Masterarbeit

Thorben Pergande

Modellbasierte Softwarearchitekturanalyse

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thorben Pergande

Modellbasierte Softwarearchitekturanalyse

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Stefan Sarstedt
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 27. Mai 2011

Thorben Pergande

Thema der Masterarbeit

Modellbasierte Softwarearchitekturanalyse

Stichworte

Softwarearchitektur, Modellierung, Analyse, Softwareanalyse, Vergleiche, Diagramme, Abbildbarkeit

Kurzzusammenfassung

Diese Arbeit befasst sich mit der modellbasierten Architekturanalyse. Dabei soll herausgearbeitet werden, in wie weit es möglich und sinnvoll ist, Diagramme, die eine Softwarearchitektur beschreiben, direkt mit einem Softwaresystem zu vergleichen. Hierzu sollen Möglichkeiten, Chancen und Grenzen erarbeitet werden. Darüber hinaus sollen die Analyseergebnisse durch einen Prototypen für drei verschiedene Diagrammtypen realisiert werden. Die Ergebnisse der Realisierung werden abschließend anhand eines Testlaufs ausgewertet.

Thorben Pergande

Title of the paper

Model-based software architecture analyses

Keywords

software architecture, modeling, analyses, software analyses, comparison, diagrams, mapping

Abstract

This work deals with the model-based architecture analysis. The aim is to figure out how far it is possible and useful to compare models directly to software systems. For this purpose, possibilities, opportunities and limits will be developed. In addition, the analysis results can be achieved by a prototype for three different types of diagrams. The results of the implementation will be finally evaluated by a test run.

Inhalt

1 Einleitung 1

1.1 Motivation 1

1.2 Ziel der Arbeit 2

1.3 Aufbau der Arbeit 3

2 Vergleichbare Arbeiten 4

2.1 „UML-based Design Test Generation“ 4

2.2 „On reverse engineering an object-oriented code into UML class diagrams incorporation extensible mechanisms“ 5

2.3 „Metamodel Approach on Model Conformance and Multiview Consistency Checking“ 6

2.4 „Automatic Code Generation: A Practical Approach“ 7

2.5 Abgrenzung 8

3 Allgemeine Grundlagen 11

3.1 Softwarearchitektur 11

3.1.1 Soll- und Ist-Architektur 11

3.1.2 Referenzarchitekturen 12

3.2 Softwarearchitekturanalyse 12

3.3 Modellierung 13

3.3.1 Unified Modeling Language 14

3.3.2 Andere Modellierungsvarianten 16

3.3.3 Meta Object Facility / Metamodellierung 16

4 Technische Grundlagen 17

4.1 Programmiersprache C# 17

4.1.1 Reflections 17

4.1.2 Common Intermediate Language (CIL) 18

4.2 XMI- Austauschformat 18

4.3 Modellierungswerkzeuge 20

5 Problemstellung und Anforderungen 21

5.1 Problemstellung 21

5.2 Anforderungen und Einschränkungen 22

6 Analyse 25

6.1 Analyse des Problemfeldes des Diagramm-Imports 25

6.2 Analyse der Quelltexteingabemöglichkeiten 29

6.3 Analyse des Problemfeldes der Vergleichbarkeit 30

6.4 Analyse des Problemfelds der Abbildbarkeit 33

6.5 Zusammenführung der Problemfelder Vergleichbarkeit und Abbildbarkeit 35

6.6 Vergleichsoperationen und Repräsentation der Ergebnisse 37
6.7 Fazit zur Analyse 40
7 Design und Systemarchitektur 42
7.1 Realisierung des Zwischenformats und Definition der Abbildungsregeln 42
7.1.1 Schichtendiagramme & C# 43
7.1.2 Klassendiagramme & C# 44
7.1.3 Komponentendiagramme & C# 47
7.2 Systemarchitektur 50
7.2.1 XML-Diagramm und Quellcode 50
7.2.2 Softwarearchitekturanalysierer 51
7.2.3 Diagramm-Abbilder 51
7.2.4 Ist-Repräsentation-Abbilder 52
7.2.5 Vergleicher 53
7.2.6 Ergebnisersteller 54
7.2.7 Ergebnis-Diagramm 55
8 Implementierung 56
8.1 Zwischenmodell-Implementierung 56
8.2 Diagramm-Abbilder 57
8.3 Ist-Architektur-Abbilder 58
8.3.1 Fachliche Klassenerkennung 58
8.3.2 Grundlegender Umgang mit Reflections und Extraktion der Information 59
8.3.3 Grundlegender Umgang mit der Verbindungserkennungen bei C#-Klassen 61
8.3.4 Laden des entsprechenden Zwischenmodells 67
8.3.5 Umsetzung der Abbildungsregeln für die unterstützen Diagramme 68
8.4 Vergleiche der Abbildungen 75
8.5 Export der Ergebnisse 76
9 Auswertung und Optimierung 80
9.1 Untersuchung der Schichtendiagramme 80
9.1.1 Anwendung und Auswertung der Analyse auf Schichtendiagramme 80
9.1.2 Optimierung des Prototypen für die Schichtendiagrammanalyse 82
9.1.3 Bewertung der Ergebnisse bezüglich der Schichtendiagrammanalyse 84
9.2 Untersuchung der Klassendiagramme 84
9.2.1 Anwendung der Analyse mit Klassendiagrammen 84
9.2.2 Auswertung der Analyse mit Klassendiagrammen 87
9.2.3 Optimierung des Prototypen für die Klassendiagrammanalyse 88
9.2.4 Bewertung der Ergebnisse bezüglich der Klassendiagrammanalyse 91

9.3 Untersuchung der Komponentendiagramme 92
9.3.1 Anwendung der Analyse auf Komponentendiagramme 92
9.3.2 Auswertung der Analyse auf Komponentendiagramme 95
9.3.3 Optimierung des Prototypen für die Komponentendiagrammeanalyse 96
9.3.4 Bewertung der Ergebnisse bezüglich der Komponentendiagrammeanalyse ...98
10 Abschließende Betrachtung 99
10.1 Zusammenfassung 99
10.2 Weiterführende Arbeiten 101
Abbildungsverzeichnis 102
Tabellenverzeichnis 103
Literaturverzeichnis 104
Versicherung über die Selbstständigkeit 107

1 Einleitung

In den folgenden Unterkapiteln werden die Motivation und Ziele inklusive der Vision dieser Arbeit vorgestellt. Des Weiteren wird der Aufbau dieser Arbeit beschrieben.

1.1 Motivation

Die Architektur eines Softwaresystems beschreibt den grundlegenden Aufbau und die Struktur dieser Software. Die Einhaltung der Vorgaben zu einer Softwarearchitektur muss fortlaufend geprüft werden, um dem Architekturverfall, der sogenannten Architekturerosion, entgegenzuwirken. Architekturerosion geschieht sowohl durch mangelndes Verständnis der Architektur und mangelnde Kommunikation der Vorgaben als auch z.B. durch erhöhten Projektdruck. Die Folgen von Architekturerosion können vielfältig sein, so können beispielsweise Qualitätsziele, wie Erweiterbarkeit oder Wartbarkeit, nicht wie gewünscht eingehalten oder die Testbarkeit verringert werden. Allgemein kann es durch Abweichungen von den Architekturvorgaben ebenfalls zu unerwünschten Seiteneffekten zwischen den Softwarebestandteilen kommen, wie z.B. die erhöhte Kopplung zwischen einzelnen Klassen (Becker-Pechau, 2010).

Um der Architekturerosion entgegenzuwirken, ist es nötig zu prüfen, ob die aktuellen Architekturvorgaben in der realisierten Software eingehalten und umgesetzt wurden. Dieser Vorgang wird als Softwarearchitekturanalyse beschrieben und kann über verschiedene Wege durchgeführt werden. So kann durch Spezialisten ein Review, d.h. die manuelle Überprüfung des Quelltextes, des Softwaresystems erfolgen. Dies ist jedoch bei großen und komplexen Softwaresystemen ein sehr aufwändiger und zeitintensiver Vorgang, der nicht zu jedem beliebigen Zeitpunkt durchgeführt werden kann. Ähnlich verhält es sich bei der metrikbasierten Softwarearchitekturanalyse. Hierbei wird ein Softwaresystem automatisiert auf Kennzahlen abgebildet, wodurch z.B. die Kopplungsstärke oder Komplexität (Hoffmann, 2008) einzelner Teile ermittelt werden kann.

Nach der Erhebung der Kennzahlen müssen diese erneut durch Spezialisten auf die Architekturvorgaben zurückgeführt und auf dieser Basis analysiert werden. Dieses Vorgehen ist sehr genau, da die Kennzahlenermittlung durch Automaten erfolgen kann, die nach definierten und verbreiteten Regeln funktionieren. Die Auswertung ist jedoch, ähnlich wie bei Reviews, sehr aufwändig und ein manueller Vorgang, der durch Spezialisten ausgeführt werden muss.

Der Weg zu einer einzusetzenden Softwarearchitektur verläuft über fachliche Anforderungen, die Umsetzung von Geschäftsprozessen und die Definition von Qualitätsmerkmalen oder Erfahrungswerten, sogenannte Best Practices. Die Erstellung und Definition einer Softwarearchitektur kann unter anderem durch textuelle Dokumentation, Definition von Szenarios oder durch Modellierung von Diagrammen stattfinden. Hierzu gibt es standardisierte Diagrammtechniken wie die Unified Modeling Language (UML) oder werkzeuggestützte Geschäftsprozessmodellierungen. Diese

Techniken ermöglichen es, ganze Architekturen oder Teile davon in Form von Diagrammen zu erstellen.

Die derzeitigen Werkzeuge, die eine Softwarearchitekturanalyse unterstützen sollen, haben zumeist keine Schnittstelle oder Anwendung für den Abgleich mit einem Diagramm. Folglich gibt es keine direkte Möglichkeit, Diagramme Ad hoc mit der Realisierung eines Softwaresystems zu vergleichen. Desweiteren sind die bereits vorgestellten Ansätze, Reviews und die metrikenbasierte Softwarearchitekturanalyse, sehr zeitaufwändig und zumeist nicht direkt von den Entwicklern durchzuführen. Dies erschwert sowohl eine kontinuierliche und kurzfristige Softwarearchitekturanalyse als auch die Vermittlung der Ergebnisse, da die Ergebnisse ohne die Auswertung durch Spezialisten nicht selbsterklärend sind.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, eine Antwort auf die Frage zu finden, ob und wie ein direkter Vergleich von modellierten Diagrammen und der Realisierung eines Softwaresystems möglich ist.

Softwarearchitekturanalysen sollen direkt aus einem Entwicklungsteam heraus durchgeführt werden können, sodass die Ergebnisse der Analysen für den Anwenderkreis selbständig auswertbar und kommunizierbar sind. Zur Erreichung dieser Ziele soll ein Werkzeug entwickelt werden, welches sowohl ein Modell bzw. Diagramm, als auch einen Repräsentanten eines zu untersuchenden Softwaresystems miteinander vergleicht. Das Ergebnis dieses Vergleichs soll ebenfalls als Diagramm dargestellt werden, in welchem die Übereinstimmungen und Abweichungen textuell und grafisch enthalten sind.

Durch die Verfolgung dieses Ansatzes soll es dem Entwicklungsteam ermöglicht werden, selbstständig und kurzfristig Architekturanalysen an den aktuell entwickelten Softwaresystemen, auch während der Entwicklungsphase durchzuführen. Die Ergebnisse sollen so dargestellt werden, dass diese zur Kommunikation mit anderen Mitgliedern aus dem Entwicklungsteam oder anderen Rollen, wie z.B. der Qualitätssicherung, genutzt werden können. Auf Basis der Ergebnisse kann nach einer kurzen Einarbeitungszeit in die Diagrammdarstellung ein gemeinsames Verständnis für den aktuellen Stand der Ist-Architektur erreicht werden. Es ist nicht das Ziel dieser Arbeit, die verbreiteten Architekturanalysemethoden, wie die metrikenbasierte Architekturanalyse, zu ersetzten, vielmehr ist eine Ergänzung für den direkten und schnellen Einsatz das Ziel.

Es wird in dieser Arbeit davon ausgegangen, dass eine Modellierung der Architektur, im folgenden Soll-Architektur genannt, stattgefunden hat. Daher soll das Ergebnisdiagramm vom selben Typ wie das eingegebene Diagramm sein, um die Einarbeitung in die Ergebnisanalyse zu vereinfachen. Die folgende Abbildung 1 skizziert die vorgestellten Ziele.

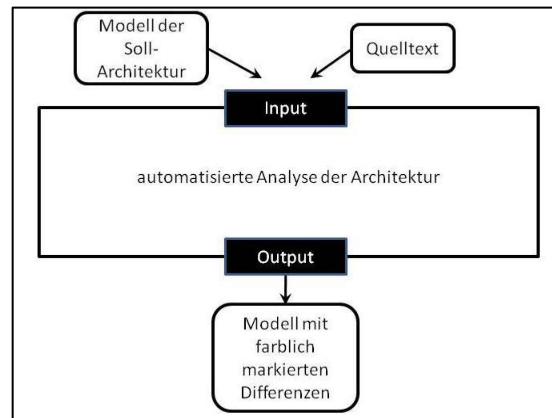

Abbildung 1: Vision einer modellbasierten Softwarearchitekturanaylse

Um die zu erarbeitenden Ansätze für die modellbasierte Softwarearchitekturanalyse zu prüfen, wird ein prototypisches Werkzeug für verschiedene Diagrammtypen und Softwarerealisationen erstellt. Mit Hilfe dieser Anwendung werden einzelne modellbasierte Softwarearchitekturanalysen durchgeführt werden.

1.3 Aufbau der Arbeit

Kapitel 1 umfasst die Motivation und die grundlegenden Ziele inklusive der Visionen dieser Arbeit.

Kapitel 2 zeigt die Hauptquellen dieser Arbeit auf, die aus der Recherche ausgesucht wurden. Dabei wurden hauptsächlich verschiedene Ansätze ausgewählt, die es ermöglichen, Abbildungen zwischen der Soll- und der Ist-Architektur herzustellen.

Die fachlichen Grundlagen, die vor allem den Kontext der Softwarearchitektur und deren Analyse aber auch die Modellierung in Form von Diagrammen erläutern, werden in Kapitel 3 behandelt. Kapitel 4 umfasst die technisch motivierten Grundlagen, die in dieser Arbeit genutzten Technologien werden näher beschrieben.

Kapitel 5 schärft und vertieft die unter Kapitel 1.2 dargestellten Ziele dieser Arbeit. Diese Ziele und Anforderungen werden in Kapitel 6 analysiert und an dieser Stelle werden Lösungsansätze für die vorhandenen Problemstellungen formuliert.

Aus den Ergebnissen der Analyse werden in Kapitel 7 Designentscheidungen und die allgemeine Systemarchitektur für die Realisierung des in dieser Arbeit beschriebenen Ansatzes zur modellbasierten Softwarearchitekturanalyse vorgestellt. Hieran schließt das Kapitel 8 an, in dem die Implementierung der realisierten Prototypen gezeigt wird.

In Kapitel 9 werden die implementieren Prototypen anhand festgelegter Testläufe getestet und die Ergebnisse der Testläufe vorgestellt. Hinzu kommt, dass in diesem Kapitel Verbesserungen und Optimierungsmöglichkeiten für jeden Prototypen entworfen, umgesetzt und dargestellt werden.

Abschließend werden in Kapitel 10 die Ergebnisse und der aktuelle Stand der Arbeit zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben.

2 Vergleichbare Arbeiten

In diesem Kapitel werden Ausarbeitungen vorgestellt, die in einem ähnlichen Kontext dieser Arbeit sind. Eine exakte Abbildung des hier vorgestellten Ansatzes hat die Recherche nicht ergeben, sodass vor allem Quellen oder Ansätze dargestellt werden, die einen Teil der Konstruktion oder Problemanalyse abdecken.

2.1 „UML-based Design Test Generation“

Die Autoren Waldemar Pires, Joao Brunet und Franklin Ramalho, allesamt von der Federal University of Campina Grande in Brasilien (Pires, et al., 2008), beschreiben einen Ansatz zur automatisierten Konformitätsprüfung von UML Klassendiagrammen und Java-Implementationen. Dabei werden Werkzeuge genutzt, um Design-Regeln, und somit Architekturregeln, als JUnit-Tests zu generieren.

Der bereits in vorherigen Veröffentlichungen vorgestellte „Design Wizard“ bietet Methoden zur Sammlung von Struktur-Informationen des Quelltexts, zum Beispiel über verwendete Klassen, Methoden und deren Referenzen zueinander. Aufgrund dieser Basis-Informationen können manuell formulierte Unit-Tests über JUnit durchgeführt werden, die Aussagen über Einhaltung und Nicht-Einhaltung von Design-Regeln ermöglichen. Dazu müssen das Design beschreibende Regeln erstellt und mittels JUnit-Test formuliert werden. Folgende Abbildung zeigt einen JUnit-Test, welcher prüft, ob ein Jar-Kompilat von einer Klasse B genutzt wird:

Code 1: DesignTest

```
1 public void testCommunication() {
2   DesignWizard dw;
3   dw = new DesignWizard(''project.jar'');
4   designwizard.ui.Class clazz;
5   clazz = dw.getClass(''A'');
6   Set<String> usedBy;
7   usedBy = clazz.getClassesUsedBy();
8   assertFalse(usedBy.contains(''B''))
9 }
```

Abbildung 2: Beispiel für einen JUnit-Test

Um im nächsten Schritt die Design-Regeln automatisiert aus konkreten Klassen-Diagrammen zu erstellen, verwenden die Autoren Methoden der modellgetriebenen Softwarearchitektur – MDA. Dazu wird ein Framework namens ATL genutzt, um, basierend auf Modellen, Meta-Modellen und Transformationen, Klassendiagramme im XMI-Format mit zuvor definierten Regeln in JUnit-Tests zu transformieren. Das Klassendiagramm, also das Modell, muss dazu einem zuvor hinterlegten UML2 Metamodell genügen.

Entwickler oder Architekten können damit ad hoc Tests auf den gerade erstellten Quelltext anwenden. Testergebnisse werden dabei farblich dargestellt, grün bei positiven und rot bei negativen Ergebnissen.

Zukünftig sollen weitere UML2-Diagramme, zum Beispiel Aktivitäts- und Interaktionsdiagramme, verarbeitet werden können. Zudem sollen OCL-Ausdrücke in die Design-Tests aufgenommen werden.

2.2 „On reverse engineering an object-oriented code into UML class diagrams incorporation extensible mechanisms“

Für die Rekonstruktion und Analyse eines Produktes ist es interessant herauszufinden, aus welchen statischen Aspekten eine Realisierung aufgebaut ist. Für diese Fragestellung haben die Autoren Vinita, A. Jain und D. Tayal (Vinita, et al., 2008) eine Lösung beschrieben, wie aus einem objekt-orientierten Quelltext ein UML Klassen-Diagramm hergeleitet werden kann. Zu diesem Ansatz gibt es schon vorherige Ausarbeitungen der Autoren, in jener liegt der Schwerpunkt jedoch darauf, viele der in UML Klassen-Diagrammen vorhandenen Elemente, z.B. abstrakte Klassen, einzubinden.

Hierzu stellen die Autoren einen Algorithmus vor, der diese Umwandlung vornimmt. Dieser Algorithmus benötigt als Eingabe den Quelltext und gibt als Ausgabe das dazugehörige UML-Klassendiagramm aus. Inhaltlich ist der Algorithmus in zehn Schritte mit insgesamt 15 Regeln unterteilt.

Einige für den Analysekontext relevante Regeln sind:

- **Regel 1**: Finde klassenbeschreibende Schlagwörter wie „class“ im Quelltext und erstelle dazu eine UML Klassen mit dem Namen, Attributen und Methoden.
- **Regel 2**: Finde Schlagwörter, die eine abstrakte Klasse beschreiben und erweitere die Klasse um das Attribut „abstract“.
- **Regel 6**: Finde gruppierende Elemente mit den Schlagwörtern „package“ oder Header-Dateien und zeichne diese entsprechend in das Diagramm ein.
- **Regel 10**: Finde Assoziationen, dabei wird geprüft, welche Klasse welche anderen Klassen nutzt (Assoziation), Verwenden sich zwei Klassen gegenseitig wird dies als bidirektionale Beziehung eingezeichnet.
- **Regel 11, 12**: Finde Aggregationen und Kompositionen, ähnlich wie Regel 10.
- **Regel 13**: Finde Vererbung, z.B. über das Schlagwort „implements“ oder über „:“.
- **Regel 15**: Identifiziere Erweiterungsmechanismen: Hier sollen Klassendiagrammerweiterungen wie „tagged values“ oder OCL-Constraints erkannt werden. Dazu muss für jede eingesetzte Sprache ein Konstrukt identifiziert werden, die solche Erweiterungsmechanismen darstellen.

Für jede zu unterstützende Sprache müssen die Regeln angepasst werden, da die zu identifizierenden Schlagwörter unterschiedlich sind, z.B. „package“ in Java und „namespace“ in .Net zur Identifizierung gruppierender Elemente.

