Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln

Petra Becker-Pechau, Bettina Karstens, Carola Lilienthal

Arbeitsbereich Softwaretechnik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg
und C1 WPS Workplace Solutions
D-22527 Hamburg
becker@informatik.uni-hamburg.de
0bkarste@informatik.uni-hamburg.de
lilienthal@informatik.uni-hamburg.de

Abstract: Die innere Qualität von Softwaresystemen hat großen Einfluss darauf, wie gut sich die Software weiterentwickeln und warten lässt. Entscheidend für die innere Qualität ist die vom Entwicklungsteam gewählte Architektur. Neben den etablierten Schichtenarchitekturen verwenden Entwicklerteams flexiblere Architekturen, die sie informell beschreiben. Dass das entwickelte System der gewählten Architektur entspricht, muss in regelmäßigen Abständen überprüft werden. Die verfügbaren Werkzeuge zur Architekturüberprüfung decken lediglich den Teilaspekt der Schichtenbildung ab.

In diesem Artikel wird gezeigt, wie sich informelle Architekturbeschreibungen als Architekturregeln formulieren lassen. Ein großer Teil solcher Regeln lässt sich formalisieren und ihre Einhaltung automatisch überprüfen. Die Regeln einer ausgewählten Modellarchitektur werden auf drei Beispielsysteme angewendet und die Ergebnisse der automatischen Überprüfung ausgewertet und diskutiert.

1 Einleitung

Software, die langfristig eingesetzt werden soll, ist vielen Veränderungen unterworfen. Damit die Software die nötige Flexibilität hat, muss ihre innere Qualität sehr hoch sein. Neben der Qualität auf Quelltextebene zählen zur inneren Qualität die gewählte Architektur und ihre Umsetzung [BCK03]. Aus unserer Sicht entscheidet sich gerade auf der Ebene der Architektur, wie verständlich und damit wartbar ein System für Entwickler ist.

Die innere Architektur eines objektorientierten Systems lässt sich über die Strukturierung der Packages in Subsysteme und deren Anordnung in Schichten beschreiben. Diese so genannten Schichtenarchitekturen sind seit langem etabliert und weit verbreitet (s. [BG+00, Fo03, Kru95, SG96]). Für flexiblere Systeme werden in den letzten Jahren vermehrt komplexere Architekturen [Ev03, Fo03, Si04, Zü04] wie z.B. Servicearchitekturen diskutiert und eingesetzt. Solche Architekturen unterliegen einer Menge von Ar-

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

chitekturregeln, die sich nicht auf Schichten und eine festgelegte Menge an Subsystemen und deren Schnittstellen abbilden lassen.

Um in Softwareprojekten die Qualität der Architektur zu erhalten, muss regelmäßig überprüft werden, ob die Ist-Architektur der festgelegten Soll-Architektur entspricht. Im Projektverlauf ist es unvermeidlich, dass Differenzen zwischen der Ist- und der Soll-Architektur entstehen. Die Gründe dafür sind vielfältig. Manchmal müssen Probleme so schnell gelöst werden, dass keine Zeit bleibt, ein passendes Design zu entwickeln. Manchen neuen Entwicklern ist die Soll-Architektur nicht präsent, so dass sie ungewollt dagegen verstoßen.

Aufgrund des Umfangs heutiger Softwaresysteme werden neben manuellen Reviews zur Analyse der Architektur vermehrt Werkzeuge eingesetzt. Für die Überprüfung von Schichtenarchitekturen existieren eine Reihe von Werkzeugen: das LDM-Tool [SJ+05], XRadar[1], SonarJ[2], Sotograph [BKL04]. Diese Werkzeuge überprüfen unter anderem, ob Schichten nur auf die darunter liegenden Schichten zugreifen und ob Subsysteme nur über ihre Schnittstellen benutzt werden. Für komplexere Architekturen, die über Schichtenarchitekturen hinausgehen, stehen bisher keine spezialisierten Werkzeuge zur Verfügung.

