Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung AW1

Thorben Pergande

Automatisierte Architekturanalyse unter Einsatz von UML 2.0 Modellen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einführung

In dieser Ausarbeitung werden mögliche Lösungsansätze für eine Softwarearchitekturanalyse anhand von UML 2.0 Modellen beschrieben.

Im Folgenden wird die Motivation und die Problemstellung für den Vergleich von Architekturen darstellt. Kapitel 2 befasst sich mit den Grundlagen der Softwarearchitektur und der Analyse dieser zum heutigen Zeitpunkt. Ein mögliches Szenario um die Problemstellung und Motivation zu untermauern wird in Kapitel 3 dargestellt. Kapitel 4 zeigt mögliche Vorgehensweisen für den Vergleich zwischen UML 2.0 Diagrammen (SOLL-Architektur) und dem Quelltext (IST-Architektur) auf. Kapitel 5 beschreibt mögliche Risiken bei der Realisierung des hier vorgestellten Ansatzes und Kapitel 6 zeigt die nächsten Schritte der Realisierung auf.

1.1 Motivation und Problemstellung

UML 2.0 ist ein Mittel, die SOLL-Architektur bzw. Teile davon graphisch darzustellen. Z.B. könne die Komponenten, Schnittstellen und die Benutzung dieser in Form eines Komponentendiagramms visualisiert werden. Der Vorteil von UML 2.0 ist, dass es sich um einen verbreiteten Ansatz für die Modellierung mit einer breiten Unterstützung an Werkzeugen handelt, sodass der Umgang mit den Werkzeugen und die Bedeutung der Modelle in vielen Entwicklerteams bekannt sind.

Die Motivation dieser Ausarbeitung besteht darin, dass eine Möglichkeit geschaffen werden soll, anhand der vorhandenen Modelle der Soll-Architektur eine Analyse der IST-Architektur automatisiert durchzuführen. Dabei stellen die Modelle SOLL-Architektur und der Quelltext des dazugehörigen Softwaresystems die Eingabe der Analyse dar. Als Ergebnis soll ebenfalls ein UML 2.0 Diagramm automatisiert erstellt werden, welches mögliche Differenzen zwischen SOLL- und IST-Architektur graphisch und ggfs. textuell darstellt. Anhand der aus der Analyse erstellten Modelle sollen eventuelle Differenzen in der Architektur mit dem Entwicklerteam erkannt, kommuniziert und diskutiert werden können.

Die Problemstellung liegt darin, wie der Quelltext eines Softwaresystems mit einem UML 2.0 konformen Diagramm verglichen und daraus ein Differenzmodell, welches dem Aufbau der Eingabemodelle entspricht, automatisiert erstellt werden kann. Dieses soll eine besserer Kommunikation durch die bekannte Darstellung mit den Entwicklern ermöglichen und den Aufwand der Analyse minimieren.

2 Grundlagen

In diesem Kapitel soll kurz verdeutlicht werden, was Softwarearchitektur ist und warum eine Analyse dieser notwendig ist.

2.1 Softwarearchitektur

Softwarearchitektur ist eine Abstraktion der konkreten Realisierung einer Softwareanwendung. Sie stellt eine Zwischenstufe zwischen den Anforderungen an das System und dem konkreten Quelltext dar. Dabei beschreibt die Architektur als grobes Systemkonzept den Schnitt der benötigten Komponenten, der dazugehörigen Schnittstellen und die Beziehungen zwischen den Komponenten. Auf die Architektur eines Systems gibt es verschiedene Sichten, die einen jeweiligen Kontext hervorheben. Die fachliche Sicht beschreibt die Anforderungen an das Softwaresystem und Qualitätsziele, wie Benutzbarkeit. Die statische Sicht definiert den Schnitt der Komponenten, Subsysteme, Schnittstellen, Verantwortlichkeiten und Beziehungen untereinander aber auch mögliche Schichten. Die Verteilungssicht ordnet die statische Sicht verschiedenen Computern, Prozessen und Netzwerken zu. Die Laufzeitsicht beschreibt die Interaktion, Synchronisation und den Datenaustausch innerhalb der Architektur. Jede dieser Sichten kann durch verschiedene UML 2.0 Diagramme (Kecher (2005)) dargestellt werden.