Dieser Ansatz wurde in der Ausarbeitung geprüft, indem ein Beispielquelltext in ein Klassendiagramm überführt wurde. Hierzu ein Auszug aus der originalen Ausarbeitung:

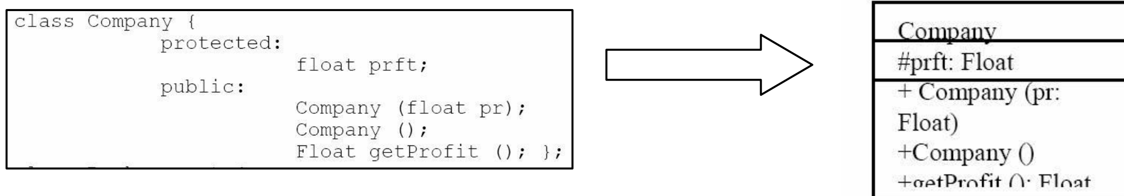

Abbildung 3: Beispiel der Anwendung der Klassendiagramm-Aufleitung

Aus Sicht der Autoren kann mit einem Algorithmus zunächst aus einer konkreten Implementation ein Modell hergestellt werden. Aus diesem Modell kann anschließend mit Hilfe von Code-Generierung eine andere Implementierungssprache gewählt und somit der Quelltext einer Programmiersprache in eine andere Programmiersprache überführt werden.

2.3 „Metamodel Approach on Model Conformance and Multiview Consistency Checking“

Verschiedene Modellierungssprachen wie BON oder UML werden in der MDA für die Spezifikation der Softwaresysteme genutzt. Die Autoren ChenShu, QuGuoQing und XioaJing (Shu, et al., 2008) befassen sich damit, wie die Konsistenz von Modellen formal beschrieben und geprüft werden kann. Ebenfalls untersuchen die Autoren die Möglichkeit, wie die dynamischen und statischen Sichten auf ein Szenario so verglichen werden können, dass diese miteinander kompatibel sind.

Aus der MDA ist der Einsatz von Meta-Modellen gebräuchlich, um die Struktur eines Modells, z.B. eines Komponentendiagramms, zu beschreiben. Als zu untersuchende Modellierungssprache wird BON genutzt. BON unterscheidet zwischen statischen Abstraktionen, z.B. Klassen, und statischen Beziehungen, z.B. Assoziationen. Die statischen Abstraktionen werden mittels logischer Constraints als Gruppen definiert.

$$\textbf{Unique_Feature_Name_Assertion} \triangleq (\forall c \in CLASS)((f_1 \in Class_Features\ (c) \wedge f_2 \in Class_Features\ (c) \Rightarrow Feature_Name(f_1) = Feature_Name(f_2)) \Rightarrow (f_1 = f_2))$$

Abbildung 4: Beispiel eines Contraint für die eindeutige Bezeichnung eines Features

Die obige Abbildung zeigt beispielhaft solch einen logischen Constraint, der sicherstellt, dass Namen für Klassen nur einmal für das gesamte System gewählt werden können. In dieser Form wird ein gesamtes Meta-Modell für einen Modellierungsaspekt erstellt. Das Ergebnis ist ein abstrakter Automat, mit dessen Hilfe Modelle aus dieser Kategorie gegen das erstellte Meta-Modell geprüft werden. Dieser abstrakte Automat wurde erstellt, um zu ermitteln, ob die definierten Regeln eingehalten werden und somit das Modell mit dem Ausgangsmodell konsistent ist. Für statische Beziehungen wird äquivalent ein weiterer Automat erstellt.

Ein weiterer Aspekt dieser Arbeit liegt darin, die dynamischen Sichten, bestehend aus Objekten, Nachrichten und Abläufen, mit der statischen Sicht zu vergleichen. Dabei wird durch die Autoren besonderer Wert auf die Konsistenz zwischen Klassen – Objekten und Nachrichten – Routinen gelegt. Dazu werden die bestehenden abstrakten Automaten der statischen Sicht um Constraints erweitert, die solche Prüfungen durchführen.

Folgende Abbildung zeigt eine derartige Erweiterung:

$$\text{Class_and_Object_Assertion} \triangleq \\ (\quad o \in \text{OBJECT})(o \in \text{union}(\text{CLASS})).$$

Abbildung 5: Erweiterung durch Constraints

Dieser Constraint stellt sicher, dass ein Objekt aus einer dynamischen Sicht auch in der statischen Sicht enthalten ist. Sollte dies nicht der Fall sein, ist entweder ein Objekt hinzugefügt oder nicht implementiert worden. Beide Fälle würde die Konsistenz zwischen den Sichten verletzten. Die Ausarbeitung zeigt zusätzlich viele weitere Constraints für die abstrakten Automaten auf.

Derzeit haben die Autoren nicht alle Aspekte, die mittels BON modelliert werden können, betrachtet. In zukünftigen Arbeiten sollen die fehlenden Aspekte realisiert werden.

2.4 „Automatic Code Generation: A Practical Approach“

George A. Papadopoulos (Papadopolous, 2008) zeigt in seiner Ausarbeitung, wie Techniken aus der MDA eingesetzt werden können, um aus UML-Modellen Quelltext für die Sprache „Manifold“ automatisiert zu erstellen. Dabei stellt der Autor verschiedene Werkzeuge vor, die für dieses Vorhaben genutzt werden können.

Um ein Diagramm zu erstellen, sollten lt. G. Papadopoulos zunächst die Objekte der höchsten Ebene identifiziert und modelliert werden. Für die Code-Generierung wird in diesem Schritt ebenfalls die Startmethode für die Realisierung am Modell definiert. Danach sollen für jede Komponenten die Operationen und Variablen gefunden und integriert werden.

Weitere Design-Vorgaben werden in der Ausarbeitung ausführlich dargestellt. Um nun von Modellen zu Quelltexten zu kommen, werden sog. Modell-Transformationen benötigt. Dies sind Regeln, die beschreiben, wie ein Diagramm als Quelltext dargestellt werden soll. Diese Regeln müssen pro Diagrammtyp und Programmiersprache definiert werden, sodass für ein breites Spektrum an nutzbaren Modellen/Sprachen ein großer Aufwand entsteht. Aus der Ausarbeitung geht hervor, dass solche Regeln nicht automatisiert, sondern manuell erstellt werden müssen.

Folgende Abbildung zeigt als Übersicht, wie solch eine Transformation in diesem Beispiel vorgenommen werden kann:

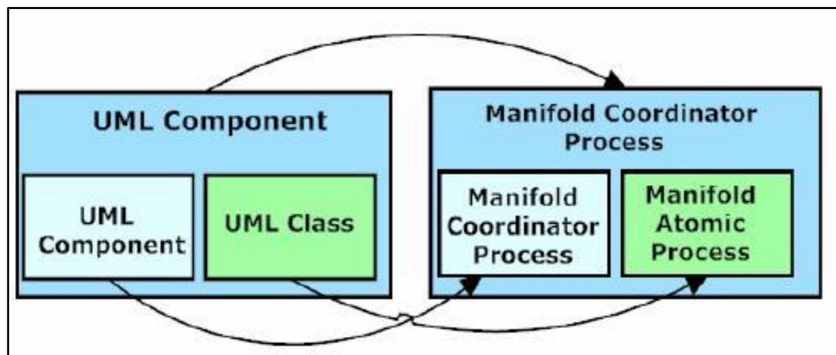

Abbildung 6: Übersicht einer Transformation

Des Weiteren stellt G. Papadopoulos vor, wie auch Verhaltensdiagramme, z.B. Sequenzdiagramme, zur Code-Generierung eingesetzt werden können. Dazu stellt der Autor, wie z.B. bei den statischen Modellen, Design-Vorgaben für die Modellierung vor, bei deren Einhaltung ebenfalls Transformationsregeln für die Konversion erstellt werden können.

Als Eingabe für diesen Beweis der Machbarkeit werden die Modelle im XMI-Format unterstützt. Diese werden dann zunächst in ein Manifold Metamodel und daraus dann in Manifold Quelltext überführt. Durch den Einsatz von XMI als Eingabeformat ist die Erstellung der Modelle fast werkzeugunabhängig, solange das jeweilige Werkzeug einen XMI-Export in derselben Version von XMI anbietet. XMI basiert auf einer XML-Struktur und kann somit mit bekannten XML-Werkzeugen und –Methoden, wie XPath o.ä., bearbeitet werden.

Weiterführend sollen die enthaltenden Methoden so erweitert werden, dass auch eine Konsistenzprüfung und eine Validierung des eingegebenen Modells vorgenommen wird. In dieser Ausarbeitung wird das eingegebene Modell nicht auf Korrektheit oder Schwächen überprüft, sondern als korrekt und Basis der weiteren Schritte vorgesehen.

2.5 Abgrenzung

In den Ausarbeitungen von (Papadopolous, 2008) und (Pires, et al., 2008) werden die Modelle im XMI-Format importiert. XMI basiert auf XML-Strukturen und kann mit entsprechenden XML-Methoden verarbeitet werden. Dies ist für die Architekturanalyse eine Vereinfachung, da ein standardisiertes Format eine einfach zu realisierende und wiederverwendbare Lösung für die automatisierte Bearbeitung von Diagrammen darstellt. Farbliche Akzente, Vergleiche und auch die automatisierte Erstellung können über Manipulation der XMI-Daten geschehen, und es ist nicht mehr nötig, werkzeugabhängige Implementationen, z.B. Interop Assembly (Microsoft, 2011) bei Microsofts Visio, anzubieten, um die Integration in Modellierungswerkzeuge zu ermöglichen. Neben XMI können auch Werkzeuge unterstützt werden, die XML als zugrundeliegende Datenstruktur für das Speicherformat nutzten, da hier dieselben Kriterien der Bearbeitung wie bei XMI gelten, nur dass an dieser Stelle die anbieterabhängigen Formate eingehalten werden müssen.

Die in der Ausarbeitung von Pires, et al. (Pires, et al., 2008) vorgestellte Methode stellt eine automatisierte modellbasierte Architekturanalyse dar. Derzeit umfasst die dargestellte Arbeit eine Lösung für Klassendiagramme und Java-Implementationen. Weitere Modelle zu unterstützen ist in nachfolgenden Arbeiten geplant. Pires, et al. zeigen, wie man die Transformation vom Modell hin zu einem speziellen Quelltext realisieren kann. Hierbei handelt es sich im Gegensatz zu den Zielen dieser Arbeit um Unit-Tests, die auf die Ist-Architektur angewandt werden.
Dieses Vorgehen bietet eine weitere Möglichkeit, Soll- mit Ist-Architekturen zu vergleichen, entspricht aber nicht den Zielen dieser Arbeit, in der Modelle in Form von Diagrammen direkt mit dem Quelltext oder dessen Repräsentanten verglichen werden sollen. Der Unterschied ist deutlich bei der Art des Vergleichens der Architektur und der Repräsentation der Ergebnisse sichtbar.

Bei Pires et al. übernimmt das JUnit-Framework die Visualisierung der Ergebnisse, grün bei Erfolg und rot sobald ein Fehlerfall auftritt. Diese Ergebnisse sind eher an Entwickler adressiert, ebenso die Erstellung der Design-Tests.

Die Motivation dieser Arbeit beinhaltet neben dem eigentlichen Vergleichen die Ermöglichung des Einsatzes des Ergebnisdiagramms als Kommunikationsmedium über die verschiedenen Rollen eines Softwareerstellungsprojekts hinweg. Daher ist es wichtig, Mechanismen für den Export akzentuierter Diagramme als Ergebnis anzubieten. Die Anpassung der Daten, sodass die genannten Akzente gesetzt werden, ist im Einzelnen noch zu realisieren.

Vinita, et al. stellen einen Algorithmus vor, der genau eine Lösung auf die Frage: „ Wie generiert man aus dem Quelltext ein Diagramm?“ für Klassendiagramme beantwortet. Die Struktur des Algorithmus ist so ausgelegt, dass eine Erweiterung und Anpassung für weitere Modelltypen, z.B. Komponentendiagramme denkbar ist. Komponentendiagramme stellen eine verbreitete Modellierungsdarstellung für Architekturen dar, sodass der Schnitt der Komponenten und deren Beziehungen zueinander vergleichsweise einfach zu realisieren ist. In den nächsten Schritten dieser Ausarbeitung soll validiert werden, ob eine Änderung des Algorithmus hin zu Komponentendiagrammen möglich ist. Dieser Algorithmus stellt eine Transformationsregel für einen gezielten Diagrammtyp dar. Solche Transformationsregeln zwischen Modellen eines Softwaresystems finden sich auch in dem Ansatz der modellgetriebenen Software Architektur (Petrasch, et al., 2006). Aufgrund des gemeinsamen Problemfelds dieser Arbeit und des MDA-Ansatzes, und zwar die automatisierte Transformation von Modellen, wird sich an den Mechanismen der MDA orientiert. Diese Mechanismen werden auch in den Ausarbeitungen (Papadopolous, 2008), (Pires, et al., 2008) und (Vinita, et al., 2008) genutzt. Alle drei Ausarbeitungen sind aus dem Jahr 2008 und zeigen, wie der MDA-Ansatz für die Softwareentwicklung/Softwarearchitekturanalyse genutzt werden kann. In allen Ausarbeitungen wird deutlich, dass die Transformationen zwischen Modellen (z.B. Quelltext und Klassendiagramm) manuell realisiert werden müssen.

Noch offen ist die Frage, wie man Modelle miteinander vergleicht, um Differenzen zu erkennen. Die Ausarbeitung von ChenShu (Shu, et al., 2008) gibt für einen spezialisierten Kontext eine Möglichkeit dafür an. Die Prämissen von ChenShu, dass alle Modelle formal geprüft werden müssen, gelten nicht für diese Ausarbeitung, da der Formalismus vor

allem darauf zielt, dass die erstellten Diagramme einem gewählten Standard genügen. In dieser Ausarbeitung zur modellbasierten Softwarearchitekturanalyse gelten die eingegebenen Soll-Architektur-Diagramme als korrekt, und das Ziel ist herauszufinden, ob die Realisierung (Ist-Architektur) diesem Diagramm genügt. Ein formales Vorgehen gibt dem Ersteller einen engen Rahmen, der Ersteller könnte bewusst gegen formale Anforderungen verstoßen, um die Aussagekraft eines Diagramms zu erhöhen. Bis zu einem gewissen Grad soll für die modellbasierte Softwarearchitekturanalyse diese Freiheit gewahrt bleiben. Vorbedingungen sind jedoch, dass nur Elemente in einem Modell aus einem Meta-Modell genutzt und nicht Elemente verschiedener Diagrammtypen, wie z.B. Klassen- und Aktivitätsdiagramme, gemischt werden können. Dies kann technisch wegen der manuell zu erstellenden Transformationsregeln nicht gewährleistet werden. Des Weiteren nutzt ChenShu keine automatisierten Techniken zur Validierung, da solche Methoden benötigt werden, um Ergebnisse der Analyse generieren zu können.

3 Allgemeine Grundlagen

In diesem Kapitel werden die Grundlagen, die nicht rein technischer Natur sind und in dieser Ausarbeitung als Voraussetzung genutzt wurden, dargestellt.

3.1 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. Die in dieser Arbeit verwendete Definition einer Softwarearchitektur bezieht sich auf folgende Quelle von Bass, Clements und Kazman (Bass, et al., 2003):

> *Software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them*

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. Diese eingeführten Sichten wurden von Reussner und Hasselbring (Reussner, et al., 2006) beschrieben.

3.1.1 Soll- und Ist-Architektur

Bei der Betrachtung der Architektur von Software ist zwischen der Soll- und der Ist-Architektur zu unterscheiden (vgl. Abbildung 1). Dabei ist die Soll-Architektur die Beschreibung des Systems an sich inklusive der Beschreibung von Strukturen und Abhängigkeiten zueinander. Die Erstellung solch einer Soll-Architektur ist ein Bearbeitungsschritt in der Erstellung von Software-Systemen. Hierzu sollte die Zusammenarbeit der Software-Architekten und Entwickler genutzt werden. Die Ist-Architektur beschreibt die realisierte Architektur, oder in anderen Worten die Sicht auf die Umsetzung der Soll-Architektur. Dabei kann von der Soll-Architektur abgewichen werden, deren Gründe vielfältig sind. Zum Beispiel kann ein Punkt aus der Soll-Architektur anders ausgelegt oder die Realisierung nicht direkt möglich gewesen sein. Die

Repräsentation der Ist-Architektur kann zum einen der Quelltext und zum anderen auch ein Softwaresystem an sich sein, das programmatisch bearbeitet werden kann.

Die Darstellung von Soll-Architekturen kann in graphischer Form, z.B. via UML aber auch textuell, als Realisierung einer Referenzarchitektur (z.B. QUASAR) dargestellt werden. Jedoch weisen diese Darstellungsmöglichkeiten meist eine Abstraktion auf. Diese Abstraktion und Strukturierung wird nicht in allen Programmiersprachen direkt unterstützt, sodass es zu einem Bruch in der Abbildung und dem Detaillierungsgrad zwischen Soll- und Ist-Architektur kommt. So sind einige Komponentenmodell-Eigenschaften nicht exakt in der Entwicklungssprache Java oder C# umsetzbar. Ist nun eine Architektur mittels Komponentenmetaphern im Soll modelliert, fehlen hier Informationen, wie ein Vergleich gemacht werden kann. Somit ist es nötig, die Ist-Architektur in der Definition als Realisierung der Soll-Architektur, z.B. in Form von Quelltext, zu erweitern. Ein Mapping zwischen den Soll-Architektur-Strukturen und den Ist-Architektur-Strukturen zu erstellen oder zu ermitteln ist nötig. Somit ist in dieser Arbeit eine Ist-Architektur als das zu untersuchende Softwaresystem zuzüglich der Abbildungsregeln auf die Soll-Architektur definiert.

3.1.2 Referenzarchitekturen

Eine Referenzarchitektur beschreibt eine Kategorisierung von Architekturen, die auf denselben Sachverhalt zurückzuführen sind. Hierbei werden grundlegende Komponenten und deren Verbindungen untereinander definiert. Ähnlich wie bei einem Baukastensystem wird solch eine Referenzarchitektur als Blaupause für das zu entwickelnde Softwaresystem genutzt. Ein bekannter Vertreter solch einer Referenzarchitektur ist z.B. die Schichtenarchitektur. Der Werkzeug- & Materialansatz (Züllighoven, et al., 1998) kann ebenfalls als eine Referenzarchitektur angesehen werden. Neben den reinen modellierungsrelevanten Informationen können weitere Definitionen innerhalb einer Referenzarchitektur angewandt werden, z.B. dass eine bestimmte Komponente oder Schicht nicht veränderbar oder serialisierbar sein muss. Solche textuellen semantischen Ergänzungen können, je nachdem, ob ein Modellierungswerkzeug solche Informationen bereitstellt, modelliert oder als Text angefügt sein.

3.2 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 Funktionalitäten programmiert wurden, die nicht mit der Soll-Architektur übereinstimmen. Als weiteres Beispiel kann aufgeführt werden, dass das Verständnis des Entwicklerteams für die Soll-Architektur nicht klar vorhanden ist, sodass Korrekturen und Erweiterungen erstellt werden, die funktionieren, die aber nicht die Architektur berücksichtigen. Somit ist eine kontinuierliche Analyse der Architektur nötig, um sowohl die Architektur einzuhalten, aber auch um 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 (hellow2morrow, 2011). Die Ergebnisse dieser Werkzeuge müssen vom Anwender interpretiert und manuell 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

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 jedoch der visuelle Vergleich als Kommunikationsmittel mit dem Entwicklerteam im Vordergrund stehen. Daher ist es nicht das primäre Ziel, eine metrikbasierte Analyse durchzuführen, sondern die Einhaltung der Soll-Architektur in dem zu untersuchenden Softwaresystem steht im Fokus dieser Arbeit.

3.3 Modellierung

Die Modellierung von Problemen ist eine häufig eingesetzte Technik, um mittels festgelegter Notationen Problemfelder zu skizzieren. Nach Stachowiak (Stachowiak, 1973) ist ein Modell durch die folgenden drei Charakteristika gekennzeichnet:

- Abbildung: Ein Modell ist immer eine Repräsentation eines natürlichen oder künstlichen Originals
- Verkürzung: Ein Modell umfasst nicht alle Eigenschaften des Originals sondern nur diejenigen, die dem Modellerschaffer als relevant erscheinen
- Pragmatismus: Modelle können für bestimmte Einsatzkontexte aufgrund der vorliegenden Eigenschaften geeigneter sein als das Original

Diese Eigenschaften gelten entsprechend auch für die Modellierung von Softwaresystemen. Dabei werden je nach Einsatzkontext nur die relevanten Eigenschaften des Originals (fachlicher Sachverhalt, Problemdefinition, Anforderungen an die Software) betrachtet. Oftmals ist der Einsatz eines Modells dann erforderlich, wenn der zu untersuchende Sachverhalt zu komplex für eine umfassende Anforderung ist. Solch eine Notation kann sowohl in grafischer als auch textueller Form vorliegen (Kecher, 2005).

3.3.1 Unified Modeling Language

Die Unified Modeling Language (Group, 2011), kurz UML, ist eine spezifizierte Modellierungsgrundlage für die Modellierung von Applikationen, Verhalten, Architekturen und auch von Abläufen und Datenstrukturen. Die UML wird fortlaufend von der Object Management Group (OMG, 2011) entwickelt und liegt derzeit in der Version 2.3 vor.

Bezug nehmend auf die vier Sichten der Softwarearchitektur, werden durch die UML beispielsweise folgende spezifizierte Diagrammdefinitionen angeboten:

- Statische Sicht: → Klassen-, Komponenten- und Paketdiagramme
- Fachliche Sicht: → Sequenz- Aktivitäts- und Anwendungsfalldiagramme
- Verteilungssicht: → Verteilungsdiagramme
- Laufzeitsicht: → Sequenz- und Interaktionsdiagramme

Neben der reinen Spezifikation gibt es eine breite Auswahl an Werkzeugen, die diese Spezifikation umgesetzt haben, z.B. ArgoUML, Visual Paradigm oder Star UML.

