Stilbasierte Architekturprüfung

Petra Becker-Pechau

Arbeitsbereich Softwaretechnik
Department Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg
becker@informatik.uni-hamburg.de

Abstract: Softwaresysteme weichen im Laufe ihrer Evolution von Architekturvorgaben ab. Dieses Phänomen wird als Architekturerosion bezeichnet. Prüfungen auf Architekturtreue sollen der Erosion entgegenwirken. Die bestehenden Ansätze prüfen Systeme bezogen auf Soll-Architekturen, nicht jedoch auf Architekturstile. Die in diesem Artikel präsentierte Architekturprüfung füllt diese Lücke. Mit ihr lassen sich Stile beschreiben und Systeme daraufhin überprüfen, ob sie einen gegebenen Stil einhalten. Verstöße werden aufgezeigt und können beseitigt werden. Die stilbasierte Architekturprüfung wirkt so einer Architekturerosion entgegen.

1 Einleitung

Architekturstile machen Aussagen darüber, wie Software-Architekturen prinzipiell strukturiert werden sollen [RH09]. Sie legen fest, welche Strukturen erlaubt und welche Strukturen ausgeschlossen sind. Wenn ein Softwaresystem nach einem Architekturstil konstruiert wird, so liefert der Stil die gewünschte Anleitung [GAO94, Kru95]. Stile spielen daher eine bedeutende Rolle in der Softwareentwicklung.

Empirische Studien haben gezeigt, dass Softwaresysteme im Laufe ihrer Evolution verstärkt von den Architekturvorgaben abweichen [PW92]. Solche Abweichungen entstehen ungewollt und unbemerkt, beispielsweise, wenn Entwicklungsteams unter hohem zeitlichen Druck arbeiten. Prüfungen auf Architekturtreue [MNS01] ermitteln, an welchen Stellen ein Softwaresystem gegen die Vorgaben verstößt. Das Entwicklungsteam kann sein System entsprechend restrukturieren.

Die bestehenden Prüfungen auf Architekturtreue vergleichen Softwaresysteme mit einer systemspezifischen Soll-Architektur, nicht jedoch mit *Architekturstilen*. Dieser Artikel präsentiert einen neuen Ansatz: die *stilbasierte Architekturprüfung*. Mit ihr lassen sich Architekturstile beschreiben und Softwaresysteme auf Stiltreue hin überprüfen. Die stilbasierte Architekturprüfung ergänzt die bestehenden Ansätze. Sie ist sprach- und vorgehensunabhängig.

Der folgende Abschnitt diskutiert Architekturstile in der Softwareentwicklung, definiert

den Begriff und zeigt ihre Bedeutung. Abschnitt 3 grenzt den präsentierten Ansatz gegen verwandte Arbeiten ab. Dann folgt der Stand der Kunst: Wir erläutern, wie Softwaresysteme auf ihr Treue zur Soll-Architektur geprüft werden. Abschnitt 5 präsentiert das Konzept der stilbasierten Architekturprüfung – den Kern dieses Artikels. Um zu zeigen, dass sich unser Ansatz praktisch umsetzen lässt, haben wir einen Prototyp erstellt, den wir in Abschnitt 6 vorstellen. Wir berichten über erste Erfahrungen mit der Prüfung realer Softwaresysteme. Der letzte Abschnitt fasst den Artikel zusammen und gibt einen Ausblick auf weitere Forschungstätigkeiten.

2 Architekturstile

2.1 Definition

Der Begriff des Architekturstils wird unterschiedlich verwendet [GAO94, Kru95, Lil09, RH09, HNS00, BCK03]. Den Definitionen ist gemein, dass Architekturstile Architekturelement-Typen beinhalten und Regeln für die Interaktion von Architekturelementen. Wir schließen uns diesem Verständnis an und differenzieren die Regeln in Beziehungs- und Schnittstellenregeln. Architekturstile umfassen somit Folgendes:

- eine Menge von Architekturelement-Arten[1]
- Regeln für die Interaktion von Architekturelementen bestimmter Arten, und zwar
  Regeln über Beziehungen (Beziehungsregeln) und
  Regeln über Schnittstellen (Schnittstellenregeln)

Die in diesem Artikel verwendete Definition konzentriert sich auf die prüfbaren Aspekte von Architekturstilen.

Architekturstile liegen auf einer Metaebene zur Architektur. Basiert eine Softwarearchitektur auf einem Architekturstil, so kann die Architektur als Exemplar des Stils betrachtet werden [BPB07].

2.2 Beispiel: Der Architekturstil des WAM-Ansatzes

Mit dem WAM-Ansatz (Werkzeug, Automat, Material) werden interaktive objektorientierte Softwaresysteme entwickelt [Zül05]. Der Ansatz stammt wesentlich aus dem Arbeitsbereich Softwaretechnik der Universität Hamburg. Er wird seit über 15 Jahren in der Praxis und an Hochschulen eingesetzt und weiterentwickelt. Der Ansatz definiert einen eigenen Architekturstil, mit mehreren Element-Arten und einer Vielzahl von Regeln. Vergleichbar umfangreiche Stile werden von Fowler [Fow03], Evans [Eva04] und Siedersleben

[1] Wir sprechen von *Art* anstelle von *Typ*, um Verwechslungen mit dem Typbegriff in Programmiersprachen zu vermeiden.