- **statische Sicht**: Klassen-, Komponenten- und Paketdiagramme
- **fachliche Sicht**: Sequenz- Aktivitäts- und Anwendungsfalldiagramme
- **Verteilungssicht**: Verteilungsdiagramme
- **Laufzeitsicht**: Sequenz- und Interaktionsdiagramme

Zu Beginn eines Softwareentwicklungsprozesses kann eine erste Darstellung der Architektur während der Analyse- und Designphase erstellt werden, die die bis dahin gestellten Anforderungen an das System beinhaltet. Diese aus den Anforderungen entstandene Architektur ist die SOLL-Architektur des Systems und stellt die Basis für die weitere Planung und Verteilung der Aufgaben innerhalb des Entwicklerteams dar. Ebenfalls können an dieser Stelle erste definierte Qualitätsziele, wie z.B. Zyklenfreiheit, Wiederverwendbarkeit oder die Einhaltung einer bestimmten Schichtenhierarchie, in den Entwurf aufgenommen und modelliert werden. In dieser Ausarbeitung wird nun fortlaufend zwischen zwei verschiedenen Stadien der Architektur unterschieden. Zum einen gibt es die SOLL-Architektur, die während der

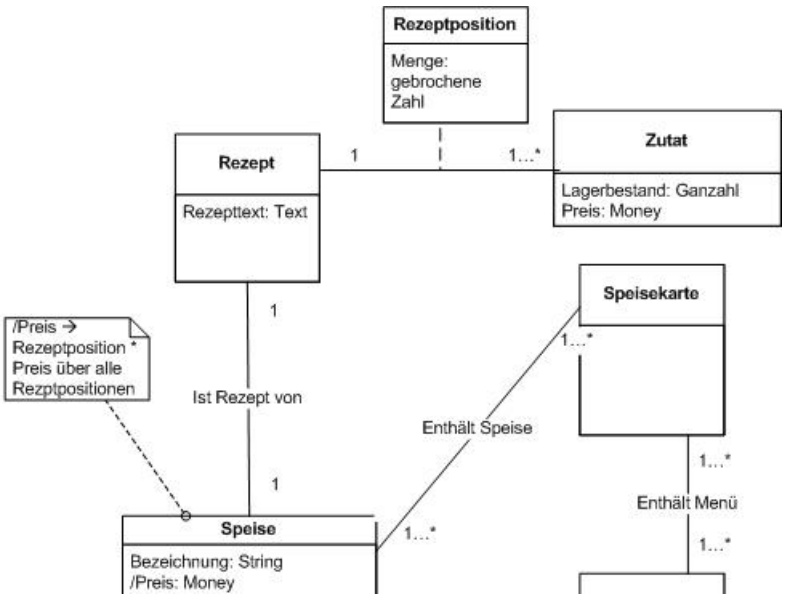

Abbildung 2.1: Auszug aus einem Klassendiagramm der statischen Sicht auf einen Partyservice

Analyse- und Designphasen erstellt wurde und somit den Bauplan für das kommende Softwareprodukt darstellt. Die Basis für diese Architektur sind die Diagramme der erwähnten vier Sichten auf die Architektur. Zum Anderen hat jede Software zu jedem Zeitpunkt eine aktuelle IST-Architektur, die auf dem Quelltext des Softwaresystems basiert. Aus dem Quelltext lässt sich nicht direkt die Architektur ablesen, diese muss erst aus dem Quelltext heraus lesbar gemacht werden.

2.2 Softwarearchitekturanalyse

Um dem Verfall der Softwarearchitektur entgegenzuwirken, muss die IST-Architektur fortlaufend analysiert werden. Die Architektur eines Systems kann degenerieren, da z.B. unter Zeitdruck neue Funktionalität programmiert wurde, die nicht mit der SOLL-Architektur übereinstimmen. Ein weiteres Beispiel ist das Verständnis des Entwicklerteam für die SOLL-Architektur nicht klar vorhanden ist, sodass Korrekturen und Erweiterungen erstellt werden, die zwar funktionieren, aber nicht die Architektur berücksichtigen. Somit ist eine kontinuierliche Analyse der Architektur nötig um sowohl die Architektur einzuhalten, aber auch weitere Qualitätsziele und Richtlinien, wie z.B. niedrige Kohäsion, zu prüfen. Die statische Architekturanalyse befasst sich mit der statischen Sicht und nimmt als Grundlage den Quelltext des Systems. Hier gibt es bislang einen Schnitt im Entwicklungszyklus, da ein weiteres Werkzeug eingesetzt werden muss, wie z.B. Sotograph (Hello2Morrow (2008)). Die Ergebnisse dieser

Werkzeuge müssen vom Anwender interpretiert und mit der SOLL-Architektur verglichen werden. Dies erfordert viel Aufwand und Erfahrung, da eine manuelle Übertragung der Ergebnisse auf die SOLL-Architektur vorgenommen werden muss. Die Ergebnisse solch einer Analyse sind meist sehr technisch und metrikenbasiert, es ergibt sich eine Fülle an Ergebnissen die erst sondiert und danach ausgewertet werden müssen. In dem hier dargestellten Ansatz soll eine visuelle Architekturanalyse auf Grundlage der Modelle der SOLL-Architektur durchgeführt werden. Daher wurden folgende minimale Qualitätskriterien für diese Art der Architekturanalyse gewählt:

- 100% der Komponenten inkl. Schnittstellen und deren Beziehungen zu anderen Komponenten sollen vorhanden sein
- Es sollen nicht mehr Komponenten oder Beziehungen als in der SOLL-Architektur vorhanden sein

2.3 Abgrenzung zur metrikbasierten Analyse

Die metrikbasierte Analyse von Quelltexten soll den Quelltext für Qualitätsziele messbar machen. Viele der vorhandenen Werkzeuge zur Architekturanalyse basieren auf Metriken (vgl. SonarJ, Sotograph). Dabei werden auch Fehler und Probleme der SOLL-Architektur aufgedeckt. In dieser Arbeit soll aber der visuelle Vergleich als Kommunikationsmittel mit dem Entwicklerteam im Vordergrund stehen, daher ist es nicht das primäre Ziel, eine metrikbasierte Analyse durchzuführen, sonder die Einhaltung der SOLL-Architektur steht hier im Fokus.

3 Szenario

In diesem Kapitel soll ein Szenario aufgezeigt werden, welches ein Bespiel dafür liefern soll, wie in ein Softwareerstellungsprojekt aufgestellt sein kann und wie darin mit der Architekturanalyse umgegangen wird.

Am Standort Hamburg möchte eine Firma XY eine neue Anwendung in dem Bereich Gastronomie erstellen. Dabei sollen Entwicklungskosten dadurch gespart werden, dass das Entwicklerteam dezentral aufgestellt wird, indem ein Architekt und eine Qualitätsmanager in Hamburg und ein Entwicklerbüro in Indien für dieses Projekt engagiert werden. Da die Analyse in den Fachbereichen ein langwieriger Prozess ist, wird ein iteratives und prototypisches Vorgehen gewählt. Alle Beteiligten sind über den groben Ablauf informiert und bereit mit den ersten Schritten zu beginnen.

Mit dem Management der Firma XY wurde das folgende Vorgehen vereinbart: Für jede Itera-

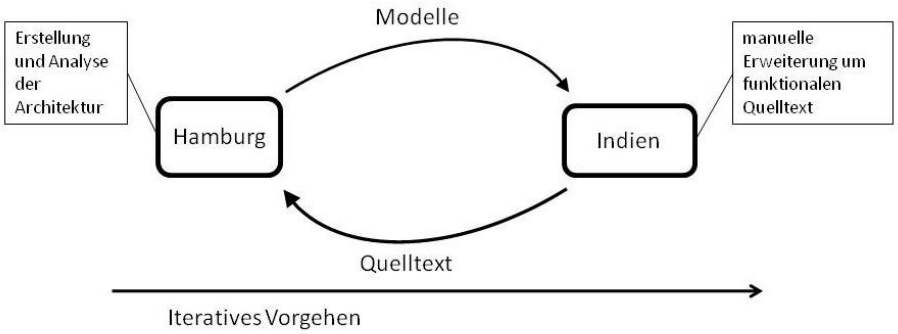

Abbildung 3.1: Vorgehensmodell des Szenarios

tion werden die Anforderungen aus dem Fachbereich und die projektinternen Qualitätsziele durch den Softwarearchitekten und Qualitätsmanager als Softwarearchitektur definiert und mittels geeigneter UML 2.0 Diagrammen dargestellt. Diese Soll-Architektur sollen den Rahmen/Bauplan für das Entwicklungsbüro in Indien darstellen .Zusätzlich müssen die fachlichen Anforderungen in textueller Form nach Indien gegeben werden. Auf diesen Informationen basierend erstellt das Entwicklungsbüro in Indien die für die jeweilige Iteration definierten Module und sendet den Quelltext inkl. Dokumentation zu Händen des Softwarearchitekten und Qualitätsmanagers nach Hamburg.

Nun beginnt die an jedem Ende einer Iteration durchzuführende Analyse des vorliegenden Quelltextes.

Aktuelle Analysewerkzeuge analysieren den Code metrikenbasiert und geben als Ergebnisse Tabellen mit ermittelten Metriken aus. Diese müssen dann vom Architekten und Qualitätsmanager nach Relevanz verdichtet und ausgewertet werden. Anschließend müssen die Metriken, also Messwerte des Quelltextes mit der SOLL-Architektur abgeglichen werden. Dazu muss manuell abgeglichen werden, ob alle Komponenten und Schnittstellen vorhanden sind. Die Beziehungen untereinander sind über Metriken nur Schwer nachzuvollziehen und werden von den meisten Werkzeugen graphisch dargestellt. Auch hier muss eine manuelle Übertragung aus dem Werkzeug auf die SOLL-Architektur erfolgen. Bei gefundenen Differenzen informieren er Architekt und der Qualitätsmanager das Entwicklungsbüro in Indien über notwendigen Nachbesserungen. Hierzu muss nun manuell eine Übersicht erstellt werden, an welchen Stellen die SOLL-Architektur nicht korrekt wurde.

