Quelltextannotationen für stilbasierte Ist-Architekturen

Petra Becker-Pechau

Arbeitsbereich Softwaretechnik, Universität Hamburg
und C1 WPS GmbH
Vogt-Kölln-Straße 30, 22527 Hamburg
becker@informatik.uni-hamburg.de

Abstract: Langlebige Softwaresysteme müssen vielfach an sich ändernde Bedingungen angepasst werden. Dabei sollte jederzeit Klarheit über den aktuellen Zustand der Softwarearchitektur bestehen. Dieser Artikel präsentiert einen Ansatz, Quelltexte so mit Architekturinformationen anzureichern, dass die aktuelle Ist-Architektur alleine aus dem annotierten Quelltext extrahiert werden kann. Da Architekturstile die kontrollierte Evolution von Softwaresystemen unterstützen, betrachtet dieser Artikel stilbasierte Architekturen.

1 Motivation

Die Ist-Architektur von Softwaresystemen muss bekannt sein, um die Systeme kontrolliert ändern zu können. Nur, wenn die Ist-Architektur bekannt ist, kann sichergestellt werden, dass Änderungen sich an die geplante Architektur halten. Und nur, wenn die Ist-Architektur bekannt ist, können Entwicklungsteams ihre Systeme auf der abstrakteren Ebene der Architektur verstehen, kommunizieren und planen, ohne die Details des Quelltextes betrachten zu müssen. Doch die Ist-Architektur ist nicht eindeutig im Quelltext zu finden [RH09]. Das Problem lässt sich anhand eines Beispiels verdeutlichen. Unser Beispielsystem enthält die drei Klassen `PatientenAufnahmeToolGui`, `PatientenAufnahmeTool` und `Patient`. Es handelt sich um ein reales Beispiel aus einem kommerziellen Softwaresystem. Abbildung 1 zeigt die Klassen und ihre Referenzen. Das System hat als Architekturvorgabe, dass der `Patientenaufnehmer` den `Patienten` benutzen darf, der `Patient` jedoch den `Patientenaufnehmer` nicht kennen darf. Abbildung 2 zeigt die gewünschte Architektur.

Das Problem ist: dem Quelltext ist nicht eindeutig anzusehen, welche Klassen zu welchen Architekturelementen gehören. Wie wirkt sich also die vorgegebene Architektur auf den Quelltext aus? Hierzu fehlt die Information, dass die Klassen `PatientenAufnahmeTool` und `PatientenAufnahmeToolGui` zusammen den Patientenaufnehmer bilden, während die Klasse `Patient` dem Architekturelement Patient zugeordnet ist. Abbildung 3 verdeutlicht diesen Zusammenhang.

Erst mit dieser Information wird klar, was die Architekturvorgabe auf Klassenebene bedeutet: Die Klasse `Patient` darf weder das `PatientenAufnahmeTool` noch die

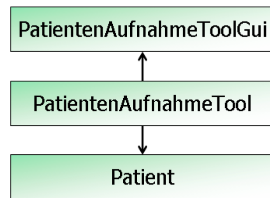

Abbildung 1: Klassendiagramm

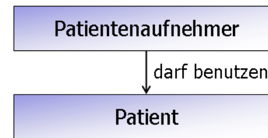

Abbildung 2: Soll-Architektur

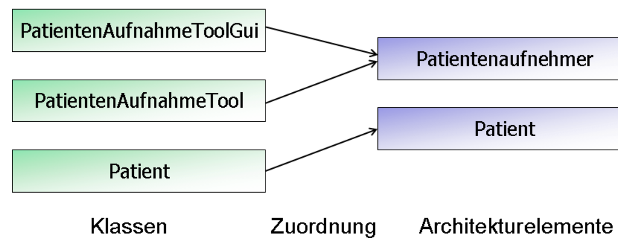

Abbildung 3: Zuordnung von Klassen und Architekturelementen

`PatientenAufnahmeToolGui` referenzieren. Beziehungen in der umgekehrten Richtung sind jedoch erlaubt.

Das Beispiel macht deutlich, dass zusätzlich zum Quelltext weitere Informationen benötigt werden, um die Ist-Architektur von Softwaresystemen zu ermitteln. Langlebige Softwaresysteme werden im Laufe ihres Einsatzes mehrfach überarbeitet und weiterentwickelt. Es genügt also nicht, die zusätzlich zum Quelltext benötigten Architekturinformationen zu Entwicklungsbeginn zu dokumentieren. Stattdessen muss diese Dokumentation permanent auf dem aktuellen Stand gehalten werden. Bei jeder Änderung des Quelltextes ist zu prüfen, ob sie noch korrekt ist.