3.3.1.1 Komponentendiagramme

Komponentendiagramme sind UML standardisierte Diagramme, die unter die Kategorie der Strukturdiagramme fallen. Mit Komponentendiagrammen ist es möglich, die Subsysteme und Teile einer Softwarearchitektur zu modellieren. Hierbei unterscheidet man zwischen der inneren und äußeren Sicht einer Komponente. Die äußere Sicht beinhaltet die benötigten und angebotenen Schnittstellen, allgemeine Verbindungen zu anderen Komponenten und Vererbungsbeziehungen. Der Zusammenhalt und die Verbindungen zwischen Komponenten beruhen auf der Kommunikation zwischen den angebotenen und benötigten Schnittstellen (siehe Abbildung 7). Da die Kommunikation im vereinfachten Fall ausschließlich über Schnittstellen stattfindet und Komponenten in der äußeren Sicht als Black Box zu verstehen sind, stellt die innere Sicht den genaueren Aufbau der Komponente dar. Hierbei werden vorwiegend Notationselemente wie Klassen (vgl. Klassendiagramme) und Parts (Kompositionsdiagramm) verwendet. Somit handelt es sich bei Komponentendiagrammen eigentlich um zwei Diagramme in einem, weil sowohl die äußere, als auch innere Sicht entscheidend für die Aussagekraft der Modelle und auch für die Realisierung ist (Kecher, 2005).

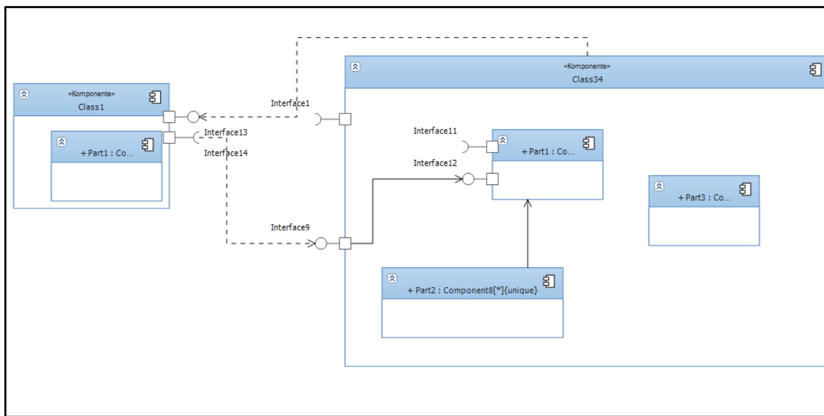

Abbildung 7: Beispiel für ein Komponentendiagramm

3.3.1.2 Klassendiagramme

Klassendiagramme sind genau wie Komponentendiagramme ein Diagrammtyp der UML aus der Familie der Strukturdiagramme. Klassendiagramme ermöglichen die direkte Modellierung einzelner Klassen und deren Verbindungen zueinander. Somit werden hier, wie in den meisten objektorientierten Programmiersprachen, Attribute, Methoden und der Name einer Klasse definiert. Darüber hinaus ist es möglich, Attribute (oder Felder) mit den entsprechenden Datentypen (auch Listen) zu definieren. Methoden können sowohl mit einem Rückgabewert als auch mit verschiedenen Parametern dargestellt werden.

Dazu können Klassen in abstrakte Klassen, „normale“ Klassen und Schnittstellen unterschieden werden.

Vererbungs- und Realisierungshierarchien werden unterstützt und sind direkt abzubilden. Neben der Beschreibung einzelner beschreibender Elemente (Klassen, Schnittstellen etc.) stehen die Verbindungen zwischen den Elementen im Mittelpunkt der Klassendiagramme. Zusätzlich zu Realisierungs- und Vererbungsverbindungen sind auch Benutzt- und Enthält-Verbindungen möglich. Eine Benutzung wird als Assoziation bezeichnet. Enthält-Verbindungen beschreiben die Aggregation und Komposition (lebensbindend). Das hieraus entstehende Geflecht an Verbindungen beschreibt sehr konkret, welche Elemente mit welchen anderen Elementen in welchem Zusammenhang stehen.

Da der Grad an Informationen sowohl über die Definition von Namen der Entitäten, Attribute und Methoden inklusive der Typen, als auch der Verbindungen unter- und zueinander sehr hoch ist, haben sich verschiedene Fokussierungen im Einsatz von Klassendiagrammen herausgebildet:

- Domain Specific View: nur die Namen der Entitäten und die Verbindungen untereinander werden modelliert
- Statische oder fachliche Sicht: nur die Namen und Attribute als auch die Verbindungen untereinander werden modelliert

- Klassische Sicht: Namen, Attribute und die kompletten Methoden als auch die Verbindungen untereinander werden modelliert

Je nach Wahl der jeweiligen Sicht auf die Klassendiagramme ändert sich somit auch der Detailgrad des Modells.

3.3.2 Andere Modellierungsvarianten

Neben der UML gibt es weitere Modellierungsverfahren, wie z.B. eGPM (exemplarische Geschäftsprozessmodellierung), BPMN (Business Process Modeling Notation), Ereignisgesteuerte Prozessketten (EPK) oder die Schichtenansicht. Dabei werden meist Geschäftsprozesse, also die dynamische Sicht einer Architektur, behandelt. Die statische Sicht wird durch die UML weitestgehend abgedeckt. Schichtenarchitekturen können ebenfalls als Sub- oder Komponentensystem, aber auch durch einfache Zeichenwerkzeuge erstellt werden. Sie sind nicht in der UML enthalten (Fähnrich, 2007).

3.3.3 Meta Object Facility / Metamodellierung

Die Meta Object Facility (MOF) ist ein durch die Object Management Group (OMG, 2011) standardisiertes Meta-Metamodell, um z.B. Modelle der UML zu definieren. Es hat das Ziel, eine Modellierungssprache für Modellierungssprachen zu definieren. Aus dem Ansatz der Model Driven Architecture (Petrasch, et al., 2006) geht hervor, dass es verschiedene Stufen von Modellen gibt:

- **M0**: konkrete Implementation bzw. Instanzen von M1 Modellen, z.B. der Quelltext einer Software
- **M1**: Modelle, die M0-Modelle beschreiben und aus Gebilden der M2-Modellierung bestehen, z.B. ein konkretes Klassendiagramm für ein Softwaresystem
- **M2**: Metamodelle, die Bausteine für einen Modellierungstypen der M1-Modelle bereitstellen
- **M3**: Meta-Metamodelle, die Bausteine für die Definition einer Modellierungssprache bereitstellen

Dabei ist die MOF ein M3-Modell für verschiedene Modellierungstypen (M2-Modelle). In der MDA ist ein Übergang von einer Modellstufe in eine andere durch sog. Transformationen möglich. So ist ein Quelltextgenerator ein Transformator, um z.B. Klassendiagramme (M1) in Quelltextgerüste (M0) zu überführen. Eine Überführung von M0-Modellen nach M1-Modellen wird in (Vinita, et al., 2008) (vgl. Kapitel 2.2) exemplarisch dargestellt.

4 Technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen beschrieben, die zum Verständnis dieser Arbeit benötigt werden.

4.1 Programmiersprache C#

C# ist eine Programmiersprache, welche von Microsoft in das DotNet Framework integriert ist (Microsoft, 2011). C# wurde eigens für das DotNet Framework als eine Ableitung von Java entwickelt. Sie zählt zu den objektorientierten Sprachen, die eine strenge Typisierung in Form von nativen Datentypen enthält. Im Gegensatz zu C++ ist in C# keine Zeigersemantik enthalten. Wie alle DotNet Programmiersprachen erfolgt die Kompilierung des Quelltextes nicht direkt in ausführbaren Maschinencode, sondern in einer Zwischenschicht namens Common Intermediate Language (CIL, 2005). Diese Common Intermediate Language (CIL) wird durch das DotNet Framework zu Laufzeit interpretiert und ausgeführt.

Attribute, wie sie in Klassendiagrammen genutzt werden, werden in C# durch Felder und Properties realisiert. Properties enthalten Felder, bieten aber erweiterte Validierungs- und Eingriffsmöglichkeiten, wie den nur lesenden Zugriff.

Methoden bestehen, wie auch in anderen Programmiersprachen, aus einem Bezeichner, einem Rückgabewert und einer Signatur inklusive Parameter. Dies alles zusammen beschreibt den Methodenkopf. Das Verhalten und der Ablauf innerhalb einer Methode bezeichnen den Methodenrumpf und sie enthalten die konkrete Implementierung.

C# unterstützt die Einfachvererbung von einer Basisklasse. Dazu kann das Verhalten mehrerer Schnittstellen zusätzlich über die Realisierungsbeziehung implementiert werden.

4.1.1 Reflections

Es besteht eine API in dem DotNet Framework, die es dem Anwender erlaubt, ein vorkompiliertes Softwaremodul (Assembly), das eine ausführbare Datei oder eine Bibliothek sein kann, zu untersuchen. Dabei ist es möglich, die enthaltenen und instanziierten Klassen (Typen) einzeln auszulesen. Somit werden Informationen wie der Name und andere Eigenschaften, z.B. ob es sich um eine Schnittstelle oder statische Klasse handelt, verfügbar. Des Weiteren ist es möglich, die Typinformationen der Felder und Properties, als auch Methoden (inklusive Konstruktoren) zu erhalten. Hierbei ist es möglich, Informationen über den Namen, die Multiplizität (ein oder mehrere Objekte) oder die Rückgabewerte etc. abzufragen. Die Hierarchie ist rekursiv aufgebaut, sodass z.B. bei der Abfrage aller Parameter einer Methodensignatur der Aufbau der Informationen gleich den Informationen des Basistyps und somit generisch analysierbar ist.

Es ist nicht möglich, über die Reflections API Informationen über den Zeitpunkt und die Art der Initialisierung einer Variablen zu erhalten. So ist unklar, ob der Typ A innerhalb

einer Klasse neu gebaut oder durch einen Parameter vom Typ A aus der dazugehörigen Methodensignatur überschrieben wurde. Ebenfalls sind Informationen über den Konstruktor-Aufruf, die wie Methoden zu behandeln sind, nicht ersichtlich.

4.1.2 Common Intermediate Language (CIL)

Die Common Intermediate Language (CIL, 2005) ist eine Zwischensprache, in der alle Programmiersprachen des DotNet Framework kompiliert werden. Dies ermöglicht dem Entwickler, die Programmiersprache seiner Wahl aus dem .Net-Framework zu nehmen und dennoch die Kompatibilität zu anderen Programmiersprachen aus dieser Familie zu wahren. Die CIL von Methoden ist über die Reflections API zugreifbar. Hier ist sie als Byte-Code abgelegt. Mittels eines von Microsoft beschriebenen Verfahrens ist es möglich, diese CIL in dem Programmablauf als nicht binäre Bezeichner zugreifbar zu machen. Diese Überführung ist ähnlich wie Assembler aufgebaut und somit über einfache Parser auswertbar. Im Gegensatz zur Reflections API sind hier Informationen über die Initialisierung einer Variablen sichtbar, ebenso welcher Konstruktor mit welchen Typen aufgerufen wurde. Dieser Mechanismus ist wichtig für die Auflösung von Verbindungen zwischen Klassen.

Hier ein Auszug eines Methodenrumpfes in CIL:

```
[0]  "0000 : ldarg.0"
[1]  "0001 : ldnull"
[2]  "0002 : stfld System.ComponentModel.RegistrierungsWerkzeug::components"
[3]  "0007 : ldarg.0"
[4]  "0008 : call instance void System.Windows.Forms.UserControl::.ctor()"
[5]  "0013 : nop"
[6]  "0014 : nop"
[7]  "0015 : ldarg.0"
[8]  "0016 : call instance System.Void.RegistrierungsWerkzeug::InitializeComponent()"
[9]  "0021 : nop"
[10]"0022 : nop"
[11]"0023 : ret"
```

4.2 XMI- Austauschformat

XML Metadata Interchange, kurz XMI (OMG-XMI, 2011), beschreibt ein Austauschformat, mit dessen Hilfe alle MOF (Meta Object Factory)-Konformen Metadaten zwischen Werkzeugen ausgetauscht werden können. Metadaten beschreiben in diesem Kontext die strukturellen Elemente eines Modells, z.B. die Attribute oder Beziehungen in einem Klassendiagramm. Dieses Verfahren wird besonders im Bereich der Modellierung eingesetzt. Es erlaubt, Modelle in ein XML-Format zu überführen und in einem Werkzeug, welches den Im- und Export von XMI unterstützt, wieder einzulesen. Bis zur Version 1.4 war es nur möglich, strukturelle Daten ohne deren graphische Repräsentation zu übertragen. In der aktuellen Version 2.1 wird diese graphische Repräsentation auch unterstützt und übertragen.

Die Implementierung von XMI in Werkzeugen, welche eigentlich einem Standard der OMG zugrunde liegen, ist jedoch nicht einheitlich. Die Unterschiede zwischen Version 1.4

und 2.1 sind neben der graphischen Repräsentation auch in der XML-Struktur der Metadaten sehr groß, sodass eine Abwärtskompatibilität nicht vorliegt.

Es ist möglich, XMI programmatisch zu untersuchen und auszuwerten, indem bekannte XML-Operationen auf die jeweilige Datei ausgeführt werden. Mittels XMI ist ein automatisiertes Einlesen und Herausschreiben von Diagrammen möglich.

Hier ein Beispiel für die Struktur von XMI 1.4:

```
<?xml version="1.0"?>
<XMI xmi.version="1.2" xmlns:UML="org.omg/UML/1.4">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>ananas.org stylesheet</XMI.exporter>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
  </XMI.header>
  <XMI.content>
    <UML:Model xmi.id="M.1" name="address" visibility="public"
                isSpecification="false" isRoot="false"
                isLeaf="false" isAbstract="false">
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id="C.1" name="address" visibility="public"
                   isSpecification="false" namespace="M.1" isRoot="true"
                   isLeaf="true" isAbstract="false" isActive="false">
          <UML:Classifier.feature>
            <UML:Attribute xmi.id="A.1" name="name" visibility="private"
                           isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.2" name="street" visibility="private"
                           isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.6" name="country" visibility="private"
                           isSpecification="false" ownerScope="instance"/>
          </UML:Classifier.feature>
        </UML:Class>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```

Und ein Beispiel für die Struktur von XMI 2.1:

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
  <uml:Model xmi:type="uml:Model" xmi:id="themodel" name="TemplateDesigner">
      <ownedMember xmi:type="uml:Package" xmi:id="BOUML_0x313d30"
      name="TemplateDesigner">
        <ownedMember xmi:type="uml:Class" name="Templates"
xmi:id="BOUML_0x32e090"        visibility="package" isAbstract="false" >
          <ownedAttribute xmi:type="uml:Property" name="Name"
          xmi:id="BOUML_0x34a160" visibility="protected">
            <type xmi:idref="BOUML_datatype_0"/>
          </ownedAttribute>
      </ownedMember>
    </ownedMember>
  </uml:Model>
</xmi:XMI>
```

4.3 Modellierungswerkzeuge

In diesem Abschnitt werden einige der in dieser Arbeit genutzten Modellierungswerkzeuge vorgestellt.

ArgoUML (Tigris, 2011) ist ein auf Java basierendes Open Source UML-Modellierungswerkzeug, welches den XMI und UML Standard in der Version 1.4 implementiert. Es ist ein relativ weit verbreitetes UML-Werkzeug, das die jeweiligen Diagramme eines Projektes in einer einzelnen Datei speichert. Diese Datei ist XML-basiert und somit programmatisch auswertbar. Aus diesen Gründen wurde ArgoUML trotz der älteren UML und XMI Implementierung auch in dieser Arbeit betrachtet.

Visual Paradigm (Paradigm, 2011) ist ein kommerzielles Produkt, das unter anderem die Modellierung von UML in den Versionen 2.1 (und niedriger) unterstützt. Ebenfalls werden die XMI Standards 1.4 bis 2.1 für den Ex- und Import implementiert. Die Diagramme eines Projekts werden hier ebenfalls in einer einzelnen Datei gespeichert, die XML-lesbar ist. Es werden alle von der OMG spezifizierten UML Modelle unterstützt. Darüber hinaus werden auch noch andere Modelle, z.B. Business Process Modeling Notation (BPMN) angeboten. Visual Paradigm ist nicht nur auf UML ausgelegt, es bietet aber ein einheitliches Benutzungsmodell und kann für vielerlei Modellierungstechniken angewandt werden. Darauf beruht auch die Verbreitung dieses Werkzeugs.

Visual Studio 2010 (MSDN, 2011), das eigentlich eine Entwicklungsumgebung für das .Net Framework von Microsoft ist, unterstützt in der kommerziellen Architecture Edition ebenfalls die Modellierung von einzelnen UML-Modellen sowie die Modellierung der Schichtenreferenzarchitektur. Im Gegensatz zu den oben beschriebenen Werkzeugen, unterstützt Microsoft nicht exakt die XMI-Standards. Die Modellierung von Klassen- und Komponentendiagrammen, als auch Anwendungsfalldiagrammen wird unterstützt. Dieses Werkzeug ist für diese Arbeit insofern interessant, als dass eine Integration in die Entwicklungsumgebung der zu untersuchenden Programmiersprache C# besteht. Es ist kein Einsatz weiterer Werkzeuge nötig. Visual Studio schreibt die Modelle pro Projekt ebenfalls in eine einzelne XML-Datei. Es ist möglich, diese Dateien programmatisch zu untersuchen. Jedoch ist ein Import von den Modellen in der derzeitigen Version von Visual Studio nicht vorgesehen, sondern erst in nachfolgenden Versionen.

Die hier vorgestellten Werkzeuge wurden vor allem danach ausgesucht, ob die generierten Dateien in einem XML-konformen Format vorliegen, damit das Einlesen der Modelle auch programmiertechnisch gelöst werden kann. XMI stellt dabei ein standardisiertes Verfahren bereit, jedoch ist mit ein wenig mehr Aufwand auch das Einlesen von XML direkt möglich. Es gibt weit mehr Werkzeuge, die sowohl UML-spezifiziertsind, als auch weitere Modellierungstechniken unterstützen, jedoch ist in diesem Kontext der Arbeit ein Modellierungswerkzeug ausgewählt worden, welches sowohl ein verbreitetes Open Source-Werkzeug (ArgoUML), als auch ein kommerzielles (Visual Paradigm) und ein in die Entwicklungsumgebung integriertes (Visual Studio) Werkzeug ist.

5 Problemstellung und Anforderungen

In diesem Kapitel wird die Problemstellung aus Kapitel 1.1 und 1.2 detailliert, und auch die Anforderungen an das weitere Vorgehen definiert.

5.1 Problemstellung

Die grundlegende Problemstellung dieser Arbeit ist, wie Diagramme direkt mit dem Quelltext verglichen werden können, bzw. in wie weit dies möglich ist, um eine modellgetriebene Softwarearchitekturanalyse zu ermöglichen. Vereinfacht formuliert ist ein Vergleich einer Soll- mit einer Ist-Architektur angestrebt. Repräsentanten für die Soll-Architektur sind in dieser Arbeit Diagramme, z.B. aus der UML. Ein Repräsentant für die Ist-Architektur ist z.B. der Quelltext eines Softwaresystems. Die folgenden Ansätze und Entwürfe sollen dazu prototypisch umgesetzt und anschließend die Möglichkeiten und Grenzen der Ansätze untersucht werden. Die Recherche zu diesem Thema hat kaum direkte bzw. umfassende Erfolge zu dem gesamten Thema erzielt, es wurden jedoch Quellen gefunden, die Teilaspekte für ein solches Vorgehen aufzeigen, z.B. die Generierung von Quelltext aus einem Klassendiagramm. Solche Transformationen sind vor allem in der Model Driven Architecture (Stahl, et al., 2007) üblich. Hier gilt der Quelltext ebenfalls als Modell, und es werden theoretisch Mechanismen gefordert, die Transformationen von einem in ein anderes Modell ermöglichen sollen. Solche konkreten Mechanismen stellen z.B. Quelltextgeneratoren dar. Diese Arbeit hat nicht das Ziel, aus beliebigen Modellen Quelltext zu generieren, sondern einen Vergleich auf der Architekturebene anzustreben. Hierzu sind aus den genannten Quellen, vor allem Vinita, et al. (Vinita, et al., 2008) angewandten Methoden interessant, die ein Mapping von den Modellen auf den Quelltext beschreiben, was zusammen lt. Definition die Ist-Architektur beschreibt (vgl. Kapitel 3.1.1).

Basierend auf der allg. Problemstellung, ob eine Architekturanalyse mittels Diagrammen direkt mit dem Quelltext einer Software möglich ist, ergeben sich weiterführende Problemstellungen:

Vergleichbarkeit: Grundlegend muss untersucht werden, inwiefern Quelltexte mit Diagrammen verglichen werden können. Diagramme zu einer Software haben zwar einen direkten Bezug, basieren jedoch auf einer nötigen Abstraktion, um gewisse Sachverhalte darstellen zu können. Bei Betrachtung der vorliegenden Formate und der Informationsabstraktion in den Diagrammen scheint ein direkter Vergleich ausgeschlossen. Des Weiteren soll ein Vergleich schnell und automatisiert für verschiedene Softwarerealisierungen und die dazugehörigen Diagramme geschehen können. Ebenfalls sollen die Vergleichsergebnisse nachvollziehbar und korrekt sein. In der MDA stellt ein Quelltext eines Softwaresystems selbst ein Modell der Software dar, ein sog. M0 Modell. Ein Diagramm stellt wiederum eine Abstraktion des Quelltextes, ein sog. M1 Modell dar. Um nun eine nachvollziehbare Vergleichbarkeit herzustellen, wird in dieser Arbeit der Ansatz verfolgt, dass die Repräsentation des Quelltextes und des Diagramms auf derselben Modellierungsebene vorliegen. Es soll somit untersucht

werden, auf welcher Modellierungsebene ein Vergleich einfach und effizient gestaltet werden kann. Dazu werden ebenfalls Ansätze benötigt, wie eine Transformation des Quelltextes und/oder des Diagramms auf die beschriebene Modellierungsstufe realisiert werden kann.