Die Vision dieser Ausarbeitung ist es, ein Werkzeug für den Architekten und Qualitätsmanager bereitzustellen, welches die Analyse anhand der SOLL-Architekturdiagramme als Blaupause und dem Quelltext des Entwicklungsbüros vornimmt.

Das Ergebnis soll ein Diagramm sein, das dem Aufbau der SOLL-Architekturdiagramme entspricht und visuell die Differenzen hervorhebt. Dieses Diagramm und zusätzlich textuelle Ergänzungen können dann als Kommunikationsgrundlage mit dem Entwicklungsbüro in Indien genutzt werden. Es sollen weniger Übertagungsschritte notwendig sein, um die Differenzen zu erkennen und dann gezielt zu analysieren. Dieses Vorgehen würde sich über die diversen Iterationen wiederholen und würde die Einhaltung der SOLL-Architektur sichern und den Prozess des Abgleich beschleunigen.

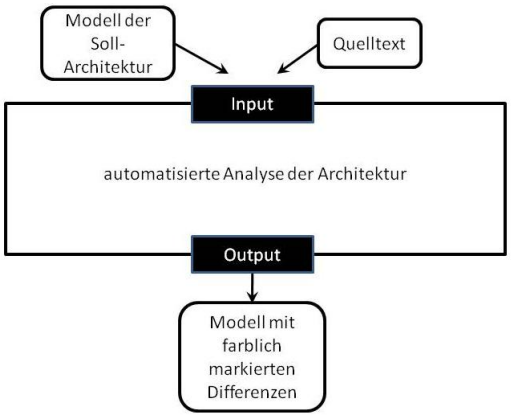

Abbildung 3.2: Blackbox

4 Ansatz zur Realisierung

Um einen Vergleich der SOLL- mit der IST-Architektur mittels UML 2.0 Diagrammen durchzuführen muss eine Möglichkeit gefunden werden UML 2.0 Diagramme mit Quelltext zu vergleichen. Dazu sollten einige Konventionen im Entwicklungsprozess eingehalten werden, damit solch ein Abgleich funktionieren kann. So sollte Pakete aus UML 2.0 auch Pakete in der jeweiligen Programmiersprache und Klassennamen auch die Klassennamen im Quelltext sein. Im Grunde genommen ist eine möglichst direkte Überführung der Vorgaben aus dem Diagramm in den Quelltext durchzuführen. Im Folgenden werden zwei mögliche Ansätze diskutiert

4.1 Bottom Up

Bei dem Bottom Up Ansatz soll aus dem Quelltext ein entsprechendes UML 2.0 Diagramm erstellt werden, welches dann mit dem Diagramm der SOLL-Architektur abgeglichen und Differenzen hervorgehoben werden sollen. Diese Methode, aus einem Quelltext diverse UML