Dieser Artikel stellt einen neuen Ansatz vor, um objektorientierte, komplexe Architekturen mit Werkzeugunterstützung überprüfbar zu machen. In Abschnitt 2 zeigen wir wie die Regeln komplexer Architekturen formalisiert werden können und führen die von uns als Beispiel gewählte WAM-Modellarchitektur ein. In Abschnitt 3 stellen wir die Ergebnisse unserer Analysen vor, die wir mit einem adaptierten Werkzeug bei der Untersuchung von drei verschiedenen Systemen gewonnen haben. Der Artikel schließt ab mit einer Diskussion verwandter Arbeiten und einem Ausblick.

2 Modellarchitektur und Architekturregeln

Wenn Entwickler ihrem System eine Architektur geben wollen, dann legen sie normalerweise Subsysteme, ihre Beziehungen und Schnittstellen sowie Schichten fest. Darüber hinaus beschreiben sie bei komplexeren Architekturen, wie die verschiedenen Elemente ihres Softwaresystems interagieren sollen. In vielen Projekten existieren informell ganze Sammlungen von Beschreibungen, die die Architektur anleiten. Wir haben solche Beschreibungen untersucht und festgestellt, dass sie sich als überprüfbare Regeln formalisieren lassen. Solche Regeln gelten für ein gesamtes Projekt, selbst wenn sich die Soll-Architektur im Projektverlauf iterativ verändert.

Wir konnten beobachten, dass diese Regeln nicht nur für einzelne Entwicklungsprojekte verwendet werden, sondern über mehrere Projekte Bestand haben (s. [Ev03, Fo03, Si04, Zü04]). Solche projektübergreifenden Regeln bezeichnen wir im Folgenden als Modell-

[1] xradar.sourceforge.net
[2] www.hello2morrow.de

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

architektur. Eine Modellarchitektur legt die Menge der Elementarten (z.B. „Service“, „Werkzeug“) und die Menge aller Regeln fest (s. [Ka05, Zü04]).

2.1 Architekturregeln

Wir definieren drei Arten von Regeln: *Elementregeln*, die nur ein Architekturelement betreffen, legen Einschränkungen für die Schnittstelle dieses Architekturelements fest oder spezifizieren, aus welchen Klassen es aufgebaut ist. *Gebotsregeln* schreiben Beziehungen zwischen Elementen bestimmter Art vor, und *Verbotsregeln* verbieten die Beziehungen zwischen Elementen bestimmter Arten. Gebots- und Verbotsregeln betreffen alle möglichen Beziehungsarten auf Sprachebene, wie z.B. Methodenaufrufe, Typzugriffe oder Vererbungsbeziehungen. Ein Beispiel für eine Verbotsregel ist: „Services dürfen nicht auf Werkzeuge zugreifen“.

Architekturregeln lassen sich dem Quelltext nicht entnehmen. Die Entwickler müssen die Regeln kennen, die für ein System gelten, um das Zusammenspiel der Klassen zu verstehen und nur Veränderungen vorzunehmen, die die Regeln nicht verletzen. Architekturregeln müssen daher außerhalb des Quelltextes definiert werden.

2.2 Architekturelemente

Um automatisch überprüfen zu können, ob die Ist-Architektur der Modellarchitektur entspricht, müssen die Architekturelemente im Quelltext identifiziert werden. Architekturelemente können je nach Elementart durch eine oder mehrere Klassen realisiert sein. Da im Quelltext für einzelne Klassen nicht eindeutig zu erkennen ist, zu welcher Elementart sie gehören, kann man verschiedene Möglichkeiten für die Identifizierung einsetzen:

- Namenskonvention: Die Klassennamen der Architekturelemente enthalten die Bezeichnung ihrer Elementart (z.B. „Service“).
- Typisierung: Im System existieren Oberklassen oder Interfaces, die für die jeweilige Elementart stehen.
- Annotationen: Im Klassentext werden Annotationen vorgenommen, die die Elementart kennzeichnen.

Sind die Elemente gefunden, so kann anhand formalisierter Regeldefinitionen automatisch überprüft werden, ob die im System vorhandenen Beziehungen und Schnittstellen den Regeln der gewählten Modellarchitektur entsprechen.