Abbildbarkeit: Neben den Mechanismen für die Vergleichbarkeit ist laut der Definition einer Ist-Architektur (vgl. Kapitel 3.1.1) eine Abbildung der Quelltextelemente auf die Elemente der Soll-Architektur nötig. Dies liegt darin begründet, dass Quelltext eine sehr explizite und realisierungsnahe Notation und Semantik beinhaltet. Diagramme sind laut Definition eine abstrakte Darstellung. So ist die Softwarearchitektur an sich eine Struktur von Strukturen (vgl. Definition 3.1) und bezieht sich nicht auf konkrete Implementationstechniken. Der damit einhergehende Informationsverlust bzw. die Fokussierung auf wesentliche und fachliche Aspekte der Realisierung verhindert eine triviale Abbildung der Soll- und auch der Ist-Architektur. Es ist daher nötig, herauszufinden, wie Elemente des Quelltextes auf Elemente der Beschreibungsnotation der Architektur, somit der Diagramme, zurückzuführen und abzubilden sind. Als Beispiel hierzu ist es relevant, zu klären, wie für eine Klasse im Quelltext eine Zugehörigkeit zu einer Schicht aus der Soll-Architektur, in diesem Fall eine Schichtenarchitekturdarstellung, ermittelt wird. Solche Abbildungen sollen dann als Regeln automatisiert anwendbar sein und dann zusammen mit den Transformationsregeln auf ein vergleichbares Format gebracht werden.

5.2 Anforderungen und Einschränkungen

Da es eine große Fülle an Programmiersprachen gibt, sollen zunächst für eine konkrete Programmiersprache die zu entwickelnden Ansätze geprüft werden. Dabei wird die Sprache C# von Microsoft auf der .Net Familie genommen. Diese Programmiersprache weist große Ähnlichkeiten zu Java auf, sodass eine Erweiterung auf diese Sprache leicht zu realisieren sein sollte. Dazu wird als Entwicklungsumgebung Microsoft Visual Studio 2010 Architecture Edition genutzt.

In der Modellierung sollen aus der UML besonders die statischen Aspekte der Architekturmodellierung besonders betrachtet werden, daher sollen Klassen- und auch Komponentendiagramme untersucht werden (vgl. Kapitel 3.3.1). Neben den flexiblen architekturbeschreibenden Möglichkeiten von Klassen- und Komponentendiagrammen soll ebenfalls untersucht werden, inwieweit bestehende Bauvorschriften für Software in Form von Referenzarchitekturen genutzt werden können. Daher soll hierzu die weit verbreitete Schichtenarchitektur angewandt werden.

Es sollen automatisierte Mechanismen für das Einlesen der Diagramme und der Repräsentanten der Ist-Architektur entwickelt werden. Hierbei wird untersucht, ob XMI oder andere proprietäre Formate genutzt werden können. Für das Einlesen der Ist-Architektur-Darstellung können Parser oder andere Mechanismen wie Reflections (vgl. Kapitel 4.1.1) genutzt werden, die es ermöglichen, relevante Informationen zu suchen und herauszufiltern. Die Eingabediagramme werden nicht auf inhaltliche oder syntaktische Korrektheit geprüft, es wird davon ausgegangen, dass diese sowohl semantisch als auch

syntaktisch korrekt als Eingabe dienen können. Die Eingabediagramme gelten daher per Definition für diese Arbeit als korrekt und bilden die Abbildungsgrundlage.

Als Ergebnis soll es dem Anwender ermöglicht werden, im Gegensatz zu den metrikbasierten Architekturanalyseverfahren, selbständig die Ergebnisse auswerten zu können. Dazu dient erneut ein Diagramm desselben Typs, wie das Eingabediagramm. In diesem sollen die Ergebnisse der Analyse abgebildet sein. Zur Orientierung und einfacheren Benutzung sollen Farben helfen, den Zustand bzw. das Ergebnis der Analyse darzustellen. So sollen korrekte Übereinstimmungen zwischen Soll- und Ist-Architektur grün, hingegen Abweichungen rot markiert werden. Mindere oder Teil- Verstöße gegen die Soll-Architektur sollen gelb akzentuiert werden. Ebenfalls sollen Abweichungen und Verstöße textuell, z.B. als Kommentar dargestellt werden, damit für den Endanwender die Transparenz über die gefundenen Abweichungen erhöht und eine Behebung vereinfacht wird.

Neben den nachfolgenden analytischen Untersuchungen und dem daraus entstehenden Design sollen Werkzeuge implementiert werden, die das Vergleichen der Untersuchungsobjekte ermöglichen. Dies soll prototypisch zur Verdeutlichung der Ansätze geschehen. Eine umfassende marktreife Implementierung der vorgestellten Ansätze ist nicht Bestandteil und Ziel dieser Arbeit. Dem Endanwender sollen Einstellungsmöglichkeiten zur Verfügung gestellt werden, die es ermöglichen, bestimmte Informationen zu filtern oder auszublenden, um die Fokussierung auf bestimmte Sachverhalte zu setzen. Dies bezeichnet der Begriff Konfiguration im Hinblick auf die zu entwickelnden Werkzeuge.

Da nicht nur die theoretische Sicht auf dieses Problem des Vergleichs zwischen Modellen und Quelltexten untersucht werden soll, werden die Ansätze auch mittels einer prototypischen Realisierung geprüft. Anhand dieser selbst erstellten Untersuchungsobjekte soll die Tauglichkeit der Ansätze verdeutlicht werden. Als Software zum Zweck der Prüfung des zu erarbeitenden Ansatzes wurde ein einfaches Shop-System entworfen. Dieses ist in C# geschehen und ist eine formbasierte Entwicklung ohne Datenbanken oder ähnlichem. Es basiert auf einer Schichtenarchitektur, die dem Werkzeug und Materialansatz (WAM) angelehnt ist.

Des Weiteren wurde ein Klassendiagramm zur Beschreibung des Shop-Systems erstellt, welches neben der Referenzarchitektur eine statische Sicht auf die Softwarearchitektur mittels UML darstellt. Dieses Shop-System weist 940 Zeilen Quelltext auf und beinhaltet circa 20 Klassen. Also ein sehr kleines Softwareprojekt, das aber einen einfachen und handhabbaren Untersuchungsgegenstand für die ersten Schritte zur Evaluierung der Ergebnisse der prototypischen Entwicklung darstellt. Die genutzte Schichtenarchitektur dieses Shop-Systems orientiert sich zugleich an dem WAM-Ansatz. WAM beinhaltet neben den vier darzustellenden Schichten noch weit mehr Implementierungsvorschriften, z.B. den Nutzen von nicht veränderbaren Fachwerten, den reglementierten Zugriff von Schichten aufeinander (z.B. Werkzeuge → Materialien, aber nie umgekehrt). Diese spiegeln sich nicht in dem Schichtenmodell wieder, sind aber in der Implementierung enthalten. Einschränkungen, wie der Zugriff zwischen Schichten, sind reglementiert. Dies ist in diesem Fall so zu verstehen, dass Tools auf Services zugreifen dürfen, aber nicht andersherum. Des Weiteren ist zu beachten, dass bei der Sicht per Schichten implizit davon ausgegangen wird, dass ein Überspringen von Schichten (z.B. der Zugriff von

Werkzeugen direkt auf Materialien) möglich ist. Wenn solch ein Verhalten nicht gewünscht ist, muss dies im Diagramm kenntlich gemacht werden, z.B. über einen Kommentar.

Des Weiteren sollen Komponentendiagramme untersucht werden. Diese sind jedoch nicht in dem hier dargestellten Shop-System enthalten, da bereits eine andere Architekturrichtlinie, und zwar die Schichtenarchitektur, genutzt wurde. Da hierbei der Fokus auf die Schichten lag, wurden keine Komponenten definiert. Für die Untersuchung an Komponenten wurde ein spezielles Softwareprojekt entwickelt, um die Vergleiche mit Komponentendiagrammen untersuchen und durchführen zu können.

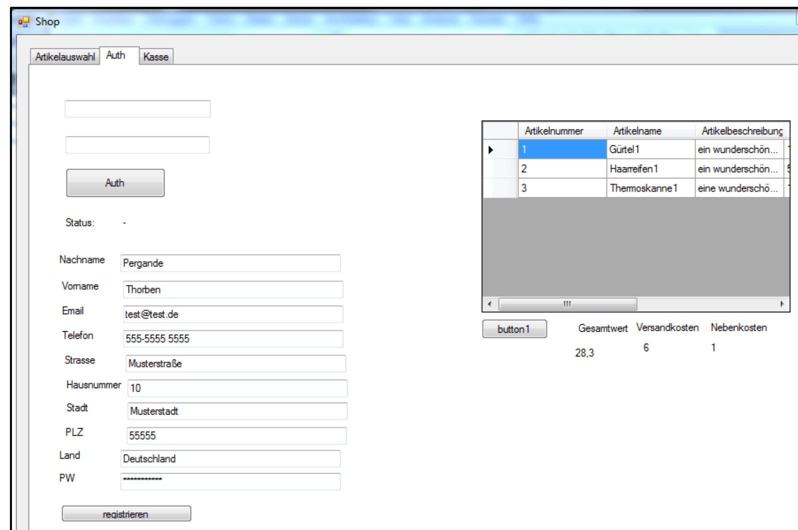

Abbildung 8: Auszug aus dem Shop-System

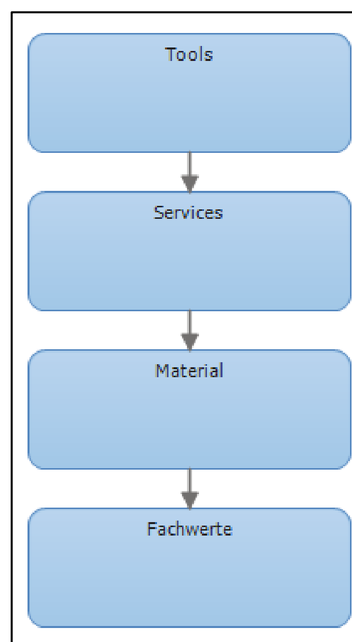

Abbildung 9: Genutzte Schichtenarchitektur

6 Analyse

Dieses Kapitel umfasst die Analyse der bestehenden Problemstellungen und versucht Lösungswege für das Erreichen der in dieser Arbeit definierten Ziele aufzuzeigen.

6.1 Analyse des Problemfeldes des Diagramm-Imports

In diesem Abschnitt wird untersucht, wie die zu sammelnden Informationen aus den XML-basierten Diagrammen ermittelt werden können. Dazu werden Beispiele aus vorliegenden Diagrammen erläutert. Neben standardisierten Formaten, wie XMI, unterscheiden sich die XML-Strukturen von Anbieter zu Anbieter. Der Ablauf der folgenden Analyse soll auch grundlegende Möglichkeiten aufzeigen, wie die zu ermittelnden Informationen gefunden werden können.

Wie im Kapitel 5.2 beschrieben, sind für den hier aufgezeigten Ansatz Anforderungen definiert, die Auswirkungen auf die Eingabemöglichkeiten haben. So kann aus diversen XML-basierten Werkzeugen ein Klassen-, Komponenten- oder Schichtendiagramm erstellt werden. Zudem ist die Repräsentation des Quelltextes in C# zu untersuchen.

Die Diagramme, die als Eingabe für diese Ausarbeitung gelten, wurden allesamt mit Visual Studio erstellt. Die Entscheidungsgrundlage hierfür ist die Integration der Erstellungswerkzeuge in die Entwicklungsumgebung, in der die Prototypen realisiert wurden. Es soll aber auch möglich sein, andere Werkzeugerzeugnisse mittels XMI- oder werkzeuggebundener XML-Schnittstellen oder -Formaten zu nutzen. Das wird in der Analyse berücksichtigt werden.

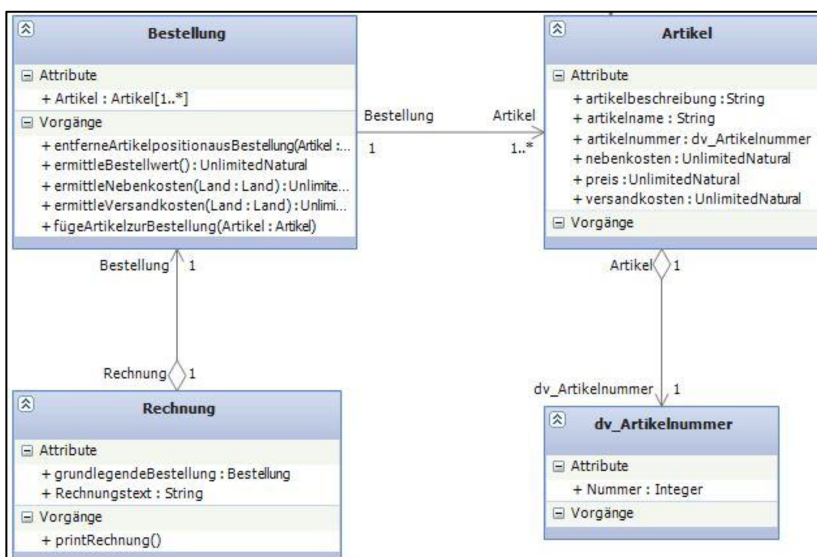

Abbildung 10: Auszug Klassendiagrammdarstellung

Die Abbildung 10 zeigt einen Auszug aus der Modellierung des zu untersuchenden Shop-Systems. Hierbei wird der Zusammenhang zwischen Artikeln, Artikelnummern,

Bestellungen und Rechnungen dargestellt. Sowohl Artikel, als auch Artikelnummern ("dv_Artikelnummer") bestehen nur aus Attributen und haben keine zu implementierenden Methoden. Der jeweilige Typ eines Attributs und auch der Typ einer Methode wird durch folgende Notation beschrieben: <Attribut-/Methodenname> : <Typ>. Somit besitzt das Attribut „artikelname“ der Klasse Artikel den Typ „String“. Die Multiplizität, also wie oft ein Attribut in einer Klasse geführt wird (Listen), ist im Fall von 1 nicht dargestellt und im Falle von mehreren Exemplaren durch die Notation <[Multiplizität]> hinter dem Typ-Namen aufgezeigt. Solch ein Beispiel findet sich im Attribut Artikel in der Bestellungs-Klasse, wobei eine Bestellung mehrere Artikel umfassen kann. Es wird ebenfalls implizit über die Verbindung zwischen Bestellung und Artikel über das „*“ an der Artikelseite dargestellt. Hierbei handelt es sich um eine unidirektionale Verbindung, d.h. nur Bestellungen können auf Artikel zugreifen, nicht umgekehrt. Dies wird durch eine Pfeilrichtung im Diagramm modelliert.

Wie gefordert, werden diese Informationen in Diagrammen aus Visual Studio XML-basiert gespeichert. Dabei soll nun fortlaufend die Repräsentation der Klasse „Bestellung“ aus der Abbildung 10 aufgeführt werden, um den Aufbau eines Eingabediagramms in XML beispielsweise vorzustellen.

```
<logicalClassDesignerModel …>
…
</logicalClassDesignerModel >
```

Der Tag „logicalClassDesignerModel“ beschreibt, dass es sich bei diesem Diagramm um ein Klassendiagramm handelt. Klassen werden wie folgt identifiziert:

```
<class Id="ea61721f-6925-4133-8e39-e4ac1942c656" name="Bestellung"
isAbstract="false">
```

An dieser Stelle findet sich die Klasse Bestellung, die zum Einen eine diagrammweite eindeutige ID als auch den Bezeichner „Bestellung“ als Attribut hat. Ebenfalls ist hier die Information enthalten, ob es sich um eine abstrakte Klasse handelt. Handelt es sich um eine Schnittstelle, ändert sich das XML-Tag in den Namen „interface“, behält aber dieselbe Struktur und den Aufbau einer Klassendarstellung.

In den XML-Tags „targetEnds“ befinden sich alle Informationen über die ausgehenden Verbindungen einer Klasse oder Schnittstelle.

```
<targetEnds>
         <association sourceEndRoleName="Bestellung"
         targetEndRoleName="Artikel" isAbstract="false">
           <classMoniker Id="cb4724a9-1c45-48ce-be92-895e3a1f662b"
           LastKnownName="Artikel" />
            <relationshipOwnedElementsInternal>
              <associationHasOwnedEnds>
                   <memberEnd Id="a1cacce6-be52-4642-b826-d22b0b43e1fc"
                   name="Bestellung" aggregation="None" isComposite="false"
                   ">
                   </memberEnd>
              </associationHasOwnedEnds>
              <associationHasOwnedEnds>
                   <memberEnd Id="ac031bc1-b01b-495e-bcca-92136b679071"
                   name="Artikel" aggregation="None" isComposite="false">
                         <lowerValueInternal>
                                     <literalString name="Lower" value="1">
```

```
                        </literalString>
                  </lowerValueInternal>
                  <upperValueInternal>
                        <literalString name="Upper" value="*"/>
                  </upperValueInternal>
            </memberEnd>
         </associationHasOwnedEnds>
       </relationshipOwnedElementsInternal>
     </association>
   </targetEnds>
```

Der XML-Tag „association" bezeichnet eine Verbindung zwischen Klassen, die keine Realisierung oder Vererbung sind. In diesem Tag werden ebenfalls die Rollennamen über Attribute (sourceEndRoleName & targetEndRoleName) definiert. Der XML-Tag „classMoniker" bezeichnet eine Referenz auf die verbundene Klasse, in diesem Fall die Klasse Artikel. Die Bezeichnungen, Multiplizitäten und die Art der Assoziation werden in den danach folgenden Zeilen durch das XML-Tag „relationshipOwnedElementsInternal" beschrieben. Über die Attribute „aggregation" und „isComposite" wird die Unterscheidung zwischen Assoziation, Aggregation und Komposition festgelegt. Die erstgenannte Verbindung stellt immer die Quelle (Bestellung) und die zweitgenannte Verbindung das Ziel (Artikel) dar. Die Multiplizität wird durch die geschachtelten XML-Tags „lowerValueInternal" und „upperValueInternal" beschrieben.

Klassenattribute, repräsentiert durch das XML-Tag „property", befinden sich unter dem XML-Tag „class" zwischen den XML-Tags „ownedAttributesInternal" und stellen sich wie folgt dar:

```
<property Id="d8988667-f1a1-472c-9fbb-fe0258d67b38" name="Artikel"
isLeaf="false" isStatic="false" isReadOnly="false" isUnique="false"
isDerived="false" isDerivedUnion="false" aggregation="None"
isComposite="false">
```

Jedes Attribut hat eine eindeutig zu referenzierende ID und einen Namen. Weiter beschreibende Informationen werden über Attribute wie z.B. „isReadOnly" hinterlegt. Eine Multiplizität wird wie in den Verbindungen durch „lower/upperValueInternal" beschrieben.

Methoden, repräsentiert durch den XML-Tag „operation" haben prinzipiell den selben Aufbau wie Attribute und Verbindungen, benötigen aber im Gegensatz zu den Attributen noch weiterführende Informationen zu den Parametern, die hier nun beispielhaft aufgezeigt werden:

```
<parameter Id="42224ab1-aa5c-4f60-a0d9-2cf843d29a6a"
name="Artikel" direction="In">
      <type_NamedElement>
            <referencedTypeMoniker Id="457d3b97-3ae4-4285-8917-
            2b20b9ec7a2a" LastKnownName="Artikel" />
      </type_NamedElement>
 </parameter>
```

Parameter haben ein Attribut, welches die Richtung des Parameters anzeigt (in, out und Return). Rückgabewerte werden somit nicht direkt in der Methode als Typ angezeigt, sondern als Parameter mit der Richtung „Return". Der eigentliche Rückgabetyp ist über den „referencedMoniker" referenziert. Vererbungs- und Realisierungsbeziehungen werden analog behandelt.

Neben Klassendiagrammen sollen auch Schichtendiagramme untersucht werden, wie in Abbildung 10 im Kapitel 5.2 aufgezeigt. Schichtendiagramme, in Visual Studio erstellt, werden ebenfalls XML-basiert gespeichert. Der Aufbau ist im Gegensatz zu den Klassendiagrammen einfacher, wenn auch sehr strukturähnlich. Schichtendiagramme werden durch das XML-Tag „layerModel“ identifiziert. Einzelne Schichten sind durch den XML-Tag „layer“ geschachtelt in der Sammlung „layers“ enthalten:

```
<layer Id="08617340-007e-4484-ae7b-c35f500c3b01" name="Tools">
```

Jede Schicht (Layer) hat eine eindeutige referenzierbare ID und einen Namen, in diesem Fall „Tool“, der als Attribut des XML-Tags abgelegt ist. Mehr Informationen sind in einer Schicht nicht vorhanden. Verbindungen werden ebenfalls wie bei den Klassendiagrammen nur auf die ausgehenden Verbindungen einer einzelnen Schicht dargestellt:

```
<dependencyToLayers>
      <dependencyFromLayerToLayer Id="0c6f8443-4083-4c7e-ab84-
      03c9313dfa45"
      <layerMoniker Id="f8a5ee22-fa3f-4d79-8b18-557b8bf2b0f6" />
      </dependencyFromLayerToLayer>
</dependencyToLayers>
```

Jede Verbindung hat eine eindeutige ID. Die verbundene Schicht wird nicht über den Namen, sondern ausschließlich über die ID der Schicht referenziert (layerMoniker). Aus diesen beiden Strukturen können die Schichten dargestellt werden.

