anweisung, die mit einem „if-else"-Ausdruck und dem Wert aus dem zuvor spezifizierten Eingabefeld erweitert ist.

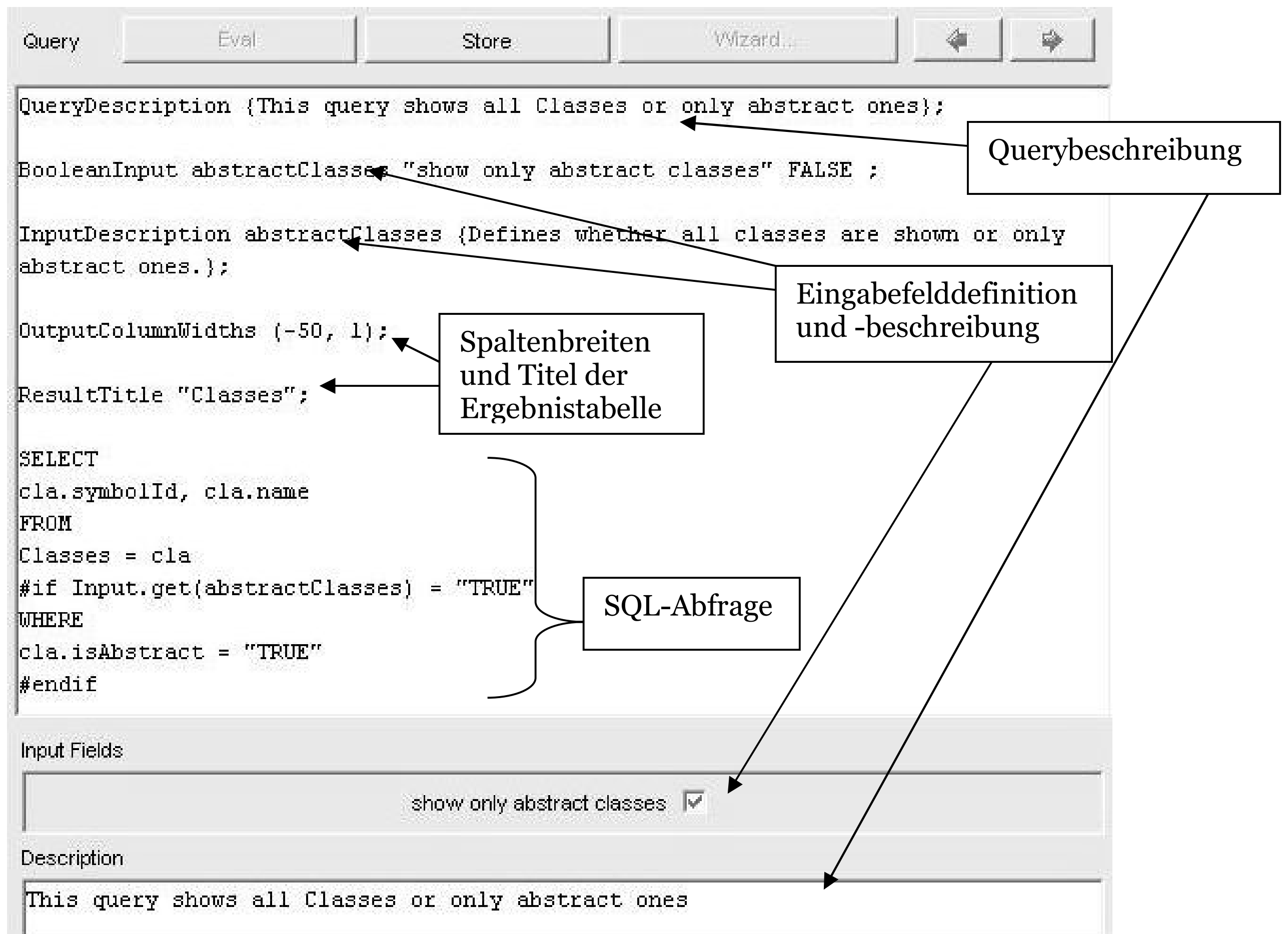

Abbildung 4.7: Definition einer Query

Der folgende Code ist der Quelltext der Query „IPs Shouldn't Know Automatons". Sie überprüft, ob die Regel „*IAKs dürfen keine Automaten kennen*" eingehalten wird. Kommentarzeilen beginnen mit „--". Nach der Querybeschreibung und den Formatierungsanweisungen für die Ergebnistabelle beginnt die SQL-Anweisung. Sie greift auf die Tabellen „IPs" und „Automatons" zu, die mit der Query „init tables" zuvor angelegt werden müssen. Die Tabelle „IPs" enthält alle Artefakte, die in Interaktionsklassen enthalten sind, die Tabelle „Automatons" enthält alle Artefakte, die in Automatenklassen enthalten sind. Neben diesen beiden Tabellen wird noch auf die Tabelle „SymbolReferences" zugegriffen, die alle Beziehungen zwischen sämtlichen Artefakten des Systems enthält. In ihr wird nach Beziehungen zwischen einer IAK und einem Automaten gesucht. In der Ergebnistabelle werden alle gefundenen Beziehungen aufgelistet. Mit der SELECT-Anweisung werden sehr viele Informationen ausgewählt. Sie ermöglichen es nach Ausführung der Query, die Ergebnisse auch graphisch dar-

stellen zu lassen und an die genaue Stelle im Quelltext zu springen. Beispiele weiterer Queries sind in Anhang B zu finden.

```
-- Beschreibung der Query
QueryDescription {The Query "init tables" in the category
"WAM-Init" must be executed before executing any of the que-
ries of this category.

This query shows all relationships from interaction parts to
automatons};

-- Festlegen der Spaltenbreite der Ergebnistabelle
OutputColumnWidths(-50, 1, -50, 1, -50, 1, -50, 1, 1, 1);

-- Titel der Ergebnistabelle
ResultTitle "Interaction parts that know automatons";

-- Beginn der SQL-Anweisung
SELECT DISTINCT

IPs.symbolId as "refingClassId",
IPs.name as "refingClass",
IPs.symId as "refingSymbolId",
IPs.symName as "refingSymbol",
Automatons.symbolId as "refedClassId",
Automatons.name as "refedClass",
Automatons.symId as "refedSymbolId",
Automatons.symName as "refedSymbol",
ref.refPos as "position of reference",
ref.referenceType

FROM

IPs,
Automatons,
SymbolReferences = ref

WHERE

IPs.symId = ref.refingSymbolId AND
Automatons.signatureId = ref.refedSignatureId
```

4.5 Die WAM-Queries

Queries können im Sotographen in Gruppen angelegt werden. Möchte man Queries in den Sotographen importieren, so muss jede Gruppe einzeln importiert werden. Um einen Kompromiss zwischen Übersichtlichkeit und möglichst wenig Aufwand beim Im- und Exportieren zu erzielen, sind die WAM-Queries in sieben Gruppen unterteilt. Es gibt die Gruppen WAM-init, WAM-Elements, WAM-Inheritance, WAM-FollowUp, WAM-Forbidden-Relationships, WAM-Enforced-Relationships und WAM-Single-Element-Rules. Nur die Gruppen WAM- Forbidden-Relationships, WAM-Enforced-

Relationships und WAM-Single-Element-Rules enthalten Queries, die Regeln der Modellarchitektur überprüfen.

Um mit der Klassifizierung der Regeln konsistent zu sein, hätten die Queries, die WAM-Regeln überprüfen, zunächst nach Elementart und weiterhin nach den verschiedenen Beziehungsregeln und Einzel-Element-Regeln unterteilt werden müssen. Im Sotographen gibt es jedoch nicht die Möglichkeit, Gruppen innerhalb von Gruppen anzulegen, was für diese Unterteilung nötig gewesen wäre. Aus diesem Grund orientiert sich die Unterteilung der Queries, die nach Verstößen gegen WAM-Regeln suchen, an der Klassifizierung der Regeln, weicht aber von ihr ab. Sie wurden gruppiert nach Queries, die Einzel-Element-Regeln überprüfen, Queries, die Verbots-Beziehungsregeln kontrollieren und Queries, die Verletzungen von Gebots-Beziehungsregeln suchen. Erstere sind in der Gruppe WAM-Single-Element-Rules zu finden, Verbots-Beziehungsregeln finden sich in der Gruppe WAM-Forbidden-Relationships, Gebots-Beziehungsregeln in der Gruppe WAM-Enforced-Relationships. Damit wurde zumindest ein Teil der Regelklassifizierung umgesetzt. Die Unterteilung nach Elementart wurde nicht berücksichtigt.

Im Folgenden wird erläutert, welche Funktionen die Queries der einzelnen Gruppen haben.

4.5.1 WAM-Init

Um die Regeln der WAM-Modellarchitektur mit Queries zu überprüfen, müssen zunächst die WAM-Elemente im Quelltext identifiziert werden. Dazu erstellt die Query „init tables", die die einzige Query in dieser Gruppe ist, Tabellen der WAM-Elemente, die alle Elemente dieser Art enthält. Im Einzelnen sind dies Tabellen für die Elemente Material, IAK, FK, Tool-Klasse, GUI, Monotool-Klasse, Service, Automat, Fachwert und eine Tabelle „Tools", die alle Elemente enthält, die zu Werkzeugen gehören. Die Tabellen für GUIs, Monotool-Klassen und Services und die Tabelle „Tools" werden jeweils für JWAM 1.8 und für JWAM 2 erstellt, da hier unterschiedliche Suchkriterien angewandt werden müssen, um die Elemente zu identifizieren. Die Erkennung der Elemente wird in dieser Query gebündelt, um zu vermeiden, dass der Code zur Identifizierung in jeder Query dupliziert wird. Da die Ergebnisse dieser Query nur intern von anderen Queries genutzt werden, wird für den Benutzer kein Ergebnis im ResultScope dargestellt.

Vor der Ausführung dieser Query kann ausgewählt werden, ob die Tabellen Klassen aus dem JWAM-Rahmenwerk enthalten sollen oder nicht. Sollen die Klassen aus dem Rahmenwerk ignoriert werden, so wird nur in Packages nach WAM-Elementen gesucht, die nicht den String „jwam" im Packagepfad haben. Dadurch tauchen in den Ergebnissen keine Klassen auf, die aus JWAM sind, die das Ergebnis verfälschen könnten. Z.B. gibt es in JWAM ein Package namens „materialChooser" mit einem Werkzeug darin. Aufgrund des Packagenamens würde dieses Werkzeug als Material identifiziert. Da es aber kein Material ist und daher auch die Regeln für Materialien nicht einhalten muss, könnten Regelverstöße gefunden werden, die in Wirklichkeit gar keine sind. In der Standardeinstellung werden Ergebnisse aus JWAM nicht berücksichtigt. Die Möglichkeit, Ergebnisse aus JWAM einzubeziehen, wurde offen gelassen, um JWAM-Beispiele oder das JWAM Rahmenwerk selbst überprüfen zu können.

4.5.2 WAM-Elements

In dieser Gruppe finden sich Queries, die jeweils eine Art von Elementen oder Gruppen von Elementen auflisten. Welche Elemente dies sind, ergibt sich aus dem Namen der Query. Es werden alle Elemente dieser Art angezeigt, die im System gefunden werden. Diese Queries überprüfen nicht, ob Regeln der WAM-Modellarchitektur befolgt werden, sie sind aber nützlich, um sich einen Überblick über ein WAM-System zu verschaffen. So z.B. um die Frage zu klären, wie viele Automaten es im System gibt.

Queries in der Gruppe WAM-Elements:

- Automatons
- Domain Values (DVs)
- Functional Parts (FPs)
- GUIs
- Interaction Parts (IPs)
- Materials
- Monotool-classes
- Services
- Tool-classes
- Tools (Complex, grouped)
- Tools (FP, IP, GUI, Tool-class, Monotool-class)
- Tools (Mono, grouped)

In der Query „Tools (FP, IP, GUI, Tool-class, Monotol-class)“ werden alle Klassen aufgelistet, die Teile eines Werkzeugs sind.

Die Query „Tools (Complex, grouped)” listet alle Werkzeuge auf, die aus GUI, FK, IAK und Tool-Klasse bestehen. Eine Zeile zeigt alle Elemente, die zu einem Werkzeug gehören. Die zusammengehörigen Werkzeugteile werden, ausgehend von der IAK, über die Namen gefunden. So werden z.B. zu einer Klasse `AKEditorIAK` GUI, FK und Tool-Klasse gesucht, deren Namen ebenfalls mit „AKEditor“ beginnen. Es werden nur die Werkzeuge aufgelistet, die vollständig gefunden wurden. Unvollständige Werkzeuge können mit der Query „Complex Tools Consist Of GUI, FP, IP And Tool-class“ aus der Gruppe WAM-Single-Element-Rules gefunden werden.

Die Query „Tools (Mono, grouped)“ zeigt alle monolithischen Werkzeuge an. Zu einem Werkzeug werden jeweils Monotool-Klasse und GUI in einer Zeile angezeigt. Die zu einem Werkzeug gehörenden Klassen werden wie bei der Query „Tools (Complex, grouped)“ über die Namen gesucht.

4.5.3 WAM-Inheritance

Die Queries dieser Gruppe überprüfen, ob das Rahmenwerk JWAM richtig verwendet wird. Es geht also nicht um die Überprüfung der WAM-Regeln aus der zweiten Ebene des WAM-Ansatzes sondern um die Prüfung der dritten Ebene, der Implementierung mit JWAM. Da die Query „init tables“ (s. Abschnitt 4.5.1) die Elemente der WAM-Modellarchitektur im System hauptsächlich über Vererbungsbeziehungen zu JWAM findet, ist es wichtig, dass die WAM-Elemente im System von den richtigen Klassen aus JWAM erben (s. Abschnitt 4.1.3). Nur so können die WAM-Elemente identifiziert werden und die WAM-Regeln mit den Queries der Gruppen WAM-Forbidden-

Relationships, WAM-Enforced-Relationships und WAM-Single-Element-Rules geprüft werden. Die Queries dieser Gruppe testen deshalb, ob die Klassen des Systems von den richtigen Klassen aus JWAM erben bzw. die richtigen Interfaces implementiert werden.

Bei den meisten Systemen werden die Queries dieser Gruppe keine Ergebnisse hervorbringen, da davon ausgegangen werden kann, dass JWAM richtig verwendet wird. Gibt es aber bei den Queries, die WAM-Regeln überprüfen, oder bei den Queries der Gruppe WAM-Elements weniger Ergebnisse als erwartet, so kann mit den Queries dieser Gruppe eine mögliche Fehlerquelle untersucht werden.

Um zu prüfen, ob die richtigen Klassen und Interfaces aus JWAM verwendet werden, wird auf die Namenskonventionen in WAM-Systemen zurückgegriffen. Die WAM-Elemente werden in den Queries dieser Gruppe über ihre Namen gefunden. Es werden z.B. alle Klassen, die im Klassennamen ein „dv“ enthalten als Fachwerte identifiziert. Daraufhin wird überprüft, ob diese Elemente die richtigen Interfaces implementieren bzw. von den richtigen Klassen aus JWAM erben. Von welchen Superklassen geerbt werden muss und welche Interfaces implementiert werden müssen, wurde in Kapitel 4.1.3 beschrieben. In dem Beispiel der Fachwerte müssten also alle Klassen, die ein „dv“ im Namen tragen, das Interface `DomainValue` aus JWAM implementieren. Ist dies nicht der Fall, so werden die Klassen in der Ergebnistabelle der Query aufgelistet.

Die Aussagekraft dieser Queries hängt davon ab, ob die Namenskonventionen für WAM-Systeme eingehalten werden. Die Namenskonventionen, über die die WAM-Elemente in den Queries dieser Gruppe gefunden werden, werden im Folgenden erläutert.

Die Bezeichner für ein Element der WAM-Modellarchitektur finden sich normalerweise am Ende des Klassennamens. Um auch Abweichungen von dieser Konvention zu berücksichtigen, wurden die Queries so implementiert, dass sie die WAM-Elemente erkennen, wenn die folgenden Namensteile in Groß- oder Kleinschreibung an irgendeiner Stelle im Namen auftreten:

- „Automaton“ oder „Automat“ für Automaten
- „DV“ für Fachwerte (engl. domain value)
- „FP“ oder „FK“ für Funktionskomponenten (engl. functional part)
- „GUI“ für GUIs
- „IP“ oder „IAK“ für Interaktionskomponenten (engl. interaction part)
- „Service“ oder „Server“ für Services
- „Tool“ für Tool-Klassen und Monotool-Klassen

Bei der Benutzung der Namenskonventionen zur Identifikation gibt es zwei Schwierigkeiten: Die erste Schwierigkeit ist, dass anhand des Namens nicht zwischen Tool-Klassen und Monotool-Klassen unterschieden werden kann, wie bereits in Kapitel 4.1.1 erläutert wurde. Es wird daher nur überprüft, ob solche Klassen das Interface `Tool` implementieren, das sowohl von Monotool-Klassen als auch von Tool-Klassen implementiert werden muss. Die zweite Schwierigkeit sind die Materialien, da es für sie keine Namenskonvention gibt. Um trotzdem zu überprüfen, ob alle Materialien das Interface `Thing` implementieren, muss hier anders vorgegangen werden. Die Materialien-Query dieser Gruppe prüft alle Klassen, die in einem Package liegen, das

„material“ im Namen hat – dieses sind vermutlich die Materialien – darauf, ob sie das Interface `Thing` implementieren.

Queries in der Gruppe WAM-Inheritance:

- Automaton Inheritance
- DV Inheritance
- FP Inheritance
- GUI Inheritance
- IP Inheritance
- Material Inheritance
- Service Inheritance
- Tool Inheritance

4.5.4 WAM-Forbidden-Relationships

Die Queries dieser Gruppe suchen nach Verstößen gegen 27 verschiedene Verbots-Beziehungsregeln der WAM-Modellarchitektur. Im weiteren Verlauf dieser Arbeit werden die Queries „Verbots-Queries“ genannt.

Alle Queries dieser Gruppe tragen den Namen „X Shouldn't Know Y“ und decken verbotene Beziehungen auf. Eine Beziehung von Klasse X zu Klasse Y besteht, wenn Klasse X in irgendeiner Form auf den Typ der Klasse Y zugreift. Gibt es zwischen zwei Klassen eine verbotene Beziehung, so taucht diese im Ergebnis nicht einmal, sondern eventuell mehrfach auf. Jeder Zugriff wird einzeln aufgelistet. Neben den Namen der Klassen, zwischen denen die verbotene Beziehung besteht enthält die Ergebnistabelle die Information, welches Artefakt der referenzierenden Klasse auf welches Artefakt der referenzierten Klasse zugreift. Hier handelt es sich meist um eine Methode, die eine andere ruft, es kann aber z.B. auch ein „instanceOf“-Konstrukt oder eine Variablendeklaration sein. Die Art der Beziehung kann der Spalte „referenceType“ in der Ergebnistabelle entnommen werden. Neben den Ids der verschiedenen Artefakte wird zu einer verbotenen Beziehung zusätzlich die Position im Quelltext angegeben, damit der Benutzer direkt an diese Stelle gelangen kann.

Eine vollständige Auflistung der Queries dieser Gruppe ist in Anhang B zu finden.

4.5.5 WAM-Enforced-Relationships

Die Queries dieser Gruppe suchen nach Verletzungen von zwölf Gebots-Beziehungsregeln. Sie werden im Folgenden als „Gebots-Queries“ bezeichnet.

Es gibt drei verschiedene Arten von Queries in dieser Gruppe, deren Funktionsweise und Zweck im Folgenden erläutert wird. Eine vollständige Auflistung der Queries dieser Gruppe ist in Anhang B zu finden.

Die Queries „X Should Know Materials“ für Automaten, FKs, Monotool-Klassen und Services überprüfen, ob diese Elemente Materialien kennen, und listen gegebenenfalls diejenigen auf, die keine Materialien kennen. Dabei suchen die Queries nicht danach, ob die Elemente ein bestimmtes Material kennen, sondern nur, ob eine Beziehung zu irgendeinem Material besteht.

Die Queries „IPs Should Know Their GUI“, „IPs Should Know Their FP“ und „Monotool-classes Should Know Their GUI“ suchen nach Verletzungen der Regel, dass Mo-

notool-Klassen ihre GUI und IAKs ihre GUI und FK kennen sollten. Dazu überprüfen die Queries, ob die jeweilige IAK oder die Monotool-Klasse eine GUI bzw. eine FK kennt, die den gleichen Namen trägt. Zu diesen Queries gibt es je eine FollowUp-Query, die in Kapitel 4.5.7 beschrieben wird.

Die Queries „The Monotool-class/Tool-class Should Initialize X“ testen, ob die Regeln befolgt werden, dass Monotool-Klassen ihre GUI erzeugen sollen und dass Tool-Klassen für das Erzeugen von IAK, FK und GUI zuständig sind. Dabei wird geprüft, ob sie GUIs, IAKs und FKs des gleichen Namens erzeugen. Beispielsweise sollte eine Monotool-Klasse `EditorTool` eine GUI `EditorGUI` erzeugen.

4.5.6 WAM-Single-Element-Rules

Queries, die nach Verstößen gegen Einzel-Element-Regeln suchen, sind in dieser Gruppe untergebracht. Sie suchen nach Verletzungen von sechs verschiedenen Regeln der WAM-Modellarchitektur. Sie werden in dieser Arbeit als „Einzel-Element-Queries“ bezeichnet.

„Materials Shouldn't Have "get"- And "set"-Methods Only” ist eine Query, die überprüft, ob es an der Schnittstelle von Materialien nur Getter und Setter gibt. Dazu sucht die Query nach Materialien, die nur Methoden haben, deren Namen entweder mit „get“ oder mit „set“ beginnen. Ist dies der Fall wird das entsprechende Material aufgelistet. Um Verstöße gegen diese Regel zu finden, gibt es nur die Möglichkeit sich an die gängigen Namenskonventionen zu halten. Da der Gebrauch von „get“ und„set“ etabliert ist, ist es sinnvoll, nach Regelverletzungen zu suchen, auch wenn es Systeme gibt, die diese Konvention nicht einhalten. Zu dieser Query gibt es eine FollowUp-Query (s. Kapitel 4.5.7).

Die Query „Complex Tools consist of GUI, FP, IP and Tool-class” überprüft, ob es zu jeder IAK auch FK, GUI und Tool-Klasse gibt. Es wird danach gesucht, ob es eine FK, GUI und Tool-Klasse gibt, die den gleichen Namen wie die IAK trägt. Wird ein zugehöriges Element nicht gefunden, wird die IAK in der Ergebnisliste aufgeführt. Zu dieser Query gibt es eine FollowUp-Query (s. Kapitel 4.5.7).

„FPs Shouldn’t Return Materials” ist eine Query, die nach Verstößen gegen die Regel sucht, dass FKs keine Materialien zurückliefern sollen. Hierzu werden die Rückgabetypen aller Methoden einer FK überprüft. In der Ergebnistabelle werden die FKs aufgelistet, die Materialien zurückliefern. Zu jeder FK werden die Methode und das Material aufgelistet, das von dieser Methode zurückgegeben wird.

Die Query „DVs Shouldn’t Have A Public Constructor“ sucht nach Verletzungen einer Regel, die unabhängig von Namenskonventionen überprüft werden kann. Die Query listet alle Fachwerte auf, die einen Konstruktor haben, der mit `public` deklariert ist.

Ob es Fachwerte gibt, die Operationen anbieten, die es erlauben, das Fachwert-Objekt zu verändern, wird mit der Query „DVs Shouldn’t Have “set”-Methods” festgestellt. Die Query überprüft dazu, ob es Fachwerte gibt, die Methoden mit dem String „set“ im Namen haben. Für diese Query gelten die gleichen Überlegungen zu Namenskonventionen wie bei der Query „Materials Shouldn’t Have “get“- And“set”-Methods Only“.

4.5.7 WAM-FollowUp

Alle FollowUp-Queries die zu WAM-Queries gehören, finden sich in dieser Gruppe. Zweck und Funktionsweise von FollowUp-Queries allgemein wurden in Kapitel 4.4.3 erläutert.

Queries der Gruppe WAM-FollowUp:

- Complex Tools FollowUp
- IPs Should Know Their GUI FollowUp
- Material Methods FollowUp
- Monotool-classes Should Know Their GUI FollowUp
- The Monotool-class Should Initialize The GUI FollowUp
- The Tool-class Should Initialize The GUI FollowUp
- The Tool-class Should Initialize The FP FollowUp
- The Tool-class Should Initialize The IP FollowUp

Zwei FollowUp-Queries gehören zu den Queries „IPs Should Know Their GUI“ und “Monotool-classes Should Know Their GUI”. Wenn zu einer IAK oder zu einer Monotool-Klasse keine passende GUI gefunden wurde, kann mit der FollowUp-Query ermittelt werden, ob die IAK oder die Monotool-Klasse überhaupt GUIs kennt.

Mit der FollowUp-Query zu der Query „Complex Tools Consist Of GUI, FP, IP And Tool-class“ kann ermittelt werden, welche Elemente mit dem gleichen Namen zu einer IAK gefunden wurden. Im Ergebnis dieser FollowUp-Query taucht mindestens eines der Elemente FK, GUI oder Tool-Klasse nicht auf. Dies liegt daran, dass die IAK im Ergebnis der ursprünglichen Query aufgelistet wurde, weil mindestens eines der zugehörigen Elemente nicht gefunden werden konnte.

Wurden mit der Query „Materials Shouldn't Have "get"- And "set"-Methods Only” Materialien gefunden, die nur get- und set-Methoden besitzen, so zeigt die FollowUp-Query „Material Methods FollowUp“ alle Methoden des ausgewählten Materials an.