2.3 Die WAM-Modellarchitektur

Die WAM-Modellarchitektur ist Teil des WAM-Ansatzes [Zü04], der für die Entwicklung objektorientierter interaktiver Softwaresysteme eingesetzt wird. Das Kürzel WAM

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

steht für Werkzeug, Automat und Material. Der WAM-Ansatz wurde am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik an der Universität Hamburg entwickelt (s. [BG+99, BLZ99, BL+99, BL+00]). An der Universität und in verschiedenen Projekten in der Praxis wurde der Ansatz während der letzten 15 Jahre eingesetzt und weiterentwickelt. Der WAM-Ansatz definiert über die WAM-Modellarchitektur hinaus viele Richtlinien für das Vorgehen in Projekten.

Die WAM-Modellarchitektur definiert eine Vielzahl von Architekturregeln. Vergleichbare Modellarchitekturen mit ähnlich umfangreichen Regelmengen werden in Fowler [Fo03] Evans [Ev03] und Siedersleben [Si04] beschrieben. Die WAM-Regeln finden sich informell beschrieben in Züllighoven [Zü04]. Wir haben die Regeln formalisiert, in die drei Regelkategorien eingeteilt und Elementarten identifiziert.

2.4 Elemente und Regeln der WAM-Modellarchitektur

Die Hauptelementarten der WAM-Modellarchitektur und ihre typischen Beziehungen zueinander sind in Abbildung 1 dargestellt. Im Folgenden werden wir die einzelnen Architekturelementarten und einige der für sie geltenden Regeln vorstellen. Weitere Regeln können in [Ka05] nachgelesen werden.

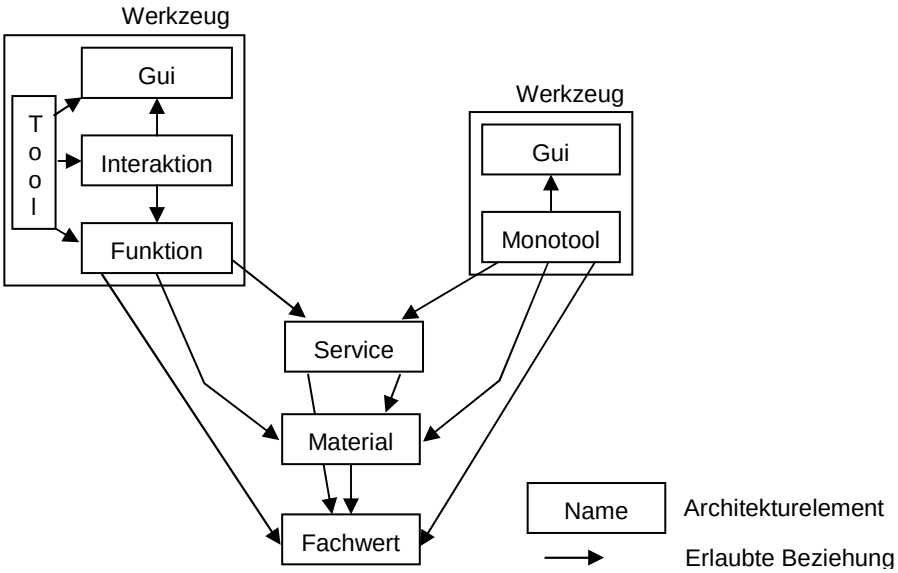

Abbildung 1: Architekturelementarten und erlaubte Beziehungen in der WAM-Modellarchitektur.

Fachwerte sind Wertobjekte, die anwendungsspezifische Werte verkörpern, wie z.B. Konto- und Kundennummern oder Geldbeträge. Normalerweise wird ein Fachwert durch eine Klasse realisiert. Für Fachwerte gelten folgende Regeln:

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

- Elementregel: Fachwertklassen haben keinen öffentlichen Konstruktor und keine Set-Operationen.
- Verbotsregel: Fachwerte dürfen keine Komponenten von Werkzeugen, keine Services und keine Materialien kennen.