Komponentendiagramme unterscheiden sich in der XML-Struktur nicht von den beiden soeben dargestellten Beispielen. Nur die Namensgebung ist entsprechend der Komponentensemantik angepasst.

Die hier dargestellten XML-Strukturen können über einen XML-Parser analysiert werden. Zunächst muss für jeden Eingang eines Diagramms aus einem Werkzeug analysiert werden, wie die XML-Struktur aufgebaut ist. Anschließend können die Informationen über standardisierte XML-Einleseverfahren, wie z.B. XPath oder selbstgeschriebene XML-Parser ermittelt werden. So ist es auch möglich, die Informationen aus Diagrammen in dem Programmfluss des Analysewerkzeugs zu bearbeiten. Es gilt aber noch zu klären, welche Informationen aus den Diagrammen genommen werden sollen, ob alle nötig und abbildbar sind. Die Frage, welche Informationen für eine Softwarearchitekturanalyse nötig sind, wird in den folgenden Abschnitten behandelt.

Allgemein muss in den vorliegenden Diagramm-Formaten, egal von welchem Anbieter, ermittelt werden, wie die architekturbeschreibenden Element, wie zum Beispiel die Klasse in einem Klassendiagramm, repräsentiert ist. So ist es möglich mit XML-Werkzeugen diese Informationen programmatisch zu extrahieren.

6.2 Analyse der Quelltexteingabemöglichkeiten

In der Programmiersprache C# werden die Quelltexte als Textdateien mit der Dateiendung ".cs" abgelegt. Zur Erlangung von Informationen über den Quelltext und dessen Inhalte müssen diese einzelnen Quelltextdateien analysiert werden. In C# ist es ebenfalls möglich, innerhalb einer Datei mehrere Klassen zu definieren, sodass nicht davon ausgegangen werden kann, dass jede cs-Datei genau eine Klasse oder Schnittstellendefinition repräsentiert. Um nun Informationen aus dem Quelltext zu erhalten, z.B. welche Attribute eine Klasse besitzt und welche Verbindungen zu anderen Klassen vorliegen, gibt es verschiedenen Ansätze.

Als Parser bezeichnet man Mechanismen, die die Zerlegung und den Informationsgewinn für Quelltexte unternehmen. Parser werden z.B. in Entwicklungsumgebungen eingesetzt, um den Prozess zur Kompilierung und Fehleranzeige zu ermöglichen. Der in Visual Studio eingesetzte Parser ist nicht frei verfügbar und somit auch nicht in dem Entwicklungsprozess zugänglich. Ein Parser würde den Zugriff auf die im Quelltext enthaltenen Informationen ermöglichen. Leider sind auch keine anderen frei verfügbaren und handhabbaren Parser in der Recherche gefunden worden. Als letzte Möglichkeit ist der Selbstbau eines Parser für C# eine Option, die aber aufgrund des sehr hohen Aufwands und Fehlerpotenzials nicht weiter verfolgt wird.

Eine Alternative, um Informationen über den Quelltext bzw. die entwickelte Software und somit Teile der Ist-Architektur zu ermitteln, ist ein API namens „Reflections“ (vgl. Kapitel 4.1.1). Reflections ermöglicht es, Informationen aus einem kompilierten und lauffähigen C#-Programm zu ermitteln, z.B. welche Typen enthalten sind, welche Schnittstellen ein Typ implementiert oder auch, welche Attribute (in C# unterschieden nach Feldern und Eigenschaften) ein Typ hat. Ein Typ entspricht hierbei einer implementierten Klasse oder Schnittstelle. Reflections nutzt ein rekursives Verfahren, sodass zu allen Attributen, welche selbst Typen sind, wiederum Informationen über den zugrundeliegenden Typ als auch weitere Informationen, wie z.B., ob es sich um eine statische Variable handelt, enthalten sind.

Durch das Fehlen einer effizienten Lösung für einen Parser soll der Ansatz der Analyse des Quelltextes bzw. des daraus resultierenden Software-Artefakts über Reflections in dieser Arbeit genutzt werden. Dazu muss eine lauffähige Softwareversion des zu untersuchenden Softwarepakets (kurz Assembly) über die Reflections API eingelesen werden. Dazu reicht ein Pfad zu der zu untersuchenden Assembly, um den programmatischen Zugriff per C# zu erhalten.

Reflections-Bibliotheken sind nicht nur für C#, sondern auch für andere Programmiersprachen wie Java verfügbar, sodass der Einsatz dieser Technik grundsätzlich nicht auf die Programmiersprache C# beschränkt ist.

6.3 Analyse des Problemfeldes der Vergleichbarkeit

In diesem Abschnitt wird das Problemfeld der Vergleichbarkeit von Diagrammen und Softwaresystemen analysiert. Grundsätzlich ist ein direkter Vergleich von in XML vorliegenden Diagrammen, und Softwaresystemen, z.B. über den Quelltext, aufgrund der Unterschiedlichkeit und verschiedenen Semantik nicht trivial herzustellen. Des Weiteren soll ein Vergleich in beherrschbarer Zeit und nachvollziehbar durchzuführen sein. Die Informationen, die in Diagrammen vorliegen, unterscheiden sich im Ausdruck und der Struktur grundsätzlich von den Ausdrücken einer Programmiersprache. So ist in der Modellierung von Schichtendiagrammen die Darstellung auf abstrakte Schichten und die Zugriffsmöglichkeiten von Schichten zueinander im Fokus. In objektorientierten Programmiersprachen werden Klassen, Schnittstellen mit Variablen und Methoden, die das eigentliche Verhalten einer Klasse beinhalten, beschrieben und programmiert. Hier liegt der Fokus auf der Funktionsweise der Klassen. Explizit werden Verbindungen zu anderen Klassen nicht „programmiert“, sondern z.B. über das Nutzen anderer Klassen und deren Verhalten verwendet. Hierbei steht aber nicht die Verbindung, sondern die Funktion im Vordergrund.

In der Model Driven Architecture (Petrasch, et al., 2006) können die Modellierungsgrade definiert werden. So ist die Realisierung einer Softwareklasse, somit der Quelltext dieser Klasse, ebenfalls eine Modellierung des lauffähigen Softwaresystems und wird mit der Modellierungsstufe M0 bezeichnet. Konkrete Diagramme, z.B. nach UML-Definition eines Klassendiagramms, stellen eine Abstraktion der Stufe M0 dar und beschreiben die Eigenschaften der M0-Modelle und aus welchen Elementen eine Software bestehen soll. Eine Ebene darüber, die M2-Modelle oder Meta-Ebene, werden Elemente beschrieben, aus denen die M1-Ebene bestehen kann, z.B. eine UML-Klasse und deren Definition. Dieser Ansatz ermöglicht eine Zerlegung der architekturbeschreibenden Elemente in granulare Informationsstücke. Beispielsweise kann so entschieden werden, aus welchen Teilen eine UML-Klasse besteht, wie z.B. UML-Attribute. Eine Zerlegung geht noch weiter, da ebenfalls definiert wird, aus welchen Teilen ein UML-Attribut besteht.

Der direkte Vergleich von Modellen unterschiedlicher Stufen stellt ein zu lösendes Problem dieser Arbeit dar. Dies liegt darin begründet, dass Modelle unterschiedlicher Stufen andere Semantiken und Ausdrucksstärken als auch Abstraktions- und Detaillierungsgrade haben. Somit wird in dieser Arbeit davon ausgegangen, dass eine Vergleichbarkeit ein Vorliegen der Modelle auf derselben Stufe bedingt. Wie die Informationen aus den vorliegenden Modellen abgebildet werden, wird im folgenden Problemfeld „Abbildbarkeit“ (vgl. Kapitel 6.4) beschrieben. Basierend auf dieser Annahme ergeben sich drei Möglichkeiten, eine Vergleichbarkeit herzustellen:

1. Reduktion des M1-Diagramms auf die M0-Stufe
2. Abstraktion des M0-Quelltextes auf die M1-Stufe
3. Abstraktion des M1-Diagramms und des M0-Quelltextes auf die M2-Stufe

Solch ein Vorgehen der Reduktion oder Abstraktion auf andere Modellierungsstufen wird in der MDA als Transformation eines Modells bezeichnet.

Die erste Variante, die Reduktion des M1-Diagramms auf die M0-Stufe, würde eine Realisierung der Diagramme hin zum Quelltext beschreiben. Solche Transformationen bestehen bereits in der Form von Codegeneratoren. Es gibt Anbieter, die für verschiedenen Programmiersprachen und Diagrammtypen Codegeneratoren anbieten, teilweise sind Codegeneratoren, wie bei ArgoUML, bereits in die Modellierungswerkzeuge integriert.

Dieser Ansatz wird ebenfalls in der Quelle (Papadopolous, 2008), vgl. Kapitel 2.4, beschrieben. Auffällig ist an diesem Ansatz, dass zumeist nur Grundgerüste des Quelltextes generiert werden können. So können die Attribute eines Klassendiagramms als auch die Klasse an sich direkt in die jeweilige Programmiersprache umgewandelt werden. Jedoch ist der Realisierungsgrad bei der Umsetzung von Methoden auf die Signatur und den Methodenkopf reduziert. Die Semantik der einzelnen Diagramme reicht meistens nicht aus, um das Verhalten einer Methode automatisiert zu erstellen. Als Workaround hierfür gibt es Ansätze, die statische und dynamische Modelle integrieren, sodass die Gerüste durch die statische Sicht, z.B. Klassendiagramme und das Verhalten durch Aktivitätsdiagramme realisiert werden. Der Nachteil solcher Lösungen besteht darin, dass der Großteil der Hersteller von Codegeneratoren nur auf das Gerüst der Software spezialisiert ist und das eigentliche Verhalten, also die Methodenrümpfe, durch Entwickler realisiert werden müssen.

Dies ist zwar in einigen Bereichen eine Arbeitserleichterung, reicht aber an dieser Stelle für diese Ausarbeitung nicht aus, da gerade das Verhalten in den Methodenrümpfen zahlreiche Verbindungen zu anderen Modellierungselementen beinhalten kann (z.B. der Zugriff auf eine andere Klasse). Hier wäre für den Ansatz des Vergleichs eine zu große Lücke. Integrierte Lösungen, wie sie Neuhardt in (Neuhardt, 2008) durch die Implementierung von Codegeneratoren inklusive Aktivitätsdiagrammen zur Modellierung der Methodenabläufe behandelt, waren für diese Ausarbeitung nicht in Form von einsatzfähigen Softwarelösungen verfügbar. Des Weiteren bedingt dieser Ansatz eine zweifache Modellierung, da zum einem die statische und zum anderen auch die dynamische Sicht modelliert sein muss. Ein Ziel des in dieser Arbeit verfolgten Ansatzes ist es auch, möglichste wenige Regeln für die Modellierung an die Entwickler zu stellen, sodass die vorgestellte Variante 1 nicht weiter verfolgt wird.

Variante 2 beschreibt ein Verfahren, indem aus dem Quelltext oder vorliegendem Softwaresystem ein M1-Modell abstrahiert werden soll. Dazu muss definiert sein, welche Elemente das Zieldiagramm hat und wie eine Abbildung der Quelltextelemente hierauf aussieht. Eine Abbildung für einen definierten Fall, die Abbildung einer objektorientierten Sprache hin zu einem Klassendiagramm, wurde bereits in Abschnitt 2.2 unter der Quelle (Vinita, et al., 2008) beschrieben. Hierbei wurden Abbildungsregeln aufgestellt, die eine Identifikation der relevanten Daten ermöglichen. Für weitere Diagramme, z.B. Schichtendiagramme oder Komponentendiagramme, müssten weitergehende spezialisierte Regeln aufgestellt werden. Allgemein erscheint dieser Ansatz für diese Arbeit geeignet.

In der letzten genannten Variante 3 müssten sowohl die M0- als auch M1-Modelle zu einem zu definierenden M2-Modell hin transformiert werden. Wie bereits unter Abschnitt 3.3 (Grundlagen Modellierung) beschrieben, stellen M2-Modelle jedoch die grundlegenden Elemente bereit, aus denen M1-Modelle bestehen. Man würde somit mit

M2-Modellen M1-Modelle definieren. Der damit einhergehende Informationsverlust durch die Abstraktion ist für eine Vergleichbarkeit von Diagrammen und Quelltexten aber nicht tragbar.

Von den vorgestellten Varianten erscheint die Variante 2 am aussichtsreichsten, da es hierfür bereits verfolgte und beschriebene Ansätze für Klassendiagramme gibt. Um eine grundlegende Vergleichbarkeit auf einer Modellierungsstufe anzustreben, soll in dieser Arbeit der Ansatz verfolgt werden, aus dem M0-Modell ein M1-Modell zu abstrahieren.

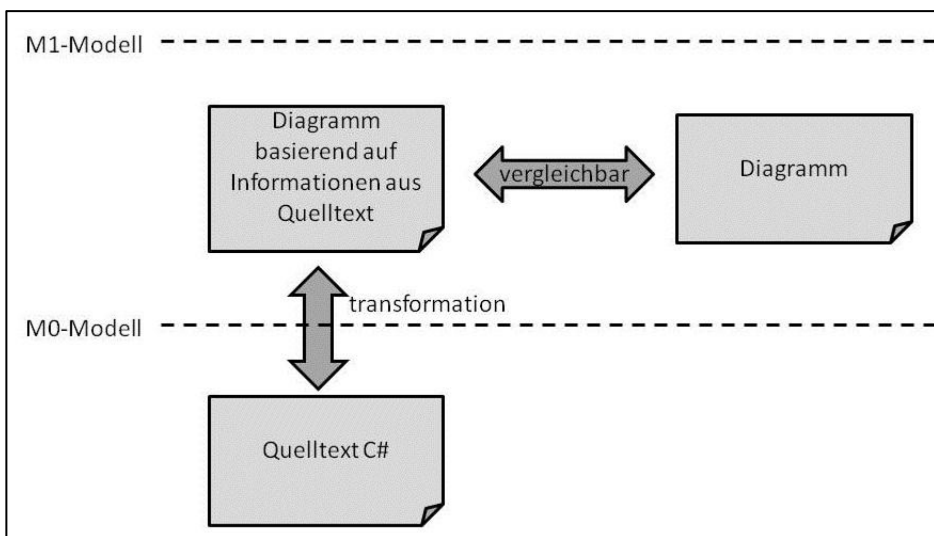

Abbildung 11: Ansatz zur Herstellung der Vergleichbarkeit

So soll beispielsweise definiert werden, wie aus einem Quelltext oder vorliegendem Softwaresystem ein Schichtendiagramm abgeleitet werden kann. Dazu muss geklärt werden, wie Schichtenzugehörigkeiten von Klassen im Quelltext definiert und extrahierbar sind. Ebenfalls muss hierzu definiert sein, welche Elemente der M1-Modellierung für die Architekturanalyse überhaupt von Relevanz sind. So kann es sein, dass Elemente und Möglichkeiten der Modellierung den Fokus der Architekturvergleiche verlassen und nicht behandelt werden sollen. Somit ist es erforderlich zu analysieren, welche Informationen und Elemente eines Diagrammes, z.B. die Komponenten eines Komponentendiagramms, untersucht werden sollen. An dieser Stelle gibt es eine enge Kopplung der Problemfelder Abhängigkeit und Abbildbarkeit. Bevor noch weitere Kriterien an die Vergleichbarkeit, wie die Geschwindigkeit und Nachvollziehbarkeit, untersucht werden, wird im Folgenden das Problemfeld der Abbildbarkeit beschrieben.

6.4 Analyse des Problemfelds der Abbildbarkeit

Eine Ist-Architektur beschreibt den Quelltext eines Softwaresystems und auch die Abbildungsregeln auf die repräsentierten Architekturelemente. Zur Erreichung dieser Definition fehlen in den gewählten Ansätzen noch die Abbildungen auf die zu repräsentierenden Architekturelemente. Dazu muss geklärt werden, welche Architekturelemente überhaupt bestimmt und wie diese im Quelltext repräsentiert werden sollen. Bei der Zerlegung in relevante Architekturelemente ist eine vorhergehende Analyse des Diagrammtyps nötig, damit erkannt werden kann, aus welchen Elementen das jeweilige Diagramm besteht. Hierzu kann beispielsweise eine Spezifikation wie die MOF oder andere gelten, die aufzeigt, aus welchen Elementen ein Diagramm besteht. Bei der Zerlegung in die zu identifizierenden Elemente gibt es Freiheitsgrade, welche Elemente für eine nachfolgende Analyse betrachtet werden sollen. Im Beispiel eines Schichtendiagramms muss nun eine manuelle Identifikation der relevanten Architekturelemente erfolgen. Dazu sollten alle einzelnen Elemente, die modelliert werden können, betrachtet werden. Dies sind im Falle eines Schichtendiagramms eine Schicht an sich und die Verbindung zu anderen Schichten. Analog zu den Diagrammdarstellungen sollten hierbei nur die ausgehenden Verbindungen einer Schicht betrachtet werden. Daraus ergeben sich folgende Informationen, die abgebildet werden sollten:

- Schichtenrepräsentation
- Name der Schicht
- Ausgehenden Verbindung zu anderen Sichten (und der Name der Schicht)

Äquivalent hierzu ist bei anderen Diagrammtypen vorzugehen. Auch hier ist eine Definition der zu repräsentierenden Elemente nötig. Je nach Komplexität eines Diagramms können auch geschachtelte Definitionen nötigt werden, um ein Diagramm zu zerlegen. Hiermit ist gemeint, dass es komplexere Strukturen als den Namen einer Schicht gibt, die eine weitere Zerlegung benötigen. Dies ist im Beispiel eines UML-Klassendiagramms bei einem Attribut der Fall. Eine Klasse besitzt eine Menge von Attributen, die wiederum aus mehreren Informationen bestehen. So hat ein Attribut einen Namen und auch einen Typ. Informationen, die für eine Analyse nicht in Betracht gezogen werden müssen, können ausgelassen werden.

Nachdem nun aufgezeigt wurde, wie zunächst die architekturbeschreibenden Elemente identifiziert werden können, und dass eine Zerlegung der Diagrammstruktur nötig ist, bleibt noch die Identifikation dieser Elemente im Quelltext offen.

Um bei dem Beispiel des Schichtendiagramms zu bleiben, gilt es nun, Repräsentanten im Quelltext eines Softwaresystems für die identifizierten Architekturelemente zu bestimmten. Dies bedarf einer Analyse der eingesetzten Sprache. Einige Elemente, wie z.B. Klassen eines Klassendiagramms, sind relativ einfach zu identifizieren, da es sich bei der Modellierung und dem Quelltext um strukturähnliche Darstellungen handelt. So gibt es in objektorientierten Sprachen klassenbezeichnende Syntax-Elemente im Quelltext. Schwieriger ist der Umgang mit Elementen, die keine direkte Repräsentation in der Syntax einer Programmiersprache besitzen, wie z.B. die Schichten eines Schichtendiagramms. An dieser Stelle ist mittels Wissen über den Aufbau einer Programmiersprache ein geeigneter Repräsentant zu finden oder festzuhalten, dass eine

direkte Abbildung nicht möglich ist. Oftmals gibt es mehrere Varianten, um solche Repräsentanten im Quelltext zu finden.

Dies soll folgendes Beispiel verdeutlichen: Es sollen Schichtendiagramme analysiert werden. Dabei wurde festgehalten, dass ein Typ der Schicht aus der Soll-Architektur auch im Quelltext repräsentiert werden soll. Da es keine direkte Abbildung in der Programmiersprache C# für Schichten gibt wurde nach alternativen Repräsentanten gesucht. Folgende Alternativen wären hier beispielsweise möglich:

- Namenskonvention im Klassennamen layer_<Layername>_<Klassenname>
- Einbeziehen des Namespace-Konstruktes in C#, das eine Zusammengehörigkeit und Sichtbarkeit darstellt, z.B. <softwareprojektname>.<layer>
- Die zugrundeliegende Ordnerstruktur in der die Quelltextdateien gespeichert sind, gelten als Repräsentanten der Schicht, dabei stellt der Ordnername den Schichtennamen dar

Aus diesen drei Alternativen kann für den jeweiligen Kontext der Architekturanalyse der jeweilige Favorit gewählt werden. Die Wahl des Favoriten sollte jedoch mit dem vorliegenden Entwicklungsprozess und den definierten Projektregeln abgeglichen werden. Wenn keine Ordnerstruktur oder Namenskonvention vorgegeben ist, ist es sehr unwahrscheinlich, dass verschiedene Entwickler diese identisch realisiert haben. Ebenfalls sollte bei der Auswahl des Favoriten eines Repräsentanten bereits auf die Umsetzbarkeit geachtet werden. Neben der Frage, wo eine Information zu finden ist, sollte auch die Frage gestellt werden, wie eine Information extrahiert werden kann, um eine Vergleichbarkeit herzustellen. Im Falle der Namenskonvention, der Namespaces und auch der Dateiordner kann der jeweilige Name einer Klasse oder der Name der Quelltextdatei mittels eines Parser oder über Reflections extrahiert werden. Somit ist eine Ermittlung der benötigten Information bei allen gewählten Beispielen gleichfalls vorhanden und es können andere, je nach Arbeitskontext vorhandene, Kriterien frei gewählt werden.