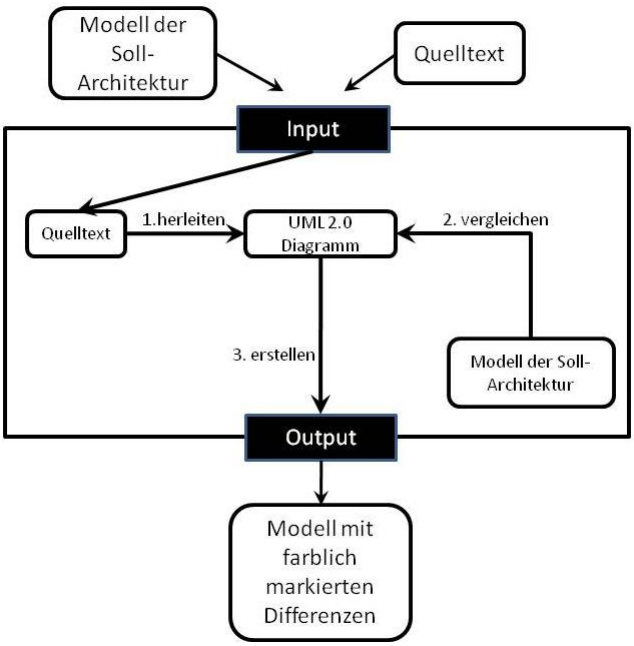

Abbildung 4.1: Bottom Up Ansatz

2.0 Diagramm zu erstellen, ist bislang noch nicht vollständig gelöst, aber für gewählte Diagramme gibt es Ansätze (Vinita u. a. (2008)). Es besteht schon das Problem zu erkennen, welche Art von Diagramm erstellt werden soll. Um eine Verbindung zwischen Diagramm und Quelltext herzustellen, ist ein Ansatz, den Quelltext um Annotationen zu erweitern. Anhand solcher Annotationen können Komponenten, Klassen und Schnittstellen einem entsprechenden Diagrammbestandteil zugeordnet werden. Annotationen müssen dem Quelltext entweder manuell hinzugefügt werden, dies bedeutet aber einen erheblichen Aufwand während der Entwicklungsphase. Eine weitere Möglichkeit ist, dass aus der Soll-Architektur gleich zu Beginn unter Einsatz eines Code-Generators das Gerüst, also die Komponenten, Schnittstellen und Beziehungen, automatisiert erstellt werden. An dieser Stelle kann der Quelltext sofort automatisiert um die benötigten Annotationen erweitert werden. Als Basis der Code-Generation ist dann die SOLL-Architektur bzw. die Darstellung dieser per UML 2.0 zu nehmen, sodass nach der Code-Generation der IST-Zustand exakt dem SOLL-Zustand der Architektur entspricht.

Code-Generation aus UML 2.0 Diagrammen wird auch in der modellgetriebenen Softwareentwicklung (MDSD) eingesetzt. Die an in dem MDSD Ansatz eingesetzten Mechanismen können an dieser Stelle angewandt werden. Der generierte Code wäre dann in zwei Teile unterteilt, zum Einen der generierte und anschließend nicht mehr zu verändernde Anteil und zum Anderen die eigentliche Funktionalität, die noch manuell erstellt werden muss. Es muss sichergestellt sein, dass an den Annotationen und dem generierten Quelltext keine Veränderungen durchgeführt werden dürfen und können.

Da nun der Quelltext so gekennzeichnet ist, dass eine Abbildung der Klassen, Schnittstellen und Beziehungen vorgenommen werden kann, ist noch ein Mechanismus zu erstellen, der aus dem Quelltext ein UML 2.0 Diagramm erstellt. Die Art des Diagramms, also ob es sich beispielsweise um Verteilungs- oder Klassendiagramm handelt, kann daran erkannt werden, welchen Diagrammtyp die SOLL-Architektur entspricht, da diese als Input der Analyse vorliegt Angenommen, diese Mechanismen sind realisiert, erhält man ein UML 2.0 konformes Diagramm, das dem Typ der als Input hinterlegten SOLL-Architektur entspricht. Nun muss ein Abgleich der beiden Diagramme durchgeführt werden, um zu sehen, ob die SOLL-Architektur eingehalten wurde. Dabei muss gewährleistet sein, dass die einzelnen Zeichnungselemente der Diagramme eindeutig gekennzeichnet und automatisiert angesprochen werden können. So könnte über eine Mengenablgeichsverfahren geprüft werden, ob alle Objekte, z.B. über den Namen, in beiden Diagrammen vorliegen. Bei Beziehungsdarstellung zwischen zwei Komponenten müsste die Richtung ebenfalls betrachtet werden, also das Klasse A einen Dienst der Klasse B nutzt und nicht umgekehrt. Bei einem positiven Abgleich eines Objektes sollte dieselbe Darstellung im Diagramm gewählt werden, wie bei dem Inputdiagramm. Bei Abweichungen, also entweder ein Objekt aus der Soll-Architektur ist nicht oder ein Objekt ist in der IST-Architektur zu viel vorhanden, muss dieses farblich abgehoben, am Besten in einer Signalfarbe, in das Output-Diagramm aufgenommen werden.

Nach diesen Schritten ist das Layout des Output-Diagramms noch dem Layout des Input-Diagramms entsprechend anzupassen, damit der Wiedererkennungseffekt verstärkt wird. Auch dies sollte automatisiert angepasst werden, somit gilt auch hier, dass die einzelnen Elemente der Soll-Architektur in dem Diagramm entsprechend angesprochen werden müssen. Dadurch kann ein ähnliches Layout erstellt werden.