Materialien realisieren anwendungsfachliche Konzepte, wie z.B. Konten. Ein Material kann aus einer, aber auch aus mehreren Klassen bestehen. Für Materialien gilt die folgende Regel:

- Verbotsregel: Materialien dürfen keine Komponenten von Werkzeugen und keine Services kennen.

Services bieten Dienste an, die die Werkzeuge bei ihren Aufgaben unterstützen sollen. Services führen oft Arbeiten auf mehreren Materialien gleichzeitig aus. Services können aus einer oder mehreren Klassen bestehen. Für Services gelten folgende Regeln:

- Verbotsregel: Services dürfen keine Komponenten von Werkzeugen kennen.
- Gebotsregel: Services müssen Materialien kennen.

Ein *Werkzeug* ermöglicht dem Benutzer Materialien zu bearbeiten. Es gibt zwei Konstruktionsvarianten für Werkzeuge. Komplexe Werkzeuge bestehen aus vier Klassen: einer Tool-Klasse, GUI, Interaktionskomponente (IAK) und Funktionskomponente (FK). Monolithische Werkzeuge bestehen aus zwei Klassen: einer GUI und der Monotool-Klasse, die die Funktionen der FK, IAK und Tool-Klasse vereinigt. Für Werkzeuge gelten die folgenden Regeln:

- Verbotsregel: Interaktionskomponenten dürfen keine Materialien kennen.
- Verbotsregel: Tool-Klassen dürfen keine Services kennen.
- Gebotsregel: Funktionskomponenten müssen Materialien kennen.
- Gebotsregel: Die Interaktionskomponente muss ihre Funktionskomponente kennen.
- Elementregel: Funktionskomponenten dürfen keine Materialien zurückgeben.

Abbildung 1 zeigt die typischerweise erlaubten Beziehungen zwischen den verschiedenen WAM-Elementarten. Einige Regeln sind in der Abbildung nicht dargestellt. Diese Regeln beschreiben die Beziehungen zwischen Elementen gleicher Art. So darf ein Material z.B. andere Materialien benutzen. Die Kardinalität der Beziehungen ist so festgelegt, dass innerhalb eines Werkzeuges alle Beziehungen 1-zu-1 Beziehungen sind. Alle anderen Beziehungen sind 1-zu-n Beziehungen, d.h. ein Werkzeug darf beispielsweise beliebig viele Materialien benutzen.

Insgesamt weist die WAM-Modellarchitektur 70 Regeln auf: 12 Elementregeln, 33 Verbotsregen und 25 Gebotsregeln. 52 Regeln (74,3 %) sind automatisch überprüfbar: sechs Elementregeln (50 % aller Elementregeln), 33 Verbotsregeln (100 % aller Verbotsregeln) und 13 Gebotsregeln (52 % aller Gebotsregeln). Nicht automatisch überprüfbar sind Regeln, die sich auf die Semantik von Operationen beziehen (Elementregeln) und die dynamische Aspekte der Architektur betreffen (Gebotsregeln).

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

3 Automatische Architekturüberprüfung

Der Sotograph ist ein Software-Analysewerkzeug, das zur Überprüfung der inneren Qualität auf Architektur- und Quelltext-Ebene eingesetzt werden kann (s. [BKL04, BL+02]). Für die Überprüfung auf Quelltext-Ebene bietet der Sotograph ca. 300 Metriken an. Um die innere Qualität auf Architektur-Ebene überprüfbar zu machen, können im Sotographen Subsysteme sowie Schichtenarchitekturen definiert werden. Für die Subsysteme können Schnittstellen spezifiziert werden und Regeln für die Benutzung zwischen Subsystemen festgelegt werden.

Die Regeln der WAM-Modellarchitektur lassen sich nicht auf der Ebene von Subsystemen und Schichten definieren, sondern müssen auf der Ebene von Architekturelementen beschrieben und überprüft werden. Hierfür haben wir die Adaptionsmöglichkeit des Sotographen genutzt, indem wir eigene Abfragen auf der Datenbank des Sotographen formuliert haben.