Neben den statischen Aspekten der Ermittlung von Informationen, wie z.B. Bezeichner von Attributen oder Schichtennamen und Zugehörigkeit, müssen auch die Verbindungen zwischen Architekturelementen gefunden werden. Auch hier muss ein wie eben dargestellter Prozess der Ermittlung von Repräsentanten stattfinden. Verbindungen in Programmiersprachen werden meist als Aufruf, Deklaration oder Instanziierung von anderen Klassen dargestellt. Somit stellt in einer Programmiersprache die Benutzung einer Klasse durch eine andere Klasse eine Verbindung dar. In der Diagrammsemantik werden jedoch in manchen Fällen unterschiedliche Verbindungen mit unterschiedlicher Semantik verwendet. So gibt es in der Modellierung von Klassendiagrammen drei semantisch unterschiedliche Verbindungsarten zwischen Klassen:

- Assoziation (benutzt-Beziehung)
- Aggregation (enthält-Beziehung)
- Komposition (lebensbindende enthält-Beziehung)

Eine Unterscheidung der verschiedenen Ausdrucksstärken dieser Verbindungen lässt sich nur schwer bis gar nicht im Quelltext identifizieren. Da im Quelltext die Benutzung einer anderen Klasse, in Methodenrümpfen, Konstruktoren, Signaturen etc., die einzige Verbindung zu anderen Klassen darstellt, gibt es an dieser Stelle ein semantisches

Ungleichgewicht in der Ausdrucksstärke der Verbindungen zwischen Modell und Quelltext. Es wäre zwar denkbar, Programmierstile und Konventionen einzusetzen, die eine Unterscheidung der Verbindungen im Quelltext zumindest teilweise ermöglichen. Jedoch würde dies den Entwicklungsprozess und eine allgemeingültige und eingehaltene Durchsetzung dieser Konventionen bedeuten. Solch eine strenge Vorgabe kann den Einsatz des hier vorgestellten Ansatzes gefährden. Daher wurde sich dafür entschieden, davon auszugehen, dass es in den Quelltexten nur eine Repräsentation von Verbindungen gibt.

Es muss somit entschieden werden, welchem Verbindungstyp der Modellierung die Repräsentation im Quelltext zugesprochen werden soll. Alle anderen Verbindungsarten für Enthält-Beziehungen werden sonst nicht weiter unterschieden. Dies muss im Folgenden auch in der Auswertung und Vergleichbarkeit beachtet werden. Des Weiteren gibt es statische Architekturelemente, die mehrere Klassen eines Quelltextes umfassen können. So ist es möglich und auch wahrscheinlich, dass in der Behandlung eines Schichtendiagramms mehrere Klassen zu einer Schicht gehören können. In solchen Fällen ist es erforderlich, dass alle umschlossenen Klassenverbindungen aller einer Schicht enthaltenen Klassen berücksichtigt werden, um die Verbindungen einer einzelnen Schicht zu repräsentieren.

6.5 Zusammenführung der Problemfelder Vergleichbarkeit und Abbildbarkeit

Nachdem in den vorhergegangenen Abschnitten gezeigt wurde, dass eine Zerlegung der architekturbeschreibenden Elemente für die Erstellung der Abbildbarkeit und auch die Transformation des Quelltextes, oder anderer Repräsentanten hierfür, in den eingegebenen Diagrammtypen für die Vergleichbarkeit essentiell sind, werden in diesem Abschnitt beide Ansätze zusammengeführt. Eine Abbildung des Quelltextes auf die architekturbeschreibenden Elemente enthält gleichermaßen Regeln für die Transformation des Quelltextes in ein M1-Modell, weil hierbei Extrakte der modellierten Diagramme als Beschreibung der abzuleitenden Elemente enthalten sind. Diese Regeln lassen sich ebenso in die entgegengesetzte Richtung anwenden. So können aus dem Quelltext direkte Repräsentanten eines Diagramms bestimmt werden, sofern es einschränkungslose Regeln sind. Diese aufgestellten Regeln können auch genutzt werden, um Diagramme auf den Quelltext abzubilden. Es handelt sich um bidirektionale Abbildungen, sodass aus dem Quelltext Verweise auf Diagrammelemente zurückzuschließen sind.

Ein Vergleich soll nachvollziehbar sein. Verschiedene Modellierungswerkzeuge nutzen unterschiedliche Formate zur Speicherung oder unterstützen ausschließlich oder zusätzlich Standards wie XMI. Zum jetzigen Zeitpunkt der Analyse wäre es denkbar, dass sowohl das XML-basierte Diagrammformat, als auch die Mapping-Regeln, die eine Zerlegung der architekturbeschreibenden Elemente enthalten, als Vergleichsgrundlage zu nehmen. Dazu müsste programmatisch ermittelt werden, welche Elemente in dem Diagramm beschrieben sind und diese müssen mit den Ergebnissen der angewandten Abbildung verglichen werden. Da sich die grundlegende Struktur aber anbieterabhängig

unterscheiden kann, wäre hier eine starke Bindung zwischen Anbieterformat und Abbildungsregeln vorhanden. So müsste z.B. für Anbieter 1 ein Vergleichswerkzeug für Klassendiagramm und C# entwickelt werden, aber da Anbieter 2 eine andere Repräsentation wählt, müsste ein Werkzeug mit derselben Logik und angewandten Regeln für den zweiten Anbieter komplett neu implementiert werden.

Somit wäre für einen Diagrammtypen die Abbildungsregel der Programmiersprache doppelt entwickelt und die Anzahl der Entwicklungsschritte steigt proportional mit der Anzahl der unterstützten Anbieter an. Ebenfalls ist für die Implementierung der Abbildungsregeln des Quelltextes Wissen über die Struktur des anbieterabhängigen Diagrammformats nötig. Folgende Übersicht soll dieses Problem verdeutlichen:

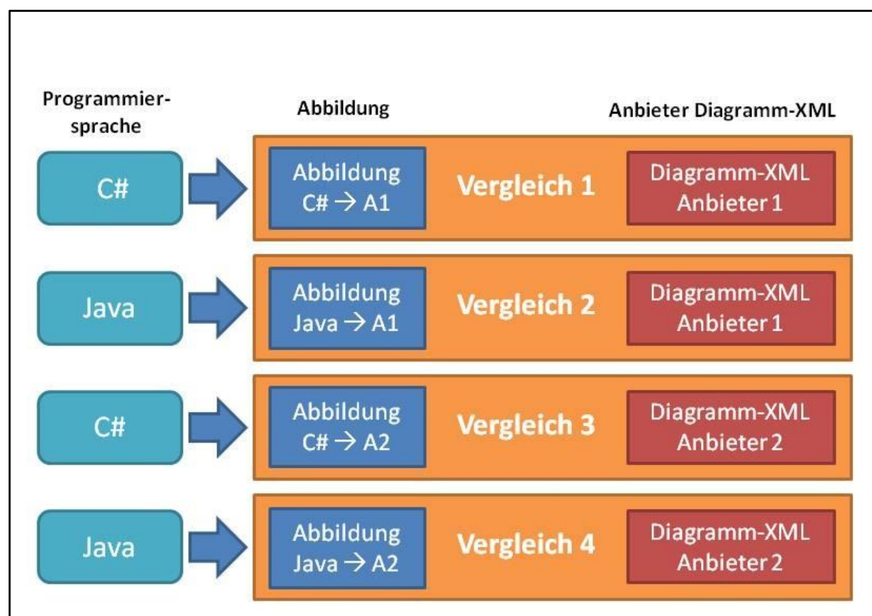

Abbildung 12: Abbildungsproblematik bei direktem XML-Vergleich

Die Abbildung 12 zeigt auf, dass im Beispiel für die Programmiersprachen C# und Java für jeden unterstützen Anbieter ein kompletter Abbildungsapparat implementiert werden muss. Für jeden weiteren Anbieter müssten Abbildungsregeln für den Quelltext definiert werden. Ebenfalls ist es nötigt, für jeden Anbieter neue Vergleichsprogramme zu entwickeln, da die Struktur zu unterschiedlich sein kann. Dies ist bei der Entwicklung und auch bei der Wartung der Werkzeuge ein sehr großer zusätzlicher Aufwand. Ebenfalls ist eine Zerteilung der Entwicklungsaufgaben schwer.

Ein Ansatz, um dieses Problem zu umgehen, ist der Entwurf eines definierten Zwischenformats für die zerlegten Architekturelemente, auf das sowohl der Quelltext als auch das Diagramm abgebildet werden. Hierdurch erreicht man eine Trennung zwischen Anbieterstrukturen und Quelltext und auch einen einzigen Vergleichsalgorithmus pro Diagrammtyp, da alle Informationen auf eine Repräsentation abgebildet werden. Hierdurch wird die Nachvollziehbarkeit des Vergleichs erhöht. Folgendes Schaubild soll dieses verdeutlichen:

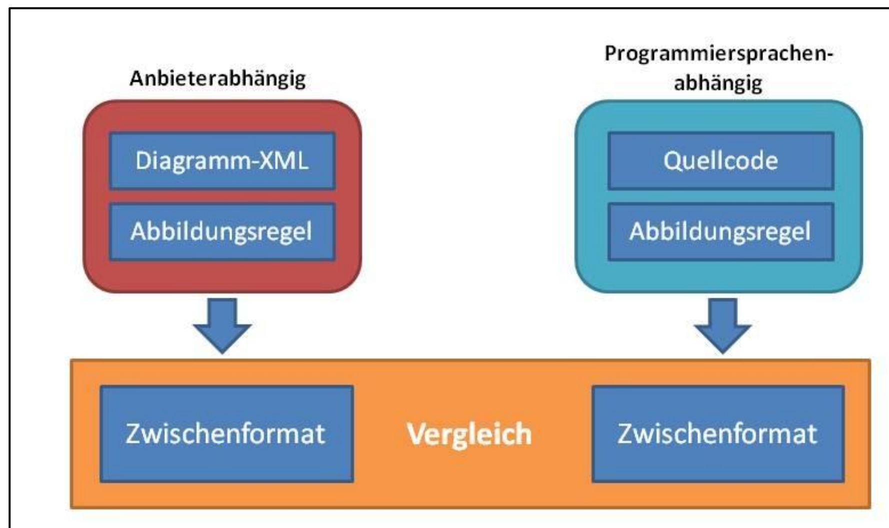

Abbildung 13: Einführung eines Zwischenformats

Der Vergleich des Diagramms mit dem Quelltext geschieht somit über ein vordefiniertes Format, sodass nur eine einzige Vergleichsinstanz für einen Diagrammtypen genutzt werden muss. Soll das Werkzeug erweitert werden um einen Anbieter 2, ist eine Realisierung der Abbildungsregeln für das Diagramm auf das definierte Zwischenformat nötig, die Abbildung für die Programmiersprache kann bestehen bleiben. Ebenso gilt dies für den Fall, dass beliebige Programmiersprachen (hier durch Quelltexte repräsentiert) mittels definierter Abbildungsregeln auf das Zwischenformat abgebildet werden können, ohne dass sich an der Implementierung der Diagrammabbildung etwas ändert. Dies ist zum einen für die Nachvollziehbarkeit und zum anderen für die Erweiterbarkeit ein entscheidender Vorteil.

6.6 Vergleichsoperationen und Repräsentation der Ergebnisse

Nachdem nun eine Lösung für die Abbildung und die Herstellung der Vergleichbarkeit gefunden wurde, in der sowohl eine Repräsentant des Quelltextes als auch des Eingabediagramms abgebildet sind, können diese Zwischenformate miteinander verglichen werden. Dabei ist wichtig, das Zwischenformat so zu wählen, dass eine einfache und nachvollziehbare Vergleichbarkeit erreicht wird, die automatisiert durchgeführt werden kann. Die Ergebnisse des Vergleichs sollen den Anwendern des Werkzeugs zur Verfügung gestellt werden, und zwar in Form eines Diagramms, das dem Eingabediagrammtypen entspricht. Eine Abbildung des Diagramms auf das Zwischenformat ist an dieser Stelle bereits geschehen. Die gleichen Regeln können nun auch angewandt werden, um aus einem Zwischenformat ein Diagramm herzuleiten, da die Abbildungsregeln bidirektional sind. Dies liegt darin begründet, dass exakte Abbildungen ohne Vermengung mit nicht unterstützten Informationen stattfanden. Lediglich die ignorierten Informationen, die bei der Erstellung der Abbildungsregeln definiert wurden lassen sich an dieser Stelle nicht mehr herstellen und können anschließend auch nicht als Elemente im Ergebnis dargestellt werden. Somit ist eine geeignete Repräsentation der Vergleichsergebnisse erneut das gewählte Zwischenformat,

da sich aus diesen für die nachfolgende Verarbeitung das Ergebnisdiagramm herleiten lässt.

Der Vergleichsalgorithmus arbeitet somit auf dem Zwischenformat, es werden im gesamten Analyseablauf dieser Arbeit somit folgende Zwischenformate benötigt:

- Zwischenformat als Repräsentation der Ist-Architektur
- Zwischenformat als Repräsentation der Soll-Architektur
- Zwischenformat als Repräsentation der Vergleichsergebnisse (Ergebnis-Zwischenformat)

Da diese Zwischenformate für den Quelltext und das Diagramm strukturgleich sein sollen, können nun die einzelnen Elemente direkt miteinander z.B. über die Namen verglichen werden. So ist in einem Schichtendiagramm zu ermitteln, ob eine Schicht aus der Soll-Architektur ebenfalls in der Ist-Architektur vorhanden ist. Ist dies der Fall, wird das Ergebnis in das Ergebniszwischenformat eingefügt und als korrekt (über den Wert „isOK“) markiert. Gibt es keinen Repräsentanten, muss ermittelt werden, ob dieser in der Soll- oder in der Ist-Architektur fehlt. In diesem Fall ist das Ergebnis dieser Analyse ebenfalls in dem Ergebniszwischenformat einzufügen. Dazu müssen die Vergleichsergebnisse gespeichert werden, für das gewählte Beispiel als eine nicht korrekte Abbildung der Ist- auf die Soll-Architektur. Dabei ist der Vergleichswert „isOk“ mit „false“ zu belegen und eine Fehlerbeschreibung zu erstellen.

Folgendes Bild soll dies verdeutlichen:

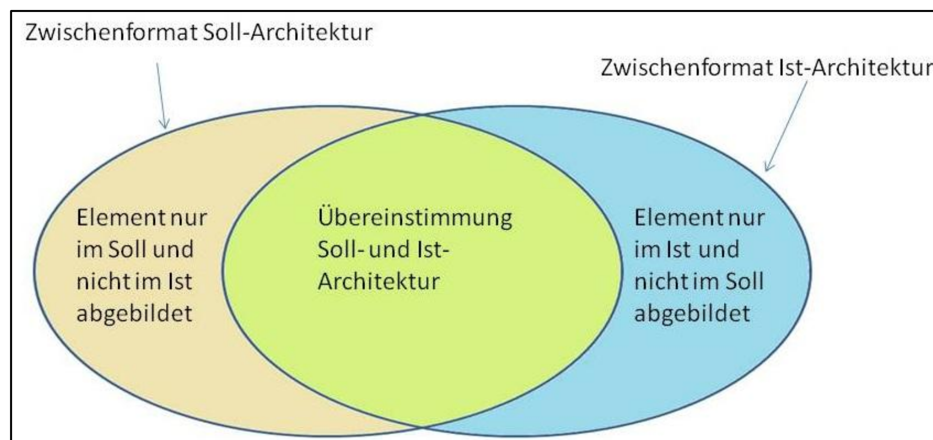

Abbildung 14: Mengenbeschreibung Vergleichsergebnisse

Abbildung 14 zeigt die drei beschriebenen Mengen auf. Der grüne Bereich beschreibt die Elemente, die sowohl in der Soll- als auch in der Ist-Architektur vorkommen und übereinstimmen, diese sind als korrekt zu kennzeichnen. Der linke Bereich beschreibt Elemente, die nur in der Soll-Architektur modelliert, aber nicht in der Ist-Architektur implementiert wurden. Diese Elemente sind entsprechend in das Ergebnis-Zwischenmodell zu überführen und als fehlerhaft zu kennzeichnen. Analog ist ein Vorgehen für den blauen Bereich, in dem Elemente enthalten sind, die nur in der Ist-Architektur aber nicht in der Soll-Architektur enthalten sind, anzuwenden.

Hier trägt der Vorteil, ein einheitliches Zwischenformat gewählt zu haben, da nur eine einzige Implementation des Vergleichsalgorithmus für ein Zwischenformat nötig und somit unabhängig von der untersuchten Programmiersprache oder dem Erstellungswerkzeugs des Eingabediagramms ist.

Nachdem nun der Vergleich stattfand und die Ergebnisse inklusive Beschreibung bei Abweichungen vorliegen und nachdem für jedes Element festgelegt wurde, ob es eine positive Übereinstimmung gab oder nicht, sollen die Ergebnisse für den Anwender erneut grafisch dargestellt werden. Hierzu soll in diesem Ansatz ein Diagramm vom selben Typ wie die Repräsentation der Soll-Architektur gewählt werden, z.B. ein Schichtendiagramm bei entsprechender Analyse. In diesem Ergebnisdiagramm sollen die Übereinstimmungen und Abweichungen, bei Abweichungen auch die Fehler-Information, grafisch erkennbar sein. Hierzu sollen Farben zur Unterstützung der Anwender genutzt werden:

- Rote Einfärbung des Elements: Ist entweder in der Ist- oder Soll-Architektur nicht enthalten, inklusive Beschreibung in welcher es vorkommt
- Grüne Einfärbung des Elements: Alle Informationen inklusive Unterelementen stimmen komplett überein
- Gelbe Einfärbung: das Grundelement, z.B. eine Klasse ist sowohl in der Ist- als auch in der Soll-Architektur enthalten, aber mindestens ein Unterelement, z.B. Attribut, stimmen nicht überein

Als Vorlage für den Export des Ergebnisdiagramms gelten die Abbildungsregeln, die bereits für die Übersetzung des Eingabediagramms in das Zwischenformat genutzt wurden. Da diese bidirektional, sprich auch für die Abbildung des Zwischenformats auf den Diagrammtypen, anwendbar sind, kann über diesen Mechanismus eine Exportfunktionalität durch XML-Operationen realisiert werden. Die entsprechende Information, z.B. den Namen eines architekturbeschreibenden Elements, ist dem Ergebniszwischenformat zu entnehmen und an der entsprechenden Stelle im XML-Export einzufügen. Hierzu müssen anbieterspezifische Exportfunktionen realisiert werden, da die Struktur der Dateien unterschiedlich ist. Anschließend kann das exportierte Diagramm aus dem Analysewerkzeug in das Werkzeug für die Modellierung importiert und die Ergebnisse durch Anwender betrachtet und ausgewertet werden.

6.7 Fazit zur Analyse

In diesem Kapitel werden die Problemfelder dieses Ansatzes analysiert und Verfahrensweisen für den Umgang mit diesen aufgezeigt. In diesem Ansatz gibt es folgende vier Stufen, die für eine modellbasierte Softwarearchitekturanalyse durchlaufen werden müssen:

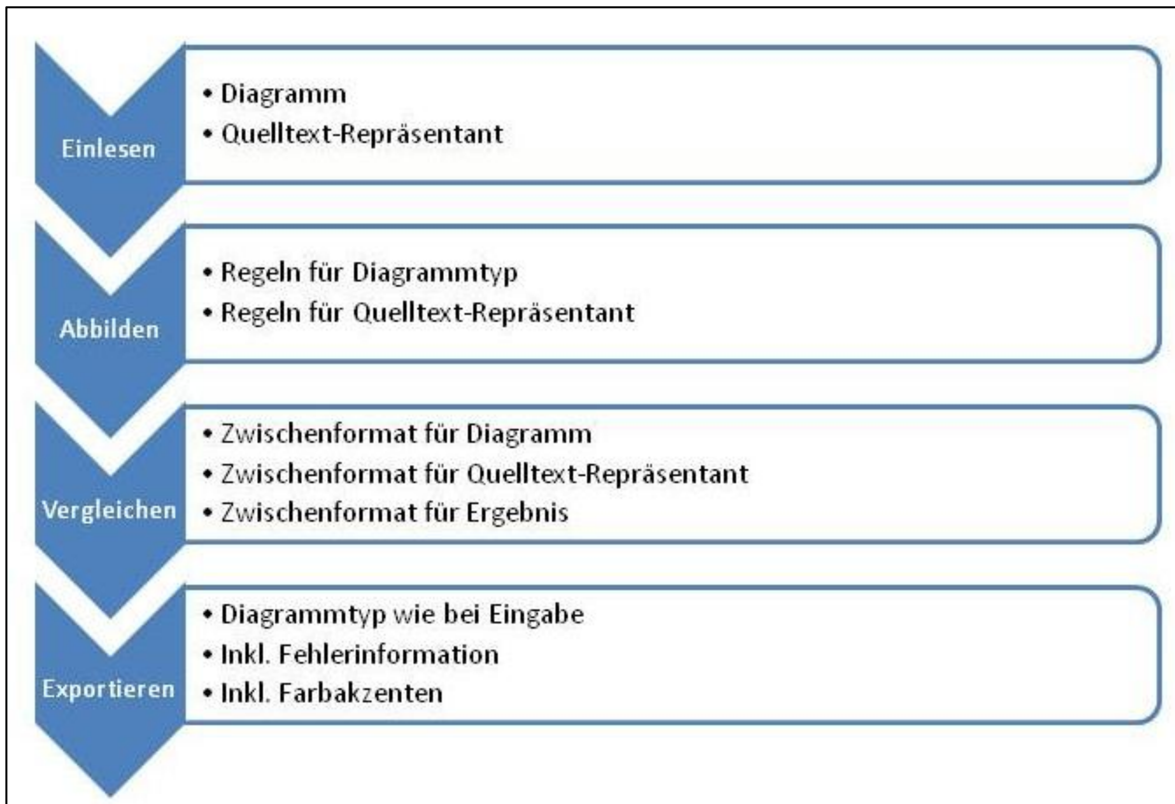

Abbildung 15: Ablauf der Softwarearchitekturanalyse

Im ersten Schritt, dem Einlesen, wird davon ausgegangen, dass für die Diagramme ein XML-basiertes Format genutzt wird, sodass dieses über XML-Parser in den Programmfluss eingelesen und benötigte Informationen extrahiert werden können. Für den Quelltext wird in dieser Arbeit nicht der Ansatz eines Parsers für die jeweilig zu untersuchende Programmiersprache zur Untersuchung und Extraktion genutzt, da keine geeigneten und freien Parser zur Verfügung standen. Stattdessen wird eine Programmierschnittstelle namens Reflections genutzt, die für verschiedene Programmiersprachen zur Verfügung steht und den Zugriff auf die statischen Aspekte des der Software zugrundliegenden Quelltextes ermöglicht. Mittels dieses Mechanismus können die relevanten Informationen extrahiert werden.