4.2 Top Down

Der Top Down Ansatz soll im Gegensatz zum Bottom Up Ansatz nicht aus der IST-Architektur (Quelltext) das Ergebnisdiagramm erzeugen, sondern aus der Soll-Architektur Code erstellen, der dann mit dem Quelltext der IST-Architektur verglichen wird. Dabei kann wie beim

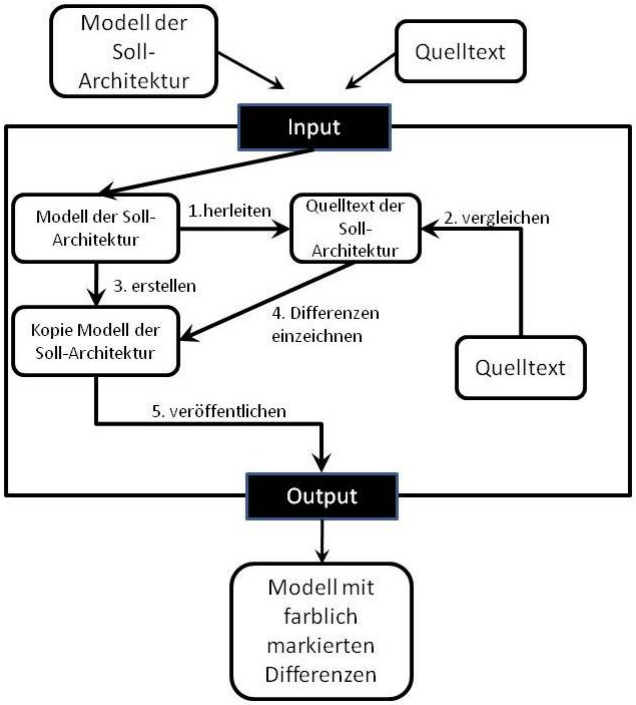

Abbildung 4.2: Top Down Ansatz

Bottom Up Ansatz auf bekannte Technologien in Form von Code-Generatoren zurückgegriffen werden. Dazu muss derselbe Generator verwendet werden, der auch für das Gerüst der IST-Architektur benutzt wurde, damit dasselbe Gerüst erstellt wird. Dieses enthält alle Komponenten, Schnittstellen und Beziehungen der SOLL-Architektur ohne manuell hinzugefügte Funktionalität. Anschließend soll im anhand des Quelltextes verglichen werden, ob alle Elemente der SOLL-Architektur vorhanden sind. Abweichungen sollen in eine Kopie des SOLL-

Architekturdiagramms eingezeichnet und farblich abhoben werden, sodass Abweichungen schnell zu lokalisieren sind.

Komplizierter ist die Erkennung von zusätzlichen Elementen, die nicht in der SOLL-Architektur eingetragen sind. Die Lokalisierung von Komponenten und Schnittstellen kann z.B. über den Namen erfolgen. Zusätzliche Komponenten oder Schnittstellen können recht einfach in das Ergebnisdiagramm eingezeichnet werden. Um Beziehungen zwischen Komponenten erkenne zu können, muss bekannt sein, wie die jeweilige Programmiersprache jegliche mögliche Beziehungen darstellt. Eine Aggregation in UML 2.0 wird in Java beispielsweise als Collection von der aggregierbaren Klasse dargestellt. So führt eine Collection von anderen Komponenten im Quelltext zu einer Aggregation oder Assoziation in dem UML 2.0 Diagramm. Einfache Beziehungen sind dadurch charakterisiert, dass keine Collections, sonder einfach die jeweilige Komponenten oder Schnittstellen von einer anderen Komponente genutzt wird. So können alle Arten von Beziehungen im Quelltext Programmiersprachenabhängig definiert und entsprechend in ein Diagramm überführt werden.

Zusätzliche Elemente sollte ebenfalls farblich gesondert in das Diagramm übertragen werden, damit diese schnell erkannt werden können. Ebenso wie beim Bottom Up Ansatz sollte für das Ergebnisdiagramm ein ähnliches Layout wie bei der SOLL-Architektur für den Wiedererkennungswert gewählt werden.

4.3 Code Generierung

In dem Ansatz der modellgetriebenen Softwareentwicklung (MDSD) wird der Einsatz von Code-Generatoren (Stahl u. a. (2007)) genutzt, um automatisiert aus Modellen den Quelltext eines Systems zu erstellen. Da in Modellen meist nicht jedes Detail der fachlichen Anforderungen, da Modelle zur Abstraktion eingesetzt werden, umgesetzt werden kann, unterteilt auch der MDSD -Ansatz den Quelltext in den generierten, nicht mehr veränderlichen, und den manuell erweiterbaren Teil. Aus dem MDSD-Ansatz gibt es zahlreiche Vorgehen und Methoden, wie ein Code-Generator erstellt werden kann. Alle Code-Generatoren haben eine bestimmte Abfolge, mittels derer der Quelltext erstellt wird:

1. Modell einlesen
2. Metamodellabbildung des Modells erstellen
3. Definierte Übersetzung des Metamodells in jeweilige Syntax der Zielprogrammiersprache