Um ein System zu analysieren, füllt der Sotograph zunächst eine Datenbank mit den strukturellen Informationen des Systems, die er aus dem Quelltext und Bytecode extrahiert [BKL04, Soto05]. In der Datenbank werden alle Artefakte des Systems und deren Beziehungen abgelegt. Artefakte sind Attribute, Methoden, Klassen, Dateien und Packages. Beziehungen sind Vererbung, Aufruf, Assoziation, Typverwendung und Attributzugriff. Die Datenbankstruktur des Sotographen ist offen gelegt und kann im Sotographen mit so genannten „Queries“ abgefragt werden. Queries werden in TQL (Tomography Query Language), einer Erweiterung von SQL geschrieben. Das Ergebnis einer Query wird in einer Tabelle angezeigt und kann auch graphisch dargestellt werden.

Um unseren Ansatz umzusetzen, haben wir zwei Arten von Queries geschrieben: Queries, die initial ausgeführt werden müssen, um die Architekturelemente zu identifizieren, und Queries, die nach Regelnverletzungen suchen.

3.1 Identifizierung der Architekturelemente

Die Identifizierung der Klassen im Quelltext, die jeweils einzeln oder gemeinsam ein Architekturelement realisieren, ist nicht trivial. Die in Abschnitt 2 vorgestellten Möglichkeiten der Identifizierung ließen sich für die WAM-Modellarchitektur wie folgt umsetzen:

- Namenskonvention: Außer für Materialien gibt es für alle Elemente der WAM-Modellarchitektur Namenskonventionen. Die Namenskonventionen werden allerdings nicht immer konsequent eingehalten.
- Typisierung: Die analysierten Softwaresysteme basieren auf dem Open Source Rahmenwerk JWAM[3]. Für die meisten WAM-Elementarten gibt es in JWAM Interfaces oder Klassen, die die Basisfunktionalität der WAM-Elemente bereit-

[3] www.jwam.de

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

stellen. Die Typisierung kann zur Identifizierung genutzt werden. So können z.B. Services darüber gefunden werden, dass sie das Interface „Service“ aus JWAM implementieren.

- Annotationen: Da wir nur Systeme untersucht haben, die mit Java 1.4 entwickelt wurden, standen Annotationen noch nicht zur Verfügung.

In unseren Untersuchungen haben wir alle Architekturelemente, die Subtypbeziehungen zu JWAM-Interfaces oder -Klassen haben, über Vererbung identifiziert. Diese Art, die Elemente im Quelltext zu finden, ist zuverlässiger als die Erkennung über Namenskonventionen. Lediglich zwei Arten von Architekturelementen, Materialien und GUIs, konnten nicht über ihre Vererbungsbeziehungen identifiziert werden. Die GUI-Elemente erkennen wir über den String „gui“ im Klassennamen, welches eine gebräuchliche Namenskonvention ist. Für die Materialien haben wir eine Kombination aus Vererbung und Namenskonvention gewählt. Alle Klassen, die in einem Package namens „material“ liegen und das Interface „Thing“ implementieren, werden als Materialien betrachtet.

3.2 Ergebnisse der Architekturüberprüfung in Projekten

Obwohl die WAM-Modellarchitektur im Detail im objektorientierten Konstruktionshandbuch [Zü04] beschrieben ist, haben wir festgestellt, dass nicht alle Architekten und Entwickler die Regeln in der gleichen Weise verstehen. Die Implementierung der Regeln als Queries führte zu einer lebhaften Diskussion der Regeln und sorgte dafür, dass sich die Architekten auf eine Sammlung von Regeln einigten.

Wir haben zuerst drei verschiedene WAM-Systeme überprüft, ein kleines Studentenprojekt, ein umfangreiches Studentenprojekt und ein professionelles System im Praxiseinsatz. Die Daten dieser Systeme stehen detailliert zur Verfügung und dienen als Basis für die folgenden Übersichten. Im Anschluss an die wissenschaftlich orientierten Untersuchungen haben wir die Regeln im professionellen Umfeld an inzwischen insgesamt fünf weiteren Systemen eingesetzt.