Dieser Artikel adressiert das geschilderte Problem auf Basis von Quelltextannotationen, die zusätzliche Architekturinformationen in den Quelltext bringen. Wir halten diese Informationen direkt im Quelltext fest, um die Gefahr zu verringern, dass Quelltext und Architekturdokumentation auseinanderlaufen. Dabei konzentrieren wir uns auf statische Architekturen, auf die Modulsicht [HNS00]. Der Artikel liefert folgende Beiträge:

- Wir zeigen auf, welche Anteile der Ist-Architektur aus dem Quelltext entnommen werden können und welche zusätzlichen Informationen benötigt werden (siehe Abschnitt 3).
- Da Architekturstile für langlebige Softwaresysteme besonders relevant sind (siehe Abschnitt 2), diskutieren wir darüber hinaus, welche zusätzlichen Informationen für stilbasierte Ist-Architekturen notwendig sind (siehe Abschnitt 3).
- Wir zeigen, an welchen Stellen im Quelltext welche Architekturinformationen ein-

gefügt werden müssen. (Abschnitt 4).

- Abschnitt 5 präsentiert eine beispielhafte Umsetzung mit Java-Annotations. Mit Hilfe eines Prototyps, der die stilbasierte Ist-Architektur von Softwaresystemen ermittelt, konnten wir die Machbarkeit unseres Ansatzes zeigen.

2 Hintergrund

Um den fachlichen Hintergrund zu verdeutlichen, diskutiert dieser Abschnitt die Bedeutung von Architekturstilen für langlebige Softwaresysteme und gibt einen kurzen Überblick über die grundlegenden Begriffe.

2.1 Bedeutung von Architekturstilen für langlebige Softwaresysteme

Architekturstile machen Aussagen darüber, wie Software-Architekturen prinzipiell strukturiert werden sollen [RH09]. Sie legen fest, welche Strukturen erlaubt und welche Strukturen ausgeschlossen sind [GAO94, Kru95]. Insbesondere für langlebige, evolvierende Softwaresysteme bietet der Einsatz von Architekturstilen umfangreiche Vorteile: Architekturstile unterstützen konsistente und verständliche Architekturen, da verschiedene Systemversionen nach den selben Prinzipien strukturiert werden, selbst wenn die Architektur evolviert. Im Projektverlauf tragen sie maßgeblich dazu bei, die Komplexität von Softwaresystemen zu bewältigen [Lil08]. Für einige Unternehmen haben hauseigene Stile sogar eine strategische Bedeutung – beispielsweise für Capgemini sd&m der Quasar-Stil [Sie04] oder für die C1 WPS der Stil des Werkzeug-und-Material-Ansatzes (WAM-Ansatz) [Zül05]. Stile können auf bestimmte Systemarten ausgerichtet sein [KG06]. Ihre Architekturelement-Arten werden als ein Vokabular für die Architekturmodellierung verstanden [GS94, KG06]. Softwareteams können damit leichter über ihr Softwaresystem kommunizieren, da das Vokabular zur Systemart passt.

Unser Ansatz erlaubt es, die stilbasierte Ist-Architektur in der Entwicklungsumgebung sichtbar zu machen, als zweite Sicht auf den Quelltext. So brauchen die Softwareteams nicht mehr zwischen den Konstrukten der Programmiersprache und ihren eigenen Architekturkonstrukten zu übersetzen.

2.2 Grundlegende Begriffe

Betrachten wir das einleitende Beispiel, so lässt sich nachvollziehen, dass im Rahmen dieses Artikels unter einer *Software-Architektur* die Architekturelemente und deren Beziehungen verstanden werden. Das Verständnis ist angelehnt an bestehende Definitionen des Begriffs, wobei je nach Definition der Begriff auch weiter gefasst wird, als es für diesen Artikel nötig ist (vgl. [RH09, BCK03]).

Unter einer *Ist-Architektur* verstehen wir eine Architektur, die – soweit möglich – anhand eines Quelltextes ermittelt wurde. Wie genau unser Ansatz den Zusammenhang zwischen Ist-Architektur und Quelltext herstellt, erläutern die folgenden Abschnitte.