Beim Einlesen des Modells muss als erstes identifiziert werden, um welche Art von Modell es sich handelt und danach müssen alle Elemente (Komponenten, Schnittstellen und Beziehungen) gesammelt werden. Hierzu muss entweder ein Werkzeug erstellt werden, dass Modelle läd oder nach Möglichkeiten gesucht werden, bestehende Werkzeuge für diese Aufgabe zu nutzen (vgl. Kapitel 4.4). Sind alle Informationen des Modells gesammelt, werden diese in das Metamodell der jeweiligen Modellart übersetzt. Das Metamodell beschreibt, aus welchen Komponenten ein Modell bestehen darf und wie diese in Beziehung miteinander stehen dürfen, also ein Modell für das Modell. Für diesen Schritt gibt es bereits einige Metamodelle, die an dieser Stelle genutzt werden können. So stellt das sog. Meta Object Facility (MOF) einen Ansatz für die Metamodellierung der in UML 2.0 genutzten Modelle dar.

Das Metamodell eines Modells kann einheitlich als Klassendiagramm repräsentiert werden, somit ist eine Transformation von jeglichen UML 2.0 Modellen in Klassendiagramme möglich. Die Übersetzung von Klassendiagrammen in Quelltext wird durch eine Vielzahl von Generatoren ermöglicht, daher ist der Zwischenschritt über das Metamodell nützlich. Als letztes muss noch das nun vorliegende Metamodell, das eine Abstraktion der SOLL-Architektur darstellt in den Quelltext übersetzt werden. Auch wenn viele Code-Generatoren die Übersetzung der Klassendiagramme in Quelltext beherrschen, muss noch die Semantik der jeweiligen Diagramme definiert werden. Templates sind eine Möglichkeit in einer Art Skriptsprache zu definieren, wie eine bestimmte Art von Modell als Quelltext dargestellt wird. Solch ein Template ist zwar sehr aufwendig zu generieren, aber es ist somit möglich die exakte Semantik der jeweiligen Komponenten darzustellen und auch um z.B. Annotationen zu erweitern.

4.4 Automatisierte Generierung von UML 2.0 Diagrammen

Es ist für die Analyse der Architektur unter Einsatz vom UML 2.0 Diagrammen notwendig, dass aus der laufenden Analyse neue Diagrammen automatisiert erstellt werden. Visio ist ein Werkzeug zur Erstellung von UML 2.0 Diagrammen, dass dem Entwickler einen breiten Satz an Modellen und unterstützenden Werkzeugen zur Verfügung stellt. Visio stellt mit dem sog. "primären Interopassembly"(Microsoft (2008)) eine Interaktionsschnittstelle zu Visio bereit, mittels derer aus dem Programmfluss der Analyse auf die Objekte der Modelle zugegriffen und diese auch verändert werden können. Dieses Werkzeug ermöglicht somit zum Einen das Einlesen der Input-Diagramme als auch die Veränderung und Erstellung der Output-Diagramme, wie es nach die "Top Down und "Bottom Up Ansätze (vgl. Kapitel 4.1 und 4.2) benötigt wird. Es ist an dieser Stelle nicht festgelegt, dass Visio als Modellierungswerkzeug genutzt werden soll, jedes andere Werkzeug das über eine entsprechende Interoperabilität verfügt, kann eingesetzt werden.

5 Risiken

Der in Kapitel 4 aufgezeigte Ansatz zur automatisierten Architekturanalyse mittels UML 2.0 Diagrammen besteht aus eine Vielzahl von Werkzeugen, die zusammenarbeiten müssen, damit ein Ergebnis erstellt werden kann. So ist es notwendig einen Loader für UML 2.0 Diagramme einzusetzen, der die UML 2.0 Diagramme aus Programmen heraus verfügbar macht. Des Weiteren wird ein Werzeug zum Vergleich von Quelltexten, ein Werkzeug zum Vergleich vom UML 2.0 Diagrammen udn ein Code-Generator benötigt.

Jedes einzelne Werkzeug besitzt eine hohe Komplexität, die eine Einarbeitung und Anpassung an die benötigten Kriterien nötig macht. So muss z.B. der Code-Generator teilweise selbst erstellt werden, wenn nicht ein genau passendes Werkzeug zur Verfügung steht. Das Zusammenspiel der einzelnen Werkzeuge erhöht die Komplexität nochmals, da sichergestellt sein muss, dass alle Werkzeuge kompatibel sind. Sollten einzelne Werkzeuge nicht kompatibel oder entsprechend erweiterbar sein, so ist es notwendig bestehende Lösungen zu erweitern oder selbst zu erstellen. Aufgrund der Komplexität der einzelnen Werkzeuge können Anpassungen oder Erstellung ein sehr Aufwendiger Prozess sein.