Die FollowUp-Queries zu den Queries „Monotool-classes/Tool-classes Should Initialize Y“ zeigen für eine Monotool- oder Tool-Klasse an, welche Elemente sie erzeugen. Die FollowUp-Query „The Tool-class Should Initialize The FP FollowUp“ zeigt z.B. für eine Tool-Klasse alle FKs an, die von ihr erzeugt werden. Diese müssen nicht genauso heißen wie die Tool-Klasse, wie bei der Ausgangsquery „The Tool-class Should Initialize The FP“ überprüft wird.

4.6 Zusammenfassung

In diesem Kapitel wurde zunächst diskutiert, wie WAM-Elemente im Quelltext erkannt werden können. Es konnte für alle Elemente eine praktikable Möglichkeit der Identifizierung gefunden werden. Auf dieser Grundlage wurden Queries implementiert, die WAM-Systeme analysieren und die Regeln der WAM-Modellarchitektur überprüfen. Sie wurden in Abschnitt 4.5 vorgestellt.

Die erstellten Queries werden in den nächsten beiden Kapiteln anhand von Beispielsystemen auf ihre Praxistauglichkeit getestet.

Kapitel 5 Analyse des Pausenplanersystems

In diesem Kapitel werden die in Kapitel 4 vorgestellten Queries auf das Pausenplanersystem angewendet, um die Nützlichkeit der Regeln an einem Beispiel zu demonstrieren. Außerdem wird durch die Untersuchung deutlich, was bei der Auswertung der Ergebnisse der Queries zu beachten ist. Die Funde der Queries werden in tabellarischer Form dargestellt und erläutert. Zuvor wird die Architektur des Pausenplanersystems herausgearbeitet, da die Ergebnisse der Analyse verständlicher sind, wenn die grobe Struktur des Systems bekannt ist. In den letzten Abschnitten werden die Ergebnisse bewertet und das Kapitel zusammengefasst.

5.1 Die Architektur des Pausenplanersystems

Im Folgenden wird die Architektur des Pausenplanersystems beschrieben. Es wird dabei nicht auf kleine Elemente wie IAKs und FKs eingegangen, sondern auf die Zusammenhänge zwischen Werkzeugen, Services, Materialien und Automaten. Auch Fachwerte werden in der Beschreibung der groben Struktur des Systems nicht berücksichtigt.

In den folgenden Abbildungen und Beschreibungen wurden die Interfaces, die es im Pausenplanersystem gibt, nicht berücksichtigt. Auf die Klassen, die diese Interfaces implementieren, wird nicht über das Interface, sondern direkt zugegriffen. Die Interfaces spielen daher in der Gesamtarchitektur eine untergeordnete Rolle und hätten die Abbildungen und Beschreibungen nur unnötig verkompliziert.

5.1.1 Werkzeuge

Mit dem Pausenplanersystem kann die Beaufsichtigung von `Pausen` durch `Aufsichtskräfte` geplant werden. Die Anzeige des `Pausenplans` und die Einplanung von `Aufsichtskräften` erfolgt im Hauptwerkzeug des Pausenplanersystems, dem `Pausenplantableau`[3]. Das `Pausenplantableau` hat verschiedene Subwerkzeuge: Im `AK-Werkzeug` werden Informationen über eine `Aufsichtskraft` angezeigt. Mit der `AKVerwaltung` und dem `AKEditor2` können `Aufsichtskräfte` gelöscht, hinzugefügt und verändert werden. Der `Pauseneditor` ermöglicht es dem Benutzer, die Art und Anzahl der `Pausen` anzupassen, mit dem `Druck-Werkzeug` kann der `Pausenplan` gedruckt werden. Neben diesen gibt es noch kleinere Werkzeuge, die u.a. für Einstellungsänderungen verwendet werden. Auf sie wird in der folgenden Beschreibung der Architektur des Pausenplanersystems nicht eingegangen.

Abbildung 5.1 zeigt die Hauptwerkzeuge des Pausenplanersystems und ihre Beziehungen untereinander. Das `AK-Werkzeug` ist ein Subwerkzeug, das in die GUI des `Pausenplantableau-Werkzeug` eingebettet ist. Das `Druck-Werkzeug`, der `PausenEditor` sowie die `AKVerwaltung` sind Subwerkzeuge des `Pausenplantableau-Werkzeugs`, die in einem eigenen Fenster dargestellt werden. Der `AKEditor2` ist ein Subwerkzeug, das in die `AKVerwaltung` eingebettet ist. Es kann aus der `AKVerwaltung` oder aus dem `AK-Werkzeug` heraus gestartet werden.

---

[3] Die Namen der Werkzeuge ohne Elementsuffix stehen stellvertretend für alle Klassen des Werkzeugs. Z.B. gehören zum `Pausenplantableau-Werkzeug` die Klassen `PausenplantableauTool, PausenplantableauIAK, PausenplantableauFK und PausenplantableauIAK`.

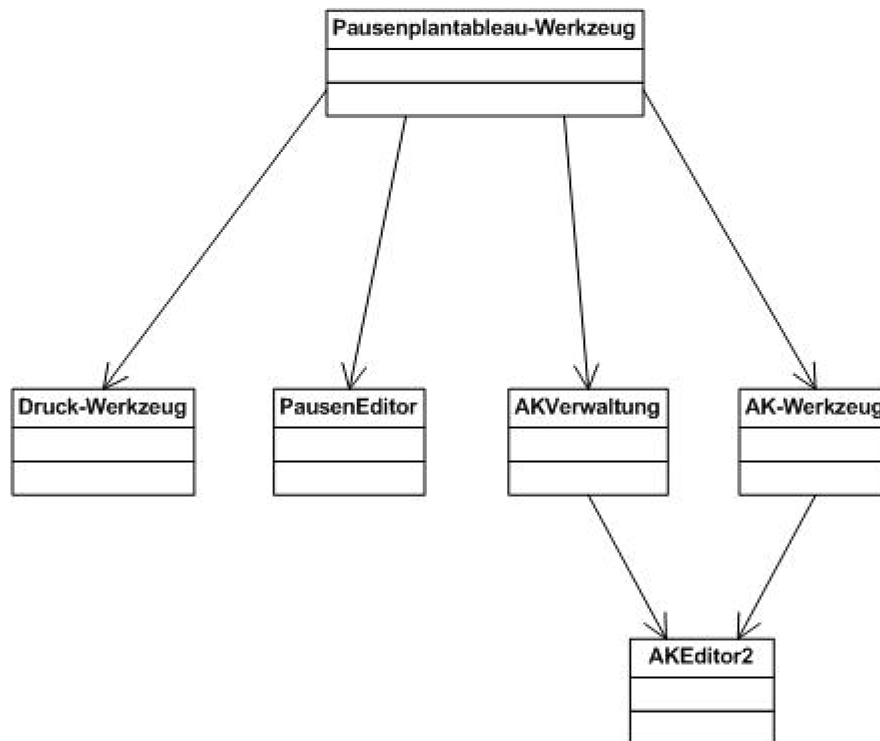

Abbildung 5.1: Die Hauptwerkzeuge im Pausenplanersystem

5.1.2 Materialien

Das Pausenplanersystem enthält sechs Materialien: `Pausenplan`, `Pause`, `Aufsicht`, `Kollegium`, `Aufsichtskraft` und `Rahmenbedingung`.

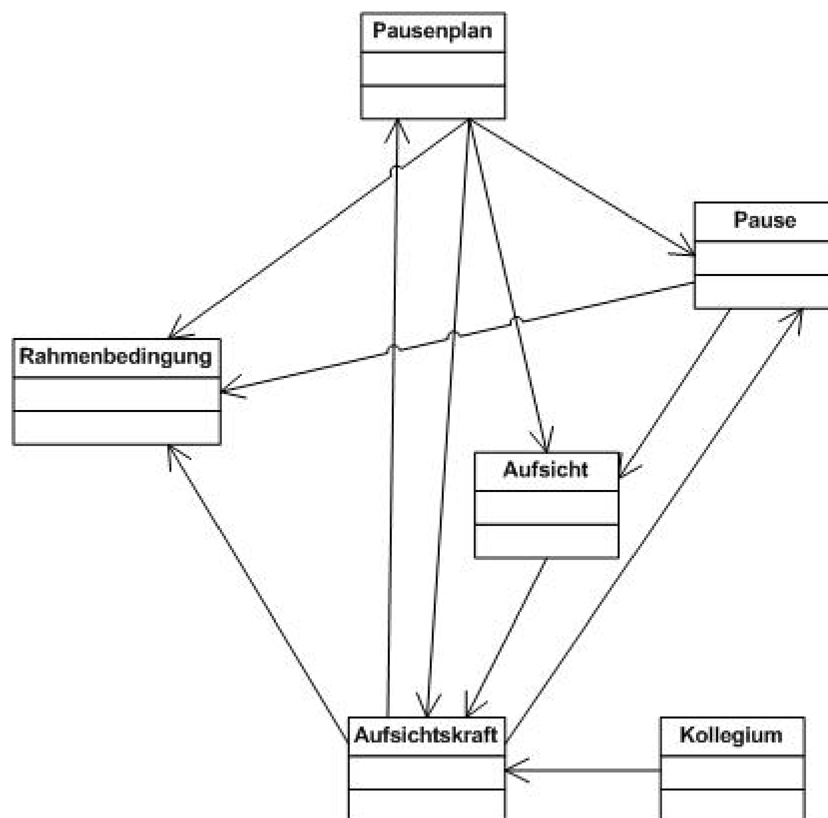

Abbildung 5.2: Materialien im Pausenplanersystem

Das `Kollegium` ist ein Behälter, der alle `Aufsichtskräfte` enthält.

Der `Pausenplan` ist unterteilt in `Pausen`, diese wiederum in `Aufsichten`. `Auf-sichten` sind Zeiträume, in die eine `Aufsichtskraft` eingeplant werden kann. `Rahmenbedingungen` gehören zu `Aufsichtskräften` und legen fest, wann eine `Aufsichtkraft` eingeplant werden kann oder wann Restriktionen die Einplanung verhindern. Die Beziehungen zwischen den Materialien des Systems sind in Abbildung 5.2 dargestellt.

5.1.3 Werkzeuge und Materialien

Abbildung 5.3 zeigt den Zugriff der verschiedenen Werkzeuge auf die Materialien `Kollegium`, `Aufsichtskraft`, `Rahmenbedingung`, `Pausenplan`, `Aufsicht` und `Pause`. Aus Gründen der Übersichtlichkeit wurden die Beziehungen zwischen Werkzeugen und Materialien untereinander weggelassen.

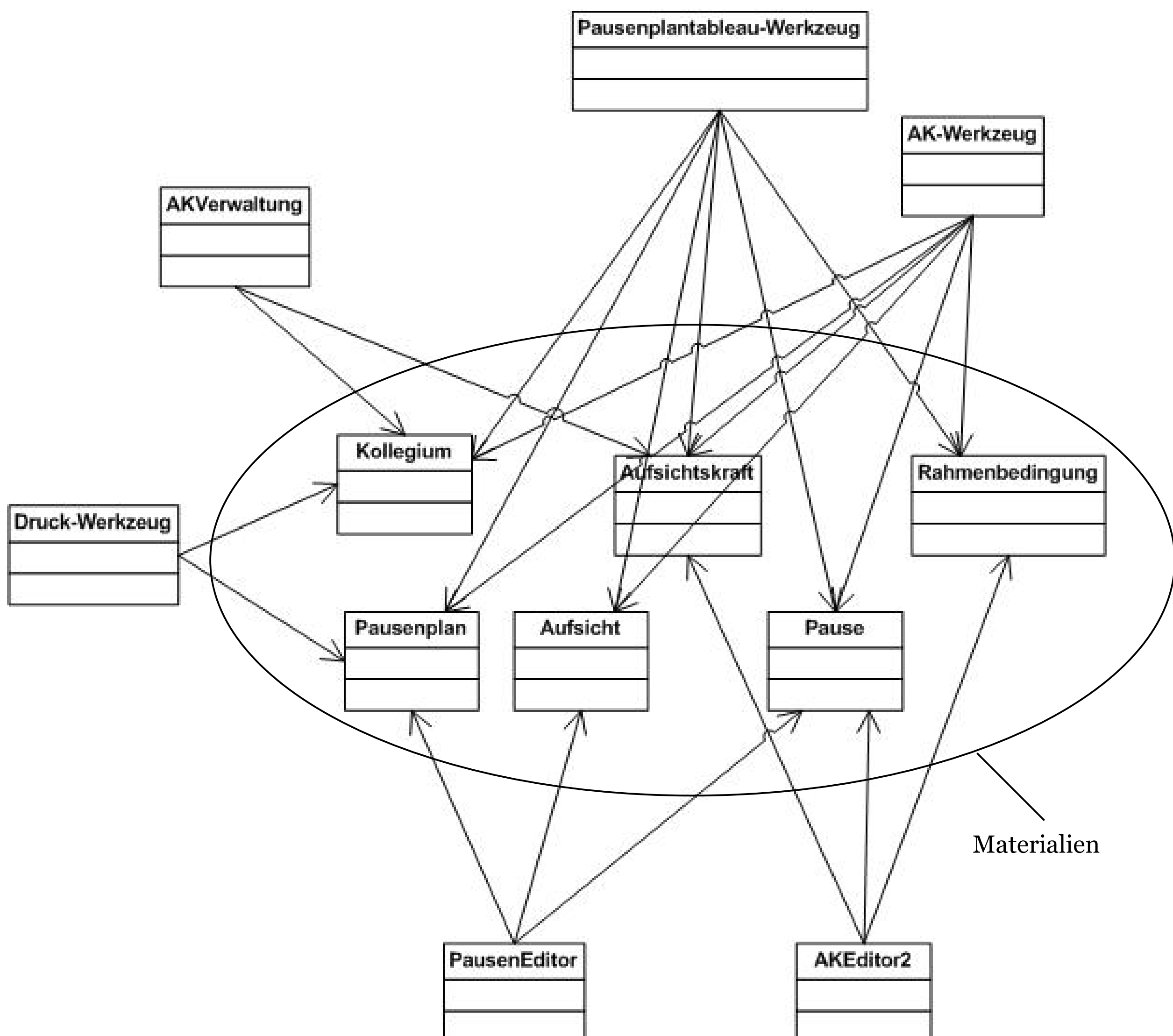

Abbildung 5.3: Materialien und Werkzeuge im Pausenplanersystem

Das `Pausenplantableau` ist das Hauptwerkzeug des Pausenplanersystems und greift ebenso wie das `AK-Werkzeug`, das als Subwerkzeug in das `Pausenplan-`

tableau-Werkzeug eingebettet ist, auf alle Materialien zu. Alle anderen Werkzeuge benutzen nur einen Teil der Materialien des Systems.

5.1.4 Werkzeuge, Materialien, Services und Automat

Es gibt im Pausenplanersystem zwei Services und einen Automat. Ihre Beziehungen zu Werkzeugen und Materialien werden in Abbildung 5.4 dargestellt. Relationen zwischen Werkzeugen und Materialien wurden der Übersichtlichkeit halber weggelassen. Weiterhin sind nur die Werkzeuge und Materialien in der Abbildung enthalten, die den Automaten oder die Services kennen.

Im oberen Teil der Abbildung befinden sich die Werkzeuge `PausenEditor`, `Pausenplantableau` und `AKVerwaltung`, die auf den `PausenplanService` bzw. den `KollegiumService` zugreifen. Der `PausenplanService` greift nur auf das Material `Pausenplan` zu, während der `KollegiumService` nur auf das Material `Kollegium` zugreift. Der `AusfuellAutomat` kennt und benutzt alle Materialien um den `Pausenplan` mit `Aufsichtskräften` zu füllen. Dafür greift er auch auf den `KollegiumService` zu.

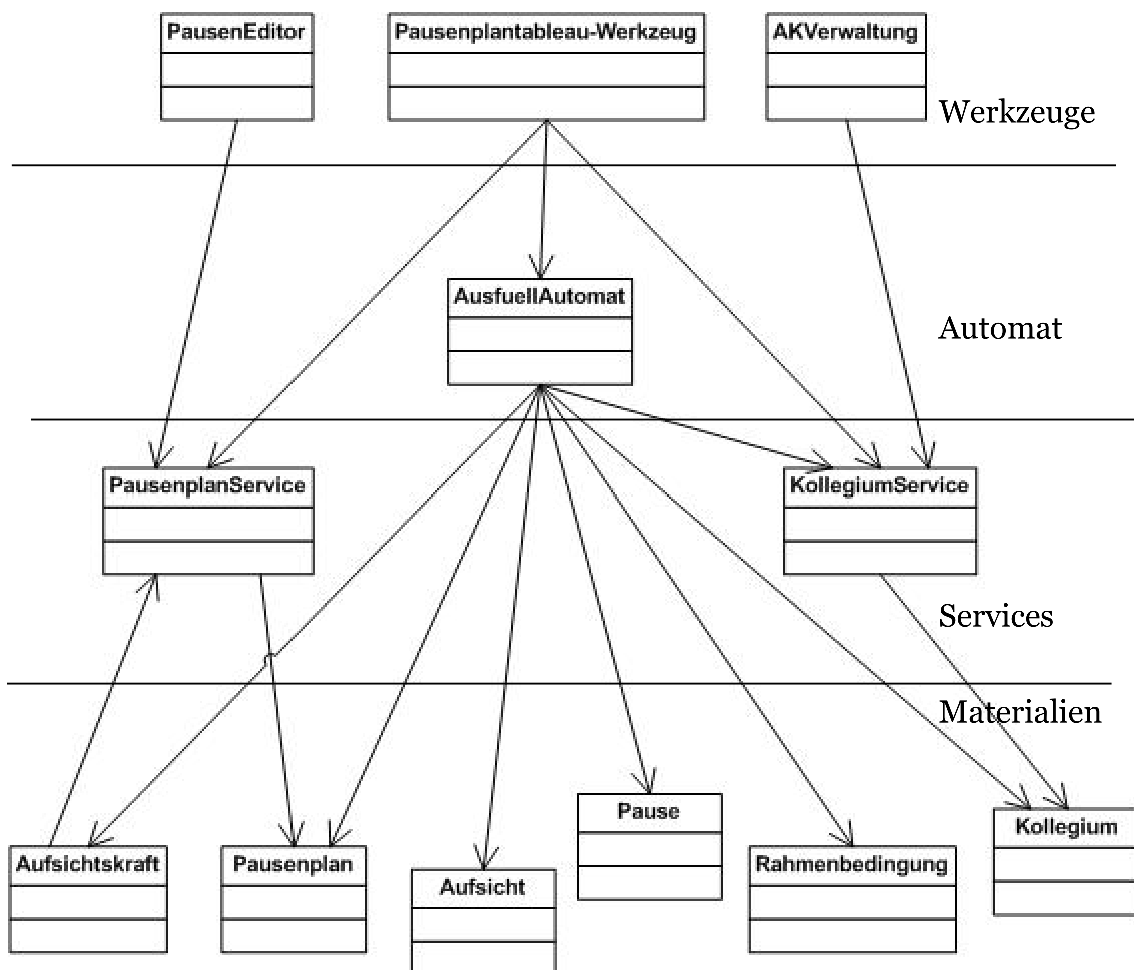

Abbildung 5.4: Werkzeuge, Services, Materialien und Automat

5.2 Ergebnisse der Analyse des Pausenplanersystems

In diesem Abschnitt werden die Ergebnisse der Analyse des Pausenplanersystems mit dem Sotographen vorgestellt. Dafür werden zunächst einige grundlegende Systemdaten aufgelistet, die mit dem Sotographen ermittelt wurden. Sie dienen dazu, eine Vorstellung über die Größe des Pausenplanersystems zu bekommen.

Im Weiteren werden die Ergebnisse der verschiedenen Queries dargestellt. Dazu werden in den einzelnen Abschnitten zunächst kurz die wichtigsten Hinweise zur Funktionsweise der Queries wiederholt. Es folgt die Tabelle mit den Ergebnissen der Queries und eine Erläuterung dieser Ergebnisse. Wie die Fehler behoben werden können, muss im Einzelnen von den Entwicklern des Systems entschieden werden. Es geht hier also nur um die Aufdeckung von Fehlern, nicht um deren Behebung. Nach den Ergebnissen der WAM-Queries folgen die im System entdeckten Zyklen.

Vor der Analyse des Pausenplanersystems wurden unbenutzte Artefakte und Testklassen entfernt. Diese Klassen haben keinen Einfluss auf die Funktionalität des Systems und sollten entfernt werden, da sie die Architektur des Systems verfälschen.

5.2.1 Allgemeine Systemdaten

Um die Anzahl der gefundenen Fehler sinnvoll interpretieren zu können, ist es zunächst notwendig, sich einen Überblick über das System zu verschaffen. Die folgende Tabelle enthält einige Metrikwerte, die mit dem Sotographen ermittelt wurden.

Metrik	Wert
SysLOC	25549
SysRelevantLines	12840
SysClasses	196
SysPackages	17
SysAnonymousClasses	95
SysMethods	1377

Tabelle 5.1: Allgemeine Daten des Pausenplanersystems

Die Metrik „SysLOC" zählt die Anzahl der Codezeilen im System. In „SysRelevantLines" werden nur die Zeilen gezählt, die mindestens einen Buchstaben enthalten der nicht auskommentiert ist. Dieser Wert beträgt ungefähr die Hälfte des Wertes von „SysLOC". Dieses ist laut Sotograph-Entwicklern und WAM-Architekten ein normales Verhältnis dieser beiden Werte.

Das Pausenplanersystem besitzt 196 Klassen, von denen allerdings nur 101 übrig bleiben, wenn man die anonymen Klassen abzieht. Die Klassen sind auf 17 Packages verteilt und haben insgesamt 1377 Methoden.

5.2.2 WAM-Elemente

Tabelle 5.2 stellt zusammen, welche Ergebnisse die Queries aus der Gruppe WAM-Elements lieferten.

Wie in Kapitel 4.1 beschrieben, werden die Elemente der WAM-Modellarchitektur über ihre Vererbungsbeziehung gefunden. Beispielsweise werden alle Klassen, die

von der Klasse `Automaton` erben, als Automat identifiziert. Eine Ausnahme bilden die Materialien. Diese werden darüber identifiziert, dass sie das Interface `Thing` implementieren und in einem Package liegen, das „material“ im Namen enthält.

Die Query „Tools (Ip, Fp, GUI, Tool-class, Monotool-class)“ listet alle Elemente auf, die zu Werkzeugen gehören.

Die Queries „Tools (Complex, grouped)“ und „Tools (Mono, grouped)”, listet zusammengehörige Werkzeugelemente auf. Es wird dafür nach Werkzeugelementen gesucht, die abgesehen von der Endung „-GUI“, „-Fk“ etc., den gleichen Namen tragen.

Die Spalte „Elementart“ in der Tabelle gibt an, nach welcher Art von WAM-Element die jeweilige Query sucht. Die zugehörigen Queries heißen ähnlich oder genau wie die gesuchte Elementart. „Elemente im Pausenplanersystem“ gibt an, wie viele Elemente der gesuchten Elementart im Pausenplaner gefunden wurden.

In den Spalten „Falsch Negative“ und „Falsch Positive“ werden Fehlinterpretationen der Queries erfasst. „Falsch Negative“ gibt an, wie viele Elemente eine Query „übersehen“ hat, d.h. Elemente, die sie hätte auflisten müssen, die sie aber nicht finden konnte. „Falsch Positive“ zeigt, wie viele Elemente in dem Ergebnis der Query auftauchen, die fälschlicherweise als diese Elemente identifiziert wurden. Im Pausenplanersystem gab es nur drei „Falsch Negative“ jedoch keine „Falsch Positiven“. Auf die „Falsch Negativen“ wird in der Erläuterung des jeweiligen Queryergebnisses eingegangen.

Elementart	**Elemente im Pausenplanersystem**	**Falsch Negative**	**Falsch Positive**
Materialien	6	0	0
Funktionskomponenten	6	0	0
Interaktionskomponenten	6	0	0
Tool-Klassen	6	0	0
GUIs	10	0	0
Monotool-Klassen	4	0	0
Services	3	0	0
Automaten	1	0	0
Fachwerte	5	0	0
Tools (Ip, Fp, GUI, Tool-class, Monotool-class)	32	0	0
Tools (Complex, grouped)	4	2	0
Tools (Mono, grouped)	3	1	0

Tabelle 5.2: WAM-Elemente im Pausenplanersystem

Es gibt in diesem System sechs Materialien und zehn Werkzeuge, die sich aufteilen in sechs komplexe Werkzeuge, bestehend aus FK, IAK, Tool-Klasse und GUI und vier monolithische Werkzeuge, bestehend aus Monotool-Klasse und GUI. Weiterhin gibt es drei Services, einen Automaten und fünf Fachwerte.

Das Ergebnis der Query „Tools (Ip, Fp, GUI, Tool-class, Monotool-class)“ zeigt, dass es 32 Werkzeugelemente im System gibt. Diese Summe ergibt sich auch aus der Summe der gefundenen Funktionskomponenten, Interaktionskomponenten, GUIs, Tool-Klassen und Monotool-Klassen.

Die Query „Tools (Complex, grouped)“, hat von vier Werkzeugen die richtigen Werkzeugteile zusammengestellt. Das `AKVerwaltung2-Werkzeug` und das `Druck3-Werkzeug` wurden nicht gefunden. Dies liegt daran, dass die Namensbezeichnung von den gewöhnlichen Bezeichnungen abweicht, wie im Folgenden erläutert wird.

Bei dem `AKVerwaltung-Werkzeug` hat die Tool-Klasse kein Suffix „Tool“ im Namen. D.h. die zusammengehörigen Werkzeugelemente heißen `AKVerwaltung2`, `AK-Verwaltung2IAK`, `AKVerwaltung2FK` und `AKVerwaltung2Gui`. Die Tool-Klasse konnte durch das fehlende Suffix „Tool“ im Namen nicht gefunden werden.