Stilbasierte Architekturen werden anhand eines Architekturstils entworfen. Das obige Beispiel stammt aus einem System, das auf dem WAM-Stil basiert. In diesem Stil gibt es unter anderem die Architekturelement-Arten "Werkzeug" und "Material". Werkzeuge dienen zur Benutzerinteraktion, mit ihnen werden Materialien bearbeitet. Materialien repräsentieren fachliche Gegenstände und werden Teil des Arbeitsergebnisses. Im WAM-Stil gilt, dass jedes Werkzeug ein Material referenzieren soll, Materialien jedoch nicht auf Werkzeuge zugreifen dürfen. Im Beispiel ist das Architekturelement namens Patientenaufnehmer ein Werkzeug, das Architekturelement namens Patient ist ein Material (siehe Abbildung 4).

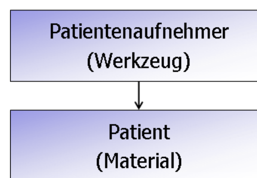

Abbildung 4: Stilbasierte Software-Architektur (Beispiel)

Architekturstile bewegen sich auf einer Metaebene zur Architektur, sie legen fest, aus welchen Architekturelementarten eine Architektur bestehen soll und definieren Regeln für Architekturelemente abhängig von ihrer Elementart. Um die stilbasierte Ist-Architektur zu ermitteln, müssen die Elementarten des zugehörigen Stils bekannt sein. Abbildung 5 zeigt ein Metamodell für stilbasierte Architekturen im Zusammenhang mit den Architekturelementarten. Formal verstehen wir unter einer stilbasierten Architektur eine Architektur, deren Architekturelemente einer Elementart zugeordnet sind.

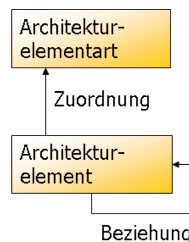

Abbildung 5: Stilbasierte Software-Architektur (Metamodell)

Dem Quelltext ist ohne Zusatzinformation nicht eindeutig anzusehen, wie die Klassen und andere Quelltextelemente den Architekturelementen zugeordnet sind. Es fehlt auch die Information, zu welcher Art die Architekturelemente gehören. Es existieren verschiedene Ansätze, diese zusätzlichen Informationen zu ermitteln:

- Der schwierigste Fall liegt vor bei Altsystemen, für die keine Architektur-Dokumen-

tation existiert und das Entwicklungsteam nicht mehr verfügbar ist. Dann müssen Reengineering-Techniken basierend auf Heuristiken verwendet werden. Dies kann manuell oder (teilweise) werkzeuggestützt erfolgen. Abhängig von der gewählten Heuristik liefern diese Techniken unterschiedliche Ist-Architekturen für ein und dasselbe Softwaresystem, da in diesen Fällen die ursprüngliche Intention des Entwicklungsteams nicht mehr bekannt ist. Man ist auf "Vermutungen" angewiesen [CS09].

- Ist die Architektur hingegen dokumentiert, so kann versucht werden, anhand der Dokumentation und des Quelltextes die Ist-Architektur zu rekonstruieren. In diesem Fall ist die Intention der Entwickler zumindest teilweise verfügbar. In der Praxis gibt es einige Stolpersteine. Es ist schwierig, die Dokumentation jederzeit konsistent zum Quelltext zu halten. Hinzu kommt, dass die Dokumentation meist informeller Natur ist, sie muss interpretiert werden.
- Sind noch Mitglieder des Entwicklungsteams verfügbar, so können diese nach ihrer ursprünglichen Intention gefragt werden. Doch gerade bei langlebigen Softwaresystemen ist es unrealistisch zu erwarten, dass die Entwickler beispielsweise für jede Klasse noch wissen, zu welchem Architekturelement sie gehören sollte. Auch ist sich das Entwicklungsteam nicht immer einig.

In der Praxis treten diese Ansätze meist gemischt auf. Wird beispielsweise die Ist-Architektur für eine Architektur-Konformanzprüfung benötigt, so müssen die zusätzlich zum Quelltext benötigten Informationen manuell im Prüfungswerkzeug beschrieben werden. Dies ist beispielsweise der Fall bei der Sotograph-Familie [BKL04], SonarJ[1], Lattix [SJSJ05] und Bauhaus[2]. Sofern noch Mitglieder des Entwicklungsteams verfügbar sind, so wird die Prüfung sicherlich von einem Mitglied durchgeführt oder unterstützt. Vorhandene Dokumentation wird genutzt, die Quelltextkommentare werden gelesen und es werden manuelle Heuristiken anhand von Bezeichnern und Quelltextstruktur verwendet. Wird das weiterentwickelte System mit demselben Werkzeug später erneut geprüft, so kann die in das Werkzeug eingegebene Beschreibung wiederverwendet werden. Wurde die Architektur des Systems jedoch geändert oder erweitert, so muss gegebenenfalls auch die Beschreibung manuell geändert werden.