[Sie04] beschrieben. Die Regeln des WAM-Stils wurden zunächst informell beschrieben [Zül05] und später formalisiert [BPKL06].

Die Abbildungen 1 und 2 zeigen den Zusammenhang zwischen Architektur und Stil anhand des WAM-Stils. Abbildung 1 zeigt einen Ausschnitt aus einer Softwarearchitektur, angelehnt an ein reales WAM-System zur Verwaltung von Zivildienstleistenden (ZDL). Die Abbildung 2 zeigt ausgewählte Element-Arten des Stils sowie erlaubte Beziehungen. Werkzeuge dienen im WAM-Stil zur Benutzerinteraktion. Komplexe Werkzeuge enthalten eine grafische Präsentation (GUI), den Zustand der Interaktion (IAK), die Funktionalität (FK) sowie ein zusammenfassendes Tool-Element. In MonoWerkzeugen werden die Aufgaben der IAK, der FK und des Tools vom MonoTool übernommen. Services stellen fachliche Dienstleistungen zur Verfügung, Automaten übernehmen längere, mehrschrittige Aufgaben, Materialien repräsentieren fachliche Gegenstände und werden Teil des Arbeitsergebnisses. Fachwerte sind unveränderliche, fachliche Werttypen.

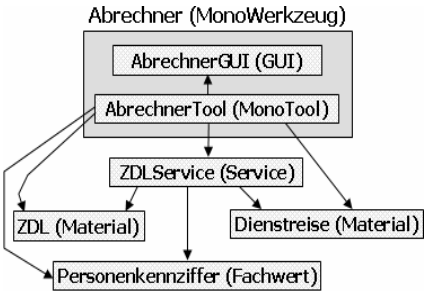

Abbildung 1: Ist-Architektur (Beispiel)

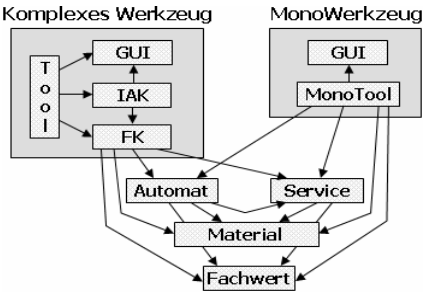

Abbildung 2: Architekturstil (Beispiel)

2.3 Bedeutung von Architekturstilen

Softwaresysteme auf der Basis von Architekturstilen zu konstruieren, bietet umfangreiche Vorteile [GS94]: Architekturstile enthalten Erfahrungswissen. Sie lassen sich als Standard-Strukturen für ähnliche Probleme wiederverwenden [BCK03]. Sie unterstützen konsistente und verständliche Architekturen, da verschiedene Systemteile nach den selben Prinzipien strukturiert werden. Neue Projektmitglieder können sich leichter einarbeiten. 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 WAM-Ansatzes.

Stile spielen für die initiale Struktur eine bedeutende Rolle; sie helfen auch, Systeme weiterzuentwickeln und zu restrukturieren. Im Projektverlauf tragen sie maßgeblich dazu bei, die Komplexität von Softwaresystemen zu bewältigen [Lil08].

Stile können auf bestimmte Systemarten ausgerichtet sein [KG06], beispielsweise auf interaktive Anwendungssysteme [Zül05], auf Unternehmenssoftware [Sie04] oder Weban-

wendungen [MD07]. 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.

3 Verwandte Arbeiten

Der hier präsentierte Ansatz der stilbasierten Architekturprüfung ordnet sich ein in den Bereich der *static software compliance checks* [KP07]. Im Deutschen wird u.a. von *Konformanzprüfung* oder von der *Prüfung auf Architekturtreue* gesprochen. Verschiedene Konzepte existieren für diese Prüfung [KP07], alle zeigen Abweichungen des Quelltexts von der erwünschten Soll-Architektur auf. Die Soll-Architektur wird meist auf der Ebene von Architekturelementen und Beziehungen beschrieben. Die Herausforderung besteht darin, diese abstrakte Beschreibung mit dem Quelltext eines Softwaresystems zu vergleichen. Dafür wird die Architektur des Systems (Ist-Architektur) auf Basis des Quelltexts ermittelt und mit der Soll-Architektur verglichen. Da dies manuell aufwändig und fehlerträchtig wäre, stehen verschiedene Werkzeuge für die Prüfung auf Architekturtreue zur Verfügung, beispielsweise die Sotograph-Familie [BKL04], SonarJ[2], Lattix[3] [SJSJ05] und Bauhaus[4].

Das Konzept der Software-Reflexionsmodelle [MNS01] bildet eine verbreitete theoretische Grundlage für Architekturprüfungen. Die Soll-Architektur der Software-Reflexionsmodelle legt fest, aus welchen Architekturelementen ein Softwaresystem bestehen soll und welche Beziehungen zwischen diesen Elementen gewünscht sind. Das Konzept wurde mehrfach erweitert, beispielsweise um Beziehungsarten zu unterscheiden und um Architekturelemente hierarchisch anzuordnen [KS03]. Die stilbasierte Architekturprüfung setzt auf die Software-Reflexionsmodelle auf. Abschnitt 4 stellt die einzelnen Teilaufgaben des Konzepts detailliert vor. Anstelle einer Soll-Architektur wird bei der stilbasierten Architekturprüfung ein einzuhaltender Stil vorgegeben.