Die Elemente des `Druck-Werkzeugs` haben am Ende des Namens jeweils eine „3“ stehen, es gibt die Elemente `DruckIAK3`, `DruckFK3`, `DruckGui3` und `DruckTool3`. Durch die „3“ am Ende des Namens konnten die Werkzeuge einander nicht zugeordnet werden.

Bei der Suche nach Monotool-Klassen und zugehöriger GUI mit der Query „Tools (Mono, grouped)“, wurde nur das `Einstellung-Werkzeug` nicht aufgelistet, da hier wie bei den zwei komplexen Werkzeugen von der üblichen Namensgebung abgewichen wurde. Die GUI zum `EinstellungTool` heißt „EinstellungToolGui“ statt nur „EinstellungGui“.

5.2.3 Vererbungsfehler

Die folgende Tabelle zeigt die Ergebnisse der Queries, die überprüfen, ob die WAM-Elemente im System von den richtigen Klassen in JWAM erben bzw. die richtigen Interfaces implementieren. Es werden nur die Queries aufgeführt, die Klassen gefunden haben.

Fast alle Queries dieser Gruppe suchen nach Klassen, die bestimmte Strings im Namen haben. Daher sind diese Ergebnisse nicht unbedingt zuverlässig und müssen genau geprüft werden. So wird z.B. überprüft, ob jede Klasse, die ein „dv“ im Namen hat, das Interface `DomainValue` implementiert.

Einzig bei der Materialien-Query werden Klassen nicht nach ihren Namen ausgewählt, da es für Materialien keine Namenskonvention gibt, an der man sie erkennen könnte. Mit der Materialien-Query werden Klassen gesucht, die in einem Package liegen, dessen Name „material“ enthält und die nicht das Interface `Thing` implementieren. Dieses Verfahren funktioniert beim Pausenplanersystem und bei den anderen untersuchten Systemen, da die Materialien dort in einem Package „material“ liegen.

In der ersten Spalte der Tabelle findet sich die Elementart, nach der gesucht wurde. Sie korrespondiert mit dem Namen der zugehörigen Query. Die Spalte „Anzahl der Funde“ gibt an, wie viele Klassen die Query als Ergebnis geliefert hat. Wie viele der Funde wirklich Klassen sind, bei denen die korrekten Vererbungsbeziehungen nicht eingehalten wurden, ist in der Spalte „Echte Vererbungsfehler“ aufgelistet. Die Zahlen aus dieser Spalte sind ein Versuch, die Query-Ergebnisse aufgrund der Ergebnislisten im Sotographen und mit einem kurzen Blick in den Quelltext zu interpretieren. Eine sichere Interpretation der Ergebnisse würde eine gründliche Quelltextanalyse erfordern.

Elementart	Anzahl der Funde	Echte Vererbungsfehler
Interaktionskomponenten	2	0
Materialien	5	1
Services	1	1
Tool-Klassen	4	0
Insgesamt	12	2

Tabelle 5.3: Vererbungsfehler im Pausenplanersystem

Die Klassen `PlanDescriptor` und `IPausenplanService` wurden von der Interaktionskomponenten-Query gefunden. Die Klassen liegen im Package „Material" und im Package „Service". An den Packages und daran wo der String „IP" im Namen zu finden ist, kann man erkennen, dass es sich hier nicht um Interaktionskomponenten handelt. Die Funde stellen also keine echten Vererbungsfehler dar.

Die Material-Query hat fünf Klassen gefunden. Eine von ihnen ist eine innere anonyme Klasse des Materials `Pausenplan`. Dieser Fund ist kein Fehler, da auf die innere Klasse nicht von außen zugegriffen werden kann und sie daher nicht als eigenständiges Material gedacht ist.

Weiterhin wurde bei der Material-Query das Interface `Paintable` gefunden, das von den Materialien `Kollegium` und `Pausenplan` implementiert wird. Der Name legt nahe, dass das Interface für das `Druck-Werkzeug` erstellt wurde. Soll das `Druck-Werkzeug` die Materialien `Pausenplan` und `Kollegium` über dieses Interface kennen, so sollte es auch das Interface `Thing` enthalten. Das `Druck-Werkzeug` greift jedoch auch direkt auf die Materialien zu, dies sollte vermieden werden. Die Beziehungen zwischen dem `Druck-Werkzeug`, der Klasse `Pausenplan` und dem Interface `Paintable` sind in Abbildung 5.5 dargestellt. Die Beziehungen des `Druck-Werkzeugs` zur Klasse `Kollegium` sind analog zu den Beziehungen des `Druck-Werkzeugs` zur Klasse `Pausenplan`.

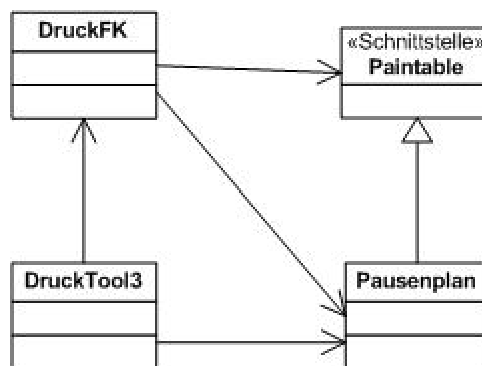

Abbildung 5.5: `Druck-Werkzeug`, `Pausenplan` und Interfaces

Weiterhin wurden zwei Klassen namens `KollegiumData` und `PausenplanData` gefunden, die jeweils nur von `Kollegium` bzw. `Pausenplan`" benutzt werden. Die „Data"-Klassen erben jeweils von `AbstractTableModel` und werden laut Klassenkommentaren für „JFreeReport", also für das Drucken benötigt. Da die Klassen nur von den Materialien `Kollegium` und `Pausenplan` und nicht von Werkzeugen, Services oder Automaten benutzt werden und keinerlei fachliche Methoden haben, scheinen sie technische Hilfsklassen zu sein und müssen nicht das Interface `Thing`

implementieren. Die Klassen sollten jedoch aus dem Materialpackage ins Util-Package verschoben werden.

Die letzte Klasse, die von der Material-Query gefunden wurde, ist `PlanDescriptor`, eine Klasse, die von der Klasse `PersistenzGui` benutzt wird. Diese gehört zum `PersistenzTool`, welches keine weiteren Materialien benutzt. Laut Klassenkommentar ist der `PlanDescriptor` eine „Zwischenschicht für Anzeige in der GUI". Es scheint sich hier also nicht um ein Material zu handeln. Es bleibt allerdings die Frage, warum sich diese Klasse im Material-Package befindet. Eine Klärung dieser Frage aus dem Quelltext heraus ist nicht möglich.

Die Service-Query liefert als Ergebnis eine Klasse namens `IKollegiumService`. Sieht man sich die Vererbungsstruktur des `KollegiumService` neben der Vererbungsstruktur des `PausenplanService` an (s. Abbildung 5.6), so ist diese weitestgehend symmetrisch. Nur `IKollegiumService` erbt nicht wie `IPausenplanService` von `ServiceProvider`. Es scheint sich also um einen Fehler zu handeln. Die Interfaces für die Services sollten auch das Interface `ServiceProvider` enthalten, da es sich um Interfaces handelt, die Services darstellen.

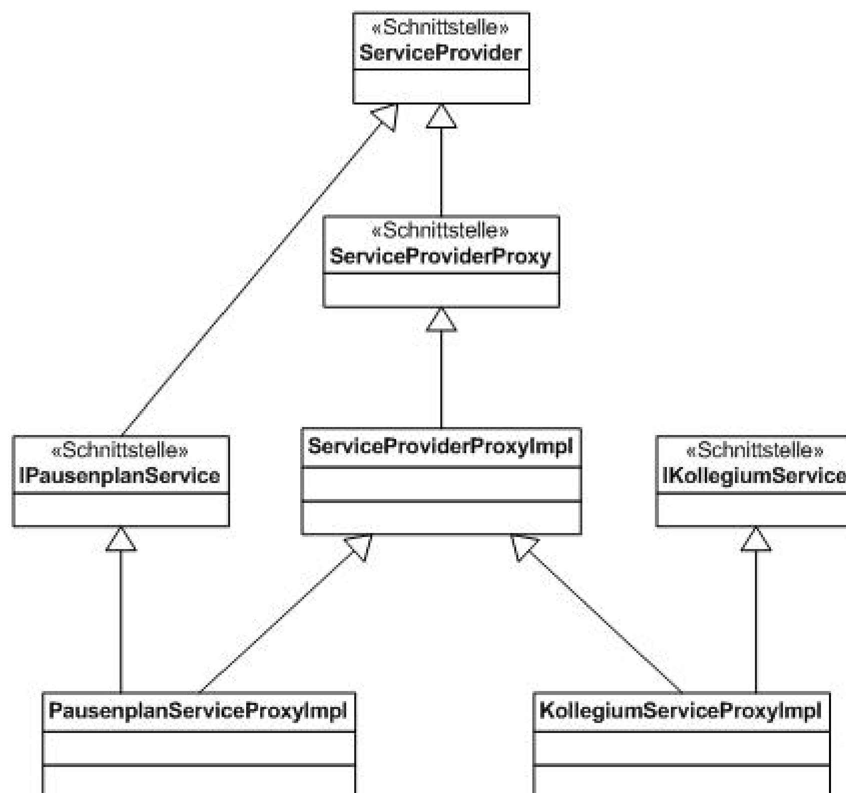

Abbildung 5.6: Service-Vererbungshierarchie im Pausenplaner

Die vier Funde der Tool-Klassen-Query sind keine Tool-Klassen, haben aber „Tool" im Namen, wie z.B. `AKToolIAK`. Da alle dieser Klassen entweder „FK", „IAK" oder „GUI" im Namen haben, besteht kein Zweifel, dass es keine Tool-Klassen sein sollen. Es liegt also kein Vererbungsfehler vor, aber eine Verletzung der Namenskonventionen.

5.2.4 WAM-Regelverletzungen

Dieser Abschnitt behandelt die Verletzungen von WAM-Regeln, die im Pausenplanersystem gefunden wurden. Es werden zunächst die Ergebnisse der Verbots-Queries aufgeführt. Es folgen die Funde von Gebots-Queries und von Einzel-Element-Queries. Es werden jeweils nur die Queries aufgelistet, die Verstöße gefunden haben.

5.2.4.1 Verstöße gegen Verbots-Beziehungsregeln

Tabelle 5.4 zeigt die Ergebnisse für Queries aus der Gruppe WAM-Forbidden-Relationships. Jede Query prüft, ob zwischen den in der Regel genannten Elementen Beziehungen bestehen.

Die Zahl in der Spalte „Gesamtanzahl Funde“ setzt sich aus allen Einzelverstößen im Quelltext zusammen. In der Spalte „Anzahl Klassenbeziehungen“ ist die Anzahl der Klassenpaare zu finden, zwischen denen eine verbotene Beziehung besteht. Nicht alle Funde sind Regelverletzungen, einige können zugelassene Ausnahmen sein. Wieviele der Klassenbeziehungen echte Verletzungen einer WAM-Regel sind, zeigt die Spalte „Echte Verletzungen (Klassenbeziehungen)“. Wie bei den Vererbungsfehlern, können auch hier einige Ausnahmen und echte Verletzungen nur durch eine gründliche Quelltextanalyse oder einen Entwickler des Systems unterschieden werden. Es handelt sich also um einen Versuch, die Ergebnisse der Queries zu interpretieren, der mit Vorsicht zu genießen ist.

Query	**Gesamtanzahl Funde**	**Anzahl Klassenbeziehungen**	**Echte Verletzungen (Klassenbeziehungen)**
FPs Shouldn't Know Tool-classes	2	1	0
GUIs Shouldn't Know Materials	66	5	5
GUIs Shouldn't Know Services	5	1	1
GUIs Shouldn’t Know Tools	14	2	0
IPs Shouldn't Know Automatons	14	1	1
IPs Shouldn't Know Materials	413	13	13
IPs Shouldn’t Know Tool-classes	3	2	2
Materials Shouldn't Know Services	2	1	1
Tool-classes Shouldn't Know Services	9	1	1
Insgesamt	528	27	24

Tabelle 5.4: Verstöße gegen Verbots-Beziehungsregeln im Pausenplanersystem

Es gibt insgesamt 528 Beziehungen zwischen Klassen, die nicht erlaubt sind. In diesem Ergebnis ist jeder verbotene Zugriff enthalten, d.h. mehrere Ergebnisse zeigen evtl. eine verbotene Beziehung zwischen den gleichen Klassen. Z.B. greift die Klasse `PausenEditorIAK` auf die Methoden `istEinzelaufsicht()` und `getStatus()` des Materials `Aufsicht` zu. Diese Zugriffe werden getrennt aufgeführt, es handelt

sich aber auf Klassenebene gesehen nur um eine verbotene Beziehung nämlich zwischen der Klasse `PausenEditorIAK` und der Klasse `Aufsicht`. Wenn man sich im Sotographen ansieht, zwischen welchen Paaren von Klassen eine verbotene Beziehung besteht, so finden sich insgesamt 27 Paare von Klassen. Bei genauer Überprüfung der Funde haben allerdings nur 24 Paare von Klassen eine verbotene Beziehung. Drei Beziehungen sind zugelassene Ausnahmen.

Die einzigen Queries, deren Funde Ausnahmen sind, sind die Queries „FPs Shouldn't Know Tool-classes" und „GUIs Shouldn't Know Tools". Bei dem Fund der Query „FPs Shouldn't Know Tool-classes" handelt es sich um eine erlaubte Beziehung zwischen einer Kontext-FK und einer Subwerkzeug-Tool-Klasse. Die Query „GUIs Shouldn't Know Tools" hat Beziehungen zwischen zwei Paaren von GUI-Klassen gefunden, die erlaubte Vererbungsbeziehungen sind.

Auffällig ist, dass die meisten Verletzungen von Regeln aus einer oder zwei verbotenen Klassenbeziehungen bestehen. Im Gegensatz dazu wurde die Regel *„IAKs sollten keine Materialien kennen"* 413-mal verletzt. Diese Verletzungen bestehen zwischen 13 Paaren von Klassen. Nicht so viele Verletzungen gibt es für die Regel *„GUIs sollten keine Materialien kennen"*. Mit 66 Verstößen gegen diese Regel und fünf Paaren von Klassen zwischen denen eine verbotene Beziehung besteht, sticht sie jedoch auch aus den Ergebnissen heraus. Diese Zahlen scheinen darauf hinzudeuten, dass diese Regeln entweder nicht bekannt waren, nicht verstanden wurden oder einfach ignoriert wurden. Versehentliche Missachtungen der Regeln sind aufgrund der Anzahl der Verstöße unwahrscheinlich.

5.2.4.2 Verstöße gegen Gebots-Beziehungsregeln

Tabelle 5.5 listet die Funde der Gebots-Queries auf.

Die Queries „IPs Should Know Their FP/GUI" listen IAKs auf, die keine FK bzw. GUI des gleichen Namens kennen. Mit einer FollowUp-Query kann zu jeder aufgelisteten IAK ermittelt werden, welche FKs bzw. GUIs sie kennt. Dabei wird nur nach Beziehungen von der IAK zu FKs/GUIs gesucht, die Namen werden nicht berücksichtigt. Wenn zu einer IAK keine FK/GUI des gleichen Namens gefunden wird, kann also geprüft werden, ob die zugehörige FK/GUI nur anders heißt, oder ob die IAK womöglich gar keine FK/GUI kennt. Das gleiche Prinzip wird bei der Query „Monotool-classes Should Know Their GUI" angewandt.

Die Query „Monotool-classes Should Know Materials" listet Monotool-Klassen auf, die keine Beziehung zu Materialien haben.

Die Queries „The Tool-class Should Initialize The GUI/FP/IP" und „The Monotool-class Should Initialize The GUI" listen die Tool-Klassen oder Monotool-Klassen auf, die kein entsprechendes Element initialisieren, das genauso heißt wie die Tool-Klasse/Monotool-Klasse.

Query	Gesamtanzahl Funde	Echte Verletzungen
IPs Should Know Their FP	1	0
IPs Should Know Their GUI	1	0
Monotool-classes Should Know Materials	3	0
Monotool-classes Should Know Their GUI	1	0
The Monotool-class Should Initialize The GUI	2	0
The Tool-class Should Initialize The FP	2	0
The Tool-class Should Initialize The GUI	2	0
The Tool-class Should Initialize The IP	2	0
Insgesamt	14	0

Tabelle 5.5: Verstöße gegen Gebots-Beziehungsregeln im Pausenplanersystem

Das Ergebnis der Query „IPs Should Know Their FP“ ist keine echte Verletzung. Es handelt sich um eine Abweichung von der Namenskonvention, so dass der IAK keine FK des gleichen Namens zugeordnet werden konnte. Die FollowUp-Query zeigte aber, dass die IAK eine FK kennt, die zu ihr gehört.

Die Queries „IPs Should Know Their GUI“ und „Monotool-classes Should Know Their GUI” haben eine IAK und eine Monotool-Klasse gefunden, die keine GUI des gleichen Namens kennen. Beim Ausführen der zugehörigen FollowUp-Query stellt sich heraus, dass sie andere GUIs kennen. Es handelt sich also nicht um IAK und Monotool-Klasse ohne GUI, sondern um eine Namensgebung die nicht der Konvention entspricht, so dass die jeweilige GUI der IAK bzw. der Monotool-Klasse vom Namen her nicht zugeordnet werden kann.

Bei den Funden der Query „Monotool-classes Should Know Materials“ handelt es sich um drei kleine Werkzeuge. Eines ist das abstrakte `PersistenzTool`, von dem der zweite Fund, das `LoadMonoTool` erbt. Der dritte Fund ist das `EinstellungTool`. Schließt man von den Namen auf die Funktionen der Werkzeuge, so liegt der Verdacht nahe, dass es sich um Ausnahmen handelt, und diese Werkzeuge keine Materialien kennen sollen.

Die Query „The Monotool-class Should Initialize The GUI“ hat zwei Monotool-Klassen gefunden. Beide Funde sind keine echten Verletzungen. Die Klasse `Persistenztool` ist eine abstrakte Oberklasse für das `LoadMonotool` und das `SaveMonotool`. Sie muss daher keine GUI selber erzeugen. Der zweite Fund ist die Klasse `EinstellungTool`. Die FollowUp-Query ergibt, dass das `EinstellungTool` eine GUI namens `EinstellungToolGui` erzeugt. Hier liegt also nur eine Verletzung der Namenskonvention vor, da die GUI eigentlich „EinstellungGui“ heißen müsste.

Die Queries „The Tool-class Should Initialize X“ finden alle die gleichen Tool-Klassen, nämlich `AKVerwaltung2` und `DruckTool3`. Es handelt sich hier in allen drei Fällen nicht um echte Verletzungen, sondern um eine Abweichung von der Namenskonvention. Die Klasse `AKVerwaltung2` initialisiert die Klassen `AKVerwaltung2Gui`, `AKVerwaltung2FK` und `AKVerwaltung2IAK`. Da der Klasse `AKVerwaltung2` die Endung „Tool“ fehlt, können diese Klassen ihr aber vom Namen her nicht zugeordnet werden. Die Klasse `DruckTool3` wurde aufgelistet, weil hier die Endung „Tool“ vor der „3“ im Namen steht und sie daher den Klassen `DruckGui3`, `DruckIAK3` und `DruckFK3` nicht zugeordnet werden kann.

5.2.4.3 Verstöße gegen Einzel-Element-Regeln

Tabelle 5.6 zeigt die Ergebnisse der Queries, die überprüfen, ob Einzel-Element-Regeln eingehalten werden.

Die Query „Complex Tools Consist Of GUI, FP, IP And Tool-class" listet IAKs auf, zu denen nicht alle dazugehörigen Werkzeugelemente gefunden werden konnten. Die Identifizierung findet über den Namen abzüglich der Endungen „-GUI", „-FK", etc. statt. D.h. zur IAK `PausenEditorIAK` wird nach Elementen `PausenEditorFK`, `PausenEditorTool` und `PausenEditorGui` gesucht. Wird eines der Elemente nicht gefunden, wird die IAK im Ergebnis aufgelistet. Mit einer FollowUp-Query kann ermittelt werden, welche Werkzeugelemente des gleichen Namens zu einer IAK gefunden wurden.

„DVs Shouldn't Have „set"-Methods" sucht nach Fachwerten, die Methoden haben, deren Namen mit „set" beginnen. Jede Methode wird einzeln mit der dazugehörigen Klasse aufgelistet.

Gibt es Methoden von FKs, die ein Material zurückliefern, so werden sie von der Query „FPs Shouldn't Return Materials" mit der dazugehörigen FK aufgelistet.

Die Gesamtzahl der Funde stimmt bei der ersten Query dieser Tabelle mit der Anzahl gefundener Klassen überein. Bei den beiden anderen Queries werden die Anzahl der Fachwertklassen, bzw. die Anzahl der FK-Klassen aufgelistet, die gegen die jeweilige Regel verstoßen. „Echte Verletzungen" sind wie in der Tabelle im vorigen Abschnitt der Versuch die Ergebnisse zu interpretieren.

Query	**Gesamtanzahl Funde**	**Anzahl gefundener Klassen**	**Echte Verletzungen**
Complex Tools Consist Of GUI, FP, IP And Tool-class	2	2	0
DVs Shouldn't Have "set"-Methods	8	5	5
FPs Shouldn't Return Materials	17	5	5
Insgesamt	27	12	10

Tabelle 5.6: Verstöße gegen Einzel-Element-Regeln im Pausenplanersystem

Die gelieferten Funde der Query „Complex Tools Consist Of GUI, FP, IP And Tool-class" sind die Klassen `AKVerwaltung2IAK` und `DruckIAK3`. Die Funde sind auf ungewöhnliche Namensgebungen zurückzuführen, die schon in Kapitel 5.2.2 bei der Erläuterung der Query „Tools (Complex, grouped)" erläutert wurden. Es handelt sich also nicht um echte Regelverletzungen.

Im Pausenplanersystem gibt es fünf Fachwert-Klassen, die Methoden haben, deren Namen mit „set" beginnen. Bei näherer Untersuchung zeigt sich, dass dies wirklich Methoden sind, mit denen das Fachwert-Objekt verändert werden könnte. Es liegen hier also Verstöße gegen die Regel *„Fachwert-Klassen bieten keine Operationen an, die es erlauben, das Fachwert-Objekt zu verändern"* vor.

Die Query „FPs Shouldn't Return Materials“ hat fünf Funktionskomponenten gefunden, die Methoden besitzen, die ein Material als Rückgabewert liefern. Dies verstößt gegen die Regel *„Die FK bietet über eine materialunabhängige Schnittstelle Informationen zur Präsentation des Materials für die IAK an“*. Es werden von diesen fünf FKs insgesamt fünf der sechs verschiedenen Materialien des Pauseplanersystems zurückgeliefert.

5.2.5 Zyklen

Als nächstes folgt eine Tabelle, die die mit dem Sotographen gefundenen Zyklen auflistet.

Zyklusart	Anzahl Zyklen	Anzahl Klassen bzw. Packages im Zyklus
Klassenzyklen	2	3/14
Packagezyklen	1	11

Tabelle 5.7: Zyklen im Pausenplanersystem

Es gibt im Pausenplanersystem insgesamt nur drei Zyklen, was im Vergleich zu anderen Systemen „normal“ scheint. Der Packagezyklus erstreckt sich über mehr als die Hälfte aller Packages im System und ist im Vergleich zu anderen Systemen sehr groß. Neben der Frage, ob es viele Zyklen gibt und wie groß sie sind, spielt bei der Bewertung von Zyklen die Frage, wie leicht sie zu entfernen sind, eine wichtige Rolle. Um einen Zyklus aufzulösen, kann es ausreichen eine Beziehung zu entfernen. Es kann aber auch sein, dass viele Verbindungen den Zyklus bilden. Zyklen mit wenig verursachenden Relationen sind leichter zu beheben als andere. Durch wie viele Beziehungen ein Zyklus entsteht, ist aber schwerer zu erkennen, je größer der Zyklus ist. Wie stark Zyklen die Wartung und Erweiterung eines Systems beeinflussen, hängt also von der Anzahl und der Größe der Zyklen, aber auch von der Anzahl der verursachenden Beziehungen ab. Ob ein Zyklus schwer oder leicht aufzulösen ist, kann durch eine gründliche Analyse beantwortet werden.

Ein wunder Punkt bei diesem System ist der große Packagezyklus, der aus der Hälfte aller Packages besteht. Aber auch die beiden Klassenzyklen sollten beseitigt werden. Zyklen haben die Eigenschaft, dass sie Systeme unflexibler und schwerer wartbar machen. Komponenten sind durch Zyklen stark gekoppelt und können nicht ausgetauscht werden ohne dass es Einfluss auf die Komponenten hätte, mit denen sie in einem Zyklus stehen, vgl. dazu [Bisch03], [Mar96] und [Gamma98]. Kleine Zyklen erscheinen nicht schlimm, sie haben jedoch die Tendenz größer zu werden. Da sich kleine Zyklen oft recht einfach beheben lassen, sollte man nicht damit warten sie zu entfernen. Einen Zyklus zu entwirren, an dem viele Klassen oder Packages beteiligt sind, bereitet oft viel größere Schwierigkeiten, als mehrere kleine aufzulösen.

Einer von mehreren Gründen für den großen Klassenzyklus ist die Nichteinhaltung einer WAM-Regel. Das Material `Aufsichtskraft` greift auf den Service `PausenplanServiceProxyImpl` zu. Diese beiden Klassen befinden sich in einem Zyklus mit 14 anderen Klassen, den Abbildung 5.7 zeigt.[4] Wird der Verstoß gegen diese Regel beseitigt, so teilt sich der Zyklus in drei kleinere Zyklen mit je vier, drei und zwei Klassen auf. Es sind also von den ursprünglich 14 nur noch neun Klassen in Zyklen,

[4] Die Graphik wurde mit dem Sotographen erstellt. Die Dicke der Pfeilspitzen ist abhängig von der Anzahl der Beziehungen zwischen den Klassen. Vererbungsbeziehungen werden wie andere Beziehungen nur mit einer einfachen Pfeilspitze dargestellt.

die in Abbildung 5.8 dargestellt sind. Die Zyklen sind jetzt wesentlich kleiner und dadurch einfacher zu beheben als vorher.