Zwei der untersuchten Systeme wurden von WAM-erfahrenen Entwicklern implementiert, eins der Studenten-Systeme von Anfängern. Zwei der Systeme wurden regelmäßigen manuellen Architekturreviews unterzogen. Daher erwarteten wir, dass einige Systeme fehlerfrei sein und andere viele Fehler enthalten würden. Die Systeme, die von Experten entwickelt wurden, haben die WAM-Regeln insgesamt besser umgesetzt, es traten jedoch trotzdem Regelverletzungen auf. Die Ergebnisse der Regelüberprüfungen wurden von Architekten der jeweiligen Systeme validiert. Dieses Ergebnis ähnelt dem von Bischofberger et al. [BKL04], die viele Systeme mit dem Sotographen untersucht haben und in allen Systemen Architekturverletzungen gefunden haben, sogar in den Systemen, die von erfahrenen Entwicklern und Architekten entwickelt wurden.

Die Architekten haben bei einem erneuten gründlichen Code-Review keine Regelverletzungen im System gefunden, die nicht auch durch die Queries entdeckt wurden (falsch Negative) (siehe Tabelle 1). 46 der durch die Queries gelisteten Regelverstöße konnten als falsch Positive identifiziert werden, weil einerseits Namenskonventionen nicht einge-

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

haltene wurden (27 falsch Positive) und andererseits die Queries Schwächen aufweisen (19 falsch Positive). Aus Sicht der Architekten handelt es sich bei den verbleibenden 43 Funden nicht um Architekturfehler sondern um gewollte Regelausnahmen.

	Funde	**Falsch Negative**	**Falsch Positive**	**Fehler in %**	**Gewollte Ausnahmen**	**Fehler in %**
Verbotsregeln	52	0	3	5,8	7	13,5
Gebotsregeln	107	0	41	38,3	36	33,6
Elementregeln	21	0	2	9,5	0	0

Tabelle 1: Zuverlässigkeit der verschiedenen Regelarten.

Für das von WAM-Anfängern entwickelt System war unsere Architekturüberprüfung besonders nützlich (s. Tabelle 2). In diesem System basierten viele Regelverletzungen auf fehlender Kenntnis der Regeln. Z.B. griff in einem System ein Material auf Services zu und umgekehrt. Daraus entstand ein sehr großer Zyklus, der das System schwer zu verstehen und schwer zu testen machte. Mit manuellen Architekturreviews war die Ursache des Zyklus nicht leicht zu erkennen, aber mit der automatischen Regelprüfung konnte dieser Verstoß gegen die Modellarchitektur schnell gefunden, behoben und damit der Zyklus deutlich verkleinert werden.

	Experten-projekt (31.931 LOC)	**Verletzungen/ 100.000 LOC**	**Anfänger-projekt (25.549 LOC)**	**Verletzungen/ 100.000 LOC**
Verbotsregeln	2	6	24	94
Gebotsregeln	0	0	0	0
Elementregel	5	16	10	39

Tabelle 2: Vergleich der Regelverletzungen zwischen zwei WAM-Systemen

Die Identifizierung der WAM-Elemente funktionierte fehlerfrei bei allen Elementen, die über Vererbungsbeziehungen zu JWAM gefunden wurden. Die Erkennung von Materialien und GUIs war etwas fehlerbehafteter, diese wurden über Namenskonventionen für die zugehörigen Packages bzw. Klassen gesucht. Für die Zukunft wäre es sinnvoll, Marker-Interfaces in JWAM einzuführen, über die Materialien und GUIs eindeutig erkannt werden. Tabelle 3 zeigt die Fehlerrate der Element-Identifizierung.

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

Elementart	Funde	Falsch Negative	Falsch Positive	Fehlerrate in %
Material	64	0	2	3,1
GUI	43	0	1	2,3

Tabelle 3: Identifizierung der Elemente Material und GUI durch Namenskonventionen