Mit unserem Ansatz lassen sich stilbasierte Ist-Architekturen, wie in Abbildung 4, aus annotiertem Quelltext berechnen. Soll zu einem mit unserem Ansatz annotierten System die Ist-Architektur berechnet werden, so werden keine Heuristen benötigt, keine externe Dokumentation. Es ist nicht nötig, Mitglieder des Entwicklungsteams zu befragen. Stattdessen wird bereits bei der Entwicklung direkt im Quelltext vermerkt, wie die Quelltextelemente aus Architektursicht einzuordnen sind. Details stellen die Abschnitte 4 und 5 vor. Dieser Ansatz hat den Vorteil, dass die Intention der Entwickler direkt bei der Programmierung festgehalten wird. Natürlich ist auch hier nicht komplett sicherzustellen, dass immer korrekt annotiert wird, aber die Gefahr, dass die Architektur-Dokumentation und der Quelltext auseinanderlaufen, ist geringer.

Informationen über den Entwurf in den Quelltext zu bringen, ist nicht vollständig neu, jedoch für Architekturinformationen noch nicht sehr verbreitet. Der Ansatz gewinnt jedoch

[1] www.hello2morrow.com
[2] www.bauhaus-stuttgart.de

an Bedeutung (siehe Abschnitt 6). Neu an unserem Ansatz ist, dass wir es ermöglichen, *stilbasierte* Ist-Architekturen aus annotiertem Quelltext zu extrahieren. Stilbasierten Ist-Architekturen helfen das System zu verstehen und weiterzuentwickeln, darüber hinaus dienen sie als Basis für Konformanzprüfungen [BP09].

3 Stilbasierte Ist-Architekturen

3.1 Formale Definition

Dieser Abschnitt enthält die formale Definition für stilbasierte Ist-Architekturen. Das Metamodell für stilbasierte Architekturen zeigt bereits die Abbildung 5. Der Begriff der stilbasierten Ist-Architektur wurde im Zusammenhang mit der stilbasierten Architekturprüfung [BP09] eingeführt. Die für diesen Artikel relevanten Aspekte werden hier kurz wiederholt. Formal umfasst eine *stilbasierte Ist-Architektur* folgende Mengen und Relationen:

- die Menge der Architekturelemente E,
- eine Relation $B_I \subseteq E \times E$, die die bestehende Abhängigkeitsbeziehung zwischen diesen Architekturelementen beschreibt (kurz "Ist-Beziehungen"),
- die Relation $Z_A \subseteq E \times A$, mit der die Architekturelemente den Architekturelementarten zugeordnet werden. Diese Relation ist rechtseindeutig.

Darüber hinaus wird die Menge A der Architekturelementarten benötigt, um Z_A definieren zu können. Die Architekturelementarten ergeben sich aus dem gewählten Architekturstil. Im oben genannten WAM-Stil beispielsweise stehen unter anderem die Architekturelementarten Werkzeug und Material zur Verfügung.

Stilbasierte Ist-Architekturen werden soweit möglich direkt aus dem Quelltext berechnet. Dafür wird die *Quelltextstruktur* benötigt. Diese umfasst Folgendes:

- die Menge der Quelltextelemente Q,
- eine Relation $B_Q \subseteq Q \times Q$, die die bestehende Abhängigkeitsbeziehung zwischen diesen Quelltextelementen beschreibt.

Die stilbasierte Ist-Architektur und die Quelltextstruktur *hängen zusammen*, indem die Quelltextelemente den Architekturelementen in der folgenden Relation zugeordnet werden:

- die Relation $Z_Q \subseteq Q \times E$ legt für die Quelltextelemente fest, zu welchen Architekturelementen sie zugeordnet werden. Diese Relation ist rechtseindeutig.

Abbildung 6 zeigt das gesamte Metamodell für die Quelltextstruktur und die stilbasierte Ist-Architektur. Alle Informationen dieses Metamodells werden benötigt, um die stilbasierte Ist-Architektur eines Softwaresystems zu berechnen.

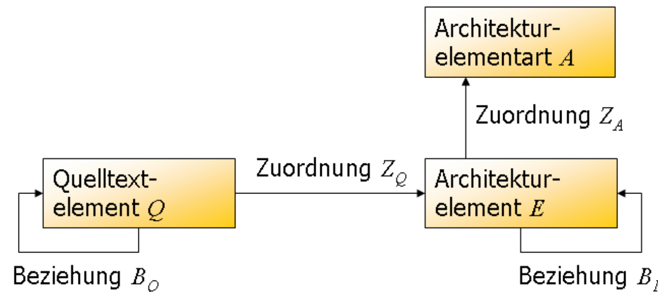

Abbildung 6: Quelltextstruktur und stilbasierte Ist-Architektur (Metamodell)

3.2 Ermittlung der stilbasierten Ist-Architektur

Die Quelltextstruktur (Q und B_Q) wird durch statische Quelltextanalyse berechnet, wie es unter anderem auch Konformanzprüfungs-Werkzeuge machen. Ein Quelltextelement kann dabei beispielsweise eine Klasse sein. Eine Abhängigkeit zu einer anderen Klasse ergibt sich beispielsweise durch Variablentypen oder eine Vererbungsbeziehung. Um den Ansatz einfach zu halten, unterscheiden wir bewusst nicht zwischen Abhängigkeitsarten. Dies ist auch in den meisten Architekturanalyse-Werkzeugen der Fall, da in der Regel sehr viel interessanter ist, von was ein Architekturelement abhängt, als welcher Art die einzelne Abhängigkeit ist. Unser Ansatz wäre jedoch leicht erweiterbar, sofern andere Fallstudien zeigen sollten, dass Abhängigkeitsarten benötigt werden.

Die Beziehungen der Ist-Architektur B_I lassen sich aus dem Quelltext berechnen. Dabei gilt: $B_I = \{(e, f) \mid \exists p, q : (p, q) \in B_Q \wedge (p, e) \in Z_Q \wedge (q, f) \in Z_Q\}$. Das heißt, wenn zwei Quelltextelemente p und q in Beziehung stehen, gibt es eine Beziehung in der Ist-Architektur zwischen den Architekturelementen e und f, sofern p und e sowie q und f einander zugeordnet sind.

Die Zuordnungen Z_Q und Z_A sowie die Menge der Architekturelemente E müssen zusätzlich beschrieben werden, sie sind dem reinen Quelltext nicht zu entnehmen.

4 Ein Konzept zur Quelltextannotation

Dieser Abschnitt behandelt die Frage, wie Z_Q, Z_A und E durch Quelltextannotation beschrieben werden können. Betrachten wir unser Beispiel in Abbildung 3. Es gilt für das Architekturelement namens Patientenaufnehmer:

- $(PatientenAufnahmeToolGui, Patientenaufnehmer) \in Z_Q$
- $(PatientenAufnahmeTool, Patientenaufnehmer) \in Z_Q$
- $(Patientenaufnehmer, Werkzeug) \in Z_A$

Damit die Annotationen mit dem Quelltext einfach konsistent gehalten werden können, sollten sie in dem jeweils betroffenen Quelltextelement festgehalten werden. Das heißt in unserem Beispiel: in den beiden Klassen `PatientenAufnahmeToolGui` und `PatientenAufnahmeTool` wird die Information benötigt “diese Klasse gehört zu dem Werkzeug namens Patientenaufnehmer”. Quelltextelemente werden annotiert mit dem zugehörigen Architekturelement und dessen Elementart. Formal dargestellt heißt das:

- annotiert werden alle Quelltextelemente q, für die gilt: $\exists(q, e) \in Z_Q$.
- Die Annotation umfasst erstens das Architekturelement e, für das gilt: $(q, e) \in Z_Q$,
- und zweitens die Elementart a, für die gilt: $(e, a) \in Z_A$.

In einigen Fällen kann die Annotation verkürzt werden. Bei der Klasse `Patient` beispielsweise liegt so ein Sonderfall vor: hier besteht zwischen dem Architekturelement und dem Quelltextelement eine 1:1-Beziehung. Dieser Fall kann auftreten, wenn der gewählte Stil eine detaillierte Anleitung für die Architektur gibt und so viele feine Architekturelemente entstehen. Beispiele für solche Stile sind der WAM-Stil oder der Quasar-Stil. In diesem Fall können die Klasse und ihr Architekturelement gleich benannt werden. Dadurch fällt die benötigte Annotation kürzer aus. Es genügt, die Elementart zu nennen.

5 Beispielhafte Umsetzung mit Java-Annotations

Um den Ansatz anhand bestehender Softwaresysteme erproben zu können, wurde eine beispielhafte Syntax für die Quelltextannotationen definiert, die auf der im vorherigen Abschnitt definierten Semantik beruht. Wir haben uns entschieden, Java-Annotations zu verwenden. Dies ist eine in die Sprache Java integrierte Möglichkeit zur Quelltext-Annotation. Java-Annotations werden ähnlich wie Typen definiert und können dann an Quelltextabschnitte geschrieben werden. In unserem Beispiel werden die Klassen `PatientenAufnahmeToolGui` und `PatientenAufnahmeTool` annotiert mit: Werkzeug(“Patientenaufnehmer”). Die Definition der Annotation für Werkzeuge ist in Abbildung 7 dargestellt.

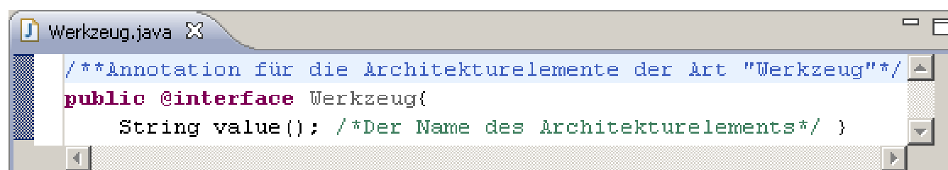

Abbildung 7: Definition der Java-Annotation für die Architekturelement-Art “Werkzeug”

Wir haben einen Prototyp namens ArchitectureExtractor realisiert, der anhand eines annotierten Quelltexts die stilbasierte Ist-Architektur ermittelt. Der Prototyp betrachtet alle

Java-Typen (Klassen, Interfaces etc.) als Quelltextelemente und ermittelt anhand des abstrakten Syntax-Baums die Referenzen zwischen diesen Elementen. Dafür nutzt er das entsprechende Eclipse-API.

Um die Machbarkeit des Ansatzes zu zeigen, haben wir den ArchitectureExtractor bisher für sechs Softwaresysteme verwendet, in der Größenordnung von 25.000 bis gut 100.000 LOC. Die Systeme folgten entweder dem WAM-Stil oder hatten einen speziell für das Projekt entworfenen, individuellen Stil. Die bereits existierenden Systeme wurden von einer projektexternen Person um Quelltextannotationen ergänzt. Die Person stützte sich dabei auf ihre guten Kenntnisse des Stils und auf Quelltextkommentare. Eine vollständige, externe Dokumentation der Architekturen existierte nicht, dazu waren die Architekturen zu feingranular, solch eine Dokumentation wäre zu schnell veraltet. Die Ergebnisse wurden mit WAM-Experten und Projektmitgliedern besprochen. Die befragten Personen bestätigten, dass die extrahierte Ist-Architektur aus ihrer Sicht hilfreich ist, da so ohne zusätzlichen Aufwand für externe Dokumentation ein Blick auf die Architektur ermöglicht wird, der sonst durch die Details des Quelltextes überdeckt wird. Die Quelltextannotationen verursachen nach unserer Einschätzung kaum zusätzlichen Aufwand, da vorher genau dieselben Informationen natürlichsprachlich dokumentiert werden mußten.

6 Verwandte Arbeiten

Architekturinformation im Quelltext unterzubringen, wird in der letzten Zeit vermehrt gefordert. Koschke und Popescu beispielsweise vergleichen verschiedene Ansätze zur Architektur-Konformanzprüfung [KP07]. Dabei kommen sie zu der Überzeugung, dass es einfacher wäre, den Quelltext und die Architekturinformationen konsistent zu halten, wenn man Architekturelemente im Quelltext kennzeichnen könnte. Clements und Shaw schreiben, dass manche Architekturinformationen im Quelltext nicht zu finden sind und nennen Schichtenarchitekturen als Beispiel [CS09]. Sie beklagen, dass bestehende Architekturanalyse-Werkzeuge nicht adäquat seien, da sie sich auf Vermutungen über Architekturkonstrukte stützen müssen. Sie sind der Meinung, dass Architektur-Markierungen im Quelltext helfen würden. Diese Forschungsrichtung betrachten sie als eine der drei vielversprechendsten für Softwarearchitekturen. Keiner der im Folgenden präsentierten Ansätze betrachtet *stilbasierte* Architekturen. Unseres Wissens sind wir die ersten, die im Quelltext Architekturinformationen unterbringen, bei denen der Zusammenhang zum gewählten Architekturstil explizit hergestellt wird.

Abi-Antoun und Aldrich extrahieren Objektgrafen durch statische Analyse aus speziell annotierten Quelltexten [AAA09]. Für die Annotationen nutzen sie Java-Generics. Direkt bei der Programmierung kann das Entwicklungsteam Architekturinformationen im Quelltext unterbringen. Die Autoren nennen als Vorteil, dass so der extrahierten Architektur die *Intention* des Entwicklungsteams zugrunde liegt und nicht eine beliebige Heuristik des genutzten Analyse-Werkzeugs. Genau das trifft auch für unseren Ansatz zu und ist der Hauptgrund, dass wir uns mit Quelltextannotationen beschäftigen. Im Gegensatz zu Abi-Antoun und Aldrich extrahieren wir jedoch keinen Objektgrafen, sondern betrachten die statische Sicht.

Einer der mittlerweile älteren und mehrfach zitierten Ansätze, Quelltexte mit Architekturinformationen anzureichern, behandelt ebenfalls die Laufzeitsicht [LR03]. Die Sprache Java wird um Subsysteme und Tokens erweitert. Eine Klasse kann einem Subsystem angehören, wenn ein Objekt erzeugt wird, so kann ihm ein Token gegeben werden. Lam und Rinard erzeugen mit Hilfe dieser Informationen Objektgrafen und ermitteln den Zusammenhang zwischen Subsystemen und Objekten. Sie heben hervor, dass ihre Modelle nicht manuell, sondern automatisch nur anhand des annotierten Quelltextes generiert werden und somit die Programmstruktur garantiert akkurat reflektieren.

Aldrich beschäftigt sich, wie wir, mit der statischen Sicht von Softwarearchitekturen [Ald08]. Er ergänzt die Sprache Java um sogenannte Komponenten-Klassen. Die Spracherweiterung ist eine Weiterentwicklung von ArchJava [ACN02]. Komponenten-Klassen können Ports definieren, über die sie sich verbinden lassen. Sie können hierarchisch geschachtelt werden. Ein mit solchen Komponentenklassen strukturierter Quelltext kann auf bestimmte Architektureigenschaften geprüft werden. Im Gegensatz zu unserem Ansatz mit Java-Annotations ist es nicht möglich, normale Java-Klassen als Architekturelemente zu kennzeichnen. Zwar nimmt Aldrich als Beispiel eine Pipes-and-Filters-Architektur, Stilinformationen können mit seiner Spracherweiterung jedoch nicht annotiert werden.

7 Zusammenfassung, Diskussion und Ausblick

Dieser Artikel präsentiert einen Ansatz, der es erlaubt, stilbasierte Ist-Architekturen aus annotierten Quelltexten zu extrahieren. Einige Informationen über Ist-Architekturen lassen sich aus dem reinen Quelltext extrahieren. Doch nicht alle Architekturkonstrukte sind hier wiederzufinden, einige verschwinden, sobald ein Architekturentwurf in Quelltext umgesetzt wird. Damit Werkzeuge nicht darauf angewiesen sind, die Architekturkonstrukte im Quelltext zu "erraten", benötigen sie weitere Informationen, die wir durch Quelltextannotation hinzufügen. Neu an unserem Ansatz ist:

- Wir betrachten Architekturen, die auf Architekturstilen beruhen. Solche stilbasierten Architekturen spielen insbesondere für langlebige Softwaresysteme eine große Rolle.
- Wir nutzen die bestehenden Annotationsmöglichkeiten der Programmiersprache Java, um zusätzliche Informationen über die statische Architektur in den Quelltext zu bringen. Dadurch können bestehende Systeme von unserem Ansatz profitieren, ohne auf eine andere Sprache umsteigen zu müssen.

Der Ansatz, den Quelltext mit Architekturinformationen anzureichern, bietet verschiedene Vorteile:

- Entwicklungsteams können auf diese Weise ihre Intention bezüglich der Architektur direkt bei der Entwicklung festhalten.
- Es ist ein pragmatischer Ansatz. Dadurch, dass die Architekturinformationen direkt

im Quelltext festgehalten werden, wird es einfacher, die Konsistenz dieser Informationen zum Quelltext zu wahren.

- Entwicklungsumgebungen können erweitert werden, so dass dem Entwicklungsteam während der Programmierung eine Architektursicht auf den Quelltext gegeben wird.

Wir zeigen eine beispielhafte Umsetzung mit Java-Annotations. Wir haben einen Prototyp erstellt, den ArchitectureExtractor. Mit ihm wurden mehrere annotierte Quelltexte untersucht und erfolgreich die stilbasierten Ist-Architekturen extrahiert. Unser Prototyp beinhaltet bisher nur eine sehr einfache Darstellung der extrahierten Architektur, wir planen, einen entsprechenden ArchitectureViewer zu ergänzen und zu evaluieren, um eine gute Architektursicht direkt zum Programmierzeitpunkt zu bieten.

Unser Ansatz ist für statisch getypte Sprachen geeignet, da die Abhängigkeitsbeziehungen der Ist-Architektur durch statische Analyse aus dem Quelltext gewonnen werden. Wie andere Ansätze zur Quelltextannotation ist auch unser Ansatz darauf angewiesen, dass korrekt annotiert wird. Dass Entwickler tatsächlich ihre Intention in den Quelltext schreiben, lässt sich selbstverständlich nicht sicherstellen. Die im Abschnitt 6 vorgestellten verwandten Arbeiten diskutieren dieses Thema nicht. Wir vermuten jedoch, dass durch Werkzeugunterstützung fehlerhafte Annotationen reduziert werden könnten. Wird beispielsweise eine annotierte Klasse umbenannt oder durch Kopieren erzeugt, so könnte ein "Dirty-Flag" den Entwickler darauf hinweisen, dass die zugehörige Annotation möglicherweise überarbeitet werden muss. Hierfür planen wir, den ArchitectureExtractor in laufenden Projekten einzusetzen, um mögliche Fehlerquellen zu identifizieren und unterstützende Ansätze wie das Dirty-Flag zu evaluieren.

Die praktischen Untersuchungen haben gezeigt, dass nur einige Klassen als architekturrelevant eingestuft werden und annotiert werden müssen. Da die Annotationen die Intention der Entwickler beschreiben, liegt es bei ihnen, zu entscheiden, ob sie ein Quelltextelement annotieren, genauso, wie sie vorher entscheiden mußten, welche Kommentare sie schreiben. Fehlerhafte Annotationen können - genauso wie fehlerhafte Kommentare - nur durch zustätzliche Maßnahmen, wie beispielsweise Code-Reviews, aufgedeckt werden. Unser Ansatz bietet den Vorteil, dass die Architektur nicht manuell anhand der Quelltextkommentare verstanden werden muss, sondern automatisiert extrahiert werden kann.

Wir haben den hier präsentierten Ansatz bereits erweitert, um auch hierarchische Architekturen behandeln zu können. Wir planen, diesen erweiterten Ansatz weiter zu evaluieren und zu veröffentlichen.

Literatur

[AAA09] Marwan Abi-Antoun und Jonathan Aldrich. Static extraction of sound hierarchical runtime object graphs. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, Seiten 51–64, New York, NY, USA, 2009. ACM.

[ACN02] Jonathan Aldrich, Craig Chambers und David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, Seiten 187–197, New York, NY, USA, 2002. ACM.

[Ald08] Jonathan Aldrich. Using Types to Enforce Architectural Structure. In *WICSA '08*, Seiten 211–220, Washington, DC, USA, 2008. IEEE Computer Society.

[BCK03] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, Reading, Mass., 2003.

[BKL04] Walter Bischofberger, Jan Kühl und Silvio Löffler. Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In F. Oquendo, Hrsg., *EWSA 2004*, Seiten 1–9. Springer-Verlag Berlin Heidelberg, 2004.

[BP09] Petra Becker-Pechau. Stilbasierte Architekturprüfung. Angenommener Artikel auf der Informatik-Konferenz. 2009.

[CS09] Paul Clements und Mary Shaw. "The Golden Age of Software Architecture" Revisited. *IEEE Software*, 26(4):70–72, 2009.

[GAO94] David Garlan, Robert Allen und John Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, Seiten 175–188, USA, 1994. ACM.

[GS94] David Garlan und Mary Shaw. An Introduction to Software Architecture. Bericht CS-94-166, Carnegie Mellon University, 1994.

[HNS00] Christine Hofmeister, Robert Nord und Dilip Soni. *Applied Software Architecture*. Object technology series. Addison-Wesley, 2000.

[KG06] Jung Soo Kim und David Garlan. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, Seiten 70–80. ACM Press, Portland, Maine, 2006.

[KP07] Jens Knodel und Daniel Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, Seite 12. IEEE Computer Society, 2007.

[Kru95] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software 12*, 6:42–50, 1995.

[Lil08] Carola Lilienthal. *Komplexität von Softwarearchitekturen - Stile und Strategien*. Dissertation, Universität Hamburg, 07 2008.

[LR03] Patrick Lam und Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Seiten 275–302, 2003.

[RH09] Ralf Reussner und Wilhelm Hasselbring, Hrsg. *Handbuch der Software-Architektur*, Jgg. 2. dpunkt.verlag, Heidelberg, 2009.

[Sie04] Johannes Siedersleben. *Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, Heidelberg, 2004.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha und Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, Seiten 167–176. ACM Press, San Diego, CA, USA, 2005.

[Zül05] Heinz Züllighoven. *Object-Oriented Construction Handbook*. Morgan Kaufmann Publishers, San Francisco, 2005.