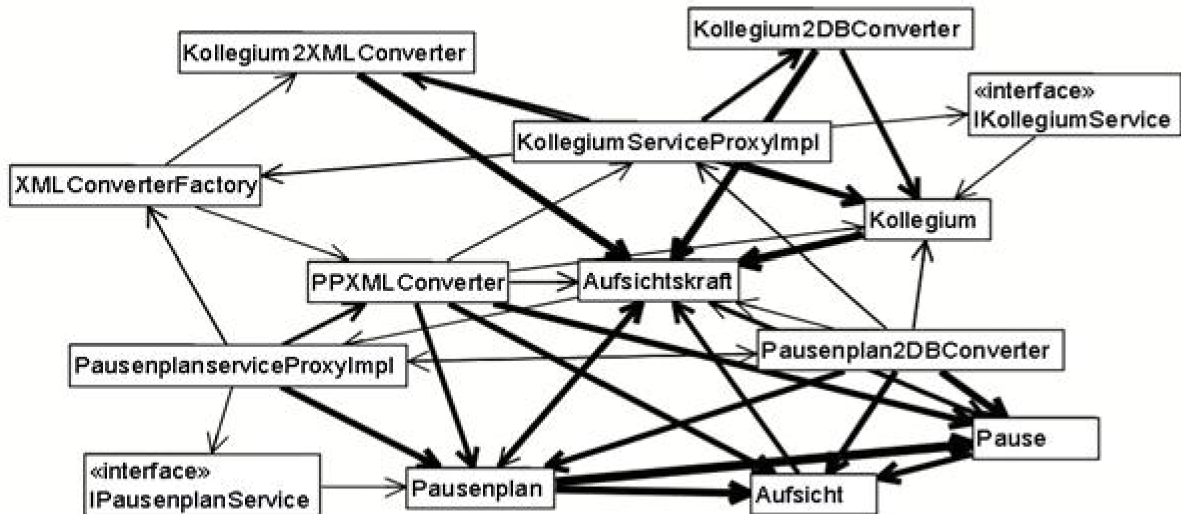

Abbildung 5.7: Großer Klassenzyklus im Pausenplanersystem

Der Zugriff des Materials auf den Service ist eine Ursache für den Packagezyklus im Pausenplanersystem. Entfernt man den Zugriff des Materials auf den Service, bleibt der Zyklus bestehen, da er noch durch andere Beziehungen verursacht wird. Eine Ursache des Zyklus wäre aber beseitigt. Dadurch wäre er leichter aufzulösen als mit dem Zugriff des Materials auf den Service.

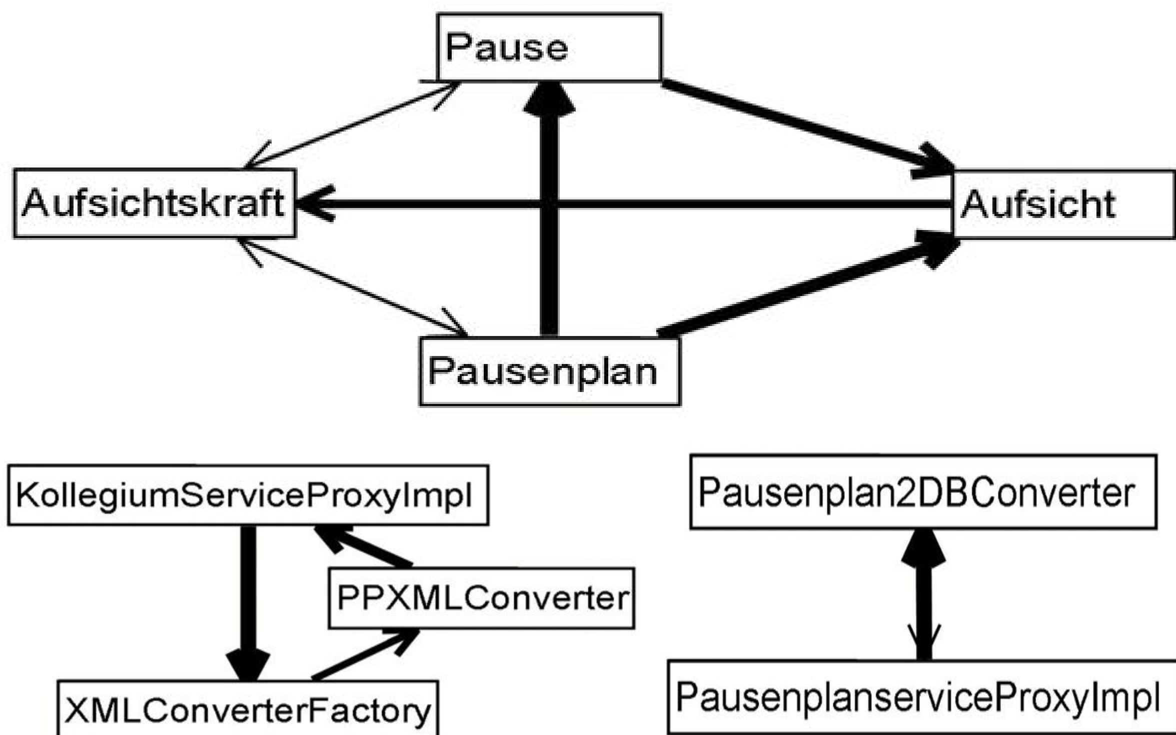

Abbildung 5.8: Drei kleine Zyklen entstanden aus dem großen Klassenzyklus im Pausenplanersystem

5.3 Bewertung der Ergebnisse

Die Ergebnisse der Queries, die Regelverstöße suchen, haben gezeigt, dass viele WAM-Regeln im Pausenplanersystem nicht eingehalten wurden.

Die Verbots-Queries haben 24 Beziehungen zwischen Klassen entdeckt, die gegen WAM-Regeln verstoßen.

Die Funde der Gebots-Queries stellten sich bei genauerer Untersuchung nicht als echte Verletzungen von WAM-Regeln heraus. Gegen diese Art von WAM-Regeln wurde also nicht verstoßen.

Zehn echte Verletzungen von WAM-Regeln traten bei den Ergebnissen der Einzel-Element-Queries auf. Wie bei den Verstößen gegen Verbots-Beziehungsregeln, wird die Masse der Verstöße durch häufige Verletzung einiger weniger Regeln verursacht.

Eine Auswertung der Ergebnisse in Bezug auf die Nützlichkeit der Queries findet sich in Kapitel 7, nach den Analysen der weiteren Beispielsysteme.

5.4 Zusammenfassung

In diesem Kapitel wurde zunächst die Architektur des Pausenplanersystems beschrieben. Im Weiteren wurden die Ergebnisse der Anwendung der Queries auf das Pausenplanersystem tabellarisch dargestellt und erläutert. Es folgte eine abschließende Bewertung der Ergebnisse, die deutlich machte, dass die WAM-Modellarchitektur im Pausenplaner mehrfach verletzt wird.

Kapitel 6 Untersuchung weiterer Systeme

Das im vorherigen Kapitel untersuchte Pausenplanersystem wurde von Studenten entwickelt, die nur wenig Erfahrung mit der WAM-Modellarchitektur aufzuweisen hatten. Das System enthält aus diesem Grund viele Verstöße gegen die WAM-Modellarchitektur.

Das EMS und das Gepardsystem, deren Untersuchung in diesem Kapitel beschrieben wird, wurden von Entwicklern implementiert, die bereits einige Erfahrung mit der WAM-Modellarchitektur haben. Es wurden dementsprechend weniger Verstöße gegen die Regeln der WAM-Modellarchitektur gefunden.

Die Architekturen des EMS und des Gepardsystems werden in diesem Kapitel nicht vorgestellt. Um die wenigen Verstöße gegen die Regeln der WAM-Modellarchitektur zu verstehen, die in diesen Systemen gefunden wurden, ist es nicht nötig, die Architektur der Systeme zu kennen.

6.1 Ergebnisse der Analyse des EMS

In diesem Abschnitt werden die Ergebnisse der Untersuchung des EMS vorgestellt. Wie auch bei der Analyse des Pausenplanersystems wurden zunächst einige allgemeine Systemdaten ermittelt, die Auskunft über die Größe des Systems geben.

Im EMS wurden keine Zyklen gefunden. Im Folgenden gibt es daher nach der Darstellung der Ergebnisse der verschiedenen Queries keinen Abschnitt über Zyklen. Im letzten Abschnitt folgt eine Bewertung der Ergebnisse.

Wie für die Analyse des Pausenplanersystems wurden unbenutzte Artefakte und Testklassen aus dem EMS entfernt.

6.1.1 Allgemeine Systemdaten

Metrik	Wert
SysLOC	31931
SysRelevantLines	9735
SysClasses	273
SysPackages	34
SysAnonymousClasses	108
SysMethods	1628

Tabelle 6.1: Allgemeine Daten des EMS

Das EMS ist mit 31931 Codezeilen größer als das Pausenplanersystem. Die Anzahl der relevanten Codezeilen ist jedoch etwas kleiner als im Pausenplanersystem. Sie beträgt ungefähr ein drittel aller Codezeilen und nicht wie üblich ungefähr die Hälfte. Das liegt daran, dass das System als Beispiel für [Zül04] erstellt und sehr gründlich kommentiert wurde. Metriken für die Anzahl der Kommentarzeilen und JavaDoc-Kommentare, deren Ergebnisse hier nicht aufgeführt sind, belegen dies.

Das EMS besteht aus 273 Klassen, von denen 108 anonyme Klassen sind. Das EMS hat also 165 nicht-anonyme Klassen, 64 mehr als das Pausenplanersystem. Die Klassen des EMS sind in 34 Packages aufgeteilt und haben insgesamt 1628 Methoden.

Das sind 251 Methoden mehr als im Pausenplanersystem. Dass das System weniger relevante Codezeilen, dafür aber mehr Methoden, Klassen und Packages als das Pausenplanersystem hat, könnte darauf hindeuten, dass das EMS besser strukturiert ist.

6.1.2 WAM-Elemente

Tabelle 6.2 stellt zusammen, welche Ergebnisse die Queries aus der Gruppe WAM-Elements lieferten.

In EMS wurde JWAM 2 als Rahmenwerk benutzt. In JWAM 2 gibt es keine Klasse mehr, von der GUIs erben müssen. Die GUI-Query findet GUIs darüber, dass ihre Klassennamen den String „GUI“ enthalten. Abgesehen von dieser Besonderheit werden die restlichen Elemente so gefunden, wie es bei der Darstellung der Analyseergebnisse des Pausenplanersystems in Kapitel 5.2.2 beschrieben wurde.

Elementart	Elemente im EMS	Falsch Negative	Falsch Positive
Materialien	17	0	2
Funktionskomponenten	5	0	0
Interaktionskomponenten	5	0	0
Tool-Klassen	5	0	0
GUIs	17	0	1
Monotool-Klassen	24	0	0
Services	16	0	0
Automaten	1	0	0
Fachwerte	14	0	0
Tools (Ip, Fp, GUI, Tool-class, Monotool-class)	56	0	0
Tools (Complex, grouped)	5	0	0
Tools (Mono, grouped)	9	10	0

Tabelle 6.2: WAM-Elemente im EMS

Im EMS wurden 17 Materialien gefunden. Weiterhin gibt es 29 Werkzeuge, die sich aufteilen in 24 monolithische Werkzeuge und fünf komplexe Werkzeuge. Das System enthält 16 Serviceklassen, einen Automat und 14 Fachwerte. Unter ihnen befinden sich auch abstrakte Klassen, Interfaces und Testklassen, die versehentlich nicht entfernt wurden. Auf Letztere und die Fehlinterpretationen wird im Detail im weiteren Verlauf dieses Abschnittes eingegangen.

Zwei der gefundenen Klassen der Material-Query sind keine Materialien. Die Klasse `MaterialboxTool` liegt im Package „materialbox“. Der Name lässt schon erkennen, dass sie kein Material ist. Weiterhin wurde im Package „remotematerialbox“ die Klasse `RemoteMaterialBoxTool` gefunden. Auch sie ist kein Material. Diese beiden Klassen sorgen später für fehlerhafte Ergebnisse in anderen Material-Queries. Da klar ist, dass diese beiden Klassen keine Materialien sind, werden sie bei den Ergebnissen der Queries, die die WAM-Regeln überprüfen, nicht berücksichtigt.

Eine der 17 Klassen, die die GUI-Query gefunden hat, ist keine GUI. Der `DnD-GuiFinder` hat ein „GUI“ im Namen. Der Name der Klasse lässt aber vermuten, dass sie keine GUI ist. Ein Blick in die Klasse bestätigt diese Vermutung. Die Funde, die diese Klasse bei anderen Queries, die sich auf GUIs beziehen verursacht, werden in

den folgenden Abschnitten nicht berücksichtigt, da sicher ist, dass es sich dabei nicht um Regelverstöße handelt.

Das Ergebnis der Query „Tools (Ip, Fp, GUI, Tool-class, Monotool-class)“ sind alle Elemente, die zu Werkzeugen gehören. Die Query hat 56 Werkzeugelemente gefunden. Diese Summe ergibt sich auch aus der Summe der gefundenen Funktionskomponenten, Interaktionskomponenten, GUIs, Tool-Klassen und Monotool-Klassen.

Unter den gefundenen Monotool-Klassen finden sich das `JemmyDummyTool` und das `DummyTool`. Dem Quelltext der Klassen ist zu entnehmen, dass sie nur für Tests benutzt werden. Testklassen wurden zwar vor der Untersuchung gelöscht, dabei wurden jedoch nur Klassen berücksichtigt, die ein „Test“ im Namen haben. Ansonsten hätte bei jeder Klasse einzeln geprüft werden müssen, ob es sich um eine Testklasse handelt. Daher wurden diese beiden Klassen, die nur für Tests benutzt werden, nicht gelöscht.

Die komplexen Werkzeuge wurden mit all ihren Elementen erkannt, die Namensgebung hält sich in diesem System stark an die gängigen Konventionen für WAM-Systeme.

Von den 24 monolithischen Werkzeugen wurden nur neun Monotool-Klassen mit ihrer GUI gefunden. Zu den anderen Monotool-Klassen wurde entweder gar keine oder keine GUI mit dem gleichen Namen gefunden. Die Gründe hierfür finden sich bei der Erläuterung zu den Ergebnissen der Query „Monotool-classes Should Know Their GUI“ in Kapitel 6.1.4.2. Diese Query listet die Monotool-Klassen auf, zu denen keine GUI des gleichen Namens gefunden wurde.

6.1.3 Vererbungsfehler

Tabelle 6.3 zeigt die Ergebnisse der Queries aus der Gruppe WAM-Inheritance. Queries, die keine Klassen gefunden haben werden nicht aufgeführt.

Da es für GUIs in JWAM 2 keine Superklasse und kein Interface gibt, wird für GUIs nicht nach Vererbungsfehlern gesucht. Wie schon in Kapitel 5.2.3 erläutert, suchen die Queries dieser Gruppe nach Klassen, die bestimmte Strings im Namen haben, aber nicht von den richtigen JWAM-Komponenten erben. Ein „IP“ oder „IAK“ im Klassennamen ist z.B. die Bezeichnung für eine Interaktionskomponente (englisch: „interaction part“). Diese Klasse sollte das Interface `ToolInteraction` implementieren.

Elementart	Anzahl der Funde	Echte Vererbungsfehler
Fachwerte	8	0
Interaktionskomponenten	2	0
Materialien	11	0
Services	4	0
Tool-Klassen	2	0
Insgesamt	27	0

Tabelle 6.3: Vererbungsfehler im EMS

Die Fachwerte-Query hat acht Klassen gefunden. Die Klasse `DateDVFillIn` wird an der Oberfläche zur Eingabe von Fachwerten genutzt. Sie ist aber kein Fachwert. Die

restlichen Klassen tragen ein „Factory" im Namen, sind also Fachwert-Fabriken. Es handelt sich daher nicht um echte Vererbungsfehler.

Die Funde der Interaktionskomponenten-Query sind die Klassen `DnDDescriptionAndSource` und `DnDDescriptionAndSourceTransferable`. Die Stellen des Strings „ip" im Namen zeigen bereits, dass es sich hier nicht um Interaktionskomponenten handelt.

Bei den Funden der Materialien-Query handelt es sich bei genauerem Hinsehen nicht um Materialien:

Sechs der elf Funde sind Klassen, die zum `MaterialBox-Werkzeug` und zum `RemoteMaterialBox-Werkzeug` gehören. Unter den verbliebenen fünf Funden gibt es zwei innere private Klassen der Materialien `Room` und `RoomMap`. Der Klassenkommentar sagt aus, dass es „utility"-Klassen sind. Es sind also keine Materialien. Weiterhin gibt es eine Klasse `ExampleMaterials`, die nur für Tests benutzt wird. Die Klasse `FrameAccessible` ist ein Interface, das für Werkzeuge benutzt wird. Es sollte in ein Werkzeug-Package oder ins „Util"-Package verschoben werden, da es keinerlei Beziehungen zu Materialien hat.

Die letzte Klasse in den Ergebnissen der Materialien-Query ist die Klasse `Cloner`, mit deren Hilfe Kopien von Materialien erstellt werden können. Sie ist kein Material, kann aber auch nicht in ein anderes Package verschoben werden, da die Klasse auf Materialien zugreift und einige Materialien auf den `Cloner`, so dass sich ein Packagezyklus ergeben würde. Das Problem könnte gelöst werden, indem die Materialien keine Methode mehr anbieten, um eine Kopie von sich zu erzeugen. Automaten, Services oder Werkzeuge, die eine Kopie eines Materials benötigen, könnten sich direkt an den `Cloner` wenden, um eine Kopie zu erhalten. Dann könnte der `Cloner` z.B. in ein Util-Package unterhalb der Werkzeugpackages verschoben werden.

Die vier Klassen, die die Service-Query gefunden hat, sind `RegistryServiceException`, „`RegistryServiceListener`, `RegistryServiceListenerImpl` und `StartupServer`. Die Namen und die Implementationen legen nahe, dass diese Klassen keine Services sind.

Die Tool-Klassen-Query findet die zwei Request-Klassen `OpenToolForMaterialRequest` und `OpenToolRequest`, die dem Namen nach keine Tool-Klassen sind. Es handelt sich also nicht um Vererbungsfehler.

6.1.4 WAM-Regelverletzungen

Im Folgenden werden die Verletzungen der WAM-Regeln, die im EMS gefunden wurden, dargestellt und erläutert. Es werden zunächst die Funde für Verbots-Queries aufgeführt. Im Anschluß folgen die Funde für Gebots- und Einzel-Element-Queries. Queries die keine Regelverstöße gefunden haben, werden nicht aufgelistet.

6.1.4.1 Verstöße gegen Verbots-Beziehungsregeln

Tabelle 6.4 zeigt alle verbotenen Beziehungen an, die zwischen Klassen im EMS gefunden wurden.

Query	Gesamt-anzahl Funde	Anzahl Klassen-beziehungen	Echte Verletzungen (Klassen-beziehungen)
FPs Shouldn't Know Tool-classes	5	1	0
GUIs Shouldn't Know Tools	2	1	0
IPs Shouldn't Know Materials	24	2	2
Monotool-classes Shouldn't Know Tool-classes	17	3	0
Tool-classes Shouldn't Know Automatons	2	2	0
Insgesamt	50	9	2

Tabelle 6.4: Verstöße gegen Verbots-Beziehungsregeln im EMS

Die Query „FPs Shouldn't Know Tool-classes" hat eine Funktionskomponente gefunden, die eine Tool-Klasse kennt. Dieser Fund ist eine zulässige Ausnahme, da die Tool-Klasse zu einem Subwerkzeug des Werkzeugs der FK gehört.

Die Funde der Query „GUIs Shouldn't Know Tools" sind keine Regelverletzungen. Die gefundene Beziehung besteht zwischen zwei GUI-Klassen. Es handelt sich um eine erlaubte Vererbungsbeziehung. Wäre die Beziehung eine Benutzt-Beziehung, wäre es ein Fehler, da eine GUI keine GUI eines anderen Werkzeuges kennen sollte. Vererbungsbeziehungen können aber nicht aus dem Ergebnis herausgefiltert werden, da dann z.B. eine Vererbungsbeziehung zwischen einer GUI und einer IAK nicht gefunden würde. Es ist zwar unwahrscheinlich, dass solch ein Fehler auftritt, er sollte aber entdeckt werden können.

Die Interaktionskomponente `DeviceCheckerResultViewerIP` kennt die Materialien `DeviceCheckerResult` und deren innere Klasse `Entry`. Dieses wurde durch die Query „IPs Shouldn't Know Materials" festgestellt. Diese Beziehungen stellen einen Regelverstoß dar und sollten aus dem System entfernt werden.

Die Query „Monotool-classes Shouldn't Know Tool-classes" hat zwei Monotool-Klassen gefunden, die eine bzw. zwei Tool-Klassen kennen. Da es sich um Subwerkzeuge der Monotool-Klassen handelt, liegt kein Regelverstoß vor.

Den Ergebnissen der Query „Tool-classes Shouldn't Know Automatons" zu Folge gibt es zwei Tool-Klassen, die einen Automaten kennen. Die einzige Beziehung besteht jeweils in der Rückgabe der Automatenklasse in der Methode `getAspect()` der Tool-Klassen. Da mit dieser Methode erfragt werden kann, worauf ein Werkzeug arbeitet, heißt das, dass die Werkzeuge auf dem Automaten arbeiten. Es handelt sich also um eine Ausnahme, die keinen Regelverstoß darstellt.

6.1.4.2 Verstöße gegen Gebots-Beziehungsregeln

Es gibt Beziehungen, die in einem WAM-System bestehen müssen. Die Ergebnisse der Queries, die diese Relationen überprüfen, sind in Tabelle 6.5 dargestellt.

Query	Gesamtanzahl Funde	Echte Verletzungen
FPs Should Know Materials	2	0
Monotool-classes Should Know Materials	5	0
Monotool-classes Should Know Their GUI	14	0
Services Should Know Materials	8	0
The Monotool-class Should Initialize The GUI	15	0
Insgesamt	44	0

Tabelle 6.5: Verstöße gegen Gebots-Beziehungsregeln im EMS

Die beiden Funktionskomponenten, die die Query „FPs Should Know Materials“ gefunden hat, kennen keine Materialien. Es scheint sich hier um Ausnahmen zu handeln. Mit den Werkzeugen können Namen von Geräten erstellt werden und gültige Namensbereiche für Geräte festgelegt werden. Die Werkzeuge arbeiten dafür auf Fachwerten.

Die Query „Monotool-classes Should Know Materials“ hat fünf Monotool-Klassen gefunden, die keine Materialien kennen. Die Funde sind jedoch alle Ausnahmen:

Das `DummyTool` und das `JemmyDummyTool` werden nur für Tests benutzt. Zwei der restlichen Werkzeuge sind abstrakt, die Implementierungen dieser Werkzeuge scheinen Materialien zu kennen, sonst wären sie in den Ergebnissen aufgeführt. Das letzte der gefundenen Werkzeuge ist das `DeviceCheckerCenterTool`. Mit diesem kann u.a. ein Automat eingestellt werden. Es handelt sich anscheinend um eine Ausnahme.

Die Query „Monotool-classes Should Know Their GUI“ listet 14 Monotool-Klassen auf. Zu dieser Query gibt es eine FollowUp-Query, die angibt, ob die aufgeführten Monotool-Klassen GUIs eines anderen Namens kennen. Aus den Ergebnissen der FollowUp-Query ergibt sich, das keiner der Funde eine echte Verletzung ist. Fünf der 14 aufgeführten Monotool-Klassen kennen eine GUI, die einen anderen Namen als die Monotool-Klasse besitzt. Unter diesen fünf GUIs taucht viermal die `TabbedMMI-Gui` auf. Sie scheint eine generische GUI zu sein, die von mehreren Werkzeugen verwendet wird. Unter den 14 gefundenen Monotool-Klassen gibt es das `DummyTool` und das `JemmyDummyTool`, die nur für Tests benutzt werden und dafür anscheinend keine GUI benötigen. Übrig bleiben sieben Monotool-Klassen, die keine GUI kennen. Dieses sind wahrscheinlich Werkzeuge, die aus der GUI ihres Kontext-Werkzeugs heraus gestartet werden. Ohne eine genauere Untersuchung des Quelltextes kann diese Vermutung jedoch nicht bestätigt werden. Es ist ungewöhnlich, dass es so viele Werkzeuge ohne zugehörige GUI gibt, da auch Subwerkzeuge meist eine eigene GUI besitzen.

Bei den acht aufgelisteten Services, die die Query „Services Should Know Materials“ gefunden hat, handelt es sich nicht um Klassen, die gegen die überprüfte WAM-Regel verstoßen. Vier Klassen gehören zum `ValuePoolService`. Darunter sind zwei Interfaces und zwei Klassen. Dieser Service verwaltet eine Fachwertmenge und kennt deshalb keine Materialien. Eine weitere der acht gefundenen Klassen heißt `Mock-CardServiceImpl` und wird dem Namen nach nur für Tests benutzt. Die restlichen drei Klassen sind Interfaces, die nicht unbedingt Materialien kennen müssen. Es reicht, wenn die Services, die diese Interfaces implementieren, Materialien kennen. Da diese Services nicht im Ergebnis dieser Query aufgeführt sind, tun sie das.

Die Query „The Monotool-class Should Initialize The GUI" hat 15 Monotool-Klassen gefunden, die keine GUI des gleichen Namens erzeugen. Es handelt sich aber in keinem der Fälle um echte Verletzungen. Zwei der gefundenen Klassen sind Klassen, die für Tests genutzt werden. Sie wurden beim Löschen der Testklassen übersehen, da sie kein „Test" im Namen haben. Unter den Ergebnissen befinden sich zwei abstrakte Klassen, die beide eine GUI eines anderen Namens, nämlich eine `DragDropGui` erzeugen. Die restlichen Monotool-Klassen, die von dieser Query gefunden wurden, erben von einer der beiden abstrakten Klassen. Da die `DragDropGui` in der jeweiligen abstrakten Oberklasse erzeugt wird, brauchen die Unterklassen keine eigene GUI zu erzeugen. Es handelt sich hier also um Ausnahmen der Regel.

6.1.4.3 Verstöße gegen Einzel-Element-Regeln

Regeln, die für einzelne WAM-Elemente gelten müssen, werden von den Queries der Gruppe WAM-Single-Element-Rules überprüft. Queries dieser Gruppe, die potentielle Regelverstöße gefunden haben, sind in Tabelle 6.6 aufgeführt.

Die Query „Materials Shouldn't Have „get"- And „set"-Methods Only" sucht nach Materialien, die außer einem Konstruktor nur Methoden besitzen, deren Namen mit „get" oder „set" beginnen.

Query	Gesamtanzahl Funde	Anzahl gefundener Klassen	Echte Verletzungen
FPs Shouldn't Return Materials	1	1	1
Materials Shouldn't Have "get"- And "set"-Methods Only	4	4	4
Insgesamt	5	5	5

Tabelle 6.6: Verstöße gegen Einzel-Element-Regeln im EMS

Die Query „FPs Shouldn't Return Materials" hat eine FK gefunden. Es handelt sich um einen Fehler, der mit den Funden der Query „IPs Shouldn't Know Materials" zusammenhängt, da IAKs einen Zugriff auf ein Material nur über ihre FK bekommen können, wenn das System alle anderen WAM-Regeln einhält. Auch im Pausenplanersystem bekamen einige IAKs Zugriff auf Materialien über ihre FK.

Bei den vier gefundenen Materialien der Query „Materials Shouldn't Have „get"- And „set"-Methods Only", handelt es sich bei den Klassen `Copyable` und `Registerable` um Interfaces und bei `Card` um eine abstrakte Klasse. Sie haben jeweils nur eine oder zwei Methoden. Wenn diese Klassen als technische Interfaces gesehen werden, sollten sie entweder nicht das Interface `Thing` implementieren oder nicht im Package „material" liegen. Der letzte Fund dieser Query ist die Klasse `Entry`, die eine innere Klasse von `DeviceCheckerResult` ist. Sie besitzt acht Methoden, die ausschließlich „get"- und „set"-Methoden sind. Diese Klasse scheint wie die Interfaces ein technisches Konstrukt zu sein, da sie keinerlei fachliche Umgangsformen besitzt.

6.1.5 Bewertung der Ergebnisse

Die Queries, die nach Vererbungsfehlern suchen haben im EMS 27 Klassen gefunden. Keiner der Funde stellte einen echten Vererbungsfehler dar. Es gibt jedoch einige Klassen im EMS, die in andere Packages verschoben werden sollten.

Insgesamt haben die Verbots-Queries neun Klassenbeziehungen gefunden, von denen nur zwei Beziehungen Regelverletzungen darstellen. Diese wenigen Regelverstöße sind nicht verwunderlich, da das System als Beispiel für ein WAM-System entwickelt wurde und demnach auch alle Regeln einhalten sollte. Es ist eher erstaunlich, dass trotz der Erfahrung der Entwickler und eines Code-Reviews nach Abschluss des Projektes zwei Fehler vorhanden sind.

Die Gebots-Queries haben insgesamt 29 Klassen gefunden. Alle Funde sind Ausnahmen.

Die fünf Funde der Einzel-Element-Queries sind alles echte Verletzungen.

Die Fragen, wie nützlich die Queries zur Analyse des Systems waren und welche Verbesserungsmöglichkeiten es gibt, werden in Kapitel 7 erörtert.

6.2 Ergebnisse der Analyse des Gepardsystems

In den folgenden Abschnitten werden die Ergebnisse der Analyse des Gepardsystems dargestellt. Nach einigen Systemdaten wird auf die Ergebnisse der Queries und der Zyklenanalyse eingegangen. Es folgt eine abschließende Bewertung der Ergebnisse.

Wie für die Analyse des Pausenplanersystems und des EMS wurden vor Beginn der Analyse unbenutzte Artefakte und Testklassen aus dem System entfernt.

6.2.1 Allgemeine Systemdaten

Metrik	Wert
SysLOC	113418
SysRelevantLines	41773
SysClasses	567
SysPackages	19
SysAnonymousClasses	158
SysMethods	4498

Tabelle 6.7: Allgemeine Daten des Gepardsystems

Mit 113418 Codezeilen ist das Gepardsystem deutlich größer als das Pausenplanersystem oder das EMS. Die Anzahl der relevanten Codezeilen beträgt weniger als die Hälfte aller Codezeilen Dies liegt daran, dass das System ebenso wie das EMS eine große Anzahl von JavaDoc-Kommentaren enthält.

Das System besteht aus 567 Klassen, davon 158 anonyme Klassen, die auf 19 Packages verteilt sind. Die Klassen enthalten insgesamt 4498 Methoden. Das System besitzt also ungefähr doppelte so viele Klassen und mehr als doppelt so viele Methoden wie das EMS, hat jedoch 15 Packages weniger.

6.2.2 WAM-Elemente

Alle WAM-Elemente, die im System gefunden wurden, werden in Tabelle 6.8 aufgeführt.

Das Gepardsystem wurde mit einer älteren JWAM-Version erstellt. Es tritt hier das gleiche Problem auf, wie bei JWAM 2, dass die GUIs nicht über Vererbungsbeziehungen gefunden werden können. Die GUIs dieser Auflistung wurden daher mit den Queries für JWAM 2 gefunden.

Eine weitere Besonderheit dieses Systems ist, dass Automaten und Services nicht von den entsprechenden JWAM-Klassen erben. Die Queries wurden daher so umgestellt, dass Services und Automaten über den String „Service“ und „Automaton“ im Namen gefunden wurden. Aufgrund der konsequenten Namensgebung wurden hiermit alle Services und Automaten erfasst. Warum die Automaten und Services nicht von den entsprechenden JWAM-Komponenten erben, ist ungeklärt.

Zur Zeit der Erstellung des Systems war es üblich, die Art der Werkzeugelemente dem Werkzeugnamen voranzustellen. Die Werkzeuge heißen also z.B. „ipWerkzeug1“ und nicht „werkzeug1IP“, wie in neueren Projekten. Aus diesem Grund finden die Queries „Tools (Complex, grouped)“ und „Tools (Mono, grouped)“ keine zusammengehörenden Werkzeugteile.

Die Query „Tools (Mono, grouped)“ würde jedoch auch bei „korrekter“ Namensgebung keine Funde liefern. Der Grund dafür ist, dass die Query entweder auf JWAM 1.8 oder auf JWAM 2 ausgelegt ist. Die Query für JWAM 1.8 sucht Monotool-Klassen mit einer Vererbungsbeziehung aus JWAM 1.8 mit einer GUI, die ebenfalls aus JWAM 1.8 erbt. Die Query für JWAM 2 dagegen sucht Monotool-Klassen, die aus JWAM 2 erben und die passende GUI wird über ihren Klassennamen gesucht. Im Gepardsystem können die Monotool-Klassen jedoch nur über die Vererbungsbeziehung wie bei JWAM 1.8 gefunden werden, die GUIs nur wie bei JWAM 2 über den Namen. D.h. die Query für JWAM 1.8 würde die GUIs nicht finden und die Query für JWAM 2 würde die Monotool-Klassen nicht finden. Die Queries „Tools (Complex, grouped)“ und „Tools (Mono, grouped)“ wurden daher in der Tabelle weggelassen. Eine Neuimplementierung der Queries wäre möglich gewesen, schien jedoch für eine so alte JWAM-Version nicht lohnenswert.

Elementart	**Elemente im Gepardsystem**	**Falsch Negative**	**Falsch Positive**
Materialien	41	0	0
Funktionskomponenten	3	0	0
Interaktionskomponenten	3	0	0
Tool-Klassen	3	0	0
GUIs	16	0	0
Monotool-Klassen	17	0	0
Services	56	0	0
Automaten	21	0	0
Fachwerte	30	0	0
Tools (Ip, Fp, GUI, Tool-class, Monotool-class)	42	0	0

Tabelle 6.8: WAM-Elemente im Gepardsystem

Es gibt im Gepardsystem drei komplexe Werkzeuge und 17 monolithische Werkzeuge, die auf 41 Materialien arbeiten. Neben 56 Serviceklassen und 21 Automaten enthält das System 30 Fachwerte.

Alle WAM-Elemente, die zu Werkzeugen gehören, werden von der Query „Tools (Ip, Fp, GUI, Tool-class, Monotool-class)" aufgelistet. Im Gepardsystem gibt es 42 Werkzeugelemente. Diese Zahl stimmt überein mit der Summe aller Funktionskomponenten, Interaktionskomponenten, GUIs, Tool-Klassen und Monotool-Klassen.

6.2.3 Vererbungsfehler

Tabelle 6.9 listet alle potentiellen Vererbungsfehler auf, die die Queries aus der Grupe WAM-Inheritance gefunden haben.

Da keiner der Automaten und Services von den entsprechenden JWAM-Komponenten erbt, konnten die Vererbungsqueries für sie keine neuen Erkenntnisse bringen und wurden wie die Query für GUIs ausgelassen.

Elementart	**Anzahl der Funde**	**Echte Vererbungsfehler**
Fachwerte	3	1
Interaktionskomponenten	1	0
Materialien	34	0
Insgesamt	38	1

Tabelle 6.9: Vererbungsfehler im Gepardsystem

Zwei Klassen, die die Fachwert-Query gefunden hat, haben ein „Mapper" im Namen und werden anscheinend für die Datenbankanbindung benutzt. Der Name der dritten Klasse klingt wie ein Fachwert. Der Name der Klasse und der Fakt, dass sich die Klasse im Package „domainvalue" befindet, lassen vermuten, dass es sich um einen echten Fehler handelt und die Klasse das Interface `DomainValue` implementieren sollte.

Das Ergebnis der Interaktionskomponenten-Query ist die Klasse `dvModelDescription`. Sie hat ein „ip" im Namen und wird daher fälschlicherweise als Interaktionskomponente interpretiert, die nicht das Interface `ToolInteraction` implementiert.

Es werden 34 Klassen im Materialpackage gefunden, die nicht das Interface „Thing" implementieren. 31 dieser Klassen haben ein „Mapper" im Namen und werden für die Datenbankanbindung benutzt. Eine der verbliebenen drei Klassen ist eine anonyme innere Klasse und wird daher kein Material sein. Eine weitere Klasse hat den Vermerk „Helper class" im Kommentar, was vermuten lässt, dass auch sie kein Material sein soll. Dem Namen nach klingt die letzte Klasse nach einer technischen Klasse. Es handelt sich also bei den 34 Klassen nicht um Vererbungsfehler. Es sollte aber überlegt werden, sie in ein anderes Package zu verschieben, da es verwirrend ist, wenn in einem Package, das „material" heißt, Klassen liegen, die keine Materialien sind.

6.2.4 WAM-Regelverletzungen

In den nächsten Abschnitten werden die Ergebnisse der Queries dargestellt und erläutert, die Verletzungen der WAM-Regeln im Gepardsystem gefunden haben. Es werden zunächst die Ergebnisse der Verbots-Queries aufgeführt. Weiterhin folgen die Funde der Gebots- und Einzel-Element-Queries.

6.2.4.1 Verstöße gegen Verbots-Beziehungsregeln

Es wurden einige verbotene Beziehungen im System gefunden, die in Tabelle 6.10 aufgelistet sind.

Query	Gesamt-anzahl Funde	Anzahl Klassen-beziehungen	Echte Verletzungen (Klassenbeziehun-gen)
DVs Shouldn't Know Materials	13	13	13
Materials Shouldn't Know Services	29	1	1
Monotool-classes Shouldn't Know Tool-classes	11	2	0
Tool-classes Shouldn't Know Services	3	2	2
Insgesamt	56	18	16

Tabelle 6.10: Verstöße gegen Verbots-Beziehungsregeln im Gepardsystem

Die Query „DVs Shouldn't Know Materials" zeigt, dass eine Fachwertklasse 13 Materialien kennt. Es wird nur der Klassenname des Materials benutzt. Trotzdem sollte diese Beziehung aufgelöst und das Problem auf andere Art gelöst werden.

Durch die Query „Materials Shouldn't Know Services" wird ein Material gefunden, das einen Service kennt. Dies ist mit hoher Wahrscheinlichkeit ein echter Regelverstoß.

Die Monotool-Klasse `DummyTool`, die die Query „Monotool-classes Shouldn't Know Tool-classes" entdeckt hat, kennt zwei Tool-Klassen. Sie wird vermutlich nur für Tests benutzt.

Es gibt eine Tool-Klasse, die zwei verschiedene Services kennt. Sie wurde durch die Query „Tool-classes Shouldn't Know Services" aufgedeckt. Die Beziehungen scheinen echte Regelverletzungen zu sein.

6.2.4.2 Verstöße gegen Gebots-Beziehungsregeln

Tabelle 6.11 zeigt die Ergebnisse der Gebots-Queries.

Query	Gesamtanzahl Funde	Echte Verletzungen
Automatons Should Know Materials	7	3
IPs Should Know Their FP	3	0
IPs Should Know Their GUI	3	3
Monotool-classes Should Know Materials	6	6
Monotool-classes Should Know Their GUI	-	-
Services Should Know Materials	29	16
The Monotool-class Should Initialize The GUI	-	-
The Tool-class Should Initialize The FP	3	0
The Tool-class Should Initialize The GUI	3	2
The Tool-class Should Initialize The IP	3	0
Insgesamt	57	30

Tabelle 6.11: Verstöße gegen Gebots-Beziehungsregeln im Gepardsystem

Sieben Automaten kennen keine Materialien und wurden von der Query „Automatons Should Know Materials“ gefunden. Zwei Klassen sind Interfaces und zwei Klassen scheinen für Tests zu sein. Für die restlichen drei Automaten lassen sich ohne eine genauere Analyse des Systems keine Gründe finden, warum sie keine Materialien kennen sollten. Es scheint sich also um echte Verletzungen der Regel *„Ein Automat arbeitet wie ein Werkzeug auf Materialien“* zu handeln.

Es gibt drei IAKs, die von der Query „IPs Should Know Their FP“ gefunden wurden. Da zur Zeit dieses Projektes die Konvention galt, dass die Elementbezeichner am Anfang des Klassennamens stehen, konnte zu den drei IAKs keine FKs des gleichen Namens gefunden werden. Die FollowUp-Query ergibt jedoch, dass alle IAKs eine FK des gleichen Namens kennen, die wie die IAKs den Elementbezeichner am Anfang des Klassennamens haben.

Die Query „IPs Should Know Their GUI“ hat drei Interaktionskomponenten gefunden, die keine GUIs des gleichen Namens kennen. Die Ausführung der FollowUp-Query ergibt, dass sie überhaupt keine GUIs kennen. Diese Interaktionskomponenten sollten genauer untersucht werden.

Abgesehen von einer abstrakten Klasse sind die Monotool-Klassen, die von der Query „Monotool-classes Should Know Materials“ gefunden werden, voll implementierte Klassen. Die Namen lassen keinerlei Rückschlüsse zu, warum sie keine Materialien kennen.

Die Queries „Monotool-classes Should Know Their GUI“ und „The Monotool-class Should Initialize The GUI“ können zu diesem System keine Ergebnisse liefern, da weder die Query für JWAM 1.8 noch die Query für JWAM 2 sowohl Monotool-Klassen als auch GUIs dieser JWAM-Version findet, sondern immer nur eines der beiden Elemente. Dies wurde bereits in Kapitel 6.2.2 erläutert. Es müsste eine neue Query geschrieben werden. Dies scheint für eine so alte JWAM-Version jedoch nicht lohnenswert.

Es gibt 29 Services, die keine Materialien kennen, wie die Query „Services Should Know Materials“ zeigt. Elf der Services sind abstrakte Klassen oder Interfaces. Einer der Services scheint nur Zugang zu anderen Services zu gewähren und es gibt einen `DummyService`, der dem Namen nach für Tests benutzt wird. Was die restlichen 16 Services im Einzelnen tun, müsste anhand des Quelltextes analysiert oder von Entwicklern erfragt werden, um festzustellen, ob es sich um Ausnahmen handelt oder ob die Klassen evtl. doch keine Services sein sollen, auch wenn sie ein „Service“ im Namen tragen.

Die Queries „The Tool-class Should Initialize The FP” und “The Tool-class Should Initialize The IP” finden jeweils die gleichen drei Tool-Klassen. Beim Ausführen der FollowUp-Queries zeigt sich, dass die Tool-Klassen jeweils FK und IAK erzeugen. Sie wurden aufgelistet, da in diesem System die Elementart im Klassennamen am Anfang steht und die Tool-Klasse ihren FKs und IAKs daher nicht zugeordnet werden konnte. Es handelt sich in allen sechs Fällen nicht um echte Verletzungen.

Die Query „The Tool-class Should Initialize The GUI“ hat drei Tool-Klassen aufgezeigt, die keine GUI des gleichen Namens erzeugen. Die FollowUp-Query zeigt, dass eine der Klassen eine GUI erzeugt, die nur dem Namen nach nicht zugeordnet wer-

den konnte. Warum die anderen Tool-Klassen keine GUIs erzeugen, sollte weiter untersucht werden.

6.2.4.3 Verstöße gegen Einzel-Element-Regeln

Die Verletzungen von Einzel-Element-Regeln werden von den Queries der Gruppe WAM-Single-Element-Rules gefunden. Die Ergebnisse dieser Queries sind in der folgenden Tabelle dargestellt.

Query	Gesamtanzahl Funde	Anzahl gefundener Klassen	Echte Verletzungen
DVs Shouldn't Have A Public Constructor	1	1	1
FPs Shouldn't Return Materials	1	1	1
Materials Shouldn't Have "get"- And "set"-Methods Only	2	2	2
Insgesamt	4	4	4

Tabelle 6.12: Verstöße gegen Einzel-Element-Regeln im Gepardsystem

Die Query „DVs Shouldn't Have A Public Constructor" zeigt, dass eine Fachwertklasse einen öffentlichen Konstruktor hat. Da in dem Kommentar über dem Konstruktor „protected constructors" steht, war dieser Fehler anscheinend ein Versehen. Sofern dieser Konstruktor nicht von anderen Klassen benutzt wird, ist der Fehler nicht besonders schlimm, da er leicht zu beheben ist.

Es gibt eine Funktionskomponente, die die Query „FPs Shouldn't Return Materials" gefunden hat, die an ihrer Schnittstelle ein Material zurückgibt. Die Methode wird nur in der Funktionskomponente selber benutzt, es handelt sich aber trotzdem um einen Regelverstoß.

Die Ergebnisse der Query „Materials Shouldn't Have "get"- And "set"-Methods Only" sind zwei abstrakte Klassen, die jeweils nur eine bzw. keine Methode haben. Die Frage, ob diese abstrakten Klassen, die keinerlei fachlich motivierte Methoden haben, Materialien sein sollen, können nur die Entwickler oder eine genauere Quelltextanalyse beantworten.

6.2.5 Zyklen

Die folgende Tabelle zeigt eine Übersicht der Klassen- und Packagezyklen, die im Gepardsystem gefunden wurden.

Zyklusart	Anzahl Zyklen	Anzahl Klassen bzw. Packages im Zyklus
Klassenzyklen	12	8 Zweierzyklen/ 19/ 5/ 3/ 8
Packagezyklen	2	3/ 6

Tabelle 6.13: Zyklen im Gepardsystem

Vergleicht man das Gepardsystem mit anderen Systemen, so ist die Anzahl der Klassenzyklen nicht überdurchschnittlich. Im Vergleich zu anderen Systemen gibt es vor allem nur wenige große Zyklen. Die Packagezyklen scheinen im Verhältnis zu anderen Systemen eher klein zu sein.

Der große Klassenzyklus, der aus 19 Klassen besteht, wird dadurch verursacht, dass ein Fachwert auf Materialien zugreift. Dieses ist ein Verstoß gegen eine WAM-Regel, die auch mit den Queries gefunden wurde. Die Materialien greifen ihrerseits auf die Fachwertklasse zu. Würden die Beziehungen von dem Fachwert zu den Materialien entfernt, löst sich der Zyklus auf.

Diese Maßnahme wäre ebenfalls nützlich, um die Packagezyklen aufzulösen, da sie den Packagezyklus der aus sechs Klassen besteht, um zwei Packages verkleinern würde. Wie man schon im Pausenplanersystem sehen konnte, kann die Entstehung einiger Zyklen verhindert werden, wenn die Regeln der WAM-Modellarchitektur befolgt werden.

Die elf kleineren Klassenzyklen haben keinen Verstoß gegen eine WAM-Regel als Ursache. Die meisten von ihnen sind Zweierzyklen, d.h. eine gegenseitige Beziehung zwischen zwei Klassen, die wahrscheinlich mit geringem Aufwand aufgelöst werden kann.

Wie schon im oberen Abschnitt erwähnt, wird der Packagezyklus, der aus sechs Packages besteht, um zwei Packages kleiner, wenn die Beziehung zwischen Fachwert und Materialien entfernt wird. Eine weitere Ursache für diesen Zyklus ist der Zugriff von einem Material auf einen Service. Durch Behebung dieses Regelverstoßes wird der Zyklus zwar nicht verkleinert, eine Ursache jedoch entfernt.

Der kleine Packagezyklus wird nicht durch einen Regelverstoß verursacht und müsste ebenso wie der Zyklus aus sechs Packages und die kleineren Klassenzyklen durch eine genauere Analyse aufgelöst werden.

6.2.6 Bewertung der Ergebnisse

Alle Automaten und Services in diesem System haben nicht von den entsprechenden JWAM-Komponenten geerbt. Aufgrund der streng eingehaltenen Namenskonventionen konnten die Queries jedoch so umgestellt werden, dass auch Verstöße gegen Regeln für diese Elemente gefunden werden konnten.

Die Verbots-Queries konnten 18 Beziehungen zwischen Klassen finden, von denen 16 echte Regelverstöße waren. Im Verhältnis zur Größe des Systems sind dies deutlich weniger Verstöße als im Pausenplanersystem. Dies liegt wahrscheinlich daran, dass das Pausenplanersystem von Studenten entwickelt wurde, die nur wenig Erfahrung mit der WAM-Modellarchitektur hatten.

Deutlich mehr Verstöße als im Pausenplanersystem wurden für die Regeln, die Beziehungen vorschreiben, gefunden. Hier sollte eine genauere Analyse des Quelltextes erfolgen.

Die Einzel-Element-Queries haben im Gepardsystem viel weniger Fehler gefunden als im Pausenplanersystem. Die zugehörigen Regeln wurden im Gepardsystem fast immer konsequent eingehalten.

Neben Erkenntnissen im Hinblick auf die Qualität des Gepardsystems, sollte die Anwendung der Queries auch zeigen, wie nützlich die Queries bei der Überprüfung der

WAM-Modellarchitektur sind. Dieser Aspekt der Untersuchung der Beispielsysteme wird in Kapitel 7 diskutiert.

6.3 Zusammenfassung

In diesem Kapitel wurden zwei WAM-Systeme untersucht, die nicht von Studenten, sondern von erfahrenen WAM-Entwicklern erstellt wurden. Es hat sich gezeigt, dass es weniger Verstöße gegen die Regeln der WAM-Modellarchitektur gibt als im Pausenplanersystem. Es wurde jedoch auch deutlich, dass in Systemen, die von erfahrenen Entwicklern erstellt wurden, Fehler vorhanden sind, die erst mit den Queries entdeckt werden.

Kapitel 7 Auswertung

In diesem Kapitel werden die Ergebnisse dieser Arbeit reflektiert und bewertet. Zunächst wird auf die Klassifizierung und Überprüfbarkeit der Regeln und auf die erstellte Regelsammlung eingegangen. Weiterhin werden die erstellten Queries bewertet und Verbesserungsmöglichkeiten aufgezeigt. Es folgen Überlegungen, wie der Sotograph erweitert werden könnte, um eine Architekturüberprüfung, die sich auf Elemente und ihre Relationen bezieht, besser zu unterstützen.

7.1 Nutzen der Regelsammlung, Klassifizierung und Überprüfbarkeit der Regeln

Basierend auf [Zül98] und [Zül04] wurde in dieser Arbeit eine Regelsammlung für die WAM-Modellarchitektur erstellt. Die Regelsammlung enthält Regeln für die wichtigsten und am häufigsten genutzten Elemente der WAM-Modellarchitektur. Es wurde damit eine Übersicht über die WAM-Regeln erstellt, die es in dieser Form noch nicht gab. Sie kann Entwicklern Unterstützung bei der Implementierung eines Systems nach der WAM-Modellarchitektur geben, indem Regeln schnell nachgeschlagen werden können.

Die Klassifizierung der Regeln war ein wichtiger Punkt, um die Sammlung so übersichtlich wie möglich zu gestalten. Aus mehreren Möglichkeiten wurde die Lösung gewählt, die Regeln zunächst nach der Elementart zu unterteilen, auf die sie sich beziehen und weiterhin nach Verbots-Beziehungsregeln, Gebots-Beziehungsregeln und Einzel-Element-Regeln. Diese Klassifizierung ist für den Gebrauch der Regelsammlung sinnvoll und stützt sich auf die Definition für Softwarearchitekturen.

In der Regelsammlung ist zu jeder Regel angegeben, ob Verletzungen gegen sie automatisch gefunden werden können oder nicht. In Kapitel 3.7 wurde herausgearbeitet, welche Regeln automatisch kontrolliert werden können, bei welchen dies nicht möglich ist oder warum es manchmal nicht sinnvoll ist, nach Verletzungen von Regeln zu suchen. Es stellte sich heraus, dass ungefähr die Hälfte der Regeln aus der Regelsammlung mit dem Sotographen automatisch überprüfbar sind.

7.2 Bewertung der Query zur Identifizierung der Elemente

Für die Identifizierung der Elemente im Quelltext konnte in Kapitel 4 eine praktikable Lösung gefunden werden. Elemente wurden für diese Arbeit über ihre Vererbungsbeziehungen zu JWAM oder über Namenskonventionen erkannt. Bei den Elementen, die nur über ihre Vererbungsbeziehungen gesucht wurden, traten bei der Analyse der Beispielsysteme keine falsch negativen und keine falsch positiven Ergebnisse auf. Diese Art der Identifizierung ist also sehr zuverlässig.

Bei der Suche nach Elementen, die nicht oder nicht nur über Vererbungsbeziehungen gefunden wurden, traten sowohl falsch negative als auch falsch positive Ergebnisse auf, wie Tabelle 7.1 zeigt. Die Ergebnisse der Queries, die Elemente über Namenskonventionen identifizieren, sind in der Tabelle für die verschiedenen Systeme addiert. Die Kürzel hinter dem Querynamen stehen für die Systeme aus denen die Ergebnisse stammen. Die Funde der Materialien-Query[5] sind für alle drei Systeme addiert. GUIs

[5] Genaugenommen suchen die angegebenen Queries für Materialien, GUIs, Services und Automaten nicht selber nach Elementen, sondern zeigen die Inhalte der Tabellen an, die von der Query „init tab-

wurden nur im EMS und Gepardsystem über ihren Namen gefunden. Services und Automaten mussten nur im Gepardsystem über ihren Namen identifiziert werden. Im Gepardsystem konnten die Tools-Queries aufgrund anderer Namenskonventionen nicht ausgeführt werden, hier stammen die Ergebnisse nur aus der Untersuchung des Pausenplanersystems und des EMS.

Query	Funde	Falsch Negative	Falsch Positive
Materialien (P, E, G)	64	0	2
GUIs (E, G)	33	0	1
Services (G)	56	0	0
Automaten (G)	21	0	0
Tools (Complex, grouped) (P, E)	9	2	0
Tools (Mono, grouped) (P, E)	12	11	0

Tabelle 7.1: Identifizierung der WAM-Elemente; P – Pausenplanersystem, E – EMS, G – Gepardsystem

Im Verhältnis zu der Anzahl der gefundenen Elemente sind die falsch positiven Ergebnisse der Queries für einzelne Elemente zu vernachlässigen. Dies zeigt, dass auch in Systemen, in denen kein Rahmenwerk benutzt wird, Überprüfungen durchgeführt werden können, sofern die Namenskonventionen eingehalten werden. Die Tools-Queries weisen dagegen einige falsch Negative auf. Dies liegt daran, dass hier nicht einzelne Elemente identifiziert, sondern mehrere Elemente einander aufgrund des Namens zugeordnet werden sollten. Wurden von den Namenskonventionen nur leicht abgewichen, erzielten die Queries falsche Ergebnisse. Da diese Ergebnisse nicht als Grundlage für andere Queries dienen, haben die falsch Negativen hier aber zumindest keine weiteren Auswirkungen. Um die falsch Negativen zu vermeiden, müsste eine Lösung gefunden werden, um zugehörige Werkzeugelemente zu kennzeichnen. Dies könnte beim Einlesen des Systems in den Sotographen geschehen (s. Abschnitt 7.6).

So gut die Lösung auch funktioniert, WAM-Elemente über Vererbungsbeziehungen und Namenskonventionen zu identifizieren, so weist sie doch Nachteile auf. Die Erkennung über Namenskonventionen ist nicht fehlerfrei. Guis können in JWAM 2 zwar relativ gut über ihre Namen gefunden werden, die Identifizierung von Materialien ist jedoch äußerst unbefriedigend. Sie werden darüber identifiziert, dass sie das Interface `Thing` implementieren und in einem Package liegen, dessen Namen den String „material" enthält. In den Beispielsystemen gab es keine Probleme, bei anderen Systemen kann dies aber ganz anders sein. Ein weiterer Nachteil ist, dass das Rahmenwerk JWAM für die Erkennung über Vererbung als library in den Sotographen eingelesen werden muss. Außerdem können so nur Systeme überprüft werden, die JWAM verwenden. Ändern sich Vererbungsbeziehungen durch eine neue JWAM-Version, müssen die Queries angepasst werden.

Eine Lösung, die diese Nachteile nicht hätte, wäre Markerinterfaces in JWAM einzuführen. Dann könnten auch Guis und Materialien eindeutig identifiziert werden. Markerinterfaces könnten jedoch auch in WAM-Systemen eingesetzt werden, die weder JWAM noch Namenskonventionen verwenden. Durch diesen geringen Mehraufwand könnten auch hier die WAM-Regeln überprüft werden. Es sollte dazu eine Kon-

les" erstellt wurden. Die Tools-Queries nutzen die erstellten Tabellen für die Werkzeugelemente, stellen über Namenskonventionen aber selbst die Zuordnung zwischen den Werkzeugelementen her.

vention geben, wie die Markerinterfaces heißen, damit die Queries nicht für jedes System angepasst werden müssen.

7.3 Bewertung der Queries der Gruppen WAM-Elements und WAM-Inheritance

Die Queries der Gruppe WAM-Elements liefern einen guten Überblick über die WAM-Komponenten des Systems. Die Queries können Entwicklern den Einstieg in ein bereits laufendes Projekt erleichtern, indem die Entwickler eine Übersicht über die Elemente des Systems bekommen.

Die Queries der Gruppe WAM-Inheritance haben nur selten echte Vererbungsfehler aufgezeigt. Dies konnte meist schon durch den Namen der gefundenen Klassen oder einem kurzen Blick in den Quelltext geklärt werden. Auch wenn die Queries selten Fehler aufgezeigt haben, sind sie nicht überflüssig. Durch sie bekommen Entwickler einen Anstoß, sich über die Funktion von Klassen und ihre Position in der Packagestruktur Gedanken zu machen. Es wurden z.B. viele Klassen in den Materialpackages der drei Systeme gefunden, die keine Materialien sind. Sie sollten in andere Packages verschoben werden.

7.4 Bewertung der Queries, die Regelverstöße suchen

In Kapitel 4 wurde erläutert, wie der Sotograph genutzt werden kann, um die WAM-Regeln zu überprüfen. Es wurden Queries erstellt, die nach Verstößen gegen WAM-Regeln suchen. Im Folgenden wird ausgewertet, wie gut diese Queries Verletzungen der Regeln gefunden haben.

	Pausenplanersystem	EMS	Gepardsystem	Gesamt	
VBR, Echte Verletzungen	24	2	16	42	77,8%
VBR, Falsch Positive	3	7	2	12	22,2%
VBR, Funde	27	9	18	54	100%
GBR, Echte Verletzungen	0	0	30	30	26,1%
GBR, Falsch Positive	14	44	27	85	73,9%
GBR, Funde	14	44	57	115	100%
ER, Echte Verletzungen	10	5	4	19	90,5%
ER, Falsch Positive	2	0	0	2	9,5%
ER, Funde	12	5	4	21	100%

Tabelle 7.2: Übersicht der Queryergebnisse; VBR: Verbots-Beziehungsregeln, GBR: Gebots-Beziehungsregeln, ER: Einzel-Element-Regeln

Tabelle 7.2 zeigt eine Übersicht der Ergebnisse der WAM-Queries, die die Einhaltung von Regeln überprüfen. Die Funde der Queries schlüsseln sich auf in echte Verletzungen und falsch Positive, d.h. Funde, die keine echten Verletzungen darstellen.

Tabelle 7.2 zeigt, dass die Einzel-Element-Queries am zuverlässigsten sind. Allerdings sind die wenigsten, nämlich nur fünf der 40 implementierten WAM-Queries aus dieser Kategorie. 24 Queries sind Verbots-Queries. Sie sind mit 77,8% echten Verletzungen relativ zuverlässig. Am unzuverlässigsten sind die elf Gebots-Queries. Im Weiteren wird auf die Ursachen für die falsch positiven Ergebnisse der verschiedenen Queryarten detailliert eingegangen.

Die zwölf falsch positiven Ergebnisse der Verbots-Queries sind in Tabelle 7.3 aufgeschlüsselt. Ein Großteil (58,3%) der falsch Positiven sind Ausnahmen der jeweils überprüften Regel. Die restlichen fünf Funde, die keine echten Verletzungen sind, setzen sich zusammen aus zwei Funden für eine nicht gelöschte Testklasse und drei Funden, die aus einer Schwäche der Query „GUIs Shouldn't Know Tools" resultieren. Auf die Schwäche dieser Query wird in Abschnitt 7.5 eingegangen.

7	58,3%	Ausnahmen
3	25%	Queryschwächen
2	16,7%	Testklassen
12	100%	

Tabelle 7.3: Falsch Positive der Verbots-Beziehungsregeln

Tabelle 7.4 zeigt die Aufschlüsselung der Ergebnisse der Gebots-Queries. 42,4% der falsch Positiven sind Ausnahmen von Regeln. Die restlichen falsch Positiven setzen sich zusammen aus 29,4% falsch Positive, die daraus resultieren, dass Namenskonventionen nicht eingehalten wurden, 18,8% falsch Positive, die auf Schwächen der Queries zurückzuführen sind und 9,4% falsch Positive, die von nicht gelöschten Testklassen verursacht wurden. Auf die Schwächen der Queries, die 18,8% der falsch Positiven verursachen, wird in Abschnitt 7.5 eingegangen.

Bei vielen der Gebots-Queries musste bei der Implementierung auf Namenskonventionen zurückgegriffen werden. Beispielsweise kann die GUI eines Werkzeugs der IAK desselben Werkzeugs nur über den Namen zugeordnet werden. Wenn diese Namenskonventionen nicht eingehalten werden, steigt die Anzahl der falsch positiven Ergebnisse.

36	42,4%	Ausnahmen
25	29,4%	Namenskonventionen
16	18,8%	Queryschwächen
8	9,4%	Testklassen
85	100%	

Tabelle 7.4: Falsch Positive der Gebots-Beziehungsregeln

Die Ergebnisse der Einzel-Element-Queries sind in Tabelle 7.5 aufgeschlüsselt. Die falsch Positiven entstehen bei diesen Queries nur aus der Verletzung von Namenskonventionen in den Systemen.

2	100%	Namenskonventionen
2	100%	

Tabelle 7.5: Falsch Positive der Einzel-Element-Regeln

Es wurde deutlich, dass die meisten falsch Positiven auf Ausnahmen der Regeln zurückzuführen sind. Solche falsch positiven Ergebnisse sind nicht zu vermeiden, da

nur durch eine fachliche Interpretation klar wird, ob falsch Positive Ausnahmen sind oder nicht. Eine Einhaltung der Namenskonventionen für WAM-Systeme könnte die Anzahl der falsch positiven Ergebnisse deutlich verringern, sie machen ebenfalls einen großen Teil der falsch Positiven aus. Es gibt jedoch auch falsch Positive, die durch Schwächen der Queries verursacht werden. Hier kann eine Überarbeitung der Queries helfen. Ein geringer Teil der falsch Positiven wurde durch Testklassen verursacht, die nicht aus dem System entfernt wurden. Abhilfe hierfür würde geschaffen, wenn Testklassen in separate Packages gelegt würden, um ihre Entfernung vor der Analyse zu erleichtern.

7.5 Schwächen und Verbesserungsmöglichkeiten einzelner Queries

Die Queries, die in dieser Arbeit für die Untersuchung der drei Systeme verwendet wurden, haben mehrere Überarbeitungszyklen hinter sich. Es haben sich durch die endgültige Auswertung aber noch weitere Verbesserungsmöglichkeiten ergeben, die im Folgenden beschrieben werden.

Die Query „GUIs Shouldn't Know Tools“ ist die einzige Verbots-Query, die falsch positive Ergebnisse erbracht hat. GUIs sollten keine anderen Werkzeugelemente kennen. Zu anderen GUIs sind Vererbungsbeziehungen erlaubt, jedoch keine Benutzt-Beziehungen. Es wurde in der Query nach Beziehungen von GUIs zu allen Werkzeugelementen gesucht. Dabei wurden im Ergebnis dann auch Vererbungsbeziehungen zwischen GUIs gefunden, die erlaubt sind. Alle Vererbungsbeziehungen können aus dem Ergebnis nicht herausgefiltert werden, da dann z.B. eine Vererbungsbeziehung zwischen einer GUI und einer IAK, auch wenn sie sehr unwahrscheinlich ist, nicht erkannt würde. Es ist aber technisch zu aufwändig in dieser Query nur die Vererbungsbeziehungen zu GUIs zu ignorieren. Die Query sollte daher in zwei Queries aufgeteilt werden, so dass eine Query nach Beziehungen aller Art zu anderen Werkzeugelementen sucht und die zweite Query nur nach Benutzt-Beziehungen zu anderen GUIs.

Die falsch positiven Ergebnisse, die durch Schwächen von Gebots-Queries, verursacht wurden, sind alle auf den gleichen Ursprung zurückzuführen. Die Queries „X Should Know Y“ unterscheiden nicht zwischen voll implementierten Klassen, abstrakten Klassen und Interfaces. Es werden in den Ergebnissen oft Interfaces aufgeführt, die das Element nicht kennen, es aber auch nicht müssen. Mindestens eine Klasse oder ein Interface in der Klassenhierarchie müssen das Element kennen. Es kann entweder die implementierende Klasse sein oder ein Ober-Typ. Meistens ist es die Klasse selber. Also reicht es, nur sie zu überprüfen, da der Aufwand für die gesamte Prüfung ineffizient und die Query aufwändig zu erstellen wäre.

Bei der Untersuchung der verschiedenen Systeme hat sich gezeigt, dass die Query „DV Inheritance” aus der Gruppe WAM-Inheritance verbessert werden könnte. Sie sucht nach Fachwerten, die den String „dv“ im Namen haben und nicht das Interface `DomainValue` implementieren. Klassen, die ein „dv“ aber auch ein „Factory“ oder „Fabrik“ im Namen haben, sollten aus dem Ergebnis herausgefiltert werden, da Fachwertfabriken das Interface `DomainValue` nicht implementieren müssen. Stattdessen sollte eine Query „DV-Factory Inheritance“ implementiert werden, die prüft, ob Klassen, die „dv“ und „factory“ im Namen haben das Interface `Factory` implementieren.

7.6 Ansätze zur Erweiterung des Sotographen

Im Laufe dieser Arbeit wurde deutlich, dass der Sotograph eine Architekturprüfung, die sich auf Elemente und ihre Relationen bezieht, noch nicht optimal unterstützt.

Mit den Möglichkeiten des ModelManager können, wie in Kapitel 4.3 beschrieben, einige Verbots-Beziehungsregeln der WAM-Modellarchitektur, aber keine Gebots-Beziehungsregeln abgebildet werden. Hier würden zwei Erweiterungen helfen:

- Zum Einen sollte es möglich sein, Klassen Subsystemen zuzuordnen Im Moment ist es nur möglich, Packages Subsystemen zuzuordnen. Dadurch könnte man Element-Subsysteme bilden, die jeweils eine Art von Elementen enthalten. Da im ModelManager relationale Ausdrücke verwendet werden können, könnten bei den meisten Elementarten die Klassen über ihre Namen einem Subsystem zugeordnet werden. WAM-Elemente, die keine eindeutige Namenskonvention enthalten, müssten einzeln zugeordnet werden. So könnten alle Verbots-Beziehungsregeln mit einem Grapharchitekturmodell dargestellt und mit dem Sotographen überprüft werden.
- Die zweite Erweiterung müsste an den Grapharchitekturmodellen vorgenommen werden. Zur Zeit können hier nur verbotene und erlaubte Beziehungen spezifiziert werden. Um auch einen Teil der Gebots-Beziehungsregeln überprüfen zu können, müssten auch vorgeschriebene Beziehungen spezifiziert werden können. Damit könnten Gebots-Beziehungsregeln kontrolliert werden, die sich nur auf Elementarten beziehen. Z.B. könnte die Regel, dass eine FK Materialien kennen sollte, dann überprüft werden. Weiterhin nicht überprüft werden könnten Gebots-Beziehungsregeln, die sich auf zwei bestimmte Elemente beziehen. Ein Beispiel dafür wäre, dass eine IAK die zum selben Werkzeug gehörende GUI kennen sollte. Einzel-Element-Regeln könnten ebenfalls nicht mit dem ModelManager überprüft werden. Der Sotograph kann Beziehungen in einem System kontrollieren, für Einzel-Element-Regeln gibt es hier noch kein Konzept. Wahrscheinlich sind hier Queries am Besten geeignet, da Einzel-Element-Regeln sehr unterschiedlich sind.

Eine Erweiterung des Sotographen daraufhin, dass der Benutzer Architekturelemente vor dem Einlesen des Systems identifiziert, wurde in Abschnitt 4.1.1 diskutiert. Die eindeutige Zuweisung der Klassen eines Systems zu Elementarten würde den Vorteil bringen, dass die Erkennung der Elemente in den Queries wegfallen würde. Eine Hürde auf dem Weg zur Regelüberprüfung würde damit im Vorwege entfernt. Eine solche Erweiterung des Sotographen wäre nur sinnvoll, wenn die Elemente ohne großen Aufwand durch den Benutzer zugewiesen werden könnten, beispielsweise, indem Klassen oder ganze Packages in einer Baumstruktur markiert werden könnten, oder indem wie im Modelmanager relationale Ausdrücke genutzt werden können. Des Weiteren muss die Elementzuweisung beim Einlesen einer neuen Version automatisch übernommen werden können. Verbots-Beziehungsregeln für diese Elemente könnten sicherlich einfach im ModelManager erstellt werden. Die Diskussion, wie der Benutzer Gebots-Beziehungsregeln und Einzel-Element-Regeln für diese Elemente komfortabel spezifizieren könnte, würde in dieser Arbeit jedoch zu weit führen. Denn außer der Möglichkeit, Verbots-Beziehungsregeln auf Subsystemebene zu spezifizieren und Queries zu definieren gibt es für selbst erstellte Regeln kein Konzept im Sotographen.

Auch wenn die zuvor beschriebenen Erweiterungen des ModelManager und des Sotographen insgesamt den Vorteil bieten, dass alle WAM-Elemente eindeutig zugeordnet würden, so entsteht dennoch ein großer Nachteil: Die WAM-Elemente müssen für jedes WAM-System von Hand zugeordnet werden. Mit der Lösung, die in dieser Arbeit gewählt wurde, funktioniert die Zuordnung der Klassen zu Elementen automatisch, wenn auch nicht ganz fehlerfrei. Sie muss auch nicht verändert werden, wenn sich das System weiterentwickelt. Bei der Erweiterungslösung müssten neu hinzugekommen Elemente auch neu zugeordnet werden.

Eine Lösung für dieses Problem könnte eine Art „Tagging" in den Javadoc-Kommentaren im Quelltext sein, durch das Klassen als Architekturelemente markiert werden. Im Sotographen müssten dann nur die Zuordnungen von einem Tag zu einem Architekturelement spezifiziert sein, damit er die Elemente automatisch erkennen könnte.

Kapitel 8 Fazit und Ausblick

8.1 Fazit

Die Hauptfragestellung dieser Arbeit war, ob sich die WAM-Modellarchitektur mit dem Sotographen automatisch überprüfen lässt. Die Untersuchung der Beispielsysteme mit den in dieser Arbeit erstellten Queries hat gezeigt, dass diese Frage positiv beantwortet werden kann.

Die Erstellung einer Regelsammlung war nötig, um eine Grundlage zu bilden, auf der überprüft werden konnte, ob gegen die Regeln der WAM-Modellarchitektur verstoßen wird. Die Regelsammlung kann darüber hinaus auch als eine Art „Handbuch“ für WAM-Entwickler genutzt werden. Regeln können bei Bedarf schnell gefunden und nachgelesen werden, um ein WAM-System fehlerfrei zu entwickeln.

Mit den Möglichkeiten des Sotographen kann ein Teil der WAM-Modellarchitektur überprüft werden. Aber erst die Erstellung von Queries, die WAM-Regeln überprüfen, lassen die Kontrolle einer größeren Zahl der WAM-Regeln zu.

Die Anwendung der WAM-Queries, die Regelverletzungen finden, hat gezeigt, dass mit ihnen Verstöße gegen die WAM-Modellarchitektur aufgedeckt werden können, die vorher nicht entdeckt wurden. Selbst im EMS, das als Beispiel für die WAM-Modellarchitektur entwickelt wurde, einem gründlichen Code-Review unterzogen wurde und mit den vorhandenen Möglichkeiten des Sotographen untersucht worden ist, konnten einige Fehler erst mit den WAM-Queries gefunden werden.

Die Regelverstöße im EMS und im Gepardsystem belegen, dass es auch für Systeme, die von erfahrenen Entwicklern erstellt wurden, sinnvoll ist nach Verletzungen der WAM-Modellarchitektur zu suchen. Auch wenn die Regeln der WAM-Modellarchitektur bekannt sind, können Verstöße vorkommen, die ohne eine automatisierte Prüfung nicht entdeckt werden.

Je ein Zyklus im Pausenplanersystem und im Gepardsystem wurde, zumindest teilweise, durch einen Regelverstoß verursacht. Die Entstehung einiger Zyklen kann also verhindert werden, wenn die WAM-Regeln befolgt werden.

Um in den Ergebnissen falsch Positive von echten Verletzungen zu unterscheiden, müssen die Queryergebnisse interpretiert werden. Oft können falsch Positive von echten Verletzungen bereits durch den Namen einer Klasse oder einen kurzen Blick in den Quelltext unterschieden werden. Manchmal ist jedoch auch eine genauere Analyse des Quelltextes oder die Befragung eines Entwicklers nötig.

Die analysierten Systeme benutzen verschiedene JWAM-Versionen. Die JWAM-Versionen 1.8 und 2 sind die Versionen, die zurzeit am häufigsten in Projekten vorzufinden sind. Sie werden im Pausenplanersystem bzw. im EMS verwendet. Die JWAM-Versionen sind zwar unterschiedlich, einige Queries konnten jedoch für beide Versionen verwendet werden. Die Anpassung der anderen Queries war zwar aufwändig, aber nicht schwierig, da das Prinzip, wie Regelverletzungen gefunden werden, von der JWAM-Version unabhängig ist. Das Gepardsystem verwendet eine noch ältere Version von JWAM. Für die meisten Regeln konnte über die vorhandenen Queries für JWAM 1.8 oder JWAM 2 überprüft werden, ob sie befolgt werden. Einige Queries hätten jedoch extra angepasst werden müssen. Dieser Aufwand schien für eine so alte

Version von JWAM nicht lohnenswert. In diesem System konnten daher nicht alle Regeln überprüft werden.

8.2 Ausblick

Im Laufe dieser Diplomarbeit haben sich verschiedene Punkte ergeben, deren weitergehende Untersuchung und Bearbeitung sinnvoll wäre.

Metriken

Queries müssen im Sotographen einzeln ausgeführt werden. Ob Queries Ergebnisse bringen, kann erst nach der Ausführung gesehen werden. Die Überprüfung der Regeln wäre deutlich angenehmer, wenn die Queries in so genannte „Metriken" umgewandelt werden. Metriken werden alle auf einmal berechnet und ihre Ergebnisse in Tabellen in der Datenbank gespeichert. Durch die Umwandlung der Queries in Metriken wäre es möglich, einen Überblick über alle Regelverletzungen anzuzeigen. Der Benutzer müsste nicht jede Query einzeln ausführen, um dann evtl. festzustellen, dass sie keine Verletzungen gefunden hat. Die Umwandlung der Queries in Metriken hat den weiteren Vorteil, dass damit eine Trendanalyse durchgeführt werden kann. D.h. für mehrere Versionen eines Systems kann verglichen werden, ob die Regelverstöße mehr oder weniger geworden sind.

Weitere Queries

Die Queries könnten in Zukunft auch für WAM-Elemente erweitert werden, die im Moment nicht in die Regelsammlung aufgenommen sind, wie z.B. Formulare oder die Fachwert-Fabriken. In der Regelsammlung dieser Arbeit wurden nur die wichtigsten WAM-Elemente berücksichtigt.

Die Queries, die überprüfen, ob von den richtigen Klassen in JWAM geerbt wird, bzw. dass die richtigen Interfaces implementiert werden, könnten noch erweitert werden. Es gibt eine Query für Materialien, die prüft, ob alle Klassen, die in einem Package liegen, das „material" im Namen hat auch `Thing` implementieren. So eine Query könnte auch für andere Elemente erstellt werden. In vielen Systemen liegen Serviceklassen in Packages, die „service" im Namen haben. Es könnte daher bei Systemen die JWAM 2 benutzen, geprüft werden, ob es Klassen gibt, die in einem Package mit „service" im Namen liegen und die nicht das Interface `Service` implementieren. Im Moment wird nur nach Klassen gesucht, die „Service" im Namen haben und nicht das Interface `Service` implementieren. Wie groß der Nutzen ist, den solche Queries bringen, müsste dann untersucht werden.

Es wäre möglich, alle Klassen aufzulisten, die von keiner Klasse aus JWAM erben. Dann hätte der Entwickler einen Überblick, wie viele „Nicht-WAM-Elemente" es in dem System gibt. Diese Klassen wären zum Großteil wahrscheinlich technische Hilfsklassen, es wäre jedoch auch möglich, dass es Architekturelemente gibt, für die es in WAM noch keine Metapher gibt. Diese könnten auf diese Weise gefunden werden.

Projektspezifische Regeln

Wenn ein Projekt während der Entwicklung kontinuierlich mit den WAM-Queries untersucht werden soll, kann es sinnvoll sein, zusätzlich zu den WAM-Regeln projektspezifische Regeln aufzustellen. Eine solche Regel könnte sein, dass Werkzeuge einer bestimmten Gruppe nur bestimmte Services nutzen dürfen. Verletzungen von solchen Regeln könnten durch Queries gefunden werden. Voraussetzung ist, dass die

Elemente, auf die sich die Regeln beziehen, identifiziert werden können. Das kann z.B. durch ein Interface sein, das von einer bestimmten Gruppe von Werkzeugen implementiert wird, oder ein bestimmter String, der in den Namen dieser Werkzeuge vorhanden ist. Werden die Queries zur Überprüfung solcher Regeln während der Entwicklung eines Projektes fortlaufend implementiert und gleich in Metriken umgewandelt, so entsteht während der Entwicklung des Projektes nur ein geringer Aufwand. Die Metriken können aber wesentlich dazu beitragen, dass die vorgesehene Architektur eingehalten wird.

Andere Modellarchitekturen

Die Übertragung des Ansatzes dieser Arbeit auf andere Modellarchitekturen ist prinzipiell möglich. Voraussetzung ist, dass die Identifizierung der Elemente durch Vererbungsbeziehungen oder Namenskonventionen möglich ist und dass Regeln existieren, die statisch geprüft werden können. Dann können auch für andere Modellarchitekturen Regeln durch Queries oder Metriken mit dem Sotographen überprüft werden.

Anhang A

Regeln der WAM-Modellarchitektur

A.1 Allgemeine WAM-Regeln

Regel: *Zyklische Strukturen sollen vermieden werden. Müssen Strukturen wechselseitig verknüpft werden, so sollte in einer Richtung eine lose Kopplung verwendet werden.*

Quelle: [Zül98], S. 344
Überprüfbarkeit: Ob die Regel eingehalten wird, ist mit den Metriken „ClassCyclicRefClass“ und „PckgCyclicRefPckg“ des Sotographen überprüfbar, die nach Zyklen zwischen Klassen bzw. Packages suchen. Diese Metriken gehören zur Grundausstattung des Sotographen und mussten nicht extra implementiert werden. Zyklen zwischen Gruppen von Packages, so genannten „Subsystemen“, können mit der Query „SubsysCyclicRefSubsys“ gefunden werden. Hierzu müssen jedoch zunächst die Subsysteme definiert werden. Es wurde noch nicht untersucht, ob es Subsystemmodelle gibt, die für alle WAM-Systeme gelten. Für jedes WAM-System müsste daher ein eigenes Subsystemmodell erstellt werden. Aus diesem Grund werden Subsystemzyklen bei der Analyse der Beispielsysteme nicht berücksichtigt.

A.2 Regeln für Materialien

A.2.1 Einzel-Element-Regeln

Regel: *Die innere Konsistenz eines Materials wird über das Vertragsmodell durch Zusicherung an den verändernden Operationen einer Klasse abgesichert.*

Quelle: [Zül98], S. 169
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht auf Methodenrümpfe zugegriffen werden kann, in denen sich die Zusicherungen befinden.

Regel: *Materialien bieten sondierende Operationen für Eigenschaften an, die für den Anwender von Interesse sind.*

Quelle: [Zül98], S. 169/170
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welches die Eigenschaften sind, die von Interesse sind.

Regel: *Materialien sollten nicht nur generische Operationen definieren.*

Quelle: [Zül98], S. 170
Anmerkungen: Materialien, die nur generische Operationen definieren, degenerieren dadurch zu einem Wertebehälter und die fachliche Funktionalität wandert in das Werkzeug.
Query: „Materials Shouldn't Have "get"- And "set"-Methods Only“

Regel: *Ist ein Material über einen Aspekt an ein Werkzeug gekoppelt, so muss es die im Aspekt formulierten Anforderungen erfüllen.*

Quelle: [Zül98], S. 206
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da es keine Möglichkeit gibt, Aspekte zu identifizieren. Wird der Aspekt als Interface realisiert, ist die Regel mit der Implementierung des Interfaces automatisch erfüllt.

A.2.2 Verbots-Beziehungsregeln

Regel: *Materialien dürfen keine Werkzeuge kennen.*
Quelle: WAM-Architekten
Anmerkungen: Der Begriff „Werkzeuge" umfasst sowohl komplexe als auch monolithische Werkzeuge. Materialklassen dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen.
Query: „Materials Shouldn't Know Tools"

Regel: *Materialien dürfen keine Services kennen.*
Quelle: WAM-Architekten
Query: „Materials Shouldn't Know Services"

Regel: *Materialien dürfen keine Automaten kennen.*
Quelle: WAM-Architekten
Query: „Materials Shouldn't Know Automatons"

Regel: *Materialien dürfen keine Events kennen.*
Quelle: WAM-Architekten
Überprüfbarkeit: Verstöße gegen diese Regel könnten nur gefunden werden, wenn das JDK als library in den Sotographen miteinbezogen würde. Der Aufwand für das Einlesen in den Sotographen würde sich dadurch deutlich erhöhen.

Regel: *Materialien dürfen keine Requests kennen.*
Quelle: WAM-Architekten
Überprüfbarkeit: Verstöße gegen diese Regel könnten nur gefunden werden, wenn das JDK als library in den Sotographen miteinbezogen würde. Der Aufwand für das Einlesen in den Sotographen würde sich dadurch deutlich erhöhen.

Regel: *Materialien dürfen keine anderen Elemente außer Materialien und Fachwerte benutzen.*
Quelle: WAM-Architekten
Anmerkungen: Die Regel gilt nur in Bezug auf WAM-Elemente. Basistypen und andere Utility-Klassen dürfen natürlich benutzt werden.
Überprüfbarkeit: Ob diese Regel eingehalten wird, wird dadurch geprüft, dass Verletzungen der Regeln „Materialien dürfen keine Werkzeuge/Services/Automaten kennen" gesucht werden.

A.2.3 Gebots-Beziehungsregeln

Regel: *Materialien können in andere Materialien eingebettet werden.*
Quelle: [Zül98], S. 168
Überprüfbarkeit: Eine Überprüfung ist nicht sinnvoll, da es keine strikte Regel ist.

Regel: *Nach Außen präsentierte Zustandinformationen sollten vorrangig Fachwerte sein.*
Quelle: [Zül98], S. 170
Überprüfbarkeit: Die Regel ist nur eine Empfehlung, deshalb ist es nicht sinnvoll sie zu prüfen.

Regel: *Materialien können Fachwerte als Bestandteile enthalten.*

Quelle: WAM-Architekten
Überprüfbarkeit: Eine Überprüfung ist nicht sinnvoll, da es keine strikte Regel ist.

A.3 Regeln für Werkzeuge (allgemein)

A.3.1 Einzel-Element-Regeln

Regel: *Ein Werkzeug sollte grundsätzlich in Funktionskomponente (FK) und Interaktionskomponente (IAK) aufgeteilt werden.*

Quelle: [Zül98], S. 236
Anmerkungen: Zu jeder FK sollte es eine IAK geben und zu jeder IAK eine FK. Es gibt aber auch Konstruktionsansätze, bei denen ein Subwerkzeug nur aus FK besteht und die Interaktion durch die Kontext-IAK implementiert wird. Diese Regel gilt nicht für monolithische Werkzeuge, bei denen die Funktionen von FK und IAK in einer Klasse vereinigt werden.
Query: „Complex Tools Consist Of GUI, FP, IP and Tool-class"

Regel: *Jedes eigenständige Werkzeug soll gegenüber dem Kontext durch eine Tool-Klasse abgeschlossen sein.*

Quelle:[Zül98], 276/289/290
Anmerkungen: D.h. Subwerkzeuge müssen nicht unbedingt eine Tool-Klasse besitzen. In der Praxis haben aber auch Subwerkzeuge immer eine Tool-Klasse, da Subwerkzeuge in JWAM nur über die Tool-Klasse gestartet werden können. Hier unterscheidet sich also die Implementierung von JWAM von der WAM-Modellarchitektur wie sie in [Zül98] beschrieben ist. Die Regel bezieht sich nur auf Werkzeuge, die aus FK und IAK zusammengesetzt sind.
Query: „Complex Tools Consist Of GUI, FP, IP and Tool-class"

A.3.2 Verbots-Beziehungsregeln

In dieser Kategorie sind keine Regeln vorhanden.

A.3.3 Gebots-Beziehungsregeln

Regel: *Werkzeuge sollten die Materialien, die sie bearbeiten unter einem Aspekt kennen.*

Quelle:[Zül98], S. 206
Anmerkungen: Bei kleinen Projekten oder Spezialwerkzeugen müssen nicht unbedingt Aspekte verwendet werden.
Überprüfbarkeit: Eine Überprüfung ist nicht sinnvoll, da es keine strikte Regel ist.

A.4 Regeln für Funktionskomponenten

Im Zusammenhang Kontext- und Subwerkzeug können Kontext-FKs auch Kontext-Monotool-Klassen und Sub-FKs auch Sub-Monotool-Klassen sein.

A.4.1 Einzel-Element-Regeln

Regel: *Die FK soll keinerlei Annahmen über ihre IAK machen. eines Werkzeugs soll verändert werden können, ohne dass die FK angepasst werden muss.*

Quelle: [Zül98], S. 235

Anmerkungen: Dadurch kann die IAK eines Werkzeugs verändert werden, ohne dass die FK angepasst werden muss.
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht auf Methodenrümpfe zugegriffen werden kann, in denen möglicherweise Annahmen über die IAK gemacht werden.

Regel: *Die Aufgaben der FK werden in einer Klasse gekapselt.*
Quelle: [Zül98], S. 239
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welches die Aufgaben der FK sind.

Regel: *Die FK bietet mittels sondierender Operationen Informationen über den eigenen Bearbeitungszustand und den Bearbeitungszustand des Materials an.*
Quelle: [Zül98], S. 240
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welche Attribute den Arbeitszustand repräsentieren.

Regel: *Die FK bietet über eine materialunbabhängige Schnittstelle Informationen zur Präsentation des Materials für die IAK an.*
Quelle: [Zül98], S. 240
Query: „FPs Shouldn't Return Materials"

Regel: *Für jede relevante Zustandsänderung der FK wird ein Event-Objekt erzeugt, das über die Schnittstelle angefordert werden kann.*
Quelle: [Zül98], S. 259
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welches die relevanten Zustandsänderungen sind.

A.4.2 Verbots-Beziehungsregeln

Regel: *Die FK soll möglichst keinerlei Kenntnis von ihrer IAK haben.*
Quelle: [Zül98], S. 236
Anmerkungen: Auch wenn in der Regel „möglichst" steht, so ist es in der Praxis immer der Fall, dass die FK ihre IAK nicht direkt kennt.
Query: „FPs Shouldn't Know IPs"

Regel: *FKs dürfen keine GUIs kennen.*
Quelle: WAM-Architekten
Query: „FPs Shouldn't Know GUIs"

Regel: *FKs dürfen keine Tool-Klassen kennen.*
Quelle: WAM-Architekten
Anmerkungen: Eine Ausnahme ist die Tool-Klasse eines Subwerkzeugs. In JWAM2 kennt eine FK ihre Toolklasse unter der Tool-Schnittstelle. Dies verstößt gegen die Regel. Auf Nachfrage bei dem für den„ToolConstruction"-Teil von JWAM zuständigen Architekten stellte sich diese Implementierung als Fehler heraus, der möglichst bald behoben werden sollte. Eine FK sollte ihre eigene Tool-Klasse höchstens als Request-Handler kennen.
Query: „FPs Shouldn't Know Tool-classes".

A.4.3 Gebots-Beziehungsregeln

Regel: *Die FK bearbeitet das Material.*

Quelle: [Zül98], S.236
Erläuterungen: Die Bearbeitung von Materialien ist der Hauptzweck von Werkzeugen. Daneben rufen sie evtl. noch Automaten und/oder Services.
Query: „FPs Should Know Materials“

Regel: *Die FK bearbeitet das Material nur unter der im Aspekt spezifizierten Schnittstelle.*

Quelle: [Zül98], S. 240
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da Aspekte nicht identifiziert werden können.

Regel: *Die FK benachrichtigt die IAK durch das Ereignismuster (oder Beobachtermuster) über Änderungen an der FK.*

Quelle: [Zül98], S. 251/283
Überprüfbarkeit: Die Einhaltung dieser Regel wäre nur dynamisch überprüfbar.

Regel: *Kontext-FKs kennen die volle Schnittstelle ihrer Sub-FKs.*

Quelle: [Zül98], S. 281
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht zwischen Kontext- und Sub-FKs unterschieden werden kann.

Regel: *Sub-FKs sind über das Ereignismuster (oder das Beobachtermuster) an ihre Kontext-FK gekoppelt.*

Quelle: [Zül98], S. 282
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht zwischen Kontext- und Sub-FKs unterschieden werden kann.

Regel: *Anforderungen werden über die Zuständigkeitskette verschickt. Eine FK bekommt ihre zugehörige Kontext-Fk, ihre Tool-Klasse oder die Umgebung als Requesthandler gesetzt.*

Quelle: [Zül98] S.292
Anmerkungen: Hier gibt es anscheinend eine Abweichung zwischen WAM und JWAM. In JWAM wird immer die Tool-Klasse oder Umgebung als RequestHandler gesetzt. Die Tool-Klasse leitet einen Request gegebenenfalls an eine Kontext-FK weiter. Die Umgebung ist eine WAM-Entwurfsmetapher. Sie stellt den Rahmen, in den alle anderen WAM-Elemente eingebettet sind. In JWAM können an ihr z.B. Werkzeuge registriert und über sie gestartet werden.
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur dynamisch feststellbar ist, welche Klasse als Requesthandler fungiert.

Regel: *Die Funktionskomponente gibt neben Standard-Datentypen nur Fachwerte an die Interaktionskomponente weiter.*

Quelle: [Zül98], S. 323
Überprüfbarkeit: Ob diese Regel befolgt wird, wird indirekt überprüft, indem geprüft wird, ob die Beziehungsregeln der IAK eingehalten werden und durch die Query „FPs Shouldn’t Return Materials“.

Regel: *Die FK startet die Subwerkzeuge dieses Werkzeugs.*

Quelle: WAM-Architekten
Überprüfbarkeit: Eine Überprüfung wäre nur dynamisch möglich.

A.5 Regeln für Interaktionskomponenten

Regeln beziehen sich nur auf Tool-Klassen, die nur dies sind und nicht auf Monotool-Klassen, die ja FK, IAK und Tool-Klasse vereinigen.

A.5.1 Einzel-Element-Regeln

Regel: *Die IAK benutzt im Regelfall Interaktionstypen um von den Widgets des verwendeten Fenstersystems abstrahieren zu können.*

Quelle: [Zül98], S. 236
Anmerkungen: Interaktionstypen (IATs) sind Elemente, die benutzt werden können, um die IAK unabhängig vom verwendeten Toolkit zu machen. In JWAM2 werden sie nicht mehr benutzt.
Überprüfbarkeit: Eine Überprüfung ist nicht sinnvoll, da es die Interaktionstypen in JWAM 2 nicht mehr gibt. Mit JWAM 1.8 wurden sie auch nur z.T. verwendet, da sie Schwächen in der Handhabung aufwiesen.

Regel: *Die Aufgaben der IAK werden in einer Klasse gekapselt.*

Quelle: [Zül98], S. 237
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welches die Aufgaben der IAK sind.

Regel: *Präsentationsbezogene Ereignisse setzt die IAK um.*

Quelle: [Zül98], S. 238
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welche Ereignisse präsentationsbezogen sind.

Regel: *IAKs machen keine Annahmen über die Auswirkungen einer Operation die sie an der FK rufen.*

Quelle: [Zül98], S. 239
Anmerkungen: IAKs veranlassen nach dem Aufrufen einer Operation an der FK nicht automatisch eine Änderung der Präsentation.
Überprüfbarkeit: Eine Überprüfung wäre nur dynamisch möglich.

A.5.2 Verbots-Beziehungsregeln

Regel: *Die IAK hat im Regelfall keinen direkten Zugriff auf das Material. In einigen Fällen (z.B. um komplexe tabellarische Materialien zu handhaben), darf die IAK Lesezugriff auf ein Material haben.*

Quelle: [Zül98], S. 240 und [Zül04], S. 226
Anmerkungen: Ursprünglich kommt die Regel aus [Zül98], die Ausnahme kommt aus [Zül04]
Query: „IPs Shouldn't Know Materials".

Regel: *IAKs dürfen keine Tool-Klasse kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Tool-classes"

Regel: *IAKs dürfen keine Monotool-Klasse kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Monotool-classes"

Regel: *IAKs dürfen keine Automaten kennen.*

Quelle: WAM-Architekten

Query: „IPs Shouldn't Know Automatons"

Regel: *IAKs dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: „IPs Shouldn't Know Services"

A.5.3 Gebots-Beziehungsregeln

Regel: *Die IAK nimmt Ereignisse von der Oberfläche entgegen.*

Quelle: [Zül98], S. 236
Anmerkungen: Mit „Oberfläche" ist hier die GUI gemeint.
Überprüfbarkeit: Eine Überprüfung wäre nur dynamisch möglich.

Regel: *Die IAK steuert die Präsentation an der Oberfläche.*

Quelle: [Zül98], S. 236
Anmerkungen: Mit „Oberfläche" ist hier die GUI gemeint. D.h. die IAK kennt die GUI unter ihrer vollen Schnittstelle und ruft Methoden an der GUI.
Query: „IPs Should Know Their GUI"

Regel: *Anwendungsbezogene Ereignisse leitet die IAK an die FK weiter.*

Quelle:[Zül98], S. 238
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welche Ereignisse anwendungsbezogen sind.

Regel: *Die IAK kennt ihre FK unter ihrer vollen Schnittstelle.*

Quelle: [Zül98], S. 246/247
Query: „IPs Should Know Their FP"

Regel: *Kontext-IAKs kennen die volle Schnittstelle ihrer Sub-IAKs, die Beziehung ist allerdings nicht immer vonnöten.*

Quelle: [Zül98], S. 281/285
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht zwischen Kontext- und Sub-IAKs unterschieden werden kann.

Regel: *Eine Sub-IAK ist über das Beobachtermuster an ihre Kontext-IAK gekoppelt, die Beziehung ist allerdings nicht immer vonnöten.*

Quelle: [Zül98], S. 285
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht zwischen Kontext- und Sub-IAKs unterschieden werden kann.

A.6 Regeln für Tool-Klassen

Diese Regeln beziehen sich nur auf Tool-Klassen und nicht auf Monotool-Klassen, die FK, IAK und Tool-Klasse vereinigen.

A.6.1 Einzel-Element-Regeln

In dieser Kategorie sind keine Regeln vorhanden.

A.6.2 Verbots-Beziehungsregeln

Regel: *Tool-Klassen dürfen keine Services kennen.*
Quelle: WAM-Architekten
Query: "Tool-classes Shouldn't Know Services"

Regel: *Tool-Klassen dürfen keine Automaten kennen.*
Quelle: WAM-Architekten
Query: "Tool-classes Shouldn't Know Automatons"

A.6.3 Gebots-Beziehungsregeln

Regel: *Die Tool-Klasse erzeugt die FK.*
Quelle: [Zül98], S. 290
Anmerkungen: Die Regel bezieht sich ursprünglich nur auf Kontext-FKs, in der Regel haben aber auch Subwerkzeuge eine Tool-Klasse, die die IAK und FK erzeugt.
Query: „The Tool-class Should Initialize The FP"

Regel: *Die Tool-Klasse erzeugt die IAK.*
Quelle: [Zül98], S. 290
Anmerkungen: Die Regel bezieht sich ursprünglich nur auf Kontext-IAKs, in der Regel haben aber auch Subwerkzeuge eine Tool-Klasse, die die IAK und FK erzeugt.
Query: „The Tool-class Should Initialize The IP"

Regel: *Die Tool-Klasse erzeugt die GUI.*
Quelle: WAM-Architekten
Query: „The Tool-class Should Initialize The GUI"

A.7 Regeln für GUIs

A.7.1 Einzel-Element-Regeln

In dieser Kategorie sind keine Regeln vorhanden.

A.7.2 Verbots-Beziehungsregeln

Regel: *GUIs dürfen keine Materialien kennen.*
Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Materials"

Regel: *GUIs dürfen keine Services kennen.*
Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Services"

Regel: *GUIs dürfen keine Automaten kennen.*
Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Automatons"

Regel: *GUIs dürfen keine FKs kennen.*
Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Tools"

Regel: *GUIs dürfen keine Tool-Klassen kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Tools".

Regel: *GUIs dürfen keine Monotool-Klassen kennen.*

Quelle: WAM-Architekten
Query: „GUIs Shouldn't Know Tools".

Regel: *Die GUI sollte ihre IAK nicht direkt kennen.*

Quelle: WAM-Architekten
Anmerkungen: Die Kopplung von der GUI zur IAK ist in Java mit dem Beobachtermuster lösbar (Listener). Die Kopplung ist aber vom Toolkit abhängig. In WAM gibt es dazu auch eine Lösung mit Präsentationsformen, die das Befehlsmuster verwendet.
Query: „GUIs Shouldn't Know Tools". Es kann allerdings nur geprüft werden, ob die GUI ihre IAK direkt kennt. Eine indirekte Kopplung ist nicht erkennbar. Bei einer Kopplung über das Befehlsmuster könnte nur geprüft werden, ob die IAK Befehle kennt und ob es Befehle gibt, die die GUI kennen. Damit wäre aber nicht sichergestellt, dass es eine Kopplung gibt.

A.7.3 Gebots-Beziehungsregeln

In dieser Kategorie sind keine Regeln vorhanden.

A.8 Regeln für Monotool-Klassen

Monotool-Klassen vereinigen die Funktionen von IAK, FK und Tool-Klasse in einer Klasse. Für den Umgang mit der GUI gelten daher die gleichen Regeln wie für IAKs. Für den Umgang mit Subwerkzeugen, Services und Automaten gelten die gleichen Regeln wie für FKs. Im Zusammenhang Kontext- und Subwerkzeug können Kontext-Monotool-Klassen auch Kontext-FKs und Sub-Monotool-Klassen auch Sub-FKs sein.

A.8.1 Einzel-Element-Regeln

In dieser Kategorie sind keine Regeln vorhanden.

A.8.2 Verbots-Beziehungsregeln

Regel: *Monotool-Klassen dürfen keine IAKs kennen.*

Quelle: WAM-Architekten
Query: „Monotool-classes Shouldn't Know IPs"

Regel: *Monotool-Klassen dürfen keine Tool-Klassen kennen.*

Quelle: WAM-Architekten
Anmerkungen: Ausnahmen sind Tool-Klassen von Subwerkzeugen.
Query: „Monotool-classes Shouldn't Know Tool-classes"

A.8.3 Gebots-Beziehungsregeln

Regel: *Die Monotool-Klasse nimmt Ereignisse von der Oberfläche entgegen.*

Quelle: WAM-Architekten
Anmerkungen: Mit „Oberfläche" ist hier die GUI gemeint.

Überprüfbarkeit: Eine Überprüfung wäre nur dynamisch möglich.

Regel: *Die Monotool-Klasse steuert die Präsentation an der Oberfläche.*

Quelle: [Zül98], S. 236
Anmerkungen: Mit „Oberfläche" ist hier die GUI gemeint. D.h. die IAK kennt die GUI unter ihrer vollen Schnittstelle und ruft Methoden an der GUI.
Query: „Monotool-classes Should Know Their GUI"

Regel: *Anforderungen werden über die Zuständigkeitskette verschickt. Eine Monotool-Klasse bekommt ihre zugehörige Kontext-Monotool-Klasse oder die Umgebung als Requesthandler gesetzt.*

Quelle: WAM-Architekten
Anmerkungen: Da monolithische Werkzeuge keine Tool-Klasse haben, wird sie natürlich nicht als Requesthandler gesetzt. Hier gibt es anscheinend eine Abweichung zwischen WAM und JWAM. In JWAM wird immer die Tool-Klasse oder Umgebung als Requesthandler gesetzt. Die Tool-Klasse leitet einen Request gegebenenfalls an eine Kontext-FK weiter. Die Umgebung ist eine WAM-Entwurfsmetapher. Sie stellt den Rahmen, in den alle anderen WAM-Elemente eingebettet sind. In JWAM können an ihr z.B. Werkzeuge registriert und über sie gestartet werden.
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur dynamisch feststellbar ist, welche Klasse als Requesthandler fungiert.

Regel: *Monotool-Klassen erzeugen sich ihre GUI selbst.*

Quelle: WAM-Architekten
Anmerkungen: Da monolithische Werkzeuge keine Tool-Klasse besitzen, müssen sie ihre GUI selber erzeugen.
Query: „The Monotool-class Should Initialize The GUI"

Regel: *Die Monotool-Klasse bearbeitet das Material.*

Quelle: WAM-Architekten
Erläuterungen: Die Bearbeitung von Materialien ist der Hauptzweck von Werkzeugen. Daneben rufen sie evtl. noch Automaten und/oder Services.
Query: „Monotool-classes Should Know Materials"

Regel: *Die Monotool-Klasse bearbeitet das Material nur unter der im Aspekt spezifizierten Schnittstelle.*

Quelle: WAM-Architekten
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da Aspekte nicht identifiziert werden können.

Regel: *Kontext-Monotool-Klassen kennen die volle Schnittstelle ihrer Sub-Monotool-Klasse.*

Quelle: WAM-Architekten
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht zwischen Kontext- und Sub-Monotool-Klassen unterschieden werden kann.

Regel: *Sub-Monotool-Klassen sind über das Ereignismuster (oder das Beobachtermuster) an ihre Kontext-Monotool-Klasse gekoppelt.*

Quelle: WAM-Architekten
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht zwischen Kontext- und Sub-Monotool-Klassen unterschieden werden kann.

Regel: *Die Monotool-Klasse startet die Subwerkzeuge dieses Werkzeugs.*

Quelle: WAM-Architekten
Überprüfbarkeit: Eine Überprüfung wäre nur dynamisch möglich.

A.9 Regeln für Automaten

A.9.1 Einzel-Element-Regeln

In dieser Kategorie sind keine Regeln vorhanden.

A.9.2 Verbots-Beziehungsregeln

Regel: *Automaten dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Mit dem Begriff „Werkzeuge“ sind sowohl komplexe als auch monolithische Werkzeuge gemeint, Automaten dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen. In bestimmten Fällen dürfen Automaten allerdings FKs kennen. Diese Ausnahmen sind in [Zül98] S.197/198 beschrieben. Sie sind in der Praxis allerdings noch nicht aufgetreten.
Query: „Automatons Shouldn't Know Tools“

A.9.3 Gebots-Beziehungsregeln

Regel: *Ein Automat arbeitet wie ein Werkzeug auf Materialien.*

Quelle: [Zül98] S. 195
Query: „Automatons Should Know Materials“

A.10 Regeln für Services

A.10.1 Einzel-Element-Regeln

Regel: *Referenzen auf Materialien in einem Service werden nicht öffentlich gemacht, d.h. ein Service gibt entweder Werte oder Kopien von Materialien.*

Quelle: [Zül04], S. 265
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da eine Unterscheidung zwischen Kopie und Original nicht möglich ist.

Regel: *Klienten sollten sich beim Service anmelden können, um über Zustandsänderungen benachrichtigt zu werden.*

Quelle: [Zül04], S. 267
Anmerkungen: Diese Regel braucht nur berücksichtigt zu werden, wenn eine zustandslose Implementierung des Services nicht möglich ist.
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nicht erkannt werden kann, ob es eine Möglichkeit zur Anmeldung beim Service gibt.

A.10.2 Verbots-Beziehungsregeln

Regel: *Services dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Mit dem Begriff „Werkzeuge“ sind sowohl komplexe als auch monolithische Werkzeuge gemeint, Services dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen.

Query: „Services Shouldn't Know Tools"

A.10.3 Gebots-Beziehungsregeln

Regel: *Services gehen mit Materialien um.*

Quelle: [Zül04], S. 265
Query: „Services Should Know Materials"

Regel: *Services können einen geforderten Service auch an andere Services oder Automaten delegieren.*

Quelle: [Zül04], S. 265
Überprüfbarkeit: Die Regel ist nur eine Empfehlung, deshalb ist es nicht sinnvoll zu überprüfen, ob sie eingehalten wird.

A.11 Regeln für Fachwerte

A.11.1 Einzel-Element-Regeln

Regel: *Bei der Implementierung von Fachwerten muss sichergestellt werden, dass sich die Exemplare dieser Klassen wie Werte verhalten.*

Quelle: [Zül98], S. 317
Anmerkungen: D.h. Fachwerte dürfen an der Schnittstelle nur Funktionen anbieten, aber keine Prozeduren.
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da es nicht möglich ist zu prüfen, ob eine Klasse Methoden hat, die das Objekt verändern.

Regel: *An der Schnittstelle von Fachwerten werden nur Operationen angeboten, die fachlich für den Umgang mit Werten dieses Typs relevant sind.*

Quelle: [Zül98], S. 318
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welche Operationen für den Umgang mit einem Fachwert relevant sind.

Regel: *An der Schnittstelle von zusammengesetzten Fachwerten gibt es neben den fachlichen Operationen auch solche, um die Fachwerte zusammenzusetzen und gezielt auf einzelne Bestandteile zugreifen zu können.*

Quelle: [Zül98], S. 318
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welchem Zweck eine Operation dient.

Regel: *Fachwerte müssen Operationen haben, die prüfen, ob der übergebene Parameter einen gültigen Wert für einen Fachwert darstellt.*

Quelle: [Zül98], S. 318
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welchem Zweck eine Operation dient.

Regel: *Fachwerte müssen Operationen haben, die es erlauben die Menge der für einen Fachwert-Typ gültigen Werte zu erfragen, falls diese Menge endlich ist.*

Quelle: [Zül98], S. 318
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welchem Zweck eine Operation dient.

Regel: *Es muss möglich sein, zu fragen, ob ein Fachwert nur eine endliche Anzahl von Werten annehmen kann.*

Quelle: [Zül98], S. 318
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welchem Zweck eine Operation dient.

Regel: *Ein Fachwert muss in der Lage sein, eine Zeichenkettenrepräsentation seines Wertes zu liefern.*

Quelle: [Zül98], S. 319
Überprüfbarkeit: Eine Überprüfung ist nicht möglich, da nur durch eine fachliche Interpretation festgestellt werden kann, welchem Zweck eine Operation dient.

Regel: *Der Konstruktor der Fachwert-Klasse darf nicht öffentlich sein.*

Quelle: [Zül98], S. 319
Query: „DVs Shouldn't Have a Public Constructor"

Regel: *Fachwert-Klassen bieten keine Operationen an, die es erlauben, das Fachwert-Objekt zu verändern.*

Quelle: [Zül98], S. 319/320
Anmerkungen: D.h. ein Fachwert darf keine Set-Methoden haben.
Query: „DVs Shouldn't Have "set"-Methods". Ob gegen diese Regel verstoßen wird, kann nur festgestellt werden, wenn verändernde Operationen den String "set" im Namen haben. Da diese Bezeichnung meistens eingehalten wird, macht es Sinn, mit einer Query zu prüfen, ob diese Regel eingehalten wird.

A.11.2 Verbots-Beziehungsregeln

Regel: *Fachwerte dürfen keine Werkzeuge kennen.*

Quelle: WAM-Architekten
Anmerkungen: Mit dem Begriff „Werkzeuge" sind sowohl komplexe als auch monolithische Werkzeuge gemeint, Fachwerte dürfen also weder IAKs, FKs, GUIs, Tool-Klassen noch Monotool-Klassen kennen.
Query: „DVs Shouldn't Know Tools"

Regel: *Fachwerte dürfen keine Services kennen.*

Quelle: WAM-Architekten
Query: „DVs Shouldn't Know Services"

Regel: *Fachwerte dürfen keine Materialien kennen.*

Quelle: WAM-Architekten
Query: „DVs Shouldn't Know Materials"

Regel: *Fachwerte dürfen keine Automaten kennen.*

Quelle: WAM-Architekten
Query: „DVs Shouldn't Know Automatons"

Regel: *Fachwerte dürfen keine Events kennen.*

Quelle: WAM-Architekten
Überprüfbarkeit: Verstöße gegen diese Regel können mit Queries nur gefunden werden, wenn das JDK als library in den Sotographen miteinbezogen würde. Der Aufwand für das Einlesen in den Sotographen würde sich dadurch deutlich erhöhen.

Regel: *Fachwerte dürfen keine Requests kennen.*

Quelle: WAM-Architekten
Überprüfbarkeit: Ob diese Regel befolgt wird, ist nur überprüfbar, wenn das JDK als library in den Sotographen miteinbezogen würde. Der Aufwand für das Einlesen in den Sotographen würde sich dadurch deutlich erhöhen.

Regel: *Fachwerte dürfen keine Elemente außer Fachwerte benutzen.*

Quelle: WAM-Architekten
Anmerkungen: Die Regel gilt nur in Bezug auf WAM-Elemente. Basistypen und andere Utility-Klassen dürfen natürlich benutzt werden.
Überprüfbarkeit: Ob diese Regel eingehalten wird, wird dadurch überprüft, dass nach Verstößen gegen die Regeln „Fachwerte dürfen keine Werkzeuge/Services/Automaten/Materialien kennen“ gesucht wird.

A.11.3 Gebots-Beziehungsregeln

Regel: *Zusammengesetzte Fachwerte sind aus mehreren Fachwerten zusammengesetzt.*

Quelle: [Zül98], S. 318
Überprüfbarkeit: Da nicht jeder Fachwert ein zusammengesetzter ist, ist es nicht sinnvoll, nach Verstößen gegen diese Regel zu suchen.

Anhang B

WAM-Queries

In diesem Anhang finden sich die Auflistungen aller WAM-Queries und Auszüge aus dem Quelltext einiger WAM-Queries. Die Auszüge reichen aus, um das Prinzip hinter den Queries zu verdeutlichen, da viele Queries ähnlich aufgebaut sind.

B.1 WAM-init

Diese Gruppe enthält nur die Query „init tables“

Der folgende Code ist ein Auszug aus der Query „init tables“, die Tabellen mit WAM-Elementen füllt.

```
QueryDescription {This query initializes the tables for the
WAM-queries".

If Preferences > Database > Optimize Type Access Number is
set, only the first type access reference per function/method
and type is shown by the relationship-queries. Otherwise, all
type access references will be shown.};

BooleanInput withJwam  "Include JWAM" FALSE ;

InputDescription withJwam
{Defines whether Elements of the JWAM-Framework are included
in the query result.};

InputToolTip withJwam
{Defines whether Elements of the JWAM-Framework are included
in the query result.};

DROP TABLE IF EXISTS Materials;

CREATE TABLE Materials

--select all classes that are materials
--(they are in the package that has "material"
--in its name and inherit from "Thing")
SELECT DISTINCT
     cla.symbolId, cla.name, cla.packageId, pack.path,
     cla.fileId, sym.name as "symName", sym.SymbolId as
     "symId", sym.signatureId
FROM
     Classes = cla,
     Classes = super,
     Packages = pack,
     InheritanceNestings = inestings,
     Symbols = sym
WHERE
```

```
     inestings.subSymbolId = cla.symbolId
     AND inestings.superSymbolId <> cla.symbolId
     AND inestings.superSymbolId = super.symbolId
     AND super.name like "Thing"
     AND sym.fileId = cla.fileId
     AND pac.packageId = cla.packageId
     AND (pack.name like "%Material%"
          OR pack.name like "%material%")
     #if Input.get(withJwam) = "FALSE"
          AND pack.path not like "%jwam%"
     #endif;

DROP TABLE IF EXISTS DomainValues;

CREATE TABLE DomainValues

--insert all classes into the DomainValues table
--that are domain values (they inherit from "DomainValue")
SELECT DISTINCT
     cla.symbolId, cla.name, cla.packageId, pac.path,
     cla.fileId, sym.name as "symName", sym.SymbolId as
     "symId", sym.signatureId
FROM
     Classes = cla,
     Classes = super,
     Packages = pac,
     InheritanceNestings = inestings,
     Symbols = sym
WHERE
     inestings.subSymbolId = cla.symbolId
     AND inestings.superSymbolId <> cla.symbolId
     AND inestings.superSymbolId = super.symbolId
     AND super.name like "DomainValue"
     AND sym.fileId = cla.fileId
     AND pac.packageId = cla.packageId
     #if Input.get(withJwam) = "FALSE"
          AND pac.path not like "%jwam%"
     #endif;
…
```

B.2 WAM-Elements

Queries in der Gruppe WAM-Elements:

- Automatons
- Domain Values (DVs)
- Functional Parts (FPs)
- GUIs
- Interaction Parts (IPs)
- Materials
- Monotool-classes
- Services

- Tool-classes
- Tools (Complex, grouped)
- Tools (FP, IP, GUI, Tool-class, Monotool-class)
- Tools (Mono, grouped)

B.3 WAM-Inheritance

Queries in der Gruppe WAM-Inheritance:

- Automaton Inheritance
- DV Inheritance
- FP Inheritance
- GUI Inheritance
- IP Inheritance
- Material Inheritance
- Service Inheritance
- Service Inheritance
- Tool Inheritance

B.4 WAM-Forbidden-Relationships

Queries in der Gruppe WAM-Forbidden-Relationships:

- Materials Shouldn't Know Tools
- Materials Shouldn't Know Services
- Materials Shouldn't Know Automatons
- FPs Shouldn't Know IPs
- FPs Shouldn't Know GUIs
- FPs Shouldn't Know Tool-classes
- IPs Shouldn't Know Materials
- IPs Shouldn't Know Tool-classes
- IPs Shouldn't Know Monotool-classes
- IPs Shouldn't Know Automatons
- IPs Shouldn't Know Services
- Tool-classes Shouldn't Know Services
- Tool-classes Shouldn't Know Automatons
- GUIs Shouldn't Know Materials
- GUIs Shouldn't Know Services
- GUIs Shouldn't Know Automatons
- GUIs Shouldn't Know Tools
- Monotool-classes Shouldn't Know IPs
- Monotool-classes Shouldn't Know Tool-classes
- Automatons Shouldn't Know Tools
- Services Shouldn't Know Tools
- DVs Shouldn't Know Tools
- DVs Shouldn't Know Services
- DVs Shouldn't Know Materials
- DVs Shouldn't Know Automatons

Nachfolgend ist der Code der Query „GUIs Shouldn't Know Tools" abgedruckt.

```
QueryDescription {The Query "init tables" in the category
"WAM-Init" must be executed before executing any of the que-
ries of this category.

This query shows all relationships from GUIs to tools};

OutputColumnWidths(-50, 1, -50, 1, -50, 1, -50, 1, 1, 1);

ResultTitle "GUIs that know tools (JWAM1.8)";

SELECT DISTINCT
     Guis18.symbolId as "refingClassId", Guis18.name as "re-
     fingClass", Guis18.symId as "refingSymbolId",
     Guis18.symName as "refingSymbol", Tools18.symbolId as
     "refedClassId", Tools18.name as "refedClass",
     Tools18.symId as "refedSymbolId", Tools18.symName as
     "refedSymbol", ref.refPos as "position of reference",
     ref.referenceType
FROM
     Guis18,
     Tools18,
     SymbolReferences = ref
WHERE
     Guis18.symId = ref.refingSymbolId
     AND Tools18.signatureId = ref.refedSignatureId
     AND Guis18.fileId <> Tools18.fileId
```

B.5 WAM-Enforced-Relationships

Queries in der Gruppe WAM-Enforced-Relationships:

- FPs Should Know Materials
- IPs Should Know Their GUI
- IPs Should Know Their FP
- The Tool-class Should Initialize The GUI
- The Tool-class Should Initialize The FP
- The Tool-class Should Initialize The IP
- Monotool-classes Should Know Materials
- Monotool-classes Should Know Their GUI
- The Monotool-class Should Initialize The GUI
- Automatons Should Know Materials
- Services Should Know Materials

B.6 WAM-Single-Element-Rules

Queries in der Gruppe WAM-Single-Element-Rules:

- Materials Shouldn't Have "get"- And "set"-Methods Only
- Complex Tools Consist Of GUI, FP, IP And Tool-class
- FPs Shouldn't Return Materials
- DVs Shouldn’t Have A Public Constructor

- DVs Shouldn't Have “set”-Methods

Im folgendenAbschnitt findet sich der Code für die Query „FPs Shouldn't Return Materials“.

```
QueryDescription {The Query "init tables" in the category
"WAM-Init" must be executed before executing any of the que-
ries of this category.

This query returns all functional parts that have methods that
return an object that is a material};

OutputColumnWidths(-50, 1, -50, 1, -50, 1);

ResultTitle "Functional parts which methods return materials";

--select those functional parts which methods return materials
SELECT DISTINCT
     FPs.symbolId, FPs.name as "functional part", Meth-
     ods.symbolId, Methods.name as "method", Materi-
     als.symbolId, Materials.name as "returned material"
FROM
     FPs,
     Materials,
     Methods
WHERE
     Methods.classId = FPs.symbolId
     AND Methods.typeId = Materials.symbolId

ORDER BY "functional part"
```

B.7 WAM-FollowUp

Queries der Gruppe WAM-FollowUp:

- Complex Tools FollowUp
- IPs Should Know Their GUI FollowUp
- Material Methods FollowUp
- Monotool-classes Should Know Their GUI FollowUp
- The Monotool-class Should Initialize The GUI FollowUp
- The Tool-class Should Initialize The GUI FollowUp
- The Tool-class Should Initialize The FP FollowUp
- The Tool-class Should Initialize The IP FollowUp

Anhang C
Ausschnitt aus dem ER-Modell des Sotographen

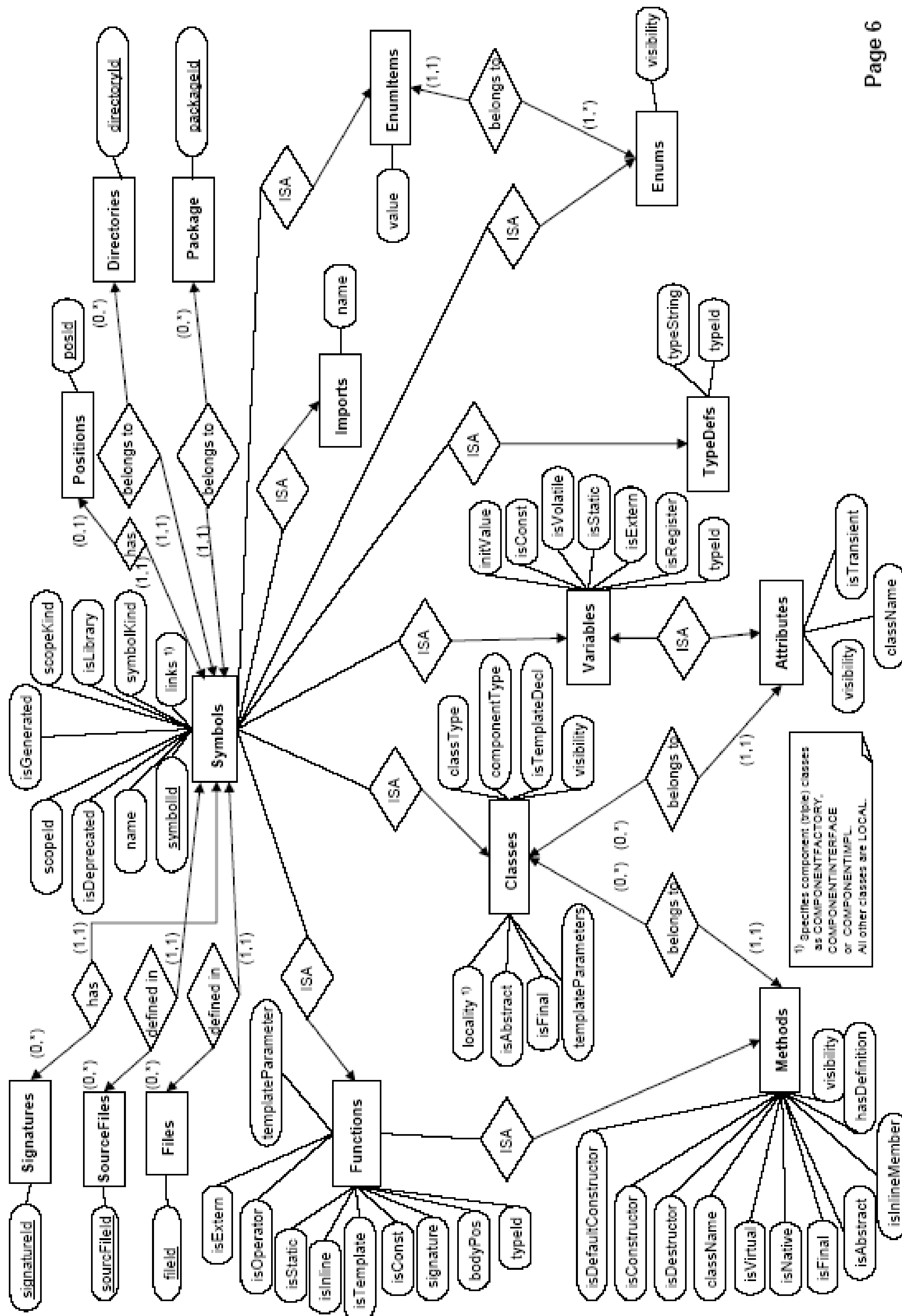

Abbildung C.0.1: Tabellen Symbols, Classes, Methods u.a.

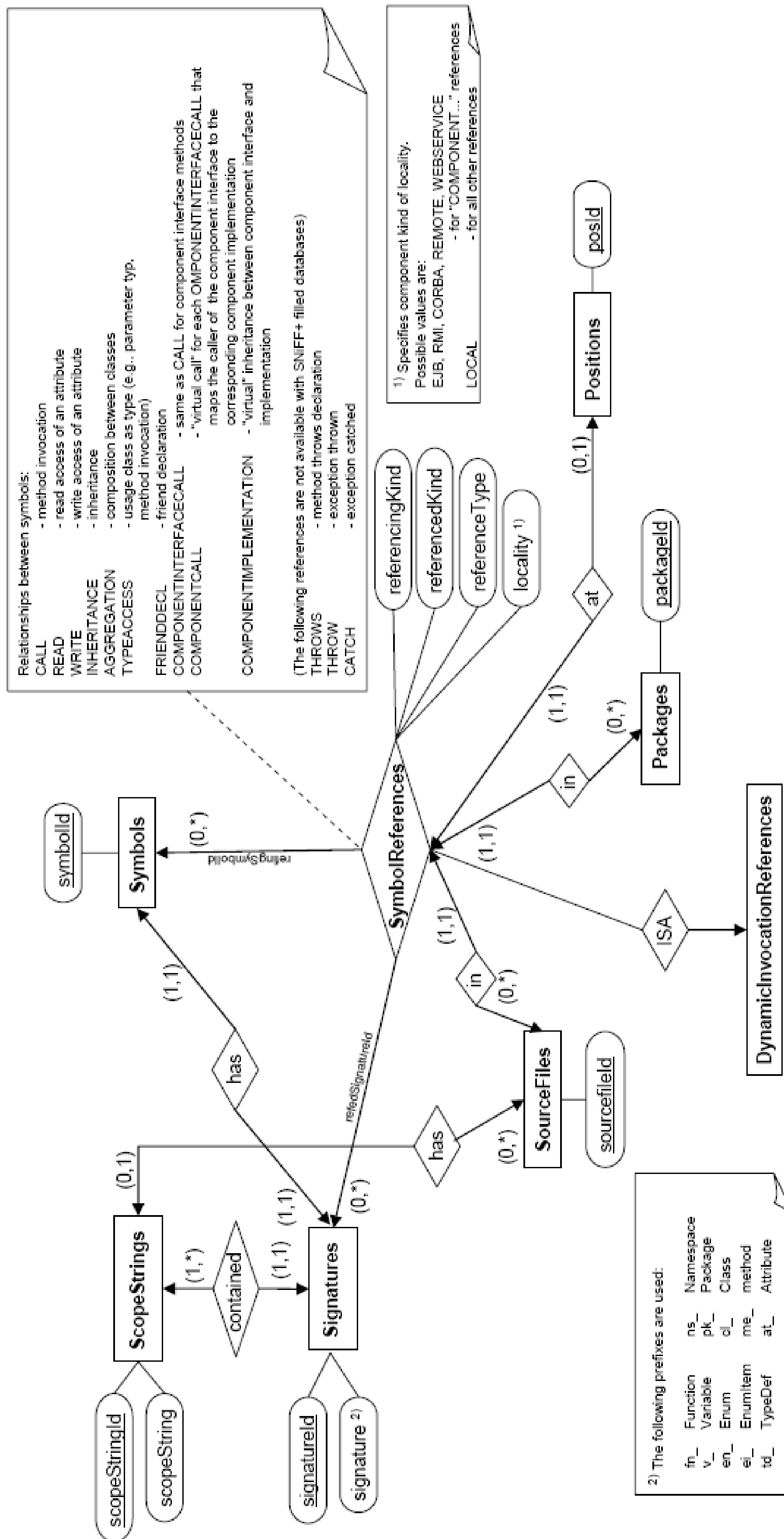

Abbildung C.0.2: Tabelle SymbolReferences u.a.

Literaturverzeichnis:

[Albin03] Stephen T. Albin: *The Art of Software Architecture*;
Wiley, 2003

[Bass98] Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice;*
Addison-Wesley, 1998

[Bisch03] W. Bischofberger: *Eclipse auf dem Prüfstand: Eine Fallstudie zur statischen Programmanalyse;*
OBJEKTspektrum 05/2003

[EMS] http://www.c1-wps.de/contell/cms/c1web/c1wps/site/loesungen/index.html

[Feijs98] L. Feijs, R. Krikhaar and R. van Ommering: *A Relational Approach to Support Software Architecture Analysis;*
In Software - Practice and Experience, vol. 28(4), pp. 371-400, April 1998

[Gamma98], Erich Gamma: *Design patterns : elements of reusable object-oriented software;*
Addison-Wesley, 1998

[Guo99] G. Y. Guo, J. M. Atlee and R. Kazman: *A Software Architecture Reconstruction Method*;
Proc. First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, February 1999.

[JWAM] http://sourceforge.net/projects/jwamtoolconstr/

[Mar96] R. Martin: *Granularity;*
C++ Report, 1996
(siehe: http://www.objectmentor.com/resources/articles/granularity.pdf)

[Murphy01] G. C. Murphy, D. Notkin and K. J. Sullivan: *Software Reflection Models: Bridging the Gap between Design and Implementation;*
In IEEE Transactions on Software Engineering, vol. 27, no. 4, pp. 364-380, April 2001

[Sefika96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell: *Monitoring Compliance of a Software System With Its High-Level Design Models;*
In 18th International Conference on Software Engineering, Berlin, Germany, March 1996.

[Sotograph] www.software-tomography.com

[Tvedt02] R. T. Tvedt, M. Lindvall and P. Costa: *Does the Code Match the Design? A Process for Architecture Evaluation;*
Proceedings of the International Conference on Software Maintenance (ICSM 2002) , IEEE Computer Society, 2002

[Zül98] Heinz Züllighoven: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz;*
Dpunkt Verlag, 1998

[Zül04] Heinz Züllighoven: *Object-Oriented Construction Handbook;*
Dpunkt Verlag 2004

Erklärung:

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

Hamburg, den 01. Juni 2005

Bettina Karstens

email: 0bkarste@informatik.uni-hamburg.de
Matrikelnr.: 5299997