4 Verwandte Arbeiten

In der Literatur werden einige Ansätze zur Überprüfung von Softwarearchitekturen vorgestellt. In [FKO98] beschreiben Feijs, Krikhaar, und van Ommering einen Ansatz, die Ist-Architektur prozeduraler (nicht objektorientierter) Systeme zu untersuchen. Im Gegensatz zu unserem Ansatz werden lediglich die bestehenden Beziehungen zwischen Subsystemen identifiziert. Tvedt, Lindvall und Costa analysieren Systemarchitekturen mit Hilfe einer Metrik, die die Kopplung zwischen Komponenten berechnet [TCL02]. Die Ergebnisse dieser Metrik sind Beziehungen zwischen Klassen, Packages und Subsystemen. Bei beiden Ansätzen muss von Hand festgestellt werden, ob die bestehenden Beziehungen die Soll-Architektur verletzen. Der innere Aufbau der Subsysteme wird nicht thematisiert, die Identifikation von Subsystemen im Softwaresystem wird nicht erläutert.

Murphy, Notkin und Sullivan gehen in [MNS01] einen Schritt weiter. Neben der Analyse der Systemarchitektur kann eine Soll-Architektur angegeben werden, die erlaubte und vorgeschriebene Beziehungen zwischen Subsystemen festlegt. Abweichungen werden grafisch dargestellt. Im Gegensatz zu unserem Ansatz können jedoch keine Regeln für Subsystem-Arten definiert werden, die Sollarchitektur beschreibt lediglich konkrete Subsysteme und deren Beziehungen. Ferner können keine Regeln für die Schnittstelle von Subsystemen oder der innere Aufbau angegeben werden. Um Subsysteme im Quelltext zu identifizieren, werden relationale Ausdrücke über die Verzeichnisstruktur und die Dateinamen benutzt. Es ist nicht möglich, Architekturelemente über Typisierung zu identifizieren, was sich in unseren Untersuchungen besonders bewährt hat.

Sefika, Sane, und Campbell stellen in [SSC96] vor, wie Entwurfsmuster automatisch in Systemen gefunden werden können. Für die gefundenen Entwurfsmuster kann überprüft werden, ob sie korrekt implementiert wurden. Ähnlich wie bei unserem Ansatz wird hier die Soll- und Ist-Struktur untersucht. Allerdings bewegt sich dieser Ansatz lediglich auf Klassenebene und nicht auf Architekturebene, so dass es nicht möglich ist, den inneren Aufbau von Architekturelementen zu definieren. Im Gegensatz zu unserem Ansatz kann nicht angegeben werden, welche Muster wo zu erwarten sind, es wird lediglich nach Stellen gesucht, die den Mustern ähnlich sind.

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

5 Fazit und Ausblick

In diesem Artikel haben wir einen neuen Ansatz zur automatischen Überprüfung von Modellarchitekturen vorgestellt. 70 Regeln der WAM-Modellarchitektur haben wir beispielhaft formalisiert und konnten 52 von ihnen mit einem Werkzeug überprüfbar machen. An drei Systemen wurde überprüft, ob die 52 Regeln eingehalten wurden. Insgesamt hat das Werkzeug 180 Regelverletzungen festgestellt. Solche automatisch gefundenen Regelverletzungen müssen vom Entwicklungsteam in einem Architekturreview interpretiert werden, da sie nur ein Indikator für Verletzungen sind. 91 Verletzungen wurden in der Diskussion mit dem Entwicklungsteam als tatsächliche Regelverletzungen identifiziert.

Ein Teil der vom Entwicklungsteam zurückgewiesenen Regelverletzungen sind durch Schwächen der Queries verursacht worden. In der Zukunft planen wir, die Queries weiter zu verfeinern und so noch zuverlässigere Ergebnisse bei der Überprüfung von Architekturen zu erreichen. Andere zurückgewiesene Regelverletzungen waren gewollte Ausnahmen. An dieser Stelle müssen die Regeln der Modellarchitektur überprüft und gegebenenfalls geschärft werden.

Nicht nur die in diesem Artikel verwendete WAM-Modellarchitektur enthält Architekturregeln. Für andere Modellarchitekturen sollten die Regeln ebenfalls formalisiert und automatisch überprüfbar gemacht werden. Hierfür kann unser Ansatz als Vorlage dienen. Dabei muss überprüft werden, ob die von uns gewählten Regelkategorien auch für andere Modellarchitekturen verwendbar sind.

Literaturverzeichnis

[BG+00] D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, H. Züllighoven, Domain-driven framework layering in large systems., ACM Computing Surveys 32(1es): 5, 2000

[BCK03] L. Bass, P. Clements, R. Kazman: Software Architecture in Practice, Addison-Wesley, Boston, MA, USA, 2003.

[BKL04] W. Bischofberger, J. Kühl, S. Löffler: Sotograph – A Pragmatic Approach to Source Code Architecture Conformance Checking. In Flavio Oquendo, Brian Warboys, Ron Morrison (eds.): Software Architecture: First European Workshop, EWSA 2004, Proceedings, LNCS, pp. 1-9, Springer-Verlag, St Andrews, UK, May 2004.

[BG+99] W.-G. Bleek, T. Görtz, C. Lilienthal, M. Lippert, S. Roock, W. Strunk, U. Weiss, H. Wolf, Interaktionsformen zur flexiblen Anbindung von Fenstersystemen., Universität Hamburg. Fachbereich Informatik. Fachbereichsmitteilung FBI-HH-M-285/99. April., 1999.

[BLZ99] W.-G. Bleek, C. Lilienthal, H. Züllighoven: Frameworkbasierte Anwendungsentwicklung (Teil 4): Fachwerte, OBJEKTspektrum 5/99, September/Oktober 99, S. 75-80, 1999.

[BL+99] W.-G. Bleek, C. Lilienthal, M. Lippert, S. Roock, W. Strunk, H. Wolf, H. Züllighoven: Frameworkbasierte Anwendungsentwicklung (Teil 2): Die Konstruktion interaktiver Anwendungen, OBJEKTspektrum 2/99, März/April, S. 78-83, 1999.

[BL+02] H. Breitling, C. Lewerentz, C. Lilienthal, M. Lippert, F. Simon, F. Steinbrückner: External Validation of a Metrics-Based Quality Assessment of the JWAM Framework,

Proceedings der Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik 28.-31. März 2006, Leipzig,

Biel, B.; Book, M.; Gruhn, V. (Hers.): Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), SE 2006, Bonn, 2006, S. 27-38, Copyright 2006 Gesellschaft für Information

In: Software- Messung und Bewertung. von Rainer Dumke (Hrsg.), Dieter Rombach (Hrsg.), Deutscher Universitäts-Verlag, 2002

[BL+00] H. Breitling, C. Lilienthal, M. Lippert, H. Züllighoven: The JWAM Framework: Inspired By Research, Reality-Tested By Commercial Utilization, in Proceedings of OOPSLA 2000 Workshop: Methods and Tools for Object-Oriented Framework Development and Specialization, 2000.

[Ev03] E. Evans: Domain-Driven Design. Addison-Wesley, Boston, MA, USA, August 2003

[FKO98] L. Feijs, R. Krikhaar, R. van Ommering: A Relational Approach to Support Software Architecture Analysis. In Software - Practice and Experience, vol. 28(4), pp. 371-400, April 1998.

[Fo03] M. Fowler: Patterns of Enterprise Application Architecture. Pearson Education, Boston, MA, USA, 2003.

[Ka05] B. Karstens: Regeln der WAM-Modellarchitektur, Diplomarbeit an der Universität Hamburg, Fachbereich Informatik, Hamburg, Oktober 2005.

[Kru95] P. Kruchten.: "The 4+1 View Model of Architecture", IEEE Software, Vol. 12, No. 6, 1995 pp. 42-50.

[MNS01] G. C. Murphy, D. Notkin, 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.

[SJ+05] N. Sangal, E. Jordan, V. Sinha, D. Jackson: Using Dependency Models to Manage Complex Software Architecture, , OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

[SSC96] M. Sefika, A. Sane, R. 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.

[SG96] M. Shaw, D. Garlan: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[Si04] J. Siedersleben: Moderne Softwarearchitektur. Dpunkt-Verlag, July 2004.

[TCL02] R. T. Tvedt, P. Costa, M. Lindvall,: 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.

[Zü04] H. Züllighoven: Object-Oriented Construction Handbook. Dpunkt-Verlag, Copublication with Morgan-Kaufmann, 2004.