Dann sollte die Analyse möglichst für viele Programmiersprachen und Plattformen möglich sein. Dies schränkt die Wahl der Werkzeuge weiter ein, da z.B. Visio derzeit nur aus der .Net-Plattform heraus angesprochen werden kann. Die kann bedeuten, dass für verschiedene Plattformen und Programmiersprachen das komplette Analysewerkzeug jeweils neu zusammengestellt bzw. erstellt werden muss, was erneut die Komplexität und Dauer der Erstellung verlängert.

Ein weiteres Risiko besteht darin, dass in diesem bisher dargestellten Ansatz die SOLL-Architektur als Input der Analyse gilt, aber bisher die SOLL-Architektur keinem Test oder Analyse unterliegt. Die Analyse einer Architektur sollte aber auch die SOLL-Architektur umfassen, sodass eine weitere Komponente für eine Untersuchung der SOLL-Architektur nötig erscheint. Diese sollte "best practisePattern, wie z.B. die lose Kopplung, Zyklenfreiheit oder niedrige Kohäsion der SOLL-Architektur prüfen. Viele dieser "best practisePattern können über bestehende metrikbasierte Werkzeuge geprüft werden, sodass eine weitere Komponente im Analysewerkzeug eingeführt werden sollte.

Um die aufgezeigten Risiken zu minimieren, sollte die Architektur des Analysewerkzeugs möglichst modular gehalten werden, damit einzelnen Komponenten getauscht und neue hinzugefügt werden können. Hierdurch wäre eine Erweiterung um metrikbasierte Werkzeuge und die Anpassung an div. Plattformen ermöglicht.

6 Ausblick

Ein erster Prototyp für die hier vorgestellte Architekturanalyse soll erstellt werden. Dazu soll zunächst nur eine Plattform und eine Programmiersprache genutzt werden. Um die prinzipielle Funktionsweise zu validieren, soll als Vereinfachung im ersten Schritt nur ein UML 2.0 Diagramm unterstützt werden, das die statische Sicht darstellt. Dies kann entweder eine Klassen- oder ein Komponentendiagramm sein. Dazu muss ein passender Code-Generator und Loader für das Diagramm erstellt werden. Da schon einige frei verfügbare und anpassbare Code-Generatoren für die .Net-Umgebung verfügbar sind und Visio hiermit ansprechbar ist, fällt die Wahl auf .Net als Programmierplattform und Visio als UML 2.0 Werkzeug. Das in dieser Ausarbeitung dargestellt Szenario soll umgesetzt und die Architektur verglichen werden.

Literaturverzeichnis

[Bass u. a. 2008] BASS, Len ; CLEMENTS, Paul C. ; KAZMAN, Rick ; KLEIN, Mark: Evaluating the Software Architecture Competence of Organizations. In: *WICSA*, IEEE Computer Society, 2008, S. 249–252

[Hello2Morrow 2008] HELLO2MORROW: *Sotograph - Overview*. 2008. – URL http://www.hello2morrow.com/products/sotograph

[Kecher 2005] KECHER, Christoph: *UML 2.0. Das umfassende Handbuch*. Galileo Press, 2005. – URL http://www.amazon.de/exec/obidos/redirect?tag=citeulike01-21\&path=ASIN/3898427382. – ISBN 3898427382

[Medvidovic u. a. 2002] MEDVIDOVIC, Nenad ; ROSENBLUM, David S. ; REDMILES, David F. ; ROBBINS, Jason E.: Modeling software architectures in the Unified Modeling Language. In: *ACM Trans. Softw. Eng. Methodol.* 11 (2002), Nr. 1, S. 2–57. – ISSN 1049-331X

[Microsoft 2008] MICROSOFT: *Übersicht über das Visio-Objektmodell*. 2008. – URL http://msdn.microsoft.com/de-de/library/cc160740.aspx

[Stahl u. a. 2007] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, 2007. – URL http://www.amazon.de/exec/obidos/redirect?tag=citeulike01-21\&path=ASIN/3898644480. – ISBN 3898644480

[Vinita u. a. 2008] VINITA ; JAIN, Amita ; TAYAL, Devendra K.: On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms. In: *SIGSOFT Softw. Eng. Notes* 33 (2008), Nr. 5, S. 1–9. – ISSN 0163-5948

[Züllighoven u. a. 2006] ZÜLLIGHOVEN, Heinz ; LILIENTHAL, Carola ; BENNICKE, Marcel: Software Architecture Analysis and Evaluation. In: HOFMEISTER, Christine (Hrsg.) ; CRNKOVIC, Ivica (Hrsg.) ; REUSSNER, Ralf (Hrsg.): *QoSA* Bd. 4214, Springer, 2006, S. 7–8. – ISBN 3-540-48819-7

Stahl u. a. (2007) Vinita u. a. (2008) Kecher (2005) Microsoft (2008) Hello2Morrow (2008)