Um die Abbildbarkeit und somit die Herstellung einer kompletten Ist-Architektur nach Definition unter Kapitel 3.1.1, herzustellen, ist definiert worden, dass zunächst die architekturbeschreibenden Elemente, basierend auf den Elementen des Eingabediagramms, gefunden werden müssen. Danach ist es möglich, Repräsentanten im Quelltext für diese Elemente zu finden. Diese sind als Abbildungsregeln festzuhalten, und bilden die Grundlage für eine automatisierte Abbildung des Quelltextes auf ein

Modell. Die Herstellung der Abbildungsregeln ist ein manueller Aufwand, der nicht direkt automatisiert wurde. Dieses wird benötigt, um eine Vergleichbarkeit herzustellen, weil nach MDA eine Vergleichbarkeit zwischen Modellen möglich ist, wenn diese auf derselben Modellierungsklassifizierung einzuordnen sind. Hierzu soll in diesem Ansatz ein Heraufheben des Quelltextes zu einem Diagramm stattfinden, das in einem nächsten Schritt dann mit dem Eingabediagramm verglichen wird.

Dazu ist zunächst der zweite Schritt im Ablauf, die Abbildung, nötig. Hierzu wurde definiert, dass ein geeignetes Zwischenformat für die Abbildung, den Vergleich und auch den Export nötig ist, damit eine Nachvollziehbarkeit und Erweiterbarkeit möglich wird. In diesem zweiten Schritt finden somit sowohl die Abbildungsregeln für den Quelltext als auch für das Diagramm auf das Zwischenformat Anwendung.

Im dritten Schritt werden die vorliegenden Zwischenformate des Quelltextes und des Diagramms miteinander verglichen, und die Ergebnisse und Fehlermeldungen zwecks Überführung zurück in ein Diagramm desselben Typs wie das Eingabediagramm erneut in dem genannten Zwischenformat gespeichert.

Im vierten und letzten Schritt ist es möglich, die Abbildungsregeln des Diagramms auf das Zwischenformat invers anzuwenden, sodass das Zwischenformat mit den Ergebnissen des Vergleiches in ein Diagramm überführt werden kann. So werden den Anwendern die Ergebnisse inklusive grafischer Repräsentationen verfügbar gemacht. Hierzu sind erneut XML-Operationen nötig.

7 Design und Systemarchitektur

Das Kapitel Design und Systemarchitektur umfasst die Ermittlung der Softwarekomponenten, die für eine modellbasierte Softwarearchitekturanalyse notwendig und umzusetzen sind. Darüber hinaus wird aus diesen Komponenten eine Systemarchitektur für die gesamten Prototypen entwickelt.

7.1 Realisierung des Zwischenformats und Definition der Abbildungsregeln

In Kapitel 6 wird zwecks Abbildbarkeit, Vergleichbarkeit und auch für den Export der Ergebnisse ein definiertes Zwischenformat gefordert. Da sowohl ein Vergleichsmechanismus programmiert werden muss, als auch eine Abbildung des Quelltextes und der Diagramme anhand der definierten Regeln automatisiert geschehen soll, wird für die Repräsentation des Zwischenformats ein statisches Klassenmodell gewählt. In den Schritten zur Erstellung der Abbildungsregeln wurden bereits die architekturbeschreibenden Elemente definiert. Diese einzelnen Elemente und deren Verbindungen untereinander können in einem Klassendiagramm dargestellt werden und dienen somit drei Zwecken:

1. Grundlage für die Abbildungsregeln des Quelltextes auf ein Diagramm in Form von zu identifizierenden Repräsentanten im Quelltext
2. Grundlage für das zu programmierende Abbildungsformat für den Vergleicher
3. Grundlage für das zu exportierende Ergebnisdiagramm der gesamten Analyse

In den Anforderungen dieser Arbeit wurde festgehalten, dass C# als Programmiersprache verwendet wird und Klassen-, Komponenten- und Schichtendiagramme als Diagrammtypen untersucht werden sollen. Wie unter Kapitel 6.4 beschrieben, ist für den in dieser Arbeit verfolgten Ansatz zur modellbasierten Softwarearchitekturanalyse eine Abbildung des Quelltextes bzw. der Programmiersprachenelemente auf die architekturbeschreibenden Elemente nötig. Dazu ist der jeweilige Diagrammtyp in die Elemente zu zerlegen, welche die relevanten und für die Architekturanalyse zu untersuchenden Elemente darstellen. Dies ist bereits für die Erstellung der Klassenmodelle der Zwischenformate für die genutzten Diagrammtypen nötig. Anschließend wird durch die Anwender definiert, wie diese Elemente in der jeweiligen Programmiersprache dargestellt oder identifiziert werden können. Wie bereits beschrieben, gibt es für das Auffinden der Repräsentanten oftmals mehrere Varianten und Einschränkungen, die berücksichtigt werden müssen.

Diese Klassenmodelle des Zwischenformats können direkt in repräsentative Programmierklassen überführt werden und automatisiert bearbeitet werden. In diesen Klassen werden somit die Ergebnisse der Abbildungen gespeichert, die Vergleichsoperation wird ausgeführt und anschließend in ein Ergebnisdiagramm exportiert. Eine direkte Einbettung in den Programmfluss ist möglich und erleichtert somit

den Umgang mit diesem Format für alle darauf basierenden Arbeitsschritte des Analysewerkzeugs.

In den folgenden Abschnitten sollen sowohl die Zwischenmodelle als auch die Abbildungsregeln für die verschiedenen angewandten Diagrammtypen und die Programmiersprache C# aufgezeigt werden.

7.1.1 Schichtendiagramme & C#

Die Schichtendiagramme weisen die einfachste Darstellungsform aller hier untersuchten Diagrammtypen auf. Es werden lediglich die Schichten inklusive Namen und die Verbindungen zu anderen Schichten dargestellt, mehr Informationen sind nicht enthalten, vgl. Abb. 16.

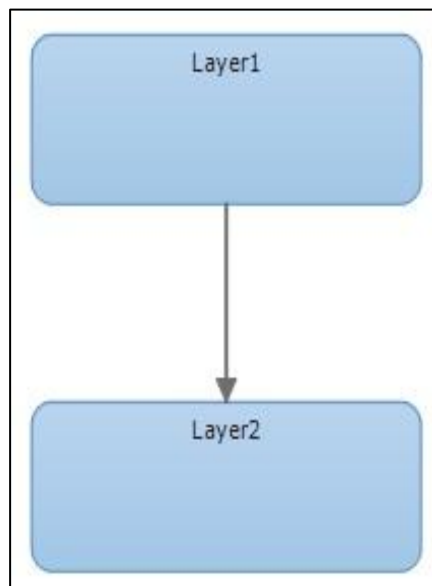

Abbildung 16: Einfaches Schichtendiagramm

Basierend auf diesen Informationen wurde ein Zwischenmodell wie in Abbildung 17 definiert, welches diese Angaben enthält.

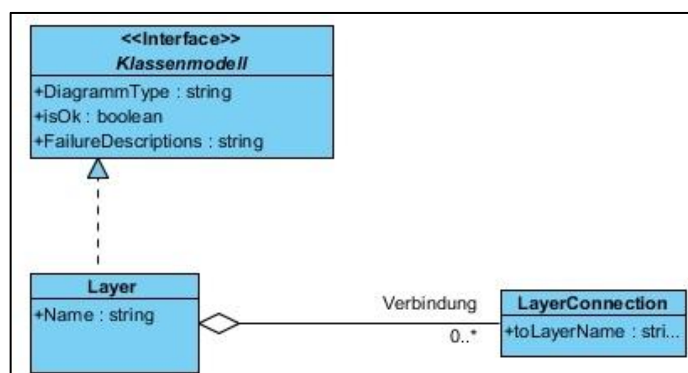

Abbildung 17: Zwischenmodell für Schichtendiagramme

Die wiederkehrende Klasse „AbstraktKlassenmodell“, von der die anderen Elemente erben, stellt eine Sammlung grundlegender Informationen dar, die jede Klasse implementieren muss, um die Ergebnisse des Vergleichs zu beinhalten. Hierbei handelt es sich um Fehlerbeschreibungen und ein XML-Tag, ob der Vergleich ein positives oder negatives Ergebnis darstellt.

Nun müssen die Repräsentanten für Schichten und die Verbindungen zwischen Schichten im Quelltext gefunden werden. In C# gibt es keine direkten Repräsentanten mit der Semantik einer Schicht, daher mussten Alternativen gesucht werden. Hierbei gibt es drei grundlegenden Varianten:

- Definition über Namespaces
- Definition über Namenskonventionen
- Definition über die zugrundeliegenden Verzeichnisstruktur

Da für diese Ausarbeitung keine Vorgaben für die Namenskonventionen und Verzeichnisstruktur erstellt wurden, wird die Definition über Namenspaces als Variante verwendet. Daraus ergeben sich folgenden Abbildungsregeln:

Klassenmodell Entität	Attribut	Variante: Namespace	Anmerkung:
Layer	**Name**	Namespacenamen	z.B. für Klasse A der Namespace "<Projektname>.Schicht1"
LayerConnction	**toLayerName**	Namespacename des referenzierten Typs	z.B. hat die referenzierte Klasse den Namespace: "<Projektname>.Schicht2"

Tabelle 1: Abbildungsregeln Schichtendiagramm -> C#

Mittels dieser Abbildungsregeln kann eine automatisierte Anwendung in einer prototypischen Realisierung stattfinden.

7.1.2 Klassendiagramme & C#

Klassendiagramme und die objektorientierte Programmiersprache C# sind strukturähnlich, da C# ebenfalls wie Klassendiagramme auf einer Klassenkonstruktion beruht. Aufgrund der größeren Anzahl an möglichen Modellierungselementen sind die Abbildungsregeln für C# auf Klassendiagramme umfangreicher als bei den Schichtendiagrammen. Klassendiagramme sind in geschachtelte Datenstrukturen zu zerlegen, bei denen die Klasse die oberste Struktur darstellt, die wiederum Attribute, Methoden und Verbindungen enthält. Eine Verbindung ist erneut zerlegbar in Vererbungs-, Realisierungs- und allgemeine Verbindungen. Jede dieser genannten Strukturen umfasst weitere Informationen, wie z.B. den Attributnamen, oder ob es sich um eine statische Methode handelt. Die Ergebnisse der Zerlegung sind in der folgenden Abbildung als Klassenmodell und somit als Zwischenmodell, wie in dieser Ausarbeitung beschrieben, zusammengefasst:

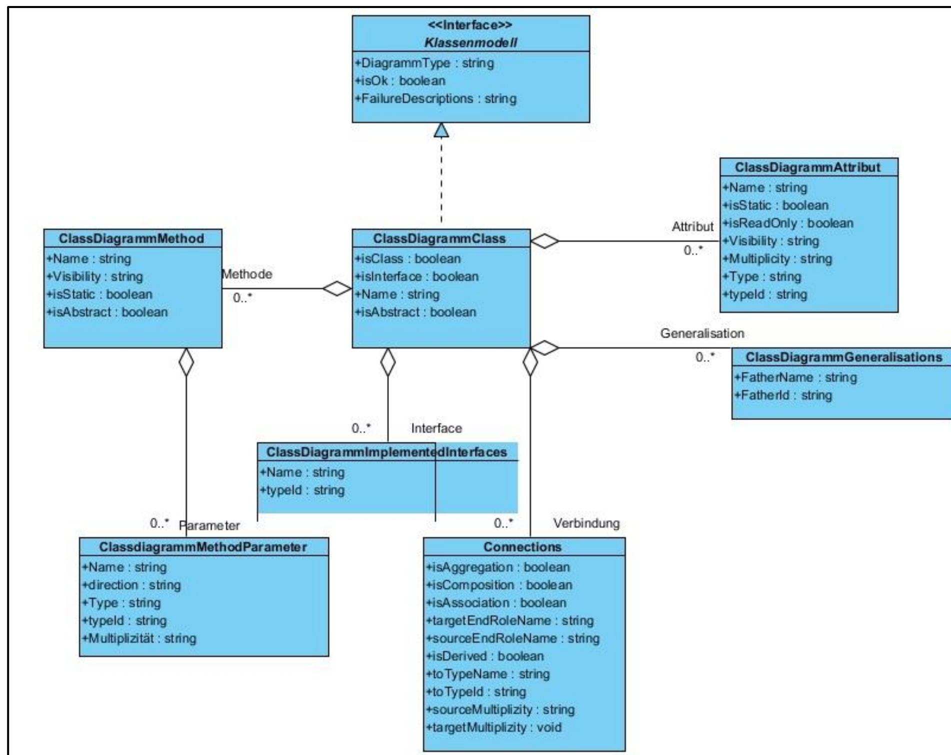

Abbildung 18: Zwischenmodell für Klassendiagramme

Nachdem nun die Zerlegung des Klassendiagramms in die zu untersuchenden architekturbeschreibenden Elemente und die Überführung in ein Zwischenmodell stattgefunden hat, sind Abbildungsrepräsentanten für die einzelnen Elemente in C# zu finden und zu definieren. Es wurden insgesamt 33 zu identifizierende Elemente definiert. Folgender Tabellenauszug stellt das Mapping auf die Zwischenformats-Klasse „ClassDiagramClass" dar:

Klassenmodell Entität	Attribut	Variante: Reflections	Anmerkung/Beispiel:
ClassDiagrammClass	**Name**	Name der aktuell Untersuchten Klasse	z.B. über Reflection: Type.Name()
ClassDiagrammClass	**isClass**	Tag abfragen	type.isClass, boolean
ClassDiagrammClass	**isInterface**	Tag abfragen --> type.isInterface	boolean
ClassDiagrammClass	**isAbstract**	Tag abfragen --> Ttype.isAbstract	boolean

Tabelle 2: Klassenabbildungsregeln für Klassendiagramme auf C#

All diese dargestellten Abbildungsregeln können direkt über die C# API „Reflections" am Quelltext abgefragt werden. Es gibt auch Elemente, die nicht direkt über Reflections erfragt werden können. Hierzu gehören z.B. die Sichtbarkeit (Visibility) und die Multiplizität von Attributen und Methoden in Klassendiagrammen. Um diese Elemente zu ermitteln, müssen andere Informationen, die mittels Reflections zugänglich sind, aus dem Quelltext genutzt werden.

Im Fall der Sichtbarkeit eines Attributs oder einer Methode wird in Reflections eine Abfragemöglichkeit für „public" und „private" zur Verfügung gestellt. Das Keyword „protected" ist nicht angeboten. Hier müssen entweder andere Varianten gefunden oder definiert werden, da „protected" nicht in diesem Analysewerkzeug untersucht werden

kann. Für die Erstellung eines Prototypen wird eine Variante verwendet, in der „public“ und „private“ Modifikatoren über eine boolesche Variable abgefragt werden. Bei dem Wahrheitswert „true“ ist der entsprechende Wert (public oder private) zu übernehmen, sind beide Werte „false“ wird die Sichtbarkeit als „protected“ definiert. Es gibt noch weitere Sichtbarkeitsmöglichkeiten in C#, z.B. „internal“, diese werden aber nicht in den Klassendiagrammen unterstützt und sind somit für die Analyse nicht von Relevanz.

Im Fall der Multiplizität soll geklärt werden, ob ein Attribut oder Parameter einer Methode eine Mehrzahl an Objekten eines Typs enthalten kann. Dabei ist es für die Untersuchung nicht relevant, ob auch tatsächlich mehrere Objekte enthalten sind oder ob es nur möglich ist, mehrere Objekte zu enthalten. Um an die genaue Anzahl an gehaltenen Objekten eines Typs zu gelangen müssten alle Varianten und Schleifen für den Ablauf der Software untersucht werden. Dies ist jedoch ein sehr großer Aufwand, falls es aufgrund der Komplexität überhaupt zu lösen ist. Daher wurde für die Analysezwecke definiert, dass es nur eine Unterscheidung zwischen den Multiplizitäten von „1“ (ein Typ repräsentiert ein Objekt) oder „0..*“ (ein Objekt kann eine Mehrzahl an Objekten enthalten, die genau Anzahl ist jedoch nicht bekannt) gibt. Es gibt in Reflections einen booleschen Wert, der angibt, ob der jeweilige Typ ein Array („isArray“) ist oder nicht. Bei einem Wahrheitswert von „true“ können in diesem Typ eine Mehrzahl von Objekten gespeichert werden. Im Fall des Wahrheitswertes „false“ handelt es sich nicht um ein Array. Im C# Namensraum „System.Collection“ werden jedoch komplexe Datenstrukturen wie Listen, Maps oder Dictionaries angeboten, die ebenfalls eine Mehrzahl von Objekten speichern können. Es ist jedoch keine Abfrage in Reflections enthalten, die prüft, ob es sich um eine Collection handelt oder nicht. Um dieses Problem zu umgehen und definieren zu können, ob ein Objekt eine Collection ist, kann der Namespace des jeweiligen Typs untersucht werden. Enthält dieser Namespace das Wort „System.Collection“, wurde aus diesem Namespace geerbt. Somit sind sowohl die Information, ob es sich um ein Array (über isArray) handelt oder im Namespace das Wort „System.Collection“ enthalten ist, für die Entscheidung der Multiplizität relevant. Im Folgenden werden die soeben behandelten Abbildungsregeln tabellarisch aufgeführt:

ClassDiagrammAttribute	**Visibility**	Tag isPublic, isPrivate abfragen	Protected kann nicht direkt abgebildet werden
ClassDiagrammAttribute	**Multiplizität**	isArray abfragen und dazu noch prüfen, ob KeyWord "Collection" im Typennamespace vorkommt	isArray oder System.Collection im Namespace, dann Multiplizität 0..* sonst 1, Problemfall: Generics, hier können verschachtelte Typen auftauchen --> Umgang: wenn Mischung aus nativen und fachlichen Typen, dann nur auf fachliche beziehen, wenn nur fachliche, dann jede als eigene Collection betrachten

Tabelle 3: Auszug Abbildungsregeln für Attribute aus Klassendiagramme

Wie unter Kapitel 6.2 aufgezeigt, wird in C# keine Unterscheidung in den Verbindungstypen gemacht. In Klassendiagrammen wird, außer für Vererbungen und Realisationen, die Verbindung zwischen zwei Klassen unterschieden in Assoziation, Aggregation und Komposition. Hier wurden keine Möglichkeiten zur Identifizierung im Quelltext für die Unterscheidung der verschiedenen in Klassendiagrammen enthaltenen Verbindungarten gefunden, die nicht eine sehr hohe Anforderung an den

Entwicklungsprozess stellen. Es soll jedoch nicht das Ziel dieser Arbeit sein, Vorgaben für die Entwicklung von Softwaresystemen zu stellen oder Einschränkungen im Entwicklungsprozess zu definieren. Daher wurde dieses Problem als nicht identifizierbar festgelegt und alle Verbindungen zwischen Klassen entsprechen Assoziationen. Aggregationen und Kompositionen werden in der Analyse ebenfalls wie Assoziationen behandelt.

7.1.3 Komponentendiagramme & C#

Komponentendiagramme eignen sich nach der Definition einer Softwarearchitektur zur Darstellung einer Struktur von Strukturen. Dies lässt sich damit begründen, dass hier austauschbare Komponenten, deren Schnittstellen und Verbindungen modelliert werden. Es besteht eine Abstraktion vom Groben (äußere Sicht) zum Feinen (konkreter innerer Aufbau).

Genau diese Eigenschaften spiegeln sich auch in der Erstellung eines Zwischenmodells für Komponentendiagramme wieder. Das zentrale architekturbeschreibende Element ist hier eine Komponente, die eine äußere und eine innere Sicht darstellt. Die innere Sicht soll von der Außenwelt gekapselt sein, und somit keine direkten Zugriffe von anderen Komponenten ermöglichen oder selbst direkte Zugriffe auf andere Komponenten durchführen. Die äußere Sicht bezieht sich auf Schnittstellen, wobei hier zwischen zwei grundlegenden Typen von Schnittstellen unterschieden wird: die angebotenen (provided) und benötigten (required) Schnittstellen. Angebotene Schnittstellen stellen Dienste einer Komponente der Außenwelt zur Verfügung, die Komponente an sich delegiert Zugriffe dieser Schnittstelle in die innere Sicht.

Benötigte Schnittstellen stellen den Gegenpart zu angebotenen Schnittstellen dar, da eine Komponente diese Schnittstellen benötigt, um ihre Dienste anbieten zu können. Somit kann die Kommunikation zwischen Komponenten über Verbindungen zwischen angebotenen und benötigten Schnittstellen beschrieben werden.

Komponenten können ebenfalls von anderen Komponenten erben und auch anderweitige Verbindungen zu anderen Komponenten enthalten, wie z.B. eine Delegation von Anfragen. Diese Beschreibung findet sich ebenfalls in der Repräsentation des Zwischenmodells wieder. Die soeben beschriebenen Elemente stellen zusammen die externe Sicht einer Komponente dar (vgl. Abbildung 19).

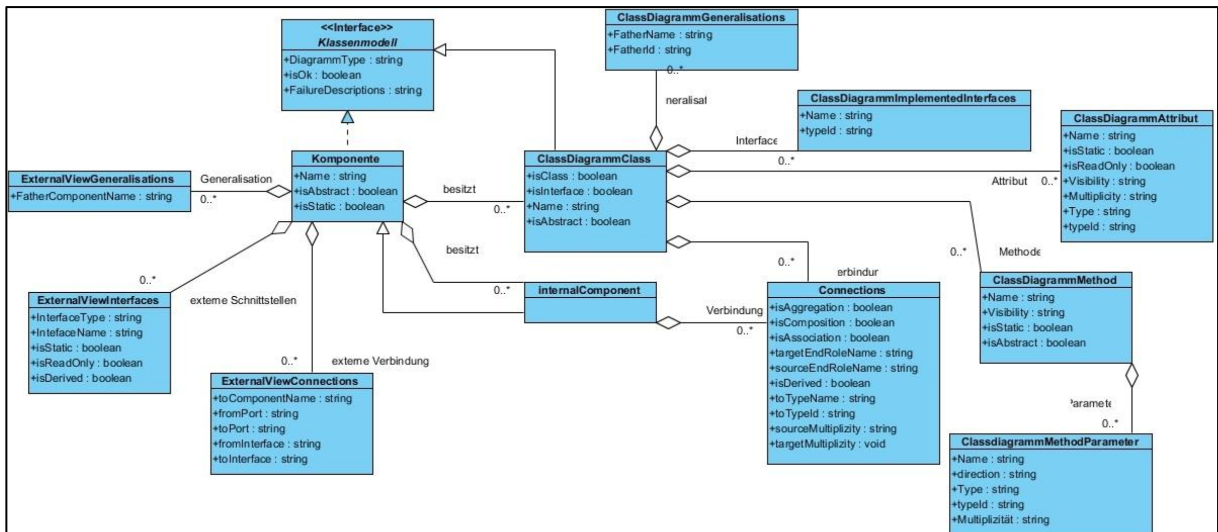

Abbildung 19: Zwischenmodell eines Komponentendiagramms

Die innere Sicht einer Komponente besteht erneut aus Komponenten und Verbindungen zwischen diesen Komponenten. Neben Komponenten in der inneren Sicht können auch einzelne Klassen (vgl. Zwischenmodell aus dem Klassendiagramm aus 7.1.2) an dieser Stelle modelliert sein. Diese rekursive Abhängigkeit und die Variante der Nutzung eines Klassendiagramms wurden entsprechend in das Zwischenmodell überführt.

Bei der Erstellung der Abbildungsregeln wird zunächst nur die externe Sicht behandelt, da der innere Aufbau einer Komponente zum einen aus einem großen Teil aus den Definitionen eines Klassendiagramms (vgl. Abbildungsregeln unter 7.1.2) und zum anderen aus den Abbildungsregeln für die externe Sicht einer Komponente besteht. Dies entspricht auch dem rekursiven Aufbau einer Komponente. C# selbst enthält kein Komponenten- oder Modulsystem, sodass ein direkter Repräsentant einer Komponente nicht direkt zu identifizieren ist. Nach einer Recherche, wie Komponenten erkannt werden können, wurden drei Varianten gefunden:

1. Einsatz des Fassaden-Entwurfsmusters (Gamma, et al., 2004), wobei die Fassade-Klasse die Komponente repräsentiert indem eine Schnittstelle implementiert wird, die sie als Komponente beschreibt
2. Einsatz des Fassaden-Entwurfsmusters, wobei die Fassaden-Klasse als Komponente repräsentiert wird, indem eine Namenskonvention eingesetzt wird
3. Die Ordnerstruktur und Namensgebung der Unterordner stellen die Komponentenstruktur dar

Bei Variante 1 und 2 wird das Fassaden-Muster angewandt, um eine Klasse im Quelltext als Komponente zu beschreiben. Dabei leitet diese Klasse alle eingehenden und ausgehenden Anfragen an die oder von der Komponente weiter, sodass für andere Klassen nur die Fassaden-Klasse als Bezugspunkt genutzt wird. Die Unterscheidung der beiden genannten Varianten liegt darin, dass zur Erkennung, ob es sich bei Variante 1 um eine Fassaden-Klasse einer Komponente handelt, geprüft wird, ob eine Schnittstelle implementiert ist, die diese Zugehörigkeit kennzeichnet. Dieses Vorgehen wird als Marker Interface (Becker, 2008) benannt. Hierbei wird eine Schnittstelle definiert, die keine Methoden oder Attribute enthält, sondern nur über den Namen eine Zugehörigkeit

ausdrückt. Variante 2 beschreibt ein ähnliches Vorgehen, nur wird hierbei kein Marker Interface, sondern eine Namenskonvention für den Einsatz der hier beschriebenen Fassaden-Klasse gefordert, z.B. <Komponentenname>_<Fassadenklassenname>. Die dritte Variante nutzt die Verzeichnisstruktur, in der die Quelltext-Dokumente abgelegt werden. Ordner, bzw. der Name des Ordners, einer bestimmten Tiefe stellen dabei eine Komponente dar. Alle Verzeichnisse und enthaltene Quelltexte darunter stellen dabei Substrukturen der Komponente dar. Da keine Namenskonventionen in dieser Arbeit bzw. bei der Erstellung der zu untersuchenden Software festgelegt wurden, kann Variante 2 nicht angewandt werden. Ebenfalls wurden keine Vorschriften für die Verzeichnisstruktur gefordert, sodass Variante 1 für die weiteren Schritte dieser Arbeit gewählt wird.

Nachdem nun die Variante zu Erkennung einer Komponente festgelegt wurde, können die Abbildungsregeln für Komponentendiagramme auf das in Abbildung 11 beschriebene Zwischenmodell definiert werden. Folgende Tabelle gibt einen Auszug aus diesen Abbildungsregeln wieder:

Klassenmodell Entität	Attribut	Variante: Reflections	Anmerkung/Beispiel:
Komponente	**allgemein**	Implementiert eine beschreibende Schnittstelle	z.B. IComponent
Komponente	**Name**	Name der Fassadenklasse, die die beschreibende Schnittstelle implementiert	type.Name
Komponente	**isAbstract**	Tag abfragen --> Ttype.isAbstract	boolean
Komponente	**isStatic**	Tag abfragen --> Ttype.isStatic	boolean

Tabelle 4: Auszug Abbildungsregeln für Komponentendiagramme auf C#-Reflections

Eine Komponente an sich wird über die Implementierung eines definierten, in diesem Falle „IComponent“, Marker Interface identifiziert. Die allgemeinen Informationen einer Komponente, wie der Name, oder ob es sich um eine abstrakte Komponente handelt, können dann direkt über die entsprechenden String- und booleschen Werte des untersuchten Typs in Reflections erfragt werden. Angebotene Schnittstellen, also ein Verhalten, dass nach außen hin sichtbar ist, können über die neben dem Marker Interface implementierten Schnittstellen per Reflections durch die Abfrage „Type t.getInterfaces()“ ermittelt werden. Schnittstellen an sich haben dieselbe Repräsentation wie in Klassendiagrammen. Benötigte Schnittstellen, also ein Verhalten, dass eine Komponente zur Lauffähigkeit benötigt, ist in diesem Fall durch die in den Konstruktorsignaturen enthaltenen Schnittstellen zur Abbildung auf C# zu erfragen. Hierzu müssen alle Konstruktoren der Fassaden-Klasse bzw. deren Parameter untersucht werden. Handelt es sich um eine Schnittstelle als Parametertyp, so stellt diese eine benötigte Schnittstelle der Komponente dar. Vererbung wird exakt wie in den Klassendiagrammen definiert, mit dem Unterschied, dass die Vererbung von Komponenten nur auf die Fassaden-Klasse angewandt wird. Mit dem hier vorgestellten Zwischenmodell und den hier beispielhaft aufgezeigten Abbildungsregeln soll in der Realisierung gearbeitet werden.

7.2 Systemarchitektur

Basierend auf den vier Stufen aus Abbildung 15, die den Ablauf der Analyse zwischen Soll- und Ist-Architektur beschreiben, wurde eine Systemarchitektur wie in Abbildung 20 erstellt. Diese Systemarchitektur enthält ebenso die Ergebnisse, die in der Analyse in Form von der Definition der Zwischenmodelle und Abbildungsregeln der Ist-Architekturelemente auf die Soll-Architekturelemente ermittelt wurden.

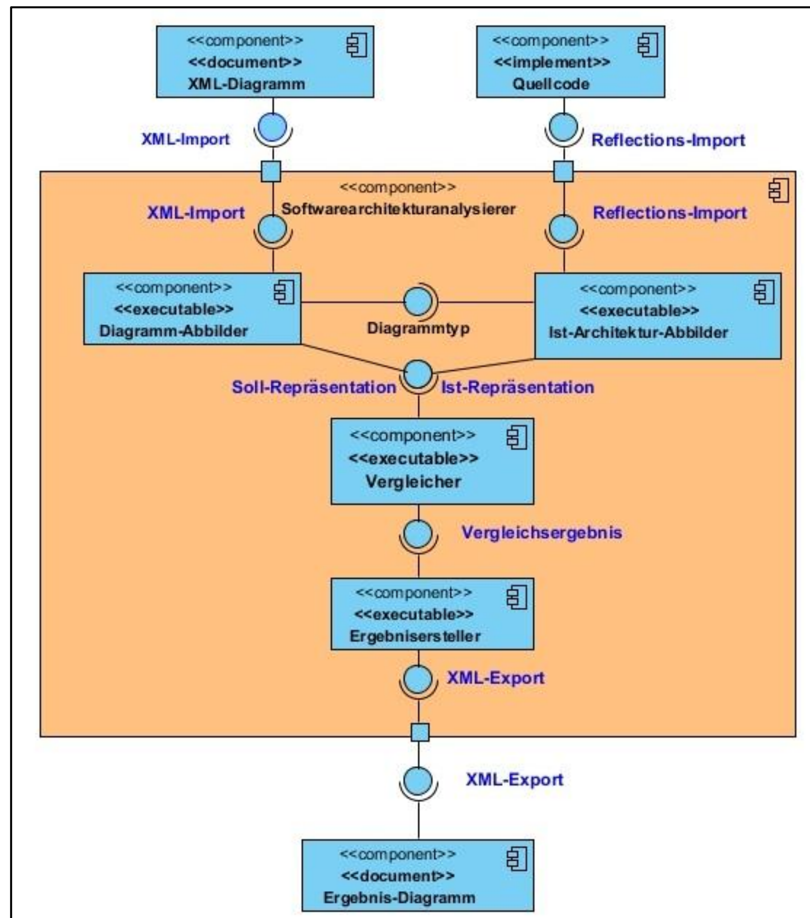

Abbildung 20: Systemarchitektur

Hier sind die einzelnen Komponenten enthalten, die an dem kompletten Ablauf einer Analyse beteiligt sind. Um die enthaltenen Komponenten darzustellen, werden die einzelnen Komponenten in einem kurzen Steckbrief nachfolgend dargestellt.

7.2.1 XML-Diagramm und Quellcode

Ausgehende Schnittstellen: XML-Datei eines Diagramms, Assembly einer Software als Repräsentation des Quelltextes, zugreifbar über Reflections

Eingehende Schnittstellen: keine

Verhalten: Hier werden die Eingabeformate der Analyse und somit die Repräsentanten der Soll- und der Ist-Architektur dargestellt

7.2.2 Softwarearchitekturanalysierer

Ausgehende Schnittstellen: XML-Export für Vergleichsergebnisse

Eingehende Schnittstellen:

1. Diagramm als XML-Import
2. Quelltext als Assembly-Import

Verhalten: Der Softwarearchitekturanalysierer stellt das gesamte Werkzeug dar, durch welches ein Vergleich zwischen einer Soll- und einer Ist-Architektur, wie in dieser Arbeit beschrieben, durchgeführt wird.

7.2.3 Diagramm-Abbilder

Ausgehende Schnittstellen: Abbildung auf programmiertes Klassenmodell des jeweiligen Diagrammtypens

Eingehende Schnittstellen: XML-Dokument eines Diagramms des derzeit untersuchten Typs

Verhalten: Der Diagramm-Abbilder führt die Abbildungsregeln auf das importierte Diagramm durch und speichert das Ergebnis der Abbildung in dem entsprechend realisierten Klassenmodell des Diagrammtypens ab

Interne Repräsentation:

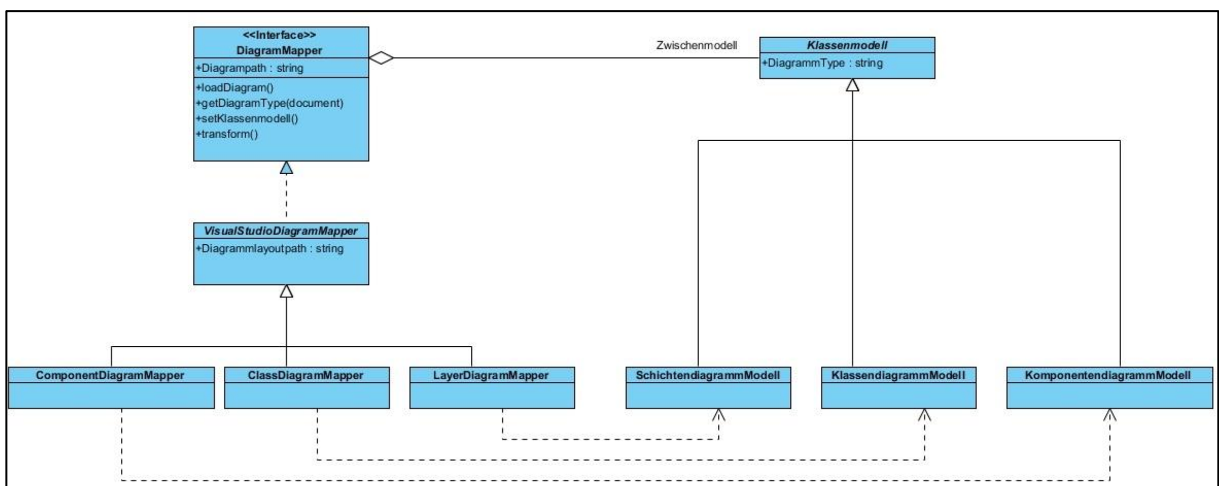

Abbildung 21: Innerer Aufbau der Diagramm-Abbilderkomponente

Der innere Aufbau der Diagramm-Abbilder-Komponente besteht aus einer zentralen Schnittstelle, welche die grundlegenden Methoden eines Diagrammabbilders definiert. Ebenfalls enthält die Schnittstelle eine Referenz auf ein Klassenmodell. Die Klassenmodelle stellen die angebotenen und implementierten Zwischenmodelle dar, in dieser Arbeit werden hier Schichten-, Klassen- und Komponentenmodelle implementiert. Weitere Zwischenmodelle können an dieser Stelle hinterlegt werden. Es ist ebenfalls möglich, mehrere Varianten von Zwischenmodellen für einen Diagrammtypen zu definieren. In diesem Fall muss der Anwender wählen, welches Zwischenmodell genutzt werden soll.

Über die Methode „getDiagramType()" der DiagramMapper-Schnittstelle wird ein Mechanismus gefordert, der den Typen des geladenen Diagramms ermittelt. Anhand dieser Information kann eine Vorauswahl der angebotenen Zwischenmodelle getroffen werden, sodass dem Anwender nur für den geladenen Diagrammtypen kompatible Zwischenmodelle angeboten werden. Die „loadDiagram()"-Methode fordert die Implementierung eines Lademechanismus für Diagramme. Die zentrale Methode „transform()" implementiert bei der Realisierung der Schnittstelle die Abbildungsregeln des Diagramms auf das Zwischenmodell und muss für verschiedenen Diagrammanbieter einzeln implementiert werden. In dem dargestellten Diagramm (vgl. Abb. 21) sind die in dieser Arbeit unterstützen Diagramme aus VisualStudio dargestellt. Für andere Anbieter muss eine Klasse implementiert werden, die eine Schnittstelle „DiagramMapper" implementiert.

7.2.4 Ist-Repräsentation-Abbilder

Ausgehende Schnittstellen: Abbildung auf ein programmiertes Zwischenmodell des importierten Quelltextes über Reflections.

Eingehende Schnittstellen:

1. Assembly, welches mittels Reflections untersucht werden kann
2. Diagrammtyp, damit festgestellt werden kann, welche Abbildungsregeln auf die Assembly angewandt werden sollen

Verhalten: Der Ist-Repräsentation-Abbilder lädt das importierte Assembly und die benötigten Informationen, nämlich um welchen Diagrammtypen es sich beim Import handelt, um die entsprechenden Abbildungsregeln anzuwenden. Anschließend werden die Abbildungsregeln angewandt und via Reflections die benötigten Informationen extrahiert und in dem dazu passenden Klassenmodell gespeichert. Dieses erstellte Klassenmodell ist das Ergebnis der Abbildung und wird über die ausgehende Schnittstelle für die weitere Verarbeitung zur Verfügung gestellt.

Interne Repräsentation:

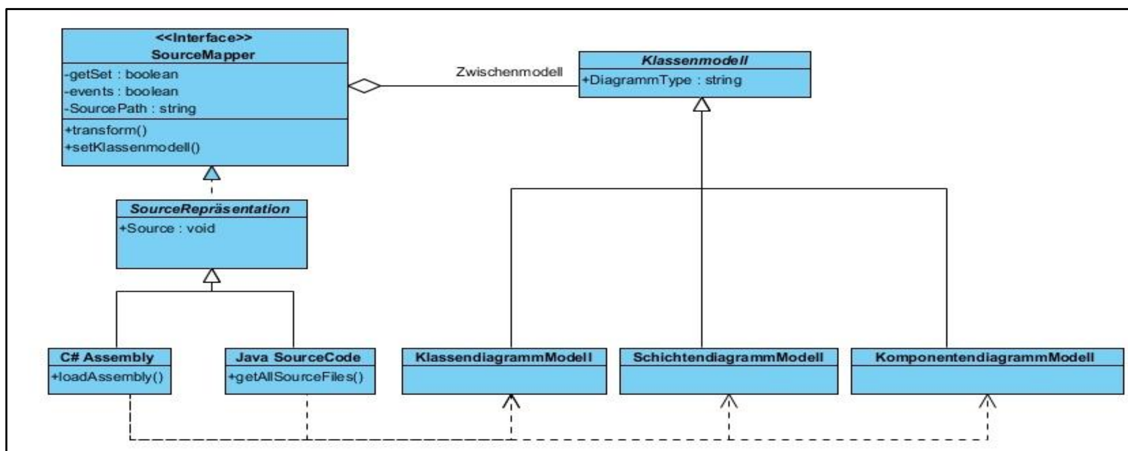

Abbildung 22: Innerer Aufbau der Ist-Architektur-Abbilderkomponente

Der interne Aufbau des Ist-Repräsentation-Abbilders orientiert sich sehr an dem Aufbau des Diagramm-Abbilders. Auch hier gibt es eine Schnittstelle, die alle Ist-Architektur-Abbilder-Implementationen realisieren müssen, die eine Methode „transform()“ beinhaltet. Hier findet für eine konkrete Programmiersprache und ein Zwischenmodell die Implementierung der Abbildungsregeln statt. Die Repräsentation des Quelltextes kann unterschiedliche Varianten enthalten. So wird der Quelltext in dieser Arbeit durch eine Reflections-Assembly repräsentiert, in Ansätzen, in denen ein Parser genutzt wird, müssen alle Quelltextdateien geladen und untersucht werden. Die Realisierung der Abbildungsregeln bezieht sich immer auf ein definiertes Zwischenmodell. Bei einer Änderung am Zwischenmodell oder durch einen Austausch eines solchen müssen entweder die Transformationsregeln angepasst oder ausgetauscht werden.

7.2.5 Vergleicher

Ausgehende Schnittstellen: Das Ergebnis des Vergleichs wird als Klassenmodell für den Export zur Verfügung gestellt.

Eingehende Schnittstellen:

1. Soll-Repräsentation: Das Klassenmodell inklusive der angereicherten Informationen aus dem Diagramm-Abbilder
2. Ist-Repräsentation: Das Klassenmodell inklusive der angereicherten Informationen aus dem Quellcode-Abbilder

Verhalten: Der Vergleicher führt die eigentliche Analyse des Werkzeugs aus. Hier werden die Repräsentation der Soll- und der Ist-Architektur miteinander verglichen. Da die beiden Formate strukturgleich sind, kann jede einzelne Information direkt aus beiden Importen miteinander verglichen werden. Die Ergebnisse des Vergleichs werden inklusive der Fehlermeldungen und der Status, ob die einzelnen Elemente korrekt sind oder nicht, an den Ergebnisersteller weitergeleitet.

Interne Repräsentation:

