

Die stilbasierte Architekturprüfung

Ein Ansatz zur Prüfung implementierter Softwarearchitekturen auf Architekturstil-Konformanz

Dissertation

zur Erlangung des akademischen Grades Dr. rer. nat.
an der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim Fachbereich Informatik von
Petra Becker-Pechau
Hamburg, 2014

Gutachter:

Prof. Dr.-Ing. Heinz Züllighoven, Universität Hamburg

Prof. Dr. Ralf Reussner, Karlsruher Institut für Technologie (KIT)

Tag der Disputation: 4. Dezember 2015

Zusammenfassung

Wenn Entwicklungsteams ihre Softwarearchitektur entwerfen, definieren sie üblicherweise eine Menge von Architekturvorgaben, welche bei der Programmierung einzuhalten sind. Es zeigt sich jedoch in der Praxis, dass die implementierte Architektur oft unbemerkt von ihren Vorgaben abweicht: die Architektur erodiert. Erodierende Softwaresysteme verlieren Schritt für Schritt an Verständlichkeit, Änderbarkeit und Wartbarkeit, bis sie in extremen Fällen durch Neuentwicklungen ersetzt werden müssen. Als Gegenmaßnahme zur Architekturerosion dienen Prüfungen auf Architekturkonformanz. Sie ermitteln, ob die im Quelltext implementierte Architektur den Vorgaben entspricht. Bestehende Ansätze adressieren Vorgaben auf den Abstraktionsebenen des Quelltextes oder der Softwarearchitektur. Bisher fehlte es jedoch an Möglichkeiten, die Konformanz zu Vorgaben zu prüfen, die sich auf einer Metaebene zur Architektur befinden.

Für Architekturvorgaben auf der Metaebene existiert noch keine einheitliche Benennung und Abgrenzung. Diese Arbeit diskutiert in ihrem konzeptionellen Teil verschiedene Arten von Architekturvorgaben, wie Referenzarchitekturen, Architekturmuster und Architekturstile, grenzt diese voneinander ab und entwickelt eine Konzeptualisierung für Vorgaben in Form expliziter Architektur-Metamodelle. Für diese Metamodelle verwendet die Arbeit den Begriff des Architekturstils. Die erarbeitete Begriffsbildung basiert auf Literaturarbeit und auf der Analyse bestehender, praxisrelevanter Architekturstile. Die Arbeit zeigt, dass Stile für die Qualität von Softwaresystemen eine entscheidende Rolle spielen. Stile erlauben, Architekturen konsistent zu strukturieren und dadurch verständlicher zu gestalten. Ferner unterstützen Stile eine Wiederverwendung auf Architekturebene, da sich mit ihnen die prinzipielle Struktur von Softwarearchitekturen von einem System auf ein anderes übertragen lässt, selbst wenn die Systeme zu unterschiedlich sind, um nach der selben Architektur strukturiert zu werden.

In ihrem konstruktiven Teil präsentiert diese Arbeit einen neuen Ansatz: die stilbasierte Architekturprüfung. Dieser Ansatz ermöglicht, die Konformanz der in Quelltexten implementierten Architektur zu Architekturvorgaben in Form frei definierbarer Architekturstile zu prüfen. Da die implementierte Architektur dem Quelltext nicht vollständig zu entnehmen ist, beinhaltet der Ansatz ein Konzept, mit dem sich die fehlenden Architekturinformationen einfach in den Quelltext integrieren lassen. Damit adressiert er das in der Praxis verbreitete Problem, dass diese Informationen veralten, wenn sie getrennt von dem Quelltext dokumentiert werden. Korrekte Architektur-Zusatzinformationen sind jedoch essentiell für Konformanzprüfungen, denn sie sind Voraussetzung für korrekte Prüfergebnisse.

Der Ansatz der stilbasierten Architekturprüfung ist auf objektorientierte, statisch getypte Sprachen ausgerichtet. Im Rahmen dieser Arbeit wurde eine beispielhafte Umsetzung für die Programmiersprache Java konzipiert. Für diese Umsetzung wurden zwei prototypische Prüfwerkzeuge erstellt. Mit Hilfe der Werkzeuge ließen sich sechs beispielhafte Softwaresysteme erfolgreich auf Stilkonformanz prüfen. Die Ergebnisse wurden durch Interviews abgesichert. In allen Systemen konnten bisher unbemerkte Verstöße gegen den gewählten Architekturstil aufgedeckt werden. Eines der Werkzeuge ist in die Entwicklungsumgebung Eclipse integriert. Es zeigt, dass sich die Prüfung unmittelbar während der Programmierung durchführen lässt. Auf diese Weise werden Verstöße gegen den gewählten Stil bereits zum Zeitpunkt und am Ort ihrer Entstehung deutlich und können unmittelbar korrigiert werden.

Mit dem vorgestellten Ansatz lassen sich stilbasierte Systeme zukünftig vor Architekturerosion schützen. So liefert diese Arbeit einen Beitrag, Softwaresysteme langfristig verständlich, änderbar und wartbar zu erhalten.

Abstract

When development teams design their software architecture, they usually define a set of architectural constraints that should be followed by the programmers. But in real world projects the implemented architecture often gradually deviates from the constraints, unnoticed by the developers: the architecture erodes. Eroding software systems progressively lose comprehensibility, changeability and maintainability. In extreme cases they finally need to be replaced. A countermeasure against architecture erosion is provided by architecture conformance checks. They determine whether the implemented architecture meets the constraints of the architectural design. Current approaches address constraints on two abstraction levels: source level and architectural level. Until now however, it was not possible to check efficiently the conformance with constraints defined at a meta level.

There are still no commonly accepted terms or criteria for demarcation of constraints at a meta level. This dissertation discusses different kinds of architectural constraints, like reference architectures, architectural patterns and styles, and develops a conceptualization for constraints at a meta level representing explicit architectural meta models. These meta models are called architectural style. The provided classification is based on a literature review and on the analysis of real life architectural styles. This dissertation shows that styles are vital for software quality. Styles are the key to structural consistent architectures and they enable software architects to re-use architectural concepts for new systems.

Based on its conceptual results, this dissertation presents a new approach: the style-based architecture conformance checking approach. This approach checks the conformance of the implemented architecture according to a user-defined architectural style. As the implemented architecture cannot be computed from the source code alone, the dissertation provides a concept how to include the missing architectural information directly into the source code. This relates to the commonly occurring problem of additional information getting outdated and inconsistent with the source code if documented separately from the code. Without this information correct conformance checks are impossible.

The presented approach has been developed for object-oriented, statically typed programming languages and provides an exemplary implementation for the programming language java. The implementation includes two prototypical checking tools that had been used to check the implemented architecture of six software systems for architectural style conformance. The results have been discussed in subsequent interviews. The new approach detects unnoticed violation against the chosen style in all systems. One of the checking tools is implemented as a plug in for the development environment Eclipse. It demonstrates the possibility to perform the checks directly while programming. Programmers are warned immediately if their newly added or changed code violates the chosen architectural style. Thus, violations can be corrected at the time and on the spot of occurrence even before entering the common code repository.

From now on development teams can protect their style-based systems from architectural erosion. Therefore, this dissertation contributes to enable long living, comprehensible, changeable and maintainable software systems.

Vorwort

Mein Interesse daran, was eine gute Softwarearchitektur ausmacht, welche Strukturierungsmöglichkeiten zur Verfügung stehen und wie man eine geeignete Architektur in Softwareprojekten auswählen und umsetzen sollte, wurde bereits während meines Studiums der Softwaretechnik an der HAW Hamburg geweckt. Hier danke ich allen Lehrenden, die mir interessante Themen und Fragestellungen der Informatik nahe gebracht haben, vor allem den Professoren Jörg Raasch und Guido Pfeiffer, die mir die Möglichkeit gaben, mich in meiner Diplomarbeit mit dem Themenbereich Softwarearchitektur zu beschäftigen, und die in lebhaften Diskussionen deutlich werden ließen, dass dieses Thema aktuell und spannend ist und keineswegs bereits abschließend geklärt. Ihnen und vielen weiteren Lehrenden an der HAW danke ich außerdem, dass sie mir Mut gemacht haben, meine fachlichen Interessen im wissenschaftlichen Bereich zu verfolgen.

Nach mehreren Jahren in der Wirtschaft hat mich Jörg Raasch mit meinem zukünftigen Doktorvater, Heinz Züllighoven, bekannt gemacht. Dieser gab mir die Möglichkeit, sowohl in seiner Firma, der Workplace Solutions, als auch in der Softwaretechnikgruppe an der Universität Hamburg seinen Architekturentwurf nach dem Werkzeug&Material-Ansatz kennen zu lernen. Ich danke ihm ganz herzlich für diese Erfahrung, bei der mir die Vorteile und Möglichkeiten guter Architekturen und Architekturstile äußerst eindrucksvoll demonstriert wurden: Gleich in den ersten Arbeitstagen, in meinem ersten Softwareprojekt, wurde deutlich, welchen enormen Vorteil der einheitliche und durchgängige Aufbau von Softwaresystemen nach einem gut ausgearbeiteten Strukturierungsprinzip bietet. Derart schnell hatte ich mich noch nie in neue Systeme eingearbeitet, selbst der Wechsel zwischen Projekten ganz unterschiedlicher Domänen fiel verblüffend leicht. Dasselbe beobachtete ich bei meinen Kollegen. Nun stand für mich erst recht fest: das Thema Softwarearchitektur sollte im Mittelpunkt meiner Promotion stehen. Vielen Dank an Heinz Züllighoven, dass er mich in diesem Vorhaben unterstützt und betreut hat.

Ich danke allen, die mit mir die Inhalte meiner Arbeit diskutiert haben. Hier möchte ich zusätzlich zu Heinz Züllighoven vor allem meinen Zweitgutachter Ralf Reussner mit seiner Arbeitsgruppe am KIT nennen und meine Kollegen in der Softwaretechnikgruppe an der Universität Hamburg. Die lebhaften Diskussionen in der Softwaretechnikgruppe über all unsere Forschungsthemen, insbesondere auf unseren Klausurtagen, werden mir in guter Erinnerung bleiben. Auch danke ich der ehemaligen Leiterin der Gruppe, Christiane Floyd, die mir in einer schwierigen Phase meines Promotionsprojektes geholfen hat, wieder einen produktiven Weg zu finden.

Gut ergänzt wurde meine Forschung durch studentische Arbeiten, hier danke ich meinen Kollegen, die mich dabei unterstützt haben, eine ganze Gruppe von Studenten mit Arbeiten in dem Themenbereich Softwarearchitekturen zu leiten, und ich danke namentlich Bettina Koch (geb. Karstens) und Arne Scharping, die mit ihren Diplomarbeiten Wesentliches zu meiner Forschung beigetragen haben.

Inhaltlich gut vorangebracht und enorm motiviert haben mich die verschiedenen Konferenzbesuche in Europa und Nordamerika. Ich bin dankbar, dass mir dies im Rahmen meiner Stelle als wissenschaftliche Mitarbeiterin ermöglicht wurde. Auf den Konferenzen konnte ich meine Forschungsergebnisse vorstellen und diskutieren. Ich danke all den Kollegen und Freunden, mit denen ich die Konferenzbeiträge gemeinsam vorbereitete und die Konferenzen besuchte, ich freue mich noch jetzt, wenn ich daran denke. Die Reisen bescherten mir spannende Erlebnisse und viel interessanten fachlichen Austausch mit Informatikern mit ganz unterschiedlicher fachlicher Ausrichtung und Erfahrung.

Ein Promotionsprojekt erfordert Zeit, aber es will gleichzeitig auch finanziert sein. Ich danke allen, die mich auf verschiedenstem Wege hierbei unterstützt haben, durch passende, unbürokratische Teilzeitstellen an der Uni Hamburg und in der Wirtschaft, durch Lehraufträge an der HAW Hamburg sowie durch mein Stipendium im Rahmen des Pro-Exzellenzia-Programms.

Im Verlauf meiner Promotion hatte ich mehrfach das Glück, mit anderen Promovierenden zusammenzuarbeiten. Insbesondere hervorheben und danken möchte ich Carola Lilienthal, Frank Heitmann und Beate Ritterbach. Mit jedem der drei bin ich einen wesentlichen Abschnitt der Wegstrecke gemeinsam gegangen. Zu Beginn meiner Promotion hat mich Carolas Energie mitgerissen. Wir haben ein Büro an der Uni geteilt, gemeinsam an unseren Exposees gearbeitet und unsere Texte gegengelesen. Ihr danke ich auch für die gemeinsame Betreuung studentischer Abschlussarbeiten im Rahmen unserer Promotionsprojekte. Sowohl bei der Themenfindung als auch bei der Betreuung der Arbeiten führten wir viele interessante fachliche Diskussionen. An Frank Heitmann ganz lieben Dank für die tolle gemeinsame Zeit in der Staatsbibliothek Hamburg, wo wir beharrlich an unseren Dissertationen schrieben, uns gegenseitig motivierten und dabei auch noch spannende Diskussionen über die Lehre in der Informatik führten. Und vielen Dank für das detailgenaue Gegenlesen insbesondere meiner formalen Definitionen, das war ausgesprochen hilfreich. Die letzte Schreibphase meiner Arbeit habe ich vorwiegend mit Beate Ritterbach verbracht, der ich ebenso sehr danke. Bekanntlich sind die letzten Meter besonders weit, da haben mir die angenehme Atmosphäre und die bei Bedarf spontan geführten Diskussionen über einen guten Textaufbau sowie das Feedback zu mehreren Textabschnitten sehr geholfen. Gemeinsam haben wir es über die Ziellinie geschafft.

Zum jetzigen Zeitpunkt ist mir natürlich die Disputation besonders frisch in Erinnerung. Ich danke allen ganz herzlich, die dazu beigetragen haben, dass es ein so gelungener Abschluss wurde. Vielen Dank auch dafür, dass sich so viele danach die Zeit genommen haben, in angenehmer Atmosphäre anzustoßen und sich mit mir zu freuen!

Ich danke all meinen Freunden für Ihren Zuspruch und die gelegentliche, sehr hilfreiche Ablenkung von meiner Promotion, aber auch für ihr Verständnis, wenn ich mich zurückgezogen habe, um konzentriert und in Ruhe voranzukommen. Auch danke ich den Freunden, die Texte meiner Arbeit gegengelesen haben, sogar wenn die Informatik bei einigen eher nicht im Mittelpunkt ihrer Interessen liegt.

Aus tiefstem Herzen danke ich meinem Mann Jörg Pechau. Ich habe das große Glück, dass wir ähnliche Interessen innerhalb der Informatik teilen und beide viel Freude am fachlichen Austausch finden. Ich erinnere mich an mehrere Situationen, in denen wir mit Begeisterung, ganz spontan in der Freizeit, interessante und für meine Promotion sehr hilfreiche Fachdiskussionen führten. Auch die gemeinsamen Konferenzbesuche und die dort von uns geleiteten Workshops möchte ich nicht missen. Ich danke ihm für all die schönen Erlebnisse, seine Unterstützung, das umfangreiche Gegenlesen meiner Textentwürfe sowie seine unerschütterliche, äußerst motivierende Begeisterung für mein Forschungsthema.

Inhalt

1	Einleitung	1
1.1	Kontext	1
1.2	Problemstellung	4
1.3	Die untersuchten Fragestellungen	5
1.4	Zielsetzung	7
1.5	Einordnung der Arbeit	7
1.6	Methodik	8
1.7	Aufbau der Arbeit	9
2	Softwarearchitektur	11
2.1	Warum Softwarearchitektur?	11
2.2	Diskussion verschiedener Softwarearchitektur-Definitionen	14
2.3	Zusammenhang zwischen Quelltext und Softwarearchitektur	17
2.4	Ist- und Soll-Architektur	24
2.5	Grenzen des Architekturentwurfs mit Soll-Architekturen	26
2.6	Zusammenfassung	30
3	Architekturstile	31
3.1	Diskussion verschiedener Architekturstil-Definitionen	31
3.2	Innerer Aufbau von Architekturstilen	34
3.3	Fallbeispiele	36
3.3.1	Der Architekturstil des WAM-Ansatzes	37
3.3.2	Der Quasar-Architekturstil	41
3.4	Vorteile des stilbasierten Architekturentwurfs	45
3.5	Zusammenfassung	49
4	Prüfungen auf Architekturtreue	51
4.1	Architekturerosion	51
4.2	Prüfungen auf Architekturtreue im Überblick	53
4.3	Software-Reflexionsmodelle	56
4.4	Weitere bisherige Ansätze und Werkzeuge	65
4.4.1	Software-Reflexionsmodelle mit getypten Beziehungen	65
4.4.2	Hierarchische Software-Reflexionsmodelle	67
4.4.3	Abhängigkeitsmodelle in Matrizendarstellung	68
4.4.4	Der LISA-Ansatz	69
4.4.5	Konformanzprüfung ohne explizite Soll-Architektur	70

4.4.6	Sonstige Werkzeuge	71
4.4.7	Zusammenfassung	74
4.5	Abgrenzung zu verwandten Forschungsfeldern	77
4.5.1	Die Unified Modeling Language UML	77
4.5.2	Modellgetriebene Software-Entwicklung	77
4.5.3	Architekturbeschreibungssprachen	78
4.6	Erste bisherige Möglichkeiten zur Prüfung auf Stiltreue	78
4.6.1	ArchJava	79
4.6.2	ArchMapper	80
4.6.3	Abgrenzung zur stilbasierten Architekturprüfung	80
4.7	Zusammenfassung	82
5	Das Konzept der stilbasierten Architekturprüfung	85
5.1	Beitrag der stilbasierten Architekturprüfung	85
5.2	Die stilbasierte Architekturprüfung im Überblick	89
5.3	Die stilbasierte Ist-Architektur ermitteln	90
5.3.1	Berechnungsschritt (a): die Quelltextstruktur ermitteln	92
5.3.2	Berechnungsschritt (b): die stilbasierte Ist-Architektur ermitteln	95
5.3.3	Zusatzinformationen und Quelltext verknüpfen	98
5.4	Den Architekturstil beschreiben	101
5.5	Die Verstöße berechnen	104
5.5.1	Berechnungsschritt (c): Die Ist-Architektur auf Stiltreue prüfen	106
5.5.2	Berechnungsschritt (d): Verstöße in den Quelltext verfolgen	112
5.6	Zusammenfassung	117
6	Ausbaustufen der stilbasierten Architekturprüfung	119
6.1	Beziehungsregeln für verschiedene Beziehungstypen	120
6.1.1	Berechnungsschritte und Eingabeinformationen	122
6.1.2	Abgrenzung zu bisherigen Ansätzen	126
6.2	Schnittstellenregeln	126
6.2.1	Berechnungsschritte und Eingabeinformationen	127
6.2.2	Abgrenzung zu bisherigen Ansätzen	141
6.3	Hierarchische Architekturen	142
6.3.1	Enthältregeln: Berechnungsschritte und Eingabeinformationen	145
6.3.2	Interne Gebotsregeln: Berechnungsschritte und Eingabeinformationen	148
6.3.3	Beziehungs- und Schnittstellenregeln in hierarchischen Architekturen	152
6.3.4	Abgrenzung zu bisherigen Ansätzen	152
6.4	Zusammenfassung	153

7	Eine beispielhafte Umsetzung	155
7.1	Die stilbasierte Architekturprüfung für Java-Systeme.....	156
7.1.1	Die Quelltextstruktur für Java-Systeme ermitteln	156
7.1.2	Die Zusatzinformationen mit Java-Annotationen beschreiben	157
7.1.3	Den Architekturstil mit XML beschreiben	160
7.2	Der StyleBasedChecker: Ein Werkzeug zur stilbasierten Architekturprüfung.	165
7.2.1	Benutzung des StyleBasedCheckers	165
7.2.2	Technische Realisierung des StyleBasedCheckers	169
7.3	Zusammenfassung	171
8	Praktische Untersuchungen und Evaluationen	173
8.1	Einordnung der Forschungsergebnisse	173
8.1.1	Konzepte.....	174
8.1.2	Modelle.....	174
8.1.3	Methoden.....	176
8.1.4	Exemplare.....	176
8.2	Evaluationsprinzipien	176
8.3	Evaluation der Architekturstil-Konzeptualisierung	179
8.3.1	Evaluationskriterien.....	180
8.3.2	Vorgehen	180
8.3.3	Ergebnisse	182
8.4	Weitere Evaluationen.....	183
8.4.1	Annotation beispielhafter Softwaresysteme	183
8.4.2	Implementierung prototypischer Werkzeuge.....	185
8.4.3	Prüfungen auf Stiltreue	186
8.4.4	Interessante Zwischenergebnisse.....	188
8.5	Zusammenfassung	188
9	Resümee und Ausblick.....	191
9.1	Kernpunkte der Arbeit	192
9.1.1	Begriffsbildung und Konzeptualisierung	192
9.1.2	Der Ansatz der stilbasierten Architekturprüfung	193
9.1.3	Entwurfskriterien für die stilbasierte Architekturprüfung	194
9.1.4	Praktische Umsetzung und Evaluation	195
9.2	Einordnung und Abgrenzung.....	196
9.3	Ausblick.....	197
	Literaturverzeichnis.....	201

Abbildungsverzeichnis	213
Symbolliste	217
Anhang A: Grammatiken der prototypischen Umsetzung.....	221
A.1 Architekturstil-Beschreibung	221
A.2 Quelltext-Annotation.....	229
Anhang B: Regellisten und Typen des WAM- und Quasar-Stils.....	231
B.1 Quasar-Stil.....	231
B.2 WAM-Stil.....	234
Anhang C: Messergebnisse.....	237
C.1 Architekturelemente	237
C.2 Verstöße.....	243

1 Einleitung

1.1 Kontext

Der Quelltext heutiger Softwaresysteme ist so umfangreich, dass er sich nicht mehr im Ganzen überblicken und verstehen lässt. Selbst kleine Softwaresysteme umfassen mehrere tausend Zeilen, große Systeme können aus mehreren Millionen Zeilen und mehr bestehen. Sowohl die Anzahl als auch die Größe von Softwaresystemen hat seit dem ersten Computereinsatz im 20. Jahrhundert stetig zugenommen, und es ist davon auszugehen, dass zukünftige Systeme noch umfangreicher werden (Balzert 2001).

Für die Softwareentwicklerinnen und -entwickler, die diese Softwaresysteme erstellen und weiterentwickeln, hat sich jedoch eines nicht geändert: Wenn sie Softwaresysteme bearbeiten, sehen sie nur einen sehr kleinen Ausschnitt des Quelltextes, üblicherweise genau die eine Bildschirmseite, in der sie gerade Quelltextzeilen ändern, löschen oder hinzufügen. Prozentual betrachtet schrumpft dieser auf einmal wachsende Quelltextausschnitt kontinuierlich; auch mittlerweile übliche Ansätze, mit mehreren Monitoren das Sichtfeld zu vergrößern, ändern daran nur wenig.

Heutigen Entwicklerinnen und Entwicklern ist es somit kaum mehr möglich, Systeme zu verstehen, indem sie den gesamten Quelltext betrachten. Die Anzahl der Quelltextzeilen ist zu groß; und da Systeme nicht mehr einzeln, sondern zumeist in Teams entwickelt werden, ändert sich der Quelltext schneller als Entwicklerinnen und Entwickler mit dem Studium des Quelltextes Schritt halten können.

Aus diesem Grunde konzentrieren sich Entwicklerinnen und Entwickler bei jeder Programmieraufgabe auf den für die Aufgabe relevanten Quelltextausschnitt. Dafür entscheiden sie, welche Quelltextstellen geändert werden müssen. Um auszuschließen, dass ihre Änderungen ungewollte Seiteneffekte verursachen, dürfen sie sich nicht nur auf die zu ändernden Stellen beschränken, sondern müssen alle Teile des Quelltextes verstehen, die von den zu ändernden Stellen direkt oder indirekt abhängen, beispielsweise durch Operationsaufrufe oder Typpräferenzen.

Wird diese transitive Hülle abhängiger Quelltextstellen ebenfalls zu umfangreich, als dass man sie komplett lesen oder auch nur ermitteln könnte, müssen die Entwicklerinnen und Entwickler wiederum eine Auswahl treffen. Diese Auswahl lässt sich nicht mehr alleine anhand des Quelltextes treffen. Stattdessen nutzen die Entwicklerinnen und Entwickler ihr zusätzliches Wissen über die Grobstruktur des Softwaresystems: die *Softwarearchitektur* (Züllighoven und Raasch 2006; Bass et al. 2012; Reussner und Hasselbring 2009).

Definition 1-1: Softwarearchitektur

Der Begriff *Softwarearchitektur* bezeichnet die Grobstruktur eines Softwaresystems, bestehend aus *Architekturelementen*, deren *Schnittstellen* und *Beziehungen*.

Für diese Arbeit relevant ist die Grobstruktur des *Quelltextes* von Softwaresystemen. Hier gilt:

- Ein *Architekturelement* ist ein zusammengehöriger Ausschnitt des Quelltextes. Beispielsweise kann ein Architekturelement aus einer Gruppe von Klassen oder aus einem oder mehreren Subsystemen bestehen (Posch et al. 2011). Ältere Definitionen bezeichnen Architekturelemente auch als Komponenten (Shaw und Garlan 1996).

- Die *Schnittstelle* eines Architekturelements sind die außerhalb des Elements sichtbaren Eigenschaften, beispielsweise kann die Schnittstelle aus allen Schnittstellen der Klassen des Architekturelements bestehen.
- Architekturelemente stehen in einer *Beziehung*, wenn die zugehörigen Quelltextauschnitte in Beziehung stehen. Beispielsweise stehen die Architekturelemente A und B in einer Beziehung, wenn eine Klasse des Architekturelements A eine Klasse des Architekturelements B referenziert.

Eine gute Architektur¹ ist der Schlüssel, mit dessen Hilfe sich auch heutige Softwaresysteme ändern und weiterentwickeln lassen. Sie bietet eine abstrakte Sicht auf die vielen Tausend bzw. Millionen Quelltextzeilen heutiger Softwaresysteme. Die Architektur strukturiert Systeme in Architekturelemente, welche jeweils viele Zeilen Quelltext zu einer Einheit zusammenfassen. In einer guten, modularen Architektur sind den einzelnen Architekturelementen bestimmte Aufgaben zugeordnet (Parnas 1972). So können einige Architekturelemente für das Layout der Benutzungsschnittstelle zuständig sein, während andere Architekturelemente ausschließlich anwendungsfachliche Klassen enthalten.

Mit der Architektur haben Entwicklerinnen und Entwickler Wissen über ihr Softwaresystem, das über den reinen Quelltext hinaus geht. Denn selbst, wenn es möglich wäre, den gesamten Quelltext großer Systeme zu lesen, so wäre ihm die Architektur nicht oder nur unvollständig zu entnehmen (Clements und Shaw 2009).

Soll beispielsweise die Farbe eines Buttons in einer grafischen Benutzungsschnittstelle geändert werden, so könnten die Entwicklerinnen und Entwickler anhand der Softwarearchitektur wissen, welche Teile des Systems von der Benutzungsschnittstelle abhängen können und welche Teile grundsätzlich niemals in Abhängigkeit zur Benutzungsschnittstelle stehen. Beispielsweise kann gelten, dass anwendungsfachliche Klassen keine Abhängigkeit zur Benutzungsschnittstelle haben (Züllighoven 2005). Wenn die Entwicklerinnen und Entwickler nun in der Lage sind, alle anwendungsfachlichen Klassen zu identifizieren, so können sie bereits einen großen Anteil des Quelltextes außen vor lassen, da er für ihre Aufgabe irrelevant ist.

Softwarearchitekturen unterstützen Entwicklerinnen und Entwickler bei ihren Programmieraufgaben u.a. in folgender Hinsicht:

- *Die Architektur erleichtert die Auswahl des zu ändernden Quelltextes:*
Soll – wie in obigem Beispiel – die Farbe eines Buttons in der Benutzungsschnittstelle geändert werden, so lässt sich anhand der Architektur eine Vorauswahl treffen. Der zu ändernde Quelltext muss zu einem Architekturelement gehören, das für die Benutzungsschnittstelle zuständig ist.
- *Die Architektur hilft, Quelltextstellen zu identifizieren, die durch Seiteneffekte betroffen sein können:*
Wie in obigem Beispiel genannt, kann die Architektur so aufgebaut sein, dass Architekturelemente, die für anwendungsfachliche Aufgaben zuständig sind (wie die Berechnung einer Versicherungsprämie), niemals von solchen Architekturelementen abhängen, die für die Benutzungsschnittstelle zuständig sind. Wird also der Button aus obigem Beispiel geändert, so ist klar: alle Quelltextelemente, die zu anwendungsfachlichen Architekturelementen gehören, brauchen für diese Änderung nicht betrachtet zu werden.

¹ Im Folgenden verwendet diese Arbeit den Begriff der Architektur synonym zu dem Begriff der Softwarearchitektur. Wenn stattdessen die Architektur von Gebäuden gemeint ist, wird dies explizit gekennzeichnet.

Damit die Entwicklerinnen und Entwickler von ihrem Architekturwissen in dieser Weise profitieren können, müssen Entwicklungsteams ihre Architektur bewusst gestalten. Dafür definieren sie üblicherweise eine Menge von *Architekturvorgaben*. Die Vorgaben dienen als Anleitung für die Programmierung. Wenn Entwicklerinnen und Entwickler ihr System bearbeiten, müssen sie auf diese Architekturvorgaben achten.

Architekturvorgaben können sehr unterschiedliche Formen annehmen. Die vorliegende Arbeit betrachtet eine Form von Architekturvorgaben, die in jüngster Zeit große Bedeutung gewinnt: die *Architekturstile*.

Definition 1-2: Architekturstil (1. Fassung)²

Ein Architekturstil ist eine prinzipielle Lösungsstruktur, die für ein Softwaresystem durchgängig angewandt werden sollte (nach Riebisch 2009).

Definition 1-3: Stilbasierte Architektur (1. Fassung)

Wird die Architektur eines Softwaresystems nach einem Architekturstil entworfen, so handelt es sich um eine stilbasierte Architektur.

Softwaresysteme mit einer stilbasierten Architektur bezeichnet diese Arbeit als *stilbasierte Systeme*. Architekturen, die auf keinem Architekturstil beruhen, werden im Folgenden kurz als *stilfreie Architekturen* bezeichnet. Analog heißen Systeme mit stilfreier Architektur in dieser Arbeit *stilfreie Systeme*.

Stile beschreiben keine festgelegte Architektur, sondern legen Architekturvorgaben in Form von *Regeln* fest. Diese Regeln beschreiben, wie zugehörige stilbasierte Architekturen aufgebaut sein sollen. Die Regeln leiten sowohl den Entwurf stilbasierter Architekturen als auch deren Evolution (Garlan et al. 1994; Kruchten 1995). Architekturen evolvieren beispielsweise, wenn Entwicklungsteams aufgrund geänderter Anforderungen neue Architekturelemente hinzufügen.

Vorgaben in Form von Stilen können für mehrere Softwaresysteme wiederverwendet werden und ermöglichen dadurch, bewährte Strukturierungsprinzipien aus existierenden Softwaresystemen in neue Systeme zu übertragen. Entwicklungsteams, die mit einem Stil vertraut sind, können ihn nutzen, um neue Architekturen zu entwerfen (Shaw und Clements 2006) und um sich in bestehende Systeme dieses Stils leichter einzuarbeiten, beispielsweise um Wartungsaufgaben zu übernehmen (Clements et al. 2002).

Stile werden in der Praxis sogar eingesetzt, ohne dass sich Entwicklungsteams darüber bewusst sind. Dies geschieht, wenn Teams ihre Architektur konsistent strukturieren möchten und dafür eigene Regeln definieren, ohne mit dem Konzept von Stilen vertraut zu sein. So entstehen sogenannte *systemspezifische Stile* (Lilienthal 2008). Manchmal finden Entwicklungsteams, die mit dem Konzept von Stilen vertraut sind, keinen passenden Stil für ihr System und entscheiden sich bewusst, einen systemspezifischen Stil zu entwerfen (Lilienthal 2008).

Systemspezifische Stile werden manchmal nur für ein einziges Softwaresystem verwendet. Untersuchungen haben gezeigt, dass stilbasierte Systeme, selbst wenn der zugehörige Stil nur einmal verwendet wird, immer noch vielfältige Vorteile bieten: sie sind beispielsweise verständlicher und wartbarer als stilfreie Systeme (Lilienthal 2008).

Architekturstile sind der zentrale Ansatzpunkt, um die Komplexität umfangreicher Softwaresysteme und Architekturen langfristig in den Griff zu bekommen (Lilienthal 2008).

² Einige in dieser Einleitung vorgestellte Definitionen sind mit „1. Fassung“ markiert. Diese Definitionen werden in späteren Kapiteln der Arbeit detailliert und ergänzt.

1.2 Problemstellung

Die Lebensdauer von Softwaresystemen beträgt häufig viele Jahre oder Jahrzehnte. Damit Entwicklungsteams dauerhaft von ihrem Wissen über die Architekturen profitieren können, müssen die Teams während der gesamten Lebensdauer von Systemen dafür sorgen, dass die Architektur der Systeme den Vorgaben entspricht. Dabei sind Unternehmen in der Praxis mit einem komplexen Umfeld konfrontiert:

- Systeme werden im Laufe ihrer Existenz üblicherweise mehrfach verändert und erweitert. Manche Änderungen und Erweiterungen von Softwaresystemen erfordern, dass die Architektur ebenfalls überarbeitet werden muss. So passt sich die Architektur von Softwaresystemen im Laufe ihrer Lebenszeit wiederholt an neue Anforderungen an: die Architektur evolviert. Dabei ist es nicht möglich, alle Änderungen der Architektur bereits von Beginn an zu planen, da manche Anforderungen nicht vorhersehbar sind. Ducasse et al. bringen es auf den Punkt: „successful systems are doomed to continually evolve and grow“ (Ducasse und Pollet 2009, S.573).
- Softwaresysteme werden in der Regel von unterschiedlichen Teams entwickelt und bearbeitet. Viele Unternehmen unterscheiden beispielsweise zwischen Entwicklungs- und Wartungsteams. Auch innerhalb von Teams wechseln die Personen: alte Teammitglieder scheiden aus, neue kommen hinzu und müssen sich in die Systeme, ihre Architektur und die Architekturvorgaben einarbeiten.

In der Praxis ist es schwierig, die Architektur von Softwaresystemen über die gesamte Lebensdauer korrekt zu halten. Empirische Studien zeigen, dass Architekturen im Laufe der Zeit zunehmend gegen ihre Vorgaben verstoßen (Perry und Wolf 1992; Rosik et al. 2008; Feilkas et al. 2009; Eick et al. 2001). Die Probleme werden umso größer, je länger Systeme eingesetzt und weiterentwickelt werden. Dieses Phänomen bezeichnet man als Architekturerosion (Jahnke 2009; Silva und Balasubramaniam 2012). Die Erosion entsteht ungewollt und unbemerkt.

Definition 1-4: Architekturerosion

Mit Architekturerosion bezeichnet man den Sachverhalt, dass die Architektur von Softwaresystemen zunehmend gegen ihre Vorgaben verstößt.

Die Architekturerosion hat weitreichende Folgen: Entwicklerinnen und Entwickler können ihr Architekturwissen nicht mehr nutzen, um korrekte Annahmen über den Quelltext zu treffen. Stattdessen werden die Entwicklerinnen und Entwickler darauf zurückgeworfen, Vermutungen über die Architektur anzustellen und so viele Anteile des Quelltextes zu lesen, wie aufgrund der Systemgröße möglich. Dies führt zu einer Abwärtsspirale: falsche Vermutungen der Entwicklerinnen und Entwickler führen zu neuen Fehlern in der Softwarearchitektur. So erodiert die Architektur betroffener Softwaresysteme zunehmend. Selbst wenn Systeme zu Beginn über eine gute, modular aufgebaute Architektur verfügen, so verliert diese nach und nach an Qualität. Seiteneffekte können bald immer mehr Systemteile betreffen. In erodierten Systemen bedeuten selbst kleinste Änderungen, dass umfangreiche Anteile des Quelltextes gelesen und verstanden werden müssen. In extremen Fällen werden Systeme unwartbar, da die zu lesende Quelltextmenge sich nicht mehr bewältigen lässt. Änderungen sind wirtschaftlich nicht mehr sinnvoll, da sie zu lange dauern und zu leicht zu neuen Fehlern durch unvorhergesehene Seiteneffekte führen. Letzten Endes bleibt nur, diese Systeme durch Neuentwicklungen abzulösen. Solche durch Erosion kaum mehr wartbaren Systeme werden als Altsysteme bezeichnet (englisch: Legacy

Software). Altsysteme sind ein weitverbreitetes Problem der heutigen Praxis (Sommerville 2001).

Alleine durch besondere Sorgfalt lässt sich die Abwärtsspirale erodierender Systeme kaum verhindern, sondern lediglich verlangsamen. Auch ist aufgrund von Termin- und Kostendruck solch eine Sorgfalt in der Praxis schwierig durchzuhalten (Jahnke 2009), insbesondere, da sich die Softwarearchitektur mit heutigen Programmiersprachen nur unvollständig abbilden lässt (Broy und Reussner 2010). Selbst manuelle Reviews, in denen die Architekturdokumentation mit dem tatsächlichen Quelltext verglichen wird, verlangsamen den Prozess der Architekturerosion oft lediglich, da sie nicht alle Fehler aufdecken. Aufgrund der enormen Quelltextmenge heutiger Systeme sind Reviews zudem zeit- und damit kostenintensiv und lassen sich in der Praxis nur in begrenztem Umfang durchführen.

1.3 Die untersuchten Fragestellungen

Das Anliegen dieser Arbeit ist es, Entwicklungsteams dabei zu unterstützen, der Architekturerosion entgegen zu wirken, damit ihre Systeme langfristig verständlich und wartbar bleiben.

Die Arbeit fokussiert auf Architekturvorgaben in Form von Architekturstilen. Architekturstile sind im Kontext der hier behandelten Problemstellung besonders interessant, da sie gerade die Bedürfnisse aktueller Softwaresysteme unterstützen: Stile eignen sich gut für langlebige Systeme, da sie nicht nur den Entwurf, sondern auch die Weiterentwicklung von Architekturen unterstützen. Und Stile eignen sich gut für große Systeme, denn ihre Größe ist unabhängig von der Systemgröße.

Kernfrage der Arbeit:

Wie lässt sich der Architekturerosion stilbasierter Systeme entgegenwirken?

Diese Kernfrage wird in mehreren Forschungsfragen konkretisiert. Da Architekturerosion entsteht, indem sich unbemerkte Verstöße gegen Architekturvorgaben ansammeln, schließt sich die folgende Frage an:

Frage 1:

Wie lassen sich Verstöße gegen Architekturstile in Softwaresystemen aufdecken?

Wenn Entwicklungsteams über die Information verfügen, wo ihr Softwaresystem gegen Architekturvorgaben verstößt, können sie das System entsprechend korrigieren und so die Erosion beseitigen. Dabei genügt es nicht zu wissen, welche Teile der Architektur gegen den Stil verstoßen; zusätzlich muss bekannt sein, welche Quelltextstellen die Verstöße verursachen. Diese Quelltextstellen sind es, die korrigiert werden müssen:

Frage 2:

Wie lässt sich ermitteln, welche Quelltextstellen für Verstöße gegen Architekturstile verantwortlich sind?

Da sich die Verstöße auf manuellem Wege nur unzureichend ermitteln lassen, aufgrund der großen Menge an Quelltextzeilen und aufgrund regelmäßiger Quelltextänderungen, will diese Arbeit erkunden, wie sich die Suche nach Verstößen automatisieren lässt. Dafür verfolgt sie folgende Frage:

Frage 3:

Inwieweit lässt sich die Suche nach Verstößen gegen Architekturstile automatisieren, welche manuellen Anteile sind notwendig?

Um Verstöße zu finden, muss die aktuelle Architektur eines Softwaresystems mit dem zugehörigen Architekturstil verglichen werden. Hieraus ergibt sich eine weitere Frage:

Frage 4:

Wie lassen sich Architekturen stilbasierter Systeme und Architekturstile dokumentieren, um eine werkzeuggestützte Suche nach Verstößen gegen Stile zu unterstützen?

Da Architekturstile ein neueres Phänomen innerhalb der Software-Entwicklung darstellen, ist der Begriff noch nicht so etabliert wie beispielsweise der Begriff der Softwarearchitektur. Deshalb soll der Begriff des Architekturstils im Rahmen dieser Arbeit anhand der wissenschaftlichen Literatur und anhand praxisrelevanter Stile so weit konkretisiert werden, dass sich eine prinzipielle Struktur für Stildokumentationen empfehlen lässt. Aus denselben Gründen soll untersucht werden, wie sich die prinzipielle Struktur stilbasierter Architekturen von anderen Architekturen unterscheidet. So werden im Rahmen dieser Arbeit Metamodelle für Architekturstile und für stilbasierte Architekturen entwickelt. Nach diesen Metamodellen lassen sich Dokumentationen strukturieren. Daraus folgt eine ergänzende Frage:

Frage 4.1:

Welche prinzipielle Struktur besitzen Architekturstile und stilbasierte Architekturen?

Die Architektur von Softwaresystemen lässt sich dem Quelltext nicht vollständig entnehmen, sie muss zusätzlich dokumentiert werden. Daraus resultiert die folgende Ergänzung der Frage 4:

Frage 4.2:

Welche Anteile der Architektur stilbasierter Systeme lassen sich direkt aus dem Quelltext entnehmen, welche müssen zusätzlich dokumentiert werden?

Wenn neue Anforderungen dazu führen, dass die Architektur eines bestehenden Softwaresystems weiterentwickelt wird, drohen Architekturdokumentationen zu veralten. Dies führt zu folgender Anschlussfrage:

Frage 4.3:

Wie lässt sich unterstützen, dass die Dokumentation stilbasierter Architekturen bei Architekturänderungen aktuell bleibt?

1.4 Zielsetzung

Diese Arbeit bearbeitet die genannten Fragestellungen mit dem Ziel, einen Prüfansatz zu entwickeln, der sowohl theoretisch fundiert als auch praxistauglich ist. Der Ansatz soll für reale heutige Architekturstile und Softwaresysteme einsetzbar sein.

Da diese Arbeit darauf zielt, Praktikerinnen und Praktiker in ihrer täglichen Arbeit zu unterstützen, wird im Rahmen der Arbeit nicht nur ein Prüfansatz konzipiert, sondern auch eine Umsetzung dieses Ansatzes entwickelt. Diese Umsetzung konkretisiert den Ansatz für eine aktuelle Programmiersprache und einen praxisrelevanten Architekturstil. Die Umsetzung wird durch ein prototypisches Werkzeug ergänzt, das dazu dient, die Machbarkeit des Ansatzes zu zeigen, indem reale Softwaresysteme mit diesem Werkzeug auf Stiltreue geprüft werden.

Die Umsetzung lässt sich als Vorlage für weitere Umsetzungen mit anderen Sprachen, Stilen und Werkzeugen nutzen.

Damit der Ansatz breit einsetzbar ist, soll er keine Annahmen über die verwendete Entwicklungsmethodik und damit über die Arbeitsteilung in Projekten enthalten. Der Ansatz soll eine werkzeuggestützte Prüfung des Quelltextes ermöglichen, unabhängig davon, wer die Prüfung durchführt: Entwicklerinnen und Entwickler sollen ihr System direkt während der Programmierung prüfen können, Architektinnen und Architekten sollen die Prüfung im Rahmen von Architekturreviews durchführen können. Auch eine automatisierte Prüfung soll möglich sein, beispielsweise wenn neuer Quelltext in ein Repository übertragen wird oder bei nächtlichen Testläufen.

Da sich die Architekturen interaktiver Anwendungssysteme deutlich von den Architekturen anderer Systemarten, wie beispielsweise eingebetteter Systeme, unterscheiden, ist es nicht sinnvoll, einen Ansatz zu entwickeln, der alle Arten von Softwaresystemen abdecken soll. Die vorliegende Arbeit fokussiert auf interaktive Anwendungssysteme, da Architekturstile in diesen Systemen ihre größte Verbreitung finden.

In modernen, interaktiven Anwendungssystemen werden üblicherweise objektorientierte Programmiersprachen eingesetzt. Der in dieser Arbeit entwickelte Prüfansatz soll deshalb für objektorientierte Sprachen einsetzbar sein.

1.5 Einordnung der Arbeit

Die vorliegende Arbeit ordnet sich ein in das Feld der *Prüfungen auf Architekturtreue*, im englischen Sprachraum bezeichnet als *architecture conformance checking* oder *architecture compliance checking* (Passos et al. 2010; Knodel und Popescu 2007). Diese Arbeit spricht auch kurz von *Architekturprüfungen*.

Architekturprüfungen dienen dazu, der Architekturerosion entgegen zu wirken. Hierfür ermitteln sie, ob die Architektur von Softwaresystemen gegen Vorgaben verstößt. Die Prüfungen untersuchen den Quelltext der Systeme, da es ihnen um die Treue der tatsächlichen Architektur der Softwaresysteme geht, und nicht – wie in anderen Forschungsbereichen – um die Treue von Architekturbeschreibungen.

Der Begriff der Softwarearchitektur bezeichnet die Grobstruktur von Softwaresystemen aus vielen Perspektiven. Als der Begriff geprägt wurde, verstand man unter einer Softwarearchitektur vorrangig die Grobstruktur des *Quelltextes*. Mittlerweile wird Softwarearchitektur breiter gefasst, sie kann auch andere Strukturen als die Struktur des Quelltextes umfassen, beispielsweise die Aufteilung von Softwaresystemen zur Laufzeit in Prozesse, welche auf verschiedenen Hardwaresystemen ausgeführt werden (Bass et al. 2012; Hofmeister et al. 2000). Heute

bezeichnet man die Grobstruktur des Quelltextes auch als statische Struktur, um sie von anderen Anteilen der Architektur abzugrenzen.

Bei Forschungen zur Architekturerosion spielt nach wie vor die statische Struktur eine zentrale Rolle. Um diesen Punkt hervorzuheben wird in der Literatur auch der Begriff der *statischen* Prüfungen auf Architekturtreue verwendet.

Ansätze und Werkzeuge, mit denen sich Systeme auf ihre Architekturtreue prüfen lassen, unterscheiden sich bezüglich verschiedener Aspekte, beispielsweise in den Annahmen, die sie über die Aufgabenteilung in Entwicklungsprojekten treffen, in der Art der zu prüfenden Vorgaben, bezüglich ihres Verständnisses von Softwarearchitektur sowie darin, ob sie die von Verstößen betroffenen Quelltextstellen aufzeigen.

Bisher fehlt es an Ansätzen, mit denen sich die Architekturtreue zu Vorgaben in Form von Stilen prüfen lässt. Hier liegt die zentrale Innovation der vorliegenden Arbeit: Mit der stilbasierten Architekturprüfung sind solche Prüfungen nun möglich. Die stilbasierte Architekturprüfung ist ein Ansatz zur *statischen Prüfung auf Stiltreue*.

1.6 Methodik

Der in dieser Arbeit vorgestellte Ansatz wurde nach der Methodik der Design-Forschung entwickelt (March und Smith 1995; Hevner et al. 2004; Vaishnavi und Kuechler 2004). Diese Methodik ist geeignet für Forschungsvorhaben, die sowohl einen analytischen als auch konstruktiven Charakter besitzen. March et al. grenzen die Naturwissenschaften von den Design-Wissenschaften folgendermaßen ab:

Whereas natural science tries to understand reality, design science attempts to create things that serve human purposes (March und Smith 1995, S. 253).

Design-Forschungsprojekte gehen iterativ-inkrementell vor, sie entwickeln und evaluieren ihre Ergebnisse anhand beispielhafter Designobjekte. Im Falle dieser Arbeit handelt es sich bei den Designobjekten um Architekturstile, Softwaresysteme und prototypische Prüfwerkzeuge.

Im Rahmen dieser Arbeit wurden drei Forschungszyklen durchgeführt: In den ersten beiden Zyklen wurde der Ansatz der stilbasierten Architekturprüfung schrittweise entwickelt und verschiedene Prototypen erstellt. Der dritte Forschungszyklus diente dazu, die Machbarkeit des Ansatzes zu zeigen.

Zwischenergebnisse der Forschung wurden in mehreren Artikeln veröffentlicht (Becker-Pechau 2009a, 2009b; Becker-Pechau und Bennicke 2007; Becker-Pechau et al. 2006). Die Forschungstätigkeiten wurden von Studierenden unterstützt, die im Rahmen der Forschungen ihre Zwischen- und Abschlussarbeiten erstellten: Arne Scharping, Felix Abraham, Bettina Karstens, Johanna und Erdal Özkan (Scharping 2008; Abraham 2006; Karstens 2005; Özkan und Özkan 2004). Die Arbeiten von Scharping, Karstens und Özkan wurden von der Autorin direkt betreut. Die Arbeit von Abraham entstand im Rahmen eines Diplomarbeitsprojekts, das von der Autorin initiiert und geleitet wurde. Dieses Projekt widmete sich dem Thema Softwarearchitektur. Im Rahmen des Projektes betreuten mehrere wissenschaftliche Mitarbeiterinnen und Mitarbeiter die Arbeiten der Studierenden. Die Arbeiten von Scharping, Abraham und Karstens trugen direkt zum Thema dieser Arbeit bei, die Arbeit von Özkan und Özkan diente als Vorarbeit. Weitere Vorarbeiten der Autorin zur vorliegenden Arbeit wurden auf der Konferenz „Information Systems Research Seminar in Scandinavia“ sowie auf zwei von der Autorin veranstalteten Workshops im Rahmen von OOPSLA-Konferenzen präsentiert und diskutiert (Becker-Pechau und Pechau 2003b, 2003a; Becker-Pechau et al. 2004).

1.7 Aufbau der Arbeit

Kapitel 2 liefert einen detaillierten Blick auf den Begriff der Softwarearchitektur, verfeinert die in der Einleitung gegebene Definition und thematisiert den Zusammenhang von Softwarearchitektur und Quelltext.

Anschließend widmet sich Kapitel 3 dem Konzept des Architekturstils. Das Kapitel ergänzt eine Definition für den inneren Aufbau von Architekturstilen, zeigt praxisrelevante Beispiele und erörtert die Bedeutung von Stilen in der Software-Entwicklung.

Aufbauend auf den in Kapitel 2 und 3 vorgestellten Konzepten gibt Kapitel 4 einen Überblick über die bereits bestehenden Ansätze zur Prüfung auf Architekturtreue. Der überwiegende Teil dieser Ansätze verwendet systemspezifische Architekturvorgaben, jedoch keine Architekturstile. Das Kapitel zeigt auf, inwiefern die bisherigen Ansätze zu kurz greifen, um die Architektur von Softwaresystemen auf ihre Treue zu *Architekturstilen* zu prüfen.

Die Kapitel 5 und 6 präsentieren das Konzept der stilbasierten Architekturprüfung. Kapitel 5 erläutert den prinzipiellen Ablauf einer Prüfung und präsentiert in den Abschnitten 5.3 bis 5.5 das Kernkonzept des Ansatzes. Als Beispiele dienen dabei die Architekturstile des vorherigen Kapitels. In Kapitel 6 werden drei Ausbaustufen vorgestellt, die das Kernkonzept ergänzen.

Kapitel 7 beschreibt, wie sich der Ansatz der stilbasierten Architekturprüfung praktisch umsetzen lässt. Die vorgestellte Umsetzung basiert auf der objektorientierten Programmiersprache Java. Das Kapitel stellt den *StyleBasedChecker* vor, ein im Rahmen dieser Arbeit erstelltes, prototypisches Prüfwerkzeug, mit dem sich Java-Systeme auf ihre Treue zu einem gegebenen Architekturstil prüfen lassen.

Kapitel 8 präsentiert die praktischen Untersuchungen, in denen mehrere reale Softwaresysteme geprüft wurden. Das Kapitel beschreibt das Vorgehen bei den Untersuchungen, ordnet diese in die gewählte Methodik der Design-Forschung ein und erläutert die Untersuchungsergebnisse.

Das abschließende Kapitel fasst die Arbeit zusammen, beurteilt kritisch das Erreichte und gibt einen Ausblick auf weitere Forschungstätigkeiten.

Anhang A enthält die Grammatiken, mit denen die beispielhafte Umsetzung Architekturstile und stilbasierte Architekturen beschreibt. Anhang B listet die Details der in dieser Arbeit beispielhaft betrachteten Architekturstile. Anhang C enthält die Messergebnisse der praktischen Untersuchungen.

2 Softwarearchitektur

Dieses Kapitel diskutiert und definiert die für diese Arbeit relevanten Begriffe im Bereich von Softwarearchitekturen. Zu Beginn dieses Kapitels wird die Herkunft und Intention des Begriffs der Softwarearchitektur diskutiert. Anschließend stellt Abschnitt 2.2 die in der Einleitung gegebene Definition von Softwarearchitektur in den Kontext anderer Definitionen. Abschnitt 2.3 thematisiert den Zusammenhang von Softwarearchitekturen zum Quelltext, verfeinert die in der Einleitung gegebene Definition und ergänzt eine formale Definition von Softwarearchitektur. Anschließend definiert Abschnitt 2.4 zwei Arten von Softwarearchitektur: die Ist- und die Soll-Architektur. Diese spielen eine zentrale Rolle bei Prüfungen auf Architekturtreue.

Basierend auf den bisher betrachteten Begriffen und Konzepten zeigt Abschnitt 2.5 die Grenzen von Architekturen auf, die lediglich auf einer Soll-Architektur basieren, und macht so den Bedarf an den in Kapitel 3 behandelten, stilbasierten Architekturen deutlich.

2.1 Warum Softwarearchitektur?

Der Begriff der Softwarearchitektur ist aus dem Bedürfnis heraus entstanden, auch große Softwaresysteme, die auf Quelltextebene nicht mehr handhabbar sind, im Ganzen *planen* und *verstehen* zu können, und somit langfristig wartbar zu halten (Hofmeister et al. 2000).

Während heutzutage sowohl das Planen als auch das Verstehen großer Systeme dem Forschungsbereich der Softwarearchitektur zugeordnet sind, wurde der Aspekt des *Verstehens* zu Beginn unter anderen Bezeichnungen diskutiert: Parnas beschäftigte sich mit der Modularisierung von Softwaresystemen (Parnas 1972). Module verbergen nach Parnas ihre Implementierungsdetails und kommunizieren über explizite Schnittstellen. So können Systeme auf Ebene der Module verstanden werden, ohne die Details der Modulimplementierung zu kennen. Nagl verwendete den Begriff „Programmieren im Großen“, ebenfalls mit der Zielrichtung, von Details der Implementierung zu abstrahieren und die Gesamtstruktur von Softwaresystemen in den Fokus zu rücken (Nagl 1990).

Die Bezeichnung Softwarearchitektur verbreitete sich maßgeblich durch Veröffentlichungen von Garlan und Shaw (Garlan und Shaw 1993; Shaw und Garlan 1996). Diese fokussierten auf die Frage, wie sich die Struktur von Softwaresystemen – die Architektur – *planen* lässt. Sie argumentierten, dass bei großen Softwaresystemen nicht mehr vorrangig die gewählten Algorithmen betrachtet werden müssen, sondern die Gesamtstruktur. Diese sei unabhängig von den funktionalen Anforderungen und müsse nach anderen Kriterien entworfen werden.

Beide Seiten von Architektur – Planen und Verstehen großer Systeme – beinhalten, dass man von Details abstrahiert. Architekturen umfassen daher nur die für das Gesamtverständnis relevanten Aspekte von Softwaresystemen (Bass et al. 2012).

Es stellt sich die Frage: Welche Aspekte eines Softwaresystems sollten Teil der Architektur sein, welche Details können weggelassen werden? Die Antwort hängt davon ab, zu welchem Zweck man die Architektur betrachtet. Verschiedene Betroffene (im Englischen Stakeholder, im Deutschen auch als Interessengruppen bezeichnet) schauen aus unterschiedlichen Blickwinkeln auf die Software (Jacobson et al. 1999): eine Softwareentwicklerin blickt anders auf Softwaresysteme als ein Systemadministrator, ein Anwender oder eine Projektleiterin. So entstand das Konzept der *Sichten* (Views) (Kruchten 1995; Behrens et al. 2009; ISO/IEC/IEEE-Standard-42010 2011). Eine Sicht *dokumentiert* einen Teil einer Softwarearchitektur. Jede Sicht beschreibt die Architektur aus einem anderen Blickwinkel. Die Blickwinkel werden üblicherweise als *Standpunkte* (Viewpoints) bezeichnet. Ein Standpunkt verkörpert miteinander verwandte

Anliegen (Concerns), die sich aus den Interessen der Betroffenen ergeben. Er legt Konventionen für die Konstruktion, Interpretation und Verwendung zugehöriger Sichten fest (Behrens et al. 2009; ISO/IEC/IEEE-Standard-42010 2011). Während eine Sicht immer eine konkrete Architektur beschreibt, sind Standpunkte allgemeine Vorgaben, die für die Dokumentation verschiedener Architekturen genutzt werden. Abhängig vom jeweiligen Standpunkt kann die Architektur desselben Systems sehr unterschiedlich aussehen. Zu Beginn der Diskussion über Sichten vertraten einige Autoren die Auffassung, dass ein Softwaresystem mehrere Architekturen habe (Soni et al. 1995); mittlerweile hat sich das Verständnis etabliert, dass Softwaresysteme jeweils nur eine Architektur haben, diese jedoch von verschiedenen Standpunkten betrachtet werden kann (Hofmeister et al. 2000; Reussner und Hasselbring 2009; ISO/IEC/IEEE-Standard-42010 2011).

In dieser Arbeit steht die Sicht auf den Quelltext im Mittelpunkt. Für diese Sicht existiert keine allgemein akzeptierte Bezeichnung, verbreitet sind die Begriffe Modulsicht (Clements et al. 2002; Hofmeister et al. 2000) und Entwicklungssicht (Kruchten 1995). Allgemeiner spricht man von der statischen Sicht und dem statischen Standpunkt (Behrens et al. 2009). So hält es auch diese Arbeit.

Eine zweite Debatte entwickelte sich zu der folgenden Frage: *Hat* ein Softwaresystem eine Architektur, oder ist die Architektur nur ein bestimmter Blick auf das System? In frühen Veröffentlichungen finden sich Formulierungen wie „die Architektur eines Softwaresystems *definiert* das System in den Begriffen [...]“ (Shaw und Garlan 1996, S. 3) oder „Softwarearchitekturen *beschreiben*, wie Systeme in [...] zerlegt werden [...]“ (Soni et al. 1995, S. 196). Hier wird die Frage, ob die Architektur *Teil* des Systems ist, noch nicht explizit diskutiert. Mittlerweile sprechen viele Autoren davon, dass Softwaresysteme eine Architektur *haben*, dass die Architektur also Teil des Systems sei und nicht erst durch den Blick des Betrachters entstehe (Bass et al. 2012; Reussner und Hasselbring 2009; ISO/IEC/IEEE-Standard-42010 2011).

Nun stellt sich die Frage: hat *jedes* System eine Architektur, selbst wenn das Entwicklungsteam sich darüber gar keine Gedanken gemacht hat? Hierzu besteht mittlerweile ein breiter Konsens: selbst Systeme, für die nicht bewusst eine Architektur entworfen wurde, haben eine Architektur (Züllighoven, Raasch 2006, S. 780), meist ist sie schlecht und schwer verständlich: Solche Architekturen sind erfahrungsgemäß nicht gut modularisiert und befinden sich auf der Abstraktionsebene der Programmiersprache. Abhängig davon, welche Ausdrucksmittel die Programmiersprache enthält, bestehen diese Architekturen beispielsweise aus allen Klassen oder allen Prozeduren des Systems und deren Beziehungen. Diese Architekturen bieten nur eine geringe Abstraktion von den Details des Quelltextes. Aufgrund ihrer schlechten Qualität und der geringen Abstraktion sind diese Architekturen nicht geeignet, Systeme planbar und verständlich zu machen. In Projekten hingegen, die explizit eine gute Architektur planen und pflegen, existiert nicht nur diese programmiersprachliche Ebene der Architektur, hier kommen höhere Ebenen mit stärkeren Abstraktionen hinzu. Beispielsweise heben solche stärkeren Abstraktionen ausgewählte Klassen als Stützpfiler hervor (Poppendieck und Poppendieck 2010) oder fassen mehrere Klassen zu jeweils einem Architekturelement zusammen (Bass et al. 2012). Architekturelemente, die mehrere Klassen enthalten, bezeichnet Lilienthal als Subsysteme. Lilienthal unterscheidet zwei Architekturebenen bei objektorientierten Systemen: die Klassen- und die Subsystemebene der Architektur. Auf der Klassenebene bestehen Architekturelemente aus einzelnen Klassen, auf der Subsystemebene enthält ein Architekturelement mehrere Klassen (Lilienthal 2008, S. 24ff). Die Klassenebene enthält abhängig von der jeweiligen Programmiersprache weitere Typen neben den Klassen, so kommen in Java beispielsweise unter anderem Interfaces hinzu³. Jedes objektorientierte Softwaresystem hat eine Architektur auf Klassenebene, diese Architektur besteht

³ Der Lesbarkeit halber steht im Folgenden der Begriff „Klasse“ stellvertretend für weitere Typen, die in objektorientierten Sprachen vorkommen, wie beispielsweise Java-Interfaces oder Enumerations.

aus allen Klassen und allen Beziehungen dieser Klassen. Zusätzlich können Systeme eine höhere Architekturebene beinhalten, indem sie auf Klassenebene einzelne Klassen hervorheben und / oder auf Subsystemebene mehrere Klassen zu Architekturelementen zusammenfassen. Diese höheren Ebenen müssen explizit realisiert werden, sie entstehen nicht von alleine.

Jedes Softwaresystem hat also eine Architektur, zumindest auf Klassenebene. Allerdings lassen sich nicht alle Aspekte von Architekturen eindeutig aus Softwaresystemen entnehmen (Züllig-hoven und Raasch 2006, S. 780; Reussner und Hasselbring 2009, S. 179; Ducasse und Pollet 2009). Was lässt sich entnehmen, was nicht? Es gibt keine generelle Aussage dazu, welche Aspekte von Architekturen eindeutig im System erkennbar sind und welche nicht, da die Antwort einerseits von der gewählten Sicht abhängt, andererseits davon, wie das jeweilige System umgesetzt ist und auf welcher Ebene sich die Architektur befindet. Beispielsweise können sprechende Namen im Quelltext dazu führen, dass anhand der Bezeichner mehr Teile der Architektur aus dem System entnommen werden können, als bei Systemen mit schlecht gewählten Bezeichnern. Anhand von Bezeichnern kann unter anderem deutlich gemacht werden, aus welchen Klassen ein Subsystem besteht.

Abbildung 1 illustriert diese Tatsache anhand eines Java-Systems, wie es in der Praxis zu finden ist. Java erlaubt, Klassen in Packages zusammenzufassen. Packages können neben Klassen auch weitere Packages enthalten. Links in der Abbildung ist ein Ausschnitt einer Baumstruktur von Packages dargestellt. Bei zwei Packages ist beispielhaft eine enthaltene Klasse zu sehen. Rechts in der Abbildung ist die Subsystemebene der Architektur dargestellt, sie besteht aus drei Subsystemen, hier als Schichten bezeichnet.

In diesem Beispiel sind die Bezeichner von drei Java-Packages so gewählt, dass die Zuordnung zu den Subsystemen der Architektur deutlich wird. Beispielsweise gilt für das Package namens *praesentation*, dass alle Klassen, die direkt oder indirekt in diesem Package enthalten sind, zum Subsystem Präsentationsschicht gehören⁴. Das bedeutet, dass die Klasse A und die Klasse B zum Architekturelement Präsentationsschicht gehören.

Im Rahmen dieser Arbeit ist eine wichtige Frage, inwieweit sich die Architektur aus dem zu prüfenden System entnehmen lässt und welche zusätzlichen Informationen benötigt werden. Die Kapitel 5 und 6 diskutieren und beantworten diese Frage. Dies geschieht in Hinblick auf stilbasierte Architekturen und auf das Thema dieser Arbeit: die Prüfung von Softwaresystemen auf Stiltreue.

⁴ Das Wurzel-Package „de“ enthält in der Regel keinen Quelltext, so dass für dieses Package keine Zuordnung benötigt wird.

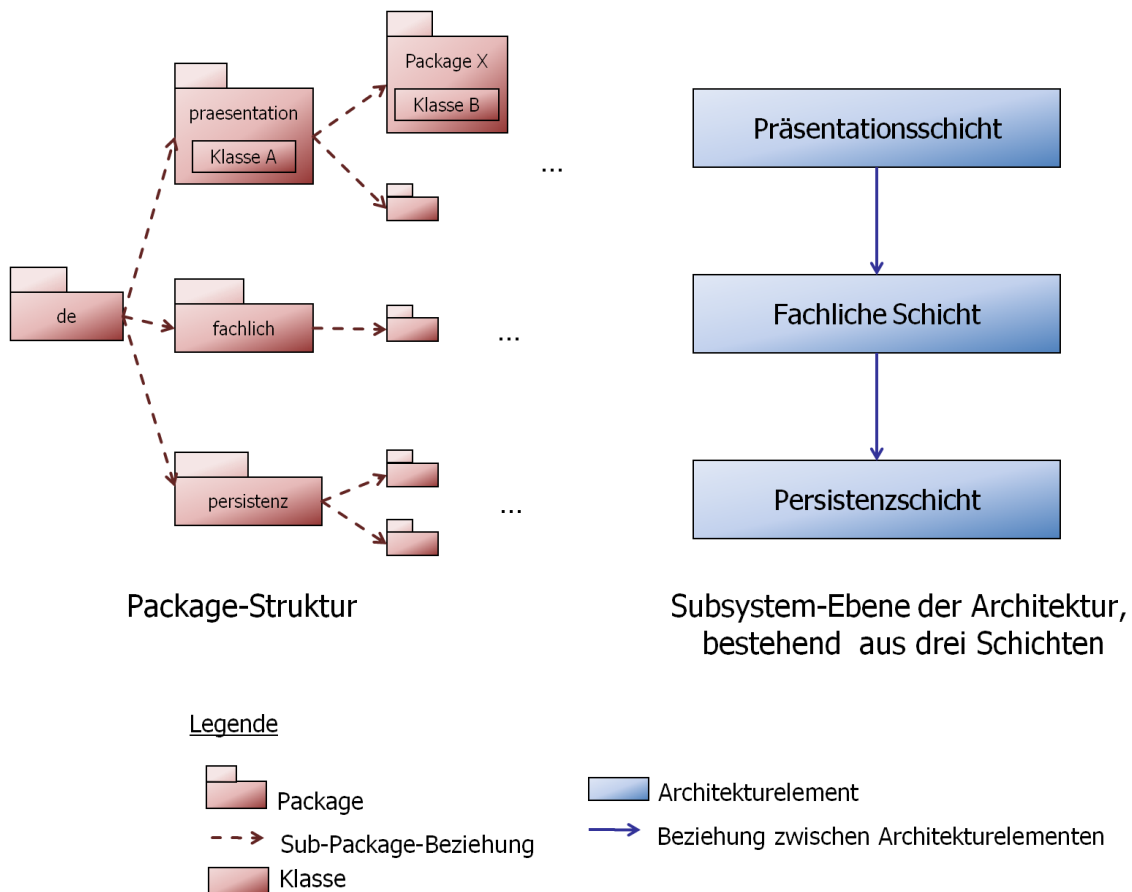


Abbildung 1:
Beispiel: Quelltextbezeichner helfen, die Architektur im Quelltext zu erkennen

2.2 Diskussion verschiedener Softwarearchitektur-Definitionen

Dieser Abschnitt diskutiert verschiedene in der Literatur zu findende Definitionen des Architekturbegriffs und erläutert in diesem Kontext die in der Einleitung gegebene Architekturdefinition. Zur Erinnerung sei an dieser Stelle die Definition aus der Einleitung wiederholt:

Definition 1-1: Softwarearchitektur

Der Begriff *Softwarearchitektur* bezeichnet die Grobstruktur eines Softwaresystems, bestehend aus *Architekturelementen*, deren *Schnittstellen* und *Beziehungen*.

Was bedeutet das konkret, woraus bestehen Architekturen? Betrachtet man den für diese Arbeit relevanten *Quelltext* eines Softwaresystems, so sind die Bestandteile der Architektur beispielsweise Klassen oder Subsysteme. Diese sind verbunden über verschiedene Arten von Beziehungen. Beispielsweise können Klassen die Operationen anderer Klassen aufrufen oder voneinander erben. Betrachtet man hingegen dasselbe Softwaresystem zur *Laufzeit*, so sieht man Prozesse, die auf Hardwarekomponenten verteilt sind. Diese können beispielsweise über verteilte Methodenaufrufe oder über Web-Services kommunizieren.

Betrachtet man den Quelltext oder ein Softwaresystem zur Laufzeit sieht man Ausschnitte der Architektur. Hier wird deutlich: Je nachdem, aus welchem Blickwinkel man schaut, können ganz unterschiedliche Teile in einer Architektur enthalten sein. Von diesen Unterschieden muss eine Definition des Begriffs der Softwarearchitektur abstrahieren. Die verschiedenen Architektursichten haben gemein, dass sie aus Architekturelementen und Beziehungen bestehen. Architekturelemente und Beziehungen finden sich als Kern in den vielfältigen Architekturdefinitionen der Literatur wieder.

Einige Definitionen bezeichnen die Architekturelemente auch als Komponenten (Garlan und Shaw 1994; Züllighoven und Raasch 2006). In neueren Definitionen ist jedoch der Begriff Architekturelement üblich, da er allgemeiner ist als Komponente und auch andere Element-Arten wie beispielsweise Module mit umfasst (Bass et al. 2012). Die vorliegende Arbeit schließt sich diesem Verständnis an und verwendet den Begriff Architekturelement oder kurz den Begriff Element.

Der ISO/IEC/IEEE-Standard 42010-2011 thematisiert die *Dokumentation* von Architekturen. In diesem Kontext definiert er den Begriff der Softwarearchitektur:

Definition von Softwarearchitektur nach dem ISO/IEC/IEEE-Standard 42010-2011:

Architecture: Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution (ISO/IEC/IEEE-Standard-42010 2011).

Der Standard geht letztendlich auf den Vorgänger-Standard IEEE 1471-2000 zurück (IEEE-Standard-1471 2000). Bei der Entwicklung des Vorgängers fiel es schwer, sich auf eine gemeinsame Definition des Begriffs der Softwarearchitektur zu einigen. Diese Definition war einer der am stärksten umstrittenen Punkte des Standards (Maier et al. 2001). Im aktuellen Standard wurde die Architekturdefinition erneut geändert. Hier zeigt sich, dass die Diskussionen bezüglich des Begriffs der Softwarearchitektur noch in Gange sind.

Die Definition des ISO/IEC/IEEE-Standards fasst unter dem Begriff der Architektur auch die *Prinzipien* für Architekturdesign und -evolution. Dies ist gut geeignet für den Zweck der Architekturdokumentation, da auch die Prinzipien dokumentiert werden müssen. Für die in dieser Arbeit thematisierten Architekturprüfungen ist diese Mischung jedoch ungeeignet. Für Architekturprüfungen muss klar unterschieden werden zwischen der Architektur – diese wird geprüft – und den leitenden Prinzipien – diese sollen von der Architektur eingehalten werden. Dieselbe Unterscheidung machen auch Bass et al.: ihre Architekturdefinition umfasst lediglich die Strukturen des Systems. Die Architekturprinzipien und -vorgaben betrachten Bass et al. als von der Architektur getrennt.

Definition von Softwarearchitektur nach Bass et al.:

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both (Bass et al. 2012, S. 4).

Die Definition von Bass et al. unterscheidet sich von der Definition des ISO/IEC/IEEE-Standards in einer weiteren Hinsicht: Die Tatsache, dass Architekturen von Details abstrahieren und nur die Aspekte umfassen, die notwendig sind, um ein System im Ganzen zu betrachten, wird in obiger Definition des ISO/IEC/IEEE-Standards nicht explizit genannt (ISO/IEC/IEEE-Standard-42010 2011). Anders die Definition nach Bass et al., dort wird dieser Aspekt explizit

deutlich gemacht, denn die Architektur umfasst nur die Anteile, die auf dieser Abstraktionsebene benötigt werden („the set of structures needed to *reason* about the system“).

Auch nennen Bass et al. explizit die *Eigenschaften* von Elementen und Beziehungen als Teil der Architektur. Für die Architektur spielen nur solche Eigenschaften eine Rolle, die außerhalb der Elemente und Beziehungen relevant sind, nicht jedoch die privaten, internen Eigenschaften (Bass et al. 2012, S. 6). Hiermit verbinden Bass et al. die modernen Architekturdefinitionen mit Erkenntnissen, die bis zu der von Parnas beschriebenen Modularisierung von Softwaresystemen zurückreichen (Parnas 1972). Parnas hebt heraus, dass Module über extern sichtbare Schnittstellen verfügen und ihre Interna verbergen.

Diesem Architekturverständnis schließt sich die vorliegende Arbeit an, verwendet jedoch – wie Parnas – anstatt des Begriffs *Eigenschaften* den Begriff der *Schnittstelle* von Architekturelementen. Der Begriff der Schnittstelle ist üblich für die in dieser Arbeit betrachtete statische Sicht und er betont, dass für die Architektur lediglich die *extern bedeutsamen Eigenschaften* eine Rolle spielen.

Während die Schnittstellen von Architekturelementen eine fundamentale Bedeutung für Softwarearchitekturen haben, da sie die Unterscheidung zwischen privaten und öffentlichen Anteilen deutlich machen, spielen die Eigenschaften der Beziehungen eine geringere Rolle. In der Literatur thematisiert man bezüglich der Beziehungen meist lediglich ihren Typ (beispielsweise Vererbung versus Operationsaufruf). Deswegen orientiert sich die Architekturdefinition dieser Arbeit an dem üblichen Verständnis, dass die Eigenschaften implizit Teil der Beziehungen seien (Reussner und Hasselbring 2009; ISO/IEC/IEEE-Standard-42010 2011).

Bass et al. nennen in ihrer Definition explizit, dass eine Architektur aus *mehreren* Strukturen bestehen kann. Keine dieser Teilstrukturen kann für sich alleine in Anspruch nehmen, *die* Architektur des Systems zu sein (Bass et al. 2012). Das Konzept der Teilstrukturen ergänzt die oben erläuterten Konzepte *Sicht* und *Standpunkt* (ISO/IEC/IEEE-Standard-42010 2011). Der Zusammenhang ist folgendermaßen: Ein *Standpunkt* beschreibt einen Blickwinkel auf Architekturen. Je nachdem, aus welchem Standpunkt man eine Architektur betrachtet, tritt eine andere *Teilstruktur* in den Vordergrund. Dokumentiert man die zu einem Standpunkt gehörige Teilstruktur, so stellt diese Dokumentation eine *Sicht* dar (Bass et al. 2012).

Für diese Arbeit ist es nicht notwendig, dass die hier gegebene Architekturdefinition explizit darauf hinweist, dass Architekturen aus mehreren Teilstrukturen bestehen, da sich die Arbeit auf die Architektur des Quelltextes, die statische Struktur, konzentriert. Wie die Definition des ISO/IEC/IEEE-Standards, so soll jedoch auch die in dieser Arbeit gegebene Definition so verstanden werden, dass Architekturen verschiedene Teilstrukturen enthalten, die sich in Architektursichten dokumentieren lassen.

Der Lesbarkeit halber wird im Folgenden der Begriff Architektur verwendet, selbst wenn lediglich die statische Struktur gemeint ist. Sollte die Gesamtheit aller Strukturen oder eine andere Struktur gemeint sein, so wird explizit darauf hingewiesen.

2.3 Zusammenhang zwischen Quelltext und Softwarearchitektur

Wie genau hängen Softwaresysteme mit ihrer Architektur zusammen? Dieser Abschnitt behandelt die Frage für die statische Sicht, das heißt: er behandelt den Zusammenhang zwischen Quelltext⁵ und Architektur.

Um diesen Zusammenhang herstellen zu können, formalisieren die verschiedenen Ansätze zur statischen Architekturprüfung die Softwarearchitektur und den Quelltext. Für diese Formalisierungen ist eine mengentheoretische Notation gebräuchlich. Übertragen auf solch eine Notation stellt sich die Architekturdefinition dieser Arbeit wie folgt dar:

Definition 2-1: Softwarearchitektur, formal

Eine Softwarearchitektur ist ein Tupel $\langle A, B_A, S_A, Z_{AS} \rangle$ mit

- A : die Menge der Architekturelemente
- B_A : die Menge der gerichteten Beziehungen zwischen Architekturelementen. Dabei gilt: $B_A \subseteq A \times A$. Jedes Tupel $(a1, a2) \in B_A$ bedeutet: es besteht eine Beziehung von $a1$ zu $a2$.
- S_A : die Menge der Schnittstellen der Architekturelemente
- Z_{AS} : die Zuordnung zwischen Architekturelementen und deren Schnittstellen. Die Zuordnung ist eine Relation, für die gilt: $Z_{AS} \subseteq A \times S_A$.

In den folgenden Kapiteln wird auf diese Definition zurückgegriffen, um die bisherigen Ansätze zur Architekturprüfung vorzustellen und abzugrenzen. Der in der vorliegenden Arbeit präsentierte Ansatz zur stilbasierten Architekturprüfung stützt sich ebenfalls auf diese mengentheoretische Darstellung.

Es sei angemerkt, dass sich die in diesem Abschnitt vorgestellten Formalisierungen der Verständlichkeit halber auf den Kern der Konzepte beschränken. Im Verlauf der vorliegenden Arbeit werden diese Formalisierungen bei Bedarf erweitert (beispielsweise um verschiedene Beziehungstypen).

Für die Formalisierung des Quelltextes gilt: zwar unterscheiden sich die Formalisierungen der verschiedenen Ansätze im Detail, es ist ihnen jedoch gemein, dass sie den Quelltext als Struktur betrachten. Diese Struktur extrahieren sie mittels einer statischen Analyse aus den Quelltextdateien (Knodel und Popescu 2007; Murphy et al. 2001; Sangal et al. 2005). Diese Arbeit bezeichnet die Struktur als Quelltextstruktur und definiert:

Definition 2-2: Quelltextstruktur

Eine Quelltextstruktur ist eine *Abstraktion* des Quelltextes, sie besteht aus *Quelltextelementen*, den *Schnittstellen* der Quelltextelemente und deren *Beziehungen*.

⁵ Es sei an dieser Stelle angemerkt, dass der Begriff des Quelltextes hier für die Implementation eines Softwaresystems steht. Abhängig von der gewählten Programmiersprache und dem gewählten Komponentenmodell kann der Quelltext auch Konfigurationsdateien (wie beispielsweise für das Rahmenwerk Spring, projects.spring.io/spring-framework) oder Meta-Informationen (wie beispielsweise Java-Annotationen) umfassen.

Die Quelltextstruktur ist eine Abstraktion des Quelltextes und basiert auf maschinell eindeutig identifizierbaren Quelltexteinheiten. Was bei einem konkreten Softwaresystem unter den Quelltextelementen, Schnittstellen und Beziehungen verstanden wird, hängt von folgenden Faktoren ab: dem Programmierparadigma, der gewählten Programmiersprache, der eventuellen Verwendung eines Komponentenmodells, dem verwendeten Werkzeug zur Architekturprüfung und den individuellen Entscheidungen des Entwicklungsteams. Das bedeutet, die Quelltextstruktur ist nicht eindeutig. Abhängig davon, welche programmiersprachlichen Konstrukte als Quelltextelement, als Schnittstelle und als Beziehung interpretiert werden, lassen sich für einen Quelltext verschiedene Quelltextstrukturen ermitteln. Die Quelltextstruktur kann als *Modell* des Quelltextes verstanden werden, im Sinne des Modellbegriffs von Stachowiak (Stachowiak 1973).

Der in dieser Arbeit vorgestellte Ansatz zielt auf das Paradigma der objektorientierten Programmiersprachen. Da jedoch einige der weiter unten vorgestellten bisherigen Ansätze zur Architekturprüfung mit prozeduralen Sprachen arbeiten, seien im Folgenden die Bestandteile von Quelltextstrukturen unabhängig von einem Programmierparadigma definiert. Es gilt:

Definition 2-3: Quelltextelement

Ein *Quelltextelement* ist ein Ausschnitt des Quelltextes.

Üblicherweise ist ein Quelltextelement eine abgeschlossene, programmiersprachliche Einheit. Bei objektorientierten Programmiersprachen ist es in der Regel eine Klasse. Quelltextelemente, die aus kleineren Einheiten als Klassen bestehen sind nicht üblich, sie bieten eine zu geringe Abstraktion von den Details des Quelltextes und werden in dieser Arbeit nicht weiter betrachtet. Dass solche Architekturelemente unüblich sind, zeigt sich unter anderem darin, dass die bestehenden Werkzeuge zur Architekturprüfung solch kleine Quelltextelemente in der Regel nicht unterstützen. Der in dieser Arbeit vorgestellte Ansatz wäre jedoch problemlos für kleinere Einheiten, wie beispielsweise Operationen, erweiterbar, sofern der Bedarf entstehen sollte.

Bei prozeduralen Systemen ist ein Quelltextelement beispielsweise eine Prozedur oder ein Datentyp.

Ferner können größere Quelltexteinheiten als Klassen oder Prozeduren als Quelltextelement betrachtet werden, sofern das verwendete Prüfwerkzeug dies unterstützt und die verwendete Programmiersprache solche Einheiten anbietet. Beispiele für solche größeren Einheiten sind Java-Packages, C++-Namensräume oder Module. Werden Komponententechnologien wie beispielsweise Eclipse Plugins verwendet, so lassen sich auch diese Komponenten als Quelltexteinheiten betrachten.

Definition 2-4: Schnittstelle eines Quelltextelements

Die *Schnittstelle* eines Quelltextelements ist die Menge aller außerhalb des Elements sichtbaren Eigenschaften.

Die Schnittstelle beinhaltet alle Anteile von Quelltextelementen, auf die von anderen Quelltextelementen aus zugegriffen werden darf. In der Begriffswelt der Modularisierung handelt es sich somit um die Export-Schnittstelle (Nagl 1990). Abhängig von der Art eines Quelltextelements und dem Kontext, in dem die Quelltextstruktur benötigt wird, kann die Schnittstelle sehr

unterschiedlich gestaltet sein. Beispielsweise kann die Schnittstelle von Klassen aus allen öffentlichen Operationen und Attributen bestehen⁶.

Definition 2-5: Beziehung zwischen Quelltextelementen

Zwei Quelltextelemente stehen in einer *Beziehung*, wenn ein Quelltextelement das andere referenziert⁷.

Für objektorientierte Systeme gilt: im Kontext von Softwarearchitekturen lassen sich üblicherweise drei Typen von Beziehungen zwischen Klassen unterscheiden: Enthält-, Benutzt- und Vererbungsbeziehung. Eine Benutztbeziehung von Klasse A zu Klasse B besteht in der Regel in folgenden Fällen: A ruft eine Operation von B auf, A greift auf eine öffentliche Variable von B zu, A verwendet B als Parameter- oder Rückgabety, A verwendet B als Typ für eine Variable, A führt eine Typumwandlung hin zu B durch (vgl. Lilienthal 2008, S. 25).

Beziehungen in prozeduralen Systemen entstehen beispielsweise, wenn eine Prozedur eine andere aufruft oder wenn ein Modul einen Datentyp verwendet, der in einem anderen Modul definiert ist. Prinzipiell gilt: Quelltextelemente können auch sich selber referenzieren. Solche Beziehungen sind im Kontext dieser Arbeit kaum von Interesse.

Die verschiedenen Ansätze zur Prüfung auf Architekturtreue konkretisieren die Quelltextstruktur auf unterschiedliche Weise. Diese Arbeit erläutert jeweils, welches Verständnis von Quelltextstruktur den weiter unten vorgestellten Ansätzen und Werkzeugen zugrunde liegt. Bei einigen Ansätzen und Werkzeugen ist bereits festgelegt, welche Programmiersprachen geprüft werden können und was bei diesen Sprachen als Quelltextelemente sowie deren Schnittstellen und Beziehungen betrachtet wird. Bei anderen Ansätzen und Werkzeugen kann die prüfende Person auswählen, was konkret unter der Quelltextstruktur zu verstehen ist.

Die Quelltextstruktur im Rahmen von Architekturprüfungen ist ein Modell des Quelltextes zum Zwecke der Prüfung, sie enthält nicht alle im Quelltext vorhandenen Elemente, Beziehungen und Schnittstellen, sondern nur diejenigen, welche für die Prüfung relevant sind. Ferner ist es verbreitet, dass die Quelltextstruktur Beziehungen zusammenfasst: sie unterscheidet nicht, ob eine oder mehrere Beziehungen von einem Quelltextelement A zu einem Quelltextelement B existieren. Stattdessen enthält die Quelltextstruktur genau eine Beziehung von A zu B, sofern im Quelltext mindestens eine für die Prüfung relevante Beziehung von A zu B existiert.

Es ist möglich, dass im Rahmen einer Architekturprüfung eines objektorientierten Systems beispielsweise nur die Vererbungsbeziehungen betrachtet werden. Würde in diesem Fall eine Klasse A eine Klasse B benutzen, aber A würde nicht von B erben, so enthielte die Quelltextstruktur, *keine* Beziehung von A nach B. Wird im Rahmen von Architekturprüfungen von einer Quelltextbeziehung, einem Quelltextelement oder einer Quelltextschnittstelle gesprochen, so ist immer die Quelltextstruktur gemeint, nicht der Quelltext. So hält es auch diese Arbeit. Ist der tatsächliche Quelltext gemeint, so wird explizit darauf hingewiesen.

Da Prüfwerkzeuge die Quelltextstruktur in maschinenlesbarer Form benötigen, sei hier eine formale Definition ergänzt:

⁶ An dieser Stelle ein kleiner Vorgriff zur Erläuterung: Im Kontext von Konformanzprüfungen ist es möglich, die Quelltextstruktur unterschiedlich zu gestalten. Mehrere Ansätze erlauben, dass die prüfende Person die Art der Quelltextstruktur individuell definiert. Bei objektorientierten Klassen beispielsweise bedeutet dies, dass die Schnittstelle in einem Fall ausschließlich die in der Klasse definierten öffentlichen Operationen und Attribute umfassen kann, in einem anderen Fall können zusätzlich die geerbten öffentlichen Operationen und Attribute als Teil der Schnittstelle begriffen werden. Die beispielhafte Umsetzung dieser Arbeit behandelt dieses Thema.

⁷ Der Begriff „referenzieren“ steht hier allgemein für Beziehungen, er macht keine Aussage über den Beziehungstyp. Synonym zu „referenzieren“ wird auch der Begriff „kennen“ verwendet (Lilienthal 2008, S. 25).

Definition 2-6: Quelltextstruktur, formal

Eine Quelltextstruktur ist ein Tupel $\langle Q, B_Q, S_Q, Z_{QS} \rangle$ mit

- Q : die Menge der Quelltextelemente
- B_Q : die Menge der gerichteten Beziehungen zwischen Quelltextelementen.
Dabei gilt: $B_Q \subseteq Q \times Q$
- S_Q : die Menge der Schnittstellen der Quelltextelemente
- Z_{QS} : die Zuordnung zwischen Quelltextelementen und deren Schnittstellen.
Die Zuordnung ist eine Relation, für die gilt: $Z_{QS} \subseteq Q \times S_Q$.

Um die Formalisierung übersichtlich zu halten, unterscheidet sie keine Beziehungstypen. Eine Formalisierung der Beziehungstypen wird im Verlauf des Textes an den benötigten Stellen ergänzt. Dasselbe gilt für die unten folgende formale Definition für Softwarearchitektur.

In der Einleitung wurde erläutert, was die in der Architekturdefinition genannten Begriffe *Architekturelement*, *Beziehung* und *Schnittstelle* für die statische Struktur bedeuten. Diese Erläuterungen seien im Folgenden in Form dreier expliziter Definitionen präzisiert, dabei wird der Zusammenhang zwischen Quelltext und Architektur aufgezeigt.

Definition 2-7: Architekturelement

Ein *Architekturelement* ist eine Strukturierungseinheit innerhalb eines Softwaresystems. Ein Architekturelement umfasst ein oder mehrere Quelltextelemente.

Die Anzahl der Architekturelemente ist bei gut entworfenen Architekturen geringer als die Anzahl der Quelltextelemente, so hilft die Architektur, einen Überblick über umfangreiche Quelltexte zu erreichen.

Formal sei die Zuordnung zwischen Quelltext- und Architekturebene folgendermaßen definiert:

Definition 2-8: Zuordnung zwischen Quelltext- und Architekturelementen

- Z_{QA} : Eine Relation zwischen den Mengen der Quelltext- und der Architekturelementen. Dabei gilt: $Z_{QA} \subseteq Q \times A$

Abbildung 2 visualisiert die Zuordnung in UML. Die Richtung der Zuordnung ist für Architekturprüfungen irrelevant, aus diesem Grunde ist die Zuordnung nicht als Pfeil, sondern als Linie dargestellt.

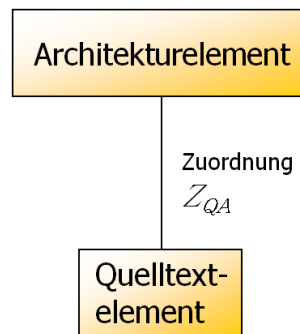


Abbildung 2: Architekturelemente umfassen Quelltextelemente

Entwicklungsteams können für ihr Softwaresystem individuell entscheiden, welche Quelltextelemente sie jeweils als Architekturelement betrachten. Für Teams, die ihre Architektur planen und entwickeln, gilt in der Regel: wird neuer Quelltext programmiert, so hat die programmierende Person eine Vorstellung davon, zu welchem Architekturelement der neue Quelltext gehören soll. Dass er mit dieser Intention entwickelt wurde, ist dem Quelltext jedoch nachträglich nicht unbedingt zu entnehmen. Das bedeutet: Entwicklungsteams müssen ihre Zuordnung zwischen Quelltext- und Architekturelementen Z_{QA} explizit festhalten.

Um die Architektur zu prüfen, gibt die prüfende Person die Zuordnung Z_{QA} in das gewählte Prüfwerkzeug ein. Abhängig von der Benutzungsschnittstelle des Prüfwerkzeugs werden die Quelltextelemente jedes Architekturelements einzeln aufgelistet oder es werden Namensmuster der Quelltextelemente genutzt, beispielsweise mit regulären Ausdrücken. Einige Werkzeuge erlauben auch eine grafische Zuordnung. Darüber hinaus ist es in manchen Werkzeugen möglich, die Zuordnung über programmiersprachliche Gruppierungen abzukürzen. Sollen beispielsweise alle Klassen eines Java-Packages demselben Architekturelement zugeordnet werden, so kann die prüfende Person diese Zuordnung verkürzt angeben, indem sie das Package dem Architekturelement zuordnet.

Häufig unterscheiden Entwicklungsteams innerhalb der statischen Struktur ihrer Softwarearchitekturen verschiedene Substrukturen (Bass et al. 2012; Hofmeister et al. 2000, S. 110). Auf einem hohen Abstraktionsniveau sind Schichtenarchitekturen sehr verbreitet. Drei-Schichten-Architekturen beispielsweise teilen alle Quelltextelemente auf drei Architekturelemente auf. Auf detaillierterer Ebene kann beispielsweise eine Schicht intern aus mehreren kleineren Architekturelementen bestehen. Architekturen, bei denen Architekturelemente geschachtelt sind, bezeichnet man als *hierarchische* Architekturen (Koschke und Simon 2003). Es gilt: wenn ein Architekturelement A ein anderes Architekturelement B enthält, so enthält A alle Quelltextelemente von B.

Ist die Relation Z_{QA} festgelegt und bekannt, so werden die Beziehungen der Quelltextstruktur durch Aggregation (Lilienthal 2008) in die Architektur übernommen. Abbildung 3 verdeutlicht diesen Zusammenhang. Die Quelltextstruktur in diesem Beispiel besteht aus 4 Quelltextelementen. Quelltextelement 1 und 2 sind dem Architekturelement A zugeordnet, Quelltextelement 3 und 4 dem Architekturelement B. Die Beziehung von Quelltextelement 1 zu 2 ist auf Architekturebene irrelevant, da beide Quelltextelemente demselben Architekturelement zugeordnet sind. Dasselbe gilt für die Beziehung von Quelltextelement 4 zu 3. Die Beziehungen der Quelltextelemente 1 und 2 zu den Quelltextelementen 3 und 4 überschreiten die Grenzen von Architekturelementen. Diese Beziehungen verlaufen von Quelltextelementen in A zu Quelltextelementen in B. Auf Architekturebene führt dies zu einer Beziehung von Architekturelement A zu Architekturelement B.

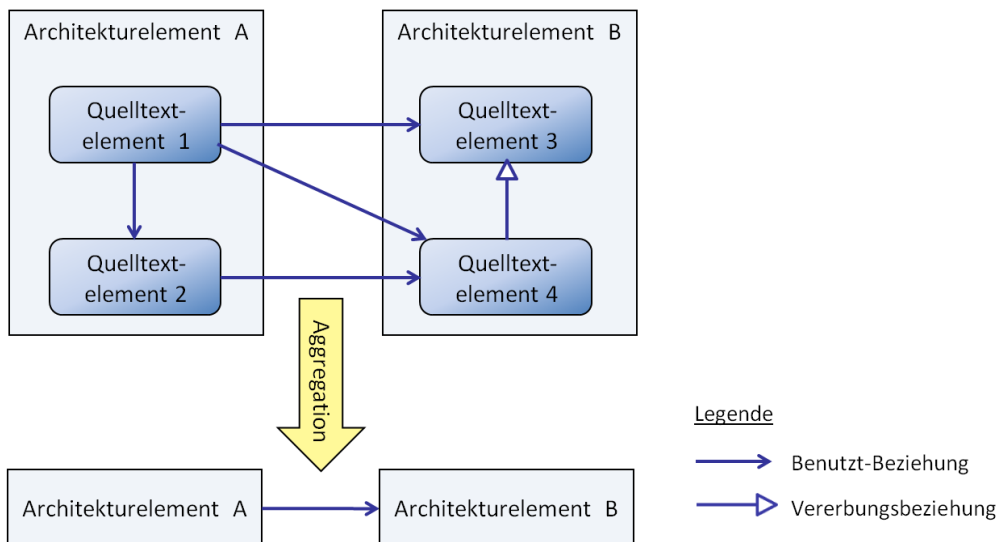


Abbildung 3: Aggregation von Beziehungen aus der Quelltextstruktur in die Architektur (nach Lilienthal 2008)

Bei hierarchischen Architekturen kommt eine weitere Beziehung zwischen Architekturelementen hinzu, die nicht auf eine Klassenbeziehung zurückzuführen ist: die Enthältbeziehung. Diese Beziehung wird von dem jeweiligen Entwicklungsteam individuell festgelegt.

Definition 2-9: Beziehung zwischen Architekturelementen

Zwei Architekturelemente stehen in einer *Beziehung*, wenn die zugehörigen Quelltextelemente in Beziehung stehen oder wenn ein Architekturelement das andere enthält. Beziehungen zwischen Architekturelementen sind gerichtet.

Klassen, die in einer programmiersprachlichen Enthältbeziehung stehen (beispielsweise geschachtelte Klassen in Java), werden im Kontext objektorientierter Softwarearchitekturen üblicherweise demselben Architekturelement zugeordnet. Somit sind für die Architekturebene lediglich die Benutzt- und die Vererbungsbeziehung zwischen Klassen relevant.

Abbildung 4 visualisiert die möglichen Beziehungen von Architekturelementen. Die Enthältbeziehungen legt das Entwicklungsteam fest, die Benutzt- und Vererbungsbeziehungen ergeben sich aus den Beziehungen auf der Klassenebene.

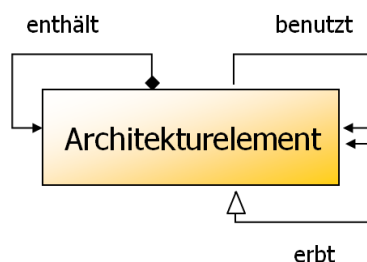


Abbildung 4: Beziehungen zwischen Architekturelementen

Die Beziehungen zwischen Architekturelementen sind gerichtet. Nur so hilft die Architektur, beispielsweise bei Änderungen, abhängige Quelltextstellen zu identifizieren. Ändert ein Teammitglied den Quelltext eines Architekturelements A, so weiß es, dass abhängige Quelltextstellen nur in solchen Architekturelementen zu finden sind, die – direkt oder indirekt – eine Beziehung hin zu A besitzen.

Einige Ansätze zur Architekturprüfung unterstützen das Konzept von Architekturelement-Schnittstellen. Auch die Schnittstellen von Architekturelementen basieren auf der Quelltextstruktur:

Definition 2-10: Schnittstelle eines Architekturelements

Die *Schnittstelle* eines Architekturelements ist die Menge aller außerhalb des Elements sichtbaren Eigenschaften. Die Schnittstelle eines Architekturelements besteht aus einer Teilmenge der Schnittstellen aller Quelltextelemente des Architekturelements.

Genau wie Parnas bei Modulen Interna und Schnittstelle unterscheidet (Parnas 1972), so lässt sich auch für Architekturelemente definieren, welche Inhalte als verborgene Interna betrachtet werden und welche Inhalte Teil der von außen zugreifbaren Schnittstelle sein sollen (siehe Abbildung 5).

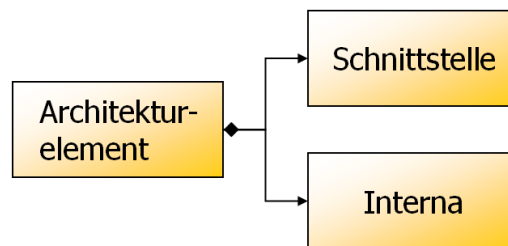


Abbildung 5: Architekturelemente besitzen eine öffentliche Schnittstelle und private Interna

Entwicklungsteams können darüber entscheiden, welche Quelltextelemente eines Architekturelements sie als öffentlich betrachten und somit der Schnittstelle zuordnen und welche Quelltextelemente Architekturelement-intern sein sollen und somit von außen nicht benutzt werden dürfen.

Üblicherweise besitzen Quelltextelemente, wie beispielsweise objektorientierte Klassen, ihrerseits eine Schnittstelle. Die Schnittstellen aller öffentlichen Quelltextelemente eines Architekturelements bilden die Schnittstelle des Architekturelements. Die Interna des Architekturelements bestehen aus den gesamten internen Quelltextelementen und aus den Interna der öffentlichen Quelltextelemente. Auf beides darf von außerhalb des Architekturelements nicht zugegriffen werden.

2.4 Ist- und Soll-Architektur

Für Architekturprüfungen werden Soll-Architektur und Ist-Architektur unterschieden (die Ist-Architektur wird auch als aktuelle oder implementierte Architektur bezeichnet). Ziel einer Architekturprüfung ist es, Abweichungen der Ist-Architektur von der Soll-Architektur werkzeuggestützt aufzudecken (Knodel und Popescu 2007; Passos et al. 2010; Simon und Meyerhoff 2002).

Definition 2-11: Ist-Architektur

Die *Ist-Architektur* ist die implementierte Architektur eines Softwaresystems.

Definition 2-12: Soll-Architektur

Die *Soll-Architektur* ist eine Form von Architekturvorgabe. Die Soll-Architektur beschreibt, wie die Ist-Architektur eines Softwaresystems sein soll, indem sie Architekturbestandteile aufzählt.

Der Aufbau von Ist- und Soll-Architekturen ist abhängig von dem gewählten Prüfansatz. Bei den meisten Ansätzen beinhalten die Ist- und die Soll-Architektur jeweils Architekturelemente und Beziehungen. Bei einigen Ansätzen kann die Soll-Architektur zusätzlich Vorgaben für die Schnittstellen der Architekturelemente enthalten. In diesen Fällen beinhaltet die Ist-Architektur die im System vorhandenen Schnittstellen.

Die hier gegebenen Definitionen orientieren sich an den bei Architekturprüfungen üblichen Begrifflichkeiten. Der Klarheit halber soll an dieser Stelle ein genauerer Blick auf die Begriffe Ist- und Soll-Architektur geworfen werden:

Damit Prüfwerkzeuge die Ist- mit der Soll-Architektur vergleichen können, benötigen sie jeweils eine *Repräsentation* von Ist und Soll. Diese Repräsentationen beinhalten die *für Architekturprüfungen* relevanten Architekturelemente, Beziehungen und deren relevante Eigenschaften. Im Sinne von Stachowiak stellen diese Repräsentationen *Modelle* dar (Stachowiak 1973): Die Repräsentation der Ist-Architektur stellt ein Modell der in einem System *implementierten* Architektur dar. Die Repräsentation der Soll-Architektur stellt ein Modell einer *gewünschten* Architektur dar.

Den Aufbau der Soll-Architektur legt die prüfende Person bzw. ein Projektteam fest, sie wird in einer für das Prüfwerkzeug einlesbaren Form beschrieben oder interaktiv in das Prüfwerkzeug eingegeben. Die Soll-Architektur hat keine physikalische Entsprechung in Form eines bestehenden Softwaresystems. Somit handelt es sich *nur* bei der Ist-Architektur tatsächlich um eine Softwarearchitektur im Sinne von Abschnitt 2.2. Die Repräsentation der Ist-Architektur kann als eine *Sicht* entsprechend des ISO-IEC-IEEE-Standards verstanden werden (ISO/IEC/IEEE-Standard-42010 2011), da sie die bestehende Architektur zu einem bestimmten Zweck (die Prüfung) aus einem hierfür nützlichen Standpunkt aus darstellt.

Im Folgenden werden die Begriffe Ist- und Soll-Architektur auch verwendet, wenn es sich genau genommen um die *Modelle* von Ist und Soll handelt, sofern die genaue Bedeutung dem Kontext entnehmbar ist. Dies ist im Bereich der Architekturprüfungen üblich. Es wird davon gesprochen, dass die Ist-Architektur *berechnet* wird, obwohl genau genommen lediglich das *Modell der Ist-Architektur* berechnet wird. Man sagt, ein Prüfwerkzeug vergleicht die Ist- mit der Soll-Architektur, obgleich genau genommen die im Prüfwerkzeug gehaltenen *Modelle* von Ist und Soll verglichen werden.

Die Modelle der Ist- und der Soll-Architektur werden folgendermaßen ermittelt:

Die Werkzeuge berechnen das Modell der *Ist-Architektur* anhand des Quelltextes sowie anhand von manuell ergänzten Zusatzinformationen. Diese Zusatzinformationen bestehen mindestens aus der oben definierten Abbildungsvorschrift zwischen Quelltext- und Architekturelementen: der Zuordnung Z_{QA} . Je nachdem, welches Prüfverfahren das benutzte Werkzeug verwendet, können Zusatzinformationen bezüglich der Schnittstellen und der Beziehungen hinzukommen.

In manchen Projekten ist das Wissen über Zusatzinformationen für die Ist-Architektur ungenügend oder verloren. Oft handelt es sich um sogenannte Altsysteme (engl. legacy systems), bei denen die Mitglieder des ursprünglichen Entwicklungsteams nicht mehr verfügbar sind und keine ausreichende, aktuelle Dokumentation vorliegt (Sommerville 2001). Ohne die Zusatzinformationen lässt sich die Ist-Architektur nicht korrekt ermitteln und folglich keine aussagekräftige Architekturprüfung durchführen.

Bei solchen Systemen kann es sinnvoll sein, die Softwarearchitektur mit Hilfe von Reverse-Engineering-Ansätzen (Chikofsky und Cross 1990) zu rekonstruieren und die in der rekonstruierten Architektur enthaltenen Zusatzinformationen zu verwenden. Es existiert eine Vielzahl an Reverse-Engineering-Verfahren zur Architektur-Rekonstruktion (Koschke 2005; Ducasse und Pollet 2009). Abhängigkeitsbasierte Ansätze gehen beispielsweise von der Vermutung aus, dass besonders eng gekoppelte Klassen zu ein und demselben Architekturelement gehören könnten. Diese Ansätze verwenden unter anderem verschiedene Entfernungsmaße, um die Kopplung zwischen Quelltextelementen zu berechnen. Die rekonstruierten Architekturen unterscheiden sich, je nachdem, welche Metriken mit welchen Gewichtungen verwendet werden (Jahnke 2009). Reverse-Engineering-Verfahren sind in der Regel werkzeuggestützt. Neben vollautomatischen Verfahren existieren halbautomatische Verfahren. Halbautomatische Verfahren beinhalten manuelle Schritte, hier können unter anderem eventuell vorhandene Dokumentationen gesichtet und Projektmitglieder befragt werden. Sind die ursprünglichen Projektmitglieder nicht mehr verfügbar und die Dokumentationen nicht ausreichend, so besteht keine Möglichkeit sicherzustellen, dass die durch Reverse-Engineering-Verfahren ermittelten Zusatzinformationen der ursprünglichen Architekturintention entsprechen (Koschke 2005). Empirische Studien legen nahe, dass sich insbesondere vollautomatisch rekonstruierte Architekturen stark von der ursprünglichen Architekturintention unterscheiden (Koschke 2000). Es gilt im Einzelfall zu entscheiden, ob eine Rekonstruktion der Architektur wirtschaftlich und erfolgversprechend durchführbar ist (Jahnke 2009).

Architekturprüfungen lassen sich zwar im Prinzip für Systeme, bei denen die Zusatzinformationen durch Reverse-Engineering-Ansätze ermittelt wurden, verwenden. In erster Linie zielen Architekturprüfungen jedoch darauf, die Entstehung solcher Systeme im Vorwege zu verhindern.

Die *Soll-Architektur* wird manuell angegeben. Sie beschreibt, wie die Architektur aufgebaut sein soll. Die Informationen darüber, wie die Soll-Architektur strukturiert ist, können je nach Projekt aus unterschiedlichen Quellen stammen.

In Softwareprojekten, in denen die Architektur überwiegend zufällig entstanden ist, oder bei den oben beschriebenen Altsystemen, muss vor der ersten Prüfung eine Soll-Architektur erstellt werden. Die Prüfung kann in diesen Fällen beispielsweise zeigen, ob sich das System zu einer guten Architektur hin umstrukturieren lässt oder ob eine Neuentwicklung wirtschaftlicher ist.

In Projekten mit einer Architekturvorstellung, in denen die Projektmitglieder neuen Quelltext mit einer konkreten Intention bezüglich der Architektur erstellen, spricht diese Arbeit von einer *intendierten* Architektur. Wenn Entwicklungsteams einen Schritt weiter gehen und ihre Architektur explizit planen, verwendet diese Arbeit den Begriff der *geplanten* Architektur.

Besteht eine geplante oder zumindest eine intendierte Architektur, so dient diese bei Architekturprüfungen als Grundlage für die in das Prüfwerkzeug einzugebende Soll-Architektur (Knodel und Popescu 2007; Passos et al. 2010; Simon und Meyerhoff 2002).

Sofern die prüfende Person die intendierte oder geplante Architektur nicht kennt, kann sie auf Architekturdokumentationen, Quelltextkommentare und Gespräche mit Teammitgliedern zurückgreifen. Oft ist die intendierte oder geplante Architektur nicht vollständig dokumentiert, die Dokumentation ist nicht mehr aktuell oder die Vorstellungen der Teammitglieder über die Architektur differieren. In diesen Fällen müssen Entwicklungsteams im Zuge von Architekturprüfungen die Architektur neu planen (Simon und Meyerhoff 2002).

Damit Software-Werkzeuge zur Architekturprüfung die Ist- mit der Soll-Architektur vergleichen können, benötigen sie eine formale Darstellung der Ist- und der Soll-Architektur. Das in Entwicklungsteams vorhandene Architekturwissen über die Zusatzinformationen und die intendierte oder geplante Architektur erreicht diesen Formalisierungsgrad häufig nicht, da dieses Architekturwissen ursprünglich lediglich der Abstimmung innerhalb des Teams diene und nicht maschinenlesbar zu sein brauchte. Prüfen Entwicklungsteams ihr System erstmalig auf Architekturtreue, müssen die informellen Kenntnisse über die Zusatzinformationen sowie die intendierte bzw. geplante Architektur in der Regel stärker formalisiert werden, um als Grundlage für Ist- und Soll-Architektur dienen zu können.

Die Struktur von Ist- und Soll-Architekturen muss für alle Prüfansätze formalisiert werden. Wie die folgenden Kapitel zeigen, unterscheidet sich die Art der Formalisierung von Ansatz zu Ansatz.

2.5 Grenzen des Architekturentwurfs mit Soll-Architekturen

Wie oben dargestellt, dienen Softwarearchitekturen dazu, große Systeme auf abstrakter Ebene zu planen und zu verstehen. Was bedeutet das konkret für die Arbeit in Entwicklungsteams?

Laut Definition 1-1 bestehen Architekturen aus Architekturelementen, deren Schnittstellen und Beziehungen. Abbildung 6 visualisiert den prinzipiellen Aufbau von Softwarearchitekturen in Form eines UML-Metamodells. Softwarearchitekturen lassen sich als Exemplare dieses Metamodells verstehen.

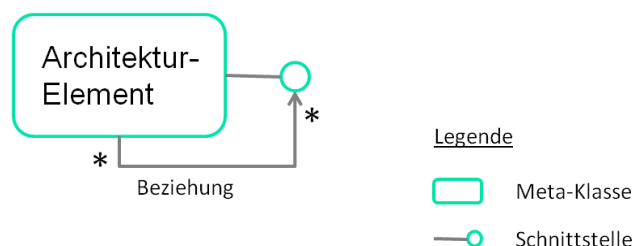


Abbildung 6: Allgemeines Metamodell für Architekturen

Wird ein neues Softwaresystem entwickelt, so entwirft das Entwicklungsteam eine passende Soll-Architektur entsprechend dieses Metamodells. Das Entwicklungsteam wählt die Architekturelemente, Schnittstellen und Beziehungen anhand der individuellen Anforderungen an das zukünftige System. Abbildung 7 zeigt beispielhaft eine kleine Architektur bestehend aus zwei Architekturelementen mit ihren Schnittstellen und einer Beziehung. Diese Architektur ist ein Exemplar des in Abbildung 6 dargestellten Metamodells.

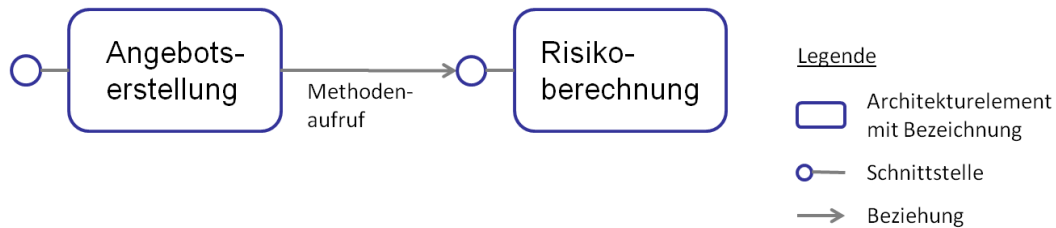


Abbildung 7: UML-Darstellung einer Softwarearchitektur aus der Versicherungswirtschaft

Im Architekturentwurf wird üblicherweise zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden (Posch et al. 2011; Nagl 1990).

Der Begriff der *funktionalen Anforderungen* bezeichnet die fachlichen und die technischen Anforderungen an die Funktionalität des Systems. Die fachlichen Anforderungen bilden die Motivation, aus der heraus Softwaresysteme entwickelt werden, sie ergeben sich aus den Aufgaben, die das zukünftige System unterstützen soll. Eine fachliche Aufgabe ist beispielsweise die Risikoberechnung für eine Versicherungspolice. Die technischen Anforderungen machen Einschränkungen bezüglich der zu verwendenden Technologien, beispielsweise kann eine technische Anforderung festlegen, welche Datenbank ein Softwaresystem verwenden soll (Pohl 2008).

Die *nicht-funktionalen Anforderungen* umfassen interne Qualitätsanforderungen, wie Wartbarkeit und Verständlichkeit und Leistungsanforderungen an Performanz und Speicherbedarf (Brügge und Dutoit 2004; Shaw und Garlan 1996). Sie werden auch als „quality attribute requirements“ bezeichnet (Bass et al. 2012). Es existieren verschiedene Qualitätsmodelle, die den allgemeinen Begriff der Qualität in Unterbegriffen konkretisieren (Balzert 1998, S. 258; ISO/IEC-Standard-25010 2010). Für das hier dargestellte Prinzip des Architekturentwurfs sind die Unterschiede zwischen den Qualitätsmodellen unerheblich, hier genügt die genannte Unterscheidung in interne Qualitätsanforderungen und Leistungsanforderungen. Tabelle 2-1 gibt eine beispielhafte Übersicht der verschiedenen Anforderungsarten.

Funktionale Anforderungen:	Fachliche Anforderungen <ul style="list-style-type: none"> • Aufgaben des Systems • z.B. fachliche Berechnungen
	Technische Anforderungen <ul style="list-style-type: none"> • Datenbank • Programmiersprache etc.
Nicht-funktionale Anforderungen:	Interne Qualitätsanforderungen <ul style="list-style-type: none"> • Wartbarkeit • Verständlichkeit etc.
	Leistungsanforderungen <ul style="list-style-type: none"> • Performanz • Speicherbedarf etc.

Tabelle 2-1: Anforderungsarten

Die Unterscheidung zwischen funktionalen und nicht-funktionalen Anforderungen ist üblich, da diese Anforderungsarten eine unterschiedliche Rolle im Architekturentwurf spielen: Entwirft ein Entwicklungsteam seine Architektur, so teilt es funktionale Anforderungen auf die zukünftigen Architekturelemente auf. Diese Aufteilung geschieht primär anhand der nicht-funktionalen Anforderungen; denn für dieselben funktionalen Anforderungen ließen sich unbegrenzt viele verschiedene Architekturen entwerfen, die sich lediglich bezüglich der nicht-funktionalen Anforderungen unterscheiden (Shaw und Garlan 1996). Ein guter Architekturentwurf berücksichtigt alle funktionalen Anforderungen und ist zusätzlich darauf ausgerichtet, die nicht-funktionalen Anforderungen möglichst weitreichend zu erfüllen.

Wie lassen sich beim Architekturentwurf die funktionalen Anforderungen so auf Architekturelemente verteilen, dass auch die nicht-funktionalen Anforderungen möglichst gut erfüllt werden? Um Softwaresysteme mit hoher interner Qualität zu entwerfen, stehen verschiedene Entwurfsprinzipien für Softwarearchitekturen zur Verfügung, wie beispielsweise die Prinzipien *Separation of Concerns*, *Information Hiding* und *konzeptionelle Integrität*⁸ (Posch et al. 2011; Reussner und Hasselbring 2009; Parnas 1972). An dieser Stelle wird beispielhaft das zentrale Prinzip *Separation of Concerns* betrachtet.

Wenn Entwicklungsteams das Prinzip des *Separation of Concerns* nutzen, dann betrachten sie die verschiedenen Zuständigkeiten innerhalb des Systems und weisen den Architekturelementen

⁸ Es sei angemerkt, dass Entwurfsprinzipien in der Regel nicht auf Softwarearchitekturen beschränkt sind, sondern sich beispielsweise auch für den Entwurf von Klassenstrukturen eignen.

jeweils eine Zuständigkeit zu (Wirfs-Brock und McKean 2007). Jedes Element sollte genau für eine Aufgabe zuständig sein, dies erleichtert u.a. die Wartbarkeit von Softwaresystemen, da sich der zu ändernde Quelltext leichter identifizieren lässt. Elemente mit einer klar definierten Aufgabe erfüllen nach Parnas das Qualitätskriterium der hohen Kohäsion (Parnas 1972). Welche Aufgaben und Aufgabenarten vorkommen können, entscheidet das Entwicklungsteam individuell.

Ähnliches lässt sich für alle Entwurfsprinzipien beobachten: Sie nennen zwar Aspekte, die im Entwurf zu beachten sind, geben jedoch keine konkrete Anleitung bezüglich der zu wählenden Architekturelemente, Schnittstellen und Beziehungen.

Für die Leistungsanforderungen haben sich bisher keine Entwurfsprinzipien etabliert, lediglich Verfahren zur Evaluation von Architekturentwürfen.

Hat ein Entwicklungsteam erfolgreich ein System mit einer guten Architektur realisiert, so muss das Team dennoch im nächsten Softwareprojekt erneut eine neue, individuelle Architektur entwickeln und evaluieren, ohne eine etablierte Möglichkeit, das im vorherigen Projekt erlangte Wissen über gut gewählte Architekturelemente, Schnittstellen und Beziehungen zu übertragen. Der Architekturentwurf beginnt üblicherweise in jedem Softwareprojekt erneut von Grund auf.

Eine Ausnahme stellen Projekte dar, in denen sich eine bereits in anderen Systemen verwendete Soll-Architektur wiederverwenden lässt. Wiederverwendbare Soll-Architekturen werden auch als *Referenzarchitektur* (Beneken 2009) oder als *Referenzmodell* (Bass et al. 2003, S. 25) bezeichnet. Hier sei darauf hingewiesen, dass mehrere Autoren den Begriff der Referenzarchitektur weiter fassen und auch für andere Arten von Architekturvorgaben verwenden (Beneken 2009; Siedersleben 2004). In dieser Arbeit wird der Begriff der Referenzarchitektur für wiederverwendbare Soll-Architektur verwendet.

Abbildung 8 skizzierte eine Referenzarchitektur aus dem Compilerbau. Genau wie die im vorherigen Abschnitt vorgestellten Soll-Architekturen, enthält auch diese Compiler-Architektur festgelegte Architekturelemente und Beziehungen. Die Implementation eines Compilers enthält Quelltext für den Scanner und für den Parser. Dabei referenziert der Quelltext des Scanners den Quelltext des Parsers, um Tokens und Attribute weiterzureichen, wie in der Vorlage festgelegt (Aho et al. 1988).

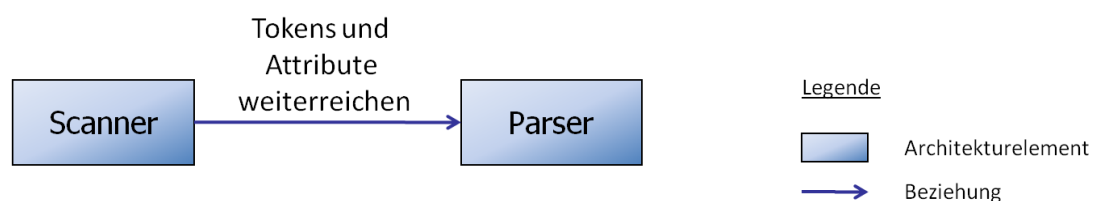


Abbildung 8: Ausschnitt aus der Referenzarchitektur für Compiler: Interaktion von Scanner und Parser (nach Aho et al. 1988, S. 70)

Genau wie Soll-Architekturen listen Referenzarchitekturen die konkreten Architekturelemente, Beziehungen und Schnittstellen auf. Im Gegensatz zum oben erläuterten, allgemeinen Architekturmodell befinden sich Referenzarchitekturen nicht auf einer Metaebene, sondern auf der selben Abstraktionsebene wie Softwarearchitekturen.

Das Einsatzfeld für Referenzarchitekturen ist beschränkt. Sie eignen sich nur für Systeme mit *den gleichen* fachlichen Anforderungen (Bass et al. 2003, S. 25). In den meisten Projekten im

Bereich interaktiver Anwendungssysteme werden jedoch neue Anforderungen umgesetzt, so hat beispielsweise jede Versicherung ihre eigenen Produkte, Abläufe und Arbeitsteilungen.

Die im nachfolgenden Kapitel betrachteten Architekturstile bieten Abhilfe. Sie eignen sich auch für Projekte mit neuen fachlichen Anforderungen. Sie befinden sich auf der Metaebene von Softwarearchitekturen.

Architekturstile stellen eine neuere Entwicklung im Bereich von Softwarearchitekturen dar. Sie bieten konkrete Anleitung für die Umsetzung von Entwurfsprinzipien und unterstützen so den Entwurf qualitativ hochwertiger Architekturen. Mit Architekturstilen brauchen Entwicklungsteams ihre Architektur nicht mehr von Grund auf neu zu entwerfen.

2.6 Zusammenfassung

In diesem Kapitel wurde der Begriff der *Softwarearchitektur* im Kontext von Architekturprüfungen beleuchtet. Begonnen wurde mit der Intention: Architekturen dienen in der Softwareentwicklung dazu, ein Gesamtbild auch großer Softwaresysteme zu ermöglichen. Architekturen helfen, Systeme im Ganzen zu planen und zu verstehen.

Anschließend wurden verschiedene Definitionen des Begriffs der Softwarearchitektur nebeneinandergestellt und die für diese Arbeit gewählte Definition 1-1 erläutert: Architekturen bestehen aus Architekturelementen, deren Schnittstellen und Beziehungen.

Um den Zusammenhang zwischen Quelltext- und Architekturebene zu erläutern wurden beide Ebenen formalisiert (Definitionen 2-1 bis 2-10). Der Zusammenhang ergibt sich über eine Zuordnung von Quelltext- und Architekturelementen. Es wurde deutlich, dass sich die Architektur dem Quelltext von Softwaresystemen nicht eindeutig entnehmen lässt. Über den Quelltext hinaus werden Zusatzinformationen benötigt, die im Kontext von Treueprüfungen in der Regel manuell angegeben werden.

Im Rahmen von Prüfungen auf Architekturtreue werden zwei Arten von Architektur unterschieden: die Soll- und die Ist-Architektur (Definitionen 2-11 und 2-12). Die Soll-Architektur gibt an, wie ein Softwaresystem aus Sicht der prüfenden Personen sein soll, sie stellt eine Architekturvorgabe dar. Die Ist-Architektur hingegen ist die in einem Softwaresystem implementierte Architektur. Prüfungen auf Architekturtreue vergleichen in der Regel die Soll- mit der Ist-Architektur.

Dieses Kapitel betrachtete Softwarearchitekturen, die auf Soll-Architekturen basieren. Der letzte Abschnitt erläuterte, dass Entwicklungsteams, die ihre Systeme basierend auf Soll-Architekturen entwickeln, für jedes neue System von Grund auf eine eigene Architektur entwickeln müssen. Eine Ausnahme bilden wiederverwendbare Soll-Architekturen, die jedoch nur für wenige fachliche Kontexte vorliegen, wie beispielsweise im Compilerbau. Das allgemeine Architekturmodell gibt keine konstruktive Anleitung für gute Architekturen, es besagt lediglich, dass Architekturen aus Elementen, Beziehungen und Schnittstellen bestehen. Als Maßstab für gute Architekturen stehen zwar verschiedene etablierte Entwurfsprinzipien zur Verfügung, jedoch müssen Entwicklungsteams für jede neue Architektur die Prinzipien individuell umsetzen.

Das folgende Kapitel wendet sich einer anderen Art von Architekturvorgaben zu: den Architekturstilen. Das Kapitel zeigt auf, inwiefern sich Stile von den oben betrachteten Soll-Architekturen unterscheiden und welche Vorteile der stilbasierte Architekturentwurf bietet.

3 Architekturstile

Dieses Kapitel widmet sich dem Konzept des Architekturstils. Abschnitt 3.1 diskutiert die in der Einleitung gegebene Definition von Architekturstil im Kontext anderer Definitionen, stellt alternative Bezeichnungen für Architekturstile vor und grenzt Stile gegen andere Arten von Architekturvorgaben ab.

Abschnitt 3.2 konkretisiert und formalisiert die in der Einleitung gegebene Stildefinition dieser Arbeit in Hinsicht auf den inneren Aufbau von Architekturstilen. Die vorgestellte Stildefinition basiert auf Literaturrecherchen und auf Abstraktionen zweier Stile, die im Rahmen dieser Arbeit untersucht wurden. Abschnitt 3.3 stellt diese Architekturstile vor.

Stile stellen ein wesentliches Mittel zur konstruktiven Qualitätssicherung von Softwarearchitekturen dar. Abschnitt 3.4 erläutert den Entwurf von Softwarearchitekturen im Vergleich zu einem rein auf Soll-Architekturen basierenden Entwurf. Der Abschnitt zeigt, wie Architekturstile den Entwurf von Softwarearchitekturen erleichtern und die resultierende Architektur verbessern.

3.1 Diskussion verschiedener Architekturstil-Definitionen

In der Literatur wird der Begriff des Architekturstils nicht einheitlich definiert. Auch werden für dasselbe Konzept unterschiedliche Begriffe verwendet.

Perry et al. übertragen den Begriff des Stils Anfang der neunziger Jahre von der Gebäude-Architektur hin zu Softwarearchitekturen (Perry und Wolf 1992). Sie schreiben:

„[...] style limits the kinds of design elements and their formal arrangements. That is, an architectural style constrains both the design elements and the formal relationships among the design elements.“ (Perry und Wolf 1992, S. 42).

Das bedeutet: Folgt eine Architektur einem Stil, so gibt der Stil eine Anleitung, indem er die Architektur limitiert. Der Stil macht Vorgaben für die Architekturelemente und legt fest, wie sie arrangiert werden dürfen. Perry et al. verwenden den Begriff des Stils noch sehr weit gefasst und auf verschiedenen Abstraktionsebenen. So ist aus ihrer Sicht sowohl die in Abbildung 8 dargestellte wiederverwendbare Soll-Architektur aus dem Compilerbau (Aho et al. 1988) ein Stil als auch die Verteilung von Prozessen auf Hardwarekomponenten. Sie sprechen von einem Kontinuum, da sie keine klare Trennlinie zwischen Architektur und Architekturstil sehen (Perry und Wolf 1992, S. 45).

Auch Garlan und Shaw haben in den neunziger Jahren maßgeblich zur Verbreitung des Stilbegriffs beigetragen. Sie definieren die Architektur eines Systems als Graph, mit Komponenten und Verbindungen (connectors) (Garlan und Shaw 1994, S. 5f). Darauf basierend definieren sie Architekturstil folgendermaßen:

„An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.“ (Garlan und Shaw 1994, S. 6).

Später präzisieren dieselben Autoren ihre Stildefinition so, dass das Vokabular von Stilen aus „component and connector types“ besteht (Shaw und Garlan 1996).

Die Definition von Shaw und Garlan sieht Stile auf einer höheren Abstraktionsebene als Architekturen. Architekturen sind *Exemplare* (instances) von Stilen. Ein Stil kann von mehreren Architekturen genutzt werden.

Anders als Perry et al. grenzen Shaw und Garlan mit ihrer Definition Stile deutlich von Referenzarchitekturen wie der Compiler-Architektur ab. Ein Stil im Sinne von Shaw und Garlan legt *nicht* fest, welche konkreten Architekturelemente (components) und Beziehungen innerhalb einer Architektur vorkommen sollen. Ein Stil gibt lediglich vor, welche *Typen* für die Architekturelemente und Beziehungen zur Verfügung stehen und er enthält *Regeln* (constraints) darüber, wie die Architekturelemente und Beziehungen abhängig von deren Typ kombiniert werden dürfen.

Der Client-Server-Architekturstil ist ein in der Literatur und Praxis oft verwendetes Beispiel für einen Architekturstil entsprechend der Definition von Garlan und Shaw (Shaw und Garlan 1996; Brügge und Dutoit 2004, S. 275). Eine Architektur, die entsprechend des Client-Server-Stils aufgebaut ist, kann mehrere Clients und mehrere Server enthalten. Abbildung 9 zeigt beispielhaft eine solche Architektur. Diese Architektur besteht aus 5 Architekturelementen, drei davon sind Clients, zwei sind Server. Die Architektur enthält Beziehungen, die von Clients hin zu Servern verlaufen.

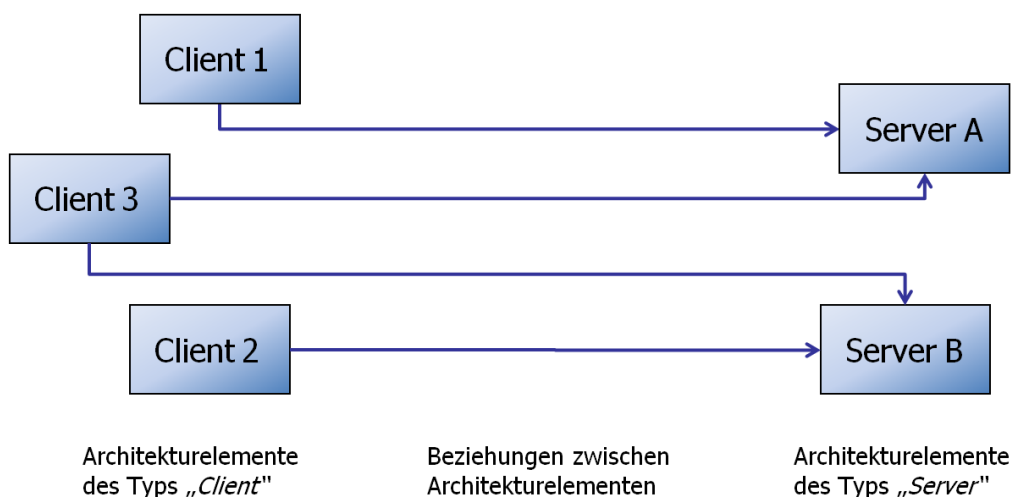


Abbildung 9: Eine nach dem Client-Server-Stil strukturierte Architektur

Abbildung 10 veranschaulicht den Client-Server-Architekturstil. Die Architekturelement-Typen in diesem Beispiel sind Client und Server, eine Regel ist, dass Clients auf Server zugreifen dürfen, nicht jedoch Server auf Clients.



Abbildung 10: Client-Server-Architekturstil

Weitere Beispiele für Stile, neben dem genannten Client-Server-Stil, sind die in dieser Arbeit betrachteten Stile Quasar (Siedersleben 2004) und WAM (Züllighoven 1998) sowie die Schichtenarchitektur und der Pipes-and-Filters-Stil (Shaw und Garlan 1996; Hofmeister et al. 2000).

Stile beziehen sich üblicherweise nicht auf alle Sichten der Softwarearchitektur. Der Client-Server-Stil beispielsweise kann sich auf die statische und die dynamische Sicht beziehen. Im Prinzip lässt sich für jede Architektursicht ein eigener Stil wählen (Kruchten 1995). Auf diese Weise hat die resultierende Softwarearchitektur mehrere Stile. Diese Arbeit betrachtet Stile für die statische Sicht.

Laut den meisten Definitionen umfassen Stile Vorgaben in Form von Regeln. Die Definitionen machen jedoch leicht unterschiedliche Aussagen über diese Regeln. Nach Garlan und Shaw beschränken Stile lediglich die Topologie der Architektur, indem sie Regeln (constraints) über die Komposition von Elementen und Beziehungen enthalten. Eine Regel, die Garlan und Shaw beispielhaft nennen, lautet: „Die Architektur soll zyklensfrei sein“ (Garlan und Shaw 1994, S. 6). Perry et al. hingegen definieren: „[...] an architectural style constrains both the design elements and the formal relationships among the design elements.“ (Perry und Wolf 1992, S. 42). Nach Perry et al. sind somit nicht nur die Beziehungen, sondern auch die Elemente Gegenstand der Regeln. Buschmann et al. machen dies noch deutlicher, indem sie schreiben, dass Stile Vorgaben machen für die *Komposition* von Elementen und Beziehungen und für die *Konstruktion* der Elemente (Buschmann et al. 1996, S. 394).

Stile definieren Typen für Architekturelemente, die in Architekturen enthaltenen Architekturelemente sind Exemplare dieser Typen. Dieses Verständnis ist weit verbreitet (Bass et al. 2003, S. 25; Züllighoven und Raasch 2006, S. 812; Hofmeister et al. 2000; Brügge und Dutoit 2004; Shaw und Garlan 1996; Kim und Garlan 2006; Lilienthal 2008)⁹. Dass Stile zusätzlich auch Typen für Beziehungen enthalten ist weniger verbreitet und wird von der Schule um Garlan und Shaw vertreten (Shaw und Garlan 1996; Kim und Garlan 2006). Viele in der Literatur vorgestellte Stile, auch einige der von Garlan und Shaw verwendeten Beispiele, enthalten lediglich Architekturelement-Typen. Dies gilt auch für den oben vorgestellten Client-Server-Stil.

Für Architekturstile finden sich in der Literatur weitere Bezeichnungen. Züllighoven verwendet den Begriff der *Modellarchitektur* (Züllighoven und Raasch 2006, S. 812). Andere Autoren benutzen auch für Stile den bereits oben im Kontext der Compiler-Architektur genannten Begriff der *Referenzarchitektur*, so Riebisch (Riebisch 2009) und Siedersleben (Siedersleben 2004), ebenso Hofmeister et al. sofern es sich um domänenspezifische Architekturen handelt, ansonsten verwenden sie den Begriff des Architekturstils (Hofmeister et al. 2000, S. 7). Einige Autoren benutzen den Begriff des Architekturmusters synonym zu Architekturstilen (Bass et al. 2003, S. 24f); jedoch findet sich der Begriff des Architekturmusters auch als Bezeichnung für andere Arten von Architekturvorgaben (Clements et al. 2002). Oft werden Stile als grundlegender betrachtet, als etwas, das sich in verschiedenen Kontexten einsetzen lässt,

⁹ Züllighoven und Raasch verwenden den Begriff *Art* anstelle von *Typ*, um ihre Definition deutlich vom Typbegriff in Programmiersprachen abzugrenzen. Gebräuchlicher ist es, von Typen zu sprechen. So hält es auch diese Arbeit.

wohingegen Muster eine spezifische Lösung für ein konkretes Problem darstellen (Taylor et al. 2009).

Der Begriff der Modellarchitektur ist nicht verbreitet, die Begriffe Referenzarchitektur und Architekturmuster werden mehrdeutig verwendet. Der Begriff des Architekturstil hingegen ist weit verbreitet und wird heutzutage nur noch für Architekturvorgaben in der Art des Client-Server-Beispiels benutzt (Brügge und Dutroit 2004; Bass et al. 2003, S. 24f; Riebisch 2009), nicht jedoch für Architekturvorgaben auf der Ebene von Soll-Architekturen wie die obige Compiler-Architektur. Aus diesem Grund verwendet diese Arbeit den Begriff des *Architekturstils*.

Für diese Arbeit ist es notwendig, die verschiedenen Arten von Architekturvorgaben klar voneinander abzugrenzen: Möchte ein Entwicklungsteam sein Softwaresystem werkzeuggestützt auf die Treue zu Architekturvorgaben prüfen, so spielt es eine große Rolle, auf welcher Ebene sich die Vorgaben befinden. Abhängig von dieser Ebene muss ein passender Prüfansatz gewählt werden.

Deshalb betrachtet diese Arbeit – anders als Perry et al. (Perry und Wolf 1992) – das Spektrum der verschiedenen Architekturvorgaben nicht als Kontinuum, sondern folgt der Definition von Shaw und Garlan (Shaw und Garlan 1996), nach der Stile sich auf einer Metaebene zu Architekturen befinden, indem sie *Typen* für Architekturelemente und deren Beziehungen festlegen und Regeln definieren. Dieses Verständnis wird mittlerweile auch von anderen Autoren geteilt (Bass et al. 2003, S. 25; Züllighoven und Raasch 2006, S. 812; Hofmeister et al. 2000; Brügge und Dutroit 2004).

3.2 Innerer Aufbau von Architekturstilen

Die hier erarbeitete Definition basiert auf Literaturrecherchen und auf Abstraktionen von realen Stilen. Für die Abstraktionen wurde im Rahmen dieser Arbeit der Quasar- und der WAM-Stil (siehe Abschnitt 3.3) detailliert untersucht und ihr prinzipieller Aufbau ermittelt. Die Erkenntnisse über den prinzipiellen Aufbau der untersuchten Stile fließen in die Stildefinition dieser Arbeit mit ein.

Diese Arbeit schließt sich der Auffassung von Perry et al. und Buschmann et al. an, dass Stile sowohl Vorgaben für die *Kompositionen* von Elementen und Beziehungen machen können, als auch für die *Konstruktion* der Elemente. Da die Interna der Elemente für Architekturen keine Rolle spielen, können sich die Konstruktions-Vorgaben ausschließlich auf die Schnittstellen der Elemente beziehen. Dieses Verständnis von Stilen wird gestützt durch die Untersuchungen des WAM- und des Quasar-Stils im Rahmen dieser Arbeit. Beide Stile beinhalten Regeln bezüglich der Komposition und der öffentlich sichtbaren Eigenschaften.

Die Kompositionsregeln von Elementen und Beziehungen bezeichnet diese Arbeit als Beziehungsregeln. Regeln, die Vorgaben für die öffentlich sichtbaren Eigenschaften machen, bezeichnet sie als Schnittstellenregeln. Beide Regeltypen machen Vorgaben abhängig von den Typen der betroffenen Architekturelemente und Beziehungen. Somit definieren sich Architekturstile im Rahmen dieser Arbeit wie folgt:

Definition 3-1: Architekturstil

Architekturstile bestehen aus Architekturelement-Typen, Beziehungstypen, Schnittstellenregeln und Beziehungsregeln.

Anhand der untersuchten Stile zeigte sich, dass sich Beziehungsregeln wiederum in verschiedene Regeltypen kategorisieren lassen, und zwar in vorgeschriebene Beziehungen (Gebots-

Beziehungsregeln), erlaubte Beziehungen (Kann-Beziehungsregeln), nicht erlaubte Beziehungen (Verbots-Beziehungsregeln) und Regeln zum Elementaufbau aus anderen Elementen (Enthält-Beziehungsregeln). Bezieht man diese Verfeinerung mit ein, so stellen sich Architekturstile in Mengennotation folgendermaßen dar:

Definition 3-2: Architekturstil, formal

Ein Architekturstil ist ein Tupel $\langle T_A, T_B, R_S, R_B \rangle$ mit

- T_A : die Menge der Architekturelement-Typen
- T_B : die Menge der Beziehungstypen
- R_S : die Menge der Schnittstellenregeln
- R_B : die Menge der Beziehungsregeln, diese unterteilt sich in folgende Regeln:
 - R_G : Gebots-Beziehungsregeln, für jeden Beziehungstyp T_B existiert eine Menge $R_G \subseteq T_A \times T_A$
 - R_K : Kann-Beziehungsregeln, für jeden Beziehungstyp T_B existiert eine Menge $R_K \subseteq T_A \times T_A$
 - R_V : Verbots-Beziehungsregeln, für jeden Beziehungstyp T_B existiert eine Menge $R_V \subseteq T_A \times T_A$
 - R_E : Enthält-Beziehungsregeln, auch als Elementaufbau-Regeln bezeichnet, $R_E \subseteq T_A \times T_A$

Die Beziehungsregeln werden auch kurz als Gebots-, Kann-, Verbots- und Enthältregeln bezeichnet.

Für jeden einzelnen Beziehungstyp gilt: die drei Mengen der Gebots-, Kann- und Verbotsregeln sind paarweise disjunkt: $R_G \cap R_K = R_G \cap R_V = R_K \cap R_V = \emptyset$. Dies ergibt sich aus der Bedeutung der Regeln, da sie sich gegenseitig ausschließen: zwischen zwei Architekturelementen der Typen A und B kann entweder nur gefordert werden, dass eine Beziehung bestehen muss, dass eine Beziehung bestehen darf oder dass keine Beziehung erlaubt ist¹⁰. Ferner gilt, dass für jedes Paar von Architekturelement-Typen entweder eine Gebots-, eine Kann- oder eine Verbotsregel existieren muss. Formal ausgedrückt gilt: $R_G \cup R_K \cup R_V = T_A \times T_A$. Folglich lässt sich jeden beliebigen dieser Regeltypen aus den anderen zwei errechnen. In der Praxis werden Architekturstile somit oft durch nur zwei dieser drei Regeltypen beschrieben, üblicherweise durch Gebots- und Verbotsregeln, wie in dem oben beschriebenen Client-Server-Stil und bei den im Folgenden vorgestellten Stilen WAM und Quasar.

Die Schnittstellenregeln und die Enthältregeln werden in Kapitel 6 in verschiedenen Ausbaustufen der stilbasierten Architekturprüfung adressiert. Die Ausbaustufen erläutern und erweitern die hier gegebene Formalisierung von Architekturstilen im Detail.

¹⁰ Dass die Kannregeln nicht die Gebotsregeln umfassen, ist eine Entwurfs-Entscheidung dieser Arbeit, die sich auf Erfahrungen aus den untersuchten Architekturstilen ergibt. Die deutsche Sprache ist hier nicht eindeutig. Würde man sich stattdessen entscheiden, dass die Kannregeln die Gebotsregeln umfassen, so ließe sich formal dasselbe ausdrücken, jedoch weniger elegant.

Definition 1-3 hat festgelegt, dass Architekturen, die nach einem Architekturstil entworfen werden, als stilbasierte Architekturen bezeichnet werden. Woraus bestehen stilbasierte Architekturen? Wie andere Architekturen auch enthalten sie Elemente, öffentlich sichtbaren Eigenschaften und Beziehungen (vergleiche Definition 1-1 von Softwarearchitektur). Darüber hinaus gilt für stilbasierte Architekturen: ihre Elemente und Beziehungen besitzen einen Typ:

Definition 3-3: Stilbasierte Architektur

Wird die Architektur eines Softwaresystems nach einem Architekturstil entworfen, so handelt es sich um eine stilbasierte Architektur. Stilbasierte Architekturen bestehen aus *Architekturelementen*, deren *Schnittstellen* und *Beziehungen*. Die Architekturelemente sind einem *Elementtyp* des Architekturstils zugeordnet. Die Beziehungen sind einem *Beziehungstyp* des Architekturstils zugeordnet.

Im Folgenden sei eine formale Definition stilbasierter Architekturen gegeben. Um die Definition übersichtlich zu halten, beinhaltet sie – wie viele Stildefinitionen – keine Beziehungstypen. Beziehungstypen würden die formale Definition an dieser Stelle unnötig verkomplizieren. Beziehungstypen werden in den nachfolgenden Kapiteln wo benötigt ergänzt. Dies betrifft auch Enthältbeziehungen.

Definition 3-4: stilbasierte Architektur, formal

Eine stilbasierte Architektur ist ein Tupel $\langle A, B_A, S_A, Z_{AS}, Z_{AT} \rangle$ mit

- A : die Menge der Architekturelemente
- B_A : Die Menge der Beziehungen zwischen Architekturelementen. Dabei gilt: $B_A \subseteq A \times A$
- S_A : die Menge der Schnittstellen der Architekturelemente.
- Z_{AS} : die Zuordnung zwischen Architekturelementen und deren Schnittstellen. Die Zuordnung ist eine Relation, für die gilt: $Z_{AS} \subseteq A \times S_A$
- Z_{AT} : die Zuordnung zwischen Architekturelementen und deren Typen. Die Zuordnung ist eine Relation, für die gilt: $Z_{AT} \subseteq A \times T_A$. Hierbei bezeichnet T_A die im zugehörigen Architekturstil enthaltene Menge der Architektur-element-Typen.

3.3 Fallbeispiele

Im Rahmen dieser Arbeit wurde beispielhaft der Aufbau des WAM-Stils (Züllighoven 1998) und des Quasar-Stils (Siedersleben 2004) anhand mehrerer Quellen untersucht: anhand der Literatur, bestehender Systeme und mittels Interviews.

Die Untersuchungen der beiden Stile dienten als Grundlage für verschiedene Aspekte dieser Arbeit:

- Die obige Definition des Stilbegriffs basiert nicht nur auf Literaturrecherchen, sondern auch auf praxisrelevanten Beispielen.

- Die Stile dienen als Basis für die beispielhafte Umsetzung in Kapitel 7.
- Der WAM-Stil sowie ein projekteigener Stil sind Grundlage der Fallbeispiele für die Machbarkeitsstudien in Kapitel 8.
- Ferner bieten die Stile die Grundlage für beispielhafte Illustrationen in verschiedenen Kapiteln.

Der WAM- und der Quasar-Stil eignen sich aus mehreren Gründen als Fallbeispiele: Beide Stile wurden und werden in der Praxis eingesetzt, so dass sie über Jahre reifen konnten. Beide Stile sind praxisrelevant, so dass genügend Softwaresysteme existieren, die sich für die Machbarkeitsnachweise nutzen ließen.

Beide Stile enthalten umfangreiche Typen und Regeln. Durch diesen Umfang bieten sie eine ausreichend große Basis für Abstraktionen darüber, welche Regeltypen in Stilen vorkommen und wie diese in realen Systemen umgesetzt werden. So war es möglich, den anhand der Literatur erarbeiteten Stilbegriff dieser Arbeit zusätzlich durch Fallbeispiele abzusichern und zu präzisieren. Vergleichbar umfangreiche Stile beschreiben Fowler (Fowler 2003) und Evans (Evans 2004).

Der WAM- und der Quasar-Stil waren vor Beginn der Forschungstätigkeiten für diese Arbeit lediglich in Prosa beschrieben. Im Rahmen der Arbeit wurden die Stile formalisiert. Dabei flossen die Ergebnisse von im Rahmen dieser Arbeit von der Autorin initiierten und betreuten Diplomarbeiten und einer Studienarbeit mit ein (Karstens 2005; Scharping 2008; Abraham 2006). Die Formalisierung zielt darauf, eine automatisierte Prüfung auf Stiltreue mit Hilfe von Prüfwerkzeugen zu ermöglichen. Darüber hinaus half die Formalisierung, die Struktur der Stile deutlich zu machen. Die dabei gewonnenen Erkenntnisse flossen in den Stilbegriff ein. Eine vollständige Liste der Elementtypen und Regeln der Stile findet sich im Anhang der Arbeit.

Die folgenden Abschnitte stellen die beiden Stile vor. Dabei werden beispielhaft einige Regeln der Stile genannt und formalisiert. Die gesamte Liste aller Regeln findet sich in Anhang B: Regellisten und Typen des WAM- und Quasar-Stils. Die Vorstellung der zwei Stile dient dazu, das Verständnis des Stilbegriffs und die in dieser Arbeit erarbeitete Formalisierung von Stilen anhand von Beispielen zu illustrieren.

3.3.1 Der Architekturstil des WAM-Ansatzes

Der Werkzeug&Material-Ansatz (kurz: WAM-Ansatz) wird seit über 20 Jahren in der Praxis und an Hochschulen eingesetzt und weiterentwickelt (Budde und Züllighoven 1990; Riehle und Züllighoven 1995; Züllighoven 1998; Züllighoven 2005). Der Ansatz ist eine Herangehens- und Sichtweise für die Entwicklung interaktiver Softwaresysteme. Er ist ausgerichtet auf anwendungsorientierte Systeme mit hoher Gebrauchsqualität. Die Systeme werden auf der Grundlage objektorientierter Entwurfs- und Konstruktionstechniken realisiert (Züllighoven 1998, S. 4). Der Ansatz unterstützt das Projektteam beim Vorgehen und bei der Konstruktion. Für die Konstruktion definiert der Ansatz einen eigenen Architekturstil (von Züllighoven als Modellarchitektur bezeichnet) (Züllighoven 2005, S. 281ff).

Im WAM-Stil stehen verschiedene Elementtypen zur Verfügung: Werkzeuge, Services, Automaten, Materialien und Fachwerte. Jeder Elementtyp ist für eine andere Aufgabe bestimmt. Werkzeuge dienen zur Benutzerinteraktion, sie werden als zusammengesetzte Architekturelemente realisiert. Sie können auf zwei Arten zusammengesetzt werden, in Form komplexer Werkzeuge oder als sogenannte MonoWerkzeuge. Komplexe Werkzeuge enthalten eine grafische Präsentation (GUI), den Zustand der Interaktion (IAK), die Funktionalität (FK) sowie ein Tool-Element. MonoWerkzeuge sind kompakter, als komplexe Werkzeuge. In Mono-

Werkzeugen werden die Aufgaben der IAK, der FK und des Tools von einem Architektur-
element des Typs MonoTool übernommen¹¹. Services stellen fachliche Dienstleistungen zur
Verfügung, Automaten übernehmen längere, mehrschrittige Aufgaben, Materialien repräsen-
tieren fachliche Gegenstände und werden Teil des Arbeitsergebnisses. Fachwerte repräsentieren
anwendungsfachliche Werte (Züllighoven 2005). Die verschiedenen Elementtypen helfen dabei,
die Funktionalität von Anwendungen in Architekturelemente zu strukturieren.

Abbildung 11 visualisiert den WAM-Stil. Sie zeigt die Elementtypen des Stils und erlaubte Be-
ziehungen. Die Grafik differenziert der Einfachheit halber nicht zwischen erlaubten und vorge-
schriebenen Beziehungen, diese Unterscheidung findet sich in der vollständigen Auflistung der
WAM-Regeln im Anhang der Arbeit. Einige Regeln sind in der Abbildung nicht dargestellt, um
die Darstellung übersichtlich zu halten. So darf ein Material beispielsweise andere Materialien
benutzen und Werkzeuge lassen sich in einer Subwerkzeugbeziehung schachteln. Die Kardinali-
tät der Beziehungen ist so festgelegt, dass die Beziehungen innerhalb eines Werkzeuges übli-
cherweise 1-zu-1 Beziehungen sind, sofern es sich nicht um geschachtelte Werkzeuge handelt.
Alle anderen Beziehungen sind 1-zu-n Beziehungen, so darf ein Werkzeug beispielsweise belie-
big viele Materialien benutzen, aber jedes Werkzeug soll mindestens ein Material benutzen.

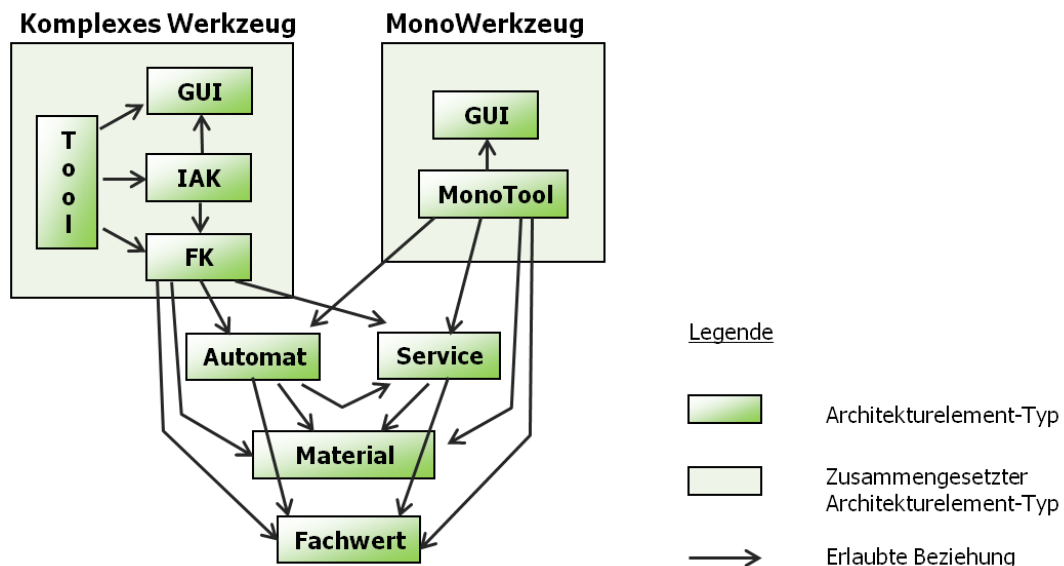


Abbildung 11: Schematische Darstellung des WAM-Architekturstils

Abbildung 12 zeigt eine Architektur, die dem WAM-Stil folgt. Das Beispiel ist angelehnt an ein
reales System zur Verwaltung von Zivildienstleistenden (ZDL). Die Beschriftung der Architek-
turelemente in dieser Darstellung gibt für jedes Architekturelement den Namen und den Typ an.

¹¹ Leserinnen und Leser könnten darüber stolpern, dass „Werkzeuge“ Elemente beinhalten, deren Typ als „*Tool“
bezeichnet wird. Es handelt sich trotz der ähnlichen Benennung um unterschiedliche Typen. Die hier gewählte Be-
nennung orientiert sich an den in der Praxis beobachteten Konventionen.

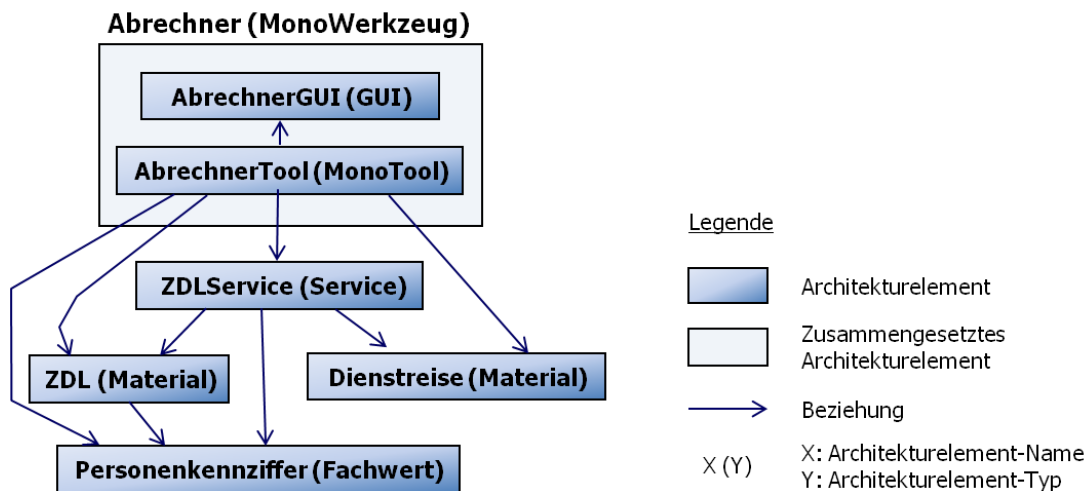


Abbildung 12: Eine WAM-Architektur

Anhand von Abbildung 11 und Abbildung 12 lässt sich beispielhaft der Zusammenhang zwischen einem Stil und einer stilbasierten Architektur nachvollziehen:

- Jedes Element der in Abbildung 12 dargestellten Architektur ist einem Elementtyp des Stils zugeordnet. Die Typen sind in Klammern dargestellt. Beispielsweise enthält die Architektur zwei Architekturelemente des Typs Material, und zwar die Architekturelemente Dienstreise und ZDL. Die in WAM-Architekturen zur Auswahl stehenden Typen finden sich in Abbildung 11.
- Neben den Typen umfassen Stile verschiedene Regeln. Beispielsweise besteht im WAM-Stil die Regel, dass Services auf Materialien zugreifen dürfen. Dies ist nachvollziehbar in Abbildung 11 durch den Pfeil zwischen Service und Material. Somit ist die Beziehung des ZDL-Services zur Dienstreise in Abbildung 12 erlaubt.

Basis für die Formalisierung des WAM-Stils im Rahmen dieser Arbeit sind die informelle Beschreibung im objektorientierten Konstruktionshandbuch (Züllighoven 1998; Züllighoven 2005) sowie Interviews mehrerer Softwarearchitektinnen und -Architekten, die nach dem WAM-Stil arbeiten. Die Formalisierung erfolgte in zwei Schritten, über eine semi-formale Auflistung der Architekturregeln im Rahmen der Forschungstätigkeit zur Diplomarbeit von Karstens (Karstens 2005; Becker-Pechau et al. 2006) bis zu der Formalisierung in mengentheoretischer Notation, die in dieser Arbeit vorgestellt wird und als Basis für Prüfwerkzeuge dient.

Abschnitt 3.2 hat eine formale Definition für Architekturstile eingeführt: ein Architekturstil ist ein Tupel $\langle T_A, T_B, R_S, R_B \rangle$, bestehend aus Architekturelement-Typen (T_A), Beziehungstypen (T_B), Schnittstellenregeln (R_S) und Beziehungsregeln (R_B) (siehe Definition 3-2, S. 35).

Nach dieser Formalisierung stellt sich die Menge der Architekturelement-Typen des WAM-Stils folgendermaßen dar:

$$T_A = \{Service, Automat, Material, Fachwert, komplexesWerkzeug, MonoWerkzeug, Tool, GUI, IAK, FK, MonoTool\}$$

Für den WAM-Stil sind drei verschiedene Beziehungstypen relevant:

- Ein Architekturelement kann andere Elemente *benutzen*.
- Zusammengesetzte Architekturelemente können aus anderen Elementen *aufgebaut* sein.
- *Vererbung* ist zwischen Architekturelementen unterschiedlichen Typs grundsätzlich nicht erlaubt.

Im Folgenden bezeichnet diese Arbeit die drei Beziehungstypen kurz als Benutzt-, Enthält- und Vererbungsbeziehungen, in Anlehnung an (Lilienthal 2008, S. 24). Als Oberbegriff für Beziehungen aller Art wird der Begriff Referenz verwendet, oder es wird davon gesprochen, dass ein Element ein anderes kennt.

Die Menge der gesamten Regeln des WAM-Stils ist umfangreich, hier sollen lediglich beispielhaft einige Regeln für die verschiedenen Regeltypen genannt werden. Eine vollständige und aktuelle Übersicht gibt Anhang B: Regellisten und Typen des WAM- und Quasar-Stils.

Als Beispiel für eine Schnittstellenregel dient folgende Aussage:

- Schnittstellenregel: „Funktionskomponenten dürfen keine Materialien zurückgeben“.

In mengentheoretischer Notation bedeutet dies, dass diese Regel ein Element der zugehörigen Regelmenge ist:

- „*Funktionskomponenten dürfen keine Materialien zurückgeben*“ $\in R_S$

Wie im vorherigen Abschnitt dargestellt lassen sich verschiedene Beziehungsregeltypen unterscheiden. Der WAM-Stil enthält eine Menge von Gebots-Beziehungsregeln (R_G), Verbots-Beziehungsregeln (R_V) und Enthält-Beziehungsregeln (R_E). Folgende Beziehungsregeln sind unter anderem in WAM enthalten.

- Gebots-Beziehungsregel: „Funktionskomponenten müssen Materialien kennen“.
- Verbots-Beziehungsregel: „Materialien dürfen keine Services kennen“
- Enthält-Beziehungsregel: „MonoWerkzeuge enthalten eine GUI“.

Dabei ist die Gebotsregel nach dem oben erläuterten Schema so zu verstehen, dass jede Funktionskomponente mindestens ein Material kennen muss. In umgekehrter Richtung macht die Regel keine Aussage, es darf somit auch Materialien geben, die von keiner Funktionskomponente gekannt werden. Ein Material darf von mehreren Funktionskomponenten genutzt werden.

In mengentheoretischer Notation dargestellt lauten die drei Regeln wie folgt:

- „*Funktionskomponenten müssen Materialien kennen*“ $\in R_G$
- „*Materialien dürfen keine Services kennen*“ $\in R_V$
- „*MonoWerkzeuge enthalten eine GUI*“ $\in R_E$

Schnittstellenregeln und Beziehungsregeln bleiben hier bewusst als natürlichsprachlicher Satz formuliert, da für das Verständnis an dieser Stelle keine weitere Formalisierung nötig ist. Die Diskussion, ob und inwieweit diese Regeln im Kontext einer werkzeuggestützten Architekturprüfung weiter formalisiert werden können und sollen, führen Kapitel 5 und 6.

3.3.2 Der Quasar-Architekturstil

Quasar ist der Architekturstil des Softwarehauses sd&m AG (mittlerweile aufgekauft durch Capgemini). Der Stil ist in der Praxis entstanden; er enthält das Erfahrungswissen aus einer Vielzahl bei sd&m durchgeführter Projekte. Der Stil ist auf betriebliche Informationssysteme ausgelegt. Er lässt sich – genau wie der WAM-Stil – in verschiedenen fachlichen Domänen einsetzen. sd&m-Research (die Forschungsabteilung der sd&m) hat den Quasar-Stil konsolidiert und dokumentiert (Siedersleben 2004, 2003b, 2003a).

Der Quasar-Architekturstil unterscheidet Architekturelement-Typen für den Anwendungskern von solchen für die GUI-Engine. Abbildung 13 zeigt die verschiedenen Elementtypen und die erlaubten Beziehungen.

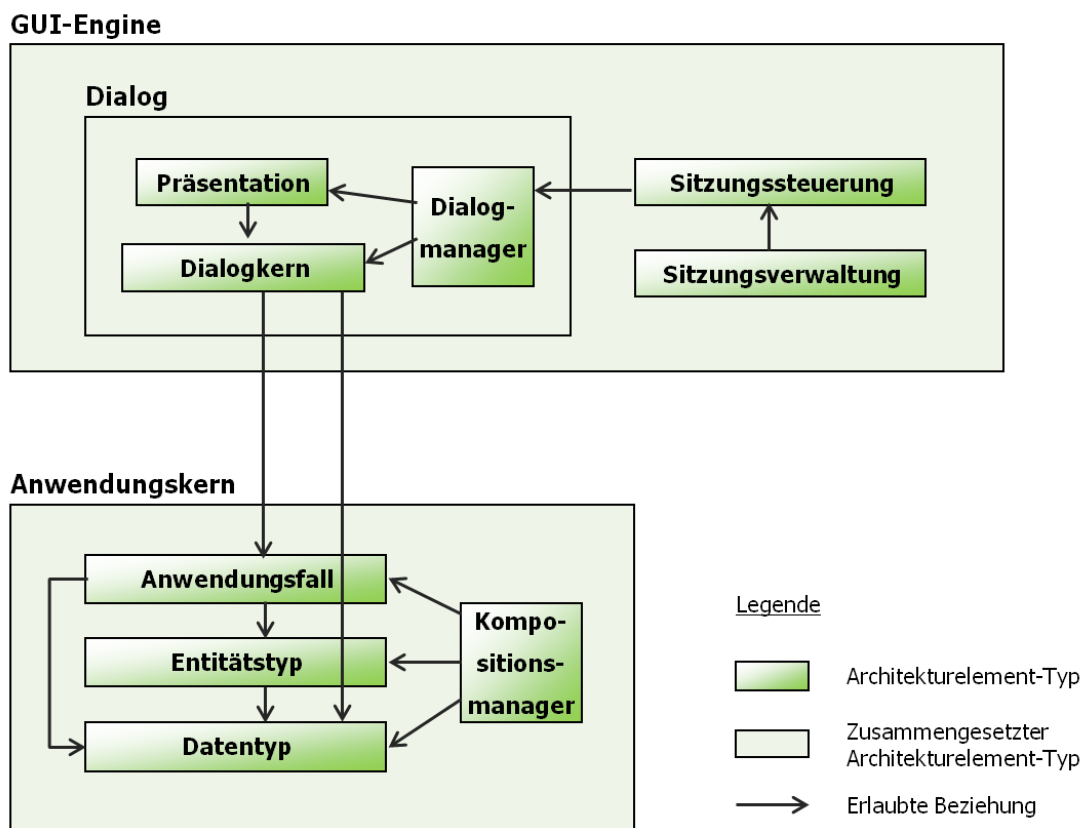


Abbildung 13: Der Quasar-Architekturstil

Der *Anwendungskern* modelliert den anwendungsfachlichen Anteil von Anwendungen, er ist unabhängig von der gewählten Technik (Siedersleben 2004, S. 166) (beispielsweise enthält er keine Referenzen auf ein GUI-Rahmenwerk oder auf einen Transaktionsmanager). Er ist ein zusammengesetztes Architekturelement, bestehend aus Anwendungsfällen, Entitätstypen, Datentypen und Kompositionsmanagern.

Entitätstypen modellieren anwendungsfachliche Konzepte wie das Konto, das Buch, den Kunden, den Artikel, die Bestellung. Ihre Exemplare (kurz: Entitäten) sind Objekte mit eigener Identität, sie besitzen einen fachlichen Schlüssel. Entitäten sind persistent und unterliegen einem

Lebenszyklus, sie lassen sich anlegen, ändern und löschen (Siedersleben 2004, S. 167, 2004, S. 176, 2003a, S. 18).

Datentypen dienen zur Konstruktion von Entitätstypen. Entitätstypen werden durch Attribute beschrieben, beispielsweise hat ein Kunde eine Bankverbindung. Zu jedem Attribut gehört ein Datentyp. Für anwendungsunabhängige Attribute können die in Programmiersprachen definierten Datentypen verwendet werden (beispielsweise bietet Java den Datentyp String für Zeichenketten). Für fachliche Attribute, wie beispielsweise die Versicherungsart einer Krankenversicherung, die ISBN eines Buchs oder eine Bankverbindung, enthält der Anwendungskern die zugehörigen Datentypen. Um deutlich zu machen, dass es sich um anwendungsfachliche Datentypen handelt, bezeichnet Quasar die Datentypen des Anwendungskerns auch als A-Datentypen (Siedersleben 2004, S. 167, 2004, S. 176, 2003a, S. 17).

Anwendungsfälle (kurz: A-Fälle) entsprechen den Use-Cases der UML. Sie stellen die „Kommandos“ des Anwendungskerns zur Verfügung (Siedersleben 2004, S. 167). Beispiele für A-Fälle sind „Buch ausleihen“, „Geld abheben“ oder „Flug stornieren“. Anwendungsfälle greifen auf Entitätstypen und Datentypen zu, beispielsweise benötigt der Anwendungsfall „Buch ausleihen“ den Entitätstyp Kunde und die dazugehörigen Datentypen wie Kundennummer.

Kompositionsmanager setzen eine Anwendung aus mehreren Architekturelementen zusammen (Siedersleben 2004, S. 61). Es ist beispielsweise möglich, dass die Anwendungsfälle keine direkten Referenzen auf Entitätstypen besitzen, sondern lediglich passende Schnittstellen referenzieren. In diesem Fall ist der zugehörige Kompositionsmanager dafür zuständig, die verschiedenen Architekturelemente zusammenzusetzen. Quasar-Anwendungen haben mindestens einen Kompositionsmanager.

Die *GUI-Engine* ist wie der Anwendungskern ein zusammengesetzter Architekturelement-Typ. Die in GUI-Engines enthaltenen Architekturelemente sind für grafische Benutzungsschnittstellen zuständig. Im Gegensatz zum Anwendungskern sind GUI-Engines abhängig von ihrem Kontext, sie kennen die verwendete GUI-Bibliothek¹². GUI-Engines vermitteln zwischen dem Anwendungskern und der GUI-Bibliothek (Siedersleben 2004, S. 239).

GUI-Engines enthalten mehrere *Dialoge*. Architekturelemente des Typs Dialog stellen den Benutzerinnen und Benutzern zusammenhängende Daten und Funktionen zur Verfügung. Jeder Dialog unterstützt einen oder mehrere Anwendungsfälle und präsentiert sich in geeigneter Form, beispielsweise durch ein Fenster oder eine HTML-Seite (Siedersleben 2004, S. 263). Dialoge sind zusammengesetzte Architekturelemente, sie bestehen aus Präsentation, Dialogkern und Dialogmanager.

Die *Präsentation* greift direkt auf die verwendete GUI-Bibliothek zu (in Java beispielsweise auf Swing oder SWT). Sie ist für die visuelle Darstellung von Informationen am Bildschirm und für die Interaktion mit den Benutzerinnen und den Benutzern zuständig (Siedersleben 2003b, S. 30).

Der *Dialogkern* übernimmt nicht-visuelle Aufgaben des Dialogs. Er ist verantwortlich dafür, welche Daten der Dialog darstellt und wie der Dialog auf Aktionen von Benutzerinnen und Benutzern reagiert. Die GUI-Bibliothek kennt er nicht (Siedersleben 2003b, S. 30).

Der *Dialogmanager* ist der Kompositionsmanager (s.o.) von Dialogen (Siedersleben 2004, S. 243). Er bringt Präsentation und Dialogkern zusammen, so dass sie sich nicht direkt zu referenzieren brauchen. Ein Dialogmanager kann für mehrere Dialoge zuständig sein.

Die *Sitzungssteuerung* verwaltet die Sitzungen verschiedener Benutzerinnen und Benutzer. Jedes Exemplar einer Sitzungssteuerung ist zur Laufzeit für genau eine Sitzung zuständig. Das

¹² Die GUI-Bibliothek ist in obiger Abbildung nicht dargestellt, sie stellt keinen Elementtyp dar, sondern eine Systemexterne Bibliothek. Zur Verwendung externer Bibliotheken in Softwarearchitekturen siehe auch Becker-Pechau und Benniscke 2007.

Exemplar referenziert die Dialog-Exemplare der Benutzerin oder des Benutzers, indem es eine Referenz auf den in jedem Dialog-Exemplar enthaltenen Dialogmanager hält (Siedersleben 2004, S. 242).

Die *Sitzungsverwaltung* kontrolliert alle Sitzungen. Sie kennt alle Exemplare der Sitzungssteuerung und verwaltet alle Ressourcen. Von der Sitzungsverwaltung existiert zur Laufzeit nur ein Exemplar (Siedersleben 2004, S. 242).

Im Rahmen der Forschungstätigkeiten für diese Arbeit hat Abraham in einer Studienarbeit ein beispielhaftes Quasar-System entworfen und implementiert (Abraham 2006). Als fachliches Fallbeispiel wählte er ein System zur Verwaltung von Baufinanzierungen. Abbildung 14 zeigt die (vereinfachte) Architektur dieses Systems. In der Abbildung finden sich Elemente aller oben genannter Elementtypen. Die oben dargestellte GUI-Engine fasst alle GUI-bezogenen Elemente zusammen, während der unten dargestellte Anwendungskern die rein fachlichen Elemente enthält. Die dargestellte Architektur enthält zwei Dialoge, BonitätPrüfenD und BaufinanzierungsAnlegenD. Der Anwendungskern enthält verschiedene Anwendungsfälle, Entitätstypen und Datentypen. Die Dialogkerne greifen auf die Anwendungsfälle im Anwendungskern zu. Um die Daten des Anwendungskerns an der Oberfläche darzustellen, referenzieren die Dialogkerne zusätzlich einige Entitätstypen und Datentypen (in der Grafik nicht dargestellt).

Das Zusammenspiel von Architekturelementen ist in Quasar, genau wie in WAM, über Regeln festgelegt. Diese Arbeit formalisiert die Regeln des Quasar-Stils, basierend auf Veröffentlichungen von Siedersleben (Siedersleben 2003b, 2003a, 2004). Wie den WAM-Stil, formalisiert diese Arbeit auch den Quasar-Stil in zwei Schritten. Eine semi-formale Auflistung der Architekturregeln findet sich in Anhang B: Regellisten und Typen des WAM- und Quasar-Stils. Diese Liste ist die Basis für eine weitergehende Formalisierung in mengentheoretische Notation. Diese mengentheoretische Formalisierung dient als Eingabe für Prüfwerkzeuge.

Im Folgenden werden beispielhaft ausgewählte Regeln verschiedener Regeltypen vorgestellt. Die Menge der gesamten Regeln des Quasar-Stils findet sich in Anhang B: Regellisten und Typen des WAM- und Quasar-Stils.

Beispielhafte Regeln:

- Schnittstellenregel: „Datentypen enthalten keine Setter“.

In mengentheoretischer Notation bedeutet dies, dass diese Regel ein Element der zugehörigen Regelmenge ist:

- „*Datentypen enthalten keine Setter*“ $\in R_S$

Der Quasar-Stil definiert Gebots-Beziehungsregeln (R_G), Verbots-Beziehungsregeln (R_V) und Enthält-Beziehungsregeln (R_E). Beispiele für Beziehungsregeln:

- Gebots-Beziehungsregel: „Dialoge müssen Anwendungsfälle kennen“.
- Verbots-Beziehungsregel: „Dialogkerne dürfen keine Präsentation kennen“
- Enthält-Beziehungsregel: „Dialoge enthalten eine Präsentation“.

In mengentheoretischer Notation dargestellt bedeutet dies:

- „*Dialoge müssen Anwendungsfälle kennen*“ $\in R_G$
- „*Dialogkerne dürfen keine Präsentation kennen*“ $\in R_V$

- „Dialoge enthalten eine Präsentation“ $\in R_E$

Der hier gewählte Formalisierungsgrad ist darauf ausgerichtet, den Quasar-Stil sowie exemplarisch ausgewählte Regeln vorzustellen. Um Systeme werkzeuggestützt auf Verstöße gegen diese Regeln zu prüfen, ist eine weitere Formalisierung nötig, diese findet sich in den Kapiteln 5 und 6.

Die GUI-Engine (GUI-Engine)

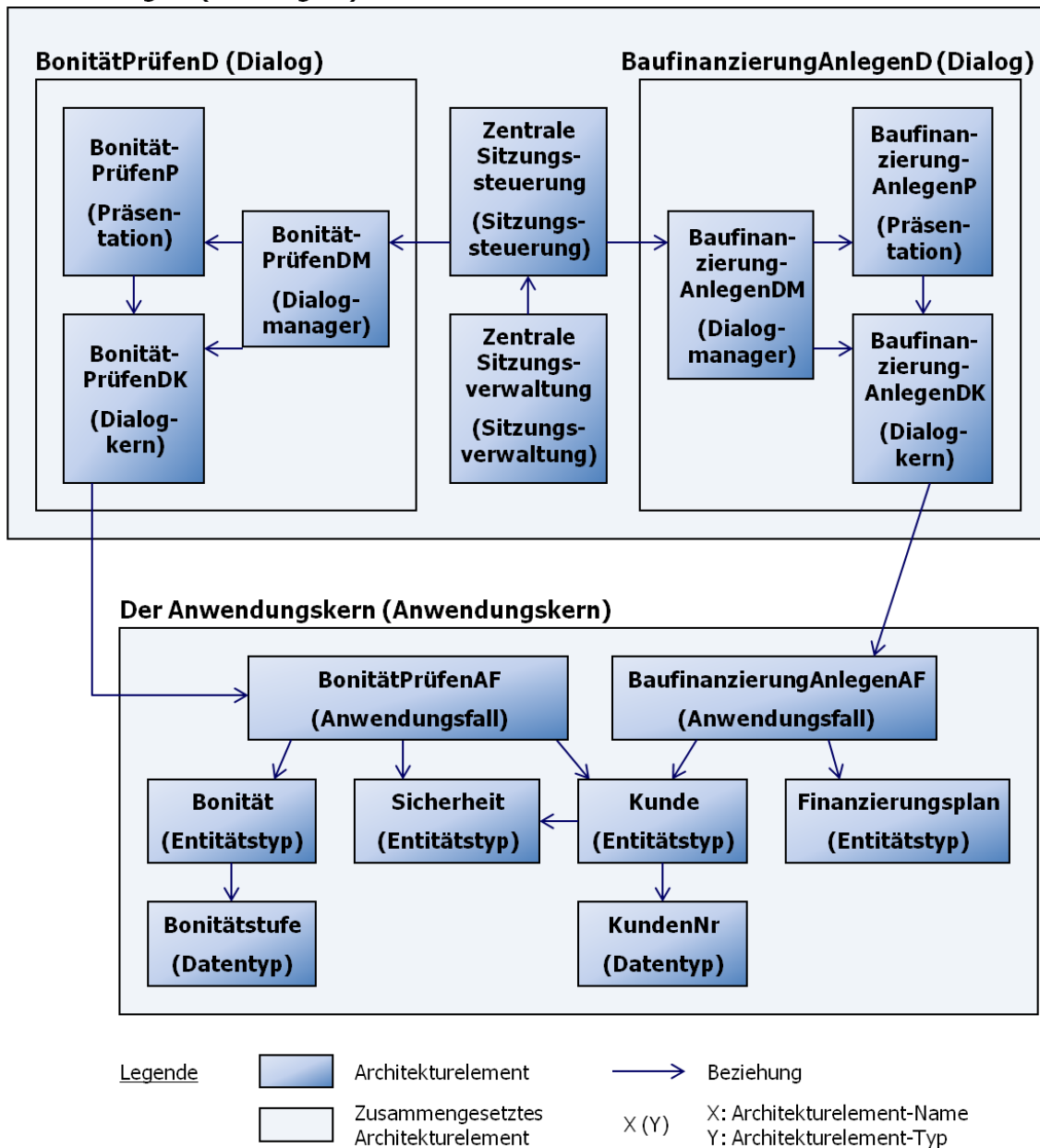


Abbildung 14: Eine Quasar-Architektur

3.4 Vorteile des stilbasierten Architekturentwurfs

An dieser Stelle werden nun – basierend auf den vorgestellten Definitionen und Stil-Beispielen – die Vorteile des stilbasierten Architekturentwurfs betrachtet.

Das allgemeine Konzept von Softwarearchitekturen gibt vor, dass es Elemente, Beziehungen und Schnittstellen geben muss. Zur Erinnerung zeigt Abbildung 15 das oben vorgestellte allgemeine Metamodell für Softwarearchitekturen.

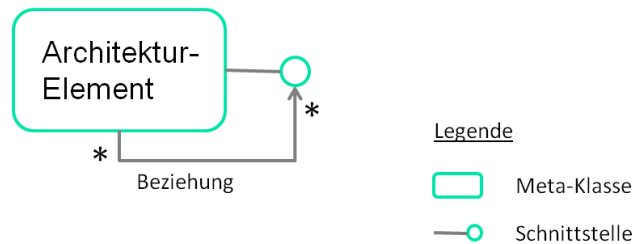


Abbildung 15: Das allgemeine Metamodell für Architekturen

Architekturstile hingegen bieten eine reichhaltigere und konkretere Anleitung für den Architekturentwurf als das allgemeine Architektur-Metamodell:

Stile umfassen – im Gegensatz zum allgemeinen Architektur-Metamodell – eine Menge an wohlverstandenen und bewährten Architekturelement-Typen (Perry und Wolf 1992, S. 43). Wenn Entwicklungsteams entscheiden, aus welchen Elementen ihre Architektur bestehen soll, so können beim Architekturentwurf aus dieser Menge auswählen. Wie umfangreich diese Menge sein kann, wurde oben beispielhaft anhand der Stile Quasar und WAM demonstriert (siehe Abbildung 11, S. 38 und Abbildung 13, S. 41).

Stile können umfangreichere Abstraktionen als das allgemeine Architektur-Metamodell bieten, da sie auf bestimmte Systemarten und ausgewählte Sichten ausgerichtet sind (Kim und Garlan 2006), beispielsweise auf interaktive Anwendungssysteme (Züllighoven 2005), Unternehmenssoftware (Siedersleben 2004) oder Webanwendungen (Mesbah und van Deursen 2007). Das allgemeine Architektur-Metamodell hingegen wird für *alle* Systemarten und *alle* Architektursichten verwendet und abstrahiert daher viel stärker als Architekturstile.

Architekturstile befreien Entwicklungsteam davon, neue Architekturen von Grund auf entwerfen zu müssen. Mit Stilen lässt sich Erfahrungswissen aus vorherigen Projekten auf neue Architekturen übertragen (Garlan und Shaw 1994), sie bieten Standard-Strukturen für ähnliche Probleme (Bass et al. 2003, 2012).

Stile unterstützen die Umsetzung von Entwurfsprinzipien

Wie oben dargestellt, beachten gute Softwarearchitekturen verschiedene Entwurfsprinzipien, wie Separation of Concerns, Information Hiding und konzeptionelle Integrität (Posch et al. 2011; Reussner und Hasselbring 2009; Parnas 1972). Wenn Entwicklungsteams Architekturen ohne Architekturstil entwerfen, so müssen sie die konkrete Umsetzung der Prinzipien für jedes System individuell erstellen und evaluieren. Dieses Vorgehen ist aufwändig und fehlerträchtig.

Stile hingegen können die Umsetzung der Architekturprinzipien durch eine konkretere Anleitung unterstützen. Dies sei im Folgenden am Beispiel des Prinzips Separation of Concerns dargestellt:

Anders als das im allgemeinen Architekturmodell definierte Konzept des Architekturelements bieten die von Stilen definierten Architekturelement-Typen eine Mittel-Zweck-Relation (Kim und Garlan 2006):

Jeder von Stilen definierte Architekturelement-Typ ist für eine bestimmte *Aufgabenart* vorgesehen.

Mit Hilfe der Mittel-Zweck-Relation können Entwicklungsteams ihre fachlichen Anforderungen auf die verschiedenen Aufgabenarten und damit auf die Architekturelement-Typen verteilen. Es ist nicht mehr nötig, dass Entwicklungsteams sich eigene Aufgabenarten für ihr System überlegen, stattdessen können sie bewährte Aufgabenarten wiederverwenden.

Für den WAM-Stil gilt, dass Werkzeuge zur Benutzerinteraktion dienen, mit ihnen können Benutzerinnen und Benutzer die Materialien erstellen und bearbeiten. Ein kurzes Beispiel: Es soll ein System für die Verwaltung von Patientendaten erstellt werden. Dieses System soll unter anderem folgende Anforderungen erfüllen:

- (a) Ärzte können während der Visite Einträge in Patientenmappen vornehmen
- (b) bei der Aufnahme neuer Patientinnen und Patienten werden ihre Daten aufgenommen und in eine neue Mappe eingefügt.

Wird dieses System nach dem WAM-Stil entwickelt, so ist klar: beide Anforderungen umfassen eine Benutzer-Interaktion mit dem Softwaresystem. Solche Interaktionen werden im WAM-Stil mit Hilfe von Werkzeugen realisiert. Das resultierende System wird voraussichtlich zwei Werkzeuge enthalten, einen Mappenbearbeiter für Anforderung (a) und einen Patientenaufnehmer für Anforderung (b). Ferner wird das System unter anderem das Material Patientenmappe enthalten. Beide Werkzeuge bearbeiten unter anderem dieses Material. Da es sich in diesem Beispiel um kleine Werkzeuge handelt, entscheidet sich das Team für MonoWerkzeuge. Abbildung 16 zeigt den resultierenden Architekturausschnitt.

Dieses Beispiel dient dazu, die Idee des stilbasierten Architekturentwurfs vermitteln und ist stark verkürzt. Der Entwicklungsprozess des WAM-Ansatzes findet sich ausführlich in dem objektorientierten Konstruktionshandbuch (Züllighoven 1998).

Üblicherweise lassen sich die meisten Anforderungen gut den verschiedenen Aufgabenarten von Stilen zuordnen, da Stile auf die passende Systemart ausgerichtet sind. Nach Perry und Wolf kann das Entwicklungsteam eines stilbasierten Systems aus den wohlbekanntesten und verstandenen Elementen des Stils wählen und gewinnt so die Freiheit, sich auf eventuell benötigte Elemente zu konzentrieren, die individuell maßgefertigt werden müssen (Perry und Wolf 1992, S. 43).

Das Architekturprinzip der Information Hiding / Schnittstellenbildung wird durch Stile konkretisiert, indem sie Schnittstellenregeln anbieten. Diese Regeln beschreiben abhängig von dem jeweiligen Elementtyp, wie die Schnittstelle gestaltet sein soll.

Das Architekturprinzip der konzeptionellen Integrität besagt, dass Architekturen einheitlich strukturiert sein sollten. Stile unterstützen die Einheitlichkeit, indem sie vorgeben, wie Architekturen im Prinzip strukturiert sein sollen. Wird beispielsweise der WAM-Stil eingesetzt, so ist klar, dass *alle* Werkzeuge auf Materialien arbeiten, und dass *kein* Werkzeug direkt auf die Datenbank zugreift, da Persistenz nicht zur Aufgabe von Werkzeugen gehört.

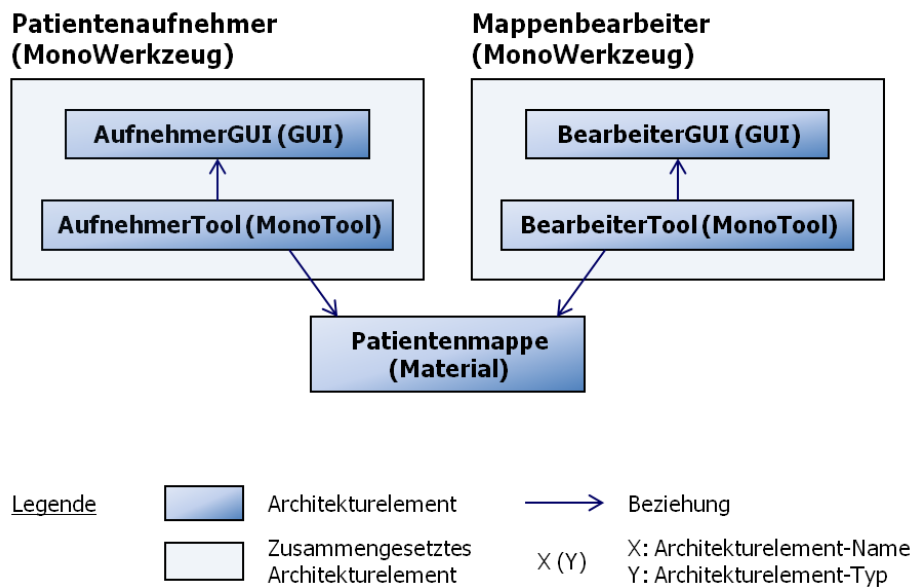


Abbildung 16: Ausschnitt aus einer stilbasierten Architektur nach WAM

Untersuchungen haben gezeigt: Architekturstile stellen in der Praxis das *wichtigste* Mittel dar, um die Komplexität großer Softwarearchitekturen handhabbar zu machen (Lilienthal 2008). Selbst Architekturen mit einem individuellen, projekteigenen Stil sind besser zu handhaben als solche ohne Architekturstil. Stilbasierte Architekturen erfüllen Qualitätskriterien, wie das Prinzip der konzeptionellen Integrität, üblicherweise besser als Architekturen ohne Architekturstil (Lilienthal 2008).

Stile verbessern die Verständlichkeit von Softwaresystemen

Architekturstile bieten nicht nur im *Entwurf* große Vorteile, sie unterstützen Entwicklerinnen und Entwickler auch dabei, Architekturen zu *verstehen*. Besser verständliche Architekturen lassen sich leichter ändern, erweitern und warten.

Stile unterstützen die *Verständlichkeit* von Softwaresystemen auf mehrfache Weise:

Stilbasierte Architekturen sind durchgängig nach denselben Prinzipien strukturiert, die verschiedenen Systemteile sind gleichartig aufgebaut. Dadurch unterstützen Stile konsistente und verständliche Architekturen. Die Konsistenz und Verständlichkeit stilbasierter Architekturen sorgen dafür, dass sich die Systeme leichter ändern und erweitern lassen, neue Projektmitglieder können sich leichter einarbeiten (Garlan et al. 1994, S. 177). Dieser Vorteil besteht selbst dann, wenn ein Projektteam einen eigenen Architekturstil für sein System entwirft. Wird ein bereits bestehender Stil wiederverwendet, so kommt der Vorteil hinzu, dass sich neue Projektmitglieder, die bereits andere Systeme des gleichen Stils entwickelt haben, besonders schnell zurechtfinden.

Ohne Architekturstil ist es schwierig und umständlich, über *Architekturen* zu sprechen, da auf dieser Ebene keine angemessene Sprache zur Verfügung steht. Dies erschwert den Entwurf, die Evolution und das Verstehen von Softwarearchitekturen. Ohne Architekturstil stehen als Standard-Vokabeln lediglich die Quelltextelementtypen zur Verfügung, die durch die gewählte Programmiersprache gegeben sind. So kann man beispielsweise in der Programmiersprache Java über Klassen, Interfaces und Packages sprechen. Diese Abstraktionen befinden sich jedoch

auf der Ebene des Quelltextes und nicht auf der abstrakteren Architekturebene. Auf Architekturebene stehen als Standard-Vokabeln lediglich die Konzepte Architekturelement, Beziehung und Schnittstelle zur Verfügung. Diese Konzepte werden üblicherweise nicht weiter differenziert.

Stile hingegen bieten eine *Sprache* auf der Ebene von Softwarearchitekturen (Garlan und Shaw 1994; Kim und Garlan 2006). Ihre Element- und Beziehungstypen dienen Entwicklungsteams als Vokabeln für die Kommunikation über stilbasierte Architekturen.

So kann in Entwicklungsteams, die mit dem WAM-Stil vertraut sind, über ein WAM-basiertes System im Medizinbereich beispielsweise gesagt werden: „dieses *Werkzeug* arbeitet auf dem *Material* Patientenakte und greift auf den *Diagnose-Service* zu“ (siehe Abbildung 17).

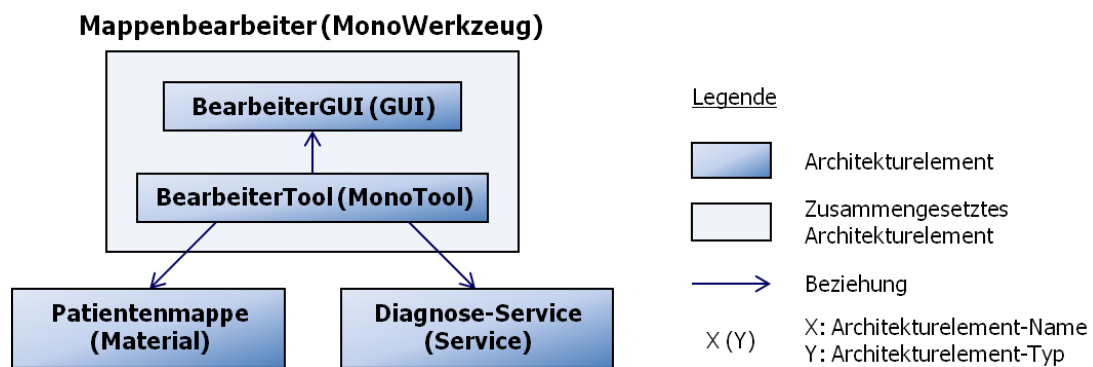


Abbildung 17: Architektur-Beispiel

Die Person, die diesen Satz hört, kann ihn mit Hilfe ihres Wissens über die Elementtypen interpretieren. Die typischen Aufgaben von Werkzeugen, Materialien und Services sind klar, und so braucht in der Kommunikation nicht mehr jede Klasse und jedes Architekturelement einzeln erklärt zu werden.

Das Vokabular von Stilen ist semantisch reichhaltiger: die Vokabel *Werkzeug* beispielsweise sagt deutlich mehr über die zugehörigen Architekturelemente aus als lediglich die Vokabeln *Architekturelement* oder *Package*. Diese reichhaltige Sprache wird dadurch ermöglicht, dass Stile auf bestimmte Systemarten und Sichten ausgerichtet sind. Das Vokabular passt zur Systemart und ist dadurch konkreter und ausdrucksmächtiger.

Wird ein Stil in mehreren Systemen verwendet, so entsteht eine Familie von Systemen, die nach den gleichen Prinzipien strukturiert ist (Garlan und Shaw 1994, S. 5f). Durch das gemeinsame Vokabular und die gleichartige Strukturierung können sich die Entwicklerinnen und Entwickler auch über die Projektgrenzen hinweg über ihre Systeme austauschen.

Zusammenfassend sei festgehalten: Architekturstile unterstützen den *Entwurf* und die *Verständlichkeit* von Softwaresystemen:

- Für den *Entwurf* bieten Stile die Möglichkeit, auch bei Systemen mit neuen fachlichen Anforderungen Erfahrungen über gut gewählte Architekturelemente, Schnittstellen und Beziehungen wiederzuverwenden. So entstehen qualitativ hochwertigere Architekturen. Der Architekturentwurf wird weniger aufwändig und somit kostengünstiger.

- Stile unterstützen die *Verständlichkeit* und damit die Änderbarkeit und Wartbarkeit von Softwaresystemen, indem sie für eine durchgängige Strukturierung von Architekturen sorgen und eine Sprache zur Verfügung stellen, mit deren Hilfe Entwicklungsteams auf Architekturebene über ihre Systeme kommunizieren können. So wird die Kommunikation über Architekturen effektiver. Sind Entwicklerinnen und Entwickler mit einem Stil vertraut, so können sie sich in Systeme desselben Stils schneller einarbeiten und sind schneller produktiv.

3.5 Zusammenfassung

Dieses Kapitel hat eine spezielle Art von Architekturvorgaben thematisiert: die *Architekturstile*. Zu Beginn wurden verschiedene Definitionen des Architekturstil-Begriffs sowie angrenzender Begriffe vorgestellt und diskutiert. Die Stildefinition der Einleitung wurde erweitert und um eine formale Definition ergänzt: Stile bestehen aus Architekturelement-Typen, Beziehungstypen, Beziehungsregeln und Schnittstellenregeln.

Architekturen, die auf einem Stil basieren, bezeichnet diese Arbeit als *stilbasierte Architekturen*. Stilbasierte Architekturen haben die Besonderheit, dass sie nach den Regeln eines Stils strukturiert sind und dass ihre Elemente und Beziehungen den Element- und Beziehungstypen ihres Stils zugeordnet sind.

Die Definitionen von Architekturstil sowie stilbasierter Architektur basieren auf Literaturrecherchen und auf zwei etablierten, praxisrelevanten Architekturstilen: WAM und Quasar. Die Stile wurden im Rahmen dieser Arbeit detailliert auf ihre innere Struktur hin untersucht.

Es wurde argumentiert, dass Entwicklungsteams auf vielfältige Weise von Architekturstilen profitieren: Architekturstile bieten eine konkrete, in der Praxis bewährte Anleitung für den Architekturentwurf. So brauchen Entwicklungsteams ihre Architekturen nicht mehr von Grund auf neu zu entwerfen. Architekturstile sorgen darüber hinaus für verständlichere Architekturen und bieten eine semantisch reichhaltige Sprache für die Kommunikation über Architekturen. Diese Vorteile führen dazu, dass Architekturstile das wichtigste Mittel darstellen, um große, komplexe Architekturen zu handhaben (Lilienthal 2008). Zukünftig wird die Bedeutung von Architekturstilen voraussichtlich noch zunehmen, da im Laufe der Zeit mehr erprobte Stile entstehen und da heutige Systeme immer umfangreicher werden.

Auf Basis der vorgestellten begrifflichen und konzeptuellen Grundlagen gibt das anschließende Kapitel einen Überblick über die bisherigen Ansätze zur Prüfung auf Architekturtreue und erläutert, an welchen Stellen diese Ansätze zu kurz greifen, als dass sich mit ihnen die Treue zu *Architekturstilen* prüfen ließe.

4 Prüfungen auf Architekturtreue

Dieses Kapitel gibt einen Überblick über existierende Ansätze zur statischen Prüfung auf Architekturtreue, auch als statische Prüfungen auf Architekturkonformanz bezeichnet. Die vorliegende Arbeit ist diesen Ansätzen zuzuordnen. Statische Prüfansätze unterscheiden sich von dynamischen Ansätzen dadurch, dass statische Ansätze das zu prüfende Softwaresystem nicht ausführen. Der leichteren Lesbarkeit halber sind im Folgenden mit „Prüfungen auf Architekturtreue“ oder „Architektur-Konformanzprüfungen“ ausschließlich die statischen Ansätze gemeint.

Dieses Kapitel beginnt mit einem detaillierten Blick auf das von Prüfungen auf Architekturtreue adressierte Problem: die Erosion von Softwaresystemen. Was konkret bedeutet Erosion, wie kommt sie zustande, welche Arten von Architekturverstößen lassen sich unterscheiden?

Anschließend gibt Abschnitt 4.2 einen Überblick über die allen Prüfansätzen zugrundeliegenden Mechanismen. Den verschiedenen Prüfansätzen ist gemein, dass sie werkzeugunterstützt Abweichungen der aktuellen Architektur von der vorgegebenen Architektur aufdecken. Jedoch unterscheidet sich das Architekturverständnis der verschiedenen Ansätze im Detail. Somit ist jeder Ansatz für bestimmte Architekturvorgaben gut geeignet, während andere Vorgaben nur umständlich oder gar nicht geprüft werden können (Passos et al. 2010; Knodel et al. 2006). Die meisten der bestehenden Ansätze sind darauf ausgerichtet, die Treue zu Architekturvorgaben in Form von Soll-Architekturen zu prüfen. Diese Ansätze sind höchstens eingeschränkt für die Prüfung auf Stiltreue geeignet.

Die Abschnitte 4.3 und 4.4 betrachten solche nicht auf Stile ausgerichteten Ansätze. So lassen sich verschiedene Varianten nachvollziehen, wie Wissenschaft und Praxis die Forschungsfragen dieser Arbeit für andere Architekturvorgaben als Stile bereits beantwortet haben. Abschnitt 4.3 stellt die Software-Reflexionsmodelle (SRM) vor (Murphy et al. 2001). Hierbei handelt es sich um einen verbreiteten Ansatz zur Prüfung auf Architekturtreue. Der Ansatz spielt eine besondere Rolle, da viele weitere Ansätze und Werkzeuge dem Prinzip der SRM folgen, auch die stilbasierte Architekturprüfung baut teilweise auf diesem Ansatz auf. Es folgt in Abschnitt 4.4 ein Überblick über weitere Ansätze und Werkzeuge sowie über angrenzende Forschungsfelder.

Abschnitt 4.5 grenzt die vorliegende Arbeit gegen angrenzende Forschungsfelder ab.

Abschließend stellt Abschnitt 4.6 erste bereits vorhandene Ansätze vor, die sich direkt mit der Prüfung auf Stiltreue beschäftigen.

4.1 Architekturerosion

Prüfungen auf Architekturtreue adressieren das Problem der Architekturerosion. Dass Architekturen erodieren, hat sich in verschiedenen Untersuchungen gezeigt (Jahnke 2009; Rosik et al. 2008; Feilkas et al. 2009; Eick et al. 2001; Silva und Balasubramaniam 2012). Das Phänomen ist auch bekannt unter den Bezeichnungen Architekturzerfall (architectural decay) (Avgeriou et al. 2005; Feilkas et al. 2009; Riaz et al. 2009), Architekturabweichung (architectural drift) (Perry und Wolf 1992) Strukturrückgang (structural decline) (Broy und Reussner 2010), Software-Entropie (software entropy) (Jacobson 1992) und Architektur-Degeneration (architectural degeneration) (Hochstein und Lindvall 2005). Wie in der Einleitung definiert, bedeutet Architekturerosion, dass die betroffene Architektur zunehmend gegen ihre Vorgaben verstößt (Definition 1-4). Die Untersuchungen im Rahmen dieser Arbeit haben bestätigt, dass – wie erwartet – dieses Problem auch bei Architekturvorgaben in Form von Stilen auftritt. Dies beobachten auch bisherige Untersuchungen (Lilienthal 2009).

Wie genau gestaltet sich Erosion? Wird ein Softwaresystem erstellt, geändert oder erweitert, so muss das Entwicklungsteam auswählen, welche Architekturelemente es hierfür erstellen oder ändern möchte. Diese Entwurfsentscheidung erfolgt auf Ebene der Architektur. Die Änderungen selber jedoch werden auf Quelltextebene durchgeführt. Wenn ein Teammitglied die Änderungen implementiert, muss es sich des Zusammenhangs zwischen Quelltext- und Architekturebene bewusst sein. Nur so ist es möglich, dass die Änderungen den Architekturvorgaben folgen. Bearbeitet ein Teammitglied beispielsweise eine Klasse namens Patientenmappe, so muss es das zugehörige Architekturelement kennen und wissen, welche Vorgaben für dieses Element und seine Beziehungen zu anderen Elementen gelten.

Handelt es sich um eine stilbasierte Architektur, so hängen die Vorgaben von dem Typ des Architekturelements ab. Gehört beispielsweise die Klasse Patientenmappe zu einem Architekturelement des Typs Material (siehe Abbildung 18), so weiß das Teammitglied, dass es auf die im Stil definierten Regeln für den Typ Material achten muss.

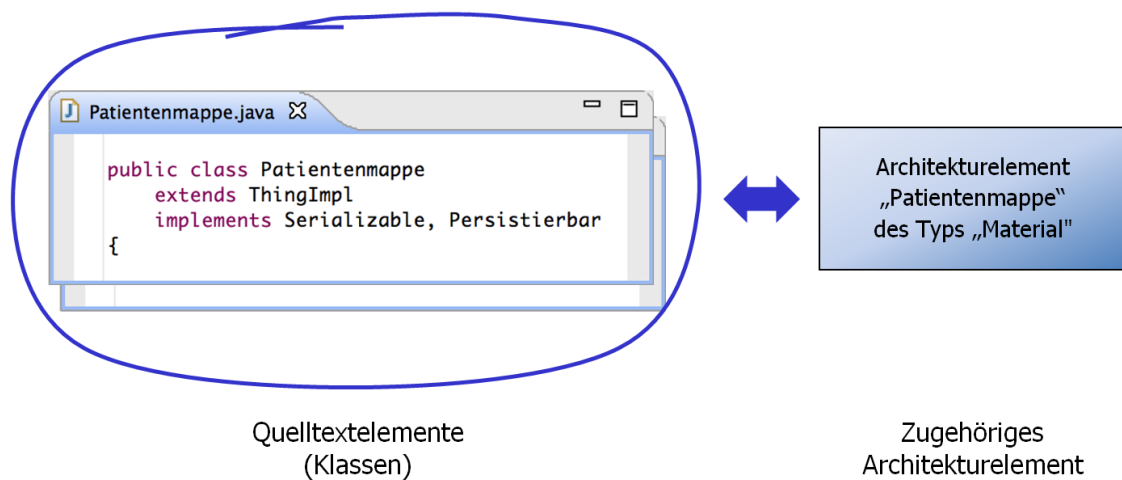


Abbildung 18: Quelltext- und Architekturebene am Beispiel der Patientenmappe

In der Praxis zeigt sich jedoch, dass Softwaresysteme selbst dann gegen ihre Architekturvorgaben verstoßen, wenn Teammitglieder im Prinzip wissen, welche Vorgaben gelten. Vier Teilprobleme lassen sich unterscheiden, die zu Verstößen führen:

Versehentliche Verstöße:

- *Zusammenhang zwischen Quelltext und Architekturvorgaben ist unbekannt:* Teammitglieder bearbeiten Quelltextelemente, kennen jedoch nicht das zugehörige Architekturelement und (bei stilbasierten Architekturen) dessen Typ. Folglich wissen die Teammitglieder nicht, welche Regeln sie beachten müssen.
- *Verstöße werden übersehen:* Teammitglieder bearbeiten Quelltextelemente und sind sich zwar bewusst, welche Regeln in diesem Fall greifen, übersehen jedoch, dass ihre Änderungen gegen diese Regeln verstoßen.

Beabsichtigte Verstöße:

- *Provisorien:*
Teammitglieder verstoßen bewusst gegen die Vorgaben, meist aufgrund von Zeitdruck. Die praktischen Untersuchungen dieser Arbeit zeigen: Dies geschieht oft mit der guten Absicht, den Quelltext später zu korrigieren.
- *Bewusste Ausnahmen:*
Selten geschieht es, dass Teammitglieder bewusst gegen Vorgaben verstoßen. Dies kann beispielsweise vorkommen, wenn der gewählte Stil für eine Komponente angepasst werden soll, die einen anderen Charakter besitzt als das restliche Softwaresystem (vgl. Garlan et al. 1994). Die Entscheidung für solche Ausnahmen ist nicht immer endgültig. Später will das Team die Ausnahmen überprüfen können und alternative Lösungen diskutieren.

Wie in den nachfolgenden Abschnitten dargestellt, zeigen die bestehenden Ansätze zur Architekturprüfung werkzeuggestützt Verstöße gegen Architekturvorgaben auf, meist handelt es sich um Vorgaben in Form von Soll-Architekturen. Viele Ansätze basieren auf einer von der Software-Entwicklung getrennten Prüfung. So verhindern diese Ansätze keine versehentlichen Verstöße, sondern ermöglichen den Entwicklerinnen und Entwicklern lediglich, ihren Quelltext nachträglich zu korrigieren.

4.2 Prüfungen auf Architekturtreue im Überblick

Die Kernaufgabe der verschiedenen Ansätze zur Prüfung auf Architekturtreue ist, Verstöße der zu prüfenden Ist-Architektur gegen die gewählten Architekturvorgaben aufzuzeigen. Der überwiegende Anteil der bisherigen Ansätze behandelt Architekturvorgaben in einer Form, die Soll-Architekturen entspricht (Knodel und Popescu 2007). Diese Ansätze umfassen – auf abstrakter Ebene betrachtet – die gleichen drei Berechnungsschritte: „statische Analyse“, „Ist-Architektur berechnen“ und „Architektur prüfen“. Die vorliegende Arbeit hat diese Berechnungsschritte als Gemeinsamkeit der bisherigen Ansätze extrahiert. Dieser Abschnitt stellt die Berechnungsschritte vor; anschließend folgt ein Überblick über die einzelnen bisherigen Ansätze und Werkzeuge.

Der zentrale Berechnungsschritt der Prüfungen vergleicht die Ist- mit der Soll-Architektur, um eventuelle Verstöße der Ist-Architektur gegen die Soll-Architektur zu ermitteln. Abbildung 19 visualisiert diesen Arbeitsschritt. Der Berechnungsschritt wird bei allen Ansätzen automatisiert durch ein Software-Werkzeug durchgeführt.

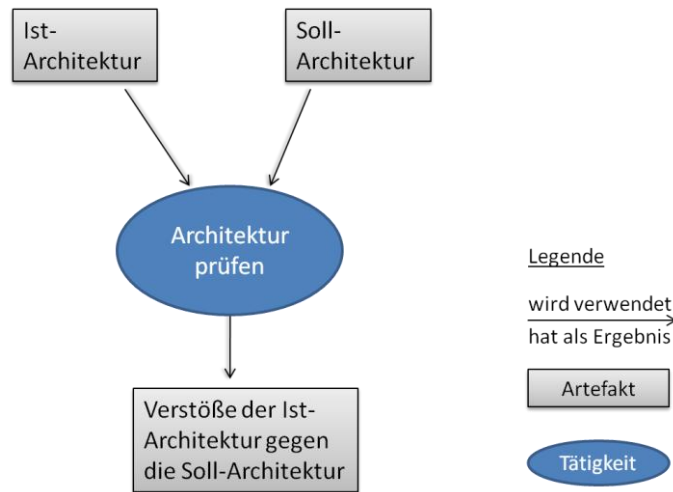


Abbildung 19: Kernaufgabe von Prüfungen auf Architekturtreue

Die Eingangsdaten für den Berechnungsschritt sind Modelle der Ist- und der Soll-Architektur. Bei allen Prüfansätzen sind manuelle Eingaben der prüfenden Personen notwendig.

Die *Soll-Architektur* wird komplett manuell beschrieben. In welcher Form diese Beschreibung eingegeben wird, variiert: einige Werkzeuge verwenden grafische, andere Werkzeuge textuelle Darstellungen, auch interaktive Eingaben sind möglich.

Die *Ist-Architektur* wird von den Prüfwerkzeugen vor der eigentlichen Architekturprüfung berechnet¹³. Die Prüfwerkzeuge benötigen für die Berechnung der Ist-Architektur die Quelltextstruktur (siehe Definition 2-2, Seite 17) sowie manuell ergänzte Zusatzinformationen, mit denen sie den Zusammenhang zwischen Quelltextstruktur und Architektur herstellen. Die Zusatzinformationen umfassen mindestens die Zuordnung zwischen Quelltext- und Architekturelementen Z_{QA} .

Welche Zusatzinformationen im Detail benötigt werden, hängt von dem jeweiligen Ansatz sowie von der verwendeten Programmiersprache ab. Auch das Eingabeformat der Zusatzinformationen variiert bei den verschiedenen Ansätzen und Werkzeugen; die Informationen können textuell oder grafisch dargestellt sein. Abbildung 20 visualisiert die zwei Berechnungsschritte „Ist-Architektur berechnen“ und „Architektur prüfen“ und stellt die Eingangswerte der Berechnungen dar.

¹³ Zur Erinnerung sei angemerkt: genau genommen wird an dieser Stelle ein *Modell* der Ist-Architektur berechnet, die Ist-Architektur ist eine Eigenschaft der Software. Es ist jedoch üblich, von der Berechnung der Ist-Architektur zu sprechen.

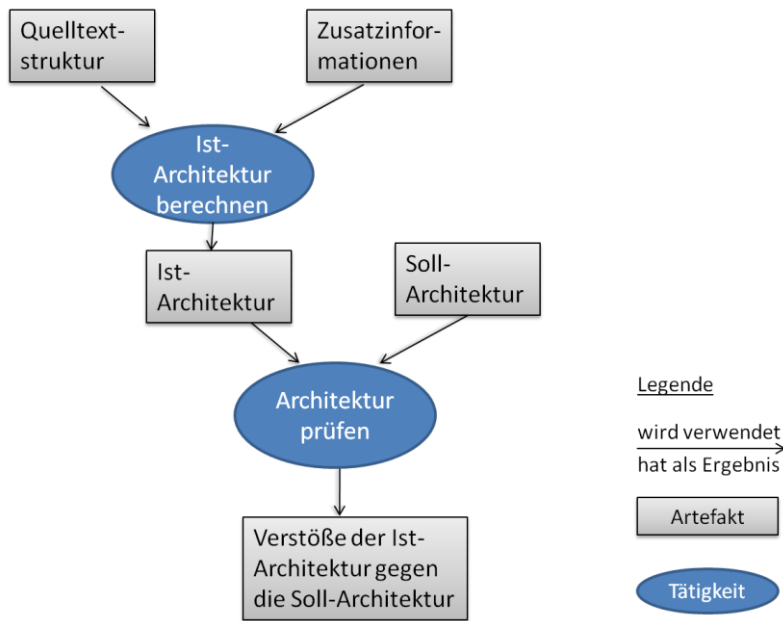


Abbildung 20: Bevor die Architektur geprüft werden kann, muss die Ist-Architektur berechnet werden.

Die Quelltextstruktur wird bei allen Ansätzen durch statische Analyse aus dem Quelltext der zu prüfenden Systeme berechnet. Die Ansätze unterscheiden sich jedoch darin, ob sie die Berechnungsvorschrift festlegen oder ob sie die Berechnungsvorschrift der prüfenden Person überlassen. Ferner unterscheiden sich die verschiedenen Prüfwerkzeuge dahingehend, ob sie die statische Analyse selber durchführen können oder ob die prüfende Person dazu externe Werkzeuge verwenden muss. Werkzeuge, welche die statische Analyse selber durchführen, bieten eine individuelle Berechnungsvorschrift für jede unterstützte Programmiersprache. Werkzeuge, welche die statische Analyse nicht unterstützen, definieren stattdessen eine Benutzungsschnittstelle, über die die Quelltextstruktur eingegeben wird, beispielsweise in Form einer Textdatei. Abbildung 21 zeigt alle drei Berechnungsschritte.

Neben diesen drei Berechnungsschritten bieten einige Ansätze und Werkzeuge weitere Berechnungen an. Einige Ansätze geben nicht nur an, an welchen Stellen die Ist-Architektur gegen die Vorgaben verstößt, sondern verfolgen die Verstöße in die Quelltextstruktur oder sogar bis in den Quelltext. So weiß die prüfende Person unmittelbar, welche Quelltextstellen die Probleme verursachen.

Die folgenden Abschnitte stellen verschiedene Ansätze und Werkzeuge zur Prüfung auf Architekturtreue vor, erläutern jeweils, wie die hier vorgestellten drei Berechnungsschritte definiert sind und zeigen die Unterschiede zwischen den verschiedenen Prüfansätzen und Werkzeugen auf.

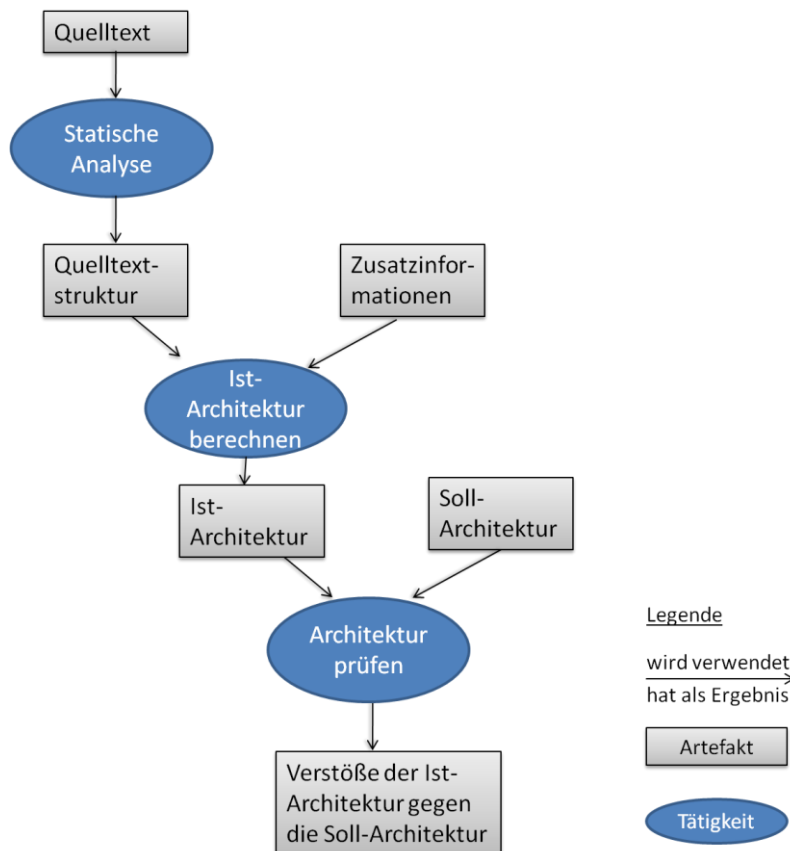


Abbildung 21: Überblick über die drei Kern-Arbeitsschritte von Prüfungen auf Architekturtreue

4.3 Software-Reflexionsmodelle

Software-Reflexionsmodelle (SRM) (Murphy et al. 1995; Murphy et al. 2001) sind ein in Wissenschaft und Praxis weit verbreiteter Ansatz zur Architekturprüfung. Die Verbreitung des Ansatzes zeigt sich unter anderem darin, dass eine Vielzahl heutiger Werkzeuge zur Architekturprüfung das Konzept der SRM nutzt, variiert und ergänzt, wie beispielsweise Sonargraph¹⁴, die Sotograph-Familie (Bischofberger et al. 2004), Lattix (Sangal et al. 2005), das Werkzeug SAVE (Duszynski et al. 2009) und Structure 101 (Sangwan et al. 2008).

Mit dem SRM-Ansatz lässt sich die Architekturtreue von Softwaresystemen werkzeuggestützt prüfen. Der SRM-Ansatz unterstützt ausschließlich Architekturvorgaben bezüglich der *Beziehungen* zwischen Architekturelementen. Das Ergebnis einer Prüfung ist ein sogenanntes SRM. Ein SRM ist ein Modell des untersuchten Softwaresystems auf Architekturebene. SRMs bestehen aus der Ist-Architektur sowie den Abweichungen und Übereinstimmungen zwischen Soll und Ist.

¹⁴ www.hello2morrow.com

Ein SRM wird in fünf Teilaufgaben erstellt und ausgewertet (Murphy et al. 1995; Murphy et al. 2001):

- Die Soll-Architektur definieren
- Die Quelltextstruktur ermitteln
- Quelltext- und Architekturelemente einander zuordnen
- Das SRM berechnen
- Das SRM interpretieren und weitere Schritte planen

Die fünf Teilaufgaben geben keine feste Reihenfolge vor, sie lassen sich auch iterativ bearbeiten. Die Teilaufgaben werden werkzeuggestützt durchgeführt. Hierfür haben die Autoren des SRM-Ansatzes ein prototypisches Werkzeug erstellt (Murphy et al. 1995; Murphy et al. 2001).

Die folgenden fünf Abschnitte beschreiben jeweils eine Teilaufgabe und erläutern die Struktur von SRMs im Detail. Die Abschnitte konzentrieren sich auf die für diese Arbeit relevanten Aspekte des Grundkonzepts des SRM-Ansatzes (Murphy et al. 1995; Murphy et al. 2001). Die verschiedenen Erweiterungen und Varianten beschreibt Abschnitt 4.4.

Für die Darstellung des Ansatzes wählt diese Arbeit eine andere Darstellungsform als Murphy et al. Die Arbeit überträgt die von Murphy et al. vorgestellten Konzepte in diese neue Darstellungsform. Die Semantik der Software-Reflexionsmodelle wurde ursprünglich in der formalen Spezifikationssprache Z beschrieben (Murphy et al. 2001). Diese Arbeit überträgt die Beschreibung der Semantik in eine rein mengentheoretische Notation, da Z im Bereich der Architekturprüfungen ansonsten nicht verbreitet ist und da sich die mengentheoretische Notation für diese Arbeit gut eignet. Zusätzlich zur mengentheoretischen Darstellung werden die Zusammenhänge in UML visualisiert und anhand von Beispielen veranschaulicht. Teilweise ändert und ergänzt diese Arbeit die Bezeichnungen der Mengen und Relationen des SRM-Ansatzes so, dass sie zu den hier gewählten, deutschen Begrifflichkeiten passen. Dies ist entsprechend markiert.

1. Teilaufgabe: Die Soll-Architektur definieren

Die Soll-Architektur wird von Murphy et al. als high-level model bezeichnet. Die Soll-Architektur des SRM-Ansatzes besteht aus Architekturelementen und Beziehungen. Die Schnittstellen der Architekturelemente werden von den SRM nicht betrachtet. Dies lässt sich dadurch erklären, dass erst neuere Architekturdefinitionen auch Schnittstellen umfassen (siehe Abschnitt 2.2).

Die Soll-Architektur wird durch die prüfende Person über die Benutzungsschnittstelle des Prüfwerkzeugs eingegeben. Formal dargestellt besteht die Soll-Architektur aus Folgendem:

Soll-Architektur des SRM-Ansatzes, $\langle A, B_S \rangle$:

- Eine Menge A von Architekturelementen
- Eine Relation $B_S \subseteq A \times A$, die die vorgeschriebenen Beziehungen zwischen diesen Architekturelementen beschreibt

Die vorgeschriebenen Beziehungen B_S werden im Folgenden kurz als Soll-Beziehungen bezeichnet. Sie sollen in dem zu prüfenden Softwaresystem vorliegen. Fehlt eine Beziehung

oder besteht eine weitere Beziehung, so verstößt das Softwaresystem gegen die Architekturvorgaben. Abbildung 22 zeigt das Konzept der Soll-Architektur in UML-Darstellung:

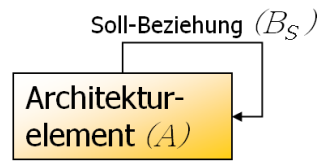


Abbildung 22: Die Soll-Architektur des SRM-Ansatzes (UML-Darstellung)

Abbildung 23 zeigt ein Beispiel für eine Soll-Architektur. Es handelt sich um einen vereinfachten Ausschnitt aus dem Rahmenwerk Spring¹⁵.

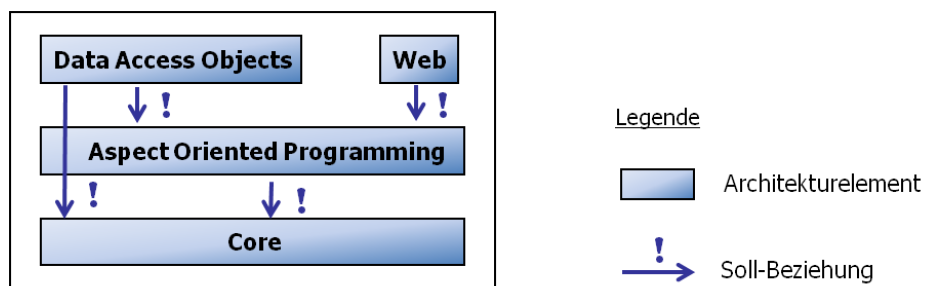


Abbildung 23: Die Soll-Architektur legt Architekturelemente und Beziehungen fest (Beispiel)

2. Teilaufgabe: Die Quelltextstruktur ermitteln

Die Quelltextstruktur bezeichnen Murphy et al. als source model. Die Quelltextstruktur wird – wie bei allen Ansätzen zur Architekturprüfung – durch statische Analyse aus dem Quelltext berechnet. Was als Quelltextelement und was als Beziehung angesehen wird, kann die prüfende Person im SRM-Ansatz frei wählen (Murphy et al. 1995, S. 19). Formal dargestellt umfasst die Quelltextstruktur Folgendes:

Quelltextstruktur des SRM-Ansatzes $\langle Q, B_Q \rangle$:

- Eine Menge Q von Quelltextelementen
- Eine Relation $B_Q \subseteq Q \times Q$ von Quelltextbeziehungen

Abbildung 24 zeigt die Quelltextstruktur des SRM-Ansatzes in UML-Darstellung.

¹⁵ projects.spring.io/spring-framework

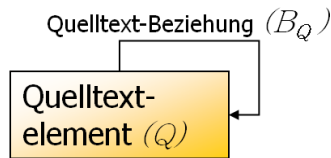


Abbildung 24: Die Quelltextstruktur (UML-Darstellung)

Die Quelltextstruktur des SRM-Ansatzes weicht von der in dieser Arbeit verwendeten Definition 2-2 dadurch ab, dass keine Schnittstellen existieren. Ferner unterscheidet der SRM-Ansatz keine Beziehungstypen.

Die Quelltextstruktur kann nicht mit dem SRM-Werkzeug erstellt werden. Stattdessen spezifiziert das Werkzeug ein Textformat für Quelltextstrukturen. Die prüfende Person muss mit einem beliebigen externen Werkzeug eine Textdatei erstellen, welche die gesamte Quelltextstruktur in dem vorgegebenen Format beschreibt. Das SRM-Werkzeug enthält eine Schnittstelle, über die es solche Textdateien einliest.

3. Teilaufgabe: Quelltext- und Architekturelemente einander zuordnen

Die prüfende Person stellt den Zusammenhang zwischen Quelltextstruktur und Soll-Architektur her: mit Hilfe des SRM-Werkzeugs ordnet sie die Quelltexte den Architekturelementen zu. Diese Zuordnung entspricht der Definition 2-8. Zur Erinnerung:

Zuordnung Z_{QA} zwischen Quelltext- und Architekturelementen:

- Die Relation $Z_{QA} \subseteq Q \times A$ beschreibt, welche Quelltexte Q und Architekturelemente A zusammen gehören.

Abbildung 25 zeigt ein Beispiel, hier sind Package-Bäume aus Spring vier verschiedenen Architekturelementen zugeordnet. Abbildung 26 visualisiert Z_{QA} zur Erinnerung in UML.

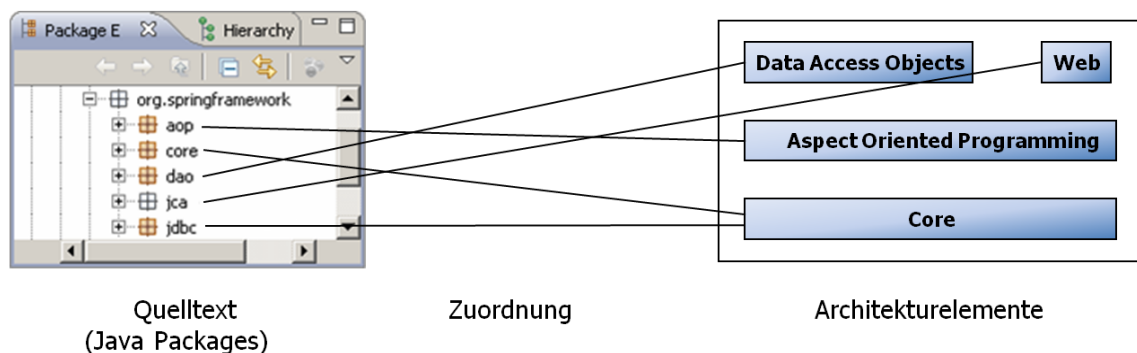


Abbildung 25: Zuordnung Z_{QA} (Beispiel)

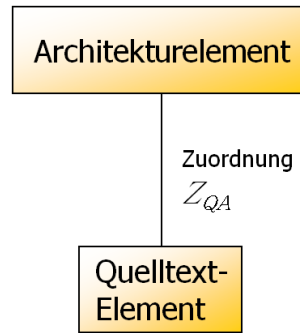


Abbildung 26: Zuordnung Z_{QA} (UML-Darstellung)

Dem Beispiel lässt sich entnehmen, dass einem Architekturelement mehrere Quelltextelemente zugeordnet werden können. Auch der umgekehrte Fall ist prinzipiell erlaubt, kommt jedoch in den zum SRM-Ansatz veröffentlichten Beispielen nicht vor.

Der SRM-Ansatz erlaubt, dass Quelltextelemente, die für die Architektur irrelevant sind, keinem Architekturelement zugeordnet werden.

4. Teilaufgabe: Das SRM berechnen

Aufbauend auf der Soll-Architektur, der Quelltextstruktur und der Zuordnung Z_{QA} ermittelt das Prüfwerkzeug automatisiert die Beziehungen der Ist-Architektur. Die Berechnung der Ist-Architektur wird von Murphy et al. nicht als eigener Schritt ausgewiesen, die Ist-Architektur ist stattdessen ein nicht getrennt ausgewiesener Teil des Software-Reflexionsmodells. Die vorliegende Arbeit hingegen unterscheidet die Ist-Architektur explizit von dem gesamten Software-Reflexionsmodell, um das Verständnis zu erleichtern. Die Ist-Architektur ist über folgende Mengen und Relationen definiert:

Ist-Architektur des SRM-Ansatzes $\langle A, B_I \rangle$

- Die Menge A von Architekturelementen
- Die Beziehungen der Ist-Architektur sind definiert als Relation über die Architekturelemente: $B_I \subseteq A \times A$.

Die Ist-Architektur des SRM-Ansatzes unterscheidet sich von der formalen Architekturdefinition 2-1 dieser Arbeit dadurch, dass hier keine Schnittstellen vorkommen. Ferner werden keine Beziehungstypen unterschieden.

Die Ist-Architektur des SRM-Ansatzes enthält dieselben Architekturelemente wie die Soll-Architektur. Die Ist- und die Soll-Architektur des SRM-Ansatzes können sich lediglich in den Beziehungen unterscheiden.

Die Beziehungen der Ist-Architektur B_I werden anhand der Beziehungen der Quelltextelemente B_Q und der Zuordnung Z_{QA} berechnet. Dabei gilt:

$$B_I = \{(a, b) | \exists p, q : (p, q) \in B_Q \wedge (p, a) \in Z_{QA} \wedge (q, b) \in Z_{QA}\}$$

Das heißt, wenn zwei Quelltextelemente p und q in Beziehung stehen, gibt es eine Beziehung in der Ist-Architektur zwischen den Architekturelementen a und b , sofern p und a sowie q und b einander zugeordnet sind.

Abbildung 27 zeigt, wie eine Ist-Architektur aus der Quelltextstruktur ermittelt wird. Dies entspricht der in Abschnitt 2.3 vorgestellten Aggregation von Quelltextbeziehungen. Die Pfeile stellen Beziehungen dar, die Kästchen repräsentieren Quelltext- und Architekturelemente. Quelltextelemente sind innerhalb der zugeordneten Architekturelemente dargestellt. Abbildung 28 zeigt die Quelltextstruktur, die Ist-Architektur und die Zuordnung Z_{QA} in UML.

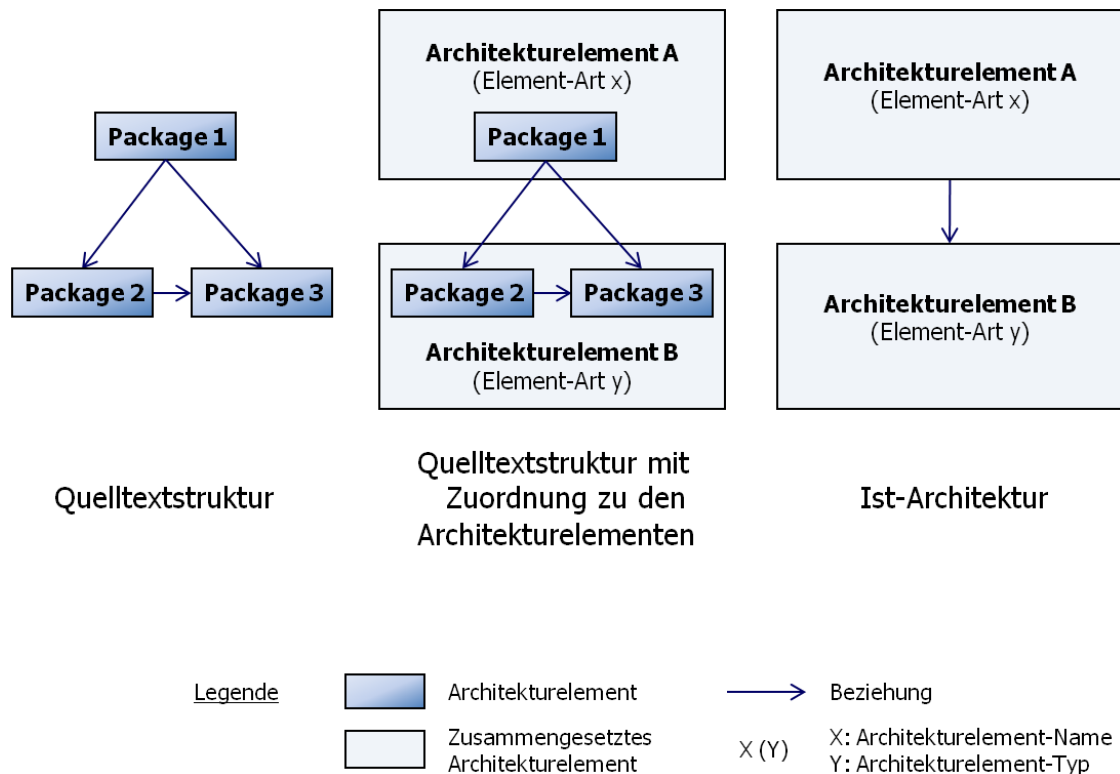


Abbildung 27: Überblick: Ist-Architektur ermitteln (Beispiel)

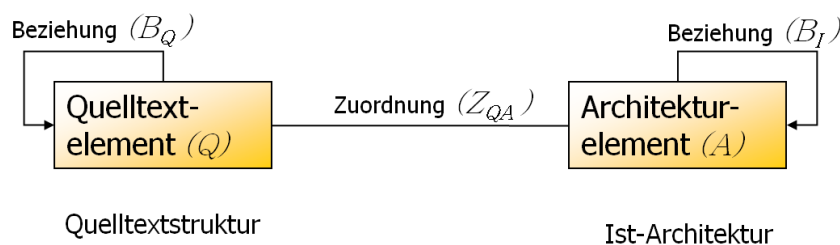


Abbildung 28: Ist-Architektur, Quelltextstruktur und Zuordnung Z_{QA} (UML-Darstellung)

Zur Veranschaulichung soll an dieser Stelle ein vereinfachter Ausschnitt aus dem Rahmenwerk Spring betrachtet werden: Die Klasse CommonsPoolTargetSource referenziert die Klasse Constants. Diese Referenz ergibt sich aus dem Quelltext (Abbildung 29).

```

CommonsPoolTargetSource.java
package org.springframework.aop.target;
import org.springframework.aop.core.Constants;
public class CommonsPoolTargetSource extends Abstr
private static final Constants constants = new

```

Abbildung 29: Quelltextbeziehung (Beispiel)

Die Klasse CommonsPoolTargetSource gehört zum Architekturelement AspectOriented-Programming, die Klasse Constants gehört zum Architekturelement Core (Abbildung 30). Aus der Beziehung zwischen den zwei Klassen ergibt sich eine Beziehung zwischen den zugeordneten Architekturelementen (Abbildung 31).

Formal stellt sich dieser Zusammenhang folgendermaßen dar: Es gilt:

$$B_I = \{(e, f) | \exists p, q : (p, q) \in B_Q \wedge (p, e) \in Z_{QA} \wedge (q, f) \in Z_{QA}\}$$

(siehe oben). Für $e = AspectOrientedProgramming$ und $f = Core$ ist diese Bedingung erfüllt (mit $p = CommonsPoolTargetSource$ und $q = Constants$).

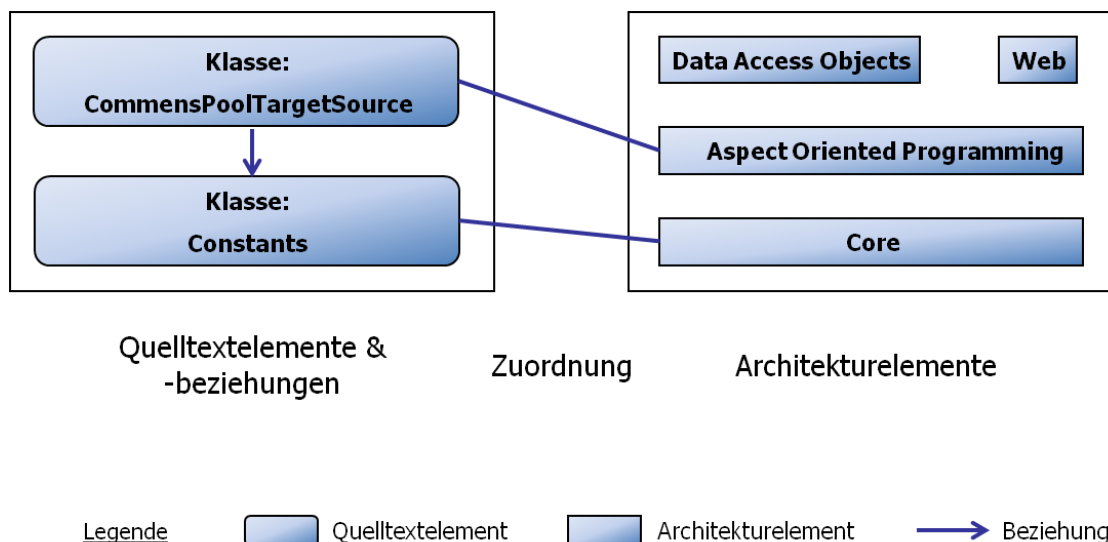


Abbildung 30: Beziehungen ermitteln: Ausgangssituation (Beispiel)

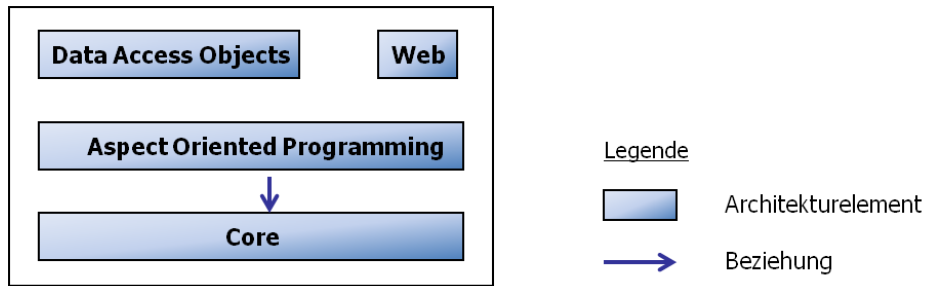


Abbildung 31: Resultierende Beziehung in der Ist-Architektur (Beispiel)

Letztendliches Ziel der verschiedenen Teilaufgaben ist, ein SRM zu berechnen. Dafür wird geprüft, ob die tatsächlichen Beziehungen in der Ist-Architektur den gewünschten Beziehungen in der Soll-Architektur entsprechen. Ein SRM enthält das Ergebnis dieses Vergleichs, es zeigt die Übereinstimmungen und Abweichungen.

Abbildung 32 visualisiert beispielhaft ein SRM. Die Abbildung zeigt die Ist- und die Soll-Architektur sowie das SRM. Die durchgezogenen Pfeile im SRM zeigen, wo die Beziehungen der Ist-Architektur mit den Vorgaben übereinstimmen, diese Beziehungen werden als Übereinstimmungen (convergence) bezeichnet. Die anderen Pfeile zeigen, wo die Beziehungen der Ist-Architektur nicht mit den Vorgaben übereinstimmen. Dabei wird zwischen nicht gewollten und fehlenden Beziehungen unterschieden. Die nicht gewollten Beziehungen werden als Abweichungen (divergence) bezeichnet, die fehlenden Beziehungen als Abwesenheit (absence).

Formal bestehen Software-Reflexionsmodelle aus den Mengen A , Co , Di und Ab . Diese Mengen werden aus den oben erläuterten Mengen B_I (Beziehungen innerhalb der Ist-Architektur) und B_S (Beziehungen innerhalb der Soll-Architektur) berechnet:

Software-Reflexionsmodell $\langle A, Co, Di, Ab \rangle$

- A bezeichnet die Menge der Architekturelemente
- Die Relation $Co = B_I \cap B_S$ beinhaltet die übereinstimmenden Beziehungen (convergences)
- Die Relation $Di = B_I \setminus B_S$ beinhaltet die abweichenden Beziehungen (divergences)
- Die Relation $Ab = B_S \setminus B_I$ beinhaltet die fehlenden Beziehungen (absences)

Ein SRM-Modell ist folgendermaßen zu interpretieren: Die Menge A gemeinsam mit der Vereinigungsmenge von Co und Di stellt die Ist-Architektur dar. Die Soll-Architektur umfasst die Menge A gemeinsam mit der Vereinigungsmenge von Co und Ab . Die Verstöße gegen die Soll-Architektur werden durch die Mengen Di und Ab beschrieben. Da die SRM-Modelle lediglich Vorgaben bezüglich der Beziehungen prüfen, beinhalten die Mengen der Verstöße ausschließlich Beziehungen.

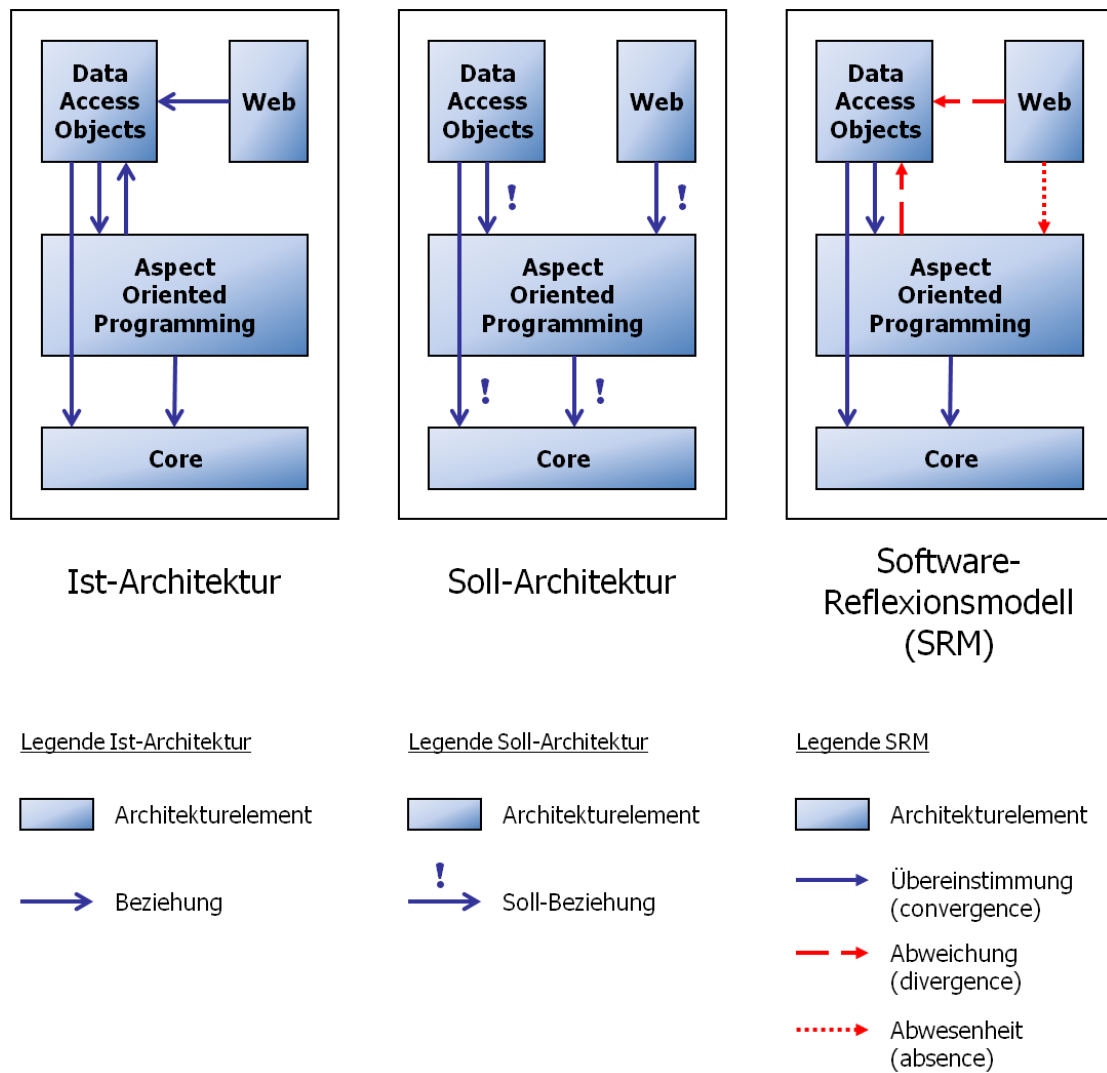


Abbildung 32: Software-Reflexionsmodell (Beispiel)

5. Teilaufgabe: Das SRM interpretieren und weitere Schritte planen

In der fünften Teilaufgabe untersucht die prüfende Person das Reflexionsmodell und plant weitere Schritte. Beispielsweise kann sie die vorherigen Teilaufgaben überarbeiten und so iterativ ihr Reflexionsmodell verfeinern oder korrigieren, oder sie kann Refactorings planen, um das System an die Soll-Architektur anzupassen.

Zusammenfassend sei festgehalten:

Der SRM-Ansatz prüft die Einhaltung von Architekturvorgaben in Form von *einzelnen aufgelisteten Beziehungen*. Es ist die Aufgabe der prüfenden Person, die vorgegebenen Beziehungen aufzulisten. Erlaubte Beziehungen lassen sich nicht spezifizieren, nur vorgeschriebene und untersagte Beziehungen.

Da die gesamte Quelltextstruktur in einem externen Werkzeug eingelesen wird, ist es mit dem ursprünglichen SRM-Ansatz nicht möglich, die Architekturprüfung direkt während der Programmierung durchzuführen. Viele der im Folgenden vorgestellten neueren Werkzeuge und

Ansätze verwenden das Prinzip der SRM, bieten jedoch eine in mehrerer Hinsicht flexiblere Lösung.

4.4 Weitere bisherige Ansätze und Werkzeuge

Die bisherigen Ansätze und Werkzeuge zur Prüfung auf Architekturtreue unterscheiden sich bezüglich verschiedener Eigenschaften. Für diese Arbeit ist die Frage entscheidend, welche Arten von Architekturvorgaben prüfbar sind und ob die Treue zu Architekturstilen unterstützt wird. Neben diesem Punkt gibt es Unterschiede bezüglich der Integration von Prüfung und Programmierung sowie von Architekturbeschreibung und Quelltext. Auch der Prüfprozess unterscheidet sich von Ansatz zu Ansatz. Im Folgenden wird ein Überblick über bisherige Ansätze in Hinblick auf diese Eigenschaften gegeben.

4.4.1 Software-Reflexionsmodelle mit getypten Beziehungen

Die Autoren der Software-Reflexionsmodelle haben ihren ursprünglichen Ansatz (Murphy et al. 1995) um getypte Beziehungen erweitert (Murphy et al. 1997). Diese Erweiterung hat Auswirkungen auf die Quelltextstruktur sowie auf die Ist- und die Soll-Architektur.

Quelltextstruktur: In der ursprünglichen Fassung des SRM-Ansatzes kann die prüfende Person frei entscheiden, was sie als Quelltextbeziehung betrachtet. Beispielsweise kann sie lediglich Operationsaufrufe betrachten, Typpräferenzen jedoch ignorieren. Nach diesem Prinzip arbeitet auch die Erweiterung. In der erweiterten Fassung ist es zusätzlich möglich, verschiedene *Typen* von Quelltextbeziehungen zu unterscheiden. Die Autoren nennen „calls“ und „references to global variables“ als Beispiele für zwei Beziehungstypen (Murphy et al. 1997). Wie in der ursprünglichen Fassung wird die Quelltextstruktur mit einem externen Werkzeug erstellt, der SRM-Ansatz stellt dazu kein Werkzeug zur Verfügung. Die Quelltextstruktur muss für das SRM-Werkzeug in einem festgelegten Textformat als Datei vorliegen.

Die *Ist-Architektur* enthält ebenfalls getypte Beziehungen. Für jeden in der Quelltextstruktur vorkommenden Beziehungstyp werden die Beziehungen der Ist-Architektur einzeln berechnet. Das Verfahren zur Berechnung entspricht dem ursprünglichen SRM-Ansatz. Abbildung 33 zeigt ein Beispiel, in dem zwei verschiedene Beziehungstypen (Benutzt- und Vererbungsbeziehungen) aggregiert werden. Oben ist die Quelltextstruktur dargestellt, die Quelltextelemente sind umrandet von den zugeordneten Architekturelementen. Unten ist die resultierende Ist-Architektur zu sehen.

Die *Soll-Architektur* kann getypte und ungetypte Beziehungen enthalten. Die Semantik ist folgende:

- Enthält die Soll-Architektur eine getypte Beziehung, so wird eine Beziehung genau dieses Typs gefordert.
- Enthält die Soll-Architektur eine ungetypte Beziehung, so bedeutet dies, dass mindestens eine Beziehung gefordert ist. Die Beziehungstypen der Ist-Architektur spielen in diesem Fall keine Rolle. Dies entspricht der Semantik des ursprünglichen SRM-Ansatzes.

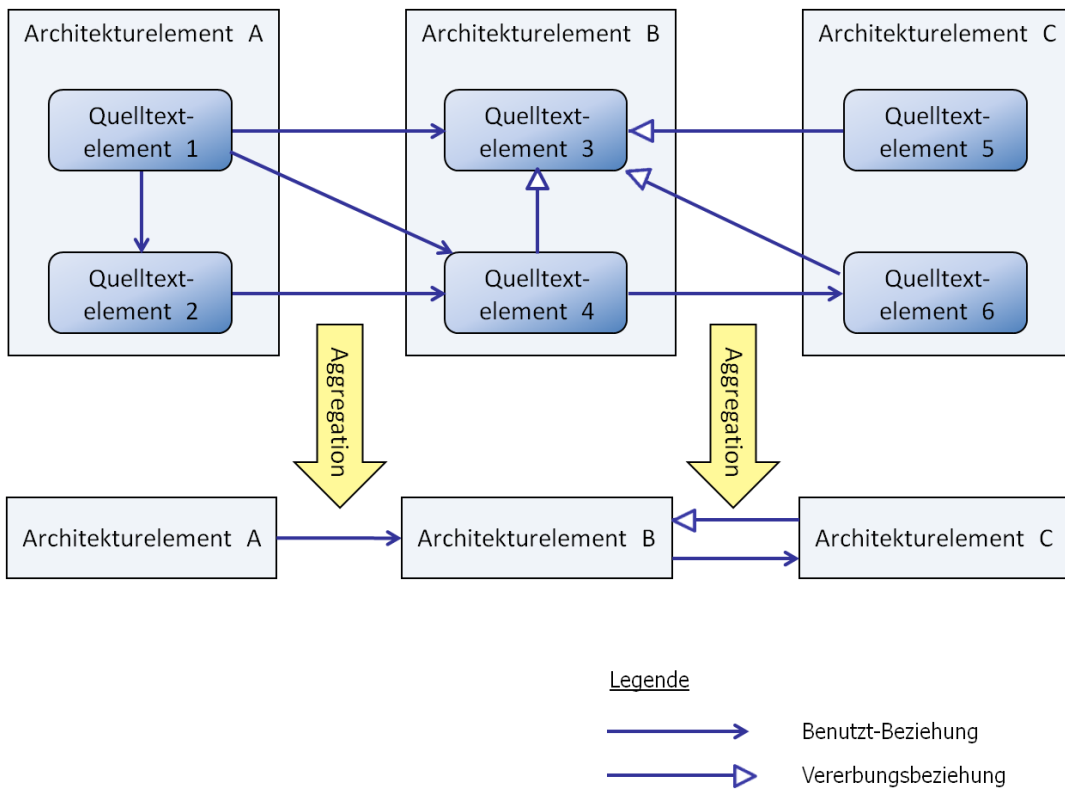


Abbildung 33: Aggregation getypter Beziehungen (Beispiel)

Abbildung 34 zeigt ein Beispiel für ein getyptes Reflexionsmodell mit getypten Beziehungen auch in der Soll-Architektur.

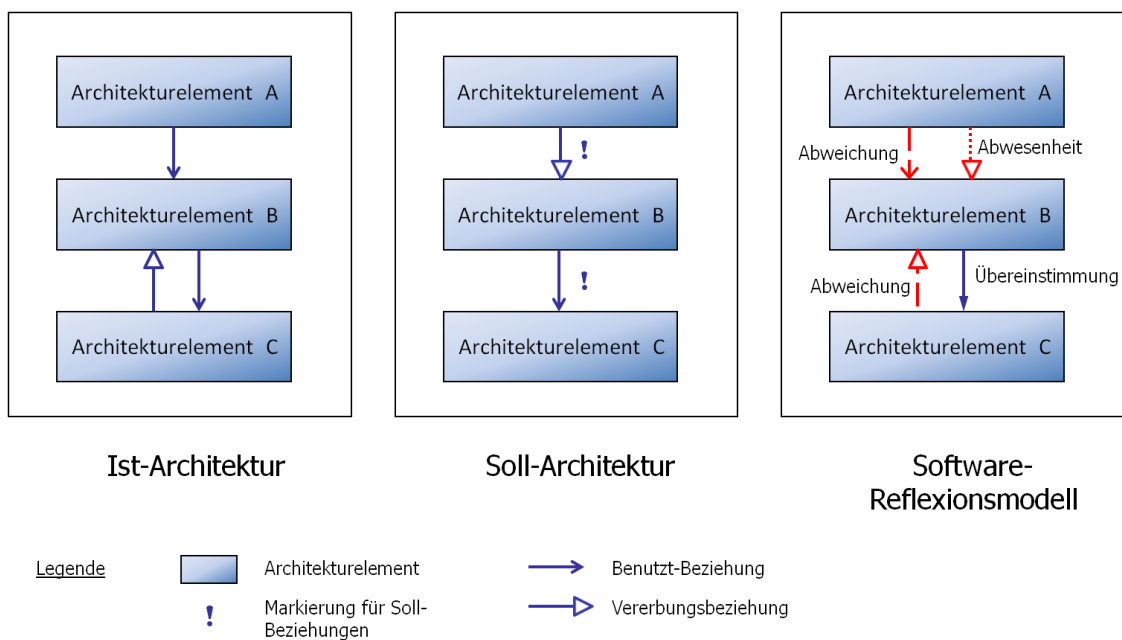


Abbildung 34: Beispiel für ein getyptes Software-Reflexionsmodell

In diesem Beispiel entspricht nur eine Beziehung den Vorgaben: die Benutzbeziehung von Architekturelement B zu Architekturelement C (diese Beziehung ist im SRM als Übereinstimmung mit den Vorgaben gekennzeichnet). Die geforderte Vererbungsbeziehung von Element A zu B fehlt (im SRM als Abwesenheit gekennzeichnet), alle weiteren Beziehungen der Ist-Architektur sind nicht gewollt (im SRM als Abweichung gekennzeichnet). Wäre innerhalb der Soll-Architektur die Beziehung von Element A zu B nicht getypt, dann würde die in der Ist-Architektur vorliegende Benutzbeziehung als Übereinstimmung angesehen.

4.4.2 Hierarchische Software-Reflexionsmodelle

Koschke et al. erweitern das ursprüngliche Konzept der SRM um hierarchische Strukturen innerhalb der Architektur (Koschke und Simon 2003) mit dem Ziel, eine abstraktere Sicht auf umfangreiche Architekturen zu ermöglichen.

Die Erweiterung erlaubt der prüfenden Person, Enthältbeziehung zwischen Architekturelementen festzulegen. Abbildung 35 zeigt ein Beispiel für eine hierarchische Soll-Architektur. Das Beispiel enthält zwei zusammengesetzte Architekturelemente (A und X) sowie vier atomare Architekturelemente (B, C, Y und Z). Ferner enthält das Beispiel zwei Soll-Beziehungen (von A zu X und von Y zu Z).

Wie im ursprünglichen SRM auch, so können in dieser Erweiterung allen Architekturelementen jeweils mehrere Quelltextelemente zugeordnet werden.

Die Soll-Beziehungen zwischen atomaren Architekturelementen haben dieselbe Semantik wie im ursprünglichen Ansatz, sie legen fest, welche Beziehungen in der Ist-Architektur erwartet werden. Das bedeutet in diesem Beispiel, dass auch die Ist-Architektur eine Beziehung von Y zu Z enthalten soll.

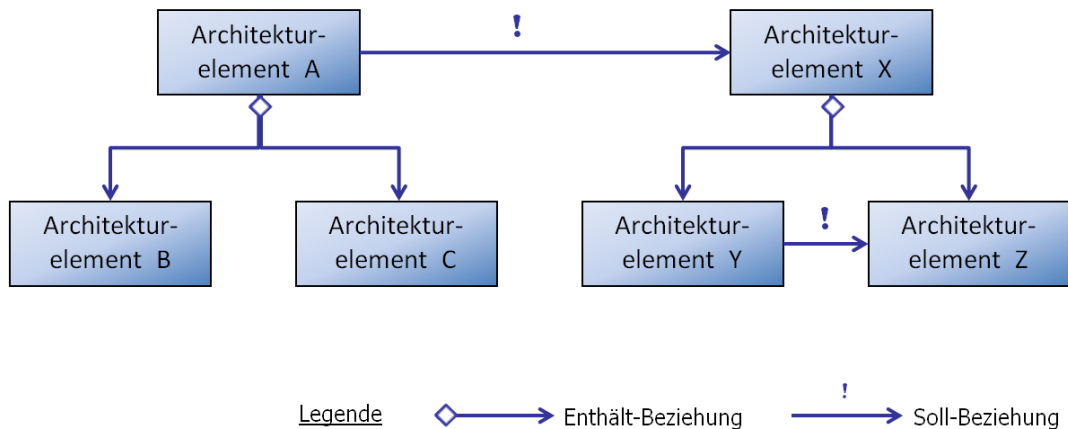


Abbildung 35: Eine hierarchische Soll-Architektur

Eine Soll-Beziehung zwischen zwei *zusammengesetzten* Architekturelementen A und X bedeutet, dass in der Ist-Architektur mindestens eine Beziehung vorliegen muss. Diese Beziehung muss ausgehen von A oder von einem in A enthaltenen Architekturelement. Die Beziehung muss hinführen zu X oder zu einem in X enthaltenen Architekturelement. Für das Beispiel in Abbildung 35 bedeutet dies, dass die Ist-Architektur mindestens eine Beziehung enthalten muss, die entweder von A, B oder C ausgeht und die zu X, Y oder Z führt.

Soll-Beziehungen sind auch zwischen atomaren und zusammengesetzten Architekturelementen möglich. Eine Soll-Beziehung zwischen einem atomaren Architekturelement B und einem zusammengesetzten Architekturelement X bedeutet, dass die Ist-Architektur mindestens eine Beziehung enthalten muss zwischen B und X oder zwischen B und einem der in X enthaltenen Architekturelemente.

4.4.3 Abhängigkeitsmodelle in Matrizendarstellung

Der Ansatz der Abhängigkeitsmodelle stellt die Abhängigkeiten von Architekturelementen in Form einer Matrix dar (Dependency Structure Matrix, kurz DSM) (Sangal et al. 2005). Zu dem Ansatz ist ein kommerzielles Software-Werkzeug verfügbar (Lattix¹⁶).

Der Ansatz ist eng verwandt mit den Software-Reflexionsmodellen, wie diese erlaubt er Vorgaben für Beziehungen zwischen Architekturelementen. Die DSM unterscheiden sich von den Software-Reflexionsmodellen in erster Linie in der Darstellung der Architekturvorgaben und der Prüfungsergebnisse.

Eine DSM ist eine quadratische Matrix die für jedes Architekturelement eine Spalte und eine Zeile enthält. Abbildung 36 zeigt eine DSM mit Architekturelementen. Die Elemente sind nummeriert, in der Beschriftung der Zeilen ist auch der Name der Architekturelemente dargestellt, so heißt beispielsweise das Architekturelement Nummer 1 „application“. Eine Zelle der Spalte x und der Zeile y enthält genau dann einen Eintrag, wenn das Architekturelement x eine Beziehung zu dem Architekturelement y aufweist. Der Eintrag in den Zellen stellt die Anzahl der Beziehungen dar. In der dargestellten Matrix bestehen beispielsweise 37 Beziehungen von application zu model. Wenn keine Beziehung besteht, ist die entsprechende Zelle leer. Dies trifft für die umgekehrte Richtung von model zu application.

		→	↺	↻	←	↷
+ application	1	.				
+ model	2	37	.			
+ domain	3	17	29	.		
+ framework	4	75	53	42	.	
+ util	5	10	13	16	13	.

Abbildung 36: Eine DSM (nach Sangal et al. 2005)

Wie die Software-Reflexionsmodelle erlaubt dieser Ansatz, die Korrektheit von *Beziehungen* in der Ist-Architektur basierend auf Vorgaben in Form einer *Soll-Architektur* zu prüfen. Die Soll-Architektur legt für jedes Paar A, B von Architekturelementen fest, ob eine Beziehung *erlaubt* oder *untersagt* ist. Hier unterscheidet sich der Ansatz von den Software-Reflexionsmodellen, bei denen die Soll-Architektur festlegt, wo Beziehungen *vorgeschrieben* und wo *untersagt* sind.

Die Vorgaben werden interaktiv definiert, indem für jede Zelle festgelegt werden kann, ob ein Eintrag erlaubt oder untersagt ist. Dabei kann die prüfende Person festlegen, ob alle

¹⁶ www.lattix.com

Beziehungen durch die jeweilige Vorgabe betroffen sind oder nur Beziehungen bestimmten Typs (wie beispielsweise nur Konstruktoraufrufe in der Programmiersprache Java).

Zellen, in denen ein Eintrag erlaubt ist, sind durch eine grüne Markierung in der oberen linken Ecke gekennzeichnet (siehe Abbildung 37). Ist kein Eintrag erlaubt ist, wird eine schwarze Markierung in der unteren linken Ecke angezeigt. Wenn eine Zelle gegen die Vorgaben verstößt, wird die obere rechte Ecke rot markiert. Das den Verstoß hervorrufende Architekturelement wird auf die gleiche Weise markiert. In Abbildung 37 beispielsweise verstößt Subsystem2 gegen die Vorgaben.

		→	2	3	4
Subsystem1	1	.	1	.	.
Subsystem2	2	1	.	.	.
Subsystem3	3
Subsystem4	4	1	1	.	.

Abbildung 37: Eine DSM mit Vorgaben und Fehlermarkierungen (nach Sangal et al. 2005)

Der Ansatz ist besonders gut geeignet für Schichtenarchitekturen, da sich diese selbst für umfangreiche Systeme einfach darstellen lassen: eine Softwarearchitektur ist genau dann geschichtet, wenn sie sich – wie in Abbildung 36 – als Dreiecksmatrix darstellen lässt.

Auch hierarchische Architekturen werden unterstützt, sie lassen sich übersichtlich darstellen, indem einzelne Zellen „aufgeklappt“ werden, so dass alle enthaltenen Architekturelemente als eigene Zeilen und Spalten dargestellt werden. Vorgaben für zusammengesetzte Architekturelemente gelten auch für die enthaltenen Elemente, sofern nicht anders spezifiziert.

Zusammenfassend lässt sich sagen, dass der Schwerpunkt des DSM-Ansatzes auf der Darstellung in Form von Matrizen liegt. Mit dem Ansatz lässt sich prüfen, ob die *Beziehungen* innerhalb einer Architektur den Vorgaben der Soll-Architektur entsprechen.

4.4.4 Der LISA-Ansatz

Der LISA-Ansatz¹⁷ (Language for Integrated Software Architecture) ist darauf ausgerichtet, architekturbezogene Aktivitäten im gesamten Architekturlebenszyklus zu unterstützen und zu integrieren (Weinreich und Buchgeher 2012). Der Ansatz besteht aus zwei zentralen Elementen: einem semi-formalen, technologieunabhängigen Metamodell für Architekturbeschreibungen (LISA-Modell) und einer Menge integrierter Software-Werkzeuge (LISA-Toolkit). Die Werkzeuge sind als Plugins für die Entwicklungsumgebung (IDE) Eclipse¹⁸ realisiert. Sie verwenden das Architektur-Metamodell für verschiedene architekturbezogene Aktivitäten, wie Analyse, Quelltextgenerierung, Dokumentation und Wissensmanagement (Buchgeher und Weinreich 2009b, 2010b, 2011).

¹⁷ lisa.se.jku.at

¹⁸ www.eclipse.org

Der LISA-Ansatz unterstützt unter anderem Prüfungen auf Architekturtreue zu einer Soll-Architektur. Dieser Aspekt des Ansatzes wird im Folgenden vorgestellt:

Der LISA-Ansatz selber verwendet den Begriff der Soll-Architektur nicht, sondern spricht von einer „high-level architecture“ sowie den Architekturvorgaben „erlaubte Beziehung“ und „Sichtbarkeitseinschränkungen“. Die high-level architecture besteht aus hierarchisch geschichteten Architekturelementen, erlaubte Beziehungen sind gerichtete Beziehungen zwischen zwei Architekturelementen, Sichtbarkeitseinschränkungen erlauben innerhalb eines zusammengesetzten Architekturelements die enthaltenen Elemente in private Elemente und andere öffentlich sichtbare Elemente zu unterteilen. Die öffentlich sichtbaren Elemente bilden die Schnittstelle eines zusammengesetzten Architekturelements (Buchgeher und Weinreich 2008).

Die mit dem LISA-Ansatz prüfbareren Vorgaben ähneln denen der hierarchischen SRM (geschichtete Architekturelemente, Vorgaben für Beziehungen, beliebige Programmiersprache), unterscheiden sich jedoch darin, dass die SRM *vorschreiben*, wo eine Beziehung erwartet wird, der LISA-Ansatz jedoch Beziehungen *erlaubt*. Ferner gehen die im LISA-Ansatz definierbaren Vorgaben über die SRM hinaus, indem sich Vorgaben für Schnittstellen definieren lassen (Buchgeher und Weinreich 2008).

Neben der Treue zu Soll-Architekturen kann der Ansatz auch die Treue zu dem Schichten-Architekturstil prüfen. Das LISA-Modell erlaubt zu diesem Zweck spezielle Architekturelemente des Typs Schicht (Buchgeher und Weinreich 2008). Die Definition dieses Architekturstils und seiner speziellen Vorgaben ist in das LISA-Modell integriert, es ist nicht möglich, die Konformanz des Quelltextes zu weiteren, benutzerdefinierten Stilen zu prüfen.

Ein Schwerpunkt des LISA-Ansatzes ist die *grafische* Darstellung und Modellierung der Architekturinformationen (Buchgeher und Weinreich 2009c, 2010b).

Der LISA-Ansatz legt – im Gegensatz zu dem SRM-Ansatz – einen Schwerpunkt auf die *Integration* von Quelltext und Architekturmodell (Buchgeher und Weinreich 2009a; Weinreich und Buchgeher 2012). Soweit möglich entnimmt der Ansatz die Architekturinformationen direkt aus dem Quelltext des Systems. Nicht im Quelltext verfügbare Architekturinformationen werden manuell ergänzt und im LISA-Modell innerhalb der Werkzeuge gespeichert. Die Integration von Quelltext und Architekturmodell ermöglicht dem LISA-Toolkit, Architekturverstöße unmittelbar zu melden, sobald der Quelltext oder das LISA-Modell geändert werden. Die Werkzeuge ziehen nur die tatsächlich von Änderungen betroffenen Anteile von Quelltext und Architektur in die Prüfung mit ein, um die Prüfdauer zu reduzieren.

Neuere Erweiterungen des LISA-Ansatzes unterstützen verschiedene aktuelle Sprachen und Komponentenmodelle. Das Quelltextmodell (auch als Quelltextstruktur bezeichnet) basiert sowohl auf reinen Quelltextdateien als auch auf ergänzenden Implementationsdateien wie beispielsweise Konfigurationsdateien für Komponenten (Weinreich et al. 2012).

4.4.5 Konformanzprüfung ohne explizite Soll-Architektur

Der Vollständigkeit halber seien an dieser Stelle Prüfansätze erwähnt, die kein oder kein vollständiges explizites Modell der Soll-Architektur definieren. Diese Ansätze für Architekturprüfungen zu nutzen hat den Nachteil, dass sich Architekturvorgaben nicht auf der Abstraktionsebene der Architektur definieren lassen und dass Architekturverletzungen auf Quelltextebene ermittelt werden. Werden Verstöße gemeldet, so bleibt es an der prüfenden Person überlassen, den Verstoß zu verstehen und nachzuvollziehen, wie der Verstoß gegen eine Vorgabe auf Quelltextebene mit den gewählten *Architekturvorgaben* zusammen hängt (Passos et al. 2010).

Ein solcher Prüfansatz wird von sogenannten „Source Code Query Languages“ verfolgt (Passos et al. 2010; Verbaere et al. 2008). Mit diesen Ansätzen lassen sich Vorgaben definieren, die der

Quelltext erfüllen soll, wie beispielsweise Quelltextkonventionen. So lassen sich auch Vorgaben definieren, die (implizit) auf einer Soll-Architektur beruhen.

Ein weiteres Beispiel sind „Relation Conformance Rules“ (Knodel und Popescu 2007; Postma 2003), diese definieren erlaubte und verbotene Beziehungen mit Hilfe von regulären Ausdrücken. Die Ausdrücke beschreiben Namensmuster für die betroffenen Quelltextelemente. Beispielsweise lässt sich die Regel definieren „A* is forbidden B*“, diese besagt, dass Elemente, deren Name mit dem Buchstaben A beginnt, keine Beziehung haben dürfen zu Elementen, die mit dem Buchstaben B beginnen.

4.4.6 Sonstige Werkzeuge

Dieser Abschnitt stellt eine Auswahl wissenschaftlicher und kommerzieller Werkzeuge für Prüfungen auf Architekturtreue vor. Beschreibungen weiterer Werkzeuge, über die hier genannten hinaus, finden sich beispielsweise in den Veröffentlichungen von van Eyck et al. und Raza et al. (van Eyck et al. 2011; Raza et al. 2006).

SAVE

Das Werkzeug SAVE (Software Architecture Visualization and Evaluation) unterstützt unter anderem Konformanzprüfungen (Duszynski et al. 2009; Knodel et al. 2008; Lindvall und Muthig 2008; Knodel und Popescu 2007; Knodel et al. 2006). Das Werkzeug zeigt die Treue zu einer Soll-Architektur.

Die Soll-Architektur in SAVE entspricht semantisch dem SRM-Ansatz inklusive der SRM-Erweiterungen für getypte Beziehungen und hierarchische Strukturen. Darüber hinaus können geschachtelte Architekturelemente innerhalb der Soll-Architektur als öffentlich gekennzeichnet werden. Die öffentlichen Elemente bilden die Schnittstelle des Eltern-Elements; auf die privaten Elemente darf von außerhalb des Eltern-Elements nicht zugegriffen werden.

Ferner unterstützt das Werkzeug Konformanzprüfungen mit Hilfe der oben genannten „Relation Conformance Rules“.

SAVE ist ein ausgereifter Forschungsprototyp. Mit Hilfe dieses Werkzeuges hat das Fraunhofer Institut IESE in mehrfachen Fallstudien die Nützlichkeit von Treueprüfungen in der Praxis gezeigt (Knodel et al. 2006; Knodel und Popescu 2007).

Mittlerweile unterstützt das Werkzeug (in der Version SAVE LiFe) direktes Feedback über Architekturverletzungen während der Programmierung (Knodel et al. 2008).

Sonargraph

Das Werkzeug Sonargraph¹⁹ (vormals SonarJ) ist ein kommerzielles Werkzeug. Es prüft die Treue zu erlaubten und verbotenen Beziehungen in hierarchischen Soll-Architekturen.

Dem Werkzeug liegt der Schichten-Architekturstil zu Grunde, ergänzt um sogenannte fachliche Schnitte. Beispiele für Schichten sind „Benutzungsschnittstelle“ und „Datenbankzugriff“. Fachliche Schnitte ergeben sich aus der Domäne des Softwaresystems, mögliche fachliche Schnitte sind beispielsweise „Vertrieb“ oder „Kunde“. Die fachlichen Schnitte liegen senkrecht zu den Schichten, so kann es in der Benutzungsschnittstelle Quelltext geben, der zum Vertrieb gehört, und auch in der Schicht namens Datenbankzugriff.

¹⁹ www.hello2morrow.com/products/sonargraph

Standardmäßig sind Verbindungen innerhalb der Schichten nur von oben nach unten und innerhalb der fachlichen Schnitte von links nach rechts erlaubt, es ist jedoch möglich, frei zu definieren, welche Beziehungen erlaubt sind.

Die Architektur lässt sich schachteln. Ein einzelnes Architekturelement kann intern wiederum aus Schichten und Schnitten bestehen. Ein einzelnes Architekturelement ist definiert über eine Schicht und einen Schnitt, so bildet beispielsweise der Schnitt Vertrieb in der Schicht Benutzungsschnittstelle ein Architekturelement.

Die Soll-Architektur wird anhand einer grafischen Darstellung definiert. Die Schichten und Schnitte sind als Rechtecke dargestellt, die erlaubten Beziehungen als Pfeile (siehe Abbildung 38).

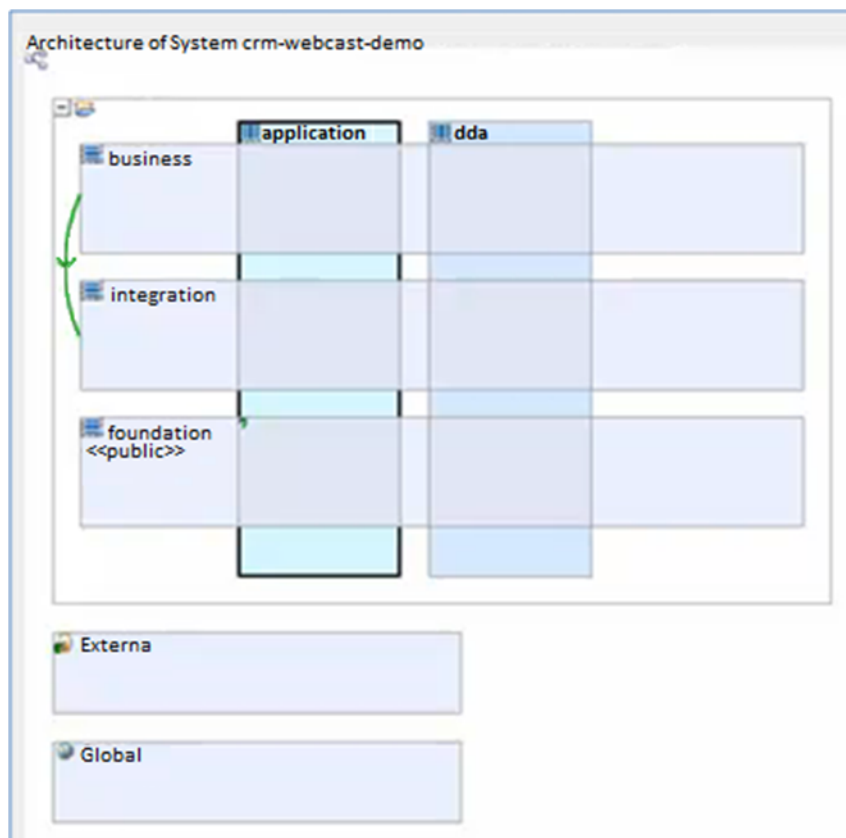


Abbildung 38: Grafische Darstellung von Schichten und Schnitten in Sonargraph²⁰

Das Werkzeug hält das Architekturmodell aus Gründen der Laufzeit im Hauptspeicher. Es steht mittlerweile auch als Plugin für Entwicklungsumgebungen zur Verfügung und ermöglicht somit Prüfungen direkt während der Programmierung.

²⁰ Quelle der Grafik: www.hello2morrow.com/videos/architecture

Sotoarc

Sotoarc²¹ ist ein kommerzielles Werkzeug. Es prüft ebenfalls die Konformanz der Ist-Architektur zu einer hierarchischen Soll-Architektur, die erlaubte und verbotene Beziehungen definiert. Ferner lassen sich Schnittstellenzugriffe prüfen: geschachtelte Architekturelemente können privat sein, auf diese geschachtelten Architekturelemente darf von außerhalb des Eltern-Elements nicht zugegriffen werden.

Das Werkzeug folgt dem Schichtenstil, standardmäßig darf von oben nach unten zugegriffen werden, es ist jedoch möglich, beliebige Beziehungen zu erlauben. Ferner lassen sich geschachtelte Architekturelemente als „nicht geschichtet“ markieren, in diesem Fall ist die Schichtenregel „Zugriff von oben nach unten“ nicht gültig, stattdessen lassen sich eigene Zugriffsregeln definieren.

Das Werkzeug hebt sich durch die grafische Modellierung der Architektur ab, beispielsweise werden Beziehungen als Halbkreise dargestellt. (siehe Abbildung 39).

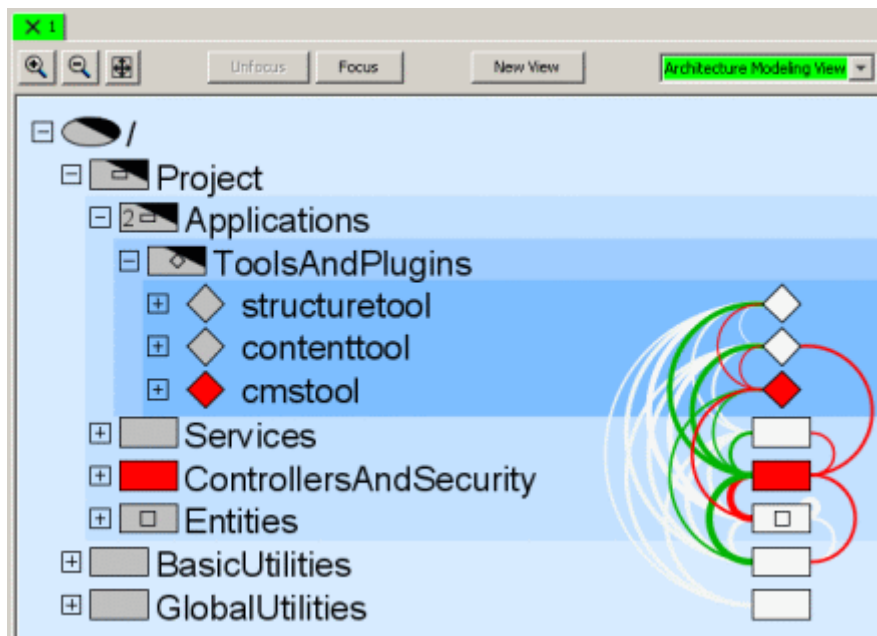


Abbildung 39: Grafische Darstellung der Prüfergebnisse in Sotoarc²²

Wie bei den meisten modernen Werkzeugen wird die Soll-Architektur– im Gegensatz zu den ursprünglichen SRMs – nicht komplett unabhängig vom Quelltext erstellt. Stattdessen erstellt Sotoarc aus Quelltextelementen wie Java-Packages und Java Package-Bäumen jeweils Architekturelemente und stellt diese grafisch dar. Die prüfende Person hat die Möglichkeit, diese aus dem Quelltext ermittelten Architekturelemente manuell anzupassen und die Architekturvorgaben innerhalb der Grafik zu ergänzen.

Das Werkzeug ist als Plugin für Entwicklungsumgebungen verfügbar und erlaubt die Prüfung während der Programmierung.

²¹ www.hello2morrow.com/products/sotoarc

²² Quelle der Grafik: www.hello2morrow.com/products/sotoarc

Sotograph

Das kommerzielle Werkzeug Sotograph²³ (Bischofberger et al. 2004) prüft die Konformanz zu erlaubten und verbotenen Beziehungen in geschachtelten Soll-Architekturen. Ferner lassen sich Schnittstellenvorgaben in Form privater geschachtelter Architekturelemente festlegen.

Die Soll-Architektur wird textuell festgelegt in einer werkzeugeigenen, textuellen Darstellung. Diese textuelle Beschreibung enthält die Architekturvorgaben sowie für jedes Architekturelement jeweils die Zuordnung zu Quelltextelementen.

Neben manuell festgelegten Vorgaben ist es möglich, Architekturelemente als Schichten zu definieren. In diesem Fall gilt die Regel des Schichtenstils, dass Schichten nur auf weiter unten liegende Schichten zugreifen dürfen.

Das Werkzeug lässt sich integriert mit Sotoarc nutzen.

Sämtliche Prüfungen innerhalb des Sotographen arbeiten auf einem Repository. Dieses Repository enthält die Quelltextstruktur innerhalb einer relationalen Datenbank.

Structure 101

Das kommerzielle Werkzeug Structure 101²⁴ (Sangwan et al. 2008) ähnelt Sotoarc und erlaubt im Prinzip die gleichen Regeln, hat jedoch im Gegensatz zu Sotoarc eine klassische grafische Darstellung der Architektur (waagerechte Rechtecke und Pfeile). Ferner lässt sich die Architektur alternativ in einer Abhängigkeitsmatrix (Dependency Structure Matrix, s.o.) darstellen.

4.4.7 Zusammenfassung

Tabelle 4-1 gibt einen Überblick über die bisher vorgestellten Ansätze zur Architekturprüfung. Die Tabelle stellt dar, welche Arten von Architekturvorgaben unterstützt werden. Die meisten Ansätze und Werkzeuge können die Konformanz zu Soll-Architekturen prüfen. Unterschiede in der Art der Soll-Architekturen (wie typisierte Beziehungen, Soll-Beziehungen oder hierarchische Architekturen mit Enthältbeziehungen) sind hier nicht ausgewiesen, sondern jeweils oben in der textuellen Darstellung der Ansätze und Werkzeuge erläutert. Einige Ansätze prüfen die Konformanz zu Vorgaben auf Quelltextebene. Mit diesen lassen sich Vorgaben, die konzeptionell auf der Architekturebene liegen, nicht direkt prüfen. Stattdessen muss die prüfende Person die Architekturvorgaben in Vorgaben auf Quelltextebene übersetzen und die auf Quelltextebene aufgedeckten Verstöße auf Ebene der Architekturvorgaben interpretieren. Einige Ansätze erlauben die Konformanzprüfung zu fest vorgegebenen Stilen, entweder zum Schichtenstil oder zu Schichtenstilen mit fachlichen Schnitten. Keiner der Ansätze unterstützt die Prüfung zu Vorgaben in Form frei definierbarer Architekturstile.

²³ www.hello2morrow.com/products/sotograph

²⁴ structure101.com

Ansatz / Werkzeug	Konformanzprüfung zu folgenden Architekturvorgaben			
	Soll- Architekturen (verschiedene Varianten)	Vorgaben auf Quelltextebene	Schichtenstil (teilweise mit fachlichen Schnitten)	Benutzer- definierte Stile
SRM original	X	-	-	-
SRM getypt	X	-	-	-
SRM hierarchisch	X	-	-	-
DSM	X	-	X	-
LISA	X	-	X	-
Source Code Query Lan- guages	-	X	-	-
Relation Con- formance Rules	-	X	-	-
SAVE	X	X	X	-
Sonargraph	X	-	X	-
Sotograph	X	-	X	-
Structure 101	X	-	X	-

Tabelle 4-1: Überblick über die unterstützten Architekturvorgaben innerhalb der vorgestellten Ansätze und Werkzeuge

Neben der Art der Architekturvorgaben gibt es weitere relevante Besonderheiten und Unterscheidungsmerkmale, diese sind in der Tabelle 4-2 aufgelistet: (1) die Möglichkeit zur Prüfung während der Programmierung, (2) die Integration von der für die Ermittlung der Ist-Architektur benötigten Zusatzinformationen in den Quelltext und (3) die Definition eines sprachunabhängigen, expliziten Metamodells für die Architekturvorgaben.

Ansatz / Werkzeug	Prüfung während der Programmie- rung	Integration Archi- tekturbeschreibung / Quelltext	Sprachunabhängi- ges, explizites Me- tamodell für die Architekturvorga- ben
SRM original	-	-	-
SRM getypt	-	-	-
SRM hierarchisch	-	-	-
DSM	X	-	-
LISA	X	-	X (für Soll- Architekturen und Schichtenstil)
Source Code Query Languages	Werkzeugabhängig	-	-
Relation Confor- mance Rules	Werkzeugabhängig	-	-
SAVE	X	-	-
Sonargraph	X	-	-
Sotograph	-	-	X
Structure 101	X	-	-

Tabelle 4-2: Relevante Besonderheiten der vorgestellten Ansätze und Werkzeuge

Neben den hier vorgestellten, für diese Arbeit relevanten Ansätzen, gibt es weitere Ansätze, die sich mit ausgewählten technischen Details oder Spezialfällen beschäftigen. Beispiele sind Prüfansätze für bestimmte Technologien wie REST (Richardson et al. 2007; Athanasopoulos et al. 2011), Prüfansätze für spezielle mehrsprachige Systeme (Rahimi und Khosravi 2010; Deissenboeck et al. 2010) und Varianten der vorgestellten Ansätze (Rosik et al. 2008).

Der folgende Abschnitt stellt verwandte Forschungsfelder vor und grenzt sie gegen das Feld der Prüfungen auf Architekturtreue ab. Anschließend werden erste Ansätze vorgestellt, die sich mit der Treue zu Architekturstilen befassen.

4.5 Abgrenzung zu verwandten Forschungsfeldern

4.5.1 Die Unified Modeling Language UML

Die UML²⁵ (Unified Modeling Language) ist eine standardisierte Modellierungssprache für die Software-Entwicklung. Bei UML handelt es sich in erster Linie um eine grafische Sprache mit einer Vielzahl an Diagrammtypen. Integriert in die UML ist zusätzlich die textuelle Sprache OCL (Object Constraint Language), mit der sich Einschränkungen ausdrücken lassen. Zahlreiche Werkzeuge unterstützen die Modellierung mit UML. Viele unterstützen die Quelltextgenerierung aus UML-Diagrammen, einige Werkzeuge ermöglichen auch das Generieren von UML-Diagrammen aus Quelltext.

Neben vielen anderen Aspekten der Software-Entwicklung lassen sich mit UML auch Softwarearchitekturen beschreiben und dokumentieren. UML ist nicht auf Treueprüfungen für Softwarearchitekturen ausgerichtet. Da UML über ein Metamodell erweiterbar ist, ließe sie sich jedoch für verschiedenste Prüfungen auf Architekturtreue verwenden, indem entsprechende Metamodelle und Architekturvorgaben ergänzt werden. So ließe sich auch der in dieser Arbeit vorgestellte Ansatz im Prinzip mit UML umsetzen.

Ob UML und OCL angemessene und leichtgewichtig verwendbare Sprachen für Softwarearchitekturen sind, wird nach wie vor diskutiert (Shaw und Clements 2006; Weinreich und Buchgeher 2012; Buchgeher und Weinreich 2009a). Es existieren vereinzelte Forschungsansätze, die UML und OCL für Konformanzprüfungen verwenden (beispielsweise Rahimi und Khosravi 2010).

4.5.2 Modellgetriebene Software-Entwicklung

Die modellgetriebene Software-Entwicklung (MDSD) bezeichnet „Software-Entwicklungsprozesse, bei denen Modelle im Mittelpunkt stehen und als eigenständige Entwicklungsartefakte genutzt werden“ (Baier et al. 2009, S. 96). In diesem Kontext finden sich mehrere Forschungsströmungen, unter anderem die MDA (Model Driven Architecture), eine Standardisierungsinitiative der OMG (Object Management Group)²⁶, sowie domänenspezifische Sprachen (Posch et al. 2011). Modellgetriebene Ansätze zielen üblicherweise darauf, Softwaresysteme auf abstrakter Ebene zu beschreiben, teilweise mit mehreren Modellen, die Modelle gegebenenfalls zu prüfen, zu transformieren und aus ihnen Quelltext zu generieren (Baier et al. 2009). Die Modelle können auf verschiedene Eigenschaften geprüft werden, beispielsweise können Entwicklungsteams den Einfluss von Architekturentwürfen auf das Laufzeitverhalten oder die Zuverlässigkeit bereits validieren, bevor sie das entsprechende System erstellen (Becker et al. 2009; Franz Brosch et al. 2011). Architekturprüfungen hingegen zielen darauf, *bestehenden Quelltext* auf *Konformanz* zu Architekturvorgaben zu prüfen. Sie setzen keine modellgetriebene Entwicklung voraus, sind jedoch prinzipiell nutzbringend mit ihr vereinbar, wenn Anteile des Quelltextes manuell erstellt werden.

Manche Autoren sprechen von einem *modellbasierten* Ansatz, wenn ein Ansatz zwar explizite Modelle verwendet, jedoch nicht auf das Generieren von Softwaresystemen zielt. Im Bereich der Prüfungen auf Architekturtreue versteht sich der oben genannte LISA-Ansatz als ein modellbasierter Ansatz, da er jeweils ein explizites Modell der Soll- und der Ist-Architektur in den

²⁵ www.uml.org

²⁶ www.omg.org/mda

Mittelpunkt stellt (Buchgeher und Weinreich 2010a). Auch die stilbasierte Architekturprüfung ordnet sich in den Bereich modellbasierter Ansätze ein, wobei sie die Besonderheit beinhaltet, dass ein mit Architekturinformationen annotierten Quelltext als *ausschließliche* Quelle für das Modell der Ist-Architektur verwendet werden kann.

Aktuell lassen sich Forschungsvorhaben beobachten, die Aspekte modellgetriebener und modellbasierter Ansätze verbinden. Ein Beispiel ist der zur Zeit entwickelte ADVERT-Ansatz für langlebige, komponentenbasierte Softwaresysteme (Konersmann et al. 2013). Der ADVERT - Ansatz soll unter anderem Quelltext aus Architekturmodellen generieren und die aktuelle Architektur aus dem annotierten Quelltext ermitteln können. Dabei soll der Quelltext – wie bei der stilbasierten Architekturprüfung – das gesamte Architekturmodell beinhalten. Der Quelltext soll nach einem Verfahren von Balz annotiert werden (Balz 2012). Balz' Verfahren ermöglicht es, Architekturmodelle für die *Laufzeitsicht* aus annotiertem Quelltext zu extrahieren. Die vorliegende Arbeit hingegen betrachtet die *statische Sicht stilbasierter* Architekturen.

4.5.3 Architekturbeschreibungssprachen

Architekturbeschreibungssprachen (Architecture Description Languages, ADLs) sind formale, textuelle Beschreibungssprachen für Softwarearchitekturen (Medvidovic und Taylor 2000). Sie erlauben es, bereits vor Beginn der Programmierung Softwarearchitekturen zu beschreiben und diese Beschreibungen werkzeuggestützt auf vielfältige Eigenschaften zu prüfen. Die typische ADL ist auf die Prüfung ausgewählter Eigenschaften spezialisiert, wie beispielsweise Laufzeitverhalten und Speicherbedarf.

Prüfungen auf Architekturtreue hingegen basieren immer auf dem tatsächlichen Softwaresystem beziehungsweise auf dessen Quelltext, sie stellen eine Verbindung zwischen Architektur und Quelltext her und prüfen die Architekturkonformanz auf Basis des Quelltextes.

Zwar sind Architekturbeschreibungssprachen nicht darauf ausgerichtet, eine enge Verbindung zwischen Quelltext und Softwarearchitektur herzustellen sowie bestehende Quelltexte kontinuierlich auf Architekturtreue zu prüfen. Jedoch lassen sich die Beschreibungsmöglichkeiten für Architekturen, die in modellbasierten Ansätzen zur Prüfung auf Architekturtreue verwendet werden, als ADL betrachten (Weinreich und Buchgeher 2010, 2012).

4.6 Erste bisherige Möglichkeiten zur Prüfung auf Stiltreue

Die in den vorherigen Abschnitten vorgestellten Ansätze und Werkzeug zur Prüfung auf Architekturtreue prüfen die Treue zu Soll-Architekturen oder zu Vorgaben auf Quelltextebene. Einige Ansätze erlauben zusätzlich die Treue zum Schichtenstil zu prüfen, ein Ansatz erweitert den Schichtenstil um fachliche Schnitte. Die Regeln des Schichtenstils sind bereits von den Ansätzen bzw. Werkzeugen vorgegeben. Keiner der Ansätze erlaubt es, die Treue zu beliebigen Stilen zu prüfen.

Der Bedarf an Prüfungen auf Stiltreue für beliebige Architekturstile wird in verschiedenen Veröffentlichungen thematisiert (Passos et al. 2010; Clements und Shaw 2009; Shaw und Clements 2006). Passos et al. zeigen am Beispiel des Model-View-Controller-Stils, dass die bisherigen Prüfansätze nicht in der Lage sind, die Treue zu diesem Stil zu prüfen. Shaw und Clements betonen allgemeiner, dass Konformanzprüfungen und die Architekturermittlung aus Quelltexten nach wie vor nicht ausreichend seien, insbesondere seien Architekturstile im Quelltext nicht zu erkennen und die Einhaltung von Architekturstilvorgaben nicht prüfbar.

Im Folgenden werden erste bestehende Ansätze vorgestellt, die – wie diese Arbeit – darauf zielen, die Treue von Softwaresystemen zu beliebigen Architekturstilen zu prüfen. Die folgende Präsentation der Ansätze konzentriert sich auf den Aspekt der Prüfung auf Stiltreue.

4.6.1 ArchJava

ArchJava (Aldrich et al. 2002; Aldrich 2008) ist eine aus dem universitärem Umfeld stammende Spracherweiterung für Java. ArchJava zielt darauf, Quelltext und Architekturinformationen zu verbinden. Da es sich um eine Spracherweiterung handelt, ist dieser Ansatz für andere Programmiersprachen nicht einsetzbar.

ArchJava ergänzt die Sprache Java um Architekturkonstrukte, die typischerweise in Architekturbeschreibungssprachen (ADLs) verwendet werden, wie Komponenten, Konnektoren und Ports. Der Ansatz erlaubt, diese im Quelltext vorkommenden Konstrukte mit der Architektur zu verbinden. Es wird jedoch nicht unterstützt, die ursprünglich in Java vorkommenden Sprachkonstrukte, wie Klassen oder Packages, der Architektur zuzuordnen.

Der Ansatz wird durch mehrere in Eclipse integrierte Werkzeuge unterstützt. Mittlerweile ist es möglich, zu einem ArchJava-System die Architekturbeschreibung zu exportieren, im Format der Architekturbeschreibungssprache (ADL) ACME (Aldrich 2008).

Für ACME-Beschreibungen wiederum stehen Werkzeuge zur Verfügung, mit denen sich die Beschreibungen auf verschiedenste Eigenschaften hin prüfen lassen. Die zu prüfenden Eigenschaften werden in einer formalen, textuellen Beschreibungssprache formuliert. Auch bestimmte Vorgaben für Architekturstile lassen sich so im Prinzip formulieren (Aldrich 2008).

Dieser Ansatz ist keine Architektur-Konformanzprüfung im eigentlichen Sinne, da er den Umweg über eine zu exportierende Architekturbeschreibung geht und so keine direkte Prüfung des Quelltextes auf Architekturtreue unterstützt. Die gemeldeten Architekturverstöße beziehen sich auf die ACME-Beschreibung und nicht auf den implementierten Quelltext.

Das im Rahmen der vorliegenden Arbeit entwickelte Metamodell für Architekturstile (siehe Definitionen 3-1 und 3-2), welches auf in Wissenschaft und Praxis relevanten Architekturstilen basiert, lässt sich mit ACME nicht direkt ausdrücken, hier fehlen die passenden Abstraktionen und Ausdrucksmittel, beispielsweise um Architekturelemente direkt mit gerichteten Beziehungen zu verbinden. Eine 1-zu-1-Abbildung der Modellelemente des Metamodells für Architekturstile zu den Modellelementen in ACME ist nicht möglich, dadurch wäre eine Beschreibung von Architekturstilen mit ACME umständlich, die prüfende Person könnte ihren Architekturstil nicht mit den gewohnten Konzepten beschreiben.

Der Ansatz ist nach eigenen Aussagen der Autoren als schwergewichtig anzusehen. Er erfordert einen hohen Aufwand bei der Einarbeitung und Verwendung, im Gegensatz zu dem leichtgewichtigen Ansatz der SRMs (Aldrich 2008).

4.6.2 ArchMapper

ArchMapper ist ein im wissenschaftlichen Kontext entwickeltes Werkzeug (Giesecke et al. 2010). ArchMapper ist den ersten Veröffentlichungen der stilbasierten Architekturprüfung zeitlich nachgelagert. Das Werkzeug erlaubt die Prüfung von Java-Systemen auf die Treue zu Middleware-basierten Stilen. Beispielsweise lässt sich mit dem Ansatz prüfen, ob ein mit dem Rahmenwerk Spring implementiertes System sich an die von Spring definierten Architekturvorgaben hält.

Der ArchMapper verwendet für die Beschreibung von Softwarearchitekturen und Architekturvorgaben die ADL ACME, mit den oben genannten Einschränkungen bezüglich der Modellierungskonzepte. Daraus folgt, dass sich das im Rahmen dieser Arbeit herausgearbeitete Verständnis von Architekturstil mit dem ArchMapper nicht bruchfrei abbilden lässt. Neuere Versionen des ArchMappers unterstützen auch eine XML²⁷-basierte Stilbeschreibung.

Mit ArchMapper lässt sich die Treue zu erlaubten Beziehungen innerhalb des vorgegebenen Stils prüfen. Es lassen sich Beziehungstypen unterscheiden (Benutzung einer Komponente versus Konstruktoraufruf). Ferner lassen sich Regeln implementieren, die sich nicht auf der im ArchMapper mit ACME beschriebenen Architekturebene bewegen. Giesecke et al. sprechen in diesem Zusammenhang von Regeln bezüglich der internen Implementierung von Klassen. Auf diese Weise scheinen sich zumindest auch einige der Schnittstellenregeln im Sinne dieser Arbeit (siehe Kapitel 3) formulieren zu lassen. Beispielsweise lässt sich die Regel prüfen, dass alle Klassen innerhalb eines bestimmten Architekturelement-Typs nur Setter und Getter haben dürfen und keine weiteren Operationen.

4.6.3 Abgrenzung zur stilbasierten Architekturprüfung

Die folgenden Tabellen geben einen Überblick über die Eigenschaften der beiden vorgestellten Ansätze zur Prüfung auf Stiltreue und listen in der unteren Zeile die Eigenschaften der in dieser Arbeit präsentierten stilbasierten Architekturprüfung. Wie die stilbasierte Architekturprüfung diese Eigenschaften erreicht, erläutern die nachfolgenden Kapitel.

Tabelle 4-3 gibt einen Überblick über Eigenschaften, die einen direkten Einfluss auf das Prüfungsergebnis haben: die unterstützten Architekturvorgaben, Sprachen und Quelltextelemente. Tabelle 4-4 listet Besonderheiten der Ansätze.

²⁷ Extensible Markup Language, www.omg.org/technology/xml

Ansatz / Werkzeug	Konformanzprüfung zu benutzerdefinierten Stilen	Für bestehende Programmiersprachen einsetzbar	Quelltextelemente frei wählbar (z.B. Klassen, Packages, Komponenten)	Anmerkungen
ArchJava	- (Keine Konformanzprüfung im eigentlichen Sinne, es wird nicht der Quelltext, sondern eine in der ADL ACME verfasste Architekturbeschreibung auf Stiltreue geprüft, s.o.)	-	-	ArchJava ist eine Java-Spracherweiterung, sie ergänzt Java um eigene Quelltextelemente. Die ursprünglichen Java-Quelltextelemente (Klassen, Packages) können keinem Architekturelement zugeordnet werden. Kann ausschließlich Systeme prüfen, die mit der Spracherweiterung entwickelt wurden.
ArchMapper	X	X	X	Verwendet u.a. die ADL ACME für die Beschreibung von Stilen, das führt zu den oben genannten Einschränkungen. Ist der stilbasierten Architekturprüfung zeitlich nachgelagert.
<i>Die stilbasierte Architekturprüfung</i>	X	X	X	

Tabelle 4-3: Überblick über die unterstützten Architekturvorgaben, Quelltextelemente und Programmiersprachen innerhalb der vorgestellten Ansätze zur Prüfung auf Stiltreue und in der stilbasierten Architekturprüfung

Ansatz / Werkzeug	Prüfung während der Programmierung	Integration Architektur- beschreibung / Quelltext	Sprachunabhängiges, explizites Metamodell für die Architekturvorgaben
ArchJava	-	X Eingeschränkt auf die Programmiersprache ArchJava, nur spezielle Quelltextelemente als Architekturelement markierbar	Lediglich implizites, sehr umfangreiches Metamodell über die ADL ACME, schwergewichtig
ArchMapper	X	-	-
<i>Die stilbasierte Architektur- prüfung</i>	X	X	X Das Metamodell berücksichtigt praxis- relevante Stile, ist auf objektorientierte und imperative Sprachen ausgerichtet

Tabelle 4-4: Relevante Besonderheiten der vorgestellten Ansätze zur Prüfung auf Stiltreue und der stilbasierten Architekturprüfung

4.7 Zusammenfassung

Der Begriff der Architekturprüfung steht in dieser Arbeit kurz für statische Prüfungen auf Architekturtreue, auch als statische Prüfungen auf Architekturkonformanz bezeichnet. In diesem Kapitel wurde der Stand von Architekturprüfungen in Forschung und Praxis vorgestellt.

Begonnen wurde mit einer Konkretisierung des von Architekturprüfungen adressierten Problems: der Erosion von Softwaresystemen. Erosion ist definiert als die Ansammlung von Verstößen gegen Architekturvorgaben. Es lassen sich versehentliche und beabsichtigte Verstöße unterscheiden. Abhängig von der Art des Verstoßes sind unterschiedliche Maßnahmen und Prüfkriterien gefordert.

Anschließend folgte ein Überblick über das generelle Konzept von Architekturprüfungen. Die verschiedenen Ansätze zur Architekturprüfungen haben drei typische Berechnungsschritte gemeinsam: (1) statische Analyse des Quelltextes, (2) Ist-Architektur berechnen, (3) Architektur prüfen.

Die Darstellung der einzelnen Ansätze zur Architekturprüfung begann mit dem Ansatz der Software-Reflexionsmodelle (SRM). Der Ansatz dient als ein Ausgangspunkt für die stilbasierte Architekturprüfung und wurde darum ausführlich vorgestellt. SRMs sind weit verbreitet in Wissenschaft und Praxis, sie werden vielfach zitiert, nach ihrem Prinzip arbeiten viele Prüfwerkzeuge. Der Ansatz der SRM ist – wie die meisten bisher bestehenden Ansätze – auf Architekturvorgaben in Form von Soll-Architekturen ausgerichtet.

Im Anschluss an die SRM wurden weitere aktuelle Ansätze und Werkzeuge zur Architekturprüfung vorgestellt, die ebenfalls in erster Linie auf Soll-Architekturen zielen. Diese Ansätze erlauben es nicht, beliebige Stile zu beschreiben und Softwaresysteme auf die Einhaltung dieser Stile zu prüfen.

Zur Einordnung wurden verwandte Forschungsfelder vorgestellt und gegen den Themenbereich dieser Arbeit abgegrenzt.

Abschließend wurden zwei erste Ansätze vorgestellt, die für Architekturvorgaben in Form von Stilen entworfen wurden. Einer der Ansätze, der ArchMapper, ist der Veröffentlichung des Konzepts der stilbasierten Architekturprüfung zeitlich nachgelagert. Er enthält – im Gegensatz zur stilbasierten Architekturprüfung – kein explizites Metamodell für Architekturstile und erlaubt es nicht, die Architekturinformationen in den Quelltext zu integrieren.

Der zweite Ansatz, ArchJava, arbeitet mit einer eigens definierten Sprache, einer Erweiterung der Programmiersprache Java. Möchte ein Entwicklungsteam sein System mit ArchJava auf Stiltreue prüfen, so kann es keine andere Programmiersprache verwenden. Die in Java bestehenden Quelltextelemente, wie Klassen und Packages, lassen sich mit diesem Ansatz nicht berücksichtigen. Stattdessen definiert die Spracherweiterung eigene Quelltextelemente, nur diese eigenen Quelltextelemente lassen sich als Teil von Architekturelementen verwenden.

Die stilbasierte Architekturprüfung hingegen kann bestehende Quelltextelemente nutzen, so ist sie für verschiedene Sprachen einsetzbar und kann auch nachträglich zur Prüfung bereits bestehender Systeme genutzt werden. Das folgende Kapitel stellt das Konzept der stilbasierten Architekturprüfung vor und erläutert, wie die hier genannten Eigenschaften des Ansatzes realisiert sind.

5 Das Konzept der stilbasierten Architekturprüfung

Kapitel 3 hat die Vorteile stilbasierter Softwaresysteme aufgezeigt. Vorteile, die verloren gehen, wenn die Systeme erodieren. Mit den bisherigen, in Kapitel 4 vorgestellten Ansätzen zur Architekturprüfung lässt sich die Erosion stilbasierter Systeme nicht ausreichend aufdecken.

Mit der in diesem Kapitel präsentierten, stilbasierten Architekturprüfung wurde ein Ansatz geschaffen, mit dem sich Verstöße von Softwaresystemen gegen Stile ermitteln lassen. Entwicklungsteams können der Erosion entgegenwirken, indem sie die mittels dieses Ansatzes aufgedeckten Verstöße korrigieren. So können die Teams langfristig von den Vorteilen stilbasierter Systeme profitieren.

Abschnitt 5.1 erläutert, wie die stilbasierte Architekturprüfung die in dieser Arbeit aufgezeigten Teilprobleme adressiert, und grenzt den präsentierten Ansatz gegen bisherige Ansätze ab. In Abschnitt □ folgt ein Überblick über den Ablauf der stilbasierten Architekturprüfung, in dem die drei Teilaufgaben des Ansatzes kurz vorgestellt werden. Um die Darstellung der stilbasierten Architekturprüfung übersichtlicher zu gestalten, ist sie in ein Kernkonzept und drei Ausbaustufen aufgeteilt. Die Abschnitte 5.3 bis 5.5 stellen das Kernkonzept vor. Anschließend beschreibt Kapitel 6 die Ausbaustufen.

5.1 Beitrag der stilbasierten Architekturprüfung

Mit der stilbasierten Architekturprüfung lässt sich der *Quelltext* von Softwaresystemen auf die Treue zu *Architekturstilen* prüfen.

Die stilbasierte Architekturprüfung greift bei allen in Abschnitt 4.1 genannten Teilproblemen der Architekturerosion. Die für die einzelnen Teilprobleme verfolgten Lösungsansätze werden im Folgenden vorgestellt. Hierbei geht es um die Sicht der Anwenderinnen und Anwender auf die stilbasierte Architekturprüfung; eine detaillierte Beschreibung der Lösungsansätze und ihrer Formalisierung folgt in den anschließenden Abschnitten.

Teilproblem 1: *Zusammenhang zwischen Quelltext und Architekturvorgaben ist unbekannt*

Lösungsansatz 1: Die stilbasierte Architekturprüfung zeigt den Zusammenhang zwischen Quelltext und Architektur explizit auf. So nehmen Teammitglieder die Architektur bei der Programmierung stärker wahr und wissen, wie ihr Quelltext mit der Architektur zusammenhängt.

Teilproblem 2: *Verstöße werden übersehen*

Lösungsansatz 2: Wenn Entwicklerinnen oder Entwickler versehentlich einen Verstoß erzeugen, dann werden sie zur Entwicklungszeit unmittelbar darauf hingewiesen, direkt in der Entwicklungsumgebung. Sie können ihren Quelltext korrigieren, bevor sie ihn weitergeben oder in ein gemeinsames Repository einchecken.

Teilprobleme 3 und 4: *Beabsichtigte Verstöße: Provisorien und bewusste Ausnahmen*

Lösungsansatz 3: Architektinnen und Architekten können ihr gesamtes System regelmäßig, automatisiert daraufhin überprüfen, ob es Verstöße enthält, Architekturreviews durchführen und bei Bedarf entsprechende Restrukturierungen planen. Es ist möglich, die Prüfungen beispielsweise in bereits bestehende Prozesse, wie wöchentliche Builds oder Testläufe, einzubinden.

Alle drei Lösungsansätze gemeinsam bilden die stilbasierte Architekturprüfung. Die Lösungsansätze 2 und 3 sind hierbei namensprägend: mit ihnen lässt sich prüfen, ob die Architektur eines Softwaresystems die Vorgaben des zugehörigen Architekturstils einhält. Für diese Prüfung benötigt man den Zusammenhang zwischen Quelltext und Architektur, der in Lösungsansatz 1 hergestellt wird. Dieser Zusammenhang dient somit nicht nur zur Information der Entwicklerinnen und Entwickler, sondern auch als Eingangsinformation für die Prüfung. Das Konzept der stilbasierten Architekturprüfung sieht vor, dass alle drei Lösungsansätze durch ein Software-Werkzeug unterstützt werden.

Die stilbasierte Architekturprüfung unterscheidet sich signifikant von den in Kapitel 4 vorgestellten, bisherigen Ansätzen zur Prüfung auf Architekturtreue:

- Bei der stilbasierten Architekturprüfung stehen den Entwicklerinnen und Entwicklern die Informationen über die *stilbasierte* Ist-Architektur und den Zusammenhang zwischen dieser Architektur und dem Quelltext unmittelbar zur Verfügung. Diese Informationen explizit zur Verfügung zu stellen und direkt mit dem Quelltext zu verbinden, ohne Einschränkung auf eine bestimmte Programmiersprache, ist ein neuer Beitrag der stilbasierten Architekturprüfung.

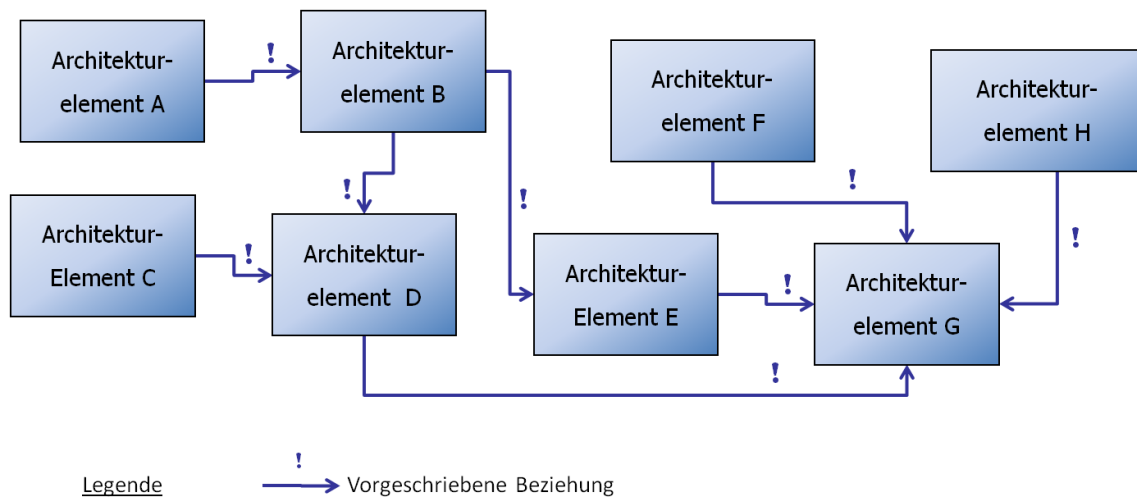


Abbildung 40: Bei Architekturvorgaben in Form von Soll-Architekturen wird jedes Architektur-element und jede Beziehung einzeln aufgezählt

Mit Stilen lassen sich nicht nur einheitliche Vorgaben für mehrere Elemente innerhalb eines Softwaresystems definieren, sondern auch einheitliche Vorgaben für *mehrere* Systeme (siehe Abbildung 43). Die stilbasierte Architekturprüfung unterstützt die systemübergreifende Wiederverwendbarkeit von Stilen, indem sie erlaubt, ein und dieselbe Stilbeschreibungen für die Prüfung verschiedener Systeme zu verwenden. So brauchen Unternehmen ihre wiederverwendeten Stile *nur einmal* zu beschreiben. Bei den bisherigen Treueprüfungen lassen sich die Beschreibungen der Architekturvorgaben nicht wiederverwenden, da die dort unterstützten Soll-Architekturen systemspezifisch sind. Evolvieren Architekturen, so gilt für die bisherigen Ansätze: die Beschreibungen von Architekturvorgaben in Form von Soll-Architekturen müssen entsprechend geändert werden. Für die stilbasierte Architekturprüfung hingegen gilt: Entwicklungsteams können die Beschreibung von Architekturvorgaben in Form eines Architekturstils unverändert weiterverwenden, sie genießen somit den Vorteil, dass der Aufwand für Änderungen entfällt.

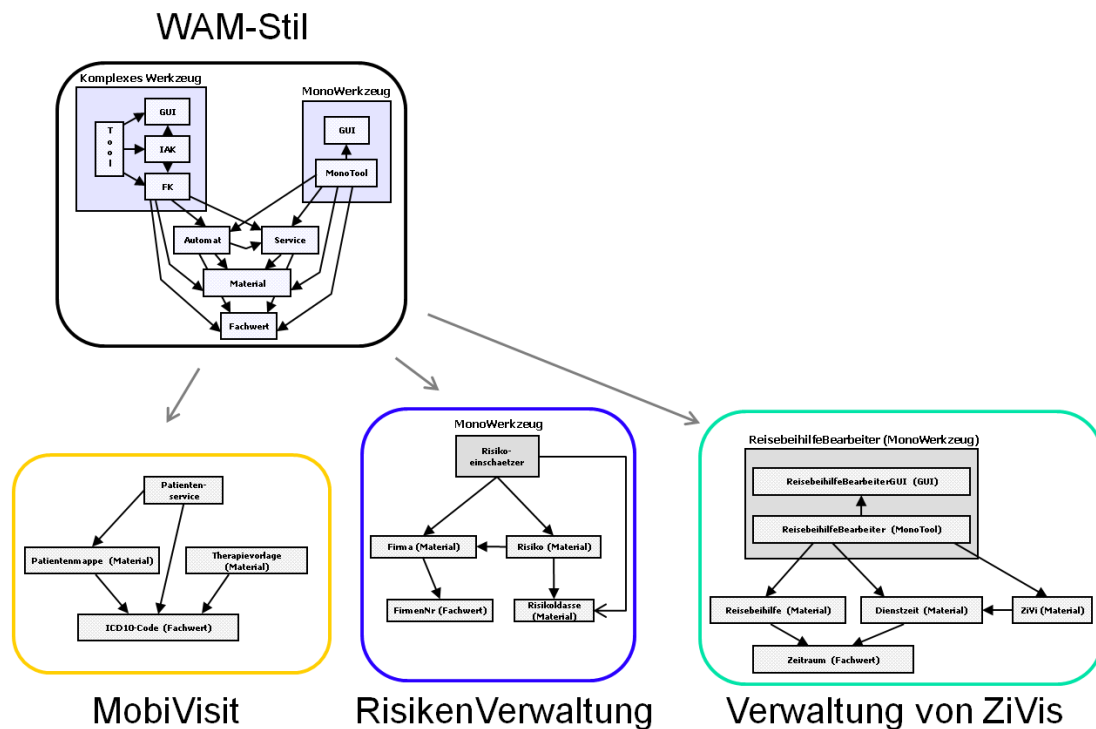


Abbildung 43: Die Vorgaben eines Stils gelten für alle Systeme, die diesem Stil folgen.

- Die zwei in Abschnitt 4.6 vorgestellten, ersten Ansätze zur Prüfung auf *Stiltreue* bieten – im Gegensatz zur stilbasierten Architekturprüfung – kein auf Stile ausgerichtetes, explizites Metamodell für die Beschreibung der Architekturvorgaben. Ein Ansatz (ArchJava) prüft nicht den Quelltext direkt, sondern Architekturbeschreibungen. Ferner beinhalten die Ansätze kein sprachunabhängiges Konzept, um die Architekturbeschreibung in den Quelltext stilbasierter Softwaresysteme zu integrieren.

5.2 Die stilbasierte Architekturprüfung im Überblick

Die stilbasierte Architekturprüfung besteht aus drei Teilaufgaben:

1. die stilbasierte Ist-Architektur ermitteln,
2. den Architekturstil beschreiben und
3. Verstöße der Ist-Architektur gegen den Stil berechnen.

Die Teilaufgaben 1 und 2 dienen dazu, die Eingangsinformationen für die Prüfung zu ermitteln. Teilaufgabe 3 stellt die eigentliche Prüfung dar, sie berechnet Verstöße der stilbasierten Ist-Architektur gegen den Architekturstil. Die stilbasierte Architekturprüfung wird mit Hilfe eines Werkzeugs durchgeführt. Abbildung 44 visualisiert das Grundprinzip des Ansatzes.

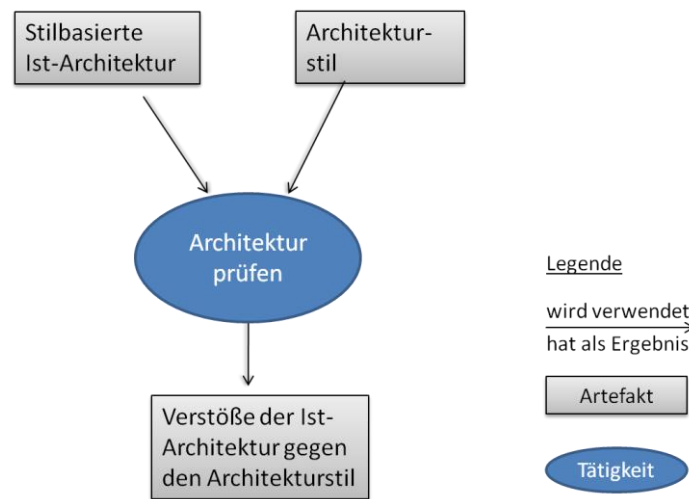


Abbildung 44: Grundprinzip der stilbasierten Architekturprüfung

An dieser Stelle soll ein kurzer Überblick über die Teilaufgaben gegeben werden. Die daran anschließenden Abschnitte erläutern die drei Teilaufgaben im Detail und zeigen jeweils den neuen Beitrag der stilbasierten Architekturprüfung auf.

Teilaufgabe 1: Die stilbasierte Ist-Architektur wird soweit möglich automatisiert aus dem Quelltext entnommen. Da sich nicht alle Aspekte von Ist-Architekturen eindeutig aus dem Quelltext entnehmen lassen, werden in diesem Schritt manuell ergänzte Zusatzinformationen über die Architektur erfasst. Die stilbasierte Architekturprüfung verknüpft diese Zusatzinformationen mit dem Quelltext. So können die Zusatzinformationen sowohl für die werkzeuggestützte Berechnung der Ist-Architektur genutzt werden als auch dafür, Entwicklerinnen und Entwicklern den Zusammenhang zwischen Quelltext und Architektur zu verdeutlichen. Dies vereinfacht die Programmierung und reduziert versehentliche Verstöße.

Die Zusatzinformationen können unverändert für erneute Prüfungen wiederverwendet werden. Nur wenn sich die Architektur eines Systems ändert, müssen Entwicklungsteams die entsprechenden Zusatzinformationen ergänzen.

Teilaufgabe 2: Die stilbasierte Architekturprüfung bietet Entwicklungsteams die Möglichkeit, ihren individuell gewählten Architekturstil zu beschreiben. Diese Beschreibung enthält die einzuhaltenden Regeln, sie dient als Grundlage für die eigentliche Prüfung. Die stilbasierte Architekturprüfung erlaubt, Systeme gegen beliebige Stile zu prüfen. Die Struktur von Stilbeschreibungen basiert auf dem in Kapitel 3 vorgestellten Stilbegriff. Wie in Kapitel 3 erläutert, orientiert sich dieser Stilbegriff an in praxisrelevanten Stilen verwendeten Konzepten. Somit können Entwicklungsteams ihren Stil in den ihnen vertrauten Konzepten beschreiben.

Teilaufgabe 3: Ergebnis der stilbasierten Architekturprüfung sind die Verstöße der Ist-Architektur gegen den gewählten Architekturstil. Zusätzlich berechnet die stilbasierte Architekturprüfung die betroffenen Quelltextstellen. So erleichtert sie den Entwicklungsteams, ihr System zu korrigieren.

Die stilbasierte Architekturprüfung geht in allen drei Teilaufgaben über bisherige Ansätze hinaus. Lediglich einige Aspekte teilt sie mit anderen Treueprüfungen. Dies betrifft vor allem Teilaufgabe 1, in der die stilbasierte Ist-Architektur ermittelt wird. Stilbasierte Ist-Architekturen bestehen teilweise aus denselben Konzepten wie die in anderen Ansätzen verwendeten Ist-Architekturen. Für die Ermittlung dieser gemeinsamen Konzepte verwendet die stilbasierte Architekturprüfung bestehende Verfahren aus dem verbreiteten Ansatz der Software-Reflexionsmodelle (SRM) (Murphy et al. 2001) (siehe Abschnitt 4.3).

Die stilbasierte Architekturprüfung besteht aus einem Kernkonzept und aus drei Ausbaustufen, welche das Kernkonzept erweitern. Das Kernkonzept und die Ausbaustufen unterscheiden sich hinsichtlich der prüfaren Regel- und Beziehungstypen:

- Das Kernkonzept unterstützt die Anteile von Architekturteilen, die sich in den allermeisten Stilen und Stildefinitionen finden: *Elementtypen* und *Beziehungsregeln* für nicht-hierarchische Architekturen.
- Die Ausbaustufen ergänzen Anteile von Architekturteilen, die sich in neueren und umfangreichen Stilen und Stildefinitionen finden: *Beziehungstypen*, *Schnittstellenregeln* und *hierarchische Architekturen*.

Die folgenden Abschnitte erläutern die Teilaufgaben des Kernkonzepts im Detail und zeigen auf, an welchen Stellen der Ansatz vollständig neue Konzepte definiert und wo er bestehende Ansätze erweitert. Die Ausbaustufen werden anschließend in Kapitel 6 vorgestellt.

5.3 Die stilbasierte Ist-Architektur ermitteln

Um ein Softwaresysteme auf Stiltreue zu prüfen, benötigt die stilbasierte Architekturprüfung die Ist-Architektur des Systems. Dieser Abschnitt thematisiert die Berechnung der Ist-Architektur als eine Teilaufgabe der stilbasierten Architekturprüfung (siehe Abbildung 45).

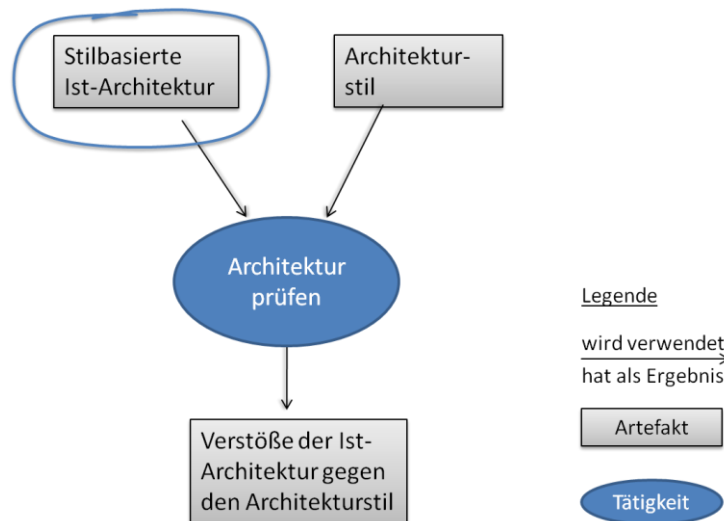


Abbildung 45: Grundprinzip der stilbasierten Architekturprüfung. Der in diesem Abschnitt behandelte Bereich ist hervorgehoben.

Die stilbasierte Architekturprüfung berechnet die Ist-Architektur in zwei Schritten. Abbildung 46 stellt beide Schritte dar, dafür detailliert sie den hervorgehobenen Teil der Abbildung 45. Im ersten Schritt (a) wird eine statische Analyse des Quelltextes durchgeführt, um die Quelltextstruktur zu berechnen. Im zweiten Schritt (b) ermittelt das Prüfwerkzeug die stilbasiert Ist-Architektur, basierend auf dieser Quelltextstruktur sowie auf den manuell ergänzten Zusatzinformationen.

Die hier vorgestellte Teilaufgabe der stilbasierten Architekturprüfung unterscheidet sich von den bisherigen Ansätzen folgendermaßen:

- Der präsentierte Ansatz benötigt andere Zusatzinformationen als die bisherigen Ansätze, da *stilbasierte* Ist-Architekturen mehr Informationen enthalten als andere Ist-Architekturen. Welche Zusatzinformationen im Detail benötigt werden, zeigen die folgenden Abschnitte.
- Im Gegensatz zu anderen Ansätzen integriert die stilbasierte Architekturprüfung die Zusatzinformationen in den Quelltext. So reduziert sich die Gefahr, dass die Zusatzinformationen veralten, da die Notwendigkeit entfällt, bei allen architekturrelevanten Quelltextänderungen auch die getrennte Architekturdokumentation konsistent anzupassen.
- Durch die Integration der Zusatzinformationen in den Quelltext sind alle Informationen der Ist-Architektur im Quelltext vorhanden. So steht während der Programmierung nicht nur die Abstraktionsebene des Quelltextes, sondern auch die Abstraktionsebene der Architektur zur Verfügung. Da alle Architekturinformationen im Quelltext vorhanden sind, ist es zusätzlich möglich, die Architektur in der Entwicklungsumgebung zu visualisieren (beispielsweise mittels eines Plugins).

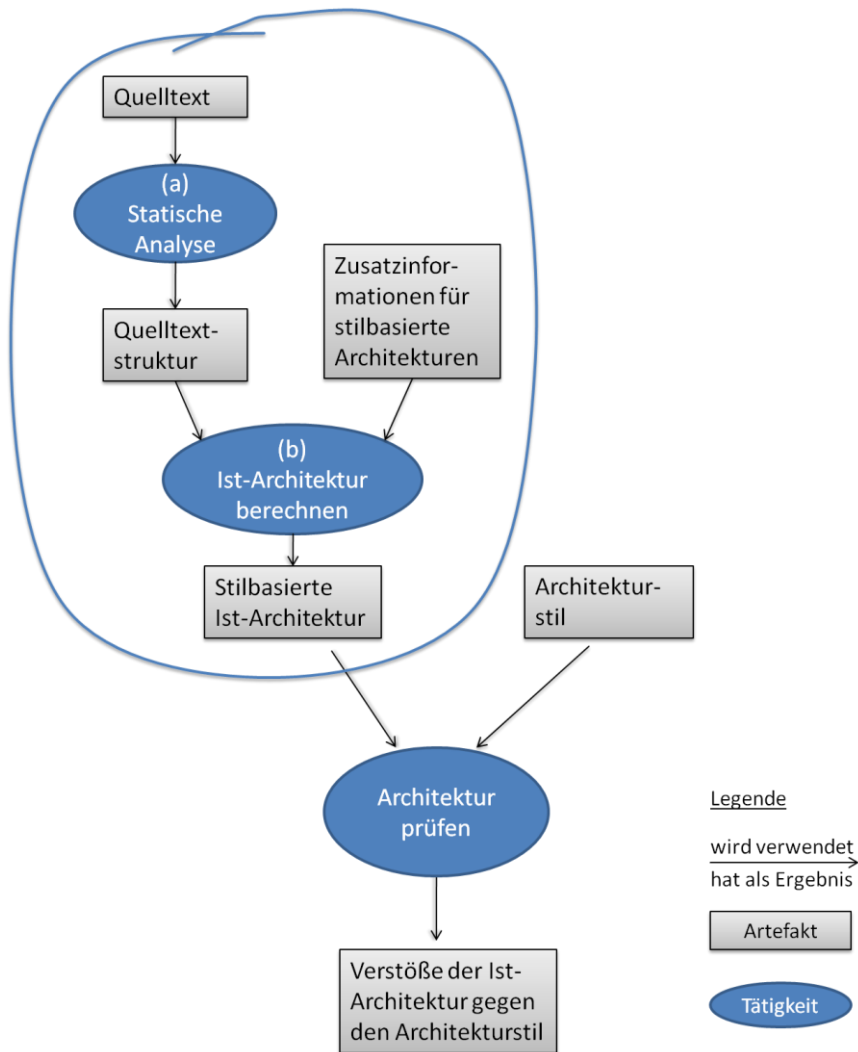


Abbildung 46: Ermittlung der stilbasierten Ist-Architektur

Die folgenden zwei Abschnitte erläutern die beiden in Abbildung 46 dargestellten Berechnungsschritte sowie deren Ergebnisse und zeigen die Unterschiede zu den bestehenden Ansätzen auf.

5.3.1 Berechnungsschritt (a): die Quelltextstruktur ermitteln

Treueprüfungen führen eine statische Quelltextanalyse durch. Ergebnis dieser Analyse ist die Quelltextstruktur. Die im Kernkonzept der stilbasierten Architekturprüfung benötigte Quelltextstruktur besteht aus Quelltextelementen und deren Beziehungen (vgl. Definition 2-2). Die im Kernkonzept verwendete Quelltextstruktur umfasst noch keine Schnittstellen für Quelltextelemente und unterscheidet noch keine Typen für Beziehungen zwischen Quelltextelementen. Schnittstellen und Beziehungstypen werden in verschiedenen Ausbaustufen behandelt.

Quelltextstrukturen beschreiben den Quelltext in Form eines Graphens, der sich aus dem Quelltext berechnet. Abbildung 47 zeigt beispielhaft einen Quelltext in der Programmiersprache Java. Die Abbildung zeigt Quelltextausschnitte aus drei Klassen. Die Klasse `PatientenaufnahmeTool` referenziert die beiden anderen Klassen: `PatientenaufnahmeTool` definiert eine Methode mit

dem statischen Rückgabetypp Patient und ruft den Konstruktor der Klasse PatientenAufnahmeToolGui.

The image shows three screenshots of Java source code editors. The top window shows the code for `PatientenAufnahmeToolGui.java`, which is a public class extending `SwingToolView` with a constructor `public PatientenAufnahmeToolGui()`. The middle window shows `PatientenAufnahmeTool.java`, which extends `OnkoMonoTool` and has a constructor `public Patient patient()` and a method `protected void doEquip()` that instantiates `PatientenAufnahmeToolGui`. The bottom window shows `Patient.java`, which extends `ThingImpl` and implements `Serializable` and `Persistierbar`, with a constructor `public Patient(NameDV nachname, NameDV vorname`.

Abbildung 47: Quelltext in der Programmiersprache Java. Die Klasse PatientenAufnahmeTool referenziert die Klassen Patient und PatientenAufnahmeToolGui

Berechnet man die Quelltextstruktur aus diesem Quelltext, so besteht die berechnete Quelltextstruktur aus drei Quelltextelementen: `PatientenAufnahmeToolGUI`, `PatientenAufnahmeTool` und `Patient`, und aus zwei Beziehungen: eine von dem `PatientenAufnahmeTool` hin zu `Patient`, die andere von `PatientenAufnahmeTool` hin zu `PatientenAufnahmeToolGUI`. Formal dargestellt umfasst die Quelltextstruktur Folgendes²⁸:

- Die Menge der Quelltextelemente:
 $Q = \{\text{PatientenAufnahmeTool}, \text{PatientenAufnahmeToolGui}, \text{Patient}\}$
- Die Relation der Quelltextbeziehungen mit $B_Q \subseteq Q \times Q$:
 $B_Q = \{(\text{PatientenAufnahmeTool}, \text{Patient}),$
 $(\text{PatientenAufnahmeTool}, \text{PatientenAufnahmeToolGui})\}$

²⁸ Die in dieser Arbeit für Mengen und Relationen verwendeten Bezeichner lassen sich auf Seite 217 in der Symbolliste nachschlagen.

Bei der stilbasierten Architekturprüfung ist wählbar, was innerhalb eines Quelltextes als Quelltextelement und als Beziehung verstanden werden soll. Beispielsweise lässt sich festlegen, dass jede Klasse als Quelltextelement interpretiert werden soll, und dass Operationsaufrufe und statische Typreferenzen als Beziehung betrachtet werden. Diese freie Wahl bietet den Vorteil, dass die stilbasierte Architekturprüfung bezüglich der Programmiersprache und der Art der zu prüfenden Architektur flexibel und breiter einsetzbar ist. Bei einem System können beispielsweise Vererbungsbeziehungen als Beziehungen für die Quelltextstruktur relevant sein und bei einem anderen System asynchrone Nachrichten.

Die in dieser Arbeit vorgestellte beispielhafte Umsetzung des Ansatzes mit dem Werkzeug StyleBasedChecker (siehe Abschnitt 7.2) verwendet die Programmiersprache Java. Wie bei objektorientierten Systemen üblich betrachtet das Werkzeug Klassen und weitere Typen (Interfaces, Enumerations, Annotations)²⁹ als Quelltextelemente. Quelltextbeziehungen liegen vor, wenn Klassen einander benutzen oder voneinander erben (siehe Kapitel 2). Dieses Verständnis von Quelltextelementen und Quelltextbeziehungen liegt auch den in dieser Arbeit vorgestellten Quelltextbeispielen zu Grunde.

Generell gilt: Werkzeuge, die die stilbasierte Architekturprüfung unterstützen, haben zwei Möglichkeiten: sie können die Wahl, was unter Quelltextelementen und Beziehungen zu verstehen ist, der prüfenden Person überlassen oder sie können sich diesbezüglich festlegen. Nach dem zweiten Prinzip arbeitet der StyleBasedChecker und so halten es auch sämtliche in Abschnitt 4.4 vorgestellte kommerzielle Werkzeuge. Diese Einschränkung gilt nur für die beispielhafte Umsetzung, nicht jedoch für das Konzept der stilbasierten Architekturprüfung.

Das *Kernkonzept* der stilbasierten Architekturprüfung unterscheidet keine Beziehungstypen, weder auf der Ebene der Quelltextstruktur noch auf der Ebene der Architektur. Das Kernkonzept ist an dieser Stelle bewusst reduziert, um eine übersichtliche Darstellung zu ermöglichen. Für die Unterscheidung von Beziehungstypen enthält die stilbasierte Architekturprüfung eine entsprechende Ausbaustufe (siehe Abschnitt 6.1). In dieser Ausbaustufe können die im Quelltext vorhandenen, unterschiedlichen Beziehungstypen auch innerhalb der Quelltextstruktur und innerhalb der Architektur unterschieden werden.

Abschnitt 5.1 hat dargestellt, dass die stilbasierte Architekturprüfung direkt während der Programmierung einsetzbar sein soll. Um dies zu ermöglichen, ist die Berechnung der Quelltextstruktur explizit Teil des Prüfungsvorgangs; die Berechnung der Quelltextstruktur erfolgt gemeinsam mit der gesamten Prüfung. Somit ist die Quelltextstruktur lediglich ein internes Zwischenergebnis. Die prüfende Person genießt den Vorteil, dass sie sich mit der Quelltextstruktur nicht zu beschäftigen braucht, sie kann direkt in einem Schritt die Prüfung durchführen lassen. In diesem Punkt ist die stilbasierte Architekturprüfung leichtgewichtiger als der ursprüngliche SRM-Ansatz (Murphy et al. 2001). Bei den SRM muss die prüfende Person zwei Schritte ausführen: in einem externen Werkzeug muss sie die Quelltextstruktur berechnen, im zweiten Schritt muss sie die Struktur in das SRM-Werkzeug eingeben, damit dieses die Prüfung auf Architekturtreue durchführen kann. Diese zwei Schritte muss sie jedes Mal wiederholen, sobald sich der Quelltext geändert hat. Diese Zweiteilung, bei der beide Schritte manuelle Eingaben erfordern, verhindert, dass die Architekturkonformanz direkt während der Programmierung automatisiert prüfbar ist. Durch die Zusammenlegung beider Schritte in der stilbasierten Architekturprüfung ist der Prüfungsvorgang automatisierbar und in Entwicklungsumgebungen so integrierbar, dass die Prüfung automatisch bei jeder Quelltextänderung erfolgt.

²⁹ Im Folgenden gilt der Lesbarkeit halber weiterhin: der Begriff „Klasse“ steht stellvertretend für weitere Typen, die in objektorientierten Sprachen vorkommen, wie beispielsweise Java-Interfaces oder Enumerations.

5.3.2 Berechnungsschritt (b): die stilbasierte Ist-Architektur ermitteln

Die stilbasierte Ist-Architektur wird aus der Quelltextstruktur und den manuell ergänzten Zusatzinformationen berechnet (siehe Abbildung 46).

Formal ausgedrückt gilt es, folgende Mengen und Relationen zu ermitteln (siehe Definition 3-3): Die Menge der Architekturelemente (A), die Menge der Beziehungen (B_A) und die Zuordnung der Elemente zu den im Stil definierten Elementtypen (Z_{AT}). Die Schnittstellen von Architekturelementen sind für das hier vorgestellte Kernkonzept nicht relevant.

Die stilbasierte Ist-Architektur geht über die Ist-Architektur der in Kapitel 4 vorgestellten Ansätze hinaus. Einige Ansätze – wie die SRM – definieren keine explizite Ist-Architektur. Andere Ansätze mit expliziter Ist-Architektur definieren diese als Graph aus Architekturelementen und Beziehungen. Die stilbasierte Architekturprüfung ordnet zusätzlich jedem Element einen Elementtyp zu. Über die Typen wird der Zusammenhang zwischen der Architektur und dem zugrundeliegenden Stil hergestellt. Dass Ist-Architekturen bei Konformanzprüfungen Informationen über Architekturelement-Typen enthalten, ist ein neuer Beitrag der vorliegenden Arbeit.

Zusatzinformationen

Zur Erinnerung: die Zusatzinformationen bei der stilbasierten Architekturprüfung beschreiben, welche Architekturelemente in der Ist-Architektur existieren und welche Quelltextelemente gemeinsam ein Architekturelement bilden. Außerdem enthalten die Zusatzinformationen die Zuordnung der Architekturelemente zu den Elementtypen des Stils.

Formal dargestellt umfassen die Zusatzinformationen somit:

- die Menge A der Architekturelemente
- Die Zuordnung Z_{QA} zwischen Quelltextelementen und Architekturelementen. Die Zuordnung ist eine Relation, für die gilt: $Z_{QA} \subseteq Q \times A$. Diese Zuordnung verbindet die Quelltextstruktur mit der Architektur.
- Die Zuordnung Z_{AT} zwischen Architekturelementen und Architekturelement-Typen.

In obigem Beispiel bilden die Klassen *PatientenaufnahmeToolGui* und *PatientenaufnahmeTool* gemeinsam das Architekturelement *Patientenaufnehmer*, die Klasse *Patient* bildet das Architekturelement *PatientArch*. Die Zuordnung der Architekturelemente zu den Elementtypen ist folgendermaßen: der *Patientenaufnehmer* ist ein Werkzeug, das Architekturelement *Patient* ist ein Material. Das Architekturelement namens *PatientArch* besteht in diesem Beispiel aus genau einer Klasse. Die Untersuchungen im Rahmen dieser Arbeit haben gezeigt, dass in einem solchen Fall der Name des Architekturelements oft mit dem Namen der Klasse übereinstimmt. Um Missverständnisse durch Namensmehrdeutigkeiten zu vermeiden, heißt das Architekturelement in dem hier vorgestellten Beispiel jedoch *PatientArch*.

Formal dargestellt umfassen die Zusatzinformationen somit Folgendes:

- $A = \{Patientenaufnehmer, PatientArch\}$
- $(PatientenaufnahmeToolGui, Patientenaufnehmer) \in Z_{QA}$
- $(PatientenaufnahmeTool, Patientenaufnehmer) \in Z_{QA}$
- $(Patient, PatientArch) \in Z_{QA}$

- $(Patientenaufnehmer, Werkzeug) \in Z_{AT}$
- $(PatientArch, Material) \in Z_{AT}$

Abbildung veranschaulicht die Bedeutung der Zusatzinformationen in diesem Beispiel.

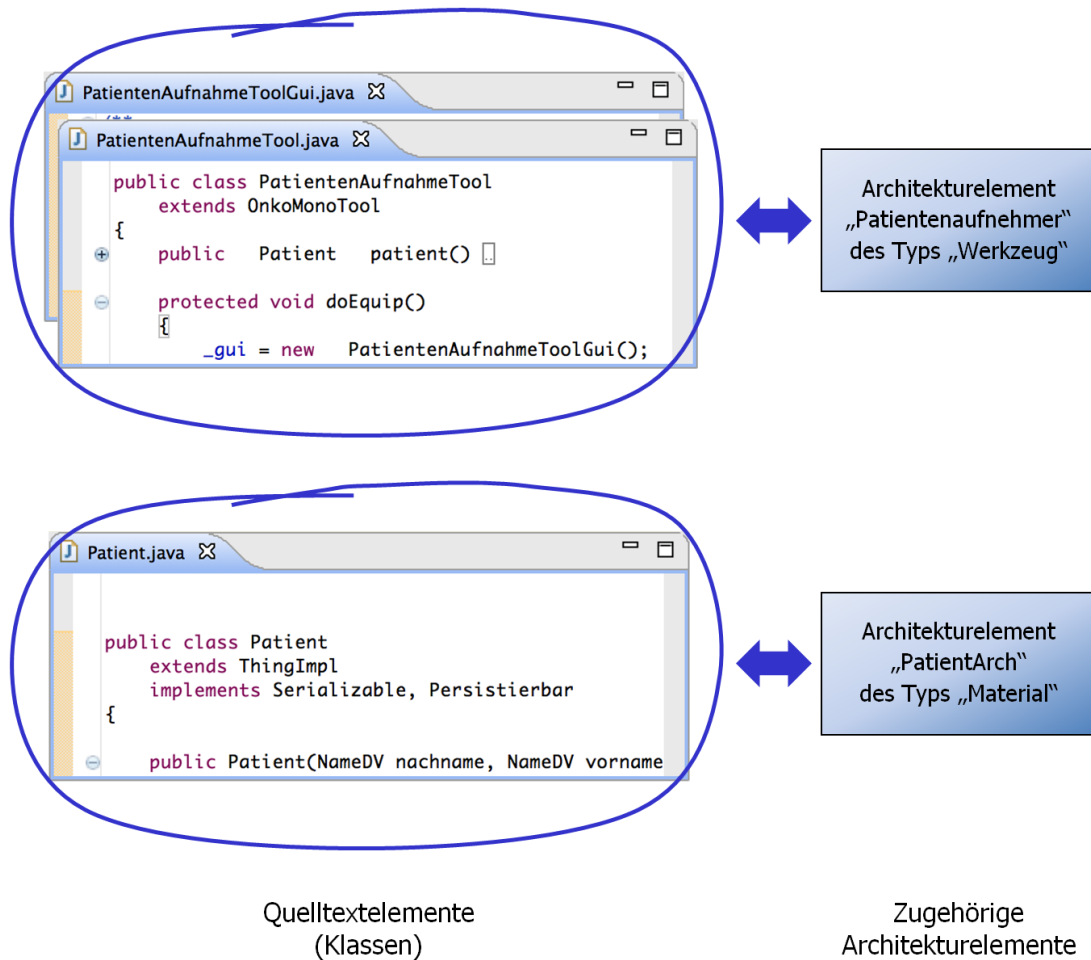


Abbildung 48: Die Zusatzinformationen stellen den Zusammenhang zwischen Quelltext und Architektur her.

Beziehungen

Sind die Zusatzinformationen A und Z_{QA} gegeben, so lassen sich die in der Architektur enthaltenen Beziehungen B_A eindeutig aus der Quelltextstruktur berechnen. Die Berechnung der Beziehungen verläuft wie im SRM-Ansatz (Kapitel 4): Die Beziehungen auf Architekturebene werden durch Aggregation der in der Quelltextstruktur vorhandenen Beziehungen ermittelt:

- $B_A = \{(a, b) | \exists p, q: (p, q) \in B_Q \wedge (p, a) \in Z_{QA} \wedge (q, b) \in Z_{QA}\}$

Das bedeutet in dem gegebenen Beispiel: $B_A = \{(Patientenaufnehmer, PatientArch)\}$, da gilt:

- $(PatientenaufnahmeTool, Patient) \in B_Q \wedge$
 $(PatientenaufnahmeTool, Patientenaufnehmer) \in Z_{QA} \wedge$
 $(Patient, PatientArch) \in Z_{QA}$

Die berechnete stilbasierte Ist-Architektur enthält somit insgesamt folgende Mengen und Relationen:

- $A = \{Patientenaufnehmer, PatientArch\}$
- $B_A = \{(Patientenaufnehmer, PatientArch)\}$
- $Z_{AT} = \{(Patientenaufnehmer, Werkzeug), (PatientArch, Material)\}$

Abbildung 49 veranschaulicht die Quelltextstruktur des Beispiels. Sie enthält zwei Beziehungen. Die oberen beiden Quelltextelemente bilden gemeinsam das Architekturelement Patientenaufnehmer. Die Beziehung zwischen diesen beiden Quelltextelementen spielt auf Architekturebene keine Rolle, da sie sich innerhalb eines Architekturelements befindet. Das untere Quelltextelement, die Klasse Patient, gehört zu einem anderen Architekturelement namens PatientArch. Die in der Abbildung nach unten verlaufende Beziehung von PatientenaufnahmeTool zu Patient ist somit architekturrelevant, da die betroffenen Quelltextelemente zu unterschiedlichen Architekturelementen gehören. Abbildung 50 zeigt die resultierende Ist-Architektur.

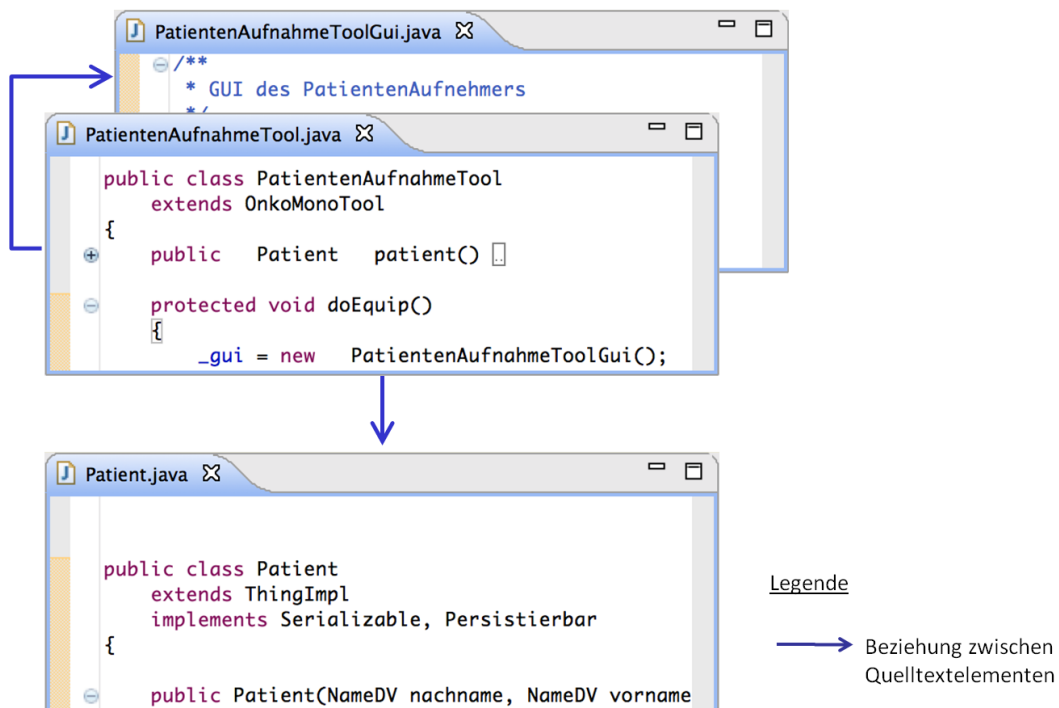


Abbildung 49: Grafische Veranschaulichung der Quelltextstruktur mit drei Quelltextelementen und zwei Beziehungen

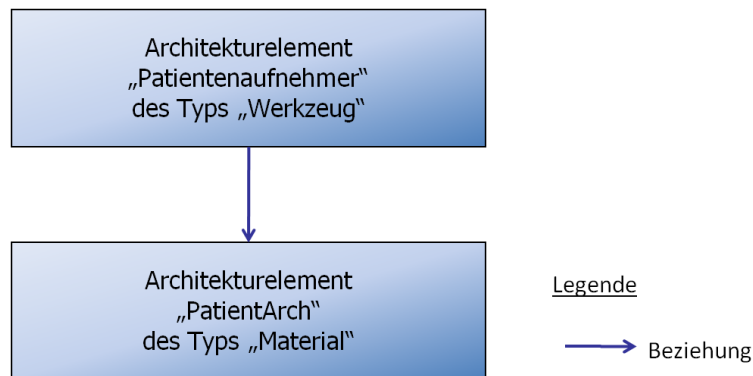


Abbildung 50: Resultierende stilbasierte Ist-Architektur

Prüfwerkzeuge zur stilbasierten Architekturprüfung führen diesen zweiten Berechnungsschritt, in welchem sie die Ist-Architektur aus der Quelltextstruktur und den Zusatzinformationen ermitteln, gemeinsam mit der eigentlichen Treueprüfung durch. Dies entspricht dem üblichen Vorgehen bei Treueprüfungen.

5.3.3 Zusatzinformationen und Quelltext verknüpfen

Es stellt sich die Frage, an welcher Stelle die benötigten Zusatzinformationen über die Softwarearchitektur festgehalten werden. Die bestehenden Ansätze zur Architekturprüfung halten diese Informationen in der Regel im Prüfwerkzeug. Dies hat zwei Nachteile:

- die Zusatzinformationen stehen nur bei der Prüfung zur Verfügung. Die Informationen werden jedoch auch bei der Programmierung benötigt. Die Programmierinnen und Programmierer müssen wissen, wie sich der gerade bearbeitete Quelltextausschnitt in die Architektur einordnet, um die für den Quelltextausschnitt relevanten Architekturvorgaben beachten zu können (siehe Abschnitt 4.1).
- Werden Architekturen geändert oder erweitert, so müssen die Änderungen und Erweiterungen an zwei Stellen festgehalten werden: der Quelltext muss in der Entwicklungsumgebung geändert werden, und die Zusatzinformationen im Prüfwerkzeug. Hier besteht die Gefahr, dass Quelltext und Zusatzinformationen auseinanderdriften.

Die stilbasierte Architekturprüfung adressiert dieses Problem, indem sie sich so dicht wie möglich am Quelltext bewegt, sie verbindet Zusatzinformationen und Quelltext. So sind alle Architekturinformationen während der Programmierung verfügbar und die Gefahr inkonsistenter Architekturinformationen wird reduziert.

Im Folgenden wird diskutiert, inwiefern sich Zusatzinformationen zumindest teilweise aus Quelltexten entnehmen lassen. Basierend auf dieser Diskussion stellt dieser Abschnitt anschließend ein Konzept vor, mit dem sich die Zusatzinformationen mit dem Quelltext verknüpfen lassen. Forschungsergebnisse zu dieser Fragestellung konnte die Autorin bereits veröffentlichen (Becker-Pechau 2009a). Vorarbeiten zum diesem Thema lieferte eine von der Autorin betreute Diplomarbeit über die Inline-Dokumentation von Entwurfsmustern (Özkan und Özkan 2004).

Wenn ein Team sein Softwaresystem stilbasiert entwickelt, dann ist es sich in der Regel über die Zusatzinformationen bewusst. Das bedeutet: Teammitglieder, die ein Quelltextelement bearbeiten, wissen, zu welchem Architekturelement es gehört und sie kennen den Typ des

Architekturelements. Beispielsweise weiß eine Programmiererin, die eine Klasse in einem Client-Server-System bearbeitet, dass diese Klasse zu dem Architekturelement Kundenserver gehört, und dass dieses Architekturelement den Typ Server hat. Ein Programmierer, der nach dem WAM-Stil arbeitet und beispielsweise gerade die Klasse TherapieplanUI hinzufügt, weiß, dass er jetzt ein Werkzeug namens Therapieplaner erstellt. Er weiß somit: das Quelltextelement TherapieplanUI ist dem Architekturelement Therapieplan zugeordnet und dieses Architekturelement hat den Typ Werkzeug.

Zwar programmieren die Teammitglieder ihr System mit diesem Wissen, ihnen fehlen jedoch praktikable Mittel, diese Informationen in den Quelltext zu integrieren: Die Information, zu welchem Architekturelement ein Quelltextelement gehört und welchen Typ dieses Architekturelement hat, lässt sich mit bestehenden programmiersprachlichen Mitteln nicht ausdrücken. Dementsprechend sind diese Informationen dem Quelltext nicht zu entnehmen.

Dennoch bemühen sich viele Entwicklungsteams, diese Informationen innerhalb des Quelltextes festzuhalten. So können sie die Architekturebene besser berücksichtigen, wenn sie ihr System ändern oder erweitern. Die Untersuchungen von Lilienthal zeigen vier üblicherweise gewählte Wege, diese Informationen im Quelltext festzuhalten (Lilienthal 2008):

- **Kommentare:** Über Klassenkommentare wird angegeben, zu welchem Architekturelement die Klasse gehört und welchen Typ das Architekturelement hat.
- **Typvererbung:** Klassen können über Vererbungsbeziehungen deutlich machen, zu welchem Architekturelement-Typ sie gehören. Hierfür lassen sich abstrakte Oberklassen oder sogenannte Marker-Interfaces verwenden.
- **Namensmuster:** Der Typ und Name eines Architekturelements wurde in mehreren untersuchten Systemen durch Namensmuster deutlich gemacht. Ein typisches Muster war „Name des Architekturelements“ + „Elementtyp“. Beispiel: TherapieplanerUI.
- **Packagenamen:** In der Programmiersprache Java lassen sich Klassen in sogenannte Packages gruppieren. In den untersuchten Systemen wurden einige Elementtypen mit Hilfe von Packagenamen kenntlich gemacht. So wurden beispielsweise alle Klassen, die jeweils ein Material implementieren, in ein Package namens materials gebündelt.

In den praktischen Untersuchungen dieser Arbeit ließen sich alle vier Varianten beobachten, oft gemischt innerhalb eines Systems. Jedoch waren die Informationen nicht verlässlich genug für die stilbasierte Architekturprüfung. Dies zeigten auch die Untersuchungen der im Rahmen dieser Arbeit entstandenen Diplomarbeit von Karstens (Karstens 2005). Es ist nicht praktikabel, die Zusatzinformationen allein aus diesen Kenntlichmachungen zu extrahieren und für die Prüfung auf Stiltreue zu nutzen. Die stilbasierte Architekturprüfung braucht eine eigene, verbindlichere und strukturiertere Möglichkeit, die Zusatzinformationen festzuhalten, die über reine Namenskonventionen hinaus geht.

Zwar können die bisher in der Praxis genutzten Kenntlichmachungen nicht zuverlässig für die stilbasierte Architekturprüfung genutzt werden, dennoch helfen die Erkenntnisse aus der Praxis bei der Konzeption der stilbasierten Architekturprüfung. Denn die in der Praxis verwendeten Kenntlichmachungen erlauben zwei Schlussfolgerungen für die stilbasierte Architekturprüfung:

- Auch ohne eine Prüfung auf Stiltreue werden die Zusatzinformationen in der Praxis durchaus gekennzeichnet, die bestehenden Mittel sind jedoch ungenügend.
- Anhand der in der Praxis angewendeten Strategien lässt sich sehen, welche Informationen an welcher Stelle kodiert werden.

Unabhängig davon, welche der vier obigen Arten der Kenntlichmachung gewählt wurden, war die kodierte Information doch immer dieselbe: Quelltextelemente wurden ergänzt um die Information, wie das zugehörige Architekturelement heißt und zu welchem Architekturelement-Typ es gehört.

Das Konzept der stilbasierten Architekturprüfung orientiert sich an diesem Vorgehen. Formal bedeutet dies, dass an jedem Quelltextelement q , das einem Architekturelement zugeordnet ist, folgende Informationen ergänzt werden:

- Das Architekturelement a . Dabei gilt: $(q, a) \in Z_{QA}$
- Der Typ t des Architekturelements a . Dabei gilt: $(a, t) \in Z_{AT}$

Hierbei gilt es zu beachten:

- Im Falle, dass ein Quelltextelement zu genau einem Architekturelement gehört, wird in der Praxis der Name des Architekturelements oft nicht zusätzlich genannt, stattdessen geben die Teams dem Quelltextelement denselben Namen wie dem Architekturelement. Ein Beispiel hierfür ist die Klasse `Patient` im Package `materials`. Das zugehörige Architekturelement heißt ebenfalls `Patient` und ist vom Typ `Material`. In diesem Fall besteht die Zusatzinformation somit nur aus dem Typ, der Architekturelement-Name entfällt.

Das in diesem Kapitel präsentierte Konzept der stilbasierten Architekturprüfung ist flexibel, um sich für verschiedene Sprachen und Werkzeuge umsetzen zu lassen. Aus diesem Grunde legt es zwar formal fest, welche Informationen an den Quelltextelementen festhalten werden, erlaubt jedoch den implementierten Werkzeugen, eine eigene Notation zu wählen.

Es lassen sich verschiedene Strategien unterscheiden, mit denen die Informationen festhalten werden können:

- Die Informationen können direkt mit Mitteln der Programmiersprache festgehalten werden. Diese Strategie verfolgt die in Kapitel 7 vorgestellte beispielhafte Umsetzung mit dem Werkzeug `StyleBasedChecker`. Hier werden die Informationen am Beispiel der Programmiersprache Java mit Java-Annotationen festgehalten. Dies ist eine Möglichkeit, das Konzept der stilbasierten Architekturprüfung umzusetzen. Vorteil dieser Umsetzung ist, dass die Informationen direkt im Quelltext sind, und somit auch dann verfügbar sind, wenn das Werkzeug `StyleBasedChecker` nicht zur Verfügung steht.
- Eine andere Möglichkeit der Werkzeugunterstützung wäre eine Umsetzung, bei der das Werkzeug die Informationen nicht direkt in den betroffenen Quelltextelementen des Softwaresystems notiert, sondern die Verbindung der Zusatzinformationen zum Quelltext beispielsweise in einer zusätzlichen Datei oder Datenbank festhält. Hier sieht die stilbasierte Architekturprüfung vor, dass so ein Werkzeug den Programmierern und Programmierern erlauben muss, die Informationen direkt bei den Quelltextelementen anzugeben und anzusehen, beispielsweise mit einem speziellen Editor. In diesem Fall braucht der Quelltext selber nicht geändert zu werden. Nachteil dieser Umsetzung ist, dass die Informationen nicht direkt bei den Quelltextelementen festgehalten sind, und somit die Gefahr größer ist, dass die Informationen und der Quelltext auseinander driften, da Abweichungen nicht unbedingt gleich sichtbar werden, beispielsweise, wenn ein eventuell vorhandener spezieller Editor gerade nicht zur Verfügung steht.
- Eine weitere Möglichkeit, die in der Literatur in verschiedenen Kontexten genannt wird: Programmiersprachen könnten erweitert werden, so dass mehr Architekturkonstrukte in ihnen ausgedrückt werden können (Broy und Reussner 2010; Knodel und Popescu

2007; van Eyck et al. 2011; Clements und Shaw 2009). Knodel und Popescu weisen darauf hin, dass es mit solchen Programmiersprachen nicht mehr nötig sei, die Architekturinformationen mit Sprachmitteln auszudrücken, die dafür weder gedacht noch optimal geeignet sind (wie die oben genannten vier Möglichkeiten).

Welche dieser Möglichkeiten gewählt wird, hängt vom Design des jeweiligen Werkzeuges zur stilbasierten Architekturprüfung ab. Die stilbasierte Architekturprüfung ist diesbezüglich flexibel. Sie enthält ein Konzept, Quelltexte mit den für stilbasierte Architekturen benötigten Zusatzinformationen zu annotieren. Die weiter unten vorgestellte, beispielhafte Umsetzung mit dem StyleBasedChecker nutzt dieses Konzept.

Alternativ erlaubt die stilbasierte Architekturprüfung, die Zusatzinformationen auf andere Weise festzuhalten, beispielsweise im Prüfwerkzeug, in der Entwicklungsumgebung oder innerhalb einer noch zu entwickelnden, auf stilbasierte Architekturen ausgerichteten Programmiersprache.

Wichtig ist, dass die Zusatzinformationen während der Programmierung direkt bei den Quelltextelementen sichtbar sind, so dass man bei der Programmierung wahrnimmt, zu welchem Architekturelement und Elementtyp ein Quelltextelement gehört.

5.4 Den Architekturstil beschreiben

Die stilbasierte Architekturprüfung benötigt eine formale Beschreibung des Stils als Eingangsinformation für die Prüfung auf Stiltreue (siehe Abbildung 51). Dieser Abschnitt thematisiert, wie das Entwicklungsteam seinen Architekturstil beschreibt.

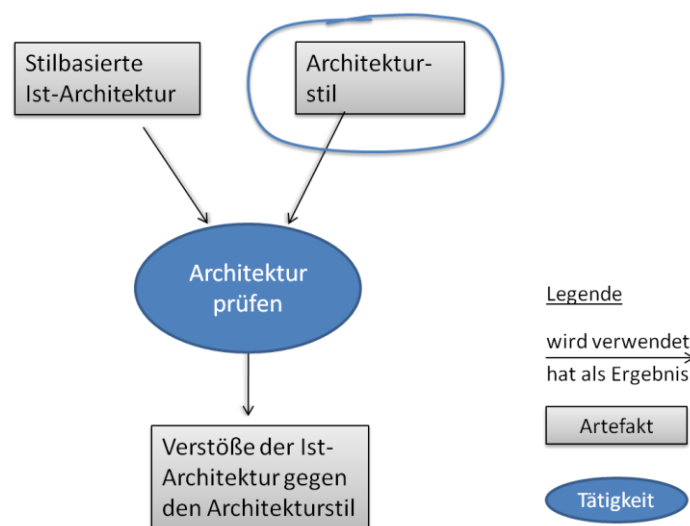


Abbildung 51: Grundprinzip der stilbasierten Architekturprüfung. Der in diesem Abschnitt behandelte Bereich ist hervorgehoben.

Welche Informationen benötigt die stilbasierte Architekturprüfung über den gewählten Stil? Zur Erinnerung: Architekturstile umfassen *Architekturelement-Typen* und *Regeln* (siehe Definition 3-1). Für das hier vorgestellte Kernkonzept der stilbasierten Architekturprüfung sind

Gebots- und Verbots-Beziehungsregeln relevant (kurz: Gebots- und Verbotsregeln). Jede Gebots- und Verbotsregel bezieht sich auf zwei Elementtypen.

Bei Gebotsregeln gilt: Jedes Element des ersten Typs, *muss* mindestens eine Beziehung zu einem Element des zweiten Typs haben. „Werkzeuge müssen Materialien kennen“ ist ein Beispiel für eine Gebotsregel.

Bei Verbotsregeln gilt: Elemente des ersten Typs *dürfen keine* Beziehung zu Elementen des zweiten Typs haben. Eine Verbotsregel ist beispielsweise „Materialien dürfen keine Services kennen“.

Wenn ein Entwicklungsteam seinen Architekturstil mit Verbots- und Gebotsregeln beschreibt, so muss es formal, nach Definition 3-2, folgende Mengen festlegen:

- Die Menge der Architekturelement-Typen T_A ,
- die Menge der Gebotsregeln R_G ,
- und die Menge der Verbotsregeln R_V .

Die Menge der Architekturelement-Typen T_A enthält die Bezeichnungen der Typen. Im WAM-Stil beispielsweise gilt:

$$T_A = \{Werkzeug, Service, Automat, Material, Fachwert\}$$

Bei den Gebotsregeln R_G und den Verbotsregeln R_V handelt es sich formal um binäre Relationen auf der Menge der Architekturelement-Typen. Es gilt: $R_G \subseteq T_A \times T_A$ und $R_V \subseteq T_A \times T_A$. Die Formalisierung in Relationen ist folgendermaßen zu verstehen: Für jedes Tupel $(t1, t2) \in R_G$ gilt: Architekturelemente des Typs t1 müssen eine Beziehung zu mindestens einem Architekturelement des Typs t2 haben. Die Gebotsregel „Werkzeuge müssen Materialien kennen“ beispielsweise wird formal beschrieben durch $(Werkzeug, Material) \in R_G$. Analog gilt bei den Verbotsregeln für jedes Tupel $(t3, t4) \in R_V$, dass Architekturelemente des Typs t3 nicht auf Architekturelemente des Typs t4 zugreifen dürfen. Die Verbotsregel „Materialien dürfen keine Services kennen“ beispielsweise wird formal beschrieben durch $(Material, Service) \in R_V$.

Abbildung 52 stellt die Struktur von Stilbeschreibungen in Form eines UML-Diagramms dar.

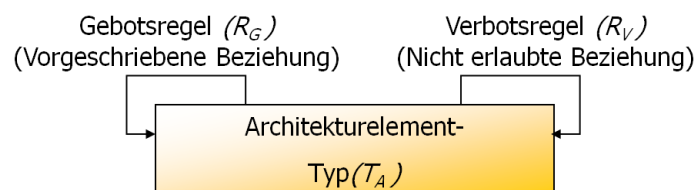


Abbildung 52: UML-Darstellung der Struktur von Architekturstil-Beschreibungen

Abbildung 53 visualisiert die oben genannten Beispiele aus dem WAM-Stil, bestehend aus drei Architekturelement-Typen und zwei Regeln. Die formale Stilbeschreibung zu diesem Beispiel umfasst folgende Mengen und Relationen:

- Die Menge der Elementtypen $T_A = \{Werkzeug, Material, Service\}$
- Die Menge der Gebotsregeln $R_G = \{(Werkzeug, Material)\}$
- Die Menge der Verbotregeln $R_V = \{(Material, Service)\}$

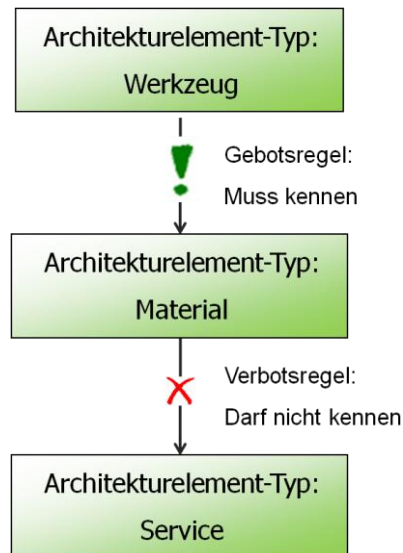


Abbildung 53: Beispielhafter Ausschnitt aus dem WAM-Stil, drei Architekturelement-Typen und zwei Regeln

Für die Beschreibung eines Architekturstils gilt folgende Konsistenzbedingung, die sich unmittelbar aus der Bedeutung der Regeln ergibt: für zwei Architekturelement-Typen kann nur maximal eine der beiden Regeln gelten. Im WAM-Stil beispielsweise gilt für die beiden Architekturelement-Typen Material und Service die Verbotregel „Materialien dürfen keine Services kennen“. Es ist selbstverständlich nicht sinnvoll, gleichzeitig die folgende Gebotsregel anzugeben: „Materialien müssen Services kennen“. Formal ausgedrückt stellt sich die Konsistenzbedingung wie folgt dar: $R_V \cap R_G = \emptyset$: Das heißt, dass ein Tupel aus zwei Architekturelement-Typen $(t_5, t_6) \subseteq T_A \times T_A$ kann nur entweder ein Element der Menge R_G oder ein Element der Menge R_V sein kann, nicht jedoch beides.

Möchte ein Entwicklungsteam die Stiltreue seines Systems mit der stilbasierten Architekturprüfung ermitteln, so muss es den gewählten Architekturstil vor der ersten Prüfung beschreiben. Für erneute Prüfungen kann das Team seine Beschreibung wiederverwenden, auch wenn die Architektur des System evolviert, und beispielsweise neue Architekturelemente hinzukommen. Für neue Systeme desselben Stils lässt sich die Beschreibung ebenfalls unverändert übernehmen. So lassen sich selbst regelmäßige und häufige Prüfungen in der Praxis schnell, einfach und wirtschaftlich durchführen.

Damit die Stilbeschreibung unverändert wiederverwendet werden kann, trennt die stilbasierte Architekturprüfung explizit die Beschreibung des Stils, der die Architekturvorgaben festlegt, von den im vorherigen Abschnitt vorgestellten Zusatzinformationen, die den Zusammenhang zwischen Quelltext und Architektur beschreiben. Im Gegensatz zur Stilbeschreibung müssen die Zusatzinformationen gegebenenfalls angepasst werden, wenn sich die Architektur ändert, beispielsweise wenn neue Architekturelemente hinzukommen.

Das Konzept der stilbasierten Architekturprüfung legt fest, *welche* Informationen Entwicklungsteams angeben müssen, um ihren Architekturstil zu beschreiben. Prüfwerkzeuge, die nach dem Ansatz der stilbasierten Architekturprüfung arbeiten, müssen es erlauben, dass die prüfende Person dem Werkzeug diese Informationen zur Verfügung stellt. Die stilbasierte Architekturprüfung schreibt keine feste *Syntax* für die Stilbeschreibung vor. Die Werkzeuge sind somit frei in der Wahl des Beschreibungsformats für Architekturstile. Das in Kapitel 7 vorgestellte, prototypische Werkzeug, der StyleBasedChecker, verwendet eine XML-Notation.

5.5 Die Verstöße berechnen

Liegen die stilbasierte Ist-Architektur und der Architekturstil vor, so werden aus diesen Informationen die Verstöße der Ist-Architektur gegen den Architekturstil berechnet. Abbildung 54 zeigt, wie sich die Berechnung der Verstöße in das Gesamtkonzept der stilbasierten Architekturprüfung einordnet.

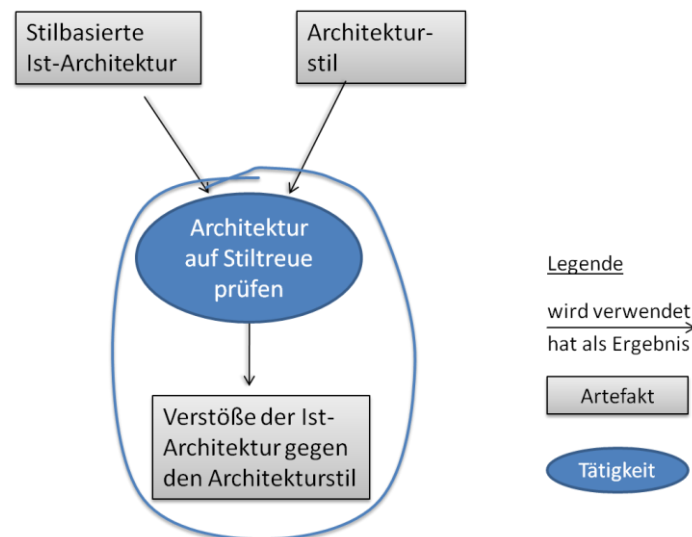


Abbildung 54: Die Berechnung der Verstöße der Ist-Architektur im Gesamtkontext der stilbasierten Architekturprüfung

Die stilbasierte Architekturprüfung berechnet, welche Quelltextelemente die Verstöße verursachen. So wissen die Entwicklungsteams, wo sie den Quelltext ändern müssen, um die gefundenen Verstöße zu korrigieren. Abbildung 55 zeigt beide Schritte: die Berechnung der Verstöße und die Verfolgung der Verstöße hin zu den betroffenen Quelltextstellen.

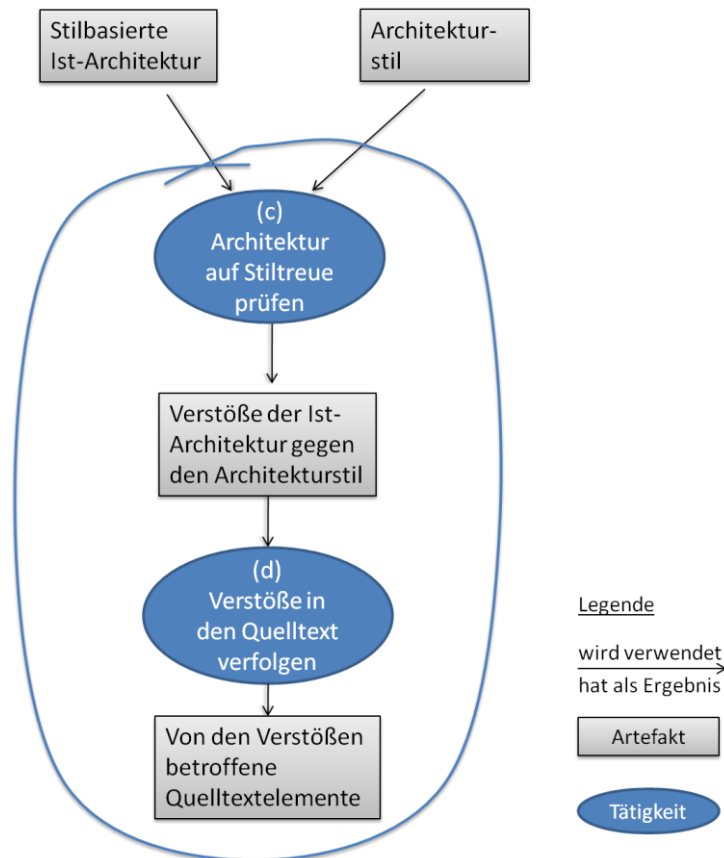


Abbildung 55: Die Berechnung der Verstöße (c) und der betroffenen Quelltextelemente (d) im Gesamtkontext der stilbasierten Architekturprüfung

Wie im vorherigen Abschnitt dargestellt, adressiert das Kernkonzept der stilbasierten Architekturprüfung zwei Regeltypen: Gebots- und Verbots-Beziehungsregeln. Die Berechnung der Verstöße ist abhängig vom Regeltyp. Die im Folgenden vorgestellten Berechnungsvorschriften sind ein neuer Beitrag der vorliegenden Arbeit. Die bisherigen Ansätze zur Treueprüfung (siehe Kapitel 4) ermitteln zwar, ob die Beziehungen innerhalb der Ist-Architektur den Vorgaben entsprechen, allerdings nur für Vorgaben in Form von Soll-Architekturen. Soll-Architekturen zählen die vorgesehenen Beziehungen einzeln auf, sie definieren keine Regeln.

Abbildung 56 zeigt eine beispielhafte Ist-Architektur. Formal dargestellt umfasst diese Ist-Architektur folgende Mengen und Relationen:

- Architekturelemente:
 $A = \{Patientenaufnehmer, Diagnosesteller, Therapieplaner, Patientenmappe, Therapieplan, PatientenService\}$
- Zuordnung der Architekturelemente und zu den Elementtypen: $Z_{AT} = \{(Patientenaufnehmer, Werkzeug), (Diagnosesteller, Werkzeug), (Therapieplaner, Werkzeug), (Patientenmappe, Material), (Therapieplan, Material), (PatientenService, Service)\}$

- Beziehungen: $B_A = \{(Patientenaufnehmer, Patientenmappe), (Diagnosesteller, Patientenmappe), (Diagnosesteller, Therapieplan), (Diagnosesteller, PatientenService), (Therapieplaner, PatientenService), (Therapieplan, PatientenService), (PatientenService, Patientenmappe)\}$

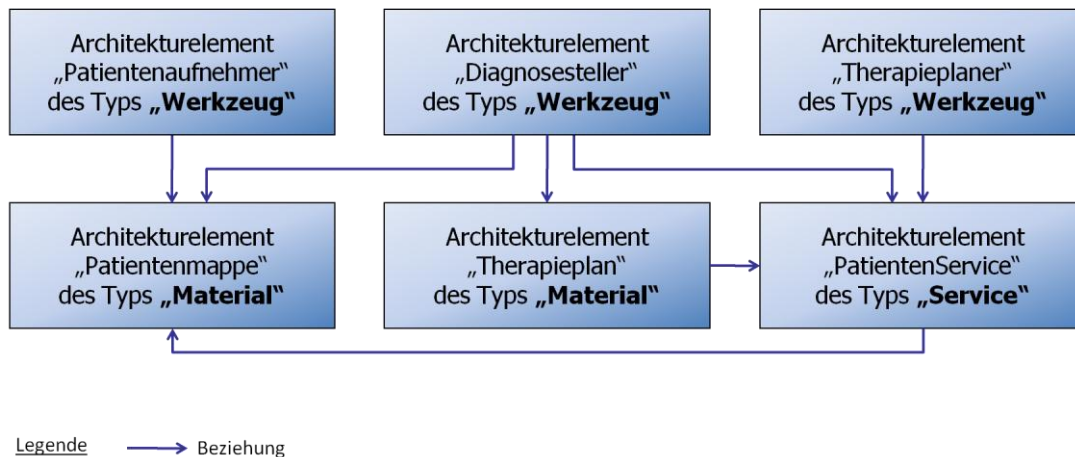


Abbildung 56: Beispiel: eine zu prüfende Ist-Architektur

Die Ist-Architektur basiert auf dem beispielhaften Architekturstil, der im vorherigen Abschnitt eingeführt wurde (es handelt sich um einen Ausschnitt des WAM-Stils). Der Stil enthält drei Architekturelement-Typen und zwei Regeln. Die Architekturelement-Typen sind: Werkzeug, Material und Service. Eine Regel ist eine Verbotsregel, sie besagt „Materialien dürfen keine Services kennen“. Die andere Regel ist eine Gebotsregel, sie lautet „Werkzeuge müssen Materialien kennen“. Formal dargestellt umfasst dieser Stil somit folgende Mengen und Relationen.

- Architekturelement-Typen: $T_A = \{Werkzeug, Material, Service\}$
- Verbotsregeln: $R_V = \{(Material, Service)\}$
- Gebotsregeln: $R_G = \{(Werkzeug, Material)\}$

Im Folgenden dient dieses Beispiel zur Illustration der vorgestellten Berechnungsverfahren.

5.5.1 Berechnungsschritt (c): Die Ist-Architektur auf Stiltreue prüfen

In diesem Berechnungsschritt ermittelt die stilbasierte Architekturprüfung die Verstöße gegen die Verbots- und gegen die Gebotsregeln.

Verstöße gegen Verbotsregeln berechnen:

Ziel dieser Berechnung ist es, alle Beziehungen innerhalb der zu prüfenden Ist-Architektur zu ermitteln, die gegen die Verbotsregeln des Stils verstoßen. Die Ist-Architektur in Abbildung 56

verstößt an genau einer Stelle gegen die im Beispiel gegebene Verbotregel, dass Materialien nicht auf Services zugreifen dürfen: das Material Therapieplan greift auf den Service PatientenService zu. Das Ergebnis der Prüfung soll somit lauten: Die Beziehung von dem Therapieplan zu dem PatientenService ist ein Verstoß gegen den Architekturstil.

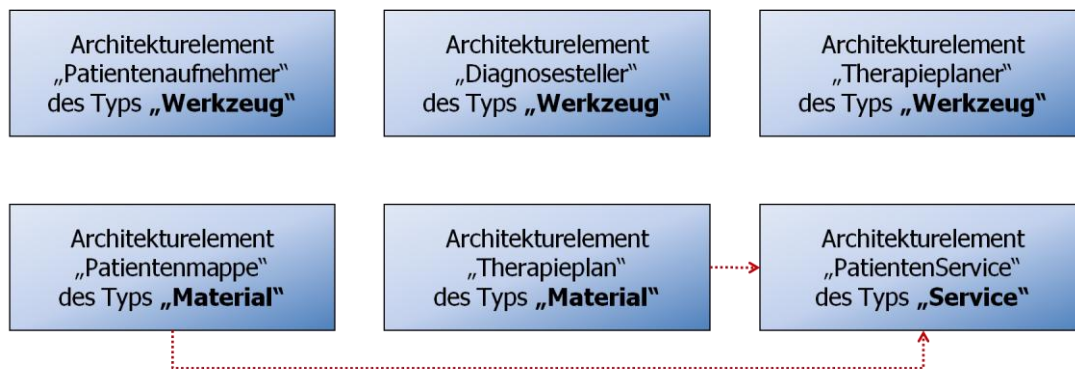
Wenn die stilbasierte Architekturprüfung die Verstöße gegen Verbotregeln berechnet, so ist das Ergebnis dieser Berechnung formal betrachtet eine Menge von Beziehungen. Die Beziehungen werden als Tupel mit je zwei Architekturelementen dargestellt:

- Die Menge der Verstöße gegen Verbotregeln: $V_{VR} \subseteq A \times A$.

V_{VR} enthält alle Beziehungen der Ist-Architektur, die gegen Verbotregeln verstoßen. Somit handelt es sich bei V_{VR} um eine Teilmenge aller Beziehungen innerhalb der Ist-Architektur. Es gilt: $V_{VR} \subseteq B_A$.

Um die Menge der Verstöße zu berechnen, ermittelt die stilbasierte Architekturprüfung im ersten Schritt eine Hilfsrelation: die Menge H_{BV} aller möglichen Architekturbeziehungen, die nicht erlaubt wären (kurz: die verbotenen Beziehungen). Für diese Hilfsrelation spielen die tatsächlichen Beziehungen der Ist-Architektur keine Rolle. Da es in obigem Beispiel zwei Materialien (Patientenmappe und Therapieplan) und einen Service (PatientenService) gibt, enthält die Hilfsrelation zwei Beziehungen:

- Die Beziehung von Patientenmappe zu PatientenService
- und die Beziehung von Therapieplan zu PatientenService.
- Abbildung 57 visualisiert die Hilfsrelation H_{BV} mit den zwei möglichen Beziehungen, die laut Verbotregeln nicht erlaubt sind.



Legende > Alle möglichen, nicht erlaubten Beziehungen

**Abbildung 57: Zwischenergebnis der Berechnung, die Hilfsrelation H_{BV} :
Alle potentiellen Beziehungen, die laut Verbotregel nicht erlaubt sind.**

Die Hilfsrelation H_{BV} der verbotenen Beziehungen wird anhand folgender Mengen und Relationen berechnet:

- die Menge A der Architekturelemente,
- die Zuordnung Z_{AT} der Architekturelemente zu den Elementtypen
- die Menge R_V der Verbotsregeln

Die Hilfsrelation der verbotenen Beziehungen enthält Tupel von Architekturelementen. Es gilt: $H_{BV} \subseteq A \times A$. Sie berechnet sich folgendermaßen:

- Hilfsrelation der verbotenen Beziehungen:

$$H_{BV} = \{(a, b) \mid \exists t1, t2 : (a, t1) \in Z_{AT} \wedge (b, t2) \in Z_{AT} \wedge (t1, t2) \in R_V\}$$

In dieser Berechnungsvorschrift bezeichnen a und b zwei Architekturelemente. $t1$ und $t2$ bezeichnen die zugehörigen Architekturelement-Typen, $t1$ ist der Typ von a , $t2$ der Typ von b . Unter diesen Voraussetzungen gilt: das Tupel (a, b) von Architekturelementen ist genau dann in der Menge H_{BV} enthalten, wenn das Tupel der beiden Typen $(t1, t2)$ in der Menge R_V enthalten ist. Wendet man diese Berechnungsvorschrift für das betrachtete Beispiel an, so enthält die Relation H_{BV} zwei Tupel:

$$H_{BV} = \{(PatientenMappe, PatientenService), (Therapieplan, PatientenService)\}$$

Am Beispiel des ersten Tupels lässt sich die Berechnung folgendermaßen erläutern: In diesem Fall ist $a = Patientenmappe$ und $b = PatientenService$. Es lässt sich folgern $t1 = Material$, $t2 = Service$, da gilt: $(Patientenmappe, Material) \in Z_{AT}$ und $(PatientenService, Service) \in Z_{AT}$. Beziehungen von Materialien zu Services sind nicht erlaubt, da $(Material, Service) \in R_V$. Somit ist keine Beziehung von Patientenmappe zu Patientenservice erlaubt. Es folgt: $(Patientenmappe, PatientenService) \in H_{BV}$.

Die Menge aller Verstöße gegen Verbotsregeln ergibt sich, wenn man die verbotenen Beziehungen H_{BV} mit der Ist-Architektur vergleicht. In der Ist-Architektur darf keine der Beziehungen aus H_{BV} vorkommen. In diesem Fall haben H_{BV} (Abbildung 57) und die Ist-Architektur (Abbildung 56) eine gemeinsame Beziehung: Die Beziehung von dem Material Therapieplan hin zu dem Service PatientenService. Diese Beziehung ist somit ein Verstoß gegen den Architekturstil.

Formal dargestellt berechnet sich die Menge der Verstöße gegen die Verbotsregeln als Schnittmenge der Beziehungen der Ist-Architektur B_A und der Hilfsrelation aller verbotenen Beziehungen H_{BV} :

- Die Menge der Verstöße gegen eine Verbotsregel: $V_{VR} = B_A \cap H_{BV}$

Für obiges Beispiel gilt:

- $B_A = \{(Patientenaufnehmer, Patientenmappe), (Diagnosesteller, Patientenmappe), (Diagnosesteller, Therapieplan), (Diagnosesteller, PatientenService), (Therapieplaner, PatientenService), (Therapieplan, PatientenService), (PatientenService, Patientenmappe)\}$

- $H_{BV} = \{(Patientenmappe, PatientenService), (Therapieplan, PatientenService)\}$
- Somit ergibt sich ein Verstoß:
 $V_{VR} = B_A \cap H_{BV} = \{(Therapieplan, PatientenService)\}$

Abbildung 58 visualisiert den Verstoß der Beispiel-Architektur gegen die gegebenen Verbotsregeln.

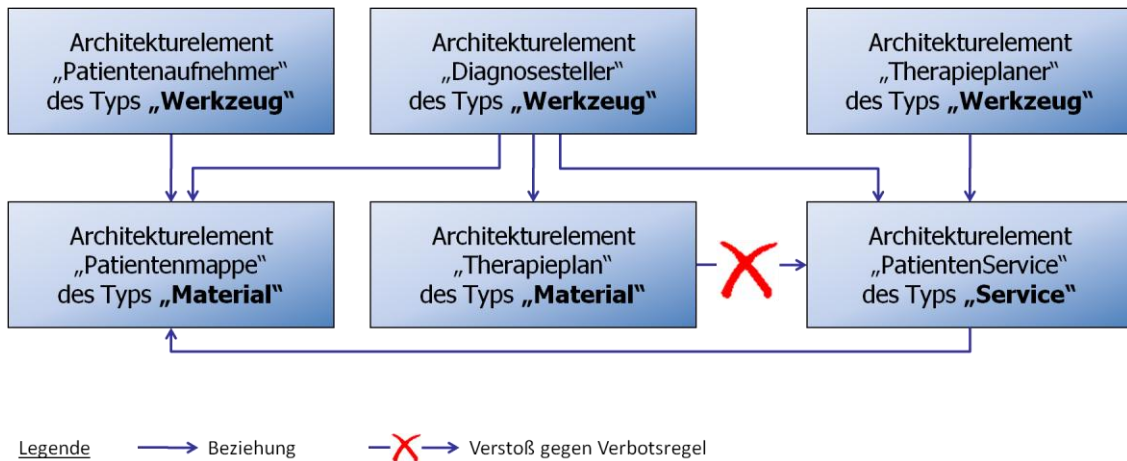


Abbildung 58: Eine Beziehung verstößt gegen die gegebenen Verbotsregeln

Verstöße gegen Gebotsregeln berechnen:

Gebotsregeln schreiben vor, dass bestimmte Beziehungen innerhalb einer Architektur vorkommen müssen. Sie haben die Form: Elemente des Typs t1 müssen Elemente des Typs t2 kennen. Die im Beispiel gegebene Gebotsregel besagt: „Werkzeuge müssen Materialien kennen“. Das bedeutet, dass jedes Architekturelement des Typs Werkzeug mindestens eine Beziehung zu einem Material haben soll. Die Ist-Architektur in Abbildung 56 verstößt an genau einer Stelle gegen diese Regel: das Werkzeug Therapieplaner kennt kein Material. Das Ergebnis der Prüfung soll lauten, dass der Therapieplaner gegen den Architekturstil verstößt, da er kein Material kennt. Verstöße gegen Gebotsregeln bestehen aus einem Paar: einem Architekturelement A und einem Elementtyp t2, wobei gilt: dem Architekturelement A fehlt eine Beziehung zu einem Element des Typs t2.

Wenn die stilbasierte Architekturprüfung die Verstöße gegen Gebotsregeln berechnet, dann ist das Ergebnis formal betrachtet eine Menge von Tupeln, die jeweils aus einem Architekturelement und einem Elementtyp bestehen:

- Die Menge der Verstöße gegen Gebotsregeln: $V_{GR} \subseteq A \times T_A$

Für die Gebotsregeln werden zwei Hilfsrelationen benötigt. Die erste Hilfsrelation HS_{AT} enthält für jedes Architekturelement die Typen der Architekturelemente, zu denen Beziehungen bestehen *sollen*. Die Relation wird im Folgenden kurz als Soll-Typ-Beziehungen bezeichnet. Sie berechnet sich aus der Ist-Architektur und den Gebotsregeln. Die tatsächlichen Beziehungen in der Ist-Architektur spielen für diese Hilfsrelation keine Rolle.

Die beispielhaft betrachtete Gebotsregel „Werkzeuge müssen Materialien kennen“ betrifft alle Architekturelemente des Typs Werkzeug. Dies sind der Patientenaufnehmer, der Diagnosesteller und der Therapieplaner. Für diese drei Architekturelemente gilt, dass sie jeweils mindestens ein Material kennen müssen. Damit enthält die Hilfsrelation der Soll-Typ-Beziehungen HS_{AT} folgende Informationen:

- Der Patientenaufnehmer soll mindestens ein Material kennen
- Der Diagnosesteller soll mindestens ein Material kennen
- Der Therapieplaner soll mindestens ein Material kennen

Formal betrachtet enthält HS_{AT} Tupel von je einem Architekturelement und einem Elementtyp: $HS_{AT} \subseteq A \times T_A$. Für das Beispiel gilt:

- $HS_{AT} = \{(Patientenaufnehmer, Material), (Diagnosesteller, Material), (Therapieplaner, Material)\}$

Die Relation berechnet sich folgendermaßen:

- Die Hilfsrelation der Soll-Typ-Beziehungen zwischen Architekturelementen und Typen: $HS_{AT} = \{(a, t2) | \exists t1: (a, t1) \in Z_{AT} \wedge (t1, t2) \in R_G\}$

Hierbei bezeichnet a ein Architekturelement und $t1$ und $t2$ jeweils einen Typ. Ein Tupel $(a, t2)$ ist genau dann in der Relation HS_{AT} , wenn folgende zwei Bedingungen gelten: Das Architekturelement a hat den Typ $t1$ (formal ausgedrückt: $(a, t1) \in Z_{AT}$) und es existiert eine Gebotsregel, die besagt, dass Elemente des Typs $t1$ mindestens ein Element des Typs $t2$ kennen müssen (formal ausgedrückt: $(t1, t2) \in R_G$).

In obigem Beispiel enthält die Hilfsrelation HS_{AT} unter anderem das Tupel (Patientenaufnehmer, Material). Anhand dieses Tupels lässt sich die Berechnungsvorschrift erklären. In diesem Fall gilt: $a = \text{Patientenaufnehmer}$ und $t2 = \text{Material}$. Der Typ des Patientenaufnehmers ist Werkzeug, somit gilt $t1 = \text{Werkzeug}$. Die Regel „Werkzeuge müssen Materialien kennen“ wird formal ausgedrückt durch $(Werkzeug, Material) \in R_G$. Somit ist folgender Ausdruck wahr: $(Patientenaufnehmer, Werkzeug) \in Z_{AT} \wedge (Werkzeug, Material) \in R_G$. Daraus folgt: $(Patientenaufnehmer, Material) \in HS_{AT}$.

Die zweite Hilfsrelation HI_{AT} basiert auf der Ist-Architektur. Sie listet für jedes Architekturelement auf, zu welchen Typen Beziehungen bestehen (kurz: die Ist-Typ-Beziehungen). Besteht eine Beziehung eines Architekturelements $a1$ zu mindestens einem anderen Architekturelement des Typs $t1$, so enthält die Hilfsrelation das Tupel $(a1, t1)$.

In dem betrachteten Beispiel enthält die Hilfsrelation HI_{AT} unter anderem die Information, dass das Architekturelement Patientenaufnehmer mindestens ein Material kennt. Abbildung 59 zeigt die Ist-Architektur des Beispiels und hebt die Anteile hervor, aus denen sich diese Information schließen lässt.

Die Menge der Verstöße V_{GR} gegen Gebotsregeln ergibt sich, wenn man die Hilfsrelationen HS_{AT} und HI_{AT} miteinander vergleicht.

- HS_{AT} beschreibt, was laut Gebotsregeln in der Ist-Architektur vorkommen soll
- HI_{AT} beschreibt den tatsächlichen Zustand der Ist-Architektur.

Alle Tupel aus HS_{AT} sollen auch in HI_{AT} vorkommen. Fehlt ein Tupel, so liegt ein Verstoß gegen eine Gebotsregel vor. Formal betrachtet berechnet sich die Menge der Verstöße V_{GR} gegen Gebotsregeln als Differenzmenge der Hilfsrelationen:

$$V_{GR} = HS_{AT} \setminus HI_{AT}$$

Im Beispiel ergibt sich mit dieser Berechnung das erwartete Ergebnis: es liegt ein Verstoß vor, es fehlt eine Beziehung des Therapieplans zu mindestens einem Material:

$$\begin{aligned} & HS_{AT} \setminus HI_{AT} \\ & = \\ & \{(Patientenaufnehmer, Material), (Diagnosesteller, Material), \\ & \quad (Therapieplaner, Material)\} \\ & \quad \setminus \\ & \{(Patientenaufnehmer, Material), (Diagnosesteller, Material), \\ & \quad (Diagnoseteller, Service), (Therapieplaner, Service), \\ & \quad (Therapieplan, Service), (PatientenService, Material)\} \\ & = \\ & \{(Therapieplaner, Material)\} \end{aligned}$$

5.5.2 Berechnungsschritt (d): Verstöße in den Quelltext verfolgen

Dieser Berechnungsschritt dient dazu, zu jedem Verstoß die verursachende Quelltextstelle zu ermitteln. Auf diese Weise erleichtert die stilbasierte Architekturprüfung den Entwicklungsteams, ihren Quelltext zu korrigieren. Die Berechnung der Quelltextstellen ist unterschiedlich, abhängig von dem betroffenen Regeltyp.

Quelltextstellen für Verstöße gegen Verbotsregeln berechnen

Verstöße gegen Verbotsregeln bestehen aus Beziehungen innerhalb der Ist-Architektur. Die Ist-Architektur in Abbildung 58 beispielsweise verstößt an einer Stelle gegen die Regel „Materialien dürfen keine Services kennen“: Die Beziehung des Architekturelements Therapieplan zu dem Architekturelement PatientenService (siehe Abbildung 60) ist nicht erlaubt, da der Therapieplan den Typ Material hat, und der PatientenService den Typ Service. Dieser Verstoß entsteht im Architekturelement Therapieplan, da dieser den PatientenService referenziert.

Möchte das Entwicklungsteam seinen Quelltext korrigieren, so genügt es nicht, zu wissen, welches Architekturelement gegen die Verbotsregel verstößt. Es braucht zusätzlich die Information, welches Quelltextelement den Verstoß verursacht.

Abschnitt 5.3 hat dargestellt, wie die Ist-Architektur und der Quelltext prinzipiell zusammenhängen. Abbildung 61 zeigt beispielhaft den Zusammenhang der von dem Verstoß betroffenen

Architekturelemente zum Quelltext: Das Architekturelement Therapieplan besteht aus der Klasse TherapieplanMaterial. Das Architekturelement PatientenService umfasst das Interface IPatientenService und die Klasse PatientenServiceImpl.

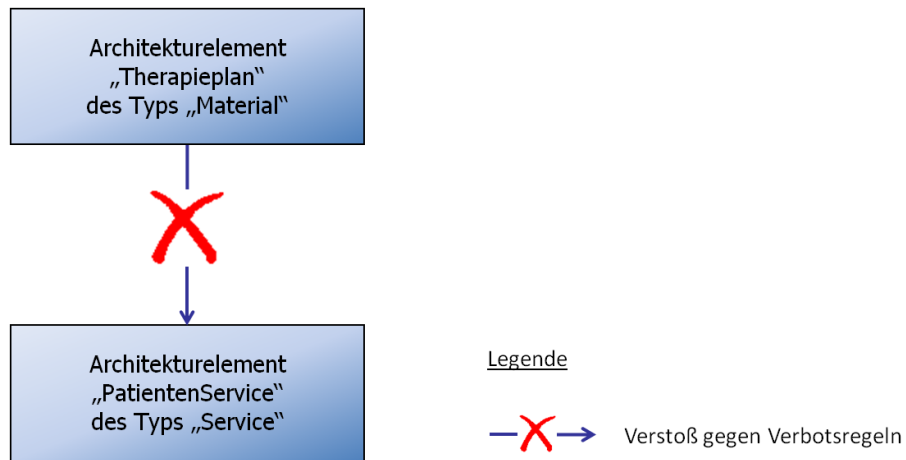


Abbildung 60: Verstoß gegen eine Verbotsregel: das Material Therapieplan referenziert den Service PatientenService

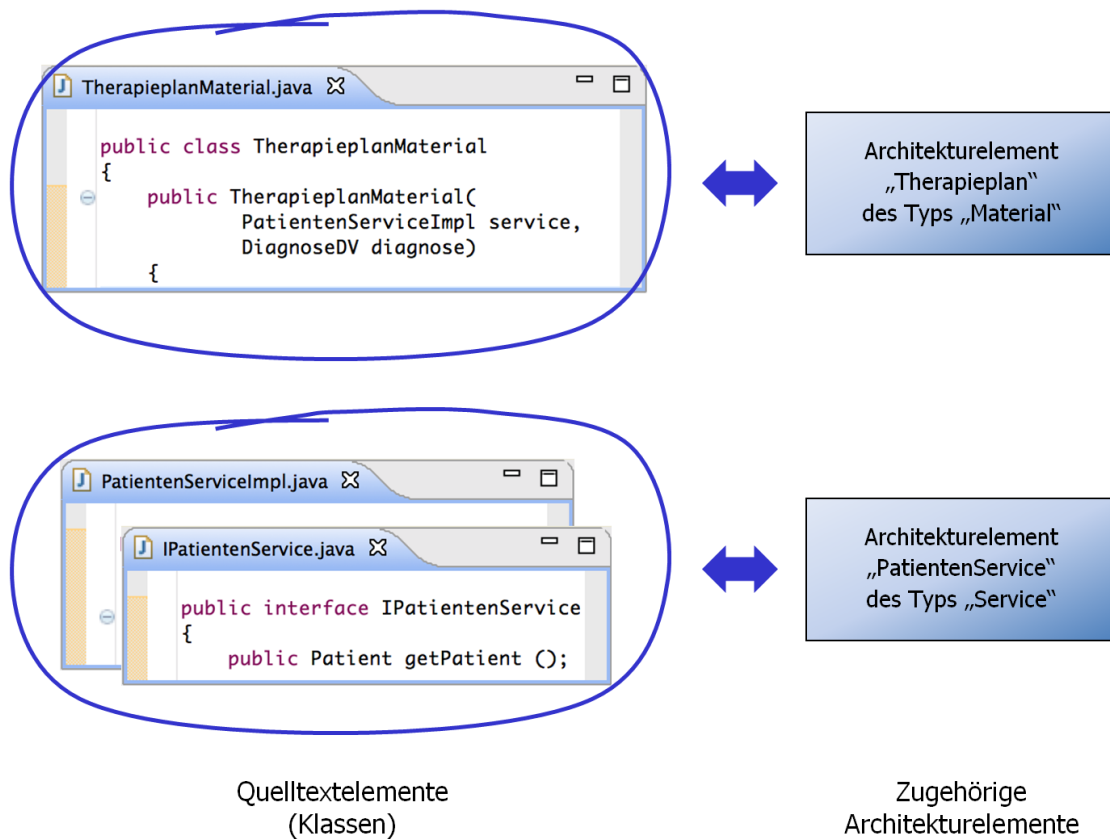


Abbildung 61: Zuordnung zwischen Quelltextelementen und Architekturelementen

In dem Beispiel referenziert die Klasse TherapieplanMaterial die Klasse PatientenServiceImpl (siehe Abbildung 62). Aus dieser Beziehung auf Ebene der Quelltextstruktur resultiert die verbotene Beziehung des Architekturelements Therapieplan zum Architekturelement PatientenService. In diesem Beispiel wird der Verstoß durch ein einzelnes Quelltextelement verursacht. Es ist jedoch im Prinzip auch möglich, dass der Verstoß durch mehrere Quelltextelemente verursacht wird. In dem Beispiel würde das bedeuten, dass das Architekturelement Therapieplan einem oder mehreren weiteren Quelltextelementen zugeordnet sein müsste, die ebenfalls Quelltextelemente referenzieren, die dem PatientenService zugeordnet sind.

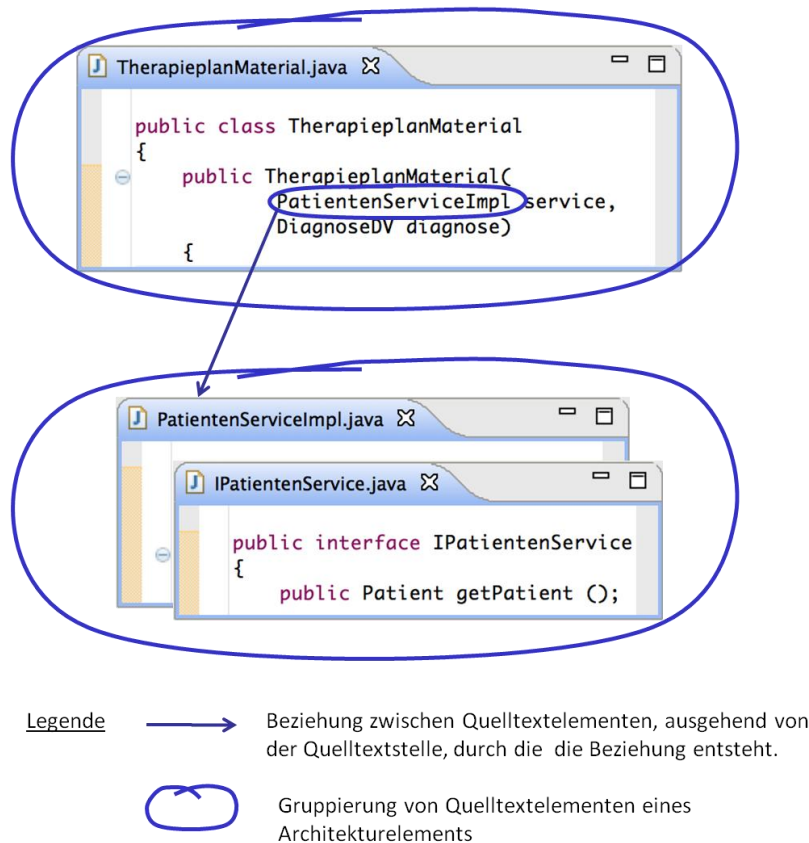


Abbildung 62: Das Quelltextelement TherapieplanMaterial referenziert das Quelltextelement PatientenService und verstößt damit gegen den Stil

Um die betroffenen Quelltextelemente zu ermitteln, benötigt man folgende Mengen und Relationen:

- Die Menge der Verstöße gegen Verbotsregeln $V_{VR} \subseteq A \times A$
- Die Zuordnung zwischen Architekturelementen und Quelltextelementen $Z_{QA} \subseteq Q \times A$
- Die Relation der Quelltextbeziehungen $B_Q \subseteq Q \times Q$

Die betroffenen Quelltextelemente $V_{QVR} \subseteq Q$ berechnen sich folgendermaßen

- Quelltextelemente, die Verstöße gegen Verbotregeln verursachen: $V_{QVR} = \{ q \mid \exists a1, a2, q2: (a1, a2) \in V_{VR} \wedge (q, a1) \in Z_{QA} \wedge (q2, a2) \in Z_{QA} \wedge (q, q2) \in B_Q \}$

Dabei bezeichnen q und $q2$ Quelltextelemente, $a1$ und $a2$ bezeichnen Architekturelemente. Das Quelltextelement q ist genau dann in der Menge der Quelltextelemente, die gegen Verbotregeln verstoßen, wenn gilt: auf Architekturebene verstößt die Beziehung von $a1$ zu $a2$ gegen Verbotregeln: $(a1, a2) \in V_{VR}$ und das Quelltextelement q ist dem Architekturelement $a1$ zugeordnet: $(q, a1) \in Z_{QA}$ und das Quelltextelement $q2$ ist dem Architekturelement $a2$ zugeordnet: $(q2, a2) \in Z_{QA}$ und das Quelltextelement q hat eine Beziehung zu dem Quelltextelement $q2$: $(q, q2) \in B_Q$.

In dem Beispiel gilt: $q = \text{TherapieplanMaterial}$, $q2 = \text{PatientenServiceImpl}$, $a1 = \text{Therapieplan}$, $a2 = \text{PatientenService}$. Die Klasse $\text{TherapieplanMaterial}$ ist Element der Menge V_{QVR} , da gilt:

$$\begin{aligned} & (\text{Therapieplan}, \text{PatientenService}) \in V_{VR} \\ & \quad \wedge \\ & (\text{TherapieplanMaterial}, \text{Therapieplan}) \in Z_{QA} \\ & \quad \wedge \\ & (\text{PatientenServiceImpl}, \text{PatientenService}) \in Z_{QA} \\ & \quad \wedge \\ & (\text{TherapieplanMaterial}, \text{PatientenServiceImpl}) \in B_Q \end{aligned}$$

Werkzeuge, die die stilbasierte Architekturprüfung implementieren, können in der Entwicklungsumgebung direkt bei der Klasse $\text{TherapieplanMaterial}$ eine Fehlermeldung anzeigen, die besagt, dass die Beziehung zum $\text{PatientenServiceImpl}$ von den Vorgaben des Architekturstils abweicht, da Materialien nicht auf Services zugreifen dürfen.

Quelltextstellen für Verstöße gegen Gebotsregeln berechnen

Gebotsregeln nennen zwei Typen. Sie schreiben vor, dass Architekturelemente des ersten Typs eine Beziehung haben müssen zu mindestens einem Architekturelement des zweiten Typs. Die in dem Beispiel betrachtete Gebotsregel lautet: „Werkzeuge müssen Materialien kennen“. Das Architekturelement Therapieplaner verstößt gegen diese Regel, es hat keine Beziehung zu einem Material.

Verstöße gegen Gebotsregel bestehen aus jeweils einem Paar: dem Architekturelement, dem die Beziehung fehlt, und dem Typ, zu dem die Beziehung gehen sollte. Im Beispiel lautet dieses Paar: Therapieplaner , Material .

Formal handelt es sich bei Verstößen um eine Relation zwischen Architekturelementen und Architekturelement-Typen: $V_{GR} \subseteq A \times T_A$. Für das Beispiel gilt:

$$V_{GR} = \{(\text{Therapieplaner}, \text{Material})\}$$

Um diesen Verstoß zu korrigieren, muss das Entwicklungsteam seinen Quelltext so ändern, dass das fehlerhafte Architekturelement eine neue Beziehung erhält, die den Gebotsregeln entspricht. Dazu benötigt das Team die Information, welche Quelltextelemente zu dem fehlerhaften Architekturelement gehören, um mindestens eines dieser Quelltextelemente um eine entsprechende Beziehung zu ergänzen.

Im Beispiel bedeutet dies, dass ein Quelltextelement geändert werden muss, das zu dem Architekturelement Therapieplaner gehört. Die stilbasierte Architekturprüfung liefert dem Entwicklungsteam die Information, dass das Architekturelement namens Therapieplaner aus den Klassen TherapieplanerUI und TherapieplanerMonoTool besteht (siehe Abbildung 63). Die neu einzufügende Quelltextbeziehung muss zu einem Quelltextelement führen, das zu einem Material gehört. Dadurch entspricht der Therapieplaner wieder der Gebotsregel, dass jedes Werkzeug mindestens ein Material kennen muss.

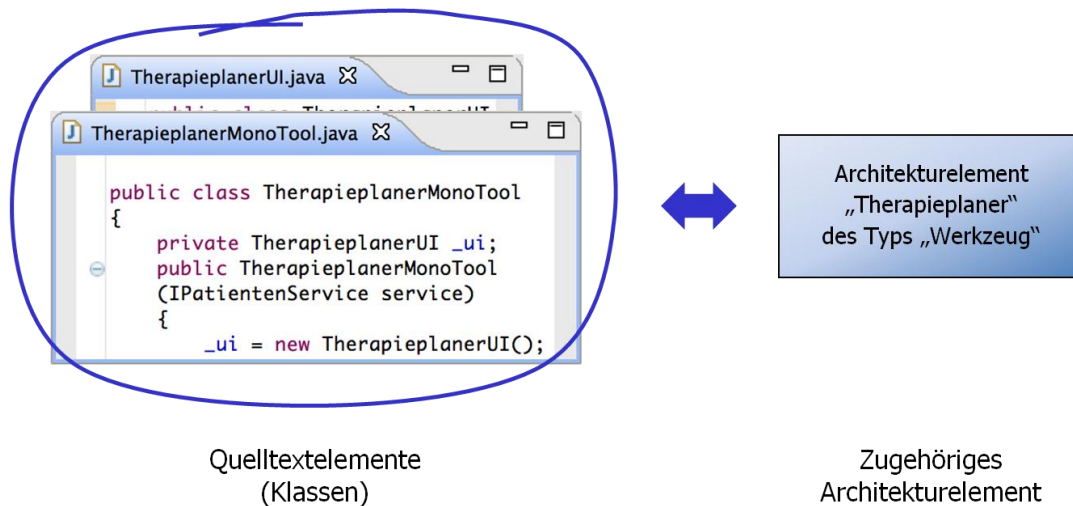


Abbildung 63: Das Architekturelement Therapieplaner gehört zu den Quelltextelementen TherapieplanerUI und TherapieplanerMonoTool

Würde das Entwicklungsteam beispielsweise die Klasse TherapieplanMonoTool um eine Beziehung zur Klasse TherapieplanMaterial ergänzen, so wäre der Verstoß beseitigt, da die Klasse TherapieplanMaterial zu einem Architekturelement des Typs Material gehört³⁰. Die Information, welche Klassen zu einem Material gehören, erhält das Entwicklungsteam dadurch, dass die stilbasierte Architekturprüfung den Zusammenhang zwischen Quelltext und Architektur in der Entwicklungsumgebung deutlich macht (siehe Abschnitt 5.3.3).

Formal gesehen berechnet sich die Menge der betroffenen Quelltextelemente bei Gebotsregeln aus folgenden Mengen und Relationen:

- Die Menge der Verstöße gegen Gebotsregeln $V_{GR} \subseteq A \times T_A$
- Die Zuordnung zwischen Architekturelementen und Quelltextelementen $Z_{QA} \subseteq Q \times A$

Die betroffenen Quelltextelemente $V_{QGR} \subseteq Q$ berechnen sich folgendermaßen

- Quelltextelemente, die Verstöße gegen Gebotsregeln verursachen:

$$V_{QGR} = \{ q \mid (q, a) \in Z_{QA} \wedge (a, t) \in V_{GR} \}$$

³⁰ In dem hier vorgestellten Beispiel ließe sich der Verstoß formal gesehen auch beseitigen, indem die Klasse TherapieplanUI eine neue Beziehung zu einer Material-Klasse erhält. Leserinnen und Leser, die den WAM-Stil kennen, könnten darüber stolpern, da sie wissen, dass UIs nicht auf Materialien zugreifen sollen. Um das Beispiel kurz und übersichtlich zu halten enthält es nur einen vereinfachten Ausschnitt des WAM-Stils.

Dabei bezeichnet q ein Quelltextelement, a ein Architekturelement und t einen Architekturelement-Typ. Es gilt: q ist genau dann in V_{QGR} enthalten, wenn q einem Architekturelement a zugeordnet ist: $(q, a) \in Z_{QA}$ und wenn a gegen eine Gebotsregel verstößt: $(a, t) \in V_{GR}$.

In dem Beispiel besteht die Menge der Quelltextelemente V_{QGR} aus den zwei Quelltextelementen TherapieplanerUI und TherapieplanerMonoTool. Beispielfähig soll die formale Begründung für das Quelltextelement TherapieplanerMonoTool genannt werden:

$$\begin{aligned} & (TherapieplanerMonoTool, Therapieplaner) \in Z_{QA} \\ & \wedge \\ & (Therapieplaner, Material) \in V_{GR} \end{aligned}$$

Ein Werkzeug, das die stilbasierte Architekturprüfung unterstützt, kann in der Entwicklungsumgebung die Fehlermeldung anzeigen, dass die Klassen TherapieplanerMonoTool und TherapieplanerUI von dem Architekturstil abweichen, da ihnen eine Beziehung zu einem Material fehlt.

5.6 Zusammenfassung

Der Ansatz der stilbasierte Architekturprüfung ermittelt *Verstöße gegen beliebige Architekturstile*. Mit dem Ansatz können Entwicklungsteams ihren *individuellen* Stil beschreiben und ihr System auf Stiltreue prüfen. Der Ansatz ermittelt nicht nur die Verstöße innerhalb der Ist-Architektur, sondern berechnet auch die von den Verstößen betroffenen *Quelltextstellen*. So können Prüfwerkzeuge, wie der in Kapitel 7 vorgestellte StyleBasedChecker, die Verstöße direkt in der Entwicklungsumgebung aufzeigen.

Zu Beginn dieses Kapitels wurde ein Überblick über den Beitrag der stilbasierten Architekturprüfung gegeben. Indem sie Architekturvorgaben in Form *individuell beschreibbarer Architekturstile* für Konformanzprüfungen adressiert, betritt sie wissenschaftliches Neuland.

Es folgte das Kernkonzept der stilbasierten Architekturprüfung, dieses erlaubt Konformanzprüfungen zu ungetypten Beziehungsregeln. Beziehungsregeln spielen eine bedeutende Rolle in Architekturstilen. In der Literatur vorgestellte Stile, so auch die im Rahmen dieser Arbeit genauer betrachteten Stile WAM und Quasar, definieren eine Vielzahl an Beziehungsregeln. Auch viele der bisherigen Ansätze für Architekturprüfungen stellen Beziehungen in den Fokus.

In diesem Kapitel wurden die einzelnen Berechnungsschritte des Kernkonzepts und die jeweils benötigten Eingabeinformationen detailliert erläutert und an Beispielen demonstriert. Der Ansatz wählt zur Formalisierung eine mengentheoretische Notation, da diese gut geeignet ist, die notwendigen Strukturinformationen und Berechnungen darzustellen.

Die stilbasierte Architekturprüfung enthält ein Konzept, das erlaubt, die Zusatzinformationen mit dem Quelltext zu verbinden (siehe Abschnitt 5.3.3). Wird dieses Konzept genutzt, so kann die gesamte Ist-Architektur dem annotierten Quelltext entnommen werden. Dies hat mehrere Vorteile:

- Durch die enge Verbindung zwischen Quelltext und Zusatzinformationen ist die Gefahr verringert, dass bei Quelltextänderungen die Zusatzinformationen veralten. Ändert ein Teammitglied den Quelltext, so sieht es unmittelbar die zugeordneten Zusatzinformationen und kann diese überarbeiten. Durch die enge Verbindung kann es nicht geschehen, dass sich Zusatzinformationen auf einen bereits gelöschten Quelltext beziehen oder auf einen Quelltextbezeichner, der mittlerweile umbenannt wurde.

- Die Zusatzinformationen ermöglichen den Programmierern und Programmierern, dass sie direkt in der Entwicklungsumgebung über die Ist-Architektur des Software-systems informiert werden. So sehen sie unmittelbar, zu welchem Architekturelement der bearbeitete Quelltext gehört und welchen Typ dieses Architekturelement hat. Die Programmierern und Programmierer benötigen den Typ, da sich die Regeln des Architekturstils auf Architekturelement-Typen beziehen. Diese Information erleichtert es, die Vorgaben des gewählten Architekturstils einzuhalten.

Werden Softwaresysteme direkt während der Programmierung geprüft, so rückt die Architektur in das Blickfeld. Auf diese Weise wird eine architekturzentrierte Programmierung unterstützt (Englisch: *architecture-aware programming*³¹). Die Programmierern und Programmierer können in der Entwicklungsumgebung die Abstraktionen auf Architekturebene sehen, auf deren Basis sie kommunizieren, wie beispielsweise Architekturelemente des Typs Use-Case oder des Typs Material. Der Architekturstil dient nicht mehr nur als Konzept und Sprache; er wird auch bei der technischen Umsetzung sichtbar und nutzbar.

Damit die Berechnung der Verstöße leichtgewichtig und automatisiert durchführbar ist, entkoppelt die stilbasierte Architekturprüfung die manuelle *Eingabe* der Stilbeschreibung und der Zusatzinformationen zur Ist-Architektur von der *Prüfung* auf Verstöße:

- Der Architekturstil braucht lediglich einmalig vor der ersten Prüfung beschrieben zu werden. Wenn die Architektur evolviert oder der Stil in anderen Systemen wiederverwendet wird, kann die Beschreibung unverändert übernommen werden. Dies ist möglich, da die in dieser Arbeit konzipierte Stilbeschreibung keinerlei Referenzen zur konkreten Architektur enthält.
- Die Zusatzinformationen zur Ist-Architektur können direkt während der Programmierung eingegeben werden, so dass sie jederzeit für die Prüfung zur Verfügung stehen. Dies erfordert für die meisten stilbasierten Projekte keinen Mehraufwand, da diese Informationen üblicherweise auch ohne die stilbasierte Architekturprüfung in Softwaresystemen annotiert werden.

Durch diese Entkopplung der Benutzereingaben von der Prüfung bietet die stilbasierte Architekturprüfung den Vorteil, dass Entwicklungsteams den Prüfungszeitpunkt beliebig wählen können. So ist der Ansatz der stilbasierten Architekturprüfung vielfältig einsetzbar: Er erlaubt die Prüfung bestehender Systeme, um bereits vorhandene Verstöße aufzudecken, beispielsweise als Grundlage für Architekturreviews. Weiterhin erlaubt der Ansatz die Prüfung während der Programmierung, um Teammitglieder unmittelbar über neue Verstöße zu informieren. So können die Teammitglieder ihren Quelltext korrigieren, bevor sie ihn weitergeben. Auch zeit- oder ereignisgesteuerte Prüfungen sind möglich, beispielsweise beim Übertragen von Quelltexten in ein Repository, in nächtlichen Prüfläufen oder bei automatisierten Builds.

Das Konzept der stilbasierten Architekturprüfung wurde für statisch getypte, objektorientierte Programmiersprachen und zugehörige Architekturstile entwickelt. Da mehrere verbreitete Stile sowohl für objektorientierte als auch für imperative Sprachen nutzbar sind (wie beispielsweise der Client-Server-Stil und der Schichtenstil), sollte sich der Ansatz auch für imperative, nicht-objektorientierte Systeme nutzen lassen. Der Ansatz unterstützt frei wählbare Stile und Architekturen und ist unabhängig von einem konkreten Werkzeug.

³¹ Die Autorin hat ihr Konzept des „architecture-aware programming“ erstmalig im Rahmen eines OOPSLA-Konferenz-Workshops vorgestellt und diskutiert (Becker-Pechau et al. 2004)

6 Ausbaustufen der stilbasierten Architekturprüfung

Dieses Kapitel präsentiert drei Ausbaustufen (auch als Erweiterungen bezeichnet). Die Ausbaustufen ergänzen das Kernkonzept der stilbasierten Architekturprüfung. Mit ihnen lässt sich die Konformanz stilbasierter Systeme zu weiteren Regeltypen prüfen. Die im Kernkonzept vorgestellten vier Berechnungsschritte gelten jeweils auch für die Ausbaustufen. Die Ausbaustufen fügen keine weiteren Schritte hinzu, sondern ergänzen bestehende Berechnungsschritte und deren Eingabeinformationen. Zur Orientierung bei der Lektüre der Ausbaustufen fasst Abbildung 64 alle im Kernkonzept vorgestellten Berechnungsschritte und Eingabeinformationen in einer Grafik zusammen.

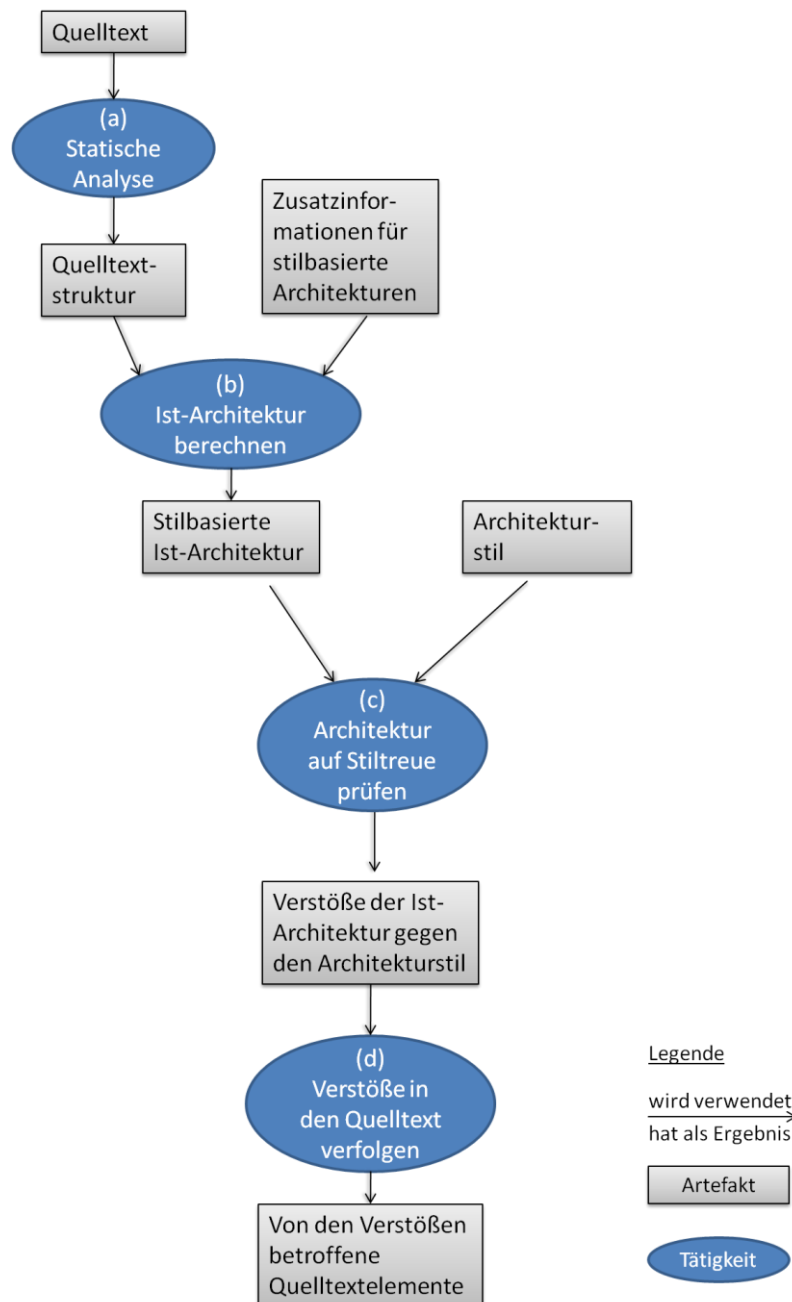


Abbildung 64: Die vier Berechnungsschritte der stilbasierten Architekturprüfung

6.1 Beziehungsregeln für verschiedene Beziehungstypen

Mit dem Kernkonzept der stilbasierten Architekturprüfung lässt sich ausdrücken, ob Beziehungen vorgeschrieben oder verboten sind. Einige Architekturstile – so auch die in dieser Arbeit ausführlich betrachteten Stile WAM und Quasar – unterscheiden jedoch bei ihren Beziehungsregeln, ob sie Vererbungs- oder Benutztbeziehungen betreffen. Im WAM-Stil gilt, dass Elemente verschiedenen Typs generell nicht voneinander erben dürfen. Alle für den Stil genannten Beziehungsregeln (siehe 3.3.1) betreffen Benutztbeziehungen, so auch die oben genannte Gebotsregel „Werkzeuge müssen Materialien kennen“. Dieser Abschnitt erweitert das Kernkonzept der stilbasierten Architekturprüfung um die Möglichkeit, *beliebige* Beziehungstypen zu unterscheiden, damit sich beispielsweise prüfen lässt, dass Werkzeuge zwar Materialien *benutzen* sollen, nicht jedoch von ihnen *erben* dürfen.

Für die in dieser Arbeit betrachtete statische Architektursicht objektorientierter Softwaresysteme werden üblicherweise die Benutzt- und die Vererbungsbeziehung unterschieden (Lilienthal 2008). Diese Unterscheidung lässt sich auch auf der Ebene des Quelltextes objektorientierter Systeme wiederfinden (siehe Kapitel 2). In der Programmiersprache Java beispielsweise lässt sich über das Schlüsselwort `extends` ausdrücken, dass eine Klasse B von einer Klasse A erbt (Abbildung 65). Die Klasse B hat in diesem Fall eine Vererbungsbeziehung zur Klasse A. Auf die gleiche Weise können in Java Interfaces einander beerben³². Klassen, die Interfaces beerben, verwenden das Schlüsselwort `implements`.

```
/**
 * Die Klasse B erbt von der Klasse A
 *
 * @author Geheim
 * @version 2.3.4
 */
public class B extends A
{
```

Abbildung 65: Eine Vererbungsbeziehung von B zu A in der Sprache Java

Eine Benutztbeziehung zwischen zwei Klassen liegt vor, wenn in dem Quelltext einer Klasse eine Operation einer anderen Klasse aufgerufen wird oder wenn der Quelltext der einen Klasse die andere Klasse als Typ verwendet (siehe Kapitel 2). Abbildung 66 zeigt ein Beispiel für eine Benutztbeziehung in Java.

³² Der Lesbarkeit halber steht der Begriff der Vererbung in dieser Arbeit generell für Subtyp-Beziehungen. Das schließt die in Java vorhandene Implementiert-Beziehung zwischen Klassen und Interfaces mit ein.


```

/**
 * Dies Klasse C benutzt die Klasse D
 *
 * @author Geheim
 * @version 2.3.4
 */
public class C
{
    // Typreferenz: D ist der Typ einer
    // Exemplarvariablen:

    private D _myD;
}

```

Abbildung 66: Eine Benutztbeziehung von C zu D in der Sprache Java

Die Benutzt- und die Vererbungsbeziehungen auf Ebene der Ist-Architektur berechnen sich durch Aggregation der Quelltextbeziehungen (siehe auch Kapitel 2): Ein Architekturelement A hat genau dann eine Benutztbeziehung zu einem Architekturelement B, wenn mindestens eines der Quelltextelemente von A eine Benutztbeziehung zu mindestens einem der Quelltextelemente von B hat. Dasselbe gilt analog für die Vererbungsbeziehungen. Abbildung 67 zeigt ein Beispiel. Die Benutztbeziehungen der Klasse 1 zu den Klassen 3 und 4 aggregieren sich auf der Architekturebene zu einer Benutztbeziehung des Architekturelements A zu dem Architekturelement B. Die Vererbungsbeziehung der Klasse 4 zur Klasse 2 führt auf Architekturebene zu einer Vererbungsbeziehung des Architekturelements B zu A.

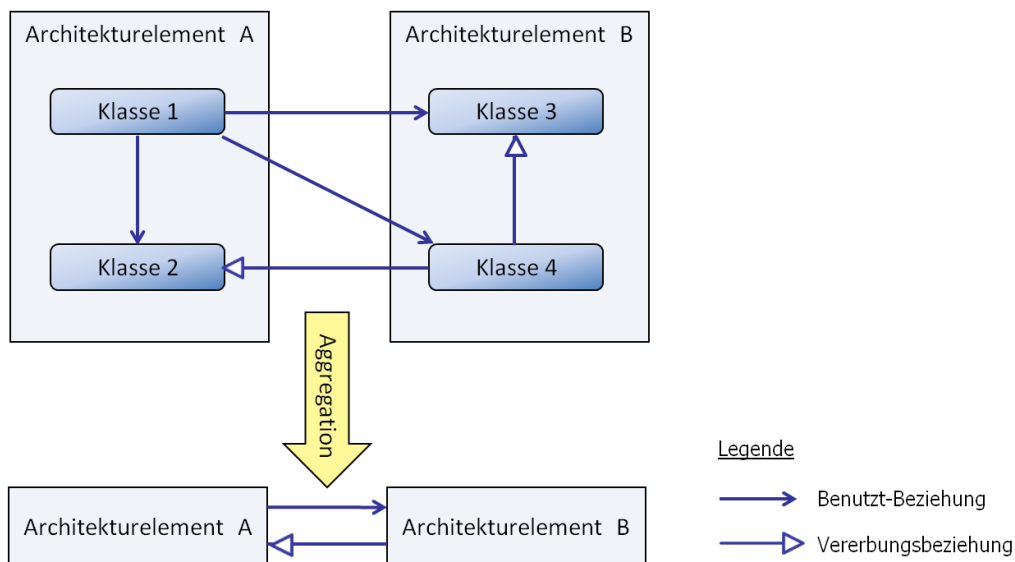


Abbildung 67: Aggregation von Benutzt- und Vererbungsbeziehungen

Mit dem Ansatz der stilbasierten Architekturprüfung lassen sich unter anderem die zwei genannten Beziehungstypen unterscheiden. Der Ansatz geht jedoch darüber hinaus, er ist in Bezug auf die Art und die Anzahl von Beziehungstypen flexibel: es ist möglich, beliebige Beziehungstypen zu unterscheiden. Beispielsweise kann man verschiedene Arten von Benutzbeziehungen unterscheiden, wie Typpreferenzen oder asynchrone Methodenaufrufe.

Welche Beziehungstypen im konkreten Fall unterschieden werden sollten, ist abhängig von der gewählten Programmiersprache, der Architektur und dem Architekturstil.

6.1.1 Berechnungsschritte und Eingabeinformationen

Diese Erweiterung ergänzt alle vier Berechnungsschritte (a bis d) des Kernkonzepts (siehe Abbildung 64, Seite 119). Ferner wird das Beschreibungsformat für Architekturstile ergänzt, damit sich Regeln für verschiedene Beziehungstypen ausdrücken lassen. Die Zusatzinformationen bleiben für diese Erweiterung unverändert.

Erweiterung des Beschreibungsformats für Architekturstile

Das Kernkonzept der stilbasierten Architekturprüfung erlaubt folgendes Beschreibungsformat für Architekturstile (siehe Abschnitt 5.4): Ein Architekturstil enthält die Menge der Architekturelement-Typen T_A , die Menge der Gebotsregeln R_G und die Menge der Verbotsregeln R_V . Im Kernkonzept existiert somit eine Relation für die Gebotsregeln und eine Relation für die Verbotsregeln.

Um Beziehungsregeln für verschiedene Beziehungstypen ausdrücken zu können, ergänzt diese Erweiterungsstufe das Beschreibungsformat für Architekturstile so, dass für jeden Beziehungstyp eine individuelle Menge an Gebots- und Verbotsregeln formuliert werden kann. Unterscheidet ein Stil beispielsweise Benutz- und Vererbungsbeziehungen, so werden die Gebots- und Verbotsregeln dieses Stils über vier Relationen beschrieben:

- Die Gebotsregel für Benutzbeziehungen
- Die Verbotsregeln für Benutzbeziehungen
- Die Gebotsregeln für Vererbungsbeziehungen
- Die Verbotsregeln für Vererbungsbeziehungen

Diese Erweiterung lässt sich am Beispiel des WAM-Stils illustrieren. Natürlichsprachlich ausgedrückt gilt für den WAM-Stil beispielsweise:

- Eine Gebotsregel für Benutzbeziehungen: „Werkzeuge müssen Materialien benutzen“. Das bedeutet, jedes Werkzeug muss mindestens ein Material benutzen.
- Eine Verbotsregel für Benutzbeziehungen: „Materialien dürfen keine Services benutzen“.
- Da der WAM-Stil Vererbungsbeziehungen zwischen verschiedenen Elementtypen generell ausschließt, existiert keine Gebotsregel für Vererbungsbeziehungen.
- Eine Verbotsregel für Vererbungsbeziehungen „Werkzeuge dürfen nicht von Materialien erben“.

Für die formale Darstellung werden die Beziehungstypen nummeriert. Die Nummer des jeweiligen Beziehungstyps wird über den hochgestellten Index dargestellt, n bezeichnet die Anzahl der Beziehungstypen. Somit ergibt sich folgendes erweitertes Beschreibungsformat für Architekturstile:

- Die Menge der Architekturelement-Typen T_A ,
- n Mengen der Gebotsregeln $R_G^1, R_G^2, \dots, R_G^n$
- und n Mengen der Verbotsregeln $R_V^1, R_V^2, \dots, R_V^n$.

Bei dem Quasar- und dem WAM-Stil hat n den Wert 2, da diese Stile mit der Vererbungsbeziehung und der Benutztbeziehung zwei verschiedene Beziehungstypen unterscheiden. Die Indizes können durch eindeutige Begriffe ersetzt werden, um die Leserlichkeit zu erhöhen. Im Beispiel des WAM-Stils lässt sich somit festlegen: die zwei Indizes werden ersetzt durch die Begriffe Vererbung und Benutzt. Formal stellen sich damit die oben genannten Regeln folgendermaßen dar:

- Die Gebotsregel für Benutztbeziehungen: „Werkzeuge müssen Materialien benutzen“ bedeutet formal: $R_G^{Benutzt} \ni (Werkzeug, Material)$.
- Die Verbotsregel für Benutztbeziehungen: „Materialien dürfen keine Services benutzen“ bedeutet formal: $R_V^{Benutzt} \ni (Material, Service)$.
- Die Verbotsregel für Vererbungsbeziehungen „Werkzeuge dürfen nicht von Materialien erben“. bedeutet formal: $R_V^{Vererbung} \ni (Werkzeug, Material)$. Der WAM-Stil schließt Vererbungsbeziehungen zwischen verschiedenen Elementtypen generell aus. Somit gilt: $R_V^{Vererbung} = T_A \times T_A \setminus \{(a, a) \mid a \in T_A\}$, das bedeutet, die Menge $R_V^{Vererbung}$ enthält alle möglichen Paare von unterschiedlichen Elementtypen. Allerdings ist Vererbung, genau wie Benutztbeziehungen, zwischen Elementen desselben Typs im WAM-Stil grundsätzlich erlaubt.

Erweiterung der vier Berechnungsschritte

Die Ausbaustufe für Beziehungstypen ergänzt die Berechnungsschritte (a) bis (d) nach demselben Prinzip wie sie das Beschreibungsformat für Architekturstile ergänzt: indem sie jeweils getrennte Mengen für die verschiedenen Beziehungstypen einführt.

Im Folgenden gilt durchgehend:

- n bezeichnet die Anzahl der Beziehungstypen,
- der hochgestellte Index m markiert einen konkreten Beziehungstyp.

Es gilt $1 \leq m \leq n$ mit $m, n \in \mathbb{N}$. Der Index m lässt sich durch sprechende Namen ersetzen, um die Lesbarkeit zu erhöhen.

Erweiterung des Berechnungsschritts (a): die Quelltextstruktur ermitteln

Die Quelltextstruktur des Kernkonzepts besteht aus der Menge Q der Quelltextelemente und der Relation $B_Q \subseteq Q \times Q$ der Quelltextbeziehungen. Um verschiedene Beziehungstypen zu

unterstützen, wird die Quelltextstruktur erweitert. Die erweiterte Version enthält eine Relation für jeden Beziehungstyp. Mit dieser Erweiterung stellen sich die Mengen und Relationen der Quelltextstruktur folgendermaßen dar.

- Die Menge der Quelltextelemente: Q
- n Relationen von Quelltextbeziehungen $B_Q^m \subseteq Q \times Q$

Für das Beispiel in Abbildung 67 gilt somit beispielsweise:

- $B_Q^{Benutzt} = \{(Klasse\ 1, Klasse\ 2), (Klasse\ 1, Klasse\ 3), (Klasse\ 1, Klasse\ 4)\}$
- $B_Q^{Vererbung} = \{(Klasse\ 4, Klasse\ 2), (Klasse\ 4, Klasse\ 3)\}$

Erweiterung des Berechnungsschritts (b): die stilbasierte Ist-Architektur ermitteln

Diese Erweiterungsstufe ergänzt auch die stilbasierte Architektur um Beziehungstypen, genauso wie die Quelltextstruktur. Die erweiterte Definition der Ist-Architektur enthält ebenfalls jeweils eine Relation für jeden Beziehungstyp. Es gilt:

- stilbasierte Architekturen enthalten n Relationen von Beziehungen zwischen Architekturelementen $B_A^m \subseteq A \times A$

Abbildung 67 stellt beispielhaft dar, wie die stilbasierte Architekturprüfung die verschiedenen typisierten Quelltextbeziehungen zu Beziehungen der Ist-Architektur durch Aggregation berechnet. Die Berechnung erfolgt für jeden Beziehungstyp in der gleichen Weise, wie im Kernkonzept der stilbasierten Architekturprüfung. Die Berechnung muss lediglich, im Gegensatz zum Kernkonzept, mehrfach ausgeführt werden, für jeden Beziehungstyp einzeln. Beispielsweise gilt in Abbildung 67 bezüglich der Vererbungsbeziehung:

- $B_Q^{Vererbung} = \{(Klasse\ 4, Klasse\ 2), (Klasse\ 4, Klasse\ 3)\}$

Ferner gilt für die Zuordnung zwischen Quelltext und Ist-Architektur:

- $Z_{QA} = \{(Klasse\ 1, Architekturelement\ A), (Klasse\ 2, Architekturelement\ A), (Klasse\ 3, Architekturelement\ B), (Klasse\ 4, Architekturelement\ B)\}$

Somit ergibt sich eine Vererbungsbeziehung innerhalb der Ist-Architektur vom Architekturelement B zu A:

- $B_A^{Vererbung} = \{(Architekturelement\ B, Architekturelement\ A)\}$

Die Formel für die Beziehungen der stilbasierten Ist-Architektur unterscheidet sich von dem Kernkonzept durch das hochgestellte m , das die Beziehungstypen indiziert. Die Beziehungen der Ist-Architektur berechnen sich für jeden Beziehungstyp m folgendermaßen:

- $B_A^m = \{(a, b) | \exists p, q : (p, q) \in B_Q^m \wedge (p, a) \in Z_{QA} \wedge (q, b) \in Z_{QA}\}$

Beispielsweise berechnet sich aus dieser Formel für die Architektur in Abbildung 67, dass gilt $(Architekturelement\ B, Architekturelement\ A) \in B_A^{Vererbung}$. Dies ergibt sich mit der

Belegung a = Architekturelement B, b = Architekturelement A, p = Klasse 4, q = Klasse 2 und m = Vererbung. In diesem Fall gilt:

$$\begin{aligned} & (\text{Klasse 4, Klasse 2}) \in B_Q^{\text{Vererbung}} \\ & \quad \wedge \\ & (\text{Klasse 4, Architekturelement B}) \in Z_{QA} \\ & \quad \wedge \\ & (\text{Klasse 2, Architekturelement A}) \in Z_{QA} \end{aligned}$$

Erweiterung des Berechnungsschritts (c): Die Ist-Architektur auf Stiltreue prüfen

Diese Erweiterungsstufe der stilbasierten Architekturprüfung berechnet die Verstöße innerhalb der Ist-Architektur gegen Verbots- und Gebotsregeln jeweils einzeln für jeden Beziehungstyp. Die Berechnung selber ändert sich nicht, sie bleibt so, wie im Kernkonzept dargestellt (siehe Abschnitt 5.5.1). Ergebnis der Prüfung auf Stiltreue bezüglich der Gebots- und Verbotsregeln sind jeweils n verschiedene Mengen an Verstößen. Die Mengen der Verstöße werden wie oben durch den hochgestellten Index unterschieden. Beispielsweise enthält die Menge $V_{VR}^{\text{Vererbung}}$ alle Verstöße gegen die Verbotsregeln in $R_V^{\text{Vererbung}}$.

Das Prüfungsergebnis enthält folgende Mengen:

- Verstöße gegen Verbotsregeln: n Mengen $V_{VR}^m \subseteq A \times A$.
- Verstöße gegen Gebotsregeln: n Mengen $V_{GR}^m \subseteq A \times T_A$

Erweiterung des Berechnungsschritts (d): Verstöße in den Quelltext verfolgen

Die Quelltextstellen, die für die Verstöße verantwortlich sind, werden für jeden Beziehungstyp einzeln berechnet. Das Kernkonzept berechnet für die Verstöße gegen Gebots- und gegen Verbotsregeln jeweils eine Menge. Diese Erweiterung berechnet jeweils n Mengen für die Verstöße gegen Gebots- und gegen Verbotsregeln. Berechnet man beispielsweise die betroffenen Quelltextstellen für die Verstöße gegen die Gebotsregeln für Vererbung, so bezeichnet $V_{GR}^{\text{Vererbung}}$ die Menge der Verstöße, und $V_{QGR}^{\text{Vererbung}}$ bezeichnet die betroffenen Quelltextstellen. Insgesamt enthält das Ergebnis dieses Berechnungsschrittes somit folgende Mengen:

- Die betroffenen Quelltextelemente der Verstöße gegen Verbotsregeln: n Mengen $V_{QVR}^m \subseteq Q$
- Die betroffenen Quelltextelemente der Verstöße gegen Gebotsregeln: n Mengen $V_{QGR}^m \subseteq Q$

Wie im vorherigen Berechnungsschritt (c), so gilt auch hier: die Berechnung selber ändert sich nicht, für jeden Beziehungstyp einzeln wird die im Kernkonzept definierte Berechnung zur Ermittlung der betroffenen Quelltextstellen verwendet (siehe Abschnitt 5.5.2.)

6.1.2 Abgrenzung zu bisherigen Ansätzen

Die in diesem Abschnitt vorgestellte Ausbaustufe der stilbasierten Architekturprüfung ermöglicht es, Stile zu definieren, die verschiedene *Beziehungstypen* unterscheiden und deren *Beziehungsregeln* sich jeweils auf diese Typen beziehen. Es handelt sich um eine Erweiterung der im Kernkonzept definierten Modellierungsmöglichkeiten für Architekturstile.

An dieser Stelle sei daran erinnert, dass bereits das Kernkonzept der stilbasierten Architekturprüfung über die in Kapitel 4 vorgestellten, vor der stilbasierten Architekturprüfung veröffentlichten Ansätze zur Architektur-Konformanzprüfung hinausgeht, indem die stilbasierte Architekturprüfung die Konformanz zu *frei definierbaren Architekturstilen* prüft. Insofern lässt sich die in diesem Abschnitt vorgestellte Erweiterung mit keinem dieser Ansätze vergleichen, da keiner der Ansätze erlaubt, dass die prüfende Person eigene Architekturstile definiert, weder mit noch ohne Beziehungstypen.

Einige der bisherigen Ansätze erlauben jedoch die Treueprüfung zu *bereits vorgegebenen* Architekturstilen (Schichten und Schnitte), so beispielsweise das Werkzeug Lattix (siehe Abschnitt 4.4.3). Bei einigen Ansätzen umfassen diese vorgegebenen Stile *bereits vorgegebene* Beziehungstypen (wie Vererbungs- und Benutzbeziehungen). Im Gegensatz zu der stilbasierten Architekturprüfung erlauben diese Ansätze allerdings keine Treueprüfung mit *beliebigen* Beziehungstypen.

Manche der bisherigen Ansätze, welche die Treue zu *Soll-Architekturen* prüfen, erlauben die Unterscheidung verschiedener Beziehungstypen innerhalb von Soll-Architekturen. So existiert eine Erweiterung der SRM (Abschnitt 4.4.1), mit der sich beliebige Beziehungstypen für Soll-Architekturen unterscheiden lassen. Diese Erweiterung der SRM und die stilbasierte Architekturprüfung haben zwar gemein, dass sie beliebige Beziehungstypen erlauben, die Definition der Vorgaben und die Berechnung von Architekturverstößen unterscheiden sich jedoch erheblich: nur die stilbasierte Architekturprüfung erlaubt, Regeln für Beziehungstypen zu definieren.

Anders als die in Kapitel 4 vorgestellten Ansätze definiert die stilbasierte Architekturprüfung ein explizites Metamodell für Architekturstile mit getypten Beziehungen und formalisiert die zugehörigen Beziehungsregeln.

6.2 Schnittstellenregeln

Mit der in diesem Abschnitt vorgestellten Ausbaustufe lassen sich *Regeln* für Schnittstellen (kurz: Schnittstellenregeln) festlegen. Schnittstellenregeln machen einheitliche Vorgaben für die Schnittstellen von Architekturelementen eines bestimmten Typs.

Ein Beispiel für eine Schnittstellenregel des WAM-Stils lautet: „Funktionskomponenten dürfen keine Materialien zurückgeben“ (siehe Abschnitt 3.3.1). Diese Regel gilt für alle Architekturelemente des Typs Funktionskomponente. An diesem Beispiel lässt sich nachvollziehen: Schnittstellenregeln machen Aussagen über den Aufbau der Schnittstelle von Architekturelementen bestimmten Typs. Abbildung 68 zeigt ein Beispiel, in dem gegen die genannte Schnittstellenregel verstoßen wird. Der obere Teil zeigt die Klasse `SingleDeviceEditorFP`. Dem Kommentar lässt sich entnehmen, dass es sich um eine Funktionskomponente (functional part) handelt. Darunter ist der Quelltext der Klasse `Device` dargestellt, der Kommentar besagt, dass dies ein Material ist. Im Quelltext der Klasse `SingleDeviceEditorFP` ist die Methode `getDevice` zu sehen, die ein `Device` als Rückgabetypp definiert, die Quelltextstelle ist rot hervorgehoben. An dieser Stelle verstößt das System gegen die genannte Schnittstellenregel, die Funktionskomponente gibt ein Material zurück.

```

SingleDeviceEditorFP.java
import javax.swing.JOptionPane;

/**
 * The functional part of the
 * single device editor tool.
 */
public class SingleDeviceEditorFP
    extends ToolFunctionalityImpl
{
    /**
     * The material worked on by this fp.
     */
    private Device _device;

    /**
     * Returns the material of this fp.
     */
    public Device getDevice()
    {
        return _device;
    }
}

Device.java
/**
 * The material Device.
 */

public class Device extends AbstractThing
    implements CopyAble, Registerable, Self
{

```

Abbildung 68: Beispiel für einen Verstoß gegen die Schnittstellenregel „Funktionskomponenten (functional parts) dürfen keine Materialien zurückgeben“

6.2.1 Berechnungsschritte und Eingabeinformationen

Die in diesem Abschnitt vorgestellte Ausbaustufe der stilbasierten Architekturprüfung erlaubt es, Schnittstellenregeln in der Art des vorgestellten Beispiels zu definieren und Systeme auf deren Einhaltung hin zu prüfen. Die Ausbaustufe ergänzt alle 4 Berechnungsschritte (siehe Abbildung 64, Seite 119) der stilbasierten Architekturprüfung und das Beschreibungsformat für Architekturstile, so dass sich Schnittstellenregeln ausdrücken lassen. Die Zusatzinformationen bleiben unverändert.

Die stilbasierten Architekturprüfung formalisiert diese Ausbaustufe weniger stark als die zuvor vorgestellten Teile des Konzepts. Die zuvor vorgestellten Teile betreffen ausschließlich Beziehungsregeln. Diese zuvor vorgestellten Teile lassen sich aus verschiedenen Gründen gut formalisieren:

- Beziehungen sind im Bereich der Softwarearchitektur ein weit verbreitetes Konzept. Sämtliche Definitionen über Softwarearchitektur erwähnen Beziehungen zwischen Architekturelementen (siehe Abschnitt 2.2).
- Es besteht ein verbreitetes gemeinsames Verständnis darüber, was innerhalb von Quelltexten als Beziehung angesehen werden kann. Über alle diese Beziehungstypen lässt

sich leicht abstrahieren, indem man eine Beziehung als gerichtete Verbindung zwischen zwei Quelltextelementen betrachtet (siehe Kapitel 2).

- Der Bereich der Treueprüfungen besteht fast ausschließlich aus Ansätzen, die Vorgaben für Beziehungen überprüfen.
- Alle in der Literatur vorgestellten Architekturstile, wie auch die zwei in dieser Arbeit beispielhaft untersuchten Stile, enthalten Beziehungsregeln. Die Anzahl der Beziehungsregeln übersteigt die Anzahl der Schnittstellenregeln deutlich.

Schnittstellenregeln sind hingegen ein neues Feld. Diese Arbeit formalisiert die Prüfung von Schnittstellenregeln weniger stark aus folgenden Gründen:

- Im Bereich der Architektur-Konformanzprüfungen wurden Schnittstellen besonders zu Beginn weniger stark beachtet als Beziehungen. Mehrere Ansätze erlauben lediglich die Vorgabe von erlaubten und verbotenen Beziehungen, nicht jedoch von Schnittstellen (beispielsweise der SRM-Ansatz, siehe Kapitel 4)
- Das Verständnis von Schnittstellen im Bereich von Quelltextstrukturen ist nicht etabliert. Die verschiedenen Programmiersprachen definieren den Begriff unterschiedlich, und selbst innerhalb einer Programmiersprache bestehen unterschiedliche Ansichten darüber, was als Schnittstelle zu verstehen ist.
- Viele in der Literatur vorgestellte Architekturstile enthalten keine Schnittstellenregeln. Erst neuere Architekturstile legen Schnittstellenregeln fest, so auch die beiden in dieser Arbeit untersuchten Stile. Die empirische Basis für Schnittstellenregeln ist jedoch noch deutlich geringer als für Beziehungsregeln.

Erweiterung des Berechnungsschritts (a): die Quelltextstruktur ermitteln

In dieser Ausbaustufe erweitert die stilbasierte Architekturprüfung das übliche Verständnis, nach dem Quelltextstrukturen bei Treueprüfungen aus Quelltextelementen und deren Beziehungen bestehen (siehe Kapitel 2 und 4). Sie ergänzt Quelltextstrukturen um Schnittstellen. Dies ist ein neuer Beitrag der vorliegenden Arbeit.

Die Quelltextstruktur der stilbasierten Architekturprüfung besteht mit dieser Ausbaustufe aus Quelltextelementen, deren Beziehungen und deren Schnittstellen. Details zu diesen Begriffen finden sich in Kapitel 2. In obigen Beispiel (Abbildung 68), das an ein reales WAM-System angelehnt ist, enthält die Schnittstelle der Klasse `SingleDeviceEditorFP` unter anderem die Operation `getDevice()` mit dem Rückgabety `Device`. Die gesamte Schnittstelle besteht in diesem Beispiel aus allen öffentlichen Operationen und dem Konstruktor der Klasse. Abbildung 69 zeigt ein weiteres Beispiel, angelehnt an dasselbe Softwaresystem wie in der vorherigen Abbildung. Die Abbildung zeigt den Quelltext der Java-Klasse namens `Device` mit allen Köpfen der Operationen. Die Rümpfe der Operationen wurden ausgeblendet, da sie für die Schnittstelle unerheblich sind. Dem Quelltext lässt sich entnehmen, dass die Schnittstelle dieser Klasse aus einem Konstruktor und sechs Operationen besteht:

- Konstruktor: `public Device()`
- Operation: `public void setDescription (String)`
- Operation: `public String getDescription()`

- Operation: public void setDeviceName(DeviceNameDV)
- Operation: public DeviceNameDV getDeviceName()
- Operation: public DateDV getPurchaseDate ()
- Operation: public void setPurchaseDate(DateDV)

```

import de.itwps.ems.domainvalue.DateDV;

/**
 * The material Device.
 */
public class Device extends AbstractThing
    implements CopyAble, Registerable, SelfDescribing
{
    public Device(){}
    public void setDescription(String description) {}
    public String getDescription() {}
    public DeviceNameDV getDeviceName() {}
    public void setDeviceName (DeviceNameDV deviceName) {}
    public DateDV getPurchaseDate() {}
    public void setPurchaseDate (DateDV purchaseDate) {}
}

```

Abbildung 69: Ein Quelltextbeispiel, angelehnt an ein reales System

In diesem Beispiel gilt, dass die Quelltextelemente aus jeweils einer Java-Klasse bestehen und dass die Schnittstelle solch eines Quelltextelements aus allen öffentlichen Operationen besteht. Dies ist in allen Java-Systemen der Fall, die dem WAM-Stil folgen.

Um bezüglich der Programmiersprache und der Art der zu prüfenden Architektur flexibel zu bleiben, legt die stilbasierte Architekturprüfung nicht unveränderlich fest, was unter einer Quelltextschnittstelle zu verstehen ist. Sie legt lediglich fest, dass Quelltextelemente Schnittstellen haben. Prüfwerkzeuge, die die stilbasierte Architekturprüfung für konkrete Programmiersprachen realisieren, können festlegen, was sie als Quelltextschnittstelle interpretieren. Alternativ können Werkzeuge die prüfende Person entscheiden lassen. Beispielsweise könnte in einer eventbasierten Architektur die Menge der versendeten oder empfangenen Event-Arten als Teil der Schnittstelle betrachtet werden oder es könnten in einer Sprache wie Eiffel, die das Vertragsmodell unterstützt, die Verträge als Teil der Schnittstelle angesehen werden (Meyer 1997).

Dieses Vorgehen ist konsistent zu dem Kernkonzept der stilbasierten Architekturprüfung. Wie in Abschnitt 5.3.1 erläutert, ist bei der stilbasierten Architekturprüfung frei wählbar, was als Quelltextelement, und was als Beziehung angesehen wird. Die Wahl, was genau unter einem Quelltextelement und einer Beziehung zu verstehen ist, wird entweder direkt von der prüfenden

Person getroffen, oder durch das Werkzeug festgelegt, das die stilbasierte Architekturprüfung unterstützt.

Mit dieser Ausbaustufe unterstützt die stilbasierte Architekturprüfung den vollen Umfang von Quelltextstrukturen, wie in Definition 2-2 festgelegt. Das Kernkonzept der stilbasierten Architekturprüfung benötigt formal dargestellt die Menge Q der Quelltextelemente und die Relation B_Q der Quelltextbeziehungen. Diese Ausbaustufe ergänzt dieses Verständnis um:

- Die Menge S_Q der Schnittstellen der Quelltextelemente
- Die Zuordnung zwischen Quelltextelementen und Schnittstellen Z_{QS} . Die Zuordnung ist eine Relation, für die gilt: $Z_{QS} \subseteq Q \times S_Q$

Für diese Ausbaustufe gilt somit: Ein Quelltextstruktur ist ein Tupel $\langle Q, B_Q, S_Q, Z_{QS} \rangle$.

Diese Ausbaustufe formalisiert die Quelltextstruktur so weit, dass jedes Quelltextelement eine Schnittstelle enthält. Die interne Struktur von Quelltextelement-Schnittstellen wird aus den oben genannten Gründen nicht formalisiert. Im Folgenden soll ein Beispiel einer WAM-Architektur illustrieren, wie diese Teilformalisierung für Quelltextstrukturen umgesetzt werden kann. Es sei angemerkt, dass die prüfenden Personen nicht mit dieser Teilformalisierung in Kontakt kommen, da die Quelltextstruktur bei der stilbasierten Architekturprüfung ein internes Zwischenergebnis ist (siehe Abschnitt 5.2). Werkzeuge, die die stilbasierte Architekturprüfung unterstützen, können die Schnittstellen von Quelltextelementen durchaus anders repräsentieren, als in diesem Beispiel. Dieses Beispiel dient lediglich dem Verständnis des Konzepts für die Leserinnen und Leser.

Abbildung 70 zeigt den obigen Quelltext der Klasse Device erneut mit zwei weiteren Klassen. Für das Tupel der Quelltextstruktur $\langle Q, B_Q, S_Q, Z_{QS} \rangle$ gilt in diesem Fall:

- Die Menge der Quelltextelemente:

$$Q = \{Device, DateDV, DeviceNameDV\}$$

- Die Quelltextstruktur enthält zwei Beziehungen. Diese ergeben sich unter anderem aus den blau eingekreisten Quelltextstellen. Um dieses Beispiel übersichtlich zu halten, wird an dieser Stelle nicht zwischen Beziehungstypen unterschieden, genau wie im Kernkonzept der stilbasierten Architekturprüfung. Die Menge der Quelltextbeziehungen enthält somit zwei Tupel:

$$B_Q = \{(Device, DeviceNameDV), (Device, DateDV)\}$$

- Die Menge der Schnittstellen enthält eine Schnittstelle für jedes Quelltextelement. Die Schnittstellen bestehen formal betrachtet aus Zeichenketten. Um dieses Beispiel gut lesbar zu halten, erhalten die Schnittstellen Namen:

*Schnittstelle 1 = „ Device(), void setDescription (String),
String getDescription(), void setDeviceName(DeviceNameDV),
DeviceNameDV getDeviceName(), void setPurchaseDate(DateDV),
DateDV getPurchaseDate ()“*

Schnittstelle 2 = „DateDV(int, int, int), String getDateAsString()“

Schnittstelle 3 = „DeviceNameDV(String), String getString()“

Die Menge der Schnittstellen enthält drei Elemente:

$$S_Q = \{\text{Schnittstelle 1}, \text{Schnittstelle 2}, \text{Schnittstelle 3}\}$$

- Die Schnittstellen sind den Quelltextelementen über folgende Relation zugeordnet:

$$Z_{QS} = \{(Device, \text{Schnittstelle 1}), (DateDV, \text{Schnittstelle 2}), (DeviceNameDV, \text{Schnittstelle 3})\}$$

```
Device.java
import de.itwps.ems.domainvalue.DateDV;

/**
 * The material Device.
 */
public class Device extends AbstractThing
    implements CopyAble, Registerable, SelfDescribing
{
    public Device()
    public void setDescription(String description)
    public String getDescription()
    public DeviceNameDV getDeviceName()
    public void setDeviceName (DeviceNameDV deviceName)
    public DateDV getPurchaseDate()
    public void setPurchaseDate (DateDV purchaseDate)

DeviceNameDV.java
package de.itwps.ems.domainvalue;

public class DeviceNameDV
{
    public DeviceNameDV(String name)

DateDV.java
package de.itwps.ems.domainvalue;

import de.jwam.domainvalue.DomainValue;

/**
 * This class implements a domain value for a date.
 */
public final class DateDV implements DomainValue, Comparable
{
    public DateDV(int year, int month, int day)
```

Abbildung 70: Ein Quelltextbeispiel zur Illustration der Quelltextstruktur

Erweiterung des Berechnungsschritts (b): die stilbasierte Ist-Architektur ermitteln

Die Schnittstellen der Architekturelemente werden auf ähnliche Weise aus der Quelltextstruktur berechnet wie die Beziehungen. Da die bisherigen Ansätze zur Architekturprüfung Schnittstellen in dieser Form nicht beinhalten, setzt die hier vorgestellte Berechnung nicht auf andere Ansätze auf. Sie wurde neu für die stilbasierte Architekturprüfung entworfen.

Im ersten Schritt sei ein Beispiel betrachtet, in dem ein Architekturelement genau aus einem Quelltextelement besteht. In diesem Fall hat das Architekturelement dieselbe Schnittstelle wie das zugehörige Quelltextelement. Die Klasse Device aus dem vorherigen Abschnitt soll als Beispiel für ein Quelltextelement dienen. Sie gehört zu einem Architekturelement namens DeviceArchElem³³ des Typs Material (siehe Abbildung 71).

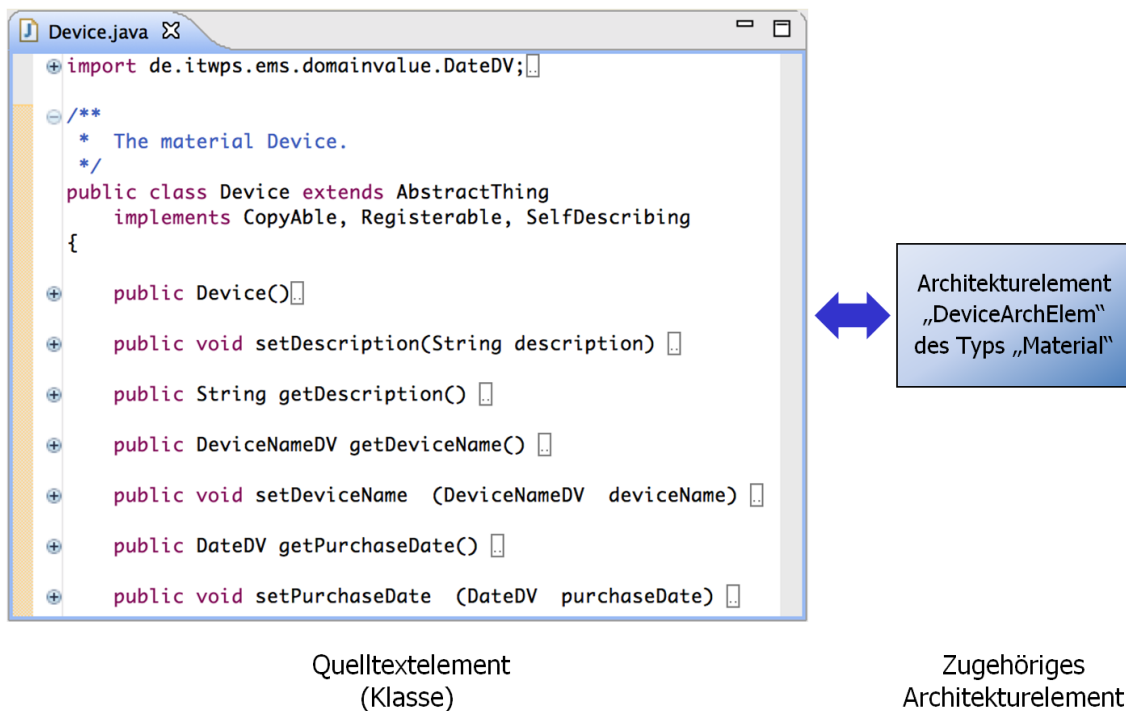


Abbildung 71: Beispiel: Quelltextelement und Architekturelement

Wie im vorherigen Abschnitt erläutert, hat die Klasse Device als Schnittstelle alle öffentlichen Konstruktoren und Operationen. Da das Architekturelement DeviceArchElem nur dem einen Quelltextelement Device zugeordnet ist, gilt in diesem Fall für die Ist-Architektur: DeviceArchElem hat dieselbe Schnittstelle wie Device. Konkret handelt es sich dabei um folgende Konstruktoren und Operationen:

³³ Üblicherweise gilt, dass Architekturelemente, die genau einem Quelltextelement zugeordnet sind, denselben Namen tragen wie das Quelltextelement. Dieses Beispiel weicht bewusst von dieser Konvention ab, um Verwechslungen zu vermeiden.

- Konstruktor: public Device()
- Operation: public void setDescription (String)
- Operation: public String getDescription()
- Operation: public void setDeviceName(DeviceNameDV)
- Operation: public DeviceNameDV getDeviceName()
- Operation: public void setPurchaseDate(DateDV)
- Operation: public DateDV getPurchaseDate ()

An dieser Stelle sei angemerkt, dass alle Schnittstellenregeln der untersuchten Stile zu Architekturelement-Typen gehörten, die üblicherweise durch genau ein Quelltextelement umgesetzt werden. Das genannte Beispiel ist repräsentativ für die untersuchten Beispiele. Im Prinzip ist es jedoch möglich, dass auch hier ein Architekturelement mehreren Quelltextelementen zugeordnet wird, dass also beispielsweise ein Material aus zwei Klassen besteht.

Aus diesem Grund definiert die stilbasierte Architekturprüfung auch für Architekturelemente, die *mehreren* Quelltextelementen zugeordnet sind, wie sich die Schnittstelle des Architekturelements berechnet.

Die Schnittstellen der Ist-Architektur lassen sich – genau wie die Beziehungen – durch Aggregation aus der Quelltextstruktur übernehmen. Durch die Zusatzinformationen ist bekannt, welche Quelltextelemente gemeinsam ein Architekturelement bilden. Der Zusammenhang ist folgendermaßen: Die Schnittstelle eines Architekturelements ergibt sich aus der Menge der Schnittstellen aller zugeordneter Quelltextelemente. Abbildung 72 zeigt ein Beispiel mit einem Architekturelement des Typs Material, das zwei Quelltextelementen zugeordnet ist. Dieses Beispiel ist angelehnt an ein reales System des WAM-Stils, wurde jedoch um der Übersichtlichkeit willen vereinfacht. Bei den zwei Quelltextelementen handelt es sich um die Klassen Behandlung und Behandlungsabschnitt.

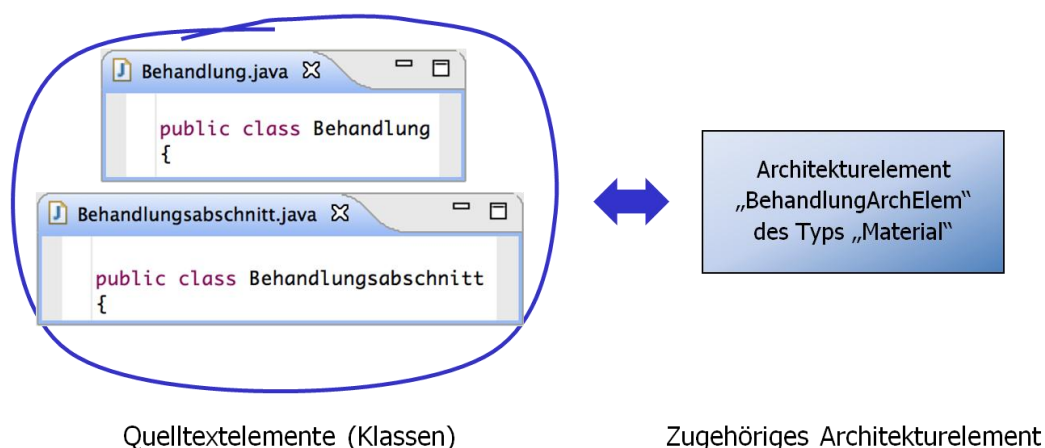


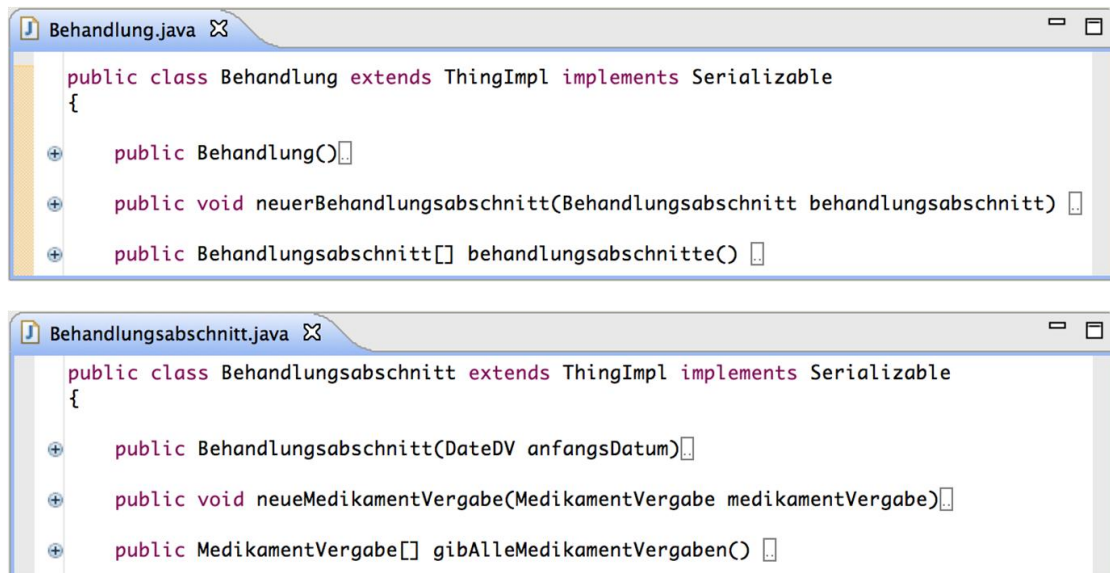
Abbildung 72: Ein Architekturelement bestehend aus zwei Quelltextelementen

Abbildung 73 zeigt Ausschnitte des Quelltextes beider Klassen, hieraus lassen sich die Schnittstellen entnehmen. Die Klasse Behandlung hat folgende Schnittstelle:

- Konstruktor: `public Behandlung()`
- Operation: `public void neuerBehandlungsabschnitt (Behandlungsabschnitt behandlungsabschnitt)`
- Operation: `public Behandlungsabschnitt [] behandlungsabschnitte()`

Die Klasse Behandlungsabschnitt hat diese Schnittstelle:

- Konstruktor: `public Behandlungsabschnitt (DateDV)`
- Operation: `public void neueMedikamentVergabe(MedikamentVergabe)`
- Operation: `public MedikamentVergabe[] gibAlleMedikamentVergaben()`



```
Behandlung.java
public class Behandlung extends ThingImpl implements Serializable
{
    public Behandlung()
    public void neuerBehandlungsabschnitt(Behandlungsabschnitt behandlungsabschnitt)
    public Behandlungsabschnitt[] behandlungsabschnitte()

Behandlungsabschnitt.java
public class Behandlungsabschnitt extends ThingImpl implements Serializable
{
    public Behandlungsabschnitt(DateDV anfangsDatum)
    public void neueMedikamentVergabe(MedikamentVergabe medikamentVergabe)
    public MedikamentVergabe[] gibAlleMedikamentVergaben()
```

Abbildung 73: Quelltext der Klassen Behandlung und Behandlungsabschnitt

Das Architekturelement BehandlungArchElem in diesem Beispiel hat eine Schnittstelle bestehend aus einer Menge mit zwei Elementen. Die Menge enthält die Schnittstelle der Klasse Behandlung und die Schnittstelle der Klasse Behandlungsabschnitt.

Abbildung 74 verdeutlicht diesen Zusammenhang grafisch. Im oberen Bereich zeigt sie das Architekturelement BehandlungArchElem, mit den zwei zugehörigen Klassen. Die Klassen sind innerhalb des Architekturelements dargestellt. Über Pfeile ist verdeutlicht, welche Quelltext-schnittstellen zu den jeweiligen Klassen gehören. Im unteren Bereich zeigt die Abbildung die Schnittstellen der Quelltextelemente und des Architekturelements. Die Schnittstelle des Architekturelements umfasst beide Quelltextschnittstellen.

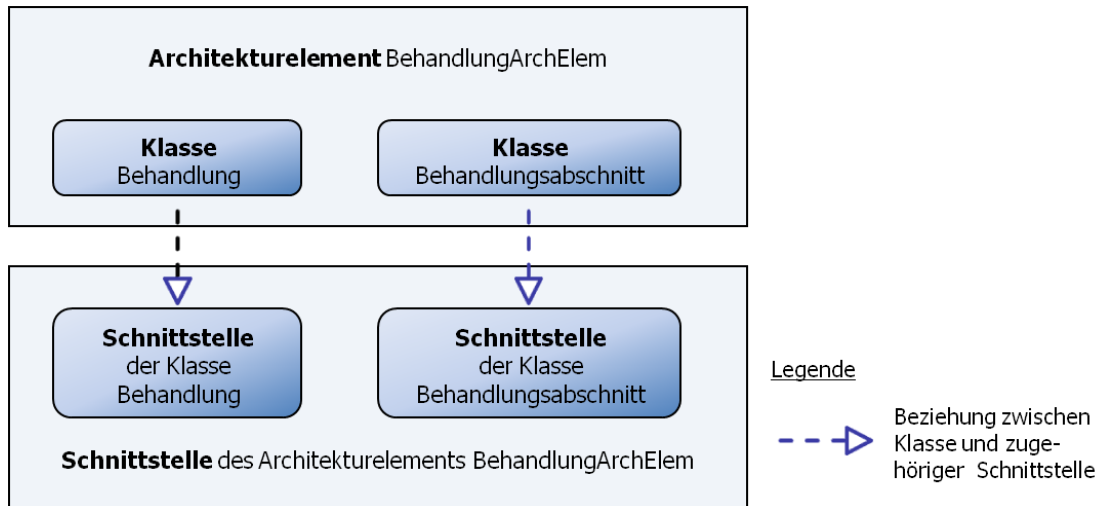


Abbildung 74: Schnittstellen berechnen (Beispiel): die Schnittstelle des Architekturelements umfasst beide Klassenschnittstellen

Formal dargestellt benötigt die stilbasierte Architekturprüfung folgende Mengen und Relationen, um die Schnittstellen der Ist-Architektur zu berechnen:

- die Zuordnung Z_{QA} zwischen Quelltext- und Architekturelementen,
- die Menge S_Q der Quelltextschnittstellen und
- die Zuordnung Z_{QS} dieser Schnittstellen zu den Quelltextelementen.

Die erstgenannte Relation Z_{QA} ist Teil der Zusatzinformationen. Alle weiteren Mengen und Relationen sind Teil der Quelltextstruktur. Diese Mengen und Relationen lassen sich anhand der Abbildung 74 erläutern. Dort gilt:

- Die Zuordnung Z_{QA} gibt an, dass das Architekturelement *BehandlungArchElem* zu zwei Quelltextelementen gehört: die Klassen *Behandlung* und *Behandlungsabschnitt*:

$$\begin{aligned}
 &(\textit{Behandlung}, \textit{BehandlungArchElem}) \in Z_{QA} \\
 &\quad \wedge \\
 &(\textit{Behandlungsabschnitt}, \textit{BehandlungArchElem}) \in Z_{QA}
 \end{aligned}$$

- Die Menge S_Q der Quelltextschnittstellen enthält die oben erläuterten Schnittstellen der beiden Klassen. Die Schnittstelle der Klasse *Behandlung* soll kurz als Schnittstelle 1 bezeichnet werden, die Schnittstelle der Klasse *Behandlungsabschnitt* kurz als Schnittstelle 2:

$$\begin{aligned}
 &\textit{Schnittstelle 1} \in S_Q \\
 &\quad \wedge \\
 &\textit{Schnittstelle 2} \in S_Q
 \end{aligned}$$

- Die Zuordnung Z_{QS} legt fest, dass Schnittstelle 1 zur Klasse Behandlung gehört und Schnittstelle 2 zur Klasse Behandlungsabschnitt:

$$\begin{aligned} & (\text{Behandlung}, \text{Schnittstelle 1}) \in Z_{QS} \\ & \quad \wedge \\ & (\text{Behandlungsabschnitt}, \text{Schnittstelle 2}) \in Z_{QS} \end{aligned}$$

Es gilt für die zu berechnenden Schnittstellen von Architekturelementen: die Schnittstellen sind Mengen, welche Quelltextschnittstellen enthalten. In obigem Beispiel gilt für die Schnittstelle des Architekturelements `BehandlungArchElem`:

- Schnittstelle von `BehandlungArchElem` = {Schnittstelle 1, Schnittstelle 2}.

Ziel der Berechnung ist es, für jedes Architekturelement die zugeordnete Schnittstelle zu berechnen. Formal müssen dafür zwei Mengen berechnet werden:

- Die Menge S_A enthält die Schnittstellen aller Architekturelemente.
- Die Menge Z_{AS} stellt die Zuordnung zwischen den Architekturelementen und ihren Schnittstellen her.

Da jede einzelne Architekturelement-Schnittstelle eine Menge von Quelltextschnittstellen ist, gilt für die Menge S_A , dass die Elemente der Menge jeweils aus einer Teilmenge von S_Q bestehen: $\forall sa \in S_A: sa \subseteq S_Q$. Die Zuordnung Z_{AS} enthält Tupel mit jeweils einem Architekturelement und dessen Schnittstelle. Formal ausgedrückt: $Z_{AS} \subseteq A \times S_A$. Die Menge der Architekturschnittstellen S_A berechnet sich folgendermaßen:

$$\forall sa \in S_A: sq \in sa \Leftrightarrow \exists (q, a) \in Z_{QA} \wedge (q, sq) \in Z_{QS}$$

Das heißt: für die Schnittstelle sa eines jeden Architekturelements a gilt: wenn a dem Quelltextelement q zugeordnet ist, dann wird die Schnittstelle von q (hier als sq bezeichnet) Teil der Schnittstelle von a . Somit gilt in diesem Fall: $sq \in sa$.

Die Zuordnung Z_{AS} enthält für jedes Architekturelement jeweils ein Tupel, bestehend aus dem Element und seiner Schnittstelle:

$$Z_{AS} = \{(a, sa) | a \in A \wedge sa \in S_A \wedge \forall sq \in sa: \exists q \in Q: (q, a) \in Z_{QA} \wedge (q, sq) \in Z_{QS}\}$$

Das bedeutet: Z_{AS} enthält Tupel, deren erstes Element aus der Menge A aller Architekturelemente stammt und deren zweites Element aus der Menge S_A aller Architekturelement-Schnittstellen stammt. Das zweite Element, sa , ist selber eine Menge, bestehend aus Quelltextschnittstellen. Für jede Quelltextschnittstelle sq in sa gilt, dass es ein Quelltextelement q gegeben muss, welches dem Architekturelement a zugeordnet ist und die Quelltextschnittstelle sq besitzt. In obigem Beispiel enthält die Menge Z_{AS} das folgende Tupel:

$$(\text{BehandlungArchElem}, \{\text{Schnittstelle 1}, \text{Schnittstelle 2}\})$$

Dieses Tupel ist enthalten, da es, wie oben erläutert, die zwei Quelltextelemente `Behandlung` und `Behandlungsabschnitt` gibt. Diese Quelltextelemente sind beide dem Architekturelement `BehandlungArchElem` zugeordnet und sie haben die Quelltextschnittstellen `Schnittstelle1` und `Schnittstelle2`.

Erweiterung des Beschreibungsformats für Architekturstile

Schnittstellenregeln können sehr unterschiedliche Aussagen über Schnittstellen machen. Ein Beispiel aus dem WAM-Stil wurde bereits zu Beginn dieses Abschnitts über Schnittstellenregeln eingeführt: „Funktionskomponenten dürfen keine Materialien zurückgeben“. Ein weiteres Beispiel aus dem WAM-Stil betrifft die Schnittstellen von Materialien. Materialien sollen eine fachliche Schnittstelle anbieten mit fachlichen Umgangsformen (siehe Abschnitt 3.3.1) (Züllig-hoven 2005). Materialien, die nur aus einer Menge von Datenfeldern bestehen, verstoßen gegen diese Vorgabe. Die zugehörige Schnittstellenregel lautet: „Materialien dürfen nicht nur Setter- und Getter-Operationen definieren“.

Die oben verwendeten Beispiele zeigen Verstöße gegen diese Regeln. Das Architekturelement `SingleDeviceEditorFP-ArchElem` ist beispielsweise genau einer Klasse zugeordnet, der Klasse `SingleDeviceEditorFP` (siehe Abbildung 75). Das Architekturelement hat den Typ Funktionskomponente.

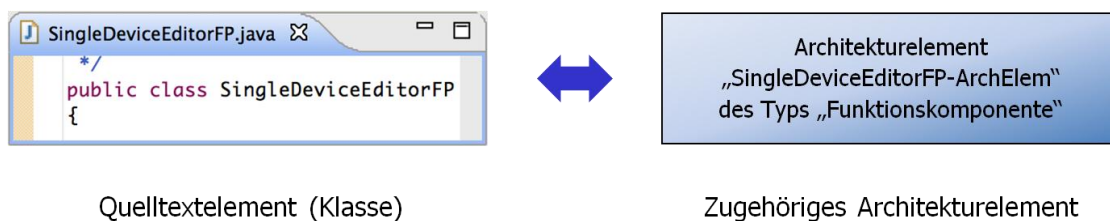


Abbildung 75: Beispiel: Zuordnung zwischen Quelltext und Architekturelement

Da dem Architekturelement in diesem Fall nur ein Quelltextelement zugeordnet ist, hat das Architekturelement dieselbe Schnittstelle wie das Quelltextelement. Die Schnittstelle in diesem Beispiel enthält u.a. die Operation: `public Device getDevice()` (siehe Abbildung 76).

```
SingleDeviceEditorFP.java
```

```
import javax.swing.JOptionPane;  
  
/**  
 * The functional part of the  
 * single device editor tool.  
 */  
public class SingleDeviceEditorFP  
    extends ToolFunctionalityImpl  
{  
    /**  
     * The material worked on by this fp.  
     */  
    private Device _device;  
  
    /**  
     * Returns the material of this fp.  
     */  
    public Device getDevice()  
    {  
        return _device;  
    }  
}
```

Abbildung 76: Quelltextausschnitt der Klasse `SingleDeviceEditorFP`

Der Rückgabetyt dieser Operation ist Device. Die Klasse Device gehört zu einem Architekturelement des Typs Material (siehe Abbildung 77). Somit liegt hier ein Verstoß vor gegen die Regel „Funktionskomponenten dürfen keine Materialien zurückgeben“.

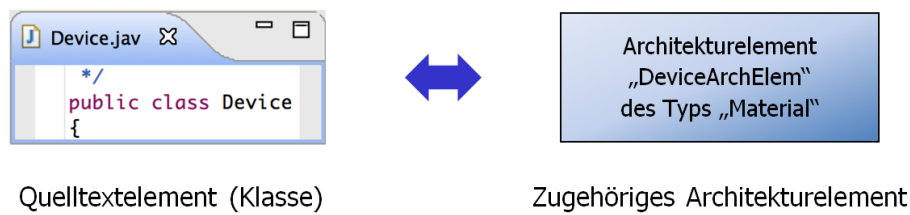


Abbildung 77: Zuordnung zwischen Quelltext und Architektur für die Klasse Device

Das Architekturelement DeviceArchElem hat den Typ Material. Das Architekturelement ist nur einem Quelltextelement zugeordnet, der Klasse Device. Somit gilt auch hier: Quelltextelement und Architekturelement haben dieselbe Schnittstelle. Die Schnittstelle lautet in diesem Fall:

- Konstruktor: public Device()
- Operation: public void setDescription (String)
- Operation: public String getDescription()
- Operation: public void setName(DeviceNameDV)
- Operation: public DeviceNameDV getName()
- Operation: public void setPurchaseDate(DateDV)
- Operation: public DateDV getPurchaseDate ()

Da alle Operationen in diesem Beispiel entweder Setter oder Getter sind, verstößt diese Schnittstelle gegen die Regel: „Materialien dürfen nicht nur Setter- und Getter-Operationen definieren“.

Die in den Beispielen genannten zwei WAM-Regeln sollen im Folgenden kurz als nichtNurSetterUndGetter und als keineMaterialienZurückgeben bezeichnet werden.

Wenn ein Entwicklungsteam seinen Stil beschreibt, muss es angeben, für welche Architekturelement-Typen welche Regeln gelten. Beschreibt ein Team den WAM-Stil müsste es somit unter anderem angeben:

- Für Materialien gilt die Regel nichtNurSetterUndGetter
- Für Funktionskomponenten gilt die Regel keineMaterialienZurückgeben

Formal bedeutet dies, die Beschreibung des Architekturstils wird in dieser Erweiterungsstufe ergänzt um eine Zuordnung der Architekturelement-Typen zu Schnittstellenregeln:

$$Z_{TSR} \subseteq T_A \times R_S$$

Dabei bezeichnet R_S die Menge der Schnittstellenregeln und T_A die im Stil definierten Architekturelement-Typen (siehe Definition 3-2).

In den Beispielen beinhaltet die Architekturstil-Beschreibung die folgenden zwei Tupel:

- $(Material, nichtNurSetterUndGetter) \in Z_{TSR}$
- $(Funktionskomponente, keineMaterialienZurückgeben) \in Z_{TSR}$

Somit ist festgelegt, welche Regeln für welchen Architekturelement-Typ gelten. Noch nicht festgelegt ist allerdings, was die einzelnen Regeln für die Prüfung bedeuten, welche Semantik sie besitzen. Bis zu dieser Stelle sind die Regeln in der Formalisierung lediglich Bezeichnungen.

Wie lässt sich die Semantik der Regeln formalisieren? Anhand der Beispiele zeigt sich, wie unterschiedlich Schnittstellenregeln sind. Die Regeln betreffen verschiedenste Aspekte der Schnittstellen, so dass sich keine so überschaubare Kategorisierung bilden lässt wie bei Beziehungsregeln. Da Schnittstellenregeln ein relativ neues Konzept sind, existieren zudem nicht genügend Beispiele, um eine ausreichend fundierte, formale Abstraktion über Schnittstellenregeln zu ermöglichen.

Als pragmatischer, umsetzbarer Ansatz bewährte sich in den praktischen Untersuchungen die Betrachtung der Schnittstellenregeln als Funktion der Schnittstelle von Architekturelementen auf einen Wahrheitswert.

Schnittstellenregeln stellen sich somit formal folgendermaßen dar, dabei bezeichnet S_A die Menge der Schnittstellen:

- Schnittstellenregeln sind Funktionen der Form $S_A \rightarrow \{true, false\}$.

Diese Funktion wird individuell programmiert. Werkzeuge, die die stilbasierte Architekturprüfung unterstützen, müssen also eine Möglichkeit für die Programmierung von Schnittstellenregeln zur Verfügung stellen. Schnittstellenregeln werden realisiert als programmiersprachliche Operationen, sie erhalten als Parameter die Schnittstelle eines Architekturelements, haben Zugriff auf das Modell der Ist-Architektur und liefern als Ergebnis einen Wahrheitswert, der besagt, ob die Regel durch das gegebene Element eingehalten wird.

Auf diese Weise erlaubt der Ansatz, alle Schnittstellenregeln zu prüfen, die sich auf die Bestandteile der Ist-Architektur beziehen. Die Ist-Architektur umfasst Signaturen und eventuelle Rückgabetypen von Konstruktoren und Methoden. Nicht prüfen lassen sich Regeln, welche sich auf die Semantik beziehen.

Beispielsweise muss für die Implementierung der genannten Regel `nichtNurSetterUndGetter` ermittelt werden, ob eine Methode ein Setter oder Getter ist. Diese Prüfung erfolgt – wie im Rahmen statischer Architekturprüfungen üblich – anhand der Struktur des Quelltextes. Im Rahmen der beispielhaften Umsetzung mit Java wird eine Methode als Setter betrachtet, wenn sie als Rückgabetypp `void` definiert und sie nach der in Java üblichen Konvention „set*“ benannt ist. Getter haben einen anderen Rückgabetypp als `void`, keine Parameter und sind „get*“ benannt. In den untersuchten Systemen waren diese Konventionen konsequent umgesetzt und ließen sich gut zur Identifikation von Gettern und Settern nutzen (siehe Karstens 2005).

Die Implementierung der zweiten genannten Regel `keineMaterialienZurückgeben` prüft den Rückgabetyt aller Methoden. Dieser Rückgabetyt darf nicht zu einer Klasse gehören, die zu einem Architekturelement des Typs `Material` gehört.

Weitere Details zur Implementierung von Schnittstellenregeln zeigt die weiter unten vorgestellte beispielhafte Umsetzung mit dem `StyleBasedChecker`.

Erweiterung des Berechnungsschritts (c): Die Ist-Architektur auf Stiltreue prüfen

Dieser Berechnungsschritt überprüft die Schnittstellen der Ist-Architektur auf ihre Stiltreue. Dazu ermittelt der Berechnungsschritt eventuelle Verstöße der Schnittstellen gegen die im Stil festgelegten Schnittstellenregeln.

Diese Prüfung benötigt die Architekturelemente, deren Typen und Schnittstellen, die Schnittstellenregeln und die Angabe, welche Schnittstellenregeln für welche Architekturelement-Typen gelten. Formal bedeutet dies, dass folgende Eingabewerte benötigt werden:

- Benötigte Anteile der Ist-Architektur:
 - $Z_{AS} \subseteq A \times S_A$: die Zuordnung zwischen Architekturelementen und deren Schnittstellen.
 - Z_{AT} : die Zuordnung zwischen Architekturelementen und deren Typen. Die Zuordnung ist eine Relation, für die gilt: $Z_{AT} \subseteq A \times T_A$
- Benötigte Anteile der Stilbeschreibung:
 - Z_{TSR} : Die Zuordnung der Architekturelement-Typen zu den Schnittstellenregeln. Es gilt $Z_{TSR} \subseteq T_T \times R_S$. Dabei bezeichnet T_A die Menge der Architekturelement-Typen, R_S bezeichnet die Menge der Schnittstellenregeln.

Die Prüfung berechnet im ersten Schritt ein Zwischenergebnis, die Hilfsrelation H_{SSR} . Dieses Zwischenergebnis besagt, für welche Schnittstelle jeweils welche Schnittstellenregel auszuführen ist. Das Zwischenergebnis ordnet die Schnittstellen der Ist-Architektur den Schnittstellenregeln zu. Es gilt somit $H_{SSR} \subseteq S_A \times R_S$.

Formal berechnet sich die Relation wie folgt:

$$H_{SSR} = \{(s, sr) | \exists t, a: (t, sr) \in Z_{TRS} \wedge (a, t) \in Z_{AT} \wedge (a, s) \in Z_{AS}\}$$

Für den Architekturelement-Typ Funktionskomponente gilt in obigem Beispiel die Schnittstellenregel `keineMaterialienZurückgeben`. Somit gilt für $t = \text{Funktionskomponente}$ und $sr = \text{keineMaterialienZurückgeben}$: $(t, sr) \in Z_{TSR}$.

Das im obigen Beispiel genannte Architekturelement `SingelDeviceEditorFP-ArchElem` hat den Typ Funktionskomponente. Das bedeutet für $a = \text{SingelDeviceEditorFP-ArchElem}$ gilt folgendes: $(a, t) \in Z_{AT}$.

Die oben erläuterte Schnittstelle s des Architekturelements `SingelDeviceEditorFP-ArchElem` wird ermittelt über $(\text{SingelDeviceEditorFP-ArchElem}, s) \in Z_{AS}$.

Es ergibt sich folglich:

$$(s, \text{keineMaterialienZurückgeben}) \in H_{SSR}$$

Somit wird die Regel keineMaterialienZurückgeben für die Schnittstelle des Architekturlements `SingelDeviceEditorFP-ArchElem` ausgeführt.

Unabhängig davon, wie die Schnittstellenregeln technisch implementiert werden, gilt: Verstöße betreffen immer genau ein Architekturelement. Ergebnis der Prüfung ist somit für jede Schnittstellenregel eine Menge von Architekturelementen, die gegen die jeweilige Regel verstoßen. Für jede Schnittstellenregel ergibt sich eine Menge an Verstößen:

$$V_{SR} \subseteq A$$

Erweiterung des Berechnungsschritts (d): Verstöße in den Quelltext verfolgen

In diesem Berechnungsschritt gilt es, für die gefundenen Verstöße die zugehörigen Quelltextstellen zu ermitteln. Dafür müssen für die betroffenen Architekturelemente die zugeordneten Quelltextelemente ermittelt werden. Die Quelltextelemente sind über die Zuordnung Z_{QA} ermittelbar. Diese Zuordnung enthält für obiges Beispiel die folgenden zwei Tupel:

- (Device, DeviceArchElem)
- (SingleDeviceEditorFP, SingelDeviceEditorFP-ArchElem)

Die Menge der Verstöße gegen die Schnittstellenregel „Materialien dürfen nicht nur Setter- und Getter-Operationen definieren“ lautet:

- $V_{SR} = \{DeviceArchElem\}$

Die Menge der betroffenen Quelltextstellen enthält Quelltextelemente: $V_{QSR} \subseteq Q$. Diese Menge berechnet sich folgendermaßen:

- $V_{QSR} = \{q | \exists a: (q, a) \in Z_{QA} \wedge a \in V_{SR}\}$

In dem gegebenen Beispiel bedeutet dies: alle Quelltextelemente, die dem Architekturelement `DeviceArchElem` zugeordnet sind, bilden die betroffenen Quelltextstellen. In diesem Fall ist dies die Klasse `Device`. Diese müsste das Entwicklungsteam ändern, um den Verstoß zu beseitigen.

6.2.2 Abgrenzung zu bisherigen Ansätzen

Abschnitt 2.2 hat dargestellt, dass im Forschungsbereich der Softwarearchitektur anfangs vor allem Architekturelemente und Beziehungen thematisiert wurden. Die Schnittstellen der Architekturelemente spielen erst in neueren Definitionen eine Rolle, so beispielsweise bei Bass et al. (Bass et al. 2012); dort umfasst die Definition von Softwarearchitektur neben den Architekturelementen und den Beziehungen auch die Eigenschaften der Elemente (siehe Abschnitt 2.2).

Analog zu dieser Entwicklung zielen die meisten Architekturprüfungen darauf, Vorgaben für Beziehungen zwischen Architekturelementen einzuhalten (siehe Abschnitt 4.1). So legen Soll-Architekturen fest, welche Architekturelemente eine Beziehung haben sollen und welche in keiner Beziehung stehen dürfen. Schnittstellen hingegen werden deutlich seltener thematisiert. Erst neuere Ansätze und Werkzeuge prüfen Softwaresysteme darauf, ob sie Vorgaben bezüglich der Schnittstellen auf Ebene der Soll-Architektur einhalten, so der LISA-Ansatz, die Sotograph-Familie, SonarJ und Structure 101 (siehe Abschnitt 4.4). Mit diesen Werkzeugen lässt sich festlegen, dass ausgewählte Quelltextelemente innerhalb eines Architekturelements zur Schnittstelle

gehören. Nur diese Quelltextelemente dürfen von außerhalb des Architekturelements genutzt werden. Alle anderen Quelltextelemente sind privat. Abbildung 78 zeigt ein Beispiel. Einige Quelltextelemente sind Teil der Schnittstelle ihres Architekturelements. Andere Architekturelemente dürfen nur auf diese Quelltextelemente zugreifen. In der Grafik ist eine Beziehung vorhanden, die nicht erlaubt ist, da sie die Schnittstelle nicht beachtet: das Quelltextelement 1 greift auf das Quelltextelement 3 zu. Dieser Zugriff ist nicht erlaubt, da das Quelltextelement 3 nicht Teil der Schnittstelle seines Architekturelements ist und die beiden Quelltextelemente zu unterschiedlichen Architekturelementen gehören.

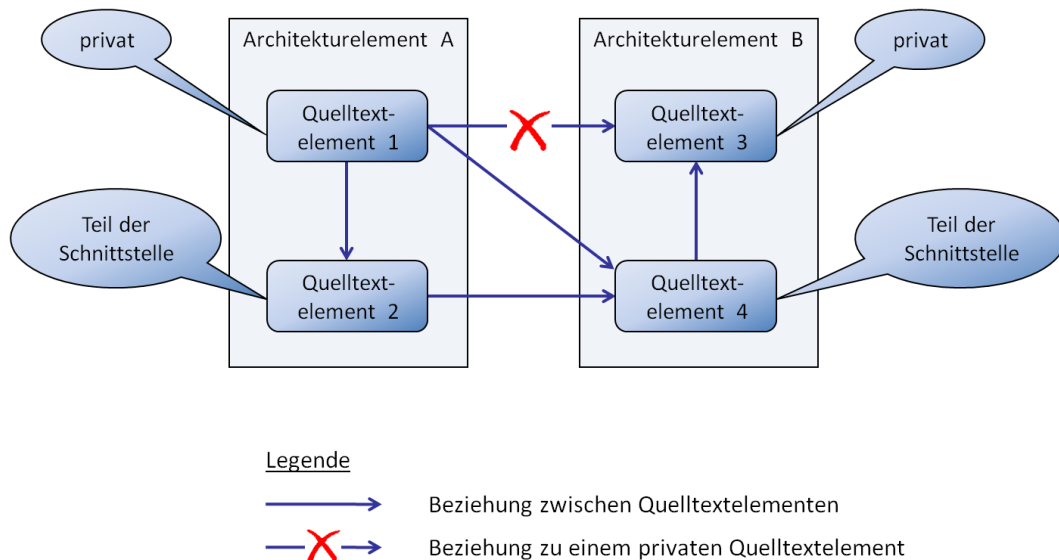


Abbildung 78: Architekturelemente mit Schnittstelle.
Alle öffentlichen Quelltextelemente sind Teil der Schnittstelle

Die Werkzeuge, die solche Vorgaben prüfen, wie in Abbildung 78 dargestellt, betten diese Vorgaben in die Definition von Soll-Architekturen ein. Für jedes einzelne Architekturelement der Soll-Architektur kann die prüfende Person vorgeben, welche der zugeordneten Quelltextelemente Teil der Schnittstelle sind.

Die stilbasierte Architekturprüfung unterscheidet sich deutlich von diesen Prüfansätzen: sie erlaubt *Regeln* für die Gestaltung von Schnittstellen festzulegen und so in einem Schritt einheitliche Vorgaben für *alle* Architekturelemente eines bestimmten Typs zu definieren. Schnittstellenregeln sind ein neuer Beitrag dieser Arbeit.

6.3 Hierarchische Architekturen

Diese Erweiterungsstufe der stilbasierten Architekturprüfung adressiert Stile mit zusammengesetzten Architekturelementen. Solche Stile besitzen zwei weitere Regeltypen: die Enthältregeln und die internen Gebotsregeln.

Zur Erinnerung: Wie in Kapitel 3 erläutert, beschreiben Enthältregeln den hierarchischen Aufbau von Architekturelementen. Sie legen den Typ zusammengesetzter Elemente sowie die Typen der enthaltenen Elemente fest. Dies sei an einem zusammengesetzten Architekturelement-Typs des WAM-Stils betrachtet: MonoWerkzeuge sind aus zwei Architekturelementen aufgebaut, eines hat den Typ GUI, das andere den Typ MonoTool. Abbildung 79 visualisiert diese hierarchische Struktur.



Abbildung 79: Beispiel eines hierarchischen Architekturelement-Typs

Für Softwaresysteme, die Architekturelemente des Typs MonoWerkzeug beinhalten, gilt: Jedes einzelne dieser Werkzeuge soll aus einer GUI und aus einem MonoTool bestehen. Die zugehörigen Enthältregeln lauten: „MonoWerkzeuge enthalten eine GUI“ und „MonoWerkzeuge enthalten ein MonoTool“.

Formal werden Enthältregeln laut Definition 3-2 mittels folgender Relation beschrieben:

- R_E : Die Relation der Enthält-Beziehungsregeln. Für die Relation gilt: $R_E \subseteq T_A \times T_A$

T_A ist die Menge der Architekturelement-Typen. Ein Tupel $(a, b) \in R_E$ besagt, dass Architekturelemente des Typs a ein Architekturelement des Typs b enthalten müssen. Die im Beispiel genannte Regel lautet formal:

$$(MonoWerkzeug, GUI) \in R_E \wedge (MonoWerkzeug, MonoTool) \in R_E$$

Abbildung 80 zeigt ein Beispiel für ein Architekturelement des Typs MonoWerkzeug. Die Beschriftung zeigt die Namen der Architekturelemente, gefolgt von dem in Klammern angegebenen Architekturelement-Typ. Das dargestellte MonoWerkzeug heißt ReisebeihilfeBearbeiter, es ist aus zwei weiteren Architekturelementen aufgebaut: der ReisebeihilfeBearbeiterGUI des Typs GUI und dem ReisebeihilfeBearbeiterTool des Typs MonoTool. Es befolgt somit die genannten Enthältregeln für MonoWerkzeuge.

ReisebeihilfeBearbeiter (MonoWerkzeug)

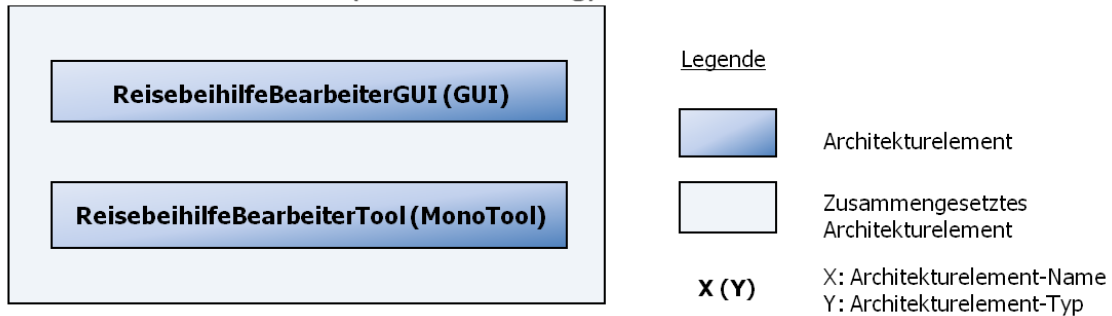


Abbildung 80: Ein zusammengesetztes Architekturelement

Die internen Gebotsregeln machen Aussagen über Beziehungen innerhalb eines zusammengesetzten Architekturelements. Sie legen fest, dass ein enthaltenes Element ein anderes enthaltenes Element kennen muss. Für das Beispiel der MonoWerkzeuge gilt: innerhalb eines MonoWerkzeugs kennt das MonoTool die GUI. Abbildung 81 visualisiert diese interne Gebotsregel.

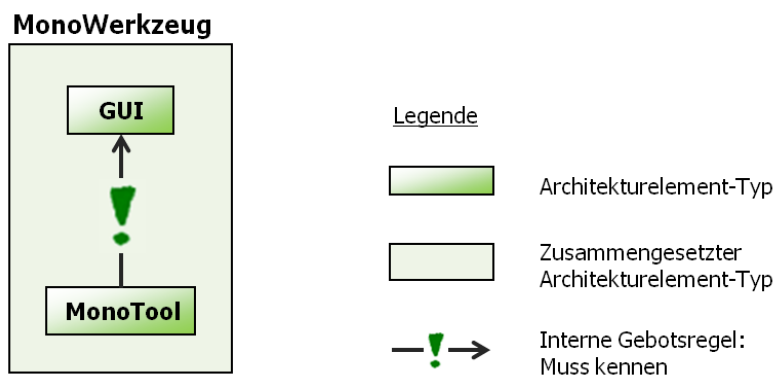


Abbildung 81: Beispiel einer internen Gebotsregel

Formal werden interne Gebotsregeln beschrieben über die Relation $R_{IG} \subseteq T_A \times T_A \times T_A$, dabei bezeichnet T_A die Menge der Architekturelement-Typen. Um formal festzulegen, dass innerhalb von MonoWerkzeugen jeweils das MonoTool die GUI kennen muss, gilt: $(MonoWerkzeug, MonoTool, GUI) \in R_{IG}$. Allgemein bedeutet $(t1, t2, t3) \in R_{IG}$ folgendes: „Innerhalb zusammengesetzter Architekturelemente des Typs $t1$ gilt: jedes enthaltene Architekturelement des Typs $t2$ muss mindestens ein enthaltenes Architekturelement des Typs $t3$ kennen.“

Die folgenden Abschnitte erläutern die formale Definition von Enthältregeln und internen Gebotsregeln. Ferner wird erläutert, wie sich die bisher vorgestellten Regeltypen bei zusammengesetzten Architekturelementen verhalten.

6.3.1 Enthältregeln: Berechnungsschritte und Eingabeinformationen

Um Enthältregeln prüfen zu können, erweitert diese Ausbaustufe die Berechnungsschritte (b) „die stilbasierte Ist-Architektur berechnen“, (c) „die Ist-Architektur auf Stiltreue prüfen“ und (d) „Verstöße in den Quelltext verfolgen“ (siehe Abbildung 64, Seite 119). Für den Berechnungsschritt (b) werden die Zusatzinformationen ergänzt. Das Beschreibungsformat von Architektur-stilen wird im Vergleich zum Kernkonzept der stilbasierte Architekturprüfung um die genannte Enthältbeziehung R_E ergänzt. Der Berechnungsschnitt (a) „die Quelltextstruktur ermitteln“ und das Format der Quelltextstruktur bleiben unverändert.

Erweiterung des Berechnungsschritts (b): die stilbasierte Ist-Architektur ermitteln

Die Definition 3-4 für stilbasierte Architekturen wird für diese Ausbaustufe um die Enthältbeziehung zwischen Architekturelementen ergänzt:

- B_E : Die Menge der direkten Enthältbeziehungen zwischen Architekturelementen. Dabei gilt: $B_E \subseteq A \times A$

Für jedes Tupel von zwei Architekturelementen $(a, b) \in B_E$ gilt: das Architekturelement a enthält das Architekturelement b . Wenn beispielsweise gilt $(a, b) \in B_E \wedge (b, c) \in B_E$, dann bedeutet dies: das Architekturelement a enthält ein Architekturelement b und dieses enthält ein Architekturelement c . Abbildung 82 zeigt das genannte Beispiel eines zweifach geschachtelten Architekturelements. Eine Architektur kann aus beliebig vielen geschachtelten und nicht geschachtelten Architekturelementen bestehen. Ein Architekturelement, das mindestens ein anderes enthält, wird als zusammengesetztes Architekturelement bezeichnet. Ein Architekturelement, das in einem anderen Architekturelement enthalten ist, wird als geschachteltes Architekturelement bezeichnet.

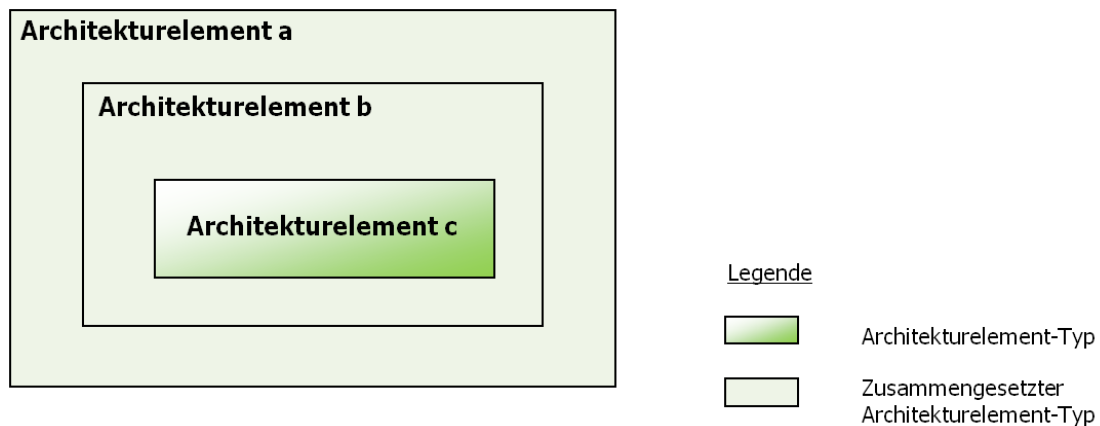


Abbildung 82: Hierarchisch geschachtelte Architekturelemente

Die Relation B_E hat folgende Eigenschaften:

- Die Relation B_E muss zyklensfrei sein, das heißt, Architekturelemente können sich selbst nicht enthalten, weder direkt noch indirekt. Somit gilt

$$(a, b) \in B_E^* \Rightarrow (b, a) \notin B_E^*$$

dabei bezeichnet B_E^* die reflexive und transitive Hülle von B_E .

- Die Tupel der Relation B_E beschreiben ausschließlich die direkte Enthältbeziehung. In dem genannten Beispiel ist c nur indirekt in a enthalten, somit gilt hier: $(a, c) \notin B_E$.

Die Enthältbeziehung zwischen Architekturelementen lässt sich mit den üblicherweise in objektorientierten Programmiersprachen vorhandenen Konzepten nicht ausdrücken. Entwicklungsteams müssen die Enthältbeziehung über Zusatzinformationen (beispielsweise mit Hilfe von Quelltextannotationen) beschreiben.

Erweiterung des Berechnungsschritts (c): Die Ist-Architektur auf Stiltreue prüfen

Dieser Berechnungsschritt benötigt folgende Mengen und Relationen:

- $B_E \subseteq A \times A$: Die zur Ist-Architektur gehörende Menge der Enthältbeziehungen zwischen Architekturelementen
- $Z_{AT} \subseteq A \times T_A$: Die zur Ist-Architektur gehörende Zuordnung zwischen Architekturelementen und deren Typen.
- $R_E \subseteq T_A \times T_A$: Die Enthält-Beziehungsregeln des Stils.

Eine Architektur verstößt gegen eine Enthält-Beziehungsregel, wenn einem zusammengesetzten Architekturelement ein vorgeschriebenes enthaltenes Architekturelement fehlt. Formal bedeutet dies, ein zusammengesetztes Architekturelement a verstößt gegen eine Beziehungsregel $(t1, t2) \in R_E$ wenn a den Architekturelement-Typ $t1$ besitzt, aber in a kein Architekturelement des Typs $t2$ enthalten ist.

Die Menge der Verstöße berechnet sich mittels einer Hilfsrelation $H_{ET} \subseteq A \times T_A$. Der Index ET steht für enthaltene Typen. Diese Hilfsrelation bildet für jedes zusammengesetzte Architekturelement ab, welche Typen bei den enthaltenen Architekturelementen vorkommen. H_{ET} beinhaltet Tupel, die jeweils aus einem Architekturelement und einem Architekturelement-Typ bestehen. Die erste Komponente der Tupel ist jeweils ein zusammengesetztes Architekturelement, die zweite Komponente ist der Typ eines darin enthaltenen Architekturelements. Für jeden enthaltenen Typ enthält die Hilfsrelation je ein Tupel. Wenn beispielsweise ein zusammengesetztes Architekturelement A drei geschachtelte Architekturelemente B , C und D enthält, wobei B den Typ $t1$ hat und C und D beide den Typ $t2$ haben, dann beinhaltet die Hilfsrelation die Tupel $(A, t1)$ und $(A, t2)$.

Als Beispiel für ein fehlerhaftes Architekturelement sei der in Abbildung 83 dargestellte Reiseplaner des Typs MonoWerkzeug betrachtet.

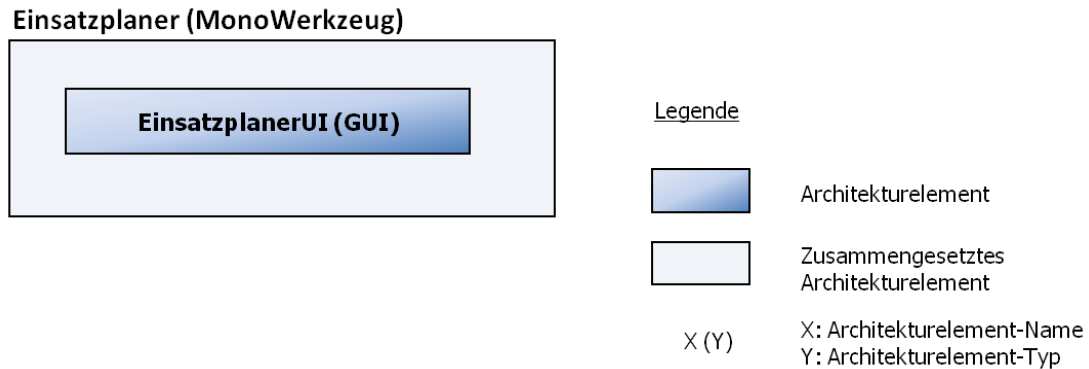


Abbildung 83: Ein zusammengesetztes Architekturelement

Die Hilfsrelation H_{ET} enthält für das Architekturelement Einsatzplaner genau ein Tupel:

$$(Einsatzplaner, GUI) \in H_{ET}$$

Formal ist die Hilfsrelation folgendermaßen definiert:

$$H_{ET} = \{(a, t) | \exists b: (a, b) \in B_E \wedge (b, t) \in Z_{AT}\}$$

Die Menge V_{ER} aller Architekturelemente, die gegen eine Enthält-Beziehungsregel $(t1, t2) \in R_E$ verstoßen, berechnet sich mittels der Hilfsrelation wie folgt:

$$V_{ER} = \{a | \exists t1, t2 : (a, t1) \in Z_{AT} \wedge (t1, t2) \in R_E \wedge (a, t2) \notin H_{ET}\}$$

Das Architekturelement Einsatzplaner verstößt gegen die Regel, dass MonoWerkzeuge ein MonoTool enthalten sollen. Diese Regel ist formal dargestellt ein Tupel:

$$(MonoWerkzeug, MonoTool) \in R_E$$

Der Einsatzplaner verstößt gegen diese Regel, denn mit $a = Einsatzplaner$, $t1 = MonoWerkzeug$ und $t2 = MonoTool$ gilt:

$$\begin{aligned} (Einsatzplaner, MonoWerkzeug) &\in Z_{AT} \wedge \\ (MonoWerkzeug, MonoTool) &\in R_E \wedge \\ (Einsatzplaner, MonoTool) &\notin H_{ET} \end{aligned}$$

Erweiterung des Berechnungsschritts (d): Verstöße in den Quelltext verfolgen

Verstöße gegen Enthältregeln entstehen, wenn einem zusammengesetzten Architekturelement ein enthaltenes Element fehlt. Auf Architekturebene sind die Verstöße jeweils den fehlerhaften, zusammengesetzten Architekturelementen zugeordnet.

Um die Verstöße in den Quelltext zu verfolgen, müssen die Quelltextelemente der fehlerhaften Architekturelemente ermittelt werden. Die Zuordnung von Quelltext- und Architekturelementen findet sich in der Relation Z_{QA} . Die Menge der betroffenen Quelltextelemente V_{QER} berechnet sich für ein fehlerhaftes Architekturelement a wie folgt:

$$V_{QER} = \{q | \exists a: (q, a) \in Z_{QA} \wedge a \in V_{ER}\}$$

Es gilt zu beachten, dass einige zusammengesetzte Architekturelemente keine eigenen Quelltextelemente besitzen. In diesem Falle wird der Verstoß den Quelltextelementen der enthaltenen Architekturelemente zugeordnet. Bei mehrfach geschichteten Architekturelementen wird dieser Vorgang so lange wiederholt, bis entweder zugeordnete Quelltextelemente existieren oder enthaltene Elemente erreicht sind, die selber keine weiteren Elemente enthalten (dieser Fall kam in den betrachteten Architekturstilen nicht vor, er wird hier nur der Vollständigkeit halber erwähnt). Die betroffenen Quelltextelemente eines zusammengesetzten Architekturelements a , welches gegen eine Enthält-Beziehungsregel verstößt und keine eigenen Quelltextelemente besitzt, berechnen sich folgendermaßen:

$$V'_{QER} \{q | (a, b) \in B_E \wedge (q, b) \in Z_{QA} \wedge a \in V_{ER}\}$$

6.3.2 Interne Gebotsregeln: Berechnungsschritte und Eingabeinformationen

Um interne Gebotsregeln prüfen zu können, erweitert diese Ausbaustufe die Berechnungsschritte (c) „die Ist-Architektur auf Stiltreue prüfen“ und (d) „Verstöße in den Quelltext verfolgen“ (siehe Abbildung 64, Seite 119). Weiterhin ergänzt diese Ausbaustufe das Beschreibungsformat für Architekturstile, damit sich interne Gebotsregeln ausdrücken lassen. Unverändert bleiben die Zusatzinformationen, das Format der Quelltextstruktur sowie die Berechnungsschritte (a) „die Quelltextstruktur ermitteln“ und (b) „die stilbasierte Ist-Architektur berechnen“.

Erweiterung des Beschreibungsformats für Architekturstile um interne Gebotsregeln

Interne Gebotsregeln legen fest, welche Typen die beiden in Beziehung stehenden Architekturelemente haben sollen, genau wie die im Kernkonzept definierten Gebotsregeln. Darüber hinaus müssen interne Gebotsregeln auch den Typ des zusammengesetzten Architekturelements nennen, innerhalb dessen die Beziehung auftreten soll. Zur Erinnerung zeigt Abbildung 84 die oben genannte interne Gebotsregel „In MonoWerkzeugen gilt: MonoTools müssen eine GUI kennen“.

Interne Gebotsregeln werden formal als eine Menge von Tripeln dargestellt. Es gilt:

- Die Menge der internen Gebotsregeln enthält Tripel mit jeweils drei Architekturelement-Typen: $R_{IG} \subseteq T_A \times T_A \times T_A$.

Der erste Typ bezieht sich auf das zusammengesetzte Architekturelement. Der zweite Typ bezieht sich auf das Element, von dem die Beziehung ausgehen soll. Der dritte Typ bezieht sich auf das Element, zu dem die Beziehung hingehet.

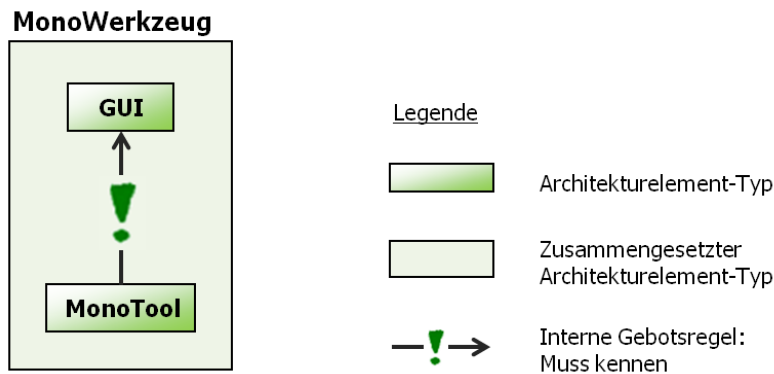


Abbildung 84: Interne Gebotsregel des WAM-Stils

Formal wird die interne Gebotsregel „In MonoWerkzeugen gilt: MonoTools müssen eine GUI kennen“ folgendermaßen ausgedrückt:

$$(MonoWerkzeug, MonoTool, GUI) \in R_{IG}$$

Interne Gebotsregeln beziehen sich immer auf direkt enthaltene Architekturelemente. Dies gilt, auch wenn es im Folgenden nicht immer explizit erwähnt wird.

Erweiterung des Berechnungsschritts (c): Die Ist-Architektur auf Stiltreue prüfen

Um Verstöße gegen eine interne Gebotsregeln ausfindig zu machen, müssen alle Architekturelemente aufgefunden werden, von denen aufgrund dieser Regel eine Beziehung ausgehen soll, aber keine derartige Beziehung vorliegt. In obigem Beispiel bedeutet dies, dass alle MonoTools gefunden werden müssen, die in einem MonoWerkzeug enthalten sind und keine Beziehung zu einer im selben MonoWerkzeug enthaltenen GUI haben.

Für die Berechnung der Verstöße wird die oben definierte Hilfsrelation $H_{ET} \subseteq A \times T_A$ benötigt, die für jedes Architekturelement a angibt, welche Typen bei den direkt enthaltenen Architekturelementen vorkommen.

Ferner wird eine zweite Hilfsrelation benötigt, $H_{ISB} \subseteq A \times A \times T_A$. Der Index ISB steht für interne Soll-Beziehungen. Diese Hilfsrelation besagt, welche internen Beziehungen laut den vorhandenen internen Gebotsregeln vorkommen müssen. Die in H_{ISB} enthaltenen Tupel bestehen jeweils aus einem zusammengesetzten Architekturelement, einem darin enthaltenen Architekturelement, von dem laut einer internen Gebotsregel eine Beziehung ausgehen soll, sowie dem Typ, den das Architekturelement haben soll, zu dem die vorgeschriebene Beziehung verlaufen soll. Die Hilfsrelation berechnet sich anhand der internen Gebotsregeln sowie folgender Relationen:

- der oben definierten Hilfsrelation H_{ET} ,
- der Zuordnung zwischen Architekturelementen zu ihrem Typ Z_{AT} ,
- und der Enthältbeziehung zwischen Architekturelementen B_E

Die Hilfsrelation berechnet sich folgendermaßen:

$$\begin{aligned}
 H_{ISB} = & \\
 & \{(a, b, t3) \mid \exists t1, t2: \\
 & (a, b) \in B_E \wedge (a, t3) \in H_{ET} \\
 & \wedge \\
 & (a, t1) \in Z_{AT} \wedge (b, t2) \in Z_{AT} \\
 & \wedge \\
 & (t1, t2, t3) \in R_{IG}\}
 \end{aligned}$$

Das bedeutet, innerhalb des Architekturelements a muss ein enthaltenes Architekturelement b eine Beziehung haben zu einem weiteren enthaltenen Architekturelement des Typs t3, sofern Folgendes zutrifft: a enthält b und a enthält ein Element des Typs t3, a hat den Typ t1, b hat den Typ t2 und es existiert eine interne Gebotsregel (t1, t2, t3).

Abbildung 85 zeigt eine Architektur mit zwei zusammengesetzten Architekturelementen des Typs MonoWerkzeug. Für diesen Typ gilt die oben genannte Regel $(MonoWerkzeug, MonoTool, GUI) \in R_{IG}$ („In MonoWerkzeugen gilt: MonoTools müssen eine GUI kennen“)

Die Hilfsrelation $H_{ISB} \subseteq A \times A \times T_A$ enthält in diesem Fall zwei Tupel:

$$\begin{aligned}
 H_{ISB} = & \\
 & \{(ReisebeihilfeBearbeiter, ReisebeihilfeBearbeiterTool, GUI), \\
 & (Abrechner, AbrechnerTool, GUI)\}
 \end{aligned}$$

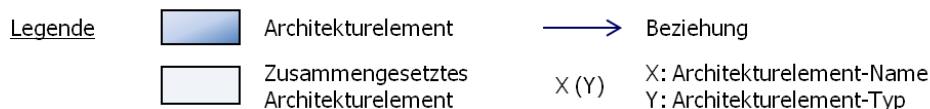
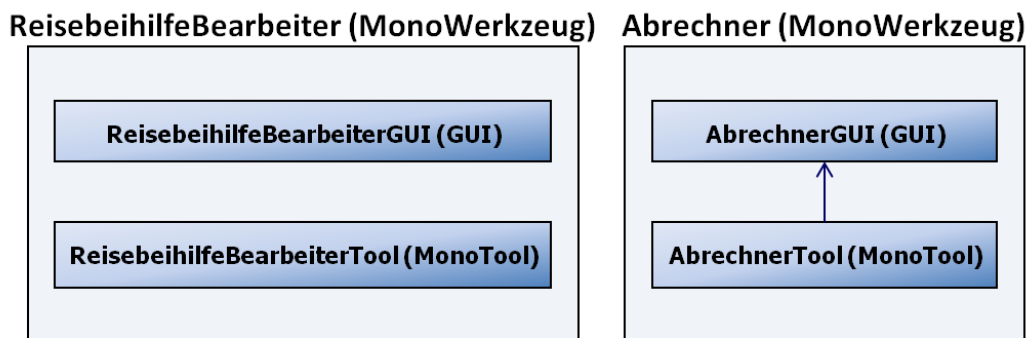


Abbildung 85: Eine beispielhafte Ist-Architektur

Eine weitere Hilfsrelation $H_{IIB} \subseteq A \times A \times T_A$ wird aus den in der Architektur tatsächlich vorhandenen internen Beziehungen berechnet. Der Index IIB steht für interne Ist-Beziehungen. Für

die Berechnung der Hilfsrelation wird die Menge B_A der Beziehungen innerhalb der Ist-Architektur verwendet:

$$H_{IB} = \{(a, b, t) | \exists c : (a, b) \in B_E \wedge (a, c) \in B_E \wedge (c, t) \in Z_{AT} \wedge (b, c) \in B_A\}$$

Ein Tupel $(a, b, t) \in H_{IB}$ besagt, dass a ein zusammengesetztes Architekturelement ist, b in a enthalten ist, und dass ein in a enthaltenes Architekturelement des Typs t existiert, zu dem eine Beziehung besteht, die von b ausgeht.

Für obiges Beispiel enthält H_{IB} nur ein Tupel:

$$H_{IB} = \{(Abrechner, AbrechnerTool, GUI)\}$$

Die Menge der Verstöße gegen interne Gebotsregeln $V_{IG} \subseteq A \times A \times T_A$ berechnet sich aus der Menge der vorgeschriebenen internen Beziehungen abzüglich der in der Ist-Architektur bestehenden internen Beziehungen:

$$V_{IG} = H_{ISB} \setminus H_{IB}$$

In obigem Beispiel findet sich ein Verstoß:

$$V_{IG} = \{(ReisebeihilfeBearbeiter, ReisebeihilfeBearbeiterTool, GUI)\}$$

Dieser Verstoß ergibt sich, da die Beziehung von ReisebeihilfeBearbeiter zu ReisebeihilfeBearbeiterGUI innerhalb des zusammengesetzten Architekturelements ReisebeihilfeBearbeiter fehlt.

Erweiterung des Berechnungsschritts (d): Verstöße in den Quelltext verfolgen

Jeder Verstoß $(a, b, t) \in V_{IG}$ gegen eine interne Gebotsregel besagt, dass das in a geschachtelte Architekturelement b fehlerhaft ist, denn b fehlt eine ausgehende Beziehung zu einem ebenfalls in a enthaltenen Architekturelement des Typs t .

Über die Zuordnung Z_{QA} lässt sich für jedes dieser fehlerhaften Architekturelemente ermitteln, welche Quelltextelemente betroffen sind.

$$V_{QIG} = \{q | \exists (a, b, t) \in V_{IG} : (q, b) \in Z_{QA}\}$$

Im Falle des obigen Beispiels besagt die Menge der Verstöße V_{IG} , dass dem Architekturelement ReisebeihilfeBearbeiterTool eine ausgehende Beziehung fehlt. Die Zuordnung Z_{QA} zwischen Quelltext- und Architekturelementen enthält in dem Beispiel das Tupel

$$(ReisebeihilfeBearbeiterToolKlasse, ReisebeihilfeBearbeiterTool).$$

In diesem Fall ist das Architekturelement genau einem Quelltextelement zugeordnet. Somit gilt:

$$V_{QIG} = \{ReisebeihilfeBearbeiterToolKlasse\}$$

6.3.3 Beziehungs- und Schnittstellenregeln in hierarchischen Architekturen

Die im Kernkonzept und in der ersten Ausbaustufe definierten Beziehungsregeln sowie die im vorherigen Abschnitt vorgestellten Schnittstellenregeln verhalten sich bei hierarchischen Architekturen (siehe Abschnitt 2.2) folgendermaßen:

- Eine Beziehungsregel des Typs Gebotsregel besagt, dass Architekturelemente des Typs t1 eine Beziehung zu einem Architekturelement des Typs t2 haben müssen. Jede Regel legt zwei Typen t1 und t2 fest. Aus Gründen der Verständlichkeit und Klarheit wird an dieser Stelle für hierarchische Architekturen festgelegt, dass sich Gebotsregeln ausschließlich auf die angegebenen Typen beziehen, nicht jedoch auf eventuell vorhandene enthaltene Architekturelemente. Dies entspricht den Gebotsregeln in allen untersuchten Stilen. Sollte zukünftig jedoch der Bedarf an Gebotsregeln bestehen, die sich auch auf enthaltene Elemente beziehen, so wäre es einfach, der stilbasierten Architekturprüfung eine entsprechende Erweiterung hinzuzufügen.
- Beziehungsregeln des Typs Verbotsregel besagen, dass Architekturelemente des Typs t1 keine Beziehung zu einem Architekturelement des Typs t2 haben dürfen. Handelt es sich bei Architekturelementen des Typs t1 oder t2 um zusammengesetzte Architekturelemente, so versteht die stilbasierte Architekturprüfung die Verbotsregel als abgekürzte Formulierung der Tatsache, dass auch alle in einem zusammengesetzten Architekturelement enthaltenen Architekturelemente von der Regel betroffen sind.
- Die erste Ausbaustufe ermöglicht, verschiedene Beziehungstypen zu unterscheiden. Diese Ausbaustufe ist orthogonal zu hierarchischen Architekturen, beide Ausbaustufen lassen sich ohne Wechselwirkungen kombinieren. Auch die internen Gebotsregeln lassen sich mit Hilfe der ersten Ausbaustufe verschiedene Beziehungstypen zu.
- Kein in dieser Arbeit betrachteter Stil, weder in der Literatur noch in der Praxis, definiert Schnittstellenregeln für zusammengesetzte Architekturelemente. Bei der Formulierung der stilbasierten Architekturprüfung wurde Wert darauf gelegt, dass sich die Regeltypen an realen Stilen orientieren. Aus diesem Grunde werden Schnittstellenregeln ausschließlich für Einzel-Elemente unterstützt. Sollte in der Zukunft der Bedarf entstehen, Schnittstellenregeln für zusammengesetzte Architekturelemente zu definieren, da neue Stile entstehen, die dieses Vorsehen, so lässt sich die stilbasierte Architekturprüfung leicht erweitern.

6.3.4 Abgrenzung zu bisherigen Ansätzen

Einige der bisherigen Ansätze erlauben es, Vorgaben für Enthältbeziehungen innerhalb von Soll-Architekturen zu definieren (vgl. Kapitel 4). Wie die anderen Vorgaben in Soll-Architekturen, so werden auch Vorgaben für Enthältbeziehungen ausschließlich auf Exemplar-ebene festgelegt, das bedeutet, für jedes Architekturelement muss einzeln festgelegt werden, welche Elemente es enthalten soll und ob es Beziehungen zwischen den enthaltenen Elementen geben darf. Die stilbasierte Architekturprüfung hingegen bietet mit Enthältregeln und internen Beziehungsregeln die Möglichkeit, für alle Elemente eines bestimmten Typs einheitliche Vorgaben festzulegen.

6.4 Zusammenfassung

In diesem Kapitel wurden die Ausbaustufen der stilbasierten Architekturprüfung vorgestellt: Die erste Ausbaustufe erlaubt es, bei Beziehungsregeln verschiedene Beziehungstypen zu unterscheiden. Die zweite Stufe adressiert Schnittstellenregeln. Die dritte Ausbaustufe ermöglicht, Regeln für hierarchische Architekturen zu definieren.

Die Gliederung in Kernkonzept und Ausbaustufen dient der klaren und strukturierten Darstellung des Ansatzes und hilft, den am häufigsten verwendeten Regeltyp – die Beziehungsregeln – hervorzuheben. Projekte können die Gliederung in Kernkonzept und Ausbaustufen nutzen, um die stilbasierte Architekturprüfung schrittweise einzuführen.

7 Eine beispielhafte Umsetzung

Um die in den vorherigen Kapiteln vorgestellte stilbasierten Architekturprüfung in der Praxis umzusetzen, bedarf es erstens einer Konkretisierung und zweitens eines Werkzeugs, mit dem sich die Prüfung durchführen lässt.

Dieses Kapitel stellt eine solche Umsetzung vor. Sie dient im Rahmen dieser Arbeit mehreren Zwecken:

- Sie soll die stilbasierte Architekturprüfung für die Leserinnen und Leser greifbarer machen, indem sie zeigt, wie sich die in den vorherigen Kapiteln vorgestellten Konzepte mittels eines prototypischen Werkzeugs praktisch umsetzen lassen.
- Die Umsetzung wurde während der Forschungstätigkeiten genutzt, um den Ansatz der stilbasierten Architekturprüfung iterativ zu entwickeln.
- Die Umsetzung diene als Basis für die im anschließenden Kapitel vorgestellten Machbarkeitsnachweise. Mit Hilfe des prototypischen Werkzeugs wurden verschiedene Softwaresysteme untersucht und erfolgreich Verstöße gegen den gewählten Architekturstil aufgedeckt.

Wird ein Werkzeug für die stilbasierte Architekturprüfung erstellt, so ist es notwendig, das Konzept des Ansatzes in folgenden Bereichen zu konkretisieren:

- *Programmiersprache der zu prüfenden Systeme:* Das Konzept der stilbasierten Architekturprüfung abstrahiert von konkreten Programmiersprachen, damit sie für verschiedene objektorientierte, statisch getypte Sprachen einsetzbar ist. Das Werkzeug muss sich auf eine oder mehrere konkrete Programmiersprachen festlegen, damit es die Quelltextstruktur durch statische Analyse aus dem Quelltext berechnen kann.
- *Repräsentation der Zusatzinformationen:* Der Ansatz der stilbasierten Architekturprüfung legt fest, *welche* Zusatzinformationen benötigt werden (vergleiche Kapitel 5 und 6). Er empfiehlt, den Quelltext mit diesen Zusatzinformationen zu *annotieren*, um auf diese Weise den Quelltext als einzige Quelle zur Ermittlung der Ist-Architektur nutzen zu können. Der Ansatz schlägt konkret vor, an welchen *Quelltextstellen* jeweils welche Zusatzinformationen annotiert werden: jeweils an Quelltextelementen, die einem Architekturelement zugeordnet sind (vergleiche Abschnitt 5.3.3). Dieser Vorschlag orientiert sich an bisherigen, stilbasierten Systemen, bei denen Teile der Zusatzinformationen beispielsweise in Form von Kommentaren im Quelltext vorhanden sind. Ferner steht es den Werkzeug-Designerinnen und -Designern frei, in besonderen Fällen die Zusatzinformationen quelltextextern zu speichern, beispielsweise wenn Systeme untersucht werden sollen, deren Quelltext nicht geändert werden darf. In diesem Fall muss das Werkzeug ermöglichen, dass die Zusatzinformationen in der Benutzungsschnittstelle des Werkzeugs direkt neben den Quelltextelementen angezeigt und bearbeitet werden können, so dass für die Benutzerinnen und Benutzer kaum ein Unterschied wahrnehmbar ist.
- Die Kapitel 5 und 6 verwenden für die *Darstellung* der Zusatzinformationen eine mengentheoretische Notation, da sich diese gut für die Erläuterung der Konzepte eignet. Wird ein Werkzeug für die stilbasierte Architekturprüfung entwickelt, so steht es den Designerinnen und Designern dieses Werkzeuges frei, eine andere Notation als die

mengentheoretische Darstellung zu wählen. In der Programmiersprache Java beispielsweise bietet es sich an, das integrierte Sprachmittel der Java-Annotationen zu nutzen.

- *Repräsentation des Architekturstils:* Auch für die Definition des Architekturstils benötigt das Werkzeug eine Beschreibungssyntax. Diese kann ebenfalls die mengentheoretische Notation des vorherigen Kapitels sein, alternativ sind andere Beschreibungssprachen wie beispielsweise XML möglich.

Für das im Rahmen der hier vorgestellten, beispielhaften Umsetzung erstellte prototypische Werkzeug, den StyleBasedChecker, galt es folgende technische Aspekte festzulegen:

- *Entwicklungsumgebung:* In welche Entwicklungsumgebung soll das Werkzeug eingefügt werden?
- *Benutzungsschnittstelle:* Wie gestaltet sich die Benutzungsschnittstelle?
- *Interner Aufbau des Werkzeugs:* Welche Architektur soll das Werkzeug haben, mit welchen Technologien und welcher Programmiersprache soll es realisiert werden?

Dieses Kapitel beginnt mit der Konkretisierung des Konzepts der stilbasierten Architekturprüfung und stellt anschließend den StyleBasedChecker vor.

7.1 Die stilbasierte Architekturprüfung für Java-Systeme

Für die hier vorgestellte Umsetzung bot sich aus mehreren Gründen die Programmiersprache Java an: Java ist eine etablierte, moderne, objektorientierte, statisch getypte Sprache. Für Java steht die ebenfalls sehr verbreitete Entwicklungsumgebung (IDE) Eclipse zur Verfügung, die umfangreiche Erweiterungsschnittstellen anbietet, so dass sich ein Prototyp zur stilbasierten Architekturprüfung als Eclipse-Plugin realisieren ließ. Der Hauptgrund für die Wahl der Sprache Java ist jedoch, dass die in dieser Arbeit betrachteten Architekturstile am häufigsten in Java-Systemen umgesetzt wurden, so dass für die im nächsten Kapitel vorgestellten Machbarkeitsnachweise eine Vielfalt an Systemen zur Verfügung stand.

7.1.1 Die Quelltextstruktur für Java-Systeme ermitteln

Zur Erinnerung: Quelltextstrukturen bestehen laut Definition 2-2 aus *Quelltextelementen*, den *Schnittstellen* der Quelltextelemente und deren *Beziehungen*.

Um die Quelltextstruktur für eine bestimmte Programmiersprache zu ermitteln, muss für diese Sprache festgelegt werden, was konkret unter den Quelltextelementen, den Schnittstellen und den Beziehungen verstanden werden soll. Dann lässt sich die Quelltextstruktur durch statische Analyse aus dem Quelltext eines Softwaresystems berechnen. Diese beispielhafte Umsetzung basiert auf dem für Java üblichen Verständnis dieser Begriffe.

- *Quelltextelemente* sind in dieser Umsetzung die verschiedenen Typen, die sich in Java definieren lassen. Neben den Klassen, die in objektorientierten Programmiersprachen üblicherweise definiert werden können, unterstützt Java drei weitere Typen: Interfaces, Enumerationen und Annotationsdefinitionen. Wird im Folgenden von Klassen gesprochen, so sind auch Interfaces, Enumerationen und Annotationsdefinitionen gemeint. Ist dies nicht der Fall, so wird es explizit gekennzeichnet. In Java ist es möglich, geschachtelte Typen zu definieren: Klassen können weitere Klassen enthalten. Klassen, die nicht

in einer anderen Klasse enthalten sind, werden als Top-Level-Klassen bezeichnet. Klassen, die in einer anderen Klasse enthalten sind, heißen geschachtelte Klassen. Die hier vorgestellte Umsetzung betrachtet jeweils jede Top-Level-Klasse gemeinsam mit allen enthaltenen Klassen als ein einziges Quelltextelement. Dies ist ein in der Praxis verbreitetes Verständnis. So sehen es auch die Entwicklungsteams der untersuchten Systeme.

- *Beziehungen:* Wie in Kapitel 2 definiert, lassen sich in objektorientierten Programmiersprachen drei Beziehungstypen unterscheiden: Vererbungs-, Benutzt-, und Enthältbeziehungen. Da in dieser Umsetzung Top-Level-Klassen und enthaltene Klassen als Einheit angesehen werden, sind hier lediglich die Vererbungs- und die Benutztbeziehungen relevant.
- *Schnittstellen:* Die Schnittstelle einer Java-Klasse umfasst alle öffentlichen Operationen und Konstruktoren der Klasse. Formaler ausgedrückt beinhaltet die Schnittstelle die Liste der Parametertypen für alle Konstruktoren und Operationen und zusätzlich den Bezeichner und den Rückgabetypp für alle Operationen. Dies entspricht den Informationen, die üblicherweise in Signaturen³⁴ enthalten sind.

7.1.2 Die Zusatzinformationen mit Java-Annotationen beschreiben

Zur Erinnerung, die Zusatzinformationen umfassen Folgendes:

- Die Zuordnung zwischen Quelltext- und Architekturelementen ($Z_{QA} \subseteq Q \times A$).
- Die Zuordnung der Architekturelemente zu ihrem jeweiligen Typ ($Z_{AT} \subseteq A \times T_A$).
- Die Enthältbeziehungen zwischen Architekturelementen ($B_E \subseteq A \times A$).

Die stilbasierte Architekturprüfung sieht vor, dass Teammitglieder diese Informationen zur Entwicklungszeit in der Entwicklungsumgebung angeben können. In dem Moment, in dem sie den Quelltext für ein Architekturelement erstellen oder verändern, versehen sie die zugehörigen Quelltextelemente mit den Zusatzinformationen. In dieser Umsetzung werden die Zusatzinformationen in Form von Java-Annotationen festgehalten.

Bei Java-Annotationen handelt es sich um ein Konzept der Programmiersprache Java, das es erlaubt, Klassen und andere Quelltextabschnitte mit Zusatzinformationen zu versehen. In dieser Umsetzung werden Annotationen ausschließlich für Top-Level-Klassen verwendet.

Java beinhaltet einige vordefinierte Annotationstypen. Darüber hinaus können Entwicklerinnen und Entwickler eigene Typen definieren. Dieser Mechanismus wird für die beispielhafte Umsetzung genutzt. Jeder Annotationstyp besitzt einen Bezeichner und kann darüber hinaus getypte Felder definieren.

Der folgende Quelltext zeigt ein Beispiel für die Definition eines Annotationstyps in Java. Der Annotationstyp beinhaltet zwei Felder des Typs String, die Felder heißen `typ` und `name`³⁵.

³⁴ Der hier verwendete Signatur-Begriff umfasst ausdrücklich den Rückgabetypp, wie bei den meisten Programmiersprachen üblich.

³⁵ Es sei angemerkt: Es wäre problemlos möglich, den Prototypen so zu ergänzen, dass die Angabe des Typs durch die Verwendung einer Enumeration vor Fehlern geschützt wird. Alternativ ließe sich ebenso leicht für jeden Architekturelement-Typ ein eigener Annotationstyp erstellen.

```

public @interface Architekturelement
{
    String typ();
    String name();
}

```

Wird eine Klasse mit einer Annotation versehen, so geschieht dies direkt in der Zeile über dem Klassenkopf. Die Annotation enthält den Namen des Annotationstyps und Werte für die zugehörigen Felder. Die im vorherigen Kapitel genannte Klasse `Device` beispielsweise gehört zu dem Architekturelement `Device-ArchElem`. Der Typ dieses Architekturelements ist `Material`. Diese Informationen lassen sich mit Java-Annotationen folgendermaßen ausdrücken:

```

@Architekturelement
(
    typ = "Material",
    name = "Device-ArchElem"
)

```

Jede Klasse, die einem Architekturelement zugeordnet ist, erhält als Annotation den Namen und den Typ des Architekturelements.

Ist das Architekturelement ein Teil eines zusammengesetzten Architekturelements, so wird die Klasse zusätzlich mit dem Namen des zusammengesetzten Architekturelements annotiert. Der hier verwendete Annotationstyp sieht somit folgendermaßen aus:

```

public @interface Architekturelement
{
    String typ();
    String name() default "";
    String[] uebergeordneteArchitekturelemente () default {};
}

```

Abbildung 86 zeigt ein bekanntes Beispiel aus dem WAM-Stil. Der WAM-Stil definiert den Architekturelement-Typ `MonoWerkzeug`. Architekturelemente dieses Typs sind zusammengesetzt. Sie enthalten jeweils ein Architekturelement des Typs `GUI` und eines des Typs `MonoTool`. Dies ist in der Grafik verdeutlicht, indem die Typen `GUI` und `MonoTool` innerhalb von `MonoWerkzeug` dargestellt sind.



Abbildung 86: Beispiel für einen zusammengesetzten Architekturelement-Typ

Das in Abbildung 87 dargestellte Architekturelement folgt den in der vorherigen Abbildung dargestellten Vorgaben. Das zusammengesetzte Architekturelement namens ReisebeihilfeBearbeiter hat den Typ MonoWerkzeug. Es enthält zwei Architekturelemente, ReisebeihilfeBearbeiterGUI des Typs GUI und ReisebeihilfeBearbeiterTool des Typs MonoTool³⁶.

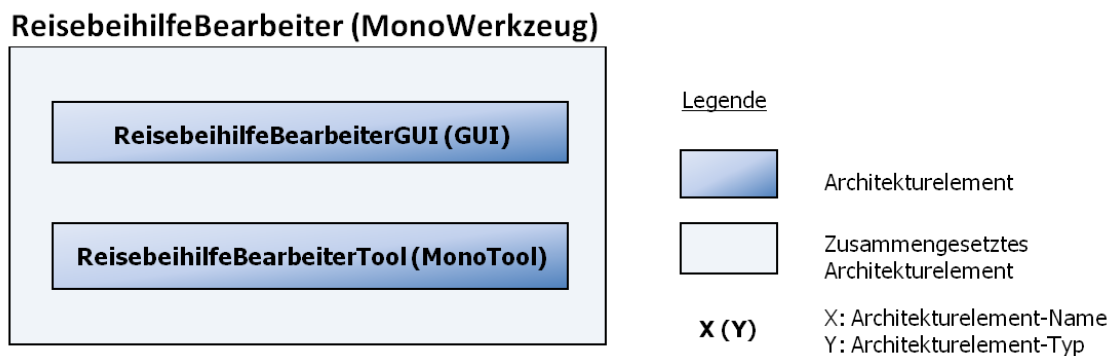


Abbildung 87: Ein zusammengesetztes Architekturelement

Das Architekturelement ReisebeihilfeBearbeiterGUI ist genau einer Klasse zugeordnet, der Klasse ReisebeihilfeBearbeiterGUI. Diese Klasse erhält folgende Annotation:

```
@ Architekturelement
(
  name = "ReisebeihilfeBearbeiterGUI",
  typ = "GUI",
  uebergeordneteArchitekturelemente = {"ReisebeihilfeBearbeiter"}
)
```

In der Praxis wird meist mit einer Schachtelungstiefe von 0 bis 1 gearbeitet, das heißt, zusammengesetzte Architekturelemente enthalten meist keine weiteren zusammengesetzten Elemente.

³⁶ Um das Verständnis zu erleichtern, sei an dieser Stelle darauf hingewiesen, dass die Typen MonoWerkzeug und MonoTool zwei unterschiedliche Typen sind, trotz der ähnlichen Benennung (vgl. Abschnitt 3.3.1).

Die Umsetzung erlaubt jedoch, beliebig viele Architekturelemente zu schachteln. Wäre beispielsweise eine weitere Schachtelung vorgesehen, so dass der ReisebeihilfeBearbeiter in einem weiteren zusammengesetzten Architekturelement namens A enthalten wäre, dann würde der Name A in dem Array der übergeordneten Architekturelemente hinter dem ReisebeihilfeBearbeiter genannt:

```
@Architekturelement
(
  name = "ReisebeihilfeBearbeiterGUI",
  typ = "GUI",
  uebergeordneteArchitekturelemente = {"ReisebeihilfeBearbeiter", "A"}
)
```

In allen untersuchten Stilen und Systemen besaßen lediglich atomare Architekturelemente eine Zuordnung zu Quelltextelementen. Die zusammengesetzten Architekturelemente erhielten den ihnen zugeordneten Quelltext indirekt über die enthaltenen atomaren Architekturelemente. Aus diesem Grund unterstützt die beispielhafte Umsetzung keine Zuordnung von Quelltextelementen zu zusammengesetzten Architekturelementen. Diese Möglichkeit ließe sich bei Bedarf jedoch leicht ergänzen.

Die Umsetzung unterstützt den in der Praxis beobachteten Fall, dass Architekturelemente, die nur aus einer Klasse bestehen, denselben Namen wie die Klasse tragen. In diesem Fall kann der Name des Architekturelements in der Annotation weggelassen werden. Wird kein Name in der Annotation genannt, erhält das Architekturelement automatisch denselben Namen wie die zugeordnete Klasse.

7.1.3 Den Architekturstil mit XML beschreiben

Die hier beispielhafte Umsetzung unterstützt das Kernkonzept der stilbasierten Architekturprüfung sowie die Ausbaustufen für Schnittstellenregeln und hierarchische Architekturen.

Zur Erinnerung: um einen Architekturstil zu beschreiben, der alle Regeltypen des Kernkonzepts sowie der genannten Ausbaustufen enthält, werden folgende Informationen benötigt (vgl. Definition 3-2 sowie die Ausbaustufen in Kapitel 6):

- Die Menge der Architekturelement-Typen: T_A
- Die Menge der Schnittstellenregeln: R_S sowie die Zuordnung von Architekturelement-Typen zu diesen Regeln: $Z_{TSR} \subseteq T_A \times R_S$
- Die Menge der Beziehungsregeln, diese umfassen folgende Regeltypen:
- Gebotsregeln schreiben Beziehungen vor: $R_G \subseteq T_A \times T_A$
- Verbotsregeln untersagen Beziehungen: $R_V \subseteq T_A \times T_A$
- Enthält-Beziehungsregeln schreiben für zusammengesetzte Architekturelemente vor, aus welchen Typen sie aufgebaut sind: $R_E \subseteq T_A \times T_A$
- Interne Gebotsregeln schreiben Beziehungen vor zwischen Architekturelementen innerhalb eines zusammengesetzten Architekturelements: $R_{IG} \subseteq T_A \times T_A \times T_A$

Die Menge der Kann-Beziehungsregeln ist hier bewusst nicht genannt, da diese Menge, wie in Abschnitt 3.2 erläutert, über die Gebots- und Verbotsregeln vollständig beschrieben ist.

Die in den vorherigen Kapiteln genutzte mathematische Mengennotation ist in ihrer Kürze und Präzision gut geeignet, um das Konzept der stilbasierten Architekturprüfung vorzustellen. Jedoch sind Mengennotationen in der täglichen Praxis der Software-Entwicklung mit Java unüblich. Die beispielhafte Umsetzung dieser Arbeit geht daher bei der Beschreibung von Architekturstilen einen anderen, praxisnäheren Weg: sie definiert eine Beschreibungssyntax für Architekturstile, die sich auf bestehende Standards für Beschreibungssprachen stützt. Sie nutzt die Extensible Markup Language (XML). Bei XML handelt es sich um ein weit verbreitetes Textformat, spezifiziert durch das World Wide Web Consortium (W3C)³⁷. XML ist eine Sprache mit der sich hierarchisch strukturierte Daten in menschen- und maschinenlesbarer Form darstellen lassen (Harold und Means 2004). XML ist auch im Java-Umfeld stark verbreitet, es wird in vielen Rahmenwerken und Werkzeugen angewendet, um diese zu konfigurieren. Für XML steht eine Vielzahl ausgereifter Bibliotheken zur Verfügung. XML ist den Entwicklerinnen und Entwicklern in ihrer täglichen Arbeit vertraut und stellt keine Hürde dar.

Mit Hilfe von XML lassen sich eigene Sprachen definieren, die die XML Syntax einschränken, so dass man selbst definierte hierarchische Datenstrukturen beschreiben kann. Auf diese Weise lassen sich Architekturstile in einem Format beschreiben, das auf die Struktur von Architekturstilen spezialisiert ist.

Die beispielhafte Umsetzung definiert eine eigene XML-Syntax für Architekturstile. Solch eine selbst definierte Syntax für XML wird mit Hilfe eines XML-Schemas beschrieben. Das Konzept der XML-Schemata ist ebenfalls durch das W3C spezifiziert (Harold und Means 2004). Für XML und XML-Schemata stehen Bibliotheken zur Verfügung, mit denen sich XML-Beschreibungen unter anderem parsen und auf ihre syntaktische Korrektheit prüfen lassen. Die Bibliotheken lassen sich in eigene Softwaresysteme einbetten.

Im Folgenden soll die gewählte Syntax vorgestellt werden. Als Beispiel dient ein vereinfachter Ausschnitt des WAM-Stils. Die komplette Syntaxdefinition in Form eines XML-Schemas ist umfangreich und findet sich im Anhang der Arbeit.

XML besteht aus einer Menge geschachtelter sogenannter Tags. Ein Tag startet mit der Tag-Bezeichnung, die in spitze Klammern eingebettet wird. Solch ein Start-Tag sieht beispielsweise folgendermaßen aus:

```
<architekturelementTypen>
```

Die nachfolgend dargestellte Stilbeschreibung beginnt mit dem Tag namens `architekturstilDefinition`. Ein Tag, das weitere Tags enthält, endet wiederum mit der Tag-Bezeichnung in spitzen Klammern, dabei folgt der ersten Spitzklammer ein Schrägstrich. Das Ende-Tag für obiges Beispiel lautet:

```
</architekturelementTypen>
```

Tags können sogenannte Attribute enthalten, das sind Schlüsselwort-Werte-Paare. Diese folgen direkt nach dem Start-Tag, vor der schließenden spitzen Klammer. Beispielsweise enthält das folgende Tag namens `architekturstilDefinition` ein Attribut mit dem Schlüsselwort `stilBezeichnung`, das in diesem Beispiel den Wert „WAM-Stil“ erhält.

³⁷ www.omg.org

```
<architekturstilDefinition stilBezeichnung="WAM-Stil">
```

Tags, die keine weiteren geschachtelten Tags enthalten, bestehen lediglich aus der Tag-Bezeichnung in spitzen Klammern, dabei geht der schließenden spitzen Klammer ein Schrägstrich voraus. Auch diese Tags können Attribute enthalten, wie beispielsweise bei dem folgenden Tag namens typ.

```
<typ typBezeichnung="MonoWerkzeug" />
```

Das folgende Beispiel zeigt die Beschreibung eines vereinfachten Ausschnitts des WAM-Stils mit der für diese Umsetzung gewählten XML-Syntax. Die einzelnen Bestandteile des Beispiels werden anschließend erläutert.

```
<architekturstilDefinition stilBezeichnung="WAM-Stil">

  <architekturelementTypen>

    <typ typBezeichnung="MonoWerkzeug" />
    <typ typBezeichnung="GUI" />
    <typ typBezeichnung="MonoTool" />
    <typ typBezeichnung="Material" />

  </architekturelementTypen>

  <regeln>

    <gebotsregel von="MonoTool" zu="Material"/>

    <verbotsregel von="Material" zu="MonoWerkzeug" />
    <verbotsregel von="GUI" zu="MonoTool" />

    <schnittstellenregel typ="Material"
      regelName="NotOnlyGetterAndSetter" />

    <aufbauregel38 typZusammengesetzt="MonoWerkzeug">
      <enthaelt typ="GUI" />
      <enthaelt typ="MonoTool" />
    </aufbauregel>

    <interneGebotsregel typZusammengesetzt= MonoWerkzeug
      von="MonoTool" zu="GUI" />

  </regeln>

</architekturstilDefinition>
```

³⁸ Enthältregeln werden in der beispielhaften Umsetzung als Aufbauregeln bezeichnet.

Was bedeutet dieses Beispiel im Detail?

- Das äußerste Tag namens `architekturstilDefinition` enthält den Namen des Stils über das Attribut `stilBezeichnung`. Es enthält zwei geschachtelte Tags: `architekturelementTypen` und `regeln`.
- Das Tag `architekturelementTypen` enthält wiederum geschachtelte Tags namens `typ`, jeweils eines für jeden im Stil definierten Architekturelement-Typ. Formal ausgedrückt definiert jedes `typ`-Tag ein Element der Menge T_A . Soll beispielsweise ausgedrückt werden, dass der Stil einen Architekturelement-Typ namens `MonoWerkzeug` enthält, so hieße dies in Mengennotation:

$$\text{MonoWerkzeug} \in T_A$$

In XML wird die Regel folgendermaßen dargestellt:

```
<typ typBezeichnung="MonoWerkzeug" />
```

- Das Tag namens `regeln` enthält für jede Regel des Stils ein geschachteltes Tag.
- Tags für Gebotsregeln haben die Bezeichnung `gebotsregel` sowie die Attribute `von` und `zu`. Das Attribut `von` enthält den Elementtyp, von dem die Beziehung ausgehen soll, `zu` enthält den Elementtyp, zu dem die Beziehung bestehen soll. Obiges Beispiel enthält die Gebotsregel „MonoTools müssen Materialien kennen“. In Mengennotation ausgedrückt bedeutet dies:

$$(\text{MonoWerkzeug}, \text{Material}) \in R_G$$

In XML wird die Regel folgendermaßen dargestellt:

```
<gebotsregel von="MonoTool" zu="Material"/>
```

- Verbotsregeln werden über Tags des Namens `verbotsregel` beschrieben. Sie enthalten die gleichen Attribute wie Gebotsregeln. Architekturelemente des im Attribut `von` angegebenen Typs dürfen keine Beziehung zu Architekturelementen des im Attribut `zu` angegebenen Tag haben. Das bedeutet formal ausgedrückt, die Tupel der Menge $R_V \subseteq T_A \times T_A$ werden jeweils über ein `verbotsregel`-Tag beschrieben. Die erste Komponente des Tupels wird über das Attribut namens `von` definiert, die zweite Komponente über das Attribut namens `zu`. Die Regel „Materialien dürfen keine MonoWerkzeuge kennen“ wird in Mengennotation sowie in XML folgendermaßen dargestellt:

$$(\text{Material}, \text{MonoWerkzeug}) \in R_V$$

```
<verbotsregel von="Material" zu="MonoWerkzeug" />
```

- Für Schnittstellenregeln muss die Zuordnung zwischen Architekturelement-Typ und Regelbezeichnung angegeben werden: $Z_{TSR} \subseteq T_A \times S_R$. Jedes Tag des Namens `schnittstellenregel` beschreibt ein Tupel dieser Zuordnung. Die erste Komponente des Tupel wird über das Attribut `typ` angegeben, die zweite Komponente über das Attribut namens `regelName`. Wie in Abschnitt 6.2 erläutert, werden Schnittstellenregeln als ausprogrammierte Funktionen der Form $S_A \rightarrow \{true, false\}$ realisiert; dabei bezeichnet S_A die Menge der Schnittstellen. Die Regelbezeichnung in der Stilbeschreibung referenziert solch eine ausprogrammierte Funktion. Die Vorgabe: „Materialien dürfen nicht nur Getter und Setter definieren“ wird hier als `notOnlyGetterAndSetter` bezeichnet. Die Regel bedeutet in Mengennotation sowie in XML Folgendes:

$$(Material, notOnlyGetterAndSetter) \in Z_{TSR}$$

```
<schnittstellenregel
  typ="Material"
  regelName= "NotOnlyGetterAndSetter" />
```

- Enthältregeln, hier als Aufbauregeln bezeichnet, werden über das Tag `aufbauregel` beschrieben, welches ein Attribut namens `typZusammengesetzt` besitzt. Mit diesem Attribut wird der Typ des zusammengesetzten Architekturelements angegeben. Ferner können beliebig viele geschachtelte Tags des Namens `enthaelt` aufgeführt werden. Jedes dieser Tags beinhaltet das Attribut `typ`. Mit diesem Attribut wird jeweils der Typ eines geschachtelten Architekturelements angegeben. Soll beispielsweise ausgedrückt werden, dass MonoWerkzeuge eine GUI und ein MonoTool enthalten, so geschieht dies in Mengennotation und in XML folgendermaßen:

$$(MonoWerkzeug, GUI) \in R_E \wedge (MonoWerkzeug, MonoTool) \in R_E$$

```
<aufbauregel typZusammengesetzt="MonoWerkzeug">
  <enthaelt typ="GUI" />
  <enthaelt typ="MonoTool" />
</aufbauregel>
```

- Tags für interne Gebotsregeln sind über das Tag `interneGebotsregel` beschrieben. Diese Tags enthalten drei Attribute. Das Attribut `typZusammengesetzt` gibt den Typ der betroffenen, zusammengesetzten Architekturelemente an. Die Attribute `von` und `zu` bezeichnen jeweils den Typ eines enthaltenen Architekturelements. Für jedes Architekturelement des zusammengesetzten Typs, welches mindestens je ein Element der unter `von` und `zu` angegebenen Typen enthält, gilt: jedes enthaltene Element des unter `von` angegebenen Typs muss eine Beziehung zu mindestens einem enthaltenen Element des unter `zu` angegebenen Typs haben. Die Regel „In MonoWerkzeugen gilt: MonoTools müssen eine GUI kennen“ wird in Mengennotation und XML folgendermaßen dargestellt:

$$(MonoWerkzeug, MonoTool, GUI) \in R_{IG}$$

```
<interneGebotsregel typZusammengesetzt= MonoWerkzeug
    von="MonoTool"
    zu="GUI" />
```

Alle Regeln, abgesehen von den Schnittstellenregeln, werden in der beispielhaften Umsetzung vollständig über solche XML-Texte beschrieben.

Die XML-Beschreibung und die ausprogrammierten Schnittstellenregeln lassen sich unverändert für verschiedene Java-Softwaresysteme des gleichen Stils wiederverwenden. Wird derselbe Stil also erneut geprüft, so entsteht keinerlei Aufwand für die Stilbeschreibung.

7.2 Der StyleBasedChecker: Ein Werkzeug zur stilbasierten Architekturprüfung

Im Rahmen der vorliegenden Arbeit wurden zwei beispielhafte Werkzeuge erstellt. Das erste Werkzeug war eine Erweiterung des kommerziellen Werkzeugs Sotograph (siehe Kapitel 4). Sie entstand im Rahmen einer von der Autorin konzipierten und betreuten Diplomarbeit (Karstens 2005). Diese erste Umsetzung stellt ein Zwischenergebnis der Forschungen zur stilbasierten Architekturprüfung dar. Interessante, mit diesem ersten Werkzeug gewonnene Erkenntnisse werden in Kapitel 8 „Praktische Untersuchungen und Evaluationen“ thematisiert. In diesem Abschnitt soll das zweite Werkzeug, der StyleBasedChecker, vorgestellt werden. Die erste Version des StyleBasedCheckers wurde im Rahmen der, von der Autorin betreuten, Diplomarbeit Arne Scharpings erstellt (Scharping 2008). Die Diplomarbeit war Teil der Forschungstätigkeiten zur stilbasierten Architekturprüfung. Nach Abschluss der Diplomarbeit wurde der StyleBasedChecker von der Autorin ergänzt und überarbeitet, passend zum fortschreitenden Forschungsstand der Arbeit.

Der StyleBasedChecker ist ausgerichtet auf die Programmiersprache Java. Er ist realisiert für die Entwicklungsumgebung Eclipse³⁹. Eclipse ist eine erweiterbare Entwicklungsumgebung, die eine offene Programmierschnittstelle (API) anbietet. Erweiterungen werden in der Eclipse-Terminologie als Plugin bezeichnet. Der StyleBasedChecker ist als ein solches Eclipse-Plugin realisiert.

Über die Eclipse-API greift der StyleBasedChecker auf die Parse-Informationen des Compilers zu. Aus diesen Informationen erstellt er die Quelltextstruktur der zu prüfenden Systeme. Ebenfalls über die Eclipse-API meldet der StyleBasedChecker die gefundenen Verstöße.

7.2.1 Benutzung des StyleBasedCheckers

Den Architekturstil beschreiben

Entscheidet sich ein Entwicklungsteam, den StyleBasedChecker einzusetzen, so beschreibt es im ersten Schritt den für das Projekt gewählten Architekturstil. Dies geschieht in der oben erläuterten XML-Notation. Hierfür nutzt der StyleBasedChecker den in Eclipse vorhandenen XML-Editor (siehe Abbildung 88).

³⁹ www.eclipse.org

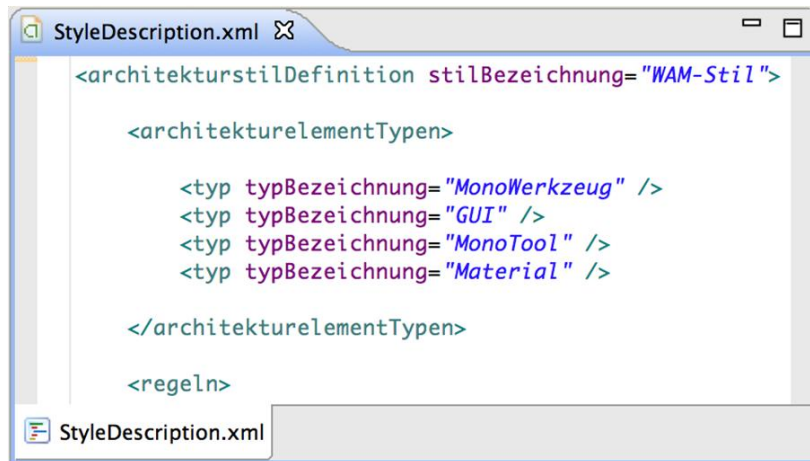


Abbildung 88: Der StyleBasedChecker nutzt den XML-Editor der Eclipse-IDE für Stilbeschreibungen

Zusätzlich zur XML-Beschreibung muss das Team seine Schnittstellenregeln programmieren. Hierfür bietet der StyleBasedChecker eine Programmierschnittstelle mit einem Java-Interface namens SchnittstellenRegel. Für jede Schnittstellenregel erstellt das Team eine dieses Interface implementierende Regelklasse. Jede Regelklasse implementiert eine in der Oberklasse deklarierte Operation namens pruefeSchnittstelle, die als Parameter die Schnittstelle eines Architekturelements erhält. Abbildung 89 zeigt dies beispielhaft für die Regelklasse namens NichtNurSetterUndGetter.

Die als Parameter übergebene Architekturelement-Schnittstelle beinhaltet die Schnittstellen aller dem Architekturelement zugeordneter Klassen. Diese Schnittstellen bestehen – wie oben beschrieben – aus den Signatur-Informationen aller Operationen und Konstruktoren. Die Schnittstellen lassen sich an Objekten des Typs ArchitekturelementSchnittstelle über entsprechende Operationen abfragen.

Ein Entwicklungsteam braucht seinen Stil nur einmalig zu beschreiben. Der StyleBasedChecker verwendet die XML-Datei und die Regelklassen für jeden Prüfvorgang. Auch wenn sich die Architektur des Systems ändert, so braucht das Team seine Stilbeschreibung nicht zu überarbeiten. Für ein neues Java-Softwaresystem desselben Stils kann das Team die XML-Datei und Schnittstellenregel-Klassen übernehmen.

Da der StyleBasedChecker als Prototyp im Rahmen dieser Arbeit lediglich für Java-Systeme entwickelt wurde, ist die Übertragbarkeit der XML-Datei und der Schnittstellenregel-Klassen auf Java beschränkt. Mit einer entsprechenden Erweiterung des StyleBasedCheckers ließe sich die XML-Beschreibung auch auf Systeme anderer objektorientierter Programmiersprachen übertragen, sie ist unabhängig von Java. Auch die Schnittstellenregel-Klassen können prinzipiell sprachunabhängig implementiert werden, indem ein sprachunabhängiges Metamodell für objektorientierte Sprachkonstrukte (Trifu und Szulman 2005; Becker et al. 2010) verwendet wird. In der prototypischen Implementierung des StyleBasedCheckers wurden die Schnittstellenregeln der Einfachheit halber sprachabhängig implementiert, sie verwenden den Parse-Baum des Java-Compilers.

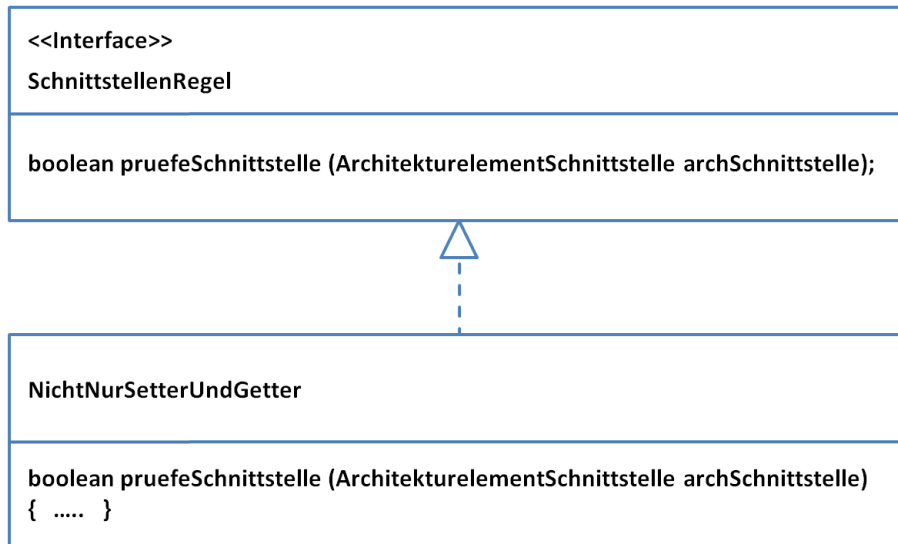


Abbildung 89: Konkrete Schnittstellenregeln implementieren das Interface SchnittstellenRegel einschließlich der Operation pruefeSchnittstelle (UML-Darstellung)

Die Zusatzinformationen einfügen

Der StyleBasedChecker verwendet Zusatzinformationen in Form von Java-Quelltext-annotationen. Wenn ein Teammitglied den Quelltext des Softwaresystems um eine neue Klasse ergänzt, so kommt aus Sicht der stilbasierten Architekturprüfung ein neues Quelltextelement hinzu. Sofern diese Quelltextelement einem Architekturelement zugeordnet ist, so muss das Teammitglied seine Klasse entsprechend annotieren, wie im vorherigen Abschnitt beschreiben (siehe Abbildung 90). Auch für den selteneren Fall, dass sich die Zusatzinformationen zu einer bereits bestehenden Klasse ändern, muss das Teammitglied diese Klasse annotieren. Da es in stilbasierten Systemen generell üblich ist, die Zusatzinformationen beispielsweise durch Kommentare an den Quelltextelementen zu annotieren, entsteht kein Mehraufwand (siehe Abschnitt 5.1.2).

```

@Architekturelement(name = "Device-ArchElem", typ = "Material")
public class Device
{

```

Abbildung 90: Die Klasse Device mit annotierten Zusatzinformationen

Die Prüfung durchführen

Der StyleBasedChecker prüft die Stiltreue bei jedem Kompilervorgang. Er unterstützt sowohl Kompilervorgänge, bei denen das komplette Projekt übersetzt wird, als auch partielle Kompilervorgänge. Komplette Projekte werden kompiliert, indem man einen entsprechenden Menüeintrag in Eclipse auswählt. Abhängig von den gewählten Einstellungen der Eclipse-IDE

können partielle Kompilervorgänge auftreten, sobald Quelltext im Editor bearbeitet wird und wenn Dateien gespeichert werden. Bei partiellen Kompilervorgängen werden die geänderten Dateien und abhängige Dateien kompiliert. Der StyleBasedChecker prüft in diesen Fällen ebenfalls nur die Anteile der Architektur, welche durch die Änderungen neue Verstöße enthalten könnten oder in denen Verstöße beseitigt sein könnten. Dies dient der Performanz des Werkzeugs bei größeren Projekten.

Basierend auf dem annotierten Quelltext, der XML-Beschreibung und den Schnittstellenregel-Klassen berechnet der StyleBasedChecker, ob das Softwaresystem gegen seinen Architekturstil verstößt. Die Ergebnisse der Prüfung zeigt das Werkzeug in einer für Eclipse üblichen Weise an: es visualisiert die gefunden Verstöße als sogenannte Probleme (engl. problems). Probleme sind einer Quelltextzeile zugeordnet und enthalten eine Problembeschreibung. Bei Verstößen, die sich einer oder mehreren konkreten Quelltextzeilen zuordnen lassen, zeigt der StyleBased-Checker die Probleme in diesen Quelltextzeilen an. Verstöße, die sich lediglich ganzen Klassen zuordnen lassen, werden in der Quelltextzeile des Klassenkopfs angezeigt. Im Quelltext erscheinen die Probleme als Warndreieck, ein Tooltip enthält die zugehörige Fehlermeldung. Darüber hinaus zeigt Eclipse alle Probleme in einer gesonderten Liste an und bietet über einen Doppelklick die Möglichkeit, von einem Listeneintrag direkt in den betroffenen Quelltext zu springen. Abbildung 91 zeigt, wie Eclipse die Probleme darstellt. Die Abbildung zeigt ein reales Beispiel aus einem im Rahmen dieser Arbeit untersuchten System.

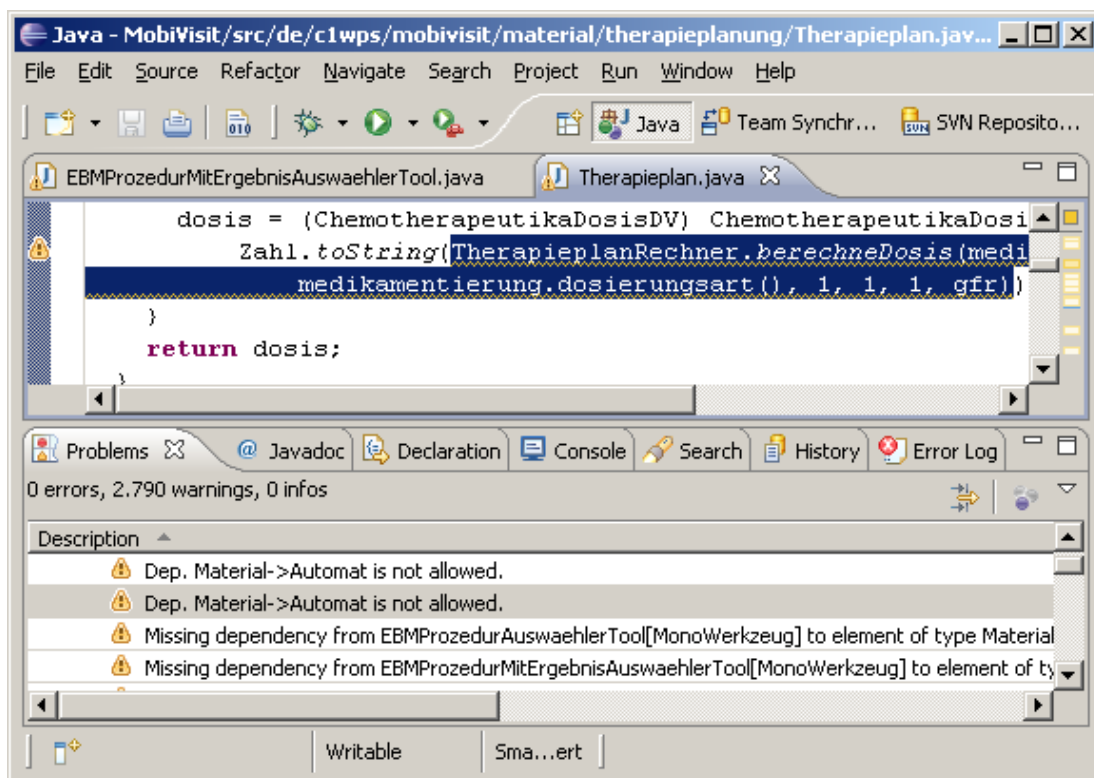


Abbildung 91: Die Verstöße werden in der Eclipse-IDE als sogenannte Probleme dargestellt. Eclipse markiert Probleme direkt im Quelltext und bietet eine Liste aller Probleme.

Teammitglieder, die ihre Architektur korrigieren möchten, können über die Liste der Verstöße direkt zu den betroffenen Quelltextstellen springen. Teammitglieder, die gerade ein Quelltext-

element bearbeiten, sehen sofort, ob dieses Element einen Architekturverstoß verursacht und können es entsprechend korrigieren. So können Entwicklerinnen und Entwickler Architekturverstöße einfach direkt während der Programmierung korrigieren, denn es fällt kein Aufwand dafür an, die Verstöße explizit zu suchen und die passenden Quelltextzeilen heraus zu suchen.

7.2.2 Technische Realisierung des StyleBasedCheckers

Die Architektur des StyleBasedCheckers orientiert sich an den Berechnungsschritten der stilbasierten Architekturprüfung. Der StyleBasedChecker besteht aus vier Architekturelementen, die jeweils für einen Berechnungsschritt der stilbasierten Architekturprüfung zuständig sind. Abbildung 92 zeigt die Struktur des StyleBasedCheckers und seinen Zugriff auf Eingabeinformationen. Die Eingabeinformationen erhält der StyleBasedChecker über die zuoberst dargestellte Eclipse-API und die rechts dargestellt XML-Beschreibung des Architekturstils. Die blau ausgefüllten Kästchen repräsentieren die vier Architekturelemente des StyleBasedCheckers. Die Pfeile zeigen Beziehungen zwischen den dargestellten Architekturelementen, die Kommentare an den Pfeilen beschreiben, wozu die Beziehungen dienen.

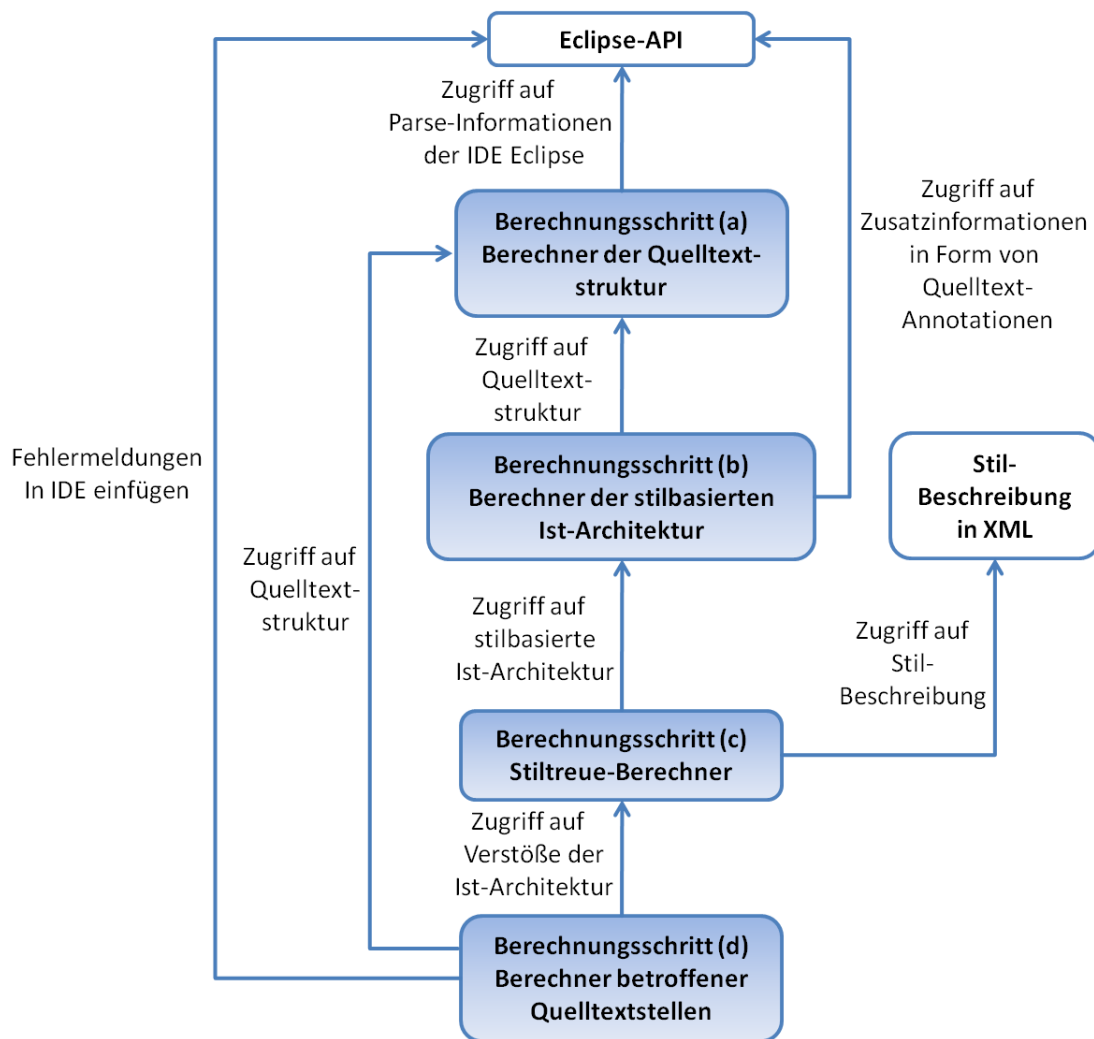


Abbildung 92: Aufbau des StyleBasedCheckers einschließlich des Zugriffs auf Eingabeinformationen

Das Werkzeug ist in die Eclipse-IDE über die API für externe Compiler eingefügt. Es wird von Eclipse informiert, wenn das Projekt kompiliert wird und kann an Eclipse anfragen, welche Quelltext-Ressourcen vom Kompilervorgang betroffen sind. Dann prüft es auf Stiltreue, indem es die im Folgenden beschriebenen vier Berechnungsschritte durchführt und meldet eventuelle Verstöße.

Berechnungsschritt (a): Berechner der Quelltextstruktur

Der Berechner der Quelltextstruktur (kurz: Quelltextstruktur-Berechner) greift auf die Parse-Informationen der Eclipse-IDE zu. Anhand dieser Informationen berechnet er die Quelltextstruktur. Diese wird beibehalten, bis eine neue Berechnung durchgeführt wird. Der Quelltextstruktur-Berechner iteriert über alle im Projekt deklarierten Klassen. Für jede Klasse sammelt er die ausgehenden Beziehungen und die Schnittstelle.

Berechnungsschritt (b): Berechner der stilbasierten Ist-Architektur

Direkt nach dem Berechner der Quelltextstruktur wird der Berechner der stilbasierten Ist-Architektur (kurz: Architektur-Berechner) aktiv. Er nutzt als Eingabeinformationen die Quelltextstruktur und die als Java-Annotationen festgehaltenen Zusatzinformationen. Die Java-Annotationen können direkt über das Eclipse-API als Teil der Parse-Informationen abgefragt werden. Der Architektur-Berechner iteriert über alle Quelltextelemente innerhalb der Quelltextstruktur und ermittelt über die Eclipse-API die jeweils annotierten Zusatzinformationen. Ergebnis ist die stilbasierte Ist-Architektur, diese bleibt gespeichert bis zum nachfolgenden Kompilervorgang.

Berechnungsschritt (c): Stiltreue-Berechner

Der Stiltreue-Berechner führt den Kern der stilbasierten Architekturprüfung durch: Er ermittelt Verstöße gegen den gewählten Architekturstil. Dafür greift er auf die im vorherigen Berechnungsschritt ermittelte stilbasierte Ist-Architektur zu. Die Vorgaben des Architekturstils erhält er einerseits über die von Benutzerinnen oder Benutzern erstellte XML-Beschreibung, andererseits über die ausprogrammierten Schnittstellenregeln, die Benutzerinnen und Benutzer über eine Erweiterungsschnittstelle in den Stiltreue-Berechner einfügen können. Für die XML-Beschreibung beinhaltet der StyleBasedChecker einen XML-Editor, der speziell auf Stilbeschreibungen ausgelegt ist. Ergebnis des Stiltreue-Berechners sind Listen von Architekturverstößen, für jede Regel des Stils berechnet er eine Liste.

Bei der Berechnung von Verstößen beachtet der StyleBasedChecker auch die geerbten Eigenschaften von Klassen. Dies betrifft alle im Projekt befindlichen Supertypen, nicht jedoch Typen importierter Bibliotheken oder der Sprache Java (wie beispielsweise Object). Dies hat sich als pragmatischer Ansatz für den Prototypen bewährt. Zukünftige Versionen des StyleBased-Checkers könnten bei Bedarf erlauben, dass Entwicklungsteams durch zusätzliche Annotationen deutlich machen, welche Supertypen beachtet werden sollen.

Berechnungsschritt (d): Berechner betroffener Quelltextstellen

Der Quelltextstellen-Berechner iteriert über die Verstoß-Listen des Stiltreue-Berechners. Für jeden Verstoß berechnet er die betroffenen Quelltextelemente. Er generiert Meldungen für die Benutzerinnen und Benutzer und verankert diese an den betroffenen Quelltextstellen. Dies geschieht über die Eclipse-API. Die Meldungen erhalten auf diese Weise ein den Benutzerinnen

und Benutzern vertrautes Aussehen, sie ähneln den von der IDE erstellten Meldungen über Kompilier-Fehler und -Warnungen.

7.3 Zusammenfassung

Die hier vorgestellte Umsetzung zeigt beispielhaft, wie sich die stilbasierte Architekturprüfung für eine konkrete Programmiersprache und ein Prüfwerkzeug umsetzen lässt. Das prototypische Prüfwerkzeug, der StyleBasedChecker, ist als Plugin für die Eclipse IDE realisiert. Er ist auf die Programmiersprache Java ausgerichtet. Der Architekturstil wird in einer XML-Syntax beschrieben. Für die Schnittstellenregeln bietet das Werkzeug eine Programmierschnittstelle, so dass sich die Schnittstellenregeln in Java ausdrücken lassen. Alle weiteren Regeln sowie die Architekturelement-Typen sind in der XML-Beschreibung des Architekturstils enthalten. Die Zusatzinformationen werden in Form von Java-Annotationen in den Quelltext eingefügt.

Mit Hilfe des StyleBasedCheckers wurden im Rahmen dieser Arbeit verschiedene Softwaresysteme beispielhaft auf ihre Stiltreue geprüft und so die Machbarkeit der stilbasierten Architekturprüfung demonstriert. Erkenntnisse aus diesen Prüfungen sind in diese Arbeit und in neuere Versionen des Werkzeugs eingeflossen. Auf diese beispielhaften Prüfungen wird im nachfolgenden Kapitel eingegangen.

8 Praktische Untersuchungen und Evaluationen

Für diese Arbeit wurden verschiedene praktische Untersuchungen an der Universität Hamburg und in dem privatwirtschaftlichen Unternehmen WPS Workplace Solutions GmbH durchgeführt und durch Interviews abgesichert.

Die Untersuchungen dienten dazu, den Ansatz der stilbasierten Architekturprüfung schrittweise zu entwickeln, zu evaluieren und seine Umsetzbarkeit zu zeigen.

An den Untersuchungen waren drei Studierende beteiligt, die im Rahmen der Forschungstätigkeiten dieser Promotion ihre Diplom- oder Studienarbeiten erstellten: Felix Abraham, Bettina Karstens und Arne Scharping (Abraham 2006; Karstens 2005; Scharping 2008).

Wie in Abschnitt 1.6 erläutert, wurde in dieser Arbeit nach der Methodik der Design-Forschung gearbeitet (March und Smith 1995; Hevner et al. 2004; Vaishnavi und Kuechler 2004; Winter 2008). Die im Rahmen von Design-Forschung behandelten Forschungsthemen sind *nicht* in erster Linie darauf ausgerichtet, die „Wahrheit“ im Sinne der Naturwissenschaften zu ermitteln (vergleiche Abschnitt 1.6). Stattdessen sollen *gute Lösungen* für *Probleme* entwickelt und schrittweise *verbessert* werden. Diese schrittweise Verbesserung wird in einem iterativen Prozess durchgeführt: Die Forschungstätigkeiten wechseln zwischen verschiedenen *Abstraktionsebenen* (Vaishnavi und Kuechler 2004) und zwischen *Entwurf* (build) und *Evaluation* (evaluate) (Hevner et al. 2004; Vaishnavi und Kuechler 2004).

Die Forschungsergebnisse sind die Resultate des Entwurfs und dienen als Lösungen für das betrachtete Problem. Aus diesem Grunde werden die Forschungsergebnisse auch als *Entwürfe* oder als *Lösungen* bezeichnet. Die von Design-Forschungsprojekten erzielten Ergebnisse können sich auf sehr unterschiedlichen Abstraktionsebenen befinden, die Spanne reicht nach March und Smith von abstrakten Konzepten bis hin zur Realisierung konkreter Artefakte (beispielsweise Softwaresysteme) (March und Smith 1995).

Die folgenden Abschnitte stellen die verschiedenen Abstraktionsebenen der im Rahmen dieser Arbeit erzielten Ergebnisse vor und erläutern deren Evaluation.

8.1 Einordnung der Forschungsergebnisse

In der als *Entwurf* bezeichneten Tätigkeit werden die Forschungsergebnisse erstellt. Im Englischen wird diese Tätigkeit als *build* bezeichnet (March und Smith 1995). Es sei explizit darauf hingewiesen, dass die Tätigkeit des Entwurfs auch die Konstruktion von Artefakten umfassen kann. Dies wird in der englischen Bezeichnung deutlicher als in der deutschen.

Da die Design-Forschung auf die schrittweise Verbesserung von Forschungsergebnissen setzt, produziert ein Forschungsprojekt mehrere Zwischenergebnisse. Im Folgenden wird der Begriff Forschungsergebnis so verwendet, dass er finale Ergebnisse und Zwischenergebnisse umfasst.

Ein Design-Forschungsprojekt erzielt üblicherweise fachlich zusammengehörige Forschungsergebnisse auf unterschiedlichen Abstraktionsebenen. March et al. kategorisieren Ergebnisse in Konzepte, Modelle, Methoden und innovative Exemplare/Ausprägungen (auch als Artefakte bezeichnet) (March und Smith 1995; Vaishnavi und Kuechler 2004).

Forschungsergebnisse auf abstrakter Ebene werden in Form exemplarischer Artefakte konkretisiert. Mit der Erstellung der Artefakte wird unter anderem die Umsetzbarkeit demonstriert. Die Konkretisierung kann über mehrere Abstraktionsebenen erfolgen (Vaishnavi und Kuechler 2004), beispielsweise von der abstrakten Ebene neuer Konzeptualisierungen über den Entwurf

einer auf diesen Konzeptualisierungen basierenden Methode bis zur Entwicklung beispielhafter Artefakte, welche den Einsatz der Methode unterstützen.

Wie sich die in der vorliegenden Arbeit erzielten Forschungsergebnisse der Kategorisierung nach March et al. Zuordnen lassen, wird in den nachfolgenden Abschnitten beschrieben.

8.1.1 Konzepte

Konzepte stellen nach March et al. das Wissen über eine Domäne dar. Die Kategorie der Konzepte umfasst *Konzeptualisierungen*, die sich als Vokabular für die Problem- und die Lösungsbeschreibung nutzen lassen. Konzeptualisierungen können informell beschrieben oder formalisiert werden, beispielsweise mit Entitäten, Beziehungen und Einschränkungen (constraints) (March und Smith 1995).

Die in den Kapiteln 2, 3 und 4 vorgestellte Konzeptarbeit über Architekturen, Architekturstile, über den Zusammenhang zwischen Quelltext, Quelltextstruktur und Softwarearchitekturen und über Architekturerosion und Architekturprüfungen lässt sich in die Kategorie der Konzepte einordnen.

Die in dieser Arbeit erstellten Konzeptualisierungen sind ebenfalls der Kategorie der Konzepte zuzuordnen. Dies umfasst die Definitionen 2-2, 3-1, 3-2 und 3-3 für Quelltextstrukturen, stilbasierte Architekturen und Architekturstile und die Berechnungsvorschriften für die verschiedenen, innerhalb von Architekturstilen unterscheidbaren Regeltypen (Kapitel 5 und 6). Die Konzeptualisierungen sind in dieser Arbeit sowohl informell beschrieben als auch in mengentheoretischer Notation formalisiert.

Nach dem Modellbegriff von Stachowiak handelt es sich bei den in dieser Arbeit vorgestellten Konzeptualisierungen für Quelltextstrukturen, stilbasierte Architekturen und Architekturstile um Modelle (Stachowiak 1973). Diese Modelle befinden sich auf einer Metaebene zu konkreten Quelltexten, stilbasierten Architekturen oder Architekturstilen.

Der Modellbegriff von March et al. entspricht nicht dem Modellbegriff von Stachowiak: Modelle nach Stachowiak werden bei March et al. als Konzept betrachtet, sofern sie Konzeptualisierungen und / oder ein Vokabular für eine Domäne darstellen. Beispielsweise kategorisieren March et al. das relationale Datenbankmodell als Konzept, nicht als Modell. Aus diesem Grunde sind die in dieser Arbeit erstellten Konzeptualisierungen an dieser Stelle unter Konzepten und nicht unter Modellen eingeordnet.

8.1.2 Modelle

Nach der Kategorisierung von March et al. sind Modelle innerhalb der Design-Forschung eine Menge von Aussagen über den Zusammenhang zwischen Problemen und Lösungen. Modelle können neue Lösungen enthalten. Sie beschreiben wie der Problem- und der Lösungsraum prinzipiell zusammenhängen. Ein Modell kann beispielsweise beschreiben, in welchen Problemsituationen welche Datenmodellierungsansätze geeignet sind und welche Notationen verwendet werden können (March und Smith 1995). Modelle können Konzepte und Konzeptualisierungen nutzen, so halten es auch die in dieser Arbeit vorgestellten Modelle, sie nutzen die oben genannten Konzeptualisierungen.

Mehrere Ergebnisse dieser Arbeit lassen sich in diesem Sinne als Modelle begreifen:

- Das Konzept zur Integration von Architekturinformationen und Quelltext. Hier lassen sich mehrere Lösungen einordnen:

- Die in Abschnitt 5.3.3 vorgestellte Lösung, dass alle Informationen über die Ist-Architektur stilbasierter Systeme *soweit möglich aus dem Quelltext entnommen* werden sollen: Diese Lösung begegnet dem Problem, dass externe Architekturdokumentationen und Quelltexte im Laufe der Zeit auseinander driften.
- Die in Abschnitt 5.3.3 vorgestellte Lösung, den Quelltext stilbasierter Softwaresysteme mit den für die Ermittlung der Ist-Architektur notwendigen *Zusatzinformationen zu annotieren* und die in Abschnitt 7.1.2 vorgestellte Konkretisierung dieser Lösung in Hinblick auf die Programmiersprache Java: Diese Lösungen begegnen zwei Problemen: dem genannten Problem, dass externe Architekturdokumentationen und Quelltexte im Laufe der Zeit auseinander laufen und dem Problem, dass Architekturinformationen direkt bei der Programmierung benötigt werden, aber üblicherweise nicht vollständig im Quelltext verfügbar sind.
- Die in Abschnitt 5.3.3 vorgestellte Lösung, dass die Zusatzinformationen in Sonderfällen Quelltext-extern gespeichert werden können: In diesem Fall müssen entsprechende Werkzeuge vorhanden sein, welche die Eingabe und Anzeige der Zusatzinformationen bei den zugehörigen Quelltextstellen ermöglichen. Diese Lösung begegnet dem Problem, dass manchmal Systeme geprüft werden sollen, ohne ihren Quelltext zu ändern.
- Lösungen für die Beschreibung von Architekturstilen:
 - Die in Abschnitt 7.1.3. vorgestellte, XML-basierte Beschreibungssprache für Architekturstile: Diese Lösung begegnet dem Problem, dass Architekturstile für die werkzeuggestützte Prüfung auf Stiltreue formal beschrieben werden müssen, und dem Problem, dass die Beschreibungssyntax möglichst leicht und schnell zu erlernen sein soll.
 - Die in Kapitel 7 vorgestellte Lösung, dass Schnittstellenregeln mit Hilfe kleiner Programme ausprogrammiert werden: Diese Lösung begegnet dem Problem, dass Schnittstellenregeln für die werkzeuggestützte Prüfung formal beschrieben werden müssen, und dass eine deklarative Beschreibung mit XML aufgrund des bisherigen Formalisierungsgrades von Schnittstellenregeln umständlich wäre.
- Lösungen für die Prüfung auf Stiltreue:
 - Sämtliche in den Kapiteln 5 und 6 vorgestellte Berechnungsverfahren zur Prüfung auf Stiltreue: Für jeden identifizierten Regeltyp wird ein Berechnungsverfahren vorgestellt. Diese Berechnungsverfahren begegnen dem Problem, dass stilbasierte Systeme unbemerkt gegen ihren Stil verstoßen können und so erodieren. Mit den Berechnungsverfahren lassen sich die Verstöße aufdecken.
 - Das Konzept, die stilbasierte Architekturprüfung so zu entwerfen, dass die Prüfung während der Programmierung erfolgen kann: Dies ist eine Lösung für das Problem, dass neue Verstöße während der Programmierung entstehen können, ohne dass die Programmiererin oder der Programmierer sie bemerkt. Und es ist eine Lösung für das Problem, dass bestehende Verstöße während der Programmierung nicht bekannt sind und deswegen nicht behoben werden.

8.1.3 Methoden

Methoden nach March et al. sind eine Menge von Schritten, die sich für eine Aufgabe nutzen lassen. Die Schritte können Algorithmen oder Anleitungen für menschliches Handeln sein. Methoden nach March et al nutzen Konzepte und Modelle.

Die in Kapitel 5 beschriebenen Schritte zur Verwendung der stilbasierten Architekturprüfung lassen sich als Methode verstehen. Sie besagen unter anderem, dass vor der ersten Prüfung der Architekturstil beschrieben werden muss, dass die Zusatzinformationen möglichst bei der Programmierung in den Quelltext integriert werden sollten, und dass die Prüfung nur erfolgen kann, wenn zuvor beide anderen Schritte (Stilbeschreibung und Zusatzinformationen integrieren) erfolgt sind.

Der Fokus dieser Arbeit liegt nicht auf der Erstellung von schrittweisen Anleitungen, wie bei den Methoden im Sinne von March et al. Stattdessen ist die stilbasierte Architekturprüfung möglichst so gestaltet, dass sie sich in verschiedenen Projektkonstellationen mit unterschiedlichen Vorgehensmodellen und Projektrollen einsetzen lässt. Aus diesem Grunde macht die stilbasierte Architekturprüfung bezüglich der Vorgehensmodelle und Rollen keine Vorgaben oder Einschränkungen.

8.1.4 Exemplare

Exemplare sind beispielhafte Umsetzungen für Konzeptualisierungen, Modelle und Methoden. Im Rahmen dieser Arbeit wurden verschiedene Exemplare erstellt:

- Die Formalisierung dreier Architekturstile in Typen und Regeln (WAM, Quasar und ein projekteigener Stil) mit Hilfe der gegebenen Konzeptualisierung für Stile. Der WAM- und der Quasar-Stil wurden in Abschnitt 3.3 vorgestellt, die Listen der Typen und Regeln finden sich im Anhang. Der projekteigene Stil ist in den Arbeiten von Scharping beschrieben (Scharping 2005, 2008), dessen Diplomarbeit im Rahmen der Forschungstätigkeiten für diese Arbeit entstand.
- Die Beschreibung zweier Stile (WAM und der projekteigene Stil) mit Hilfe der vorgeschlagenen XML-Notation und mit Hilfe der Programmierung von Schnittstellenregeln.
- Die Annotation bestehender Java-Softwaresysteme um Zusatzinformationen.
- Mehrere Prüfwerkzeuge, die auf den erarbeiteten Konzepten, Modellen und Methoden aufsetzen.

8.2 Evaluationsprinzipien

In der Design-Forschung können die Forschungsergebnisse aller Abstraktionsebenen (Konzepte, Modelle, Methoden und Exemplare) evaluiert werden (March und Smith 1995; Hevner et al. 2004). Die *Kriterien* für die Evaluation orientieren sich an der Problemstellung und an der Umgebung, in der das Problem beobachtet wurde (Hevner et al. 2004). Die Problemumgebung kann menschliche Rollen und Fähigkeiten, Organisationsstrategien und -kulturen und Technologien wie verwendete Programmiersprachen und die technische Infrastruktur umfassen. Abbildung 93 zeigt, wie die Forschungstätigkeiten (einschließlich der Evaluation) in der Design-Forschung mit der Problemumgebung und mit der Wissensbasis des Forschungsfeldes interagieren.

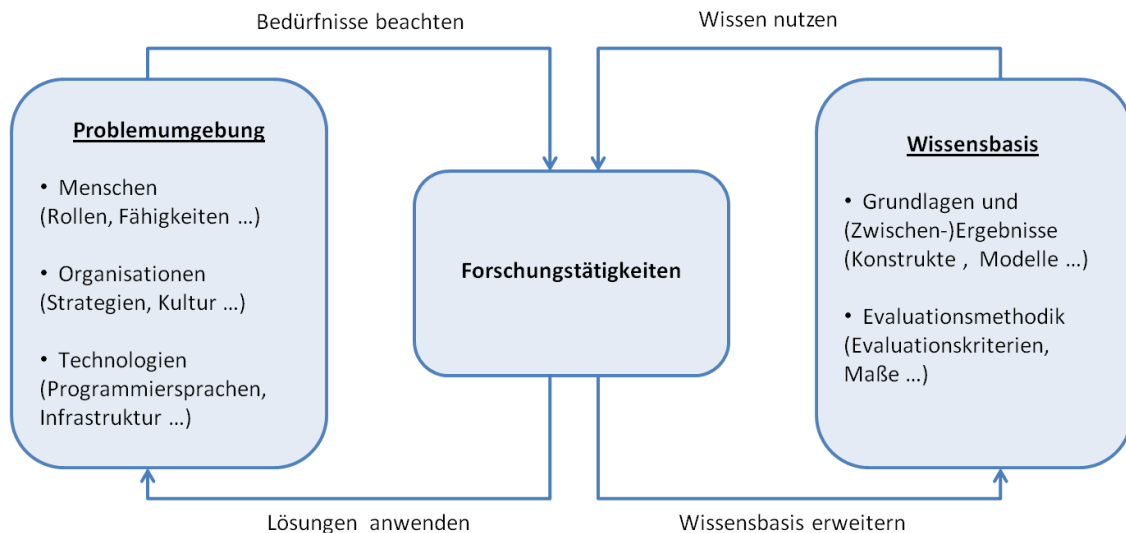


Abbildung 93: Interaktion der Design-Forschung mit der Problemumgebung und der Wissensbasis nach Hevner et al. (Hevner et al. 2004)

Grundsätzlich lassen sich in der Literatur über die Design-Forschung zwei Evaluationstypen identifizieren (March und Smith 1995; Hevner et al. 2004; Vaishnavi und Kuechler 2004; Winter 2008). Diese Typen sollen in dieser Arbeit explizit unterschieden werden:

Definition 7-1: Evaluation mittels exemplarischer Artefakte

Integraler Bestandteil der Design-Forschung ist, dass Konzepte, Modelle und Methoden hinsichtlich ihrer *Umsetzbarkeit* (auch als *Machbarkeit*⁴⁰ bezeichnet) evaluiert werden, indem man darauf aufsetzende, beispielhafte Artefakte konstruiert. Dabei ist die Problemumgebung zu beachten. Dieser Evaluationstyp ist zentral für die Design-Forschung: mit Hilfe beispielhafter Umsetzungen wird das Forschungsergebnis schrittweise verbessert.

Definition 7-2: Direkte Evaluation

Zusätzlich lassen sich Forschungsergebnisse aller Kategorien direkt evaluieren, hinsichtlich verschiedener, Forschungsprojekt-spezifischer Kriterien. Die Evaluationsmethodik ist bewusst offen gelassen, sie kann je nach Evaluationsziel beispielsweise Interviews umfassen, den beispielhaften Einsatz in der Praxis oder in der technischen Zielumgebung, kontrollierte Experimente oder mathematische Auswertungen. In der Regel werden beispielhafte Artefakte direkt evaluiert, bei Bedarf werden zusätzlich Forschungsergebnisse anderer Kategorien evaluiert.

⁴⁰ Es sei darauf hingewiesen, dass die Machbarkeitsnachweise (Umsetzungen) keine Beweise der Allgemeingültigkeit der Lösungen darstellen. Dies ist in der Design-Forschung kaum möglich, da diese üblicherweise nach Lösungen sucht, die menschliches Handeln unterstützen. Die Machbarkeitsnachweise stärken durch geeignete Auswahl der beispielhaften Exemplare das Vertrauen in die Lösungen und dienen der schrittweisen Verbesserung der Lösungen (March und Smith 1995; Hevner et al. 2004; Vaishnavi und Kuechler 2004). So sind auch die Machbarkeitsnachweise dieser Arbeit zu verstehen. Zur Relevanz und Verallgemeinerbarkeit von Ergebnissen beispielhafter Fallstudien siehe auch (Flyvbjerg 2006).

Evaluationen im Rahmen der Design-Forschung umfassen meist beide Evaluationstypen: zuerst wird ein exemplarisches Artefakt erstellt, um die Umsetzbarkeit eines Konzepts, Modells oder einer Methode zu zeigen. Anschließend wird das Artefakt direkt evaluiert.

Die Evaluationsergebnisse beider Evaluationstypen liefern Erkenntnisse über das evaluierte Ergebnis und über weitere Ergebnisse auf derselben und auf anderen Abstraktionsebenen, sofern diese Ergebnisse mit dem evaluierten Ergebnis fachlich zusammenhängen (March und Smith 1995; Hevner et al. 2004; Vaishnavi und Kuechler 2004).

In der *finalen Iteration* eines Forschungsprojektes wird mittels der Evaluationsergebnisse gezeigt, dass das Forschungsergebnis das betrachtete Problem ausreichend klärt und dabei die Problemumgebung berücksichtigt. Ferner wird der Forschungsbedarf für weitere Forschungsprojekte bestimmt. Alternativ kann die Evaluation zeigen, dass die Lösung noch nicht ausreicht und weitere Forschungszyklen notwendig sind.

Die Forschungsergebnisse *einer nicht-finalen* Iteration wirken sich auf die nachfolgende Iteration aus: die bisherigen Forschungsergebnisse werden anhand der Evaluationsergebnisse überarbeitet oder durch neue Entwürfe ersetzt. Ferner können die Forschungsergebnisse zu einem verbesserten oder korrigierten Problemverständnis führen. In diesen Fällen werden bei Bedarf die Evaluationskriterien für den nachfolgenden Zyklus angepasst (Vaishnavi und Kuechler 2004). Abbildung 94 zeigt die Details der iterativen Forschungstätigkeiten in der Design-Forschung.

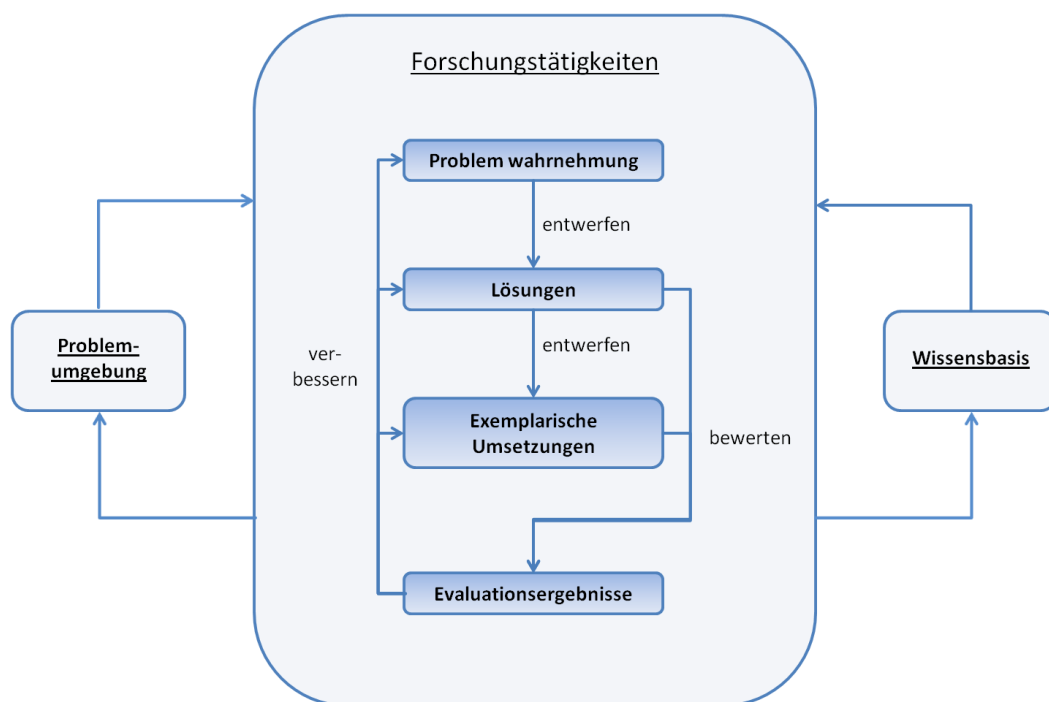


Abbildung 94: Iterative Forschungstätigkeiten in der Design-Forschung
(in Anlehnung an Vaishnavi und Kuechler 2004; Hevner et al. 2004)

Im Rahmen dieser Arbeit wurden drei Forschungszyklen durchgeführt, in jedem Zyklus wurden Konzeptualisierungen, Modelle, Methoden und Exemplare erstellt oder anhand des vorherigen Forschungszykluses überarbeitet, und es wurde das Problemverständnis vertieft. In jedem Zyklus wurden mehrere praktische Untersuchungen durchgeführt.

Es würde den Rahmen dieser Arbeit sprengen, sämtliche erzielte und verworfene Zwischenergebnisse, die Details der Evaluationen und der jeweiligen Evaluationskriterien vorzustellen. Aus diesem Grunde wird im Folgenden beispielhaft die Evaluation der Konzeptualisierung für Architekturstile ausführlich vorgestellt. Anschließend wird ein Überblick über die weiteren Untersuchungen sowie interessante Zwischenergebnisse gegeben.

8.3 Evaluation der Architekturstil-Konzeptualisierung

Die Konzeptualisierung für Architekturstile findet sich in den Definitionen 3-1 und 3-2, sowie in den Regel-Berechnungsvorschriften in den Kapiteln 5 und 6. Die Umsetzbarkeit der Konzeptualisierung wurde unter anderem anhand beispielhafter Stilbeschreibungen evaluiert (Evaluationstyp „Evaluation mittels exemplarischer Artefakte“, siehe Definition 7-1). Diese Evaluation wird im Folgenden exemplarisch vorgestellt.

Die zur Evaluation genutzten Stilbeschreibungen stellen exemplarische Artefakte im Sinne der Design-Forschung dar. Die Beschreibungen wurden mit Hilfe von Interviews und anhand einer Fallstudie evaluiert (Evaluationstyp „direkte Evaluation“, siehe Definition 7-2).

Die Architekturstil-Konzeptualisierung wurde schrittweise konkretisiert, bis hin zu den exemplarischen Artefakten. Folgende Forschungsergebnisse stellen Zwischenschritte der Konkretisierung dar:

- Die XML-basierte Beschreibungssprache von Architekturstilen (siehe, Abschnitt 8.1). Diese Beschreibungssprache basiert auf der Konzeptualisierung von Architekturstilen. Im Sinne der Design-Forschung handelt es sich bei dem Forschungsergebnis „Beschreibungssprache“ um eine Konkretisierung des Forschungsergebnisses „Konzeptualisierung“.
- Die in Abschnitt 3.3 vorgestellte und auch für die Regellisten im Anhang genutzte, strukturierte Darstellung von Architekturstil-Regeln, welche sich ein-eindeutig in die XML-basierte Beschreibungsmöglichkeit übersetzen lässt.

Basierend auf diesen Zwischenergebnissen wurden in dieser Untersuchung sechs exemplarische Artefakte erstellt, welche die Architekturstil-Konzeptualisierung beispielhaft umsetzen:

- Jeweils eine Beschreibung der drei Architekturstile WAM, Quasar und des projekt-eigenen Stil mittels der XML-basierten Beschreibungssprache (siehe Abschnitt 7.1.3).
- Und jeweils eine Beschreibung der drei Stile mittels der strukturierten Darstellung von Architekturstil-Regeln (siehe Abschnitt 3.3 und Anhang).

8.3.1 Evaluationskriterien

Die Umsetzung sollte zeigen, dass die Konzeptualisierung adäquat von konkreten Architekturstilen abstrahiert. Hierfür wurde darauf geachtet, dass die in der Konzeptualisierung vorkommenden Konstrukte (Architekturelement-Typen, Beziehungstypen, Regeltypen) in den beispielhaft betrachteten Architekturstilen vorkommen, und dass – in der umgekehrten Richtung – alle für Konformanzprüfungen relevanten Bestandteile der Architekturstile in der Konzeptualisierung vorhanden sind.

Ferner sollte geprüft werden, ob sich in Stilen gleichartige Regeln finden, so dass es möglich ist, eine Konzeptualisierung für Architekturstile zu erstellen, welche die Berechnungsvorschriften für gleichartige Regeln umfasst.

Bei der Umsetzung wurden unter anderem folgende Aspekte der Problemstellung und der Problemumgebung beachtet:

- Die Architekturstil-Konzeptualisierung zielt darauf, statische Konformanzprüfungen der Ist-Architektur durchzuführen. Somit sind ausschließlich die Anteile von Architekturstilen relevant, welche sich auf die statische Struktur von Ist-Architekturen beziehen.
- Die Konzeptualisierung sollte für praxisrelevante Stile geeignet sein. Dieses Kriterium hatte Einfluss auf die Auswahl der beispielhaften Stile.
- Die Konzeptualisierung sollte dem Stilverständnis der Projektmitglieder in realen, stilbasierten Projekten entsprechen. Das bedeutet, dass die Konzeptualisierung dieselben Konzepte enthält, welche die Projektmitglieder bisher für Architekturstile verwendeten. Die Beschreibung sollte als leichtgewichtig empfunden werden.

8.3.2 Vorgehen

Allgemeines Vorgehen

In allen Forschungsiterationen wurde die Konzeptualisierung für Stile bearbeitet, es wurden beispielhafte Stile beschrieben und die Ergebnisse anhand von Interviews mit Projektmitgliedern, Architektinnen und Architekten und durch eine Fallstudie abgesichert. Als Ergebnis der Iterationen wurde festgehalten, welche Aspekte der Konzeptualisierung bereits den Evaluationskriterien entsprechen und in welcher Hinsicht die Konzeptualisierung bei Bedarf noch ergänzt oder überarbeitet werden musste. Die Evaluationsergebnisse flossen in die nachfolgende Iteration ein. Die Schlussiteration zeigte keinen weiteren Überarbeitungsbedarf für die Konzeptualisierung.

Konkretes Vorgehen

Besonderheiten der ersten Forschungsiteration: In der ersten Forschungsiteration wurde die Konzeptualisierung anhand von Literaturrecherchen erstellt und es wurden die zu untersuchenden, beispielhaften Stile ausgewählt: WAM, Quasar und ein projekteigener Stil. Die Auswahl der Stile wird im folgenden Abschnitt erläutert. In der ersten Iteration wurde der WAM-Stil mit Hilfe von strukturierten Regellisten beschrieben. Für den Quasar-Stil wurde eine Fallstudie erstellt (Abraham 2006).

Besonderheiten der zweiten und dritten Forschungsiteration: die Konzeptualisierung wurde jeweils anhand der Ergebnisse der jeweils vorherigen Iteration überarbeitet und durch die beispielhafte Beschreibung von Stilen überprüft. In der zweiten Iteration kam der projekteigene Stil hinzu, in der letzten Iteration diente zusätzlich der Quasar-Stil zur finalen Evaluation. In beiden Iterationen wurde – zusätzlich zur Beschreibung in Form von Regellisten – eine Beschreibung mit der in dieser Arbeit vorgestellten XML-basierten Beschreibungssprache erstellt. Die Beschreibungen aller drei Stile wurden durch Interviews abgesichert.

Primäre Quelle für die WAM- und Quasar-Regeln waren die in der Literatur verfügbaren Beschreibungen der Stile (Züllighoven 2005; Siedersleben 2003a, 2003b, 2004). Aus diesen schriftsprachlichen Beschreibungen wurden die Regeln extrahiert.

Eine erste Version der Regeln des WAM-Stils wurde in der ersten Iteration gemeinsam mit Karstens erarbeitet (Karstens 2005). Bei den im Anhang dieser Arbeit gelisteten Regeln des WAM-Stils handelt es sich um eine von der Autorin überarbeitete Version, basierend auf Untersuchungen der letzten Forschungsiterationen. Die Regeln des Quasar-Stils wurden von der Autorin ermittelt und durch ein Interview und eine Fallstudie (Abraham 2006) abgesichert. Die Regeln des projekteigenen Stils wurden anhand der Projektdokumentation und der eigenen Erfahrung als Projektmitglied von Scharping im Rahmen seiner Diplomarbeit ermittelt und durch Feedback anderer Projektmitglieder abgesichert (Scharping 2008).

Auswahl der beispielhaften Stile

Die Stile wurden nach verschiedenen Kriterien ausgewählt, dabei sollte jedes Kriterium von mindestens einem Stil erfüllt werden:

- Die Stile sollten praxisrelevant sein,
- sie sollten in Softwareprojekten eingesetzt worden sein,
- sie sollten über mehrere Jahre gereift sein, für verschiedene Softwaresysteme genutzt worden sein und eine bedeutende Rolle in den die Stile nutzenden Organisationen spielen.
- Für die Stile sollten in der Literatur umfangreiche Beschreibungen zur Verfügung stehen, die für die Extraktion der Regeln geeignet sind.
- Es sollten Personen verfügbar sein, die über Expertise in den ausgewählten Stilen verfügen, wie Projektmitglieder, die Systeme mit den ausgewählten Stilen entwickelt haben, oder Softwarearchitektinnen oder -architekten, die Softwaresysteme der ausgewählten Stile entworfen oder die ausgewählten Stile entwickelt haben.

Tabelle 8-1 zeigt, welche Kriterien von welchen der drei Stile erfüllt wurden. Der WAM- und der Quasar-Stil erfüllten alle Kriterien und waren somit optimal geeignet. Der projekteigene Stil erfüllt alle Kriterien bis auf die jahrelange Reife und die bedeutende Rolle in einer Organisation. Da projekteigene Stile nicht über Projektgrenzen kommuniziert werden müssen, ist üblicherweise keine vollständige Beschreibung verfügbar. Dies wurde ausgeglichen durch die Architekturdokumentationen des Projekts sowie durch die Verfügbarkeit von Projektmitgliedern und Architekten des Stils.

Kriterien	Quasar-Stil	WAM-Stil	Projekteigener Stil
In mindestens einem Projekte eingesetzt	X	X	X
Über Jahre in vielen Projekten gereift	X	X	-
Bedeutende Rolle in Organisation	X	X	-
Beschreibungen verfügbar	X	X	(X)
Projektmitglieder verfügbar	X	X	X
Architektinnen / Architekten verfügbar	X	X	X

Tabelle 8-1: Erfüllung der Auswahlkriterien durch die betrachteten Stile

8.3.3 Ergebnisse

Die Konzeptualisierung für Architekturstile, welche initial basierend auf umfangreicher Literaturarbeit entstand, wurde mit Hilfe beispielhafter Umsetzungen schrittweise verbessert und evaluiert. Die Evaluation der letzten Iteration zeigt anhand dreier, unter Beachtung der Problemumgebung ausgewählter, praxisrelevanter Architekturstile, dass die Konzeptualisierung für die Beschreibung der drei Stile geeignet ist, d.h. für die Beschreibung ihrer Typen und Regeln. Anders formuliert: Es wurde gezeigt, dass die Konzeptualisierung adäquat von diesen drei exemplarisch ausgewählten Stilen abstrahiert.

Die resultierende Konzeptualisierung findet sich in den Definitionen 3-1 und 3-2 sowie in den Berechnungsvorschriften für die verschiedenen Regeltypen in den Kapiteln 5 und 6. Die Konzeptualisierung bildet das Herz der stilbasierten Architekturprüfung.

Die in dieser Untersuchung erstellten Zwischenergebnisse und Artefakte werden in folgenden Kapiteln und Veröffentlichungen vorgestellt: Die Beschreibungstechnik mit strukturierten Regellisten und ein Überblick über WAM und Quasar findet sich in Abschnitt 3.3. Die XML-basierte Beschreibungssyntax erläutert Abschnitt 7.1.3. Die in der ersten Forschungsiteration ermittelten Regeln des WAM-Stils sind in der Diplomarbeit von Karstens gelistet (Karstens

2005). Die in der finalen Iteration erstellten Beschreibungen des WAM- und des Quasar-Stils finden sich im Anhang dieser Arbeit. Die Quasar-Fallstudie wurde in der Studienarbeit von Abraham erstellt (Abraham 2006). Die Regeln des projekteigenen Stils sind in der Diplomarbeit von Scharping veröffentlicht (Scharping 2008).

Ein Beispiel für ein im Rahmen dieser Untersuchungen verworfenes Zwischenergebnis war der Ansatz, alle Regeltypen mit einer mengentheoretischen Notation und den dort üblichen prädikatenlogischen Ausdrücken zu beschreiben. Dies war für fast alle Regeltypen gut geeignet, stellte sich jedoch bei Schnittstellenregeln als nicht praxistauglich heraus, da für Schnittstellenregeln sehr umfangreiche, als schwergewichtig und unleserlich empfundene prädikatenlogische Ausdrücke nötig waren.

Ein Beispiel für Forschungsergebnisse, die sich im Laufe der drei Zyklen verfeinerten und konkretisierten, sind die Berechnungsvorschriften für die verschiedenen Regeltypen (siehe Kapitel 5 und 6).

8.4 Weitere Evaluationen

Die weiteren Untersuchungen und Evaluationen im Rahmen dieser Arbeit wurden nach einem ähnlichen Muster durchgeführt. Im Rahmen der Untersuchungen wurden verschiedene exemplarische Artefakte erstellt. Da sich nicht alle Untersuchungen an dieser Stelle ausführlich darstellen lassen, werden im Folgenden mehrere in dieser Arbeit erstellte exemplarische Artefakte kurz erläutert und ausgewählte, interessante Ergebnisse der Untersuchungen vorgestellt.

8.4.1 Annotation beispielhafter Softwaresysteme

Vorgehen im Überblick

Sechs Softwaresysteme wurden beispielhaft mit den in Abschnitt 5.3 erläuterten Zusatzinformationen annotiert, mittels der in Abschnitt 7.1.2 vorgestellten Java-Annotationen.

Drei der Systeme sind kommerzielle, in der Praxis eingesetzte Individualentwicklungen, in den Bereichen Behörde, Fahrzeugvermietung und Gesundheit. Die anderen drei Systeme sind ein Planer für die Pausenaufsicht in Schulen, der in einem einjährigen Studentenprojekt entwickelt wurde, ein frei verfügbares Konstruktionsbeispiel für das objektorientierte Konstruktionshandbuch (Züllighoven 2005) und eine universitäre Kommunikationsplattform. Alle sechs Systeme wurden in Java entwickelt. Tabelle 8-2 listet die verwendeten Architekturstile sowie den Erfahrungsgrad des Entwicklungsteams.

System	Architekturstil	Anfänger / Experten
Konstruktionsbeispiel	WAM	Gemischt
Behörde	WAM	Experten
Fahrzeugvermietung	WAM	Experten
Schule	WAM	Anfänger
Gesundheit	WAM	Experten
Kommunikation	Projekteigener Stil	Gemischt

Tabelle 8-2: Softwaresysteme: Architekturstil und Erfahrungsgrad des Projektteams

Tabelle 8-3 zeigt einen Überblick über die Größenordnung der Systeme. LOC (lines of code) bezeichnet die Anzahl der Quelltextzeilen in den Projekten. Top-Level-Typen sind nicht geschachtelte Klassen, Interfaces, Enumerationen und Annotationen. Die Kennzahlen wurden mit Hilfe des Architekturanalysewerkzeugs Sotograph ermittelt. Bei dem System Fahrzeugvermietung lagen die Testklassen nicht vor. Weitere Kennzahlen der Softwaresysteme finden sich im Anhang.

System	Quelltextzeilen (LOC)	Top-Level-Typen
Konstruktionsbeispiel	43.279	235
Behörde	110.611	619
Fahrzeugvermietung	131.668	447+Testklassen
Schule	27.618	112
Gesundheit	146.817	566
Kommunikation	47.565	222

Tabelle 8-3: Kennzahlen der Systeme

Ausgewählte Ergebnisse

In allen sechs Systemen waren die Zusatzinformationen bereits informell im Quelltext markiert, beispielsweise durch Kommentare, Namensmuster oder Marker-Interfaces (siehe Abschnitt 5.3.3). Diese Markierungen dienten erfolgreich als Ausgangspunkt für die Annotationen. Interviews bestätigten die Korrektheit der Annotationen.

Dass die Zusatzinformationen in den Systemen bereits informell markiert waren, stützt die These dieser Arbeit, dass durch die Java-Annotationen kein Zusatzaufwand entsteht, sofern sie von Anfang an in Projekten verwendet werden (anstelle der genannten, bisher verwendeten, informellen Markierungen).

Wird ein bereits bestehendes System geprüft, so kann eine mit dem System vertraute Person die Annotationen in kurzer Zeit ergänzen. Dies wurde im Rahmen dieser Arbeit beispielhaft für das System aus dem medizinischen Bereich gezeigt, hier wurden die Zusatzinformationen in 45 Minuten durch ein Projektmitglied korrekt ergänzt.

8.4.2 Implementierung prototypischer Werkzeuge

Vorgehen im Überblick

Es wurden zwei prototypische Werkzeuge für die Prüfung auf Stiltreue entwickelt.

- In der ersten Forschungsiteration programmierte Karstens einen Prototyp, indem sie ein bestehendes Werkzeug zur Architekturprüfung – den Sotographen (Bischofberger et al. 2004) – um zusätzliche Funktionalität erweiterte. Der Sotograph speichert die Quelltextstruktur zu untersuchender Systeme in einer relationalen Datenbank. Karstens nutzte die Erweiterungsschnittstelle des Sotographen für benutzerdefinierte Queries auf dieser Datenbank in einer SQL-artigen Notation. Die Ergebnisse der Untersuchungen finden sich – zusätzlich zu den Beschreibungen in dieser Arbeit – in Karstens' Diplomarbeit (Karstens 2005) sowie in einer gemeinsamen Veröffentlichung der Autorin mit Karstens und Lilienthal (Becker-Pechau et al. 2006).
- In der zweiten Iteration wurde ein eigenes Werkzeug, der in Abschnitt 7.2 vorgestellte StyleBasedChecker, erstellt. Dabei wurde Wert darauf gelegt, dass das Werkzeug – im Gegensatz zu der Sotograph-Erweiterung – in eine IDE integriert ist, so dass sich die Regeln zur Entwicklungszeit prüfen lassen, und zwar automatisch bei jedem Kompiliervorgang. Die erste Version dieses Werkzeugs erstellte Scharping im Rahmen seiner Diplomarbeit (Scharping 2008). Die in dieser Arbeit vorgestellte Version, welche die finale Fassung der stilbasierten Architekturprüfung unterstützt, wurde von der Autorin in der dritten Forschungsiteration erstellt.

Ausgewählte Ergebnisse

Durch die Konstruktion dieser Artefakte wurde unter anderem gezeigt, dass die Konzepte und Berechnungsregeln der stilbasierten Architekturprüfung in Form von Werkzeugen praktisch umsetzbar sind.

Mit der Implementierung des StyleBasedCheckers als Plugin für die Eclipse-IDE wurde gezeigt, dass es möglich ist, ein Werkzeug zur stilbasierten Architekturprüfung so in eine IDE zu integrieren, dass bei jedem Kompilervorgang automatisch auf Stiltreue geprüft wird, so dass Verstöße gegen den Stil unmittelbar während der Eingabe und Bearbeitung des Quelltextes angezeigt werden (siehe Abschnitt 7.2). Damit wurde ein wichtiges Ziel dieser Arbeit erreicht.

8.4.3 Prüfungen auf Stiltreue

Vorgehen im Überblick

Mit Hilfe der prototypischen Werkzeuge wurden alle sechs genannten Softwaresysteme auf Stiltreue geprüft. Die meisten Systeme wurden mehrfach, mit verschiedenen Versionen der prototypischen Werkzeuge geprüft. Die Prüfergebnisse wurden bei Bedarf in Gesprächen und Interviews mit Projektmitgliedern sowie Architekten des jeweiligen Architekturstils abgesichert.

Die Untersuchungen wurden von der Autorin sowie von den Studierenden Karstens und Scharping durchgeführt, die im Rahmen der Forschungen für diese Arbeit ihre von der Autorin betreuten Diplomarbeiten erstellt haben.

Die Prüfungsvorgänge der Autorin wurden durch Tagebucheinträge dokumentiert. Hierbei handelt es sich um eine Forschungstechnik aus den Sozialwissenschaften, mit der die unmittelbaren Erfahrungen und Ergebnisse der Forschungstätigkeit gesichert werden. Das Führen von Forschungstagebüchern reduziert die Gefahr, dass Forschungsergebnisse vergessen werden, und dass nachträglich Interpretationen der forschenden Person die Ergebnisse verfälschen (Bortz und Döring 2006). Den Einsatz von Tagebüchern für Studien innerhalb der Software-Entwicklung thematisiert Naur (Naur 1983; Jepsen et al. 1989). Auch die Tagebucheinträge lassen sich als Artefakte im Sinne der Design-Forschung verstehen.

Ausgewählte Ergebnisse

Es wurden unter anderem folgende Ergebnisse erzielt:

- Die Untersuchungen konnten die Relevanz der stilbasierten Architekturprüfung untermauern, indem gezeigt wurde, dass im Einsatz befindliche, stilbasierte Softwaresysteme tatsächlich gegen die Regeln ihres Stils verstoßen. Alle untersuchten Systeme wiesen Verstöße auf. Die gefunden Verstöße waren – bis auf wenige Einzelfälle – bisher unentdeckt geblieben.
- Auch für Expertensysteme erwies sich die Prüfung auf Stiltreue als relevant. Tabelle 8-4 zeigt beispielhaft eine Zusammenfassung der Untersuchungsergebnisse für ein Experten- und ein Anfängersystem. Die Tabelle stellt die Gesamtanzahl der von allen Regelverstößen betroffenen Quelltextstellen dar (dort kurz als Quelltextfehler bezeichnet). Ferner zeigt die Tabelle das Verhältnis der Quelltextfehler zur Systemgröße, gemessen an der Anzahl der Top-Level-Typen.
- In den Untersuchungen wurde deutlich: sofern die prüfenden Person den Architekturstil kennt, kann sie sowohl Systeme prüfen, die ihr als Projektmitglied bekannt sind, als auch unbekannte Systeme.
- Für die finale Version der stilbasierten Architekturprüfung ließ sich untermauern, dass die Berechnungsvorschriften korrekt sind, da weder falsch positive noch falsch negative

Funde auftraten. Ein falsch positiver Fund liegt vor, wenn die Prüfung einen Verstoß meldet, obgleich keiner vorliegt. Um falsch positive Funde auszuschließen, wurden die gefunden Verstöße Teammitgliedern zur Prüfung vorgelegt. Ein falsch negativer Fund liegt vor, wenn die Prüfung einen bestehenden Verstoß übersieht. Um falsch negative Funde auszuschließen wurde in gründlichen Quelltext-Reviews nach Verstößen gesucht. Die Reviews durch Teammitglieder stärken das Vertrauen in die finale Version der stilbasierten Architekturprüfung, wie bei der Design-Forschung üblich (Vaishnavi und Kuechler 2004). Absolute Sicherheit, dass die Teammitglieder keine Fehler übersehen haben, kann es selbstverständlich nicht geben.

- Es ließ sich erfolgreich mit mehreren Systemen desselben Stils bestätigen, dass die XML-basierten Stilbeschreibungen sowie die Implementierungen der Schnittstellenregeln im StyleBasedChecker nur einmal erstellt zu werden brauchen. Für alle weiteren Systeme desselben Stils ließen sie sich, wie angestrebt, unverändert wiederverwenden. Dieses Ziel wurde erreicht, indem die Beschreibung des Architekturstils von den Zusatzinformationen getrennt wurde. Während der Stil über alle untersuchten Systeme desselben Stils stabil war, sind die Zusatzinformationen grundsätzlich systemspezifisch, da sie die Abbildung zwischen dem konkreten Quelltext und der Architektur beschreiben (siehe Abschnitt 5.3). In der prototypischen Umsetzung ist die Übertragbarkeit auf Java-Systeme beschränkt, da der StyleBasedChecker ausschließlich mit der Programmiersprache Java arbeitet. Die XML-basierte Beschreibung, welche alle Architektur-element-Typen und alle Regeln außer den Schnittstellenregeln umfasst, ist sprachunabhängig und somit mit entsprechender Erweiterung des StyleBasedCheckers prinzipiell auch für andere Programmiersprachen verwendbar. Einzige Ausnahme bilden die Schnittstellenregeln, sie verwenden den Syntax-Baum des Java-Compilers. Mit Hilfe sprachunabhängiger Metamodelle für Parse-Informationen von Softwaresystemen, wie sie im Reverse-Engineering eingesetzt werden (Trifu und Szulman 2005; Becker et al. 2010), ließen sich die Schnittstellenregeln jedoch auch sprachunabhängig realisieren.
- Durch die Untersuchung von Systemen zweier verschiedener Stile (WAM-Stil und projekteigener Stil) ließ sich zeigen, dass der Prüfansatz für mehrere Stile nutzbar ist. Zwar wurden im Rahmen dieser Arbeit aus Zeitgründen keine Systeme des Quasar-Stils praktisch geprüft, aber die in der oben beschriebenen Untersuchung vorgestellte Beschreibung des Quasar-Stils, welche durch eine Fallstudie und ein Interview abgesichert wurde, zeigt, dass sich der Quasar-Stil anhand derselben Regeltypen formalisieren lässt wie WAM und wie der projekteigene Stil. Diese Regeltypen werden durch die prototypischen Werkzeuge unterstützt. Somit ist es naheliegend, dass auch Systeme des Quasar-Stils mit diesen Werkzeugen erfolgreich geprüft werden können. Dies kann in anschließenden Forschungstätigkeiten evaluiert werden.

System	Anfänger / Experten	Quelltextzeilen (LOC)	Top-Level-Typen	Von Verstößen betroffene Quelltextstellen (kurz: Quelltextfehler)	Quelltextfehler pro Top-Level-Typ (Durchschnitt)
Behörde	Experten	110.611	619	64	0,10
Schule	Anfänger	27.618	112	522	4,28

Tabelle 8-4: Vergleich Anfängersystem / Expertensystem

Im Anhang der Arbeit finden sich Tabellen mit detaillierten Untersuchungsergebnissen für alle betrachteten Systeme.

8.4.4 Interessante Zwischenergebnisse

Das in der ersten Forschungsiteration entwickelte Prüfwerkzeug meldete nicht alle Verstöße und berechnete einige falsch positive Funde, d.h. es wurden Verstöße gemeldet, die im System nicht vorlagen (Karstens 2005). Diese Fehler ließen sich auf zwei Ursachen zurückführen: einige Regeln waren bewusst nicht vollständig implementiert, um den Implementierungsaufwand in einem realistischen Rahmen zu halten.

Die zweite, für die Entwicklung des Ansatzes interessantere Ursache war, dass teilweise Architekturelemente in den Softwaresystemen nicht korrekt erkannt wurden, da in dieser ersten Version der stilbasierten Architekturprüfung nicht mit Quelltextannotationen gearbeitet wurde. Stattdessen wurde der Ansatz verfolgt, die Zusatzinformationen aus den im System bereits vorhandenen Markierungen zu entnehmen (wie Marker-Interfaces, Vererbungsbeziehungen und Namensmuster, siehe Abschnitt 5.3.3). Die Annahme war, dass auf diese Weise an den zu untersuchenden Softwaresystemen keinerlei Änderungen nötig wären, da sämtliche für die stilbasierte Architekturprüfung notwendige Zusatzinformationen bereits im Quelltext vorhanden seien.

Diese Markierungen stellten sich jedoch als unzuverlässig heraus (Becker-Pechau et al. 2006; Karstens 2005): die Markierungen wurden nicht durchgängig verwendet und waren somit für die Elementidentifikation nicht verlässlich genug. Ferner waren die Markierungen sehr uneinheitlich, sowohl von System zu System als auch innerhalb eines Systems wurden verschiedene Markierungsarten verwendet.

So fiel zwar der Aufwand für nachträgliche Annotationen innerhalb bestehender Systeme weg, jedoch kam der Aufwand hinzu, das Werkzeug für jedes System so anzupassen, dass es die systemspezifischen Markierungen nutzen konnte (Becker-Pechau et al. 2006; Karstens 2005).

Aus diesen Gründen arbeitet die in dieser Arbeit vorgestellte finale Version der stilbasierten Architekturprüfung mit Quelltext-Annotationen (siehe Abschnitt 5.3.3 und 7.1.2). Wird ein System von Anfang an mit diesen Annotationen entwickelt, anstelle der bisher üblichen, jedoch nicht zuverlässigen Markierungen, so entsteht kein Mehraufwand, es ändert sich lediglich die Notation für die Zusatzinformationen.

8.5 Zusammenfassung

Die in diesem Kapitel vorgestellten praktischen Untersuchungen wurden nach der Methodik der Design-Forschung organisiert (Hevner et al. 2004): die stilbasierte Architekturprüfung wurde iterativ entwickelt und anhand praktischer Untersuchungen mit Hilfe exemplarischer Artefakte verbessert.

Die Untersuchungen dienten erfolgreich als Feedback bei der iterativen Entwicklung der stilbasierten Architekturprüfung. Ferner wurde die Machbarkeit der finalen Version der stilbasierten Architekturprüfung beispielhaft anhand der praktischen Untersuchungen geprüft.

Die Untersuchungen wurden von der Autorin und von Studierenden durchgeführt, welche im Rahmen der Forschungen für die stilbasierte Architekturprüfung ihre Arbeiten erstellten (Karstens 2005; Scharping 2008; Abraham 2006). Die Autorin hat die Themen der studentischen Arbeiten konzipiert und die Arbeiten von Karstens und Scharping betreut. Die Arbeit von

Abraham entstand im Rahmen eines von der Autorin geleiteten Diplom- und Studienarbeitsprojekts. Das Projekt umfasste mehrere Arbeiten im Themenbereich Softwarearchitektur.

In den praktischen Untersuchungen wurden die Typen und die Regeln des WAM-Stils, des Quasar-Stils und des projekteigenen Stils aus schriftsprachlichen Texten und mit Hilfe von Interviews ermittelt. Die aus dem WAM-Stil und dem projekteigenen Stil gewonnenen Erkenntnisse über den typischen Aufbau von Stilen – über die in Stilen definierten Typen und über die vorkommenden Regeltypen – flossen in die Konzeptualisierung von Architekturstilen ein. So wurde die ursprünglich auf Literaturarbeit basierende Konzeptualisierung nach der Methodik der Design-Forschung schrittweise evaluiert und verbessert. Mit Hilfe aller drei Stile wurde die finale Version der Konzeptualisierung (siehe Definitionen 3-1 und 3-2 und die Formalisierungen der Regeln in den Kapiteln 5 und 6) erfolgreich evaluiert.

Es wurden zwei Werkzeuge erstellt, die den Ansatz der stilbasierten Architekturprüfung beispielhaft für die Programmiersprache Java unterstützen: eine Erweiterung des bestehenden Architekturanalyse-Werkzeugs Sotograph sowie eine Neuentwicklung, der StyleBasedChecker. Es wurde gezeigt, dass sich ein Werkzeug zur stilbasierten Architekturprüfung erfolgreich in eine IDE einbinden lässt, so dass die Prüfung auf Stiltreue direkt zum Entwicklungszeitpunkt durchführbar ist. So erhalten Entwicklerinnen und Entwickler unmittelbares Feedback darüber, ob ihre Änderungen den gewählten Stil beachten.

Es wurden fünf Systeme des WAM-Stils und ein System mit projekteigenem Stil beispielhaft mit Architekturinformationen annotiert und erfolgreich auf Stiltreue geprüft. Grundlage dieser Prüfungen war die bereits mit allen drei Stilen evaluierte Konzeptualisierung für Architekturstile einschließlich der in der Konzeptualisierung enthaltenen Regeltypen.

Die Untersuchungen demonstrierten die Relevanz des Ansatzes: bei allen Systemen deckten die Untersuchungen bisher unbekannte Verstöße gegen den gewählten Stil auf, selbst dann, wenn die Systeme von Experten des jeweiligen Stils entwickelt wurden.

Es ließ sich zeigen, dass sich der Aufwand für die Prüfungen auf Stiltreue in einem geringen Rahmen hält. Die Untersuchungen zeigten, dass sich der Quelltext mit sehr geringem Mehraufwand nachträglich annotieren lässt: die korrekte Annotation eines System mit 146.817 Quelltextzeilen durch ein Projektmitglied dauerte 45 Minuten. Wird ein System bereits während der Programmierung annotiert, so ist es naheliegend, dass kein Mehraufwand im Vergleich zu stilbasierten Systemen ohne Quelltextannotationen entsteht, denn in allen untersuchten Systemen waren dieselben Informationen lediglich in anderer Notation (Kommentare, Marker-Interfaces etc.) bereits markiert. Diese bestehenden Markierungen variierten jedoch von System zu System und waren zu unzuverlässig für die stilbasierte Architekturprüfung. Aus diesem Grunde verwendet die stilbasierte Architekturprüfung mit den Quelltextannotationen eine formalere Notation, mit der sich die benötigten Informationen eindeutig festhalten lassen.

Wenn Entwicklungsteams mehrere Systeme desselben Stils prüfen, wird der Aufwand noch geringer, da sich dieselbe Stilbeschreibung unverändert weiterverwenden lässt. Dies wurde beispielhaft an den untersuchten Java-Systemen gezeigt.

Alle Untersuchungsergebnisse einschließlich der beispielhaften Stilbeschreibungen und Prüfergebnisse wurden zur Diskussion gestellt. Sie wurden in Gesprächen und Interviews mit Projektmitgliedern und Architekten des jeweiligen Stils abgesichert und in der wissenschaftlichen Gemeinde vorgestellt (Becker-Pechau 2009a, 2009b; Becker-Pechau und Bennicke 2007; Becker-Pechau et al. 2006).

9 Resümee und Ausblick

In der heutigen Praxis der Software-Entwicklung führt *Architekturerosion* zu erheblichen Problemen: Erodierende Softwaresysteme verlieren Schritt für Schritt an Verständlichkeit, Änderbarkeit und Wartbarkeit, bis sie in extremen Fällen zu unwartbaren Legacy-Systemen werden und durch Neuentwicklungen ersetzt werden müssen (Reussner und Hasselbring 2009; Sommerville 2001).

Systeme erodieren, indem sich die Konformanz der implementierten Architektur zu den Architekturvorgaben unbemerkt reduziert (Silva und Balasubramaniam 2012). Diesem Problem begegnet das Feld der *Architektur-Konformanzprüfungen*, auch bezeichnet als *Prüfungen auf Architekturtreue*. Die Prüfungen decken unbemerkte Verstöße gegen die gewählte Architektur anhand des Quelltextes auf. Anschließend können Entwicklungsteams die aufgedeckten Verstöße korrigieren und so die Architekturkonformanz wieder herstellen (Knodel und Popescu 2007).

Bisher wird die für ein Softwaresystem gewählte Architektur üblicherweise in Form einer *Soll-Architektur* vorgegeben. Soll-Architekturen zählen Architekturbestandteile auf (Architekturelemente, Beziehungen, manchmal auch Schnittstellen). Somit befinden sich Soll-Architekturen auf derselben Abstraktionsebene wie implementierte Architekturen. Sie werden für jedes System individuell entworfen und müssen geändert werden, sobald die gewünschte Architektur erweitert oder angepasst wird, beispielsweise aufgrund neuer Anforderungen. Damit einher geht ein gewisses Maß an Mehraufwand und Inflexibilität.

Eine weitere Klasse an Architekturvorgaben stellen *Architekturstile* dar (Garlan und Shaw 1994; Reussner und Hasselbring 2009). Sie gewinnen sowohl in der Forschung als auch in der Praxis an Bedeutung und ihre Vielfalt nimmt zu (Siedersleben 2004; Züllighoven 1998; Lilienthal 2008; Hofmeister et al. 2000). Architekturstile bewegen sich auf einer Metaebene zur Architektur, indem sie Vorgaben für *Typen* von Architekturbestandteilen machen. Dadurch können Stile – anders als Soll-Architekturen – bei Änderungen der gewünschten Architektur unverändert erhalten bleiben. Stile sind wiederverwendbar, beliebig viele Softwaresysteme können demselben Stil folgen.

Die vorliegende Arbeit diskutiert ausführlich den Begriff des Architekturstils. Sie zeigt auf, dass Softwareteams vielfältige Vorteile genießen, wenn sie ihre Softwarearchitektur anhand eines Architekturstils entwickeln und nicht (nur) anhand einer Soll-Architektur. Zu den Vorteilen von Stilen gehört – neben ihrer Wiederverwendbarkeit und langfristigen Gültigkeit – dass sie einheitlich strukturierte Architekturen fördern. Diese Arbeit erläutert, dass Stile somit den Entwurf, die Erweiterung und die Evolution von Architekturen unterstützen und zu verständlichen, änderbaren und wartbaren Softwaresystemen beitragen.

Bisher fehlte es jedoch an Konformanzprüfungen, mit denen sich die Konformanz einer implementierten Architektur zu einem *frei wählbaren Architekturstil* prüfen lässt. Diesem Problem widmet sich die vorliegende Arbeit, sie präsentiert einen neuen Ansatz, *die stilbasierte Architekturprüfung*.

Der Ansatz wurde nach dem Prinzip der Design-Forschung entwickelt (Hevner et al. 2004; Vaishnavi und Kuechler 2004). Nach dieser Forschungsmethodik entwickelte Innovationen werden in einem iterativen Prozess entwickelt und schrittweise verbessert. Die einzelnen Iterationen wechseln zwischen verschiedenen Abstraktionsebenen. Sie behandeln die Problemstellung, den Entwurf, die Implementierung beispielhafter Designobjekte sowie die Evaluation der erzielten Ergebnisse.

Vorarbeiten, Zwischenschritte und die finale Version der stilbasierten Architekturprüfung wurden auf Konferenzen und Workshops zur Diskussion gestellt (u.a. Becker-Pechau et al. 2005; Becker-Pechau et al. 2004; Becker-Pechau und Pechau 2003a) und in mehreren Artikeln

veröffentlicht (Becker-Pechau 2009a, 2009b; Becker-Pechau und Bennicke 2007; Becker-Pechau et al. 2006; Becker-Pechau und Pechau 2003b).

9.1 Kernpunkte der Arbeit

Die entscheidenden Beiträge dieser Arbeit lassen sich wie folgt auf den Punkt bringen:

- *Begriffsbildung*: die Arbeit liefert eine Abgrenzung und Konzeptualisierung für den Begriff des *Architekturstils*, basierend auf Literaturarbeit und auf der detaillierten Analyse bestehender, praxisrelevanter Architekturstile.
- Entwurf, Ausarbeitung, Umsetzung und Evaluation *eines neuen Ansatzes zur Architektur-Konformanzprüfung*, mit dem sich die im *Quelltext* von Softwaresystemen implementierte Architektur auf ihre Treue zu einem *frei definierbaren Architekturstil* prüfen lässt.

In beiden Aspekten geht diese Arbeit substantiell über bestehende Arbeiten hinaus und liefert innovative eigene Beiträge:

Die stilbasierte Architekturprüfung ist ein vollständig neuer Ansatz zur Konformanzprüfung. Mit frei definierbaren Architekturstilen unterstützt die stilbasierte Architekturprüfung eine signifikant andere Art von Architekturvorgaben als bisherige Ansätze.

Mit der stilbasierten Architekturprüfung liefert diese Arbeit nicht nur einen neuen Ansatz im Feld der Prüfungen auf Architekturtreue, sondern eröffnet ein neues Feld: die *Prüfungen auf Stiltreue*.

Im Folgenden werden die genannten Kernpunkte kurz erläutert.

9.1.1 Begriffsbildung und Konzeptualisierung

Eine gründliche Auseinandersetzung mit dem Begriff des Architekturstils in Abgrenzung zu anderen Arten von Architekturvorgaben liefert die Basis der stilbasierten Architekturprüfung. Die Diskussion über Architekturstile in Wissenschaft und Praxis ist noch nicht abgeschlossen. In der Literatur finden sich viele, teilweise widersprüchliche Definitionen. Andere verbreitete Bezeichnungen für Architekturstile sind *Architekturmuster* und *Referenzarchitekturen*, wobei diese Begriffe auch für andere Arten von Architekturvorgaben verwendet werden.

Stile zeichnet aus, dass sie – im Gegensatz zu anderen Arten von Architekturvorgaben – ein *Metamodell* für Softwarearchitekturen definieren. Architekturstile beinhalten

- *Typen* von Architekturbestandteilen (Architekturelement-Typen und Beziehungstypen)
- *Regeln* für Architekturbestandteile abhängig von deren Typ.

Stilbasierte Architekturen, d.h. Architekturen, die nach einem Architekturstil entworfen wurden, lassen sich als *Exemplare* des Stils verstehen. Zu einem Stil lassen sich beliebig viele zugehörige Architekturen definieren.

Die Arbeit liefert *formalisierte Konzeptualisierungen* für *Architekturstile*, für *stilbasierte Architekturen* und für *Quelltextstrukturen* in Form von *Metamodellen*. Die Formalisierung dient der klaren Darstellung und lässt sich als Ausgangspunkt für die Realisierung einer Werkzeugunterstützung nutzen (siehe unten).

9.1.2 Der Ansatz der stilbasierten Architekturprüfung

Eine stilbasierte Architekturprüfung wird in mehreren Schritten mit Hilfe eines Prüfwerkzeugs durchgeführt:

Einmalige Vorbereitung der Prüfung:

Vor der ersten Prüfung muss der gewählte Stil in einer für das Prüfwerkzeug geeigneten Notation beschrieben werden. Die Beschreibung enthält die Architekturelement- und Beziehungstypen sowie sämtliche Regeln des Stils. Diese Beschreibung braucht nur einmalig erstellt zu werden, mit ihr lassen sich beliebig viele Systeme desselben Stils prüfen. Abhängig von der Realisierung des Prüfwerkzeugs ließe sich die Beschreibung sogar für Systeme unterschiedlicher Programmiersprachen wiederverwenden (siehe Kapitel 7).

Quelltext während der Software-Entwicklung annotieren:

Während der Programmierung fügen Entwicklungsteams – wie bei stilbasierten Systemen üblich – Zusatzinformationen über die Architektur in den Quelltext ein. Diese Zusatzinformationen sind nötig, da nicht alle Architekturinformationen direkt in der jeweiligen Programmiersprache ausdrückbar sind. Die Zusatzinformationen wurden bisher informell ergänzt, unter anderem in Form von Quelltextkommentaren. Für die stilbasierte Prüfung wird eine formale Notation, beispielsweise mit Java-Annotationen, benötigt, da damit die Zusatzinformationen von einem Prüfwerkzeug eingelesen werden können. Die stilbasierte Architekturprüfung erlaubt bei Bedarf, die Zusatzinformationen nicht im Quelltext, sondern im Werkzeug oder in einer anderen externen Quelle zu vermerken. Dies ist beispielsweise nützlich bei Systemen, deren Quelltext nicht verändert werden soll, da man sie nur einmalig und nicht kontinuierlich prüfen möchte.

Prüfung durchführen:

Die eigentliche Prüfung auf Stiltreue lässt sich in verschiedenen Kontexten durchführen:

- eingebettet in eine Entwicklungsumgebung, für unmittelbares Feedback während der Programmierung,
- in einem separaten Prüfwerkzeug, beispielsweise für Architekturreviews,
- automatisiert, beispielsweise wenn Quelltext in ein zentrales Repository übertragen wird oder in nächtlichen Prüfläufen.

Die Prüfung wird durch ein Prüfwerkzeug durchgeführt. Das Werkzeug durchläuft dabei intern folgende Schritte:

- *Stilbeschreibung einlesen:*
Ergebnis dieses Schritts ist ein im Prüfwerkzeug gehaltenes Modell des gegebenen Stils. Dieser Schritt kann alternativ beim Start des Werkzeugs geschehen und nur bei Änderungen der Stilbeschreibung wiederholt werden.
- *Implementierte Architektur (Ist-Architektur) ermitteln:*
Ergebnis dieses Schritts ist ein Modell der Ist-Architektur des zu prüfenden Systems. Jedes Modellelement enthält einen Verweis zum zugehörigen Quelltextausschnitt. Das Modell wird ermittelt, indem ein spezieller Parser den Quelltext nebst Zusatzinformationen analysiert. Bei Bedarf lässt sich das Modell auch inkrementell ermitteln, dann braucht der Parser gegebenenfalls nur Ausschnitte des Quelltextes zu analysieren.
- *Verstöße innerhalb der Ist-Architektur ermitteln:*
Das Werkzeug iteriert über die Regeln des Architekturstils und prüft deren Einhaltung

innerhalb der Ist-Architektur. Dafür verwendet das Werkzeug das zuvor berechnete Modell der Ist-Architektur. Ergebnis ist eine Menge von Verstößen gegen die Regeln des Stils. Für jeden Verstoß werden die fehlerhaften Architekturbestandteile sowie die betroffene Regel erfasst. Auch dieser Schritt lässt sich inkrementell ausführen: wird die Prüfung aufgrund einer Quelltextänderung angestoßen, so ist es möglich, nur die Anteile der Ist-Architektur zu prüfen, die von dieser Änderung betroffen sind.

- *Betroffene Quelltextstellen ermitteln:*
Das Werkzeug ermittelt für jeden Verstoß die zugehörigen Quelltextstellen. Dafür werden die im Modell der Ist-Architektur gespeicherten Quelltext-Verweise verwendet. Für jeden Verstoß werden die fehlerhaften Architekturbestandteile betrachtet und die ihnen zugeordneten Quelltextstellen erfasst.
- *Verstöße melden:*
Der Ansatz erlaubt, die gefunden Verstöße direkt im Zusammenhang mit den betroffenen Quelltextstellen zu melden. So wissen Entwicklungsteams, welche Stellen voraussichtlich von Korrekturen betroffen sein werden. Die Art der Meldung ist den Designern des Prüfwerkzeugs überlassen. Bei einer in eine IDE integrierten Prüfung bietet es sich beispielsweise an, die dort übliche Form für Warnungen zu verwenden.

9.1.3 Entwurfskriterien für die stilbasierte Architekturprüfung

In dieser Arbeit wurde eine Reihe von Entwurfskriterien aufgestellt, die als Leitlinie bei der Entwicklung der stilbasierten Architekturprüfung dienen. Im Folgenden werden diese Kriterien noch einmal zusammengefasst und die zu ihrer Erfüllung genutzten Maßnahmen rekapituliert:

Quelltextnahe Ist-Architektur: Die zu prüfende Architektur wird im größtmöglichen Umfang aus dem Quelltext ermittelt und so wenig wie möglich aus externen Quellen. Allerdings lassen sich Architekturen (unabhängig davon, ob sie auf Stilen basieren) dem Quelltext nicht vollständig entnehmen. Die vorliegende Arbeit untersucht, welche Anteile einer stilbasierten Architektur in Quelltexten vorhanden sind. Sämtliche im Quelltext befindliche Anteile werden für die stilbasierte Architekturprüfung genutzt. Für die Architekturanteile, die dem Quelltext nicht entnehmbar sind, enthält die stilbasierte Architekturprüfung ein Verfahren zur Annotation des Quelltextes mit den benötigten Zusatzinformationen. So kann der Quelltext als *ausschließliche Quelle* zur Ermittlung der Ist-Architektur dienen. Dieses Verfahren begegnet dem Problem, dass Quelltexte und externe Architekturdokumentationen über die Zeit unbemerkt auseinander driften können, und Konformanzprüfungen dadurch zu falschen Prüfergebnissen kommen. Der in dieser Arbeit vorgestellte Ansatz, den Quelltext mit Architekturinformationen zu annotieren, folgt dem Prinzip der Inline-Dokumentation. Als Vorarbeit hierfür diente eine von der Autorin betreute Diplomarbeit über die Inline-Dokumentation von Entwurfsmustern (Özkan und Özkan 2004).

Leichtgewichtigkeit: Es wurde darauf geachtet, dass der vorgestellte Ansatz möglichst leichtgewichtig ist und die Darstellung des Ansatzes keine Verständnishürden aufbaut. Für die Formalisierung wurde eine in der Informatik allgemein bekannte Notation verwendet: mengentheoretische Ausdrücke. Ferner wurde die stilbasierte Architekturprüfung in ein Kernkonzept mit Ausbaustufen aufgeteilt. Diese Aufteilung dient der Darstellung des Ansatzes und kann in Projekten als Basis genutzt werden, um die stilbasierte Architekturprüfung schrittweise einzuführen. Das Kernkonzept unterstützt die am weitesten verbreiteten und etablierten Regeltypen, die sich in annähernd jedem Stil finden. Die Ausbaustufen unterstützen speziellere Regeltypen. Bei Bedarf lässt sich der Ansatz leicht um weitere Ausbaustufen ergänzen.

Möglichst frühzeitiges Aufdecken von Architekturerosion: Je länger Architekturverstöße im Quelltext vorliegen, umso größer wird die Gefahr, dass diese Verstöße zu Problemen bei der Weiterentwicklung und Wartung führen. Der vorgestellte Ansatz ermöglicht es, Architekturverstöße unmittelbar *innerhalb der Entwicklungsumgebung* zu melden. So können Entwicklerinnen und Entwickler ihren Quelltext bereits korrigieren, *bevor* sie ihn in das genutzte Repository übertragen.

Praxistauglichkeit: Entwicklungsteams sollen ihr eigenes Verständnis von *Architekturstilen* und *stilbasierten Architekturen* unverändert für die stilbasierte Architekturprüfung nutzen können. Damit soll vermieden werden, dass die Entwicklungsteams zwei unterschiedliche Modelle aufeinander abbilden müssen („model mismatch“). Um dieses Ziel zu erreichen, wurde die vorgestellte Konzeptualisierung anhand von *realen, praxisrelevanten* Stilen entwickelt (Quasar, WAM und ein projektspezifischer Stil). Der Quasar-Stil wurde basierend auf Literaturarbeit formalisiert und anhand einer beispielhaften Umsetzung angewendet. Die Formalisierung des WAM-Stils basierte auf Literaturarbeit und auf der detaillierten Untersuchung und wiederholten Prüfung mehrerer WAM-basierter Softwaresysteme. Ferner wurde ein System mit projektspezifischem Stil untersucht und auf Stiltreue geprüft. Die Formalisierungen und Untersuchungen wurden durch Interviews abgesichert. Vorarbeiten für die Formalisierung des Quasar-Stils lieferte eine Studienarbeit (Abraham 2006), die im Rahmen eines von der Autorin geleiteten Studien- und Diplomarbeiterprojektes im Themenbereich Softwarearchitekturen entstand. Die Formalisierung des WAM-Stils entstand in mehreren Forschungsiterationen und wurde durch die von der Autorin betreuten Diplomarbeiten von Karstens und Scharping unterstützt (Karstens 2005; Scharping 2008).

9.1.4 Praktische Umsetzung und Evaluation

Im Rahmen dieser Arbeit wurden zwei beispielhafte Werkzeuge für die stilbasierte Architekturprüfung erstellt. Beide Werkzeuge können in der Programmiersprache Java realisierte Systeme prüfen. Das *Konzept* der stilbasierten Architekturprüfung hingegen ist nicht auf eine Programmiersprache festgelegt.

Die Grundlage für eine Werkzeugunterstützung zur stilbasierten Architekturprüfung liefern die in Abschnitt 9.1.1 genannten, formalen Konzeptualisierungen, insbesondere die in der Stil-Konzeptualisierung enthaltenen Berechnungsvorschriften für die verschiedenen Regeltypen.

Das erste Werkzeug ist eine Erweiterung des kommerziellen Architekturanalyse-Werkzeugs Sotograph (Bischofberger et al. 2004). Es diente zur Evaluation der Ergebnisse eines frühen Forschungszyklus. Die Umsetzung wurde im Rahmen einer von der Autorin konzipierten und betreuten Diplomarbeit erstellt (Karstens 2005).

Das zweite Werkzeug, der StyleBasedChecker, ist ein prototypisches Plugin für die Eclipse-IDE. Es ermöglicht, Softwaresysteme direkt während der Programmierung zu prüfen. Der StyleBasedChecker entstand ebenfalls in einer von der Autorin betreuten Diplomarbeit (Scharping 2008) und wurde anschließend durch die Autorin weiterentwickelt.

Die Werkzeuge wurden verwendet, um mehrere Softwaresysteme des WAM-Stils und ein System mit projektspezifischem Stil erfolgreich auf Stiltreue zu prüfen. Die Prüfergebnisse wurden durch Interviews abgesichert. Interviewt wurden mit den betrachteten Stilen vertraute Softwarearchitektinnen und -architekten sowie Projektmitglieder der untersuchten Systeme.

Mit Hilfe des StyleBasedChecker ließ sich beispielhaft an mehreren Java-Systemen zeigen, dass eine konsequente Trennung der Stilbeschreibung und der für die Ermittlung der Ist-Architektur benötigten Zusatzinformationen möglich ist. Diese Trennung erlaubte in den praktischen Untersuchungen, dieselbe Stilbeschreibung für verschiedene Systemversionen und für mehrere Systeme desselben Stils wiederzuverwenden.

Das Werkzeug zeigt, dass sich eine inkrementelle Prüfung realisieren lässt. Beim Start des Prüfwerkzeugs wird das gesamte System geprüft. Werden anschließend der Quelltext oder die Stilbeschreibung geändert, so löst dies eine neue Prüfung aus (integriert in den Kompilierungsvorgang der IDE). Für die Prüfung werden nur die potentiell betroffenen Anteile der Ist-Architektur neu berechnet und auf Stiltreue geprüft.

Die Untersuchungen in Form von Prüfungen und Interviews untermauern die Relevanz des Ansatzes. Selbst in Systemen, die von Experten des jeweiligen Architekturstils entwickelt wurden, konnten durch die Prüfungen mehrfache, bisher unbekannte Verstöße gegen die Regeln des Architekturstils aufgedeckt werden (siehe Kapitel 8 und Anhang C).

9.2 Einordnung und Abgrenzung

Kapitel 4 enthält eine ausführliche Einordnung und Abgrenzung der stilbasierten Architekturprüfung. An dieser Stelle seien die Kernpunkte genannt:

Der vorgestellte Ansatz prüft die *tatsächliche Implementierung* von Softwaresystemen auf Konformanz, in Abgrenzung zu Ansätzen, die eine formale *Beschreibung* der Softwarearchitektur prüfen. So werden Prüffehler vermieden, die aus einer Abweichung der Architekturbeschreibung von der tatsächlichen Softwarearchitektur resultieren. Ferner ersparen sich Entwicklungsteams die aufwändige und potentiell fehlerträchtige Ermittlung der von Architekturfehlern betroffenen Quelltextstellen.

Bei der stilbasierten Architekturprüfung handelt es sich um einen Ansatz zur *statischen* Architektur-Konformanzprüfung. Bei statischen Ansätzen wird das zu prüfende Softwaresystem nicht ausgeführt. Die stilbasierte Architekturprüfung untersucht den Quelltext von Softwaresystemen mittels statischer Analyse. Der Begriff des Quelltextes ist hier so breit zu verstehen, dass auch beispielsweise Konfigurationsdateien bei Bedarf in die Prüfung mit einbezogen werden können.

Die Arbeit definiert und formalisiert den *Begriff des Architekturstils* mit dem Ziel, einen Ansatz zur statischen Konformanzprüfung zu entwickeln, der Architekturvorgaben in Form von Stilen erlaubt. Die Arbeit konzentriert sich auf die für dieses Ziel relevanten Eigenschaften von Stilen. Über die hier betrachteten Eigenschaften hinaus können Stile auch semantische Vorgaben und Interpretationen für den Architekturentwurf beinhalten. Diese Vorgaben und Interpretationen stellen einen Zusammenhang zwischen der Architektur und anwendungsfachlichen Modellen her (Garlan et al. 1994; Züllighoven 2005). Die Einhaltung solcher Vorgaben zu prüfen geht über statische Architekturprüfungen hinaus, da hierfür nicht nur Ist-Architekturen und Architekturstile, sondern darüber hinaus anwendungsfachliche Modelle in die Prüfung einfließen müssten.

Der *zentrale Unterschied* des vorgestellten Ansatzes zu anderen Ansätzen im Bereich der Architektur-Konformanzprüfung ist, dass die stilbasierte Architekturprüfung als Architekturvorgabe *frei definierbare Stile* erlaubt. Die Stile lassen sich unabhängig von der gewählten Technologie (wie Rahmenwerke und Komponentenmodelle) definieren. Die Mehrzahl der bisherigen Ansätze prüft die Konformanz zu Architekturvorgaben in Form von Soll-Architekturen. Ferner gibt es einige Ansätze, die Vorgaben auf der Ebene des Quelltextes behandeln (beispielsweise Vorgaben für Methodenparameter oder Bezeichner). Will man die Stil-Konformanz mit diesen Ansätzen prüfen, so muss man den gewünschten Stil manuell in Vorgaben auf Architektur- oder auf Quelltextebene übersetzen. Dies ist aufwendiger, unkomfortabler und fehlerträchtiger, als die Vorgaben auf der Abstraktionsebene des Stils zu beschreiben.

Mit dem bereits erwähnten Konzept zur Quelltextannotation erlaubt der vorgestellte Ansatz, die Ist-Architektur ausschließlich aus *einer* Quelle zu ermitteln. Dies ist ein weiterer wesentlicher Unterschied zu anderen bisherigen Ansätzen.

Die stilbasierte Architekturprüfung ist weitestgehend programmiersprachenunabhängig, jedoch erfordern statische Architektur-Konformanzprüfungen generell, dass sich die Typinformationen durch statische Analyse aus dem Quelltext entnehmen lassen. Somit ist auch die stilbasierte Architekturprüfung auf *statisch getypte* Sprachen eingeschränkt. Die statische Typisierung darf selbstverständlich nicht durch Mechanismen wie beispielsweise Java Reflections umgangen werden.

Im Rahmen der vorliegenden Arbeit wurden *objektorientierte* Sprachen, Systeme und zugehörige Stile betrachtet. Die untersuchten Architekturstile WAM und Quasar werden für verschiedene objektorientierte Programmiersprachen verwendet. Das prototypische Werkzeug, der StyleBasedChecker, wurde beispielhaft für die Prüfung von Java-Systemen entwickelt.

Es ist jedoch naheliegend, dass sich der Ansatz generell für *imperative* Sprachen nutzen lässt, da mehrere gängige Architekturstile (beispielsweise der Client-Server-Stil) für beide Paradigmen verwendet werden.

9.3 Ausblick

An die vorliegende Arbeit schließen sich vielfältige interessante Forschungsfragen an. An dieser Stelle sei eine Auswahl genannt:

Werkzeug mit Produktreife: Ein wichtiger Aspekt bei Forschungsprojekten innerhalb der Softwaretechnik ist die praktische Evaluierung. Im Rahmen der vorliegenden Arbeit wurden reale Stile und Softwaresysteme als Basis für die Konzeptualisierung und die beispielhafte Umsetzung genutzt und die Ergebnisse durch Interviews abgesichert. In der Zukunft wäre es wünschenswert, den Ansatz durch ein Werkzeug in Produktreife zu unterstützen und dessen Einsatz in der Praxis längerfristig zu evaluieren und dabei auch weitere Stile und Programmiersprachen zu betrachten. Im Sinne des Design-Researchs stellt dies eine konsequente Fortführung der Forschung dar.

Notation für Architekturstil-Beschreibungen: Die in dieser Arbeit entwickelte Konzeptualisierung für Architekturstile ist an keine konkrete Beschreibungssyntax gebunden. Der Prototyp StyleBasedChecker verwendet beispielhaft die Sprache XML für alle Aspekte von Architekturstilen, einschließlich aller Regeltypen, außer den Schnittstellenregeln. Die Schnittstellenregeln bilden eine Ausnahme, sie werden in Form von Regelklassen ausprogrammiert. Andere Stil-Beschreibungssprachen als XML und Regelklassen sind denkbar. Der Fokus dieser Arbeit lag auf der Entwicklung eines Ansatzes zur Prüfung auf Stiltreue, nicht jedoch auf einer optimalen Beschreibungsnotation für Architekturstile. Es schließt sich die interessante Frage an, welche Notation hierfür am besten geeignet ist und nach welchen Kriterien dies zu bewerten sei. Denkbar sind textuelle und grafische Notationen. Es könnten bestehende Beschreibungsnotationen geprüft und gegebenenfalls angepasst werden. Darüber hinaus ließe sich prüfen, ob die Entwicklung einer eigenen Architekturbeschreibungssyntax (ADL) anhand der gegebenen Konzeptualisierung sinnvoll ist. Für Schnittstellenregeln ließe sich prüfen, welche Programmiersprachen-Metamodelle geeignet sind, die Regeln sprachunabhängig zu formulieren (vgl. Trifu und Szulman 2005; Becker et al. 2010). In diesem Falle könnten alle Bestandteile des Architekturstils, einschließlich der Schnittstellenregeln, einheitlich, beispielsweise mit XML, beschrieben werden. Die Regelklassen wären überflüssig.

Automatisierte Konsistenzprüfung von Stilbeschreibungen: Zwar bewegen sich alle untersuchten Architekturstile in einer Größenordnung, die sich gut überblicken lässt, dennoch könnte es hilfreich sein, wenn Prüfwerkzeuge automatische Konsistenzprüfungen von Stilbeschreibungen durchführen und so fehlerhafte Vorgaben (wie beispielsweise sich widersprechende Regeln) automatisiert aufdecken.

Systeme mit mehreren Stilen oder mehreren Stilvarianten: Prinzipiell ist es denkbar, dass Systeme mehreren Stilen oder verschiedenen Versionen desselben Stils folgen. Solche Systeme können beispielsweise entstehen, wenn ein älteres System mit neuer Funktionalität ergänzt wird und der neue Quelltext einer aktuelleren Variante des ursprünglichen Architekturstils folgt. Auch solche Systeme lassen sich mit der stilbasierten Architekturprüfung untersuchen, wenn die verschiedenen Stile in *getrennten* Systemteilen auftreten. Es wäre jedoch interessant zu ermitteln, inwieweit in der Praxis Systeme eine Rolle spielen, bei denen mehrere Stile oder Stilvarianten innerhalb *desselben* Systems *gemischt* werden. Ein möglicher Umgang hiermit wäre, die verwendeten Stile zu einem neuen, einzelnen Stil zu kombinieren. So könnte die stilbasierte Architekturprüfung unverändert genutzt werden. Ähnlich wurde es beispielsweise im WAM-Stil gehandhabt, der zwei Varianten des Architekturelement-Typs Werkzeug enthält. Wenn mehrere Varianten desselben Stils gemischt werden, könnte es alternativ nützlich sein, die Möglichkeiten zur Stilbeschreibung so zu erweitern, dass sich die Regeln und Elementtypen jeweils einer oder mehreren *Stilvarianten* zuordnen lassen.

Dynamische Typisierung: Ein interessante Frage ist, ob sich der Ansatz auch nutzbringend für dynamisch getypte Programmiersprachen verwenden lässt. Dies erscheint möglich, sofern die Typinformationen in Form von Annotationen vorliegen (wie beispielsweise in der Programmiersprache Smalltalk üblich).

Andere Paradigmen: Diese Arbeit betrachtet *objektorientierte* Architekturstile und Programmiersprachen. Eine interessante Fortführung der Forschungstätigkeiten wäre, bestehende Architekturstile für imperative, *nicht-objektorientierte* Sprachen zu untersuchen und – sofern nötig – die in dieser Arbeit erstellte Konzeptualisierung anzupassen. Ferner ließe sich untersuchen, ob sich auch in Programmiersprachen anderer Paradigmen bereits Architekturstile etabliert haben, inwieweit sich die stilbasierte Architekturprüfung bei diesen Sprachen einsetzen lässt und ob Anpassungen des Ansatzes nötig sind.

Integration mit Traceability-Ansätzen: Das Forschungsfeld der Traceability beschäftigt sich mit der Frage, wie sich konzeptionelle Verbindungen zwischen verschiedenen Artefakten der Software-Entwicklung ermitteln und dokumentieren lassen (Cleland-Huang et al. 2012). Ein Beispiel ist die Verbindung von Anforderungen zu den Quelltextausschnitten, welche die Anforderungen realisieren. Auch Verbindungen zwischen Architektur und Quelltext werden adressiert. Hier ließe sich untersuchen, inwieweit sich die bestehenden Traceability-Ansätze in die stilbasierte Architekturprüfung integrieren lassen, als alternativer Ansatz zur Quelltextannotation. Zwar hätte dies ggf. den Nachteil, dass die Verbindungen Quelltext-extern gespeichert werden, aber sofern Projekte bereits ein Traceability-Werkzeug zur Speicherung der Verbindungen verwenden, könnte es nützlich sein, diese Informationen für die stilbasierte Architekturprüfung zu verwenden, um eine redundante Speicherung der Informationen zu vermeiden.

Qualität der Zusatzinformationen im Projektverlauf: Wie alle Architektur-Konformanzprüfungen hängt die stilbasierte Architekturprüfung von einer korrekten Zuordnung zwischen Quelltext- und Architekturelementen ab. Die stilbasierte Architekturprüfung begegnet diesem Problem und geht einen Schritt weiter als andere Ansätze, indem sie die Zusatzinformationen in den Quelltext integriert und zwar idealerweise bereits während der Programmierung. Die Hoffnung, dass sich Zuordnungsfehler und resultierende Prüffehler hierdurch vermeiden lassen, wurde durch die Untersuchungen dieser Arbeit gestützt: Vor der beispielhaften Prüfung wurden die in den Systemen bereits vorhandenen, informellen Zusatzinformationen in Java-Annotationen übersetzt. Nur ein einziges Mal entstand ein Prüffehler durch eine fehlerhafte Annotation, wobei der Prüffehler sofort offensichtlich wurde. Dennoch wäre es interessant, die Güte der Zusatzinformationen durch den langfristigen Einsatz der stilbasierten Architekturprüfung in realen Projekten zu untersuchen. Ferner ließe sich untersuchen, ob es sinnvoll ist, die manuelle Ergänzung von Zusatzinformationen mit Ansätzen zur automatisierten Elementerkennung zu verbinden, ähnlich wie im Ansatz von Bittencourt et al. (Bittencourt et al. 2010). Hierfür könnte geprüft werden, ob beispielsweise Clustering-basierte Ansätze oder Verfahren zur

Graphmustererkennung geeignet sind, stilbasierter Architekturelemente im Sinne dieser Arbeit zu ermitteln, und ob sich existierende Werkzeuge wie beispielsweise SISSy, SoMoX, Reclipse oder PINOT (Becker et al. 2010; Detten et al. 2010; Shi und Olsson 2006) nutzen oder um entsprechende Funktionalität erweitern lassen. Zwar zeigen die praktischen Untersuchungen dieser Arbeit, dass die automatisierte Erkennung von Architekturelementen als alleiniges Mittel nicht zuverlässig genug ist, sie könnte jedoch nützlich sein, um potentielle Annotationsfehler aufzuzeigen und um einen Ausgangspunkt zu liefern, falls man Systeme nachträglich annotieren möchte.

Programmiersprachen mit integrierten Architekturkonstrukten: Die Lücke zwischen Quelltext und Softwarearchitektur stellt in der Praxis ein erhebliches Problem dar: Werden Architekturentwürfe implementiert, so verschwinden essentielle Architekturinformationen, da sich diese Informationen in den bestehenden Programmiersprachen nicht ausdrücken lassen. Folglich lässt sich die Architektur dem Quelltext nicht mehr vollständig entnehmen. Dies kann nicht nur zu Architekturerosion führen, sondern auch zu Fehlentscheidungen bei der Evolution von Softwarearchitekturen, mit negativen Auswirkungen auf zentrale Qualitätsattribute wie Verständlichkeit und Wartbarkeit. Aus diesem Grunde wird im Kontext der Architekturforschung wiederholt die Forderung nach einer entsprechenden Erweiterung von Programmiersprachen geäußert (beispielsweise Broy und Reussner 2010; van Eyck et al. 2011; Knodel und Popescu 2007; Clements und Shaw 2009). Es ist die Überzeugung der Autorin, dass solche Erweiterungen von Programmiersprachen eine große Rolle in der zukünftigen Forschung und Praxis spielen werden. Der in der vorliegenden Arbeit vorgestellte Ansatz zur Quelltextannotation adressiert das genannte Problem. Hier ließe sich einen Schritt weiter gehen, so dass zukünftige Programmiersprachen erlauben, Architekturkonstrukte für stilbasierte Architekturen als integralen Bestandteil der Sprache auszudrücken.

Architektur-Wahrnehmung während der Programmierung: Der im Rahmen der Forschungstätigkeiten dieser Arbeit entwickelte StyleBasedChecker geht über die stilbasierte Architekturprüfung hinaus, indem er eine (rudimentäre) Architektursicht während der Programmierung in die IDE integriert. Die Autorin hat diese Idee des „architecture-aware programming“ im Rahmen eines Konferenz-Workshop vorgestellt und diskutiert (Becker-Pechau et al. 2004). Architektursichten in Entwicklungsumgebungen zu integrieren stellt nach wie vor eine interessante Herausforderung im Bereich der Architekturforschung dar, sowohl für stilbasierte als auch für andere Architekturen. Dabei ließen sich Architektursichten für verschiedene Zwecke nutzen. Beispielsweise könnten die Architektursichten einer besseren Darstellung von Architekturverstößen dienen, das Architekturverständnis der Entwicklerinnen und Entwickler erhöhen und so die Qualität der implementierten Architektur verbessern. Die in dieser Arbeit entwickelte Konzeptualisierung kann als Ausgangspunkt für die Visualisierung stilbasierter Architekturen dienen.

Diese Aufzählung verdeutlicht, dass die erzielten Ergebnisse zu weiteren spannenden Forschungsfragen führen. Über die wissenschaftlichen Ergebnisse hinaus ist es die Hoffnung der Autorin, dass der vorgestellte Ansatz seinen Weg in die Praxis der Software-Entwicklung fortsetzt. Das bisherige Feedback aus der Praxis zeigt, dass Interesse und Bedarf an diesem Thema besteht. Die Autorin wird sich sowohl auf wissenschaftlicher als auch auf praktischer Ebene weiterhin mit der stilbasierten Architekturprüfung und angrenzenden Fragestellungen befassen.

Literaturverzeichnis

- Abraham, Felix (2006): Vergleichende Implementierung von Arbeitsprozessen in verschiedenen Modellarchitekturen. Studienarbeit. Universität Hamburg, Deutschland.
- Aho, Alfred V.; Sethi, Ravi; Ullmann, Jeffrey D. (1988): Compilerbau, Teil 1. München, Deutschland: Oldenbourg.
- Aldrich, Jonathan (2008): Using Types to Enforce Architectural Structure. In: Kruchten, Philippe; Garlan, David; Woods, Eoin (Hg.): WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture. Los Alamitos, CA, USA: IEEE Computer Society, S. 211-220.
- Aldrich, Jonathan; Chambers, Craig; Notkin, David (2002): ArchJava: Connecting Software Architecture to Implementation. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering. New York, NY, USA: ACM, S. 187-197.
- Athanasopoulos, Michael; Kontogiannis, Kostas; Brealey, Chris (2011): Towards an Interpretation Framework for Assessing Interface Uniformity in REST. In: Pautasso, Cesare; Wilde, Erik (Hg.): WS-REST '11: Proceedings of the Second International Workshop on RESTful Design. New York, NY, USA: ACM, S. 47-50.
- Avgeriou, Paris; Guelfi, Nicolas; Medvidović, Nenad (2005): Software Architecture Description and UML. In: Jardim Nunes, Nuno; Selic, Bran; Rodrigues da Silva, Alberto; Toval Alvarez, Ambrosio (Hg.): UML Modeling Languages and Applications. Berlin, Heidelberg, Deutschland: Springer (Lecture Notes in Computer Science, 3297), S. 23-32.
- Baier, Achim; Becker, Steffen; Jung, Martin; Krogmann, Klaus; Röttgers, Carsten; Streekmann, Niels et al. (2009): Modellgetriebene Software-Entwicklung. In: Reussner, Ralf; Hasselbring, Wilhelm (Hg.): Handbuch der Software-Architektur. 2. Auflage. Heidelberg, Deutschland: dpunkt-Verlag, S. 93-122.
- Balz, Moritz (2012): Embedding Model Specifications in Object-Oriented Program Code. A Bottom-up Approach for Model-based Software Development. Dissertation. Universität Duisburg-Essen, Deutschland.
- Balzert, Helmut (1998): Lehrbuch der Softwaretechnik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Heidelberg, Berlin, Deutschland: Spektrum Akademischer Verlag.
- Balzert, Helmut (2001): Lehrbuch der Softwaretechnik. 2. Auflage. Heidelberg, Deutschland: Spektrum Akademischer Verlag (Lehrbücher der Informatik, 1).
- Bass, Len; Clements, Paul; Kazman, Rick (2003): Software Architecture in Practice. Reading, MA, USA: Addison-Wesley (SEI Series in Software Engineering).
- Bass, Len; Clements, Paul; Kazman, Rick (2012): Software Architecture in Practice. 3. Auflage. Upper Saddle River, NJ, USA: Addison-Wesley (SEI Series in Software Engineering).

- Becker, Steffen; Hauck, Michael; Trifu, Mircea; Krogmann, Klaus; Kofroň, Jan (2010): Reverse Engineering Component Models for Quality Predictions. In: CSMR '10: Proceedings of the 14th European Conference on Software Maintenance and Reengineering. Los Alamitos, CA, USA; IEEE Computer Society, S. 194-197.
- Becker, Steffen; Koziolok, Heiko; Reussner, Ralf (2009): The Palladio Component Model for Model-driven Performance Prediction. In: *Journal of Systems and Software* 82, S. 3-22.
- Becker-Pechau, Petra (2009a): Quelltextannotationen für stilbasierte Ist-Architekturen. In: Engels, Gregor; Reussner, Ralf; Momm, Christof; Sauer, Stefan (Hg.): Design for Future – Langlebige Softwaresysteme. 1. Workshop des GI-Arbeitskreises "Langlebige Softwaresysteme (L2S2)", Bd. 537. Karlsruhe, Deutschland: CEUR Workshop Proceedings, S. 3-14.
- Becker-Pechau, Petra (2009b): Stilbasierte Architekturprüfung. In: Fischer, Stefan; Mähle, Erik; Reischuk, Rüdiger (Hg.): Informatik '09, P-154. Bonn, Deutschland: Gesellschaft für Informatik (GI) (Lecture Notes in Informatics), S. 3264-3275.
- Becker-Pechau, Petra; Bennicke, Marcel (2007): Concepts of Modeling Architectural Module Views for Compliance Checks Based on Architectural Styles. In: Smith, J. (Hg.): SEA '07: Proceedings of the Eleventh Conference on Software Engineering and Application. Cambridge, MA, USA: Acta, S. 168-175.
- Becker-Pechau, Petra; Grenon, Pierre; Lycett, Mark; Partridge, Chris; Pechau, Jörg; Siebert, Dirk (2005): Philosophy, Ontology, and Information Systems. In: Malenfant, Jacques; Østvold, Bjarte M. (Hg.): ECOOP '04: Object-Oriented Technology, Workshop Reader, Oslo, Norway, June 14-18, 2004, Final Reports. Berlin, Heidelberg, Deutschland: Springer (Lecture Notes in Computer Science, 3344), S. 62-66.
- Becker-Pechau, Petra; Karstens, Bettina; Lilienthal, Carola (2006): Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln. In: Biel, Bettina; Book, Matthias; Gruhn, Volker (Hg.): Software Engineering 2006. Leipzig, Deutschland: Gesellschaft für Informatik (GI) (Lecture Notes in Informatics), S. 27-38.
- Becker-Pechau, Petra; Klischewski, Ralf; Lippert, Martin; Pechau, Jörg (2004): Ontologies as Software Engineering Artifacts. Ganztägiger Konferenz-Workshop. In: OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM.
- Becker-Pechau, Petra; Pechau, Jörg (2003a): How to use Ontologies and Modularization to Explicitly Describe the Concept Model of a Software Systems Architecture. Ganztägiger Konferenz-Workshop. In: OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM.
- Becker-Pechau, Petra; Pechau, Jörg (2003b): Mapping of Architectural Elements into Objectoriented Software Systems. In: Laukkanen, S.; Sarpola, S. (Hg.): IRIS '03: Proceedings of the 26th Information Systems Research Seminar in Scandinavia. New Ways of Working in IS. Haikko Manor, Porvoo, Finland.

- Behrens, Jan; Giesecke, Simon; Jost, Henning; Matevska, Jasminka; Schneider, Ulf (2009): Architekturbeschreibung. In: Reussner, Ralf; Hasselbring, Wilhelm (Hg.): Handbuch der Software-Architektur. 2. Auflage. Heidelberg, Deutschland: dpunkt-Verlag, S. 33-68.
- Beneken, Gerd (2009): Referenzarchitekturen. In: Reussner, Ralf; Hasselbring, Wilhelm (Hg.): Handbuch der Software-Architektur. 2. Auflage. Heidelberg, Deutschland: dpunkt-Verlag, S. 345-358.
- Bischofberger, Walter; Kühl, Jan; Löffler, Silvio (2004): Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In: Oquendo, Flavio; Warboys, Brian; Morrison, Ron (Hg.): EWSA '04: First European Workshop on Software Architecture. Berlin, Heidelberg, Deutschland: Springer (Lecture Notes in Computer Science, 3047), S. 1-9.
- Bittencourt, Roberto A.; de Souza Santos, Gustavo J.; Guerrero, Dalton D. S.; Murphy, Gail C. (2010): Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques. In: Antoniol, Giuliano; Pinzger, Martin; Chikofsky, Elliot (Hg.): WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering. Los Alamitos, CA, USA: IEEE Computer Society, S. 163-172.
- Bortz, Jürgen; Döring, Nicola (2006): Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler. 4. Auflage. Heidelberg, Deutschland: Springer.
- Broy, Manfred; Reussner, Ralf (2010): Architectural Concepts in Programming Languages. In: *Computer* 43 (10), S. 88-91.
- Brügge, Bernd; Dutroit, Allen H. (2004): Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java. München, Deutschland: Pearson Studium.
- Buchgeher, Georg; Weinreich, Rainer (2008): Integrated Software Architecture Management and Validation. In: Mannaert, Herwig; Ohta, Tadashi; Dini, Cosmin; Pellerin, Robert (Hg.): ICSEA '08: Proceedings of the Third International Conference on Software Engineering Advances. Los Alamitos, CA, USA: IEEE Computer Society, S. 427-436.
- Buchgeher, Georg; Weinreich, Rainer (2009a): Connecting Architecture and Implementation. In: Meersman, Robert; Herrero, Pilar; Dillon, Tharam (Hg.): On the Move to Meaningful Internet Systems: OTM 2009 Workshops. Berlin, Heidelberg, Deutschland: Springer (Lecture Notes in Computer Science, 5872), S. 316-326.
- Buchgeher, Georg; Weinreich, Rainer (2009b): Software Architecture Engineering. In: Buchberger, Bruno; Affenzeller, Michael; Ferscha, Alois; Haller, Michael; Jebelean, Tudor; Klement, Erich Peter et al. (Hg.): Hagenberg Research. Berlin, Heidelberg, Deutschland: Springer, S. 200-214.
- Buchgeher, Georg; Weinreich, Rainer (2009c): Tool Support for Component-Based Software Architectures. In: APSEC '09: Proceedings of the 16th Asia-Pacific Software Engineering Conference. Washington, DC, USA: IEEE Computer Society, S. 127-134.

- Buchgeher, Georg; Weinreich, Rainer (2010a): An Approach for Combining Model-Based and Scenario-Based Software Architecture Analysis. In: Hall, Jon; Kaindl, Hermann; Lavazza, Luigi; Buchgeher, Georg; Takaki, Osamu (Hg.): ICSEA '10: Proceedings of the Fifth International Conference on Software Engineering Advances. Washington, DC, USA: IEEE Computer Society, S. 141-148.
- Buchgeher, Georg; Weinreich, Rainer (2010b): Tool Demonstration: A Toolkit for Architecture-Centric Software Development. In: PPPJ '10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. New York, NY, USA: ACM, S. 158-161.
- Buchgeher, Georg; Weinreich, Rainer (2011): Automatic Tracing of Decisions to Architecture and Implementation. In: WICSA '11: Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture. Los Alamitos, CA, USA: IEEE Computer Society, S. 46-55.
- Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael (1996): Pattern-Oriented Software Architecture. A System of Patterns. Chichester, England, New York, USA: Wiley.
- Budde, Reinhard; Züllighoven, Heinz (1990): Software-Werkzeuge in einer Programmierwerkstatt. Ansätze eines hermeneutisch fundierten Werkzeug- und Maschinenbegriffs. Dissertation, TU Berlin, 1989. München, Deutschland: Oldenbourg (Berichte der Gesellschaft für Mathematik und Datenverarbeitung, 182).
- Chikofsky, Elliot J.; Cross, James H., II (1990): Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 7 (1), S. 13-17.
- Cleland-Huang, Jane; Gotel, Orlena; Zisman, Andrea (2012): Software and Systems Traceability. New York, NY, USA: Springer.
- Clements, Paul; Bachmann, Felix; Bass, Len; Garlan, David; Ivers, James; Little, Reed; Nord, Robert; Stafford, Judith (2002): Documenting Software Architectures: Views and Beyond. Boston, MA, USA: Pearson Education.
- Clements, Paul; Shaw, Mary (2009): "The Golden Age of Software Architecture" Revisited. In: *IEEE Software* 26 (4), S. 70-72.
- Deissenboeck, Florian; Heinemann, Lars; Hummel, Benjamin; Juergens, Elmar (2010): Flexible Architecture Conformance Assessment with ConQAT. In: Kramer, Jeff; Bishop, Judith; Devanbu, Premkumar T.; Uchitel, Sebastian (Hg.): ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. New York, NY, USA: ACM, S. 247-250.
- Detten, Markus von; Meyer, Matthias; Travkin, Dietrich (2010): Reverse Engineering with the Reclipse Tool Suite. In: Kramer, Jeff; Bishop, Judith; Devanbu, Premkumar T.; Uchitel, Sebastian (Hg.): ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. New York, NY, USA: ACM, S. 299-300.

- Ducasse, Stephane; Pollet, Damien (2009): Software Architecture Reconstruction: A Process-Oriented Taxonomy. In: *IEEE Transactions on Software Engineering* 35, S. 573-591.
- Duszynski, Slawomir; Knodel, Jens; Lindvall, Mikael (2009): SAVE: Software Architecture Visualization and Evaluation. In: Winter, Andreas; Ferenc, Rudolf; Knodel, Jens: CSMR '09: Proceedings of the 13th European Conference on Software Maintenance and Reengineering. Los Alamitos, CA, USA: IEEE Computer Society, S. 323-324.
- Eick, Stephen G.; Graves, Todd L.; Karr, Alan F.; Marron, James S; Mockus, Audris (2001): Does Code Decay? Assessing the Evidence from Change Management Data. In: *IEEE Transactions on Software Engineering* 27 (1), S. 1-12.
- Evans, Eric (2004): Domain Driven Design: Tackling Complexity in the Heart of Software. Boston, MA, USA: Addison-Wesley.
- Feilkas, Martein; Ratiu, Daniel; Jürgens, Elmar (2009): The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study. In: ICPC '09: Proceedings of the IEEE 17th International Conference on Program Comprehension. Piscataway, NJ, USA: IEEE Computer Society, S. 188-197.
- Flyvbjerg, Bent (2006): Five Misunderstandings About Case-Study Research. In: *Qualitative Inquiry* 12 (2), S. 219-245.
- Fowler, Martin (2003): Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley.
- Franz Brosch; Heiko Koziol; Barbora Buhnova; Ralf Reussner (2011): Architecture-based Reliability Prediction with the Palladio Component Model. In: *Transactions on Software Engineering* 38 (6).
- Garlan, David; Allen, Robert; Ockerbloom, John (1994): Exploiting Style in Architectural Design Environments. In: Adrion, W. Richards and Wile, David (Hg.): FSE '94: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering. New York, NY, USA: ACM, S. 175-188.
- Garlan, David; Shaw, Mary (1993): An Introduction to Software Architecture. In: Ambriola, Vincenzo; Tortora, Genevieve (Hg.): Advances in Software Engineering and Knowledge Engineering. Singapore: World Scientific (Series on Software Engineering and Knowledge Engineering, Volume 2), S. 1-39.
- Garlan, David; Shaw, Mary (1994): An Introduction to Software Architecture. Carnegie Mellon University, USA (Technical Report CMU-CS-94-166). Online verfügbar unter www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html, zuletzt geprüft am 31.07.2014.

- Giesecke, Simon; Gottschalk, Michael; Hasselbring, Wilhelm (2010): The ArchMapper Approach to Architectural Conformance Checks: An Eclipse-based Tool for Style-oriented Architecture to Code Mappings. In: Wagner, Stefan; Broy, Manfred; Deissenboeck, Florian; Münch, Jürgen; Ligesmeyer, Peter (Hg.): SQMB '10: Tagungsband des 3. Workshops zur Software-Qualitätsmodellierung und -bewertung. TU München (Technical Report TUM-I1001), S. 71-80.
- Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad (2005): The Java(tm) Language Specification. 3. Auflage. Upper Saddle River, NJ, USA: Addison-Wesley (Java Series).
- Harold, Elliott Rusty; Means, W. Scott (2004): XML in a Nutshell. 3. Auflage. Sebastopol, CA, USA: O'Reilly.
- Hevner, Alan R.; March, Salvatore T.; Park, Jinsoo (2004): Design Science in Information Systems Research. In: *MIS Quarterly* Vol. 28 (No. 1), S. 75-105.
- Hochstein, Lorin; Lindvall, Mikael (2005): Combating Architectural Degeneration: A Survey. In: *Information and Software Technology* 47 (10), S. 643-656.
- Hofmeister, Christine; Nord, Robert; Soni, Dilip (2000): Applied Software Architecture. Readings, MA, USA: Addison-Wesley (Object Technology Series).
- IEEE-Standard-1471 (2000): IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. In: *IEEE Std 1471-2000 DOI: 10.1109/IEEESTD.2000.91944*, S. i-23.
- ISO/IEC/IEEE-Standard-42010 (2011): ISO/IEC/IEEE Standard: Systems and Software Engineering – Architecture Description. In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, S. 1-46. DOI: 10.1109/IEEESTD.2011.6129467.
- ISO/IEC-Standard-25010 (2010): ISO/IEC 25010 - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models. Online verfügbar unter www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en, zuletzt geprüft am 04.08.2014.
- Jacobson, Ivar (1992): Object-Oriented Software Engineering – A Use Case Driven Approach. Reading, MA, USA: Addison-Wesley.
- Jacobson, Ivar; Booch, Grady; Rumbaugh, James (1999): The Unified Software Development Process. Reading, MA, USA: Addison-Wesley (Object Technology Series).
- Jahnke, Jens H. (2009): Reverse Engineering von Software-Architekturbeschreibungen. In: Reussner, Ralf; Hasselbring, Wilhelm (Hg.): Handbuch der Software-Architektur. 2. Auflage. Heidelberg, Deutschland: dpunkt-Verlag, S. 199-211.
- Jepsen, Leif Obel; Mathiassen, Lars; Nielsen, Peter Axel (1989): Back to Thinking Mode – Diaries as a Medium for Effective Management of Information Systems Development Projects. In: *Behaviour & Information Technology* 8 (3), S. 207-217.

- Karstens, Bettina (2005): Regeln der WAM-Modellarchitektur. Diplomarbeit. Universität Hamburg, Deutschland.
- Kim, Jung Soo; Garlan, David (2006): Analyzing Architectural Styles with Alloy. In: ROSATEA '06: Proceedings of the ISSTA Workshop on Role of Software Architecture for Testing and Analysis. New York, NY, USA: ACM, S. 70-80.
- Knodel, Jens; Muthig, Dirk; Naab, Matthias; Lindvall, Mikael (2006): Static Evaluation of Software Architectures. In: CSMR '06: Proceedings of the 10th European Conference on Software Maintenance and Reengineering. Los Alamitos, CA, USA: IEEE Computer Society.
- Knodel, Jens; Muthig, Dirk; Rost, Dominik (2008): Constructive Architecture Compliance Checking - An Experiment on Support By Live Feedback. In: ICSM '08: Proceedings of the IEEE International Conference on Software Maintenance. Adelaide, Australia: IEEE Computer Society, S. 287-296.
- Knodel, Jens; Popescu, Daniel (2007): A Comparison of Static Architecture Compliance Checking Approaches. In: WICSA '07: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture. Washington, DC, USA: IEEE Computer Society, S. 12 ff.
- Konersmann, Marco; Durdik, Zoya; Goedicke, Michael; Reussner, Ralf H. (2013): Towards Architecture-centric Evolution of Long-living Systems (the ADVERT Approach). In: QoSA '13: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures. New York, NY, USA: ACM, S. 163-168.
- Koschke, Rainer (2000): Atomic Architectural Component Recovery for Program Understanding and Evolution. Dissertation. Universität Stuttgart, Deutschland.
- Koschke, Rainer (2005): Rekonstruktion von Software-Architekturen: Blickwinkel, Sichten, Ansichten und Aussichten. Ein Literatur- und Methoden-Überblick zum Stand der Wissenschaft. In: *Informatik – Forschung und Entwicklung* 19 (3): Springer.
- Koschke, Rainer; Simon, Daniel (2003): Hierarchical Reflexion Models. In: WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering. Washington, DC, USA: IEEE Computer Society, S. 36-45.
- Kruchten, Philippe B. (1995): The 4+1 View Model of Architecture. In: *IEEE Software* 12 (6), S. 42-50.
- Lilienthal, Carola (2008): Komplexität von Softwarearchitekturen. Stile und Strategien. Dissertation. Universität Hamburg, Deutschland.
- Lilienthal, Carola (2009): Architectural Complexity of Large-Scale Software Systems. In: Winter, Andreas; Ferenc, Rudolf; Knodel, Jens (Hg.): CSMR '09: Proceedings of the 13th European Conference on Software Maintenance and Reengineering. Los Alamitos, CA, USA: IEEE Computer Society, S. 17-26.

- Lindvall, Mikael; Muthig, Dirk (2008): Bridging the Software Architecture Gap. In: *Computer* 41 (6), S. 98-101.
- Maier, Mark W.; Emery, David; Hilliard, Rich (2001): Software Architecture: Introducing IEEE Standard 1471. In: *Computer* 34 (4), S. 107-109.
- March, Salvatore T.; Smith, Gerald F. (1995): Design and Natural Science Research on Information Technology. In: *Decision Support Systems* 15 (4), S. 251-266.
- Medvidović, Nenad; Taylor, Richard N. (2000): A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Transactions on Software Engineering* 26 (1), S. 70-93.
- Mesbah, Ali; van Deursen, Arie (2007): An Architectural Style for Ajax. In: WICSA '07: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture. Washington, DC, USA: IEEE Computer Society, S. 9 ff.
- Meyer, Bertrand (1997): Object-Oriented Software Construction. 2. Auflage. Upper Saddle River, NJ, USA: Prentice Hall.
- Murphy, Gail C.; Notkin, David; Sullivan, Kevin (1995): Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In: Kaiser, Gail E. (Hg.): FSE '95: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering. Washington, DC, USA: ACM, S. 18-28.
- Murphy, Gail C.; Notkin, David; Sullivan, Kevin (1997): Extending and Managing Software Reflexion Models. University of British Columbia, USA, Department of Computer Science (Technical Report TR-97-15).
- Murphy, Gail C.; Notkin, David; Sullivan, Kevin J. (2001): Software Reflexion Models: Bridging the Gap between Design and Implementation. In: *IEEE Transaction on Software Engineering* 27 (4), S. 364-380.
- Nagl, Manfred (1990): Softwaretechnik: methodisches Programmieren im Großen. Berlin, Deutschland: Springer.
- Naur, Peter (1983): Program Development Studies Based on Diaries. In: Green, Thomas R.; Payne, Stephen J.; Veer, Gerrit C. (Hg.): Psychology of Computer Use. London, England: Academic Press, S. 159-170.
- Özkan, Johanna; Özkan, Erdal (2004): Automatisierte Dokumentation. Diplomarbeit. Universität Hamburg, Deutschland.
- Parnas, David L. (1972): On the Criteria to be Used in Decomposing Systems into Modules. In: *Communications of the ACM* 15 (12), S. 1053-1058.
- Passos, Leonardo; Terra, Ricardo; Diniz, Renato; Valente, Marco Tulio; Mendonca, Nabor das Chagas (2010): Static Architecture Conformance Checking – An Illustrative Overview. In: *IEEE Software* 27 (5), S. 82-89.

- Perry, Dewayne E.; Wolf, Alexander L. (1992): Foundations for the Study of Software Architecture. In: *ACM Sigsoft, Software Engineering Notes* 17 (4), S. 40-52.
- Pohl, Klaus (2008): Requirements Engineering. Grundlagen, Prinzipien, Techniken. 2., korrigierte Auflage. Heidelberg, Deutschland: dpunkt-Verlag.
- Poppendieck, Mary; Poppendieck, Tom (2010): Leading Lean Software Development. Results Are Not the Point. 2. Auflage. Upper Saddle River, NJ, USA: Addison-Wesley (A Kent Beck Signature Book).
- Posch, Torsten; Birken, Klaus; Gerdorn, Michael (2011): Basiswissen Softwarearchitektur. Verstehen, entwerfen, wiederverwenden. 3. Auflage. Heidelberg, Deutschland: dpunkt-Verlag.
- Postma, André (2003): A Method for Module Architecture Verification and its Application on a Large Component-based System. In: *Information and Software Technology* 45 (4), S. 171-194.
- Rahimi, R.; Khosravi, R. (2010): Architecture Conformance Checking of Multi-Language Applications. In: AICCSA '10: Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications. Washington, DC, USA: IEEE Computer Society, S. 1-8.
- Raza, Aoun; Vogel, Gunther; Plödereder, Erhard (2006): Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In: Pinho, LuísMiguel; González Harbour, Michael (Hg.): Ada-Europe '06: Proceedings of the 11th International Conference on Reliable Software Technologies. Berlin, Heidelberg, Deutschland: Springer (Lecture Notes in Computer Science, 4006). S. 71.
- Reussner, Ralf; Hasselbring, Wilhelm (Hg.) (2009): Handbuch der Software-Architektur. 2. Auflage. Heidelberg, Deutschland: dpunkt-Verlag.
- Riaz, Mehwish; Sulayman, Muhammad; Naqvi, Husnain (2009): Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture. In: Ślęzak, Dominik, Kim, Tai-hoon, Kiumi, Akingbehin, Jiang, Tao, Verner, June; Abrahão, Silvia (Hg.): Advances in Software Engineering, Bd. 59. Berlin, Heidelberg, Deutschland: Springer (Communications in Computer and Information Science), S. 119-126.
- Richardson, Leonard; Ruby, Sam; Demmig, Thomas (2007): Web-Services mit REST. Frischer Wind für Web-Services durch REST. Köln, Deutschland: O'Reilly.
- Riebisch, Matthias (2009): Vorgehen bei der Architektur und Komponentenentwicklung. In: Reussner, Ralf; Hasselbring, Wilhelm (Hg.): Handbuch der Software-Architektur. 2. Auflage. Heidelberg, Deutschland: dpunkt-Verlag, S. 69-91.
- Riehle, Dirk; Züllighoven, Heinz (1995): A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In: Coplien, James O. und Vlissides, John M. (Hg.): PLoP '95: Proceedings of the First Conference on Pattern Languages of Program Design. Reading, MA, USA: Addison-Wesley, S. 9-42.

- Rosik, Jacek; Le Gear, Andrew; Buckley, Jim; Ali Babar, Muhammad (2008): An Industrial Case Study of Architecture Conformance. In: ESEM '08: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. New York, NY, USA: ACM, S. 80-89.
- Sangal, Neeraj; Jordan, Ev; Sinha, Vineet; Jackson, Daniel (2005): Using Dependency Models to Manage Complex Software Architecture. In: OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. San Diego, CA, USA: ACM, S. 167-176.
- Sangwan, Raghvinder S.; Vercellone-Smith, Pamela; Laplante, Phillip A. (2008): Structural Epochs in the Complexity of Software over Time. In: *IEEE Software* 25 (4), S. 66-73.
- Scharping, Arne (2005): Architekturvereinbarungen des JCommSys. Baccalaureatsarbeit. Universität Hamburg, Deutschland.
- Scharping, Arne (2008): Automatisierte Prüfung von Architekturregeln zur Entwicklungszeit. Diplomarbeit. Universität Hamburg, Deutschland.
- Shaw, Mary; Clements, Paul (2006): The Golden Age of Software Architecture. In: *IEEE Software* 23 (2), S. 31-39.
- Shaw, Mary; Garlan, David (1996): Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, NJ, USA: Prentice-Hall.
- Shi, Nija; Olsson, Ronald A. (2006): Reverse Engineering of Design Patterns from Java Source Code. In: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. Los Alamitos, CA, USA: IEEE Computer Society, S. 123-134.
- Siedersleben, Johannes (Hg.) (2003a): Quasar: Die sd&m Standardarchitektur. Teil 1. 2. Auflage. Technischer Bericht, sd&m Research, Deutschland. Online verfügbar unter www.fbi.h-da.de/fileadmin/personal/b.humm/Publicationen/Siedersleben_-_Quasar_1__sd_m_Brosch_re_.pdf, zuletzt geprüft am 31.07.2014.
- Siedersleben, Johannes (Hg.) (2003b): Quasar: Die sd&m Standardarchitektur. Teil 2. 2. Auflage. Technischer Bericht, sd&m research, Deutschland. Online verfügbar unter www.fbi.h-da.de/fileadmin/personal/b.humm/Publicationen/Siedersleben_-_Quasar_2__sd_m_Brosch_re_.pdf, zuletzt geprüft am 31.07.2014.
- Siedersleben, Johannes (2004): Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar. Heidelberg, Deutschland: dpunkt-Verlag.
- Silva, Lakshitha de; Balasubramaniam, Dharini (2012): Controlling Software Architecture Erosion: A Survey. In: *Journal of Systems and Software* 85 (1), S. 132-151.
- Simon, Frank; Meyerhoff, Dirk (2002): OO-Metriken zeigen grosse Qualitätspotenziale in komplexen Softwaresystemen. In: *Objektspektrum* (6), S. 28-35.

- Sommerville, Ian (2001): *Software Engineering*. 6. Auflage. Harlow, England: Addison-Wesley (International Computer Science Series).
- Soni, Dilip; Nord, Robert; Hofmeister, Christine (1995): *Software Architecture in Industrial Applications*. In: *ICSE '95: Proceedings of the 17th International Conference on Software Engineering*. Seattle, WA, USA: ACM, S. 196-207.
- Stachowiak, Herbert (1973): *Allgemeine Modelltheorie*. Wien, Österreich: Springer.
- Taylor, Richard N.; Medvidović, Nenad; Dashofy, Eric M. (2009): *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ, USA: Wiley.
- Trifu, Mircea; Szulman, Peter (2005): *Language Independent Abstract Metamodel for Quality Analysis and Improvement of OO Systems*. In: *WSR 05: Proceedings of the 7th German Workshop on Software-Reengineering*. Bonn, Deutschland: Gesellschaft für Informatik (GI) (Softwaretechnik-Trends, 25-2).
- Vaishnavi, V.; Kuechler, W. (2004): *Design Research in Information Systems*. Association for Information Systems (AIS). Online verfügbar unter desrist.org/design-research-in-information-systems, zuletzt aktualisiert am 23.10.2013, zuletzt geprüft am 31.07.2014.
- van Eyck, Jo; Boucké, Nelis; Helleboogh, Alexander; Holvoet, Tom (2011): *Using Code Analysis Tools for Architectural Conformance Checking*. In: *SHARK '11: Proceedings of the 6th International Workshop on SHARING and Reusing Architectural Knowledge*. New York, NY, USA: ACM, S. 53-54.
- Verbaere, M.; Godfrey, M.W; Girba, T. (2008): *Query Technologies and Applications for Program Comprehension (QTAPC 2008)*. In: Krikhaar, René; Lämmel, Ralf; Verhoef, Chris (Hg.): *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society, S. 285-288.
- Weinreich, Rainer; Buchgeher, Georg (2010): *Paving the Road for Formally Defined Architecture Description in Software Development*. In: *SAC '10: Proceedings of the 25th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, S. 2337-2343.
- Weinreich, Rainer; Buchgeher, Georg (2012): *Towards Supporting the Software Architecture Life Cycle*. In: *Journal of Systems and Software* 85 (3), S. 546-561.
- Weinreich, Rainer; Miesbauer, Cornelia; Buchgeher, Georg; Kriechbaum, Thomas (2012): *Extracting and Facilitating Architecture in Service-Oriented Software Systems*. In: Ali Babar, Muhammad; Cuesta, Carlos; Savolainen, Juha; Männistö, Tomi: *WICSA/ECSA '12: Proceedings of the 10th Joint Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture*. Los Alamitos, CA, USA: IEEE Computer Society, S. 81-90.
- Winter, Robert (2008): *Design Science Research in Europe*. In: *European Journal of Information Systems* 17 (5), S. 470-475.

Wirfs-Brock, Rebecca; McKean, Alan (2007): Object Design. Roles, Responsibilities, and Collaborations. 3. Auflage. Boston, MA, USA: Addison-Wesley.

Züllighoven, Heinz (1998): Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz. Heidelberg, Deutschland: dpunkt-Verlag.

Züllighoven, Heinz (2005): Object-Oriented Construction Handbook. San Francisco, CA, USA: Morgan Kaufmann Publishers.

Züllighoven, Heinz; Raasch, Jörg (2006): Softwaretechnik. In: Rechenberg, Peter und Pomberger, Gustav (Hg.): Informatik-Handbuch. 4. Auflage. München, Deutschland, Wien, Österreich: Hanser-Verlag, S. 795–837.

Abbildungsverzeichnis

1	Beispiel: Quelltextbezeichner helfen, die Architektur im Quelltext zu erkennen	14
2	Architekturelemente umfassen Quelltextelemente	21
3	Aggregation von Beziehungen aus der Quelltextstruktur in die Architektur.....	22
4	Beziehungen zwischen Architekturelementen.....	22
5	Architekturelemente besitzen eine öffentliche Schnittstelle und private Interna	23
6	Allgemeines Metamodell für Architekturen	26
7	UML-Darstellung einer Softwarearchitektur aus der Versicherungswirtschaft	27
8	Ausschnitt aus der Referenzarchitektur für Compiler	29
9	Eine nach dem Client-Server-Stil strukturierte Architektur	32
10	Client-Server-Architekturstil	33
11	Schematische Darstellung des WAM-Architekturstils	38
12	Eine WAM-Architektur	39
13	Der Quasar-Architekturstil	41
14	Eine Quasar-Architektur.....	44
15	Das allgemeine Metamodell für Architekturen	45
16	Ausschnitt aus einer stilbasierten Architektur nach WAM	47
17	Architektur-Beispiel	48
18	Quelltext- und Architekturebene am Beispiel der Patientenmappe	52
19	Kernaufgabe von Prüfungen auf Architekturtreue.....	54
20	Bevor geprüft werden kann, muss die Ist-Architektur berechnet werden.	55
21	Überblick über die drei Kern-Arbeitsschritte von Prüfungen auf Architekturtreue	56
22	Die Soll-Architektur des SRM-Ansatzes (UML-Darstellung)	58
23	Die Soll-Architektur legt Architekturelemente und Beziehungen fest (Beispiel)	58
24	Die Quelltextstruktur (UML-Darstellung).....	59
25	Zuordnung Z_{QA} (Beispiel)	59
26	Zuordnung Z_{QA} (UML-Darstellung).....	60
27	Überblick: Ist-Architektur ermitteln (Beispiel)	61
28	Ist-Architektur, Quelltextstruktur und Zuordnung Z_{QA} (UML-Darstellung).....	61
29	Quelltextbeziehung (Beispiel)	62
30	Beziehungen ermitteln: Ausgangssituation (Beispiel).....	62
31	Resultierende Beziehung in der Ist-Architektur (Beispiel).....	63
32	Software-Reflexionsmodell (Beispiel)	64
33	Aggregation getypter Beziehungen (Beispiel).....	66
34	Beispiel für ein getyptes Software-Reflexionsmodell	66

35	Eine hierarchische Soll-Architektur.....	67
36	Eine DSM (nach Sangal et al. 2005)	68
37	Eine DSM mit Vorgaben und Fehlermarkierungen (nach Sangal et al. 2005)	69
38	Grafische Darstellung von Schichten und Schnitten in Sonargraph.....	72
39	Grafische Darstellung der Prüfergebnisse in Sotoarc	73
40	Bei Soll-Architekturen wird jedes Architekturelement und jede Beziehung einzeln aufgezählt.....	86
41	Zwei Architekturelement- <i>Typen</i> und eine zugehörige <i>Regel</i> in einem <i>Architekturstil</i>	87
42	Ausschnitt aus einer <i>stilbasierten</i> Architektur.....	87
43	Die Vorgaben eines Stils gelten für alle Systeme, die diesem Stil folgen.....	88
44	Grundprinzip der stilbasierten Architekturprüfung	89
45	Grundprinzip der stilbasierten Architekturprüfung	91
46	Ermittlung der stilbasierten Ist-Architektur.....	92
47	Quelltext in der Programmiersprache Java.....	93
48	Die Zusatzinformationen stellen den Zusammenhang zwischen Quelltext und Architektur her.....	96
49	Grafische Veranschaulichung der Quelltextstruktur.....	97
50	Resultierende stilbasierte Ist-Architektur	98
51	Grundprinzip der stilbasierten Architekturprüfung	101
52	UML-Darstellung der Struktur von Architekturstil-Beschreibungen	102
53	Beispielhafter Ausschnitt aus dem WAM-Stil.....	103
54	Die Berechnung der Verstöße der Ist-Architektur im Gesamtkontext der stilbasierten Architekturprüfung.....	104
55	Die Berechnung der Verstöße (c) und der betroffenen Quelltextelemente (d) im Gesamtkontext der stilbasierten Architekturprüfung.....	105
56	Beispiel: eine zu prüfende Ist-Architektur.....	106
57	Zwischenergebnis der Berechnung, die Hilfsrelation H_{BV}	107
58	Eine Beziehung verstößt gegen die gegebenen Verbotsregeln	109
59	Der Patientenaufnehmer kennt ein Material. Diese Information ist Teil der Hilfsrelation HI_{AT}	111
60	Verstoß gegen eine Verbotsregel.....	113
61	Zuordnung zwischen Quelltextelementen und Architekturelementen.....	113
62	Das Quelltextelement TherapieplanMaterial verstößt gegen den Stil	114
63	Das Architekturelement Therapieplaner gehört zu den Quelltextelementen TherapieplanerUI und TherapieplanerMonoTool.....	116
64	Die vier Berechnungsschritte der stilbasierten Architekturprüfung	119
65	Eine Vererbungsbeziehung von B zu A in der Sprache Java.....	120

66	Eine Benutzbeziehung von C zu D in der Sprache Java.....	121
67	Aggregation von Benutzt- und Vererbungsbeziehungen.....	121
68	Beispiel für einen Verstoß gegen die Schnittstellenregel „Funktionskomponenten (functional parts) dürfen keine Materialien zurückgeben“	127
69	Ein Quelltextbeispiel, angelehnt an ein reales System	129
70	Ein Quelltextbeispiel zur Illustration der Quelltextstruktur.....	131
71	Beispiel: Quelltextelement und Architekturelement.....	132
72	Ein Architekturelement bestehend aus zwei Quelltextelementen.....	133
73	Quelltext der Klassen Behandlung und Behandlungsabschnitt	134
74	Schnittstellen berechnen (Beispiel)	135
75	Beispiel: Zuordnung zwischen Quelltext und Architekturelement.....	137
76	Quelltextausschnitt der Klasse SingleDeviceEditorFP.....	137
77	Zuordnung zwischen Quelltext und Architektur für die Klasse Device	138
78	Architekturelemente mit Schnittstelle	142
79	Beispiel eines hierarchischen Architekturelement-Typs	143
80	Ein zusammengesetztes Architekturelement	144
81	Beispiel einer internen Gebotsregel.....	144
82	Hierarchisch geschachtelte Architekturelemente.....	145
83	Ein zusammengesetztes Architekturelement	147
84	Interne Gebotsregel des WAM-Stils.....	149
85	Eine beispielhafte Ist-Architektur.....	150
86	Beispiel für einen zusammengesetzten Architekturelement-Typ	159
87	Ein zusammengesetztes Architekturelement	159
88	Der StileBasedChecker nutzt den XML-Editor der Eclipse-IDE für Stilbeschreibungen.....	166
89	Konkrete Schnittstellenregeln implementieren das Interface SchnittstellenRegel	167
90	Die Klasse Device mit annotierten Zusatzinformationen	167
91	Die Verstöße werden in der Eclipse-IDE als sogenannte Probleme dargestellt	168
92	Aufbau des StyleBasedCheckers	169
93	Interaktion der Design-Forschung mit der Problemumgebung und der Wissensbasis	177
94	Iterative Forschungstätigkeiten in der Design-Forschung	178

Symbolliste

Symbole für grundlegende Definitionen:

$\langle A, B_A, S_A, Z_{AS} \rangle$	Softwarearchitektur (siehe Definition 2-1, S. 17).
A	Menge der Architekturelemente.
$B_A \subseteq A \times A$	Menge der gerichteten Beziehungen zwischen Architekturelementen.
S_A	Menge der Architekturelement-Schnittstellen.
$Z_{AS} \subseteq A \times S_A$	Zuordnung zwischen Architekturelementen und deren Schnittstellen.
$\langle Q, B_Q, S_Q, Z_{QS} \rangle$	Quelltextstruktur (siehe Definition 2-6, S. 20).
Q	Menge der Quelltextelemente.
$B_Q \subseteq Q \times Q$	Menge der gerichteten Beziehungen zwischen Quelltextelementen.
S_Q	Menge der Schnittstellen der Quelltextelemente.
$Z_{QS} \subseteq Q \times S_Q$	Zuordnung zwischen Quelltextelementen und deren Schnittstellen.
$Z_{QA} \subseteq Q \times A$	Zuordnung zwischen Quelltext- und Architekturelementen. Diese Zuordnung stellt die Verbindung zwischen der Quelltextstruktur und der Softwarearchitektur her.
$\langle T_A, T_B, R_S, R_B \rangle$	Architekturstil (siehe Definition 3-2, S. 35).
T_A	Menge der Architekturelement-Typen.
T_B	Menge der Beziehungstypen.
R_S	Menge der Schnittstellenregeln.
R_B	Menge der Beziehungsregeln, diese unterteilt sich in folgende Regeln: R_G, R_K, R_V, R_E .

$R_G \subseteq T_A \times T_A$	Gebots-Beziehungsregeln, kurz Gebotsregeln, für jeden Beziehungstyp T_B .
$R_K \subseteq T_A \times T_A$	Kann-Beziehungsregeln, kurz erlaubte Beziehungen, für jeden Beziehungstyp T_B .
$R_V \subseteq T_A \times T_A$	Verbots-Beziehungsregeln, kurz Verbotsregeln, für jeden Beziehungstyp T_B .
$R_E \subseteq T_A \times T_A$	Enthält-Beziehungsregeln, auch als Elementaufbau-Regeln bezeichnet.
$\langle A, B_A, S_A, Z_{AS}, Z_{AT} \rangle$	Stilbasierte Architektur (siehe Definition 3-4, S. 36). (A, B_A, S_A, Z_{AS} wie oben, Definition Softwarearchitektur).
$Z_{AT} \subseteq A \times T_A$	Zuordnung zwischen Architekturelementen und deren Typen T_A . Hierbei bezeichnet T_A die im zugehörigen Architekturstil enthaltene Menge der Architekturelement-Typen.

Symbole für das Kernkonzept der stilbasierten Architekturprüfung:

$V_{VR} \subseteq A \times A$	Verstöße der Ist-Architektur gegen Verbotsregeln.
$H_{BV} \subseteq A \times A$	Hilfsrelation zur Berechnung der Verstöße gegen Verbotsregeln. Enthält alle Beziehungen, die laut Verbotsregeln untersagt sind.
$V_{QVR} \subseteq Q$	Menge der betroffenen Quelltextstellen zu den Verstößen gegen Gebotsregeln V_{VR} .
$V_{GR} \subseteq A \times T_A$	Verstöße der Ist-Architektur gegen Gebotsregeln.
$HS_{AT} \subseteq A \times T_A$	Erste Hilfsrelation zur Berechnung der Verstöße gegen Gebotsregeln. Die Relation enthält für jedes Architekturelement die Typen der Architekturelemente, zu denen Beziehungen bestehen <i>sollen</i> .
$HI_{AT} \subseteq A \times T$	Zweite Hilfsrelation zur Berechnung der Verstöße gegen Gebotsregeln. Die Relation listet für jedes Architekturelement auf, zu welchen Typen Beziehungen innerhalb der Ist-Architektur bestehen.
$V_{QGR} \subseteq Q$	Menge der betroffenen Quelltextstellen zu den Verstößen gegen Gebotsregeln V_{GR} .

Symbole für die Ausbaustufe „Beziehungstypen“:

Der hochgestellte Index m steht jeweils für den Beziehungstyp, mit $1 \leq m \leq n$ und $n = |T_B|$ (Mächtigkeit der Menge der Beziehungstypen). Als Index kann entweder die Bezeichnung des Beziehungstyps angegeben werden, wie beispielsweise $B_Q^{Vererbung}$, oder es können die Beziehungstypen nummeriert werden, wie beispielsweise B_Q^1 .

In der Folgenden Liste steht X_Y^m kurz dafür, dass die Mengen $X_Y^1, X_Y^2, \dots, X_Y^n$ existieren.

$B_Q^m \subseteq Q \times Q$	Mengen der Quelltextbeziehungen.
$B_A^m \subseteq A \times A$	Mengen der Architekturelement-Beziehungen.
$R_G^m \subseteq T_A \times T_A$	Mengen der Gebotsregeln.
$R_V^m \subseteq T_A \times T_A$	Mengen der Verbotsregeln.
$V_{GR}^m \subseteq A \times T_A$	Mengen der Verstöße gegen Gebotsregeln.
$V_{VR}^m \subseteq A \times A$	Mengen der Verstöße gegen Verbotsregeln.
$V_{QGR}^m \subseteq Q$	Die betroffenen Quelltextstellen der Verstöße gegen Gebotsregeln.
$V_{QVR}^m \subseteq Q$	Die betroffenen Quelltextstellen der Verstöße gegen Verbotsregeln.

Symbole für die Ausbaustufe „Schnittstellenregeln“:

S_A	Menge der Architekturelement-Schnittstellen. Für jedes $sa \in S_A$ gilt: $sa \subseteq S_Q$, dabei bezeichnet S_Q die Menge der Quelltextelement-Schnittstellen.
$Z_{TSR} \subseteq T_A \times R_S$	Zuordnung der Architekturelement-Typen zu Schnittstellenregeln. Legt fest, für welche Typen die Regeln gelten.
$V_{SR} \subseteq A$	Menge der Verstöße pro Schnittstellenregel.
$H_{SSR} \subseteq S_A \times R_S$	Hilfsrelation zur Berechnung der Verstöße gegen Schnittstellenregeln. Legt für alle Architekturelement-Schnittstellen fest, welche Schnittstellenregeln für sie gelten.
$V_{QSR} \subseteq Q$	Menge der betroffenen Quelltextstellen zu den Verstößen gegen Schnittstellenregeln.

Symbole für die Ausbaustufe „hierarchische Architekturen“:

$R_E \subseteq T_A \times T_A$	Menge der Enthältregeln.
$V_{ER} \subseteq A \times T_A$	Menge der Verstöße gegen Enthältregeln.
$H_{ET} \subseteq A \times T_A$	Hilfsrelation zur Berechnung der Verstöße gegen Enthältregeln und gegen interne Gebotsregeln. Berechnet sich ausschließlich aus der Architektur, gibt für jedes Architekturelement wieder, welche Typen bei den direkt enthaltenen Architekturelementen vorkommen. Der Index ET steht für enthaltene Typen.
$V_{QER} \subseteq Pot(Q)$	Menge der von Verstößen gegen Enthältregeln betroffenen Quelltextstellen. $Pot(Q)$ bezeichnet dabei die Potenzmenge von Q .
$R_{IG} \subseteq T_A \times T_A \times T_A$	Menge der internen Gebotsregeln. Für jedes Tupel $(t1, t2, t3) \in R_{IG}$ gilt: $t1$ bezeichnet den Typ zusammengesetzter Architekturelemente. $t2$ und $t3$ bezeichnen den Typ direkt enthaltener Elemente, wobei Elemente des Typs $t2$ eine Beziehung zu Elementen des Typs $t3$ haben sollen.
$V_{IG} \subseteq A \times A \times T_A$	Verstöße gegen interne Gebotsregeln.
$H_{ISB} \subseteq A \times A \times T_A$	Zweite Hilfsrelation (neben H_{ET}) zur Berechnung der Verstöße gegen interne Gebotsregeln, die Tupel enthalten jeweils ein zusammengesetztes Architekturelement, ein enthaltenes Architekturelement, von dem laut einer internen Gebotsregel eine Beziehung ausgehen soll, sowie den Typ, den das Architekturelement haben soll, zu dem die vorgeschriebene Beziehung verlaufen soll. Der Index ISB steht für interne Soll-Beziehungen.
$H_{IIB} \subseteq A \times A \times T_A$	Dritte Hilfsrelation zur Berechnung der Verstöße gegen interne Gebotsregeln. Ein Tupel (a, b, t) bedeutet: in der Architektur existiert ein zusammengesetztes Architekturelement a , dieses enthält ein Architekturelement b und von b existiert eine Beziehung zu einem ebenfalls in a enthaltenen Architekturelement des Typs t . Der Index IIB steht für interne Ist-Beziehungen.
$V_{QIG} \subseteq Q$	Menge der betroffenen Quelltextstellen zu den Verstößen gegen interne Gebotsregeln.

Anhang A: Grammatiken der prototypischen Umsetzung

A.1 Architekturstil-Beschreibung

XML-Schema für die Beschreibung von Architekturstilen

XML-Schemata beschreiben die prinzipielle Struktur von XML-Dateien (wie Grammatiken für formale Sprachen). Das im Folgenden aufgeführte Schema unterstützt das Kernkonzept der stilbasierten Architekturprüfung sowie die Erweiterungen für hierarchische Architekturen und Schnittstellenregeln. Die in Abschnitt 7.1.3 beispielhaft vorgestellte Stilbeschreibung basiert auf diesem Schema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="architekturstilDefinition">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="architekturelementTypen" minOccurs="0" maxOccurs="1" />
        <xs:element ref="regeln" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="stilBezeichnung" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

<xs:key name="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="elementArtDefinition" />
  <xs:field xpath="@elementArt" />
</xs:key>

<xs:keyref name="gebotsregelVonIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="gebotsregel/@von" />
</xs:keyref>

<xs:keyref name="gebotsregelZuIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="gebotsregel/@zu" />
</xs:keyref>

<xs:keyref name="verbotsregelVonIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="verbotsregel/@von" />
</xs:keyref>

<xs:keyref name="verbotsregelZuIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="verbotsregel/@zu" />
</xs:keyref>

```

```

<xs:keyref name="schnittstellenregelTypIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="schnittstellenregel/@typ" />
</xs:keyref>

<xs:unique name="aufbauregelTypZusammengesetztIstUnique">
  <xs:selector xpath="regeln" />
  <xs:field xpath="aufbauRegel/@typZusammengesetzt" />
</xs:unique>

<xs:keyref name="aufbauregelTypEnthaltenIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="aufbauRegel/enthaelt/@typ" />
</xs:keyref>

<xs:keyref name="interneGebotsregelTypZusammengesetztIstReferenz"
              refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="interneGebotsregel/@typZusammengesetzt" />
</xs:keyref>

<xs:keyref name="interneGebotsregelVonIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
  <xs:selector xpath="regeln" />
  <xs:field xpath="interneGebotsregel/@von" />
</xs:keyref>

```

```

    <xs:keyref name="interneGebotsregelZuIstReferenz" refer="architekturelementTypBezeichnungIstUniqueKey">
      <xs:selector xpath="regeln" />
      <xs:field xpath="interneGebotsregel/@zu" />
    </xs:keyref>

  </xs:element>

  <xs:element name="architekturelementTypen">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="typ" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="typ">
    <xs:complexType>
      <xs:attribute name="typBezeichnung" type="XMLTypeFuerBezeichnungen" use="required" />
    </xs:complexType>
  </xs:element>

```



```
<xs:element name="regeln">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="gebotsregel" minOccurs="0" maxOccurs="unbounded" />
      <xs:element ref="verbotsregel" minOccurs="0" maxOccurs="unbounded" />
      <xs:element ref="schnittstellenregel" minOccurs="0" maxOccurs="unbounded" />
      <xs:element ref="aufbauregel" minOccurs="0" maxOccurs="unbounded" />
      <xs:element ref="interneGebotsregel" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="gebotsregel">
  <xs:complexType>
    <xs:attribute name="von" type="XMLTypeFuerBezeichnungen" use="required" />
    <xs:attribute name="zu" type="XMLTypeFuerBezeichnungen" use="required" />
  </xs:complexType>
</xs:element>
```

```
<xs:element name="verbotsregel">
  <xs:complexType>
    <xs:attribute name="von" type="XMLTypeFuerBezeichnungen" use="required" />
    <xs:attribute name="zu" type="XMLTypeFuerBezeichnungen" use="required" />
  </xs:complexType>
</xs:element>
```

```

<xs:element name="schnittstellenregel">
  <xs:complexType>
    <xs:attribute name="typ" type="XMLTypeFuerBezeichnungen" use="required" />
    <xs:attribute name="regelName" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="aufbauregel">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="enthaelt" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="typ" type="XMLTypeFuerBezeichnungen" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="typZusammengesetzt" type="XMLTypeFuerBezeichnungen" use="required" />
  </xs:complexType>
</xs:element>

```

```
<xs:element name="interneGebotsregel">
  <xs:complexType>
    <xs:attribute name="typZusammengesetzt" type="XMLTypeFuerBezeichnungen" use="required" />
    <xs:attribute name="von" type="XMLTypeFuerBezeichnungen" use="required" />
    <xs:attribute name="zu" type="XMLTypeFuerBezeichnungen" use="required" />
  </xs:complexType>
</xs:element>

<xs:simpleType name="XMLTypeFuerBezeichnungen">
  <xs:restriction base="xs:string">
    <xs:minLength value="1" />
    <xs:maxLength value="128" />
    <xs:pattern value="[\w_]{1,128}" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```


A.2 Quelltext-Annotation

EBNF für die Annotationen der Zusatzinformationen in Java

Wie in Abschnitt 7.1.2 beschrieben, verwendet die prototypische Umsetzung mit dem StyleBasedChecker Java-Annotationen, um die Zusatzinformationen in den Quelltext zu integrieren. Hierfür wurde mit den bestehenden Java-Sprachmechanismen ein eigener Annotationstyp namens *Architekturelement* definiert. Diese Art der Annotation konkretisiert exemplarisch das Konzept der stilbasierten Architekturprüfung für Java. Die Programmiersprache Java wurde hierfür nicht verändert.

Zur Veranschaulichung wird an dieser Stelle die Grammatik der Programmiersprache spezialisiert, um die Syntax der im StyleBasedChecker verwendeten Annotationen deutlich zu machen.

Da die untersuchten Systeme mit der Java Version SE 6 oder kompatiblen Versionen umgesetzt wurden, basiert die hier vorgestellte Grammatik auf der gleichen Java-Version (Java Language Specification (JLS) Gosling et al. 2005). Es wäre problemlos möglich, die selbe Art der Notation in der aktuellen Java-Version zu verwenden.

Die hier vorgestellte Grammatik ist in die Java-Programmiersprache integriert. Die ersten drei Produktionen sind aus der Java-Programmiersprache entnommen. Die Produktionen wurden leicht äquivalent umgeformt, um die hier ergänzten Annotationen kompakt und lesbar einfügen zu können. Die Grammatik zeigt und modifiziert nur den Annotationen betreffenden Ausschnitt der Java-Grammatik.

Diese Grammatik verwendet die folgenden EBNF-Konventionen:

[x] - x kann einmalig oder gar nicht auftreten.

{ x } - x kann beliebig oft auftreten (auch gar nicht)

(x | y) Alternativen werden durch einen senkrechten Strich getrennt und ggf. in runde Klammern gesetzt oder sie stehen jeweils auf eigenen Zeilen

Terminale sind **blau**, Nicht-Terminale sind in *Kursiv* gesetzt.

// Kommentare werden durch zwei Schrägstriche eingeleitet.

//Diese Produktionen stammen aus der JLS. Weggelassene Alternativen aus der JLS-Grammatik
//bleiben weiterhin unverändert bestehen.

ClassOrInterfaceDeclaration:

Modifiers (ClassDeclaration | InterfaceDeclaration)

Modifiers:

Annotations

Annotations:

{ @ TypeName ({ ElementValuePairs }) }

ElementValuePairs

Identifier = ElementValue

Identifier = ElementValue , ElementValuePairs

//Diese Produktionen wurden ergänzt:

Annotations:

```
{ @ Architekturelement ( name = ElementName , typ = ElementTyp ) }
```

ElementTyp:

StringLiteral

ElementName:

StringLiteral

StringLiteral:

// Eine Zeichenkette, siehe JLS „Lexical Structure, String Literals“ (Gosling et al. 2005)

Anhang B: Regellisten und Typen des WAM- und Quasar-Stils

Im Folgenden sind die in dieser Arbeit ermittelten Regellisten des WAM- und des Quasar-Stils aufgeführt. Es sei darauf hingewiesen, dass diese Listen alle Regeln enthalten, welche in den praktischen Untersuchungen und in den Interviews für diese Arbeit als besonders relevant betrachtet wurden, jedoch keinen Anspruch auf Vollständigkeit erheben. Für eine Erläuterung der Stile und der enthaltenen Architekturelement-Typen siehe Abschnitt 3.3.

Hinweise zur Notation

Verwendung von Oberbegriffen: Teilweise werden Oberbegriffe verwendet, die für mehrere Typen stehen. Beispielsweise steht der Begriff des Werkzeugs im WAM-Stil für Mono-Werkzeuge und komplexe Werkzeuge. Die Verbotsregel „Automaten dürfen keine Werkzeuge kennen“ bedeutet somit detailliert ausgedrückt: „Automaten dürfen keine komplexen Werkzeuge kennen“ und „Automaten dürfen keine MonoWerkzeuge kennen“.

Verbotsregeln: das Verbot bezieht sich bei hierarchischen Architekturelementen auch auf enthaltene Elemente. Beispielsweise bedeutet die Verbotsregel „Automaten dürfen keine komplexen Werkzeuge kennen“, dass Automaten auch keine in einem komplexen Werkzeug enthaltene Funktionskomponente kennen dürfen.

B.1 Quasar-Stil

Architekturelement-Typen

- Anwendungs-Entitätstyp (kurz A-Entitätstyp oder Entitätstyp)
- Anwendungsfall (kurz A-Fall)
- Anwendungs-Datentyp (kurz A-Datentyp oder Datentyp)
- Präsentation
- Dialogkern
- Sitzungssteuerung
- Sitzungsverwaltung
- Kompositionsmanager
- Dialogmanager (=spezieller Kompositionsmanager)

Typen zusammengesetzter Architekturelemente

- GUI-Engine
- Dialog
- Komponente: zusammengesetzte Elemente werden als Komponenten bezeichnet.
- Anwendungs-Komponente (kurz A-Komponente): Komponenten, die Entitätstypen, Datentypen und / oder Anwendungsfälle enthalten.
- Anwendungskern: Als Anwendungskern wird die oberste A-Komponente bezeichnet, d.h. die A-Komponente, die alle anderen A-Komponenten enthält.

Gebots-Beziehungsregeln

Vererbung zwischen verschiedenen Architekturelement-Typen ist nicht erlaubt, alle Gebots-Beziehungsregeln betreffen Benutzbeziehungen.

- Sitzungsverwaltungen müssen eine Sitzungssteuerung kennen.
- Sitzungssteuerungen müssen einen Dialogmanager kennen.
- Dialogmanager müssen eine Präsentation kennen.
- Dialogmanager müssen einen Dialogkern kennen.
- Präsentationen müssen einen Dialogkern kennen.
- Dialogkerne müssen einen Anwendungsfall kennen.
- Dialogkerne müssen einen Datentyp kennen.
- Anwendungsfälle müssen einen Entitätstyp kennen.
- Anwendungsfälle müssen einen Datentyp kennen.
- Entitätstypen müssen einen Datentyp kennen.

Verbots-Beziehungsregeln

- Kein Typ darf Kompositionsmanager kennen.
- Kein Typ außer Kompositionsmanagern darf Sitzungsverwaltungen kennen.
- Kein Typ außer Kompositionsmanagern und Sitzungsverwaltungen darf Sitzungssteuerungen kennen.
- Kein Typ außer Sitzungssteuerungen darf Dialogmanager kennen.
- Dialogkerne dürfen keine Präsentation kennen.

- Datentypen dürfen keine anderen Typen kennen.
- Entitätstypen dürfen keine anderen Typen kennen außer Datentypen.
- Kein A-Typ und keine A-Komponenten darf eine GUI-Engine kennen.

Enthält-Beziehungsregeln

- GUI-Engines enthalten eine Sitzungssteuerung.
- GUI-Engines enthalten eine Sitzungsverwaltung.
- GUI-Engines enthalten einen Dialog.
- Dialoge enthalten eine Präsentation.
- Dialoge enthalten einen Dialogkern.
- Dialoge enthalten einen Dialogmanager.
- A-Komponenten enthalten einen Entitätstyp.
- A-Komponenten enthalten einen A-Fall.
- A-Komponenten enthalten einen Datentyp.

Interne Gebotsregeln

- In Dialogen gilt: Präsentationen müssen einen Dialogkern kennen.
- In Dialogen gilt: Dialogmanager müssen eine Präsentation kennen.
- In Dialogen gilt: Dialogmanager müssen einen Dialogkern kennen.

Schnittstellenregeln

- Datentypen enthalten keine Setter.

B.2 WAM-Stil

Architekturelement-Typen

- Funktionskomponente (kurz FK)
- Interaktionskomponente (kurz IAK)
- GUI
- Tool
- MonoTool
- Automat
- Service
- Material
- Fachwert

Typen zusammengesetzter Architekturelemente

- Komplexes Werkzeug
- MonoWerkzeug
- Werkzeug: Oberbegriff für komplexe und MonoWerkzeuge

Gebots-Beziehungsregeln

Vererbung zwischen verschiedenen Architekturelement-Typen ist nicht erlaubt, alle Gebots-Beziehungsregeln betreffen Benutzbeziehungen.

- FKs müssen ein Material kennen.
- MonoTools müssen ein Material kennen.
- Automaten müssen ein Material kennen.
- Services müssen ein Material kennen.
- Materialien müssen einen Fachwert kennen.

Verbots-Beziehungsregeln

- Materialien dürfen kein Werkzeug kennen.

- Materialien dürfen keinen Automaten kennen.
- Materialien dürfen keinen Service kennen.
- FKs dürfen keine IAK kennen.
- FKs dürfen keine GUI kennen.
- FKs dürfen kein Tool kennen.
- IAKs dürfen keine anderen Typen kennen außer GUI und FK.
- Tools dürfen keine anderen Typen kennen außer GUI, IAK und FK.
- GUIs dürfen keine anderen Typen kennen.
- MonoTools dürfen kein Tool kennen.
- Automaten dürfen kein Werkzeug kennen.
- Services dürfen kein Werkzeug kennen.
- Services dürfen keinen Automaten kennen.
- Fachwerte dürfen keine anderen Typen kennen.

Enthält-Beziehungsregeln

- Komplexe Werkzeuge enthalten eine FK.
- Komplexe Werkzeuge enthalten eine IAK.
- Komplexe Werkzeuge enthalten ein Tool.
- Werkzeuge enthalten eine GUI.
- MonoWerkzeuge enthalten ein MonoTool.

Interne Gebotsregeln

- In komplexen Werkzeugen gilt: IAKs müssen eine GUI kennen.
- In komplexen Werkzeugen gilt: IAKs müssen eine FK kennen.
- In komplexen Werkzeugen gilt: Tools müssen eine FK kennen.
- In komplexen Werkzeugen gilt: Tools müssen eine IAK kennen.
- In komplexen Werkzeugen gilt: Tools müssen eine GUI kennen.
- In MonoWerkzeugen gilt: MonoTools müssen eine GUI kennen.

Schnittstellenregeln

- Materialien enthalten mindestens eine Methode, die weder Getter noch Setter ist.
- FKs enthalten keine Methode, die ein Material zurück gibt.
- Fachwerte besitzen keinen öffentlichen Konstruktor.
- Fachwerte enthalten keine Setter.

Anhang C: Messergebnisse

C.1 Architekturelemente

Elementtyp	Anzahl Elemente
Materialien	19
Funktionskomponenten	0
Interaktionskomponenten	0
Tools	0
GUIs	17
MonoTools	31
Services	33
Automaten	7
Fachwerte	34
Anzahl komplexe Werkzeuge	0
Anzahl MonoWerkzeuge	22
Anzahl Architekturelemente gesamt	163

Tabelle C-1: Anzahl der Architekturelemente im System "Behörde"

Elementtyp	Anzahl Elemente
Materialien	19
Funktionskomponenten	0
Interaktionskomponenten	0
Tools	0
GUIs	31
MonoTools	34
Services	20
Automaten	5
Fachwerte	9
Anzahl komplexe Werkzeuge	0
Anzahl MonoWerkzeuge	32
Anzahl Architekturelemente gesamt	150

Tabelle C-2: Anzahl der Architekturelemente im System "Gesundheit"

Elementtyp	Anzahl
Materialien	8
Funktionskomponenten	6
Interaktionskomponenten	6
Tools	6
GUIs	10
MonoTools	4
Services	4
Automaten	1
Fachwerte	5
Anzahl komplexe Werkzeuge	6
Anzahl MonoWerkzeuge	4
Anzahl Architekturelemente gesamt	60

Tabelle C-3: Anzahl der Architekturelemente im System "Schule"

Elementtyp	Anzahl Elemente
Materialien	15
Funktionskomponenten	5
Interaktionskomponenten	5
Tools	5
GUIs	16
MonoTools	24
Services	16
Automaten	1
Fachwerte	14
Anzahl komplexe Werkzeuge	5
Anzahl MonoWerkzeuge	19
Anzahl Architekturelemente gesamt	125

Tabelle C-4: Anzahl der Architekturelemente im System "Konstruktionsbeispiel"

Elementtyp	Anzahl Elemente
Materialien	41
Funktionskomponenten	3
Interaktionskomponenten	3
Tools	3
GUIs	16
MonoTools	17
Services	56
Automaten	21
Fachwerte	30
Anzahl komplexe Werkzeuge	3
Anzahl MonoWerkzeuge	17
Anzahl Architekturelemente gesamt	210

Tabelle C-5: Anzahl der Architekturelemente im System " Fahrzeugvermietung"

Elementtyp	Anzahl Elemente
Subsysteme	6
Teilung Test- und Produktivcode	2
Teilung neu/migriert	2
Fachwerte	15
Items	11
Services	12
Anzahl Architekturelemente gesamt	48

Tabelle C-6: Anzahl der Architekturelemente im System " Kommunikation"

C.2 Verstöße

Die folgenden Tabellen listen die ermittelten Verstöße. Es wird jeweils die Anzahl der betroffenen Quelltextstellen aufgeführt. Sofern bekannt wird die Anzahl der fehlerhaften Architekturelemente aufgeführt, diese Anzahl wurde nicht in allen Fällen erhoben.

Hinweise zur Notation

Wurden keine Daten erhoben, so ist das entsprechende Feld leer.

Nicht zutreffende Felder sind mit einem waagerechten Stich gekennzeichnet.

Es sind nur die Regeln aufgeführt, gegen die Verstöße vorlagen.

Die von Verstößen betroffenen Typen sind exakt benannt. Beispielsweise wurde die im WAM-Stil vorliegende Verbotregel „GUIs dürfen keine anderen Typen kennen“ in der ersten Tabelle in zwei Zeilen unterteilt: „GUIs dürfen kein Material kennen“ und „GUIs dürfen keinen Service kennen“.

Regel	Anzahl betroffener Quelltextstellen	Anzahl fehlerhafter Architekturelemente	Anzahl fehlerhafter Beziehungen auf Architekturebene
Verbotsregel: Automaten dürfen keine MonoTools kennen	1	1	1
Verbotsregel: GUIs dürfen kein Material kennen	8	2	2
Verbotsregel: GUIs dürfen keinen Service kennen	40	5	5
Gebotsregel: Services müssen ein Material kennen	12	12	-
Gebotsregel: MonoTools müssen ein Material kennen	1	1	-
Schnittstellenregel: Fachwerte enthalten keine Setter	2	1	-

Tabelle C-7: Anzahl der Verstöße im System "Behörde"

Regel	Anzahl betroffener Quelltext- stellen	Anzahl fehlerhafter Architektur- elemente	Anzahl fehlerhafter Beziehungen auf Archi- tekturebene
Verbotsregel: Materialien dürfen keinen Automaten kennen	2	1	1
Gebotsregel: MonoTools müssen ein Material kennen	7	7	-
Gebotsregel: Services müssen ein Material kennen	4	3	-
Gebotsregel: Automaten müssen ein Material kennen	3	3	-
Schnittstellenregel: Fachwerte besitzen keinen öffentlichen Konstruk- tor	1	1	-

Tabelle C-8: Anzahl der Verstöße im System "Gesundheit"

Regel	Anzahl betroffener Quelltext- stellen	Anzahl fehlerhafter Architektur- elemente	Anzahl fehlerhafter Beziehungen auf Archi- tekturebene
Verbotsregel: GUIs dürfen keinen Automaten kennen	66		5
Verbotsregel: GUIs dürfen keinen Service kennen	5	1	1
Verbotsregel: IAKs dürfen keinen Automaten kennen	14	1	1
Verbotsregel: IAKs dürfen keine Material kennen	413		13
Verbotsregel: IAKs dürfen kein Tool kennen	3		2
Verbotsregel: Materialien dürfen keinen Service kennen	2	1	1
Verbotsregel: Tools dürfen keinen Service kennen	9	1	1
Schnittstellenregel: Fachwerte enthalten keine Setter	5		-
Schnittstellenregel: FKs enthalten keine Methode, die ein Material zurück gibt	5		-

Tabelle C-9: Anzahl der Verstöße im System "Schule"

Regel	Anzahl betroffener Quelltext- stellen	Anzahl fehlerhafter Architektur- elemente	Anzahl fehlerhafter Beziehungen auf Archi- tekturebene
Verbotsregel: IAKs dürfen keine Material kennen	24		2
Schnittstellenregel: FKs enthalten keine Methode, die ein Material zurück gibt	1	1	-
Schnittstellenregel: Materialen enthalten mindestens eine Methode, die weder Getter noch Setter ist.	4	4	-

Tabelle C-10: Anzahl der Verstöße im System "Konstruktionsbeispiel"

Regel	Anzahl betroffener Quelltextstellen	Anzahl fehlerhafter Architekturelemente	Anzahl fehlerhafter Beziehungen auf Architekturebene
Verbotsregel: Fachwerte dürfen keine Material kennen	13		13
Verbotsregel: Materialien dürfen keinen Service kennen	29	1	1
Verbotsregel: Tools dürfen keinen Service kennen	3		2
Gebotsregel: Automaten müssen ein Material kennen	3		-
Gebotsregel: MonoTools müssen ein Material kennen	6		-
Gebotsregel: Services müssen ein Material kennen	16		-
Interne Gebotsregel: In komplexen Werkzeugen gilt: IAKs müssen ihre GUI kennen	3		-
Schnittstellenregel: Fachwerte besitzen keinen öffentlichen Konstruktor	1	1	-
Schnittstellenregel: FKs enthalten keine Methode, die ein Material zurück gibt	1	1	-
Schnittstellenregel: Materialien enthalten mindestens eine Methode, die weder Getter noch Setter ist.	2	2	-

Tabelle C-11: Anzahl der Verstöße im System "Fahrzeugvermietung"

Regel	Anzahl betroffener Quelltext- stellen	Anzahl fehlerhafter Architek- turelemente	Anzahl fehlerhafter Beziehungen auf Archi- tekturebene
Verbotsregel: ProduktivCode darf keinen TestCode kennen	2	1	1
Verbotsregel: Die Darstellung darf das fachliche Modell nicht kennen	57	9	15
Verbotsregel: Die Darstellung darf das DB-Mapping nicht kennen	1	1	1
Verbotsregel: Das DB-Mapping darf keinen Service kennen	2	2	2
Gebotsregel: Services gehen mit Items um.	4	4	4
Schnittstellenregel: Fachwerte enthalten keine ändernden Methoden	10	5	5

Tabelle C-12: Anzahl der Verstöße im System "Kommunikation"