Knodel et al. unterscheiden zwei weitere Ansätze zur Prüfung auf Architekturtreue: Component Access Rules und Relation Conformance Rules [KP07]. Als Component Access Rules bezeichnen sie Regeln, mit denen sich Teile von Schnittstellen verbergen lassen. Andere Architekturelemente dürfen nicht auf diese verborgenen Teile zugreifen. Anders als die Component Access Rules legen die Schnittstellenregeln der stilbasierten Architekturprüfung fest, wie Schnittstellen für bestimmte Architekturelemente aufgebaut sein sollen. Mit Relation Conformance Rules können ähnliche Architekturverstöße wie bei den Software-Reflexionsmodellen ermittelt werden. Architekturvorgaben werden hier in Form von Regeln formuliert. Dabei können die Architekturelemente anhand ihres Namens mit Hilfe von relationalen Ausdrücken ermittelt werden. Im Gegensatz zur stilbasierten Architekturprüfung können keine Architekturstile vorgegeben werden.

Ein verwandtes Forschungsgebiet beschäftigt sich mit der Frage, wie sich bestehende Entwurfsmuster in Softwaresystemen erkennen lassen [PSRN05]. Einige Autoren überprü-

[2] www.hello2morrow.com
[3] www.lattix.com
[4] www.bauhaus-stuttgart.de

fen, ob die gefundenen Muster korrekt implementiert wurden [SSC96]. Im Gegensatz zur stilbasierten Architekturprüfung wird nicht vorgegeben, welche Muster erwartet werden.

Die modellgetriebene Softwareentwicklung zielt darauf, Softwaresysteme auf abstrakter Ebene zu beschreiben, die Modelle gegebenenfalls zu prüfen und aus ihnen den Quelltext zu generieren. Beispielsweise können Entwicklungsteams den Einfluss von Architekturentwürfen auf das Laufzeitverhalten bereits validieren, bevor sie das entsprechende System generieren [BKR09]. Unser Ansatz hingegeben prüft existierenden Quelltext von Softwaresystemen.

4 Stand der Kunst: Software-Reflexionsmodelle

Die stilbasierte Architekturprüfung setzt auf die Software-Reflexionsmodelle auf. Um unseren Ansatz zu erläutern, genügt es, sich auf das Grundkonzept der Software-Reflexionsmodelle [MNS01] zu konzentrieren. Software-Reflexionsmodelle können für die Prüfung auf Architekturtreue verwendet werden und um unbekannte Softwaresysteme zu verstehen. Dieser Artikel fokussiert auf den Aspekt der Prüfung mit den verschiedenen Teilaufgaben: die Soll-Architektur definieren, die Ist-Architektur aus dem Quelltext extrahieren und Abweichungen zwischen Soll und Ist in Form eines Software-Reflexionsmodells berechnen. Die Teilaufgaben können in beliebigen Arbeitsschritten, auch iterativ, bearbeitet werden.

Die Semantik der Software-Reflexionsmodelle wurde in der formalen Spezifikationssprache Z beschrieben [MNS01]. Dieser Artikel überträgt die Beschreibung der Semantik in eine mengentheoretische Notation, da die Kenntnis von Z nicht grundsätzlich vorausgesetzt werden kann und die mengentheoretische Notation für die weitere Arbeit ausreicht. Zusätzlich zur mengentheoretischen Darstellung werden die Zusammenhänge in UML visualisiert und anhand von Beispielen veranschaulicht.

4.1 Teilaufgabe: Soll-Architektur beschreiben

Die Soll-Architektur (von Murphy et al. als *high-level model* bezeichnet [MNS01]) legt Architekturelemente und Beziehungen fest. Formal beschrieben besteht sie aus Folgendem:

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

Die Beziehungen B_S werden auch 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 3 zeigt ein Beispiel für eine Soll-Architektur. Die Kästchen stehen für Architekturelemente, die Pfeile für Beziehungen. Es handelt sich um einen vereinfachten Ausschnitt aus dem Rahmenwerk Spring[5].

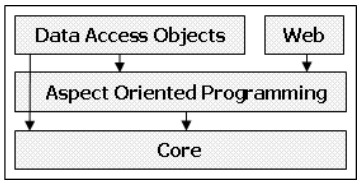

Abbildung 3: Die Soll-Architektur legt Architekturelemente und Beziehungen fest (Beispiel)

4.2 Teilaufgabe: Ist-Architektur ermitteln

Die Ist-Architektur wird in zwei Teilschritten ermittelt: Zuerst müssen den Quelltextelementen Architekturelemente zugeordnet werden, dann lassen sich die Beziehungen der Ist-Architektur feststellen. Was als Quelltextelement angesehen wird, ist frei wählbar. Bei objektorientierten Systemen können dies beispielsweise Klassen oder sprachspezifische Ausdrucksmittel wie Java-Interfaces, Java-Packages oder C++-Namensräume sein.

Für die Zuordnung der Quelltextelemente zu den Architekturelementen werden die folgenden Mengen und Relationen benötigt:

- Die Menge A von Architekturelementen, die auch für die Definition der Soll-Architektur benötigt wird
- Eine Menge Q von Quelltextelementen
- Die Zuordnung $Z_1 \subseteq Q \times A$ zwischen Quelltext- und Architekturelementen

Abbildung 4 zeigt ein Beispiel, hier werden Packagebäume aus Spring vier verschiedenen Architekturelementen zugeordnet. Abbildung 5 visualisiert die Zuordnung in UML.

Der Ansatz der Software-Reflexionsmodelle macht keine Einschränkungen bezüglich der Eigenschaften von Z_1. Dem Beispiel lässt sich entnehmen, dass einem Architekturelement mehrere Quelltextelemente zugeordnet werden können. Der umgekehrte Fall ist prinzipiell erlaubt, wird jedoch von manchen Werkzeugen ausgeschlossen.

Die Beziehungen der Ist-Architektur werden aus den Quelltextbeziehungen eines zu prüfenden Systems berechnet. Was als eine Beziehung auf Quelltextebene angesehen wird, ist – genau wie die Definition eines Quelltextelements – frei wählbar. Beispielsweise ist es üblich, Typreferenzen und Operationsaufrufe als Beziehung zu interpretieren. Aus den Beziehungen der Quelltextelemente und der Zuordnung Z_1 werden die Beziehungen B_I

[5] www.springsource.org

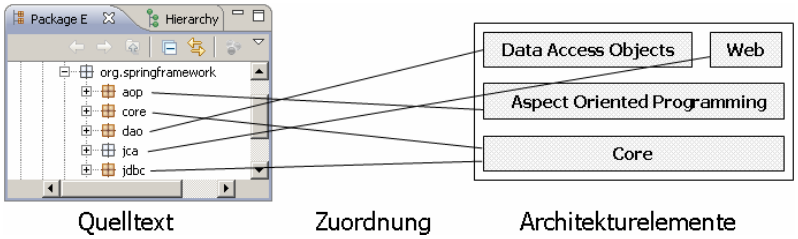

Abbildung 4: Zuordnung Z_1 (Beispiel)

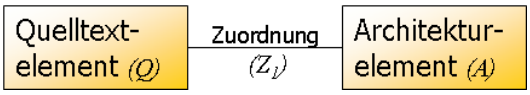

Abbildung 5: Zuordnung Z_1 (UML-Darstellung)

der Ist-Architektur ermittelt. Hierfür werden zusätzlich folgende Mengen und Relationen definiert:

- Eine Menge $B_Q \subseteq Q \times Q$ von Quelltextbeziehungen
- Die Beziehungen der Ist-Architektur sind definiert als Relation über die Architekturelemente: $B_I \subseteq A \times A$.
  Dabei gilt: $B_I = \{(a,b) \mid \exists p,q : (p,q) \in B_Q \wedge (p,a) \in Z_1 \wedge (q,b) \in Z_1\}$.
  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 6 zeigt, wie eine Ist-Architektur aus der Quelltextstruktur ermittelt wird. Die Pfeile stellen Beziehungen dar, die Kästchen repräsentieren Quelltext- und Architekturelemente. Quelltextelemente sind innerhalb der zugeordneten Architekturelemente dargestellt. Abbildung 7 zeigt die Quelltextstruktur, die Ist-Architektur und die Zuordnung Z_1 in UML.

Zur Veranschaulichung betrachten wir einen vereinfachten Ausschnitt aus dem Rahmenwerk Spring. Die Klasse CommonsPoolTargetSource referenziert die Klasse Constants. Diese Referenz ergibt sich aus dem Quelltext (Abbildung 8). Die Klasse CommonsPoolTargetSource gehört zum Architekturelement AspectOrientedProgramming, die Klasse Constants gehört zum Architekturelement Core (Abbildung 9). Aus der Beziehung zwischen den zwei Klassen ergibt sich eine Beziehung zwischen den zugeordneten Architekturelementen (Abbildung 10).

Formal stellt sich dieser Zusammenhang folgendermaßen dar: Es gilt: $B_I = \{(a,b) \mid \exists p,q : (p,q) \in B_Q \wedge (p,a) \in Z_1 \wedge (q,b) \in Z_1\}$ (siehe oben). Für $a = AspectOrientedProgramming$ und $b = Core$ ist diese Bedingung erfüllt (mit $p = CommonsPoolTargetSource$ und $q = Constants$).

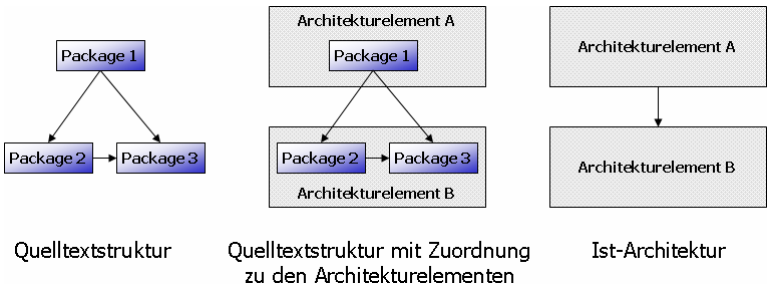

Abbildung 6: Überblick: Ist-Architektur ermitteln (Beispiel)

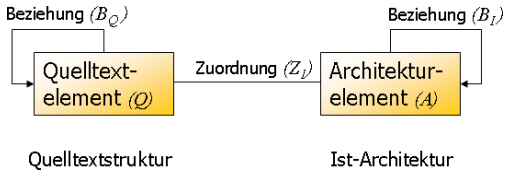

Abbildung 7: Ist-Architektur, Quelltextstruktur und Zuordnung Z_1 (UML-Darstellung)

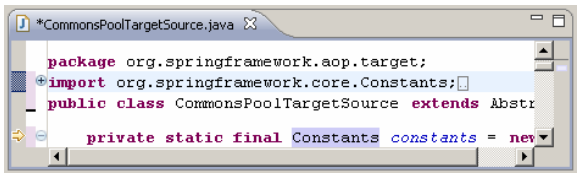

Abbildung 8: Quelltextbeziehung (Beispiel)

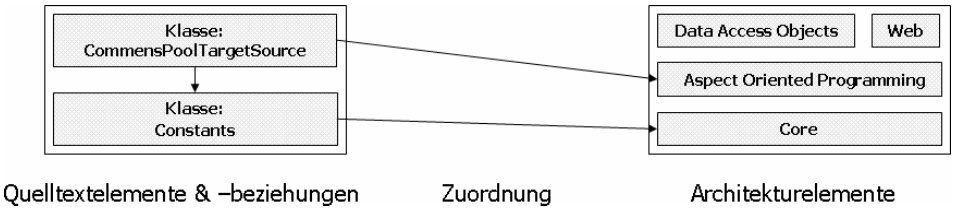

Abbildung 9: Beziehungen ermitteln: Ausgangssituation (Beispiel)

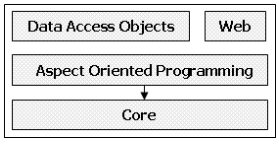

Abbildung 10: Resultierende Beziehung in der Ist-Architektur (Beispiel)

4.3 Teilaufgabe: Software-Reflexionsmodell berechnen

Ziel der verschiedenen Teilaufgaben ist es, ein Software-Reflexionsmodell zu berechnen. Dazu wird die Ist-Architektur mit der Soll-Architektur verglichen. Es wird geprüft, ob die tatsächlichen Beziehungen in der Ist-Architektur den gewünschten Beziehungen in der Soll-Architektur entsprechen und welche Abweichungen bestehen. Software-Reflexionsmodelle enthalten das Ergebnis dieses Vergleichs. Sie bestehen aus den Mengen A, Co, Di und Ab. Diese Mengen sind wie folgt definiert:

- A ist die Menge der Architekturelemente (siehe Abschnitt 4.1)
- Die Relation $Co = B_S \cap B_I$ beinhaltet die übereinstimmenden Beziehungen (convergences)
- Die Relation $Di = B_I \backslash B_S$ beinhaltet die abweichenden Beziehungen (divergences)
- Die Relation $Ab = B_S \setminus B_I$ beinhaltet die fehlenden Beziehungen (absences)

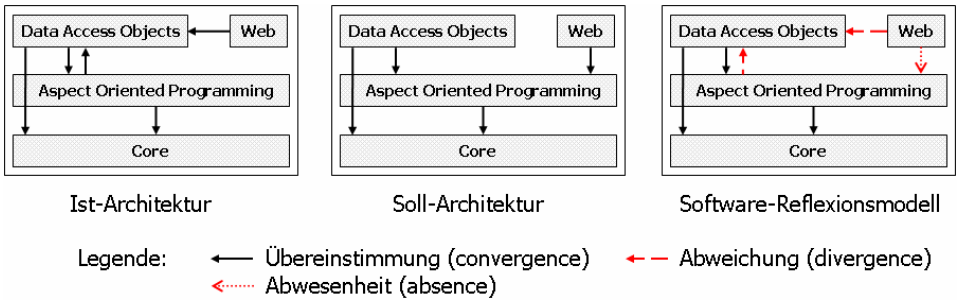

Abbildung 11: Software-Reflexionsmodell (Beispiel)

Aus dem Software-Reflexionsmodell lässt sich ablesen, welche Beziehungen in der Ist-Architektur genau so sind wie gewollt, wo nicht gewollte Beziehungen vorliegen und wo erwünschte Beziehungen fehlen. Dies ist beispielhaft an Abbildung 11 nachzuvollziehen.

5 Die stilbasierte Architekturprüfung

Verstöße gegen Architekturvorgaben sind leicht zu übersehen und selbst von erfahrenen Entwicklern kaum zu vermeiden [PW92]. Das trifft auch auf Verstöße gegen Architekturstile zu [BPKL06]. Die stilbasierte Architekturprüfung hilft eine derartige Architekturerosion zu verhindern, indem sie überprüft, ob der Quelltext eines Softwaresystems den gewählten Architekturstil einhält. Sie deckt Verstöße gegen den Stil auf, so dass diese korrigiert werden können.

Die stilbasierte Architekturprüfung setzt auf das Konzept der Software-Reflexionsmodelle auf. Sie ändert und erweitert dieses Konzept. Anstelle einer Soll-Architektur wird ein Architekturstil beschrieben und die Ist-Architektur auf die Einhaltung dieses Stils hin überprüft. Wie bei den Software-Reflexionsmodellen müssen für die stilbasierte Architekturprüfung verschiedene Teilaufgaben durchgeführt werden. Die folgenden Abschnitte erläutern diese Teilaufgaben.

5.1 Teilaufgabe: Architekturstil beschreiben

Wir haben in Abschnitt 2.1 den Begriff des Architekturstils definiert. Ein Architekturstil besteht aus Architekturelement-Arten, Beziehungs- und Schnittstellenregeln. Wie sich Schnittstellenregeln beschreiben und prüfen lassen soll aus Platzgründen in diesem Artikel nicht weiter thematisiert werden: wir konzentrieren uns auf die Beziehungsregeln. Um Architekturstile beschreiben zu können, unterscheiden wird drei verschiedene Arten von Beziehungsregeln: erlaubte Beziehungen, vorgeschriebene Beziehungen und Enthält-Beziehungen. Wir betrachten im Folgenden eine bestimmte Regel-Art beispielhaft: die erlaubten Beziehungen. Diese Regel-Art ist definiert als eine zweistellige Relation auf den Element-Arten (siehe Abbildung 12). Andere Beziehungsregel-Arten wurden in ähnlicher Weise formalisiert. Formal dargestellt umfasst ein Architekturstil im Rahmen der stilbasierten Architekturprüfung Folgendes:

- die Menge der Architekturelement-Arten E
- Beziehungsregeln, u.a. erlaubte Beziehungen: $B_R \subseteq E \times E$
- Schnittstellenregeln

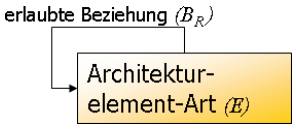

Abbildung 12: Erlaubte Beziehungen (UML-Darstellung)

5.2 Teilaufgabe: Stilbasierte Ist-Architektur ermitteln

Die Ist-Architektur wird in zwei Teilschritten ermittelt: Die Architekturelemente werden den Quelltextelementen sowie den Element-Arten zugeordnet, dann werden die Beziehungen zwischen den Architekturelementen berechnet.

Bei der stilbasierten Architekturprüfung müssen Quelltext- und Architekturelemente einander zugeordnet werden, genauso wie bei den Software-Reflexionsmodellen (siehe Abschnitt 4.2). Eine zweite Zuordnung wird benötigt, um den Zusammenhang zum gewählten

Architekturstil herzustellen, und zwar muss für jedes Architekturelement definiert werden, zu welcher Element-Art es gehört. Dies geschieht über die Relation Z_2 (siehe auch Abbildung 13). Folgende Mengen und Relationen werden für diesen Teilschritt benötigt:

- Die Menge Q von Quelltextelementen
- Die Menge A von Architekturelementen
- Die Menge E von Architekturelement-Arten
- Die Relation $Z_1 \subseteq Q \times A$, mit der die Quelltextelemente den Architekturelementen zugeordnet werden
- Die Relation $Z_2 \subseteq A \times E$, die für Architekturelemente festlegt, zu welcher Architekturelement-Art sie gehören.

Die Beziehungen zwischen den Architekturelementen der Ist-Architektur ergeben sich aus dem Quelltext, genau so wie bei den Software-Reflexionsmodellen (siehe Abschnitt 4.2). Eine Ist-Architektur, deren Architekturelemente über die Relation Z_2 den Element-Arten zugeordnet sind, bezeichnen wir als *stilbasierte Ist-Architektur*. Wir machen formal keine Einschränkungen bezüglich der Eigenschaften von Z_2, um flexibel zu bleiben, beispielsweise für Zwischenschritte der Prüfung. Normalerweise hat jedes Architekturelement genau eine Element-Art.

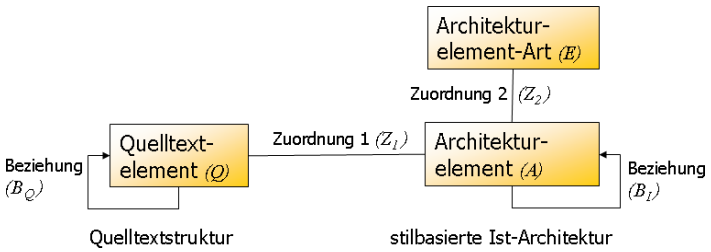

Abbildung 13: Zuordnungen und Beziehungen (UML-Darstellung)

5.3 Teilaufgabe: Ergebnis berechnen

In Anlehnung an Software-Reflexionsmodelle definieren wir ein *stilbasiertes Software-Reflexionsmodell*. Dieses stellt das Prüfungsergebnis dar. Hierfür wird die Hilfsrelation $B_E \subseteq A \times A$ benötigt; sie repräsentiert die erlaubten Beziehungen zwischen Architekturelementen, abhängig von deren Element-Arten. Es gilt: $B_E = \{(a,b) \mid \exists e,f : (e,f) \in B_R \wedge (a,e) \in Z_2 \wedge (b,f) \in Z_2\}$. Ein stilbasiertes Software-Reflexionsmodell umfasst folgende Mengen und Relationen:

- Architekturelemente A

- Die Relation $Co = B_I \cap B_E$ beinhaltet die übereinstimmenden Beziehungen (convergences)
- Die Relation $Di = B_I \backslash B_E$ beinhaltet die abweichenden Beziehungen (divergences)

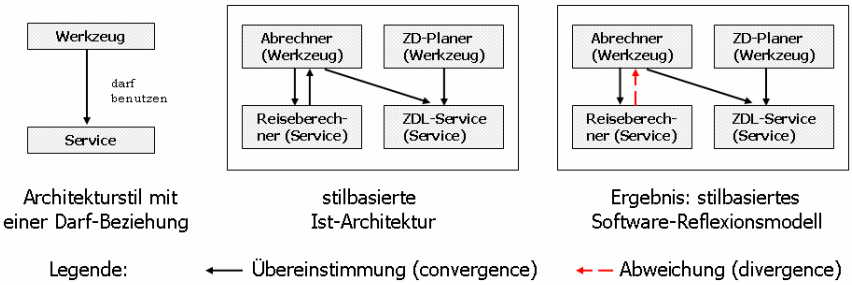

Abbildung 14: Stilbasiertes Software-Reflexionsmodell (Beispiel)

Abbildung 14 zeigt ein Beispiel für die stilbasierte Architekturprüfung. Der dort gezeigte Architekturstil definiert die Architekturelemente Werkzeug und Service und eine erlaubte Beziehung von Werkzeugen zu Services. Die Ist-Architektur enthält vier Architekturelemente: zwei Werkzeuge und zwei Services. Diese Architekturelemente stehen in vier Beziehungen. Drei dieser Beziehungen verlaufen von einem Werkzeug zu einem Service; die vierte Beziehung verläuft von dem Service "Reiseberechner" zu dem Werkzeug "Abrechner". Diese Beziehung ist laut Architekturstil nicht erlaubt und wird daher im resultierenden stilbasierten Software-Reflexionsmodell als Abweichung eingeordnet. Die anderen drei Beziehungen sind Übereinstimmungen.

Es wird deutlich, dass bei der stilbasierten Architekturprüfung nicht eine konkrete Architektur, sondern eine Klasse von Architekturen vorgeschrieben wird. Ein und derselbe Architekturstil kann von verschiedenen Architekturen eingehalten werden; Stile sind wiederverwendbar. In Abbildung 14 könnte beispielsweise eine zusätzliche Beziehung zwischen Zivildienst-Planer (ZD-Planer) und Reiseberechner existieren, dadurch würde sich die Menge der Architekturverletzungen (Abweichungen) nicht ändern.

6 Erfahrungen

Um die Machbarkeit der stilbasierten Architekturprüfung zu zeigen, haben wir mehrere Softwaresysteme geprüft. Mit Hilfe von Erweiterungen im Softwarewerkzeug Sotograph[6] ließen sich Verstöße gegen den gewählten Stil aufzeigen [BPKL06]. Es wurde deutlich, dass selbst erfahrene Entwicklerinnen und Entwickler versehentlich gegen den Stil verstoßen. Diese Verstöße bleiben teilweise unbemerkt, so dass die Architektur erodiert. Der Ansatz mit Hilfe des Sotographen hatte den Nachteil, dass die Erweiterungen in Form

[6] www.hello2morrow.com

von Datenbankabfragen realisiert wurden. Die Beschreibung der Architekturstile in einer SQL-ähnlichen Sprache und die Auflistung der Architekturverstöße in Tabellenform waren nicht intuitiv verständlich. SQL ist eine Sprache für die Domäne der relationalen Datenbanken, nicht für Software-Architekturen. Hinzu kommt, dass die Entwicklungsumgebung und der erweiterte Sotograph separate Werkzeuge waren, so dass die Systeme nur getrennt entwickelt und geprüft werden konnten.

Dann haben wir ein prototypisches Werkzeug, den ArchitectureChecker, implementiert [Sch07]. Er ist ein Plug-in für die Eclipse-Entwicklungsumgebung und erlaubt, während der Programmierung zu überprüfen, ob das bearbeitete Softwaresystem den gegebenen Architekturstil einhält. Mit Hilfe dieses Prototyps wurden bereits mehrere akademische und kommerziell eingesetzte Softwaresysteme untersucht. Am Beispiel des WAM-Stils zeigte sich, dass die Stilbeschreibung ohne Änderungen für verschiedene Softwaresysteme des gleichen Stils genutzt werden kann.

Softwaresysteme direkt während der Programmierung zu prüfen, rückt die Architektur in das Blickfeld der Programmiererinnen und Programmierer. So wird eine architekturzentrierte Programmierung unterstützt (im Englischen sprechen wir von *architecture-aware programming*). Die Programmiererinnen und Programmierer können in der Entwicklungsumgebung die Abstraktionen sehen, auf deren Basis sie kommunizieren, wie beispielsweise Architekturelemente der Art “Use-Case” oder “Material”. Der Architekturstil dient nicht mehr nur als Konzept und Sprache; er wird auch bei der technischen Umsetzung sichtbar und nutzbar.

Die geprüften Systeme wurden in der Programmiersprache Java entwickelt. Der ArchitectureChecker identifiziert die Beziehungen der Quelltextstruktur anhand des AST (abstract syntax tree). Java ist eine getypte Sprache; der AST gibt somit umfangreiche Informationen über verschiedene Beziehungen innerhalb des Quelltexts. Unser Ansatz sollte generell für getypte, objektorientierte Sprachen einsetzbar sein, nicht jedoch für ungetypte Sprachen, allenfalls mit umfangreichen Quelltext-Annotationen. Diese Grenze gilt auch für Software-Reflexionsmodelle: Sie sind davon abhängig, dass sich die Beziehungen aus dem Quelltext extrahieren lassen.

Im Anschluss an die Untersuchungen der Softwaresysteme wurden einige Entwickler interviewt. Sie bestätigten, dass es sich bei den Funden des Werkzeugs um Fehler in der Architektur handelt. Einige der Verstöße waren bereits bekannt, andere dagegen waren bisher unentdeckt geblieben.

7 Zusammenfassung und Ausblick

Architekturstile spielen eine wichtige Rolle für Architekturentwurf und Evolution. Mit der hier vorgestellten stilbasierten Architekturprüfung lässt sich feststellen, ob der Quelltext eines Softwaresystems den zugehörigen Stil einhält. Die stilbasierte Architekturprüfung wirkt so einer Architekturerosion entgegen.

Die stilbasierte Architekturprüfung lässt sich einordnen in den Bereich der statischen Prüfung auf Architekturtreue. Der Ansatz ist ebenso sprachunabhängig wie die Software-

Reflexionsmodelle, auf die er aufsetzt. Er macht keine Annahmen oder Einschränkungen über die benötigten Rollen und die Reihenfolge der verschiedenen Tätigkeiten in der Softwareentwicklung.

Die stilbasierte Architekturprüfung hat sich bei ersten Anwendungen bewährt. Es wurden erfolgreich Verstöße gegen Architekturstile aufgedeckt. Ein und dieselbe Stilbeschreibung ließ sich für verschiedene Systeme wiederverwenden.

Für die Prüfung wurde ein Prototyp realisiert, der ArchitectureChecker. Der Checker bietet zusätzlich eine rudimentäre Architektursicht. Wir planen, den Prototyp mit einer verbesserten Architektursicht zu ergänzen und diese in Projekten zu evaluieren.

Für den ArchitectureChecker müssen die Architekturstile und die Zuordnungen in einer maschinenlesbaren Form vorliegen. Zur Darstellung dieser Informationen wurde eine Syntax definiert, welche die in Abschnitt 5 vorgestellte Semantik unterstützt. Die verschiedenen Ansätze für solch eine Syntax sollen weiter untersucht werden.

Es ist geplant, den ArchitectureChecker in weiteren Projekten einzusetzen, die Ergebnisse systematisch auszuwerten und gemeinsam mit den bereits gewonnenen Untersuchungsergebnissen vorzustellen.

Literatur

[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.

[BKR09] Steffen Becker, Heiko Koziolek und Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[BPB07] Petra Becker-Pechau und Marcel Bennicke. Concepts of Modeling Architectural Module Views for Compliance Checks Based on Architectural Styles. In J. Smith, Hrsg., *Software Engineering and Application (SEA 2007)*, Massachusetts, 2007. Acta Press.

[BPKL06] Petra Becker-Pechau, Bettina Karstens und Carola Lilienthal. Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln. In Bettina Biel, Matthias Book und Volker Gruhn, Hrsg., *Software Engineering 2006*, Lecture Notes in Informatics, Seiten 27–38, Leipzig, 2006. Gesellschaft für Informatik.

[Eva04] Eric Evans. *Domain Driven Design: Tackling Complexity in the heart of Software*. Addison-Wesley, 2004.

[Fow03] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[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, NY, 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.

[KS03] Rainer Koschke und Daniel Simon. Hierarchical Reflexion Models. In *Proceedings of the 10th Working Conference on Reverse Engineering*, Seite 36. IEEE Computer Society, 2003.

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

[Lil09] Carola Lilienthal. Architectural Complexity of Large-Scale Software Systems. In A. Winter R. Ferenc, J. Knodel, Hrsg., *European Conference on Software Mainenance and Reengineering (CSMR 2009)*, Seiten 17–26. IEEE Computer Society, 2009.

[MD07] Ali Mesbah und Arie van Deursen. An Architectural Style for Ajax. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, Seite 9. IEEE Computer Society, 2007.

[MNS01] Gail C. Murphy, David Notkin und Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transaction on Software Engineering*, 27(4):364–380, 2001.

[PSRN05] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch und Sebastian Naumann. An Approach for Reverse Engineering of Design Patterns. *Software & Systems Modeling*, 4(1):55–70, February 2005.

[PW92] Dewayne E. Perry und Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM Sigsoft, Software Engineering Notes*, 17(4):40–52, 1992.

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

[Sch07] Arne Scharping. Automatisierte Prüfung von Architekturregeln zur Entwicklungszeit. Diplomarbeit, Universität Hamburg, 2007.

[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.

[SSC96] Mohlalefi Sefika, Aamod Sane und Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, Seiten 387–396, Washington, DC, USA, 1996. IEEE Computer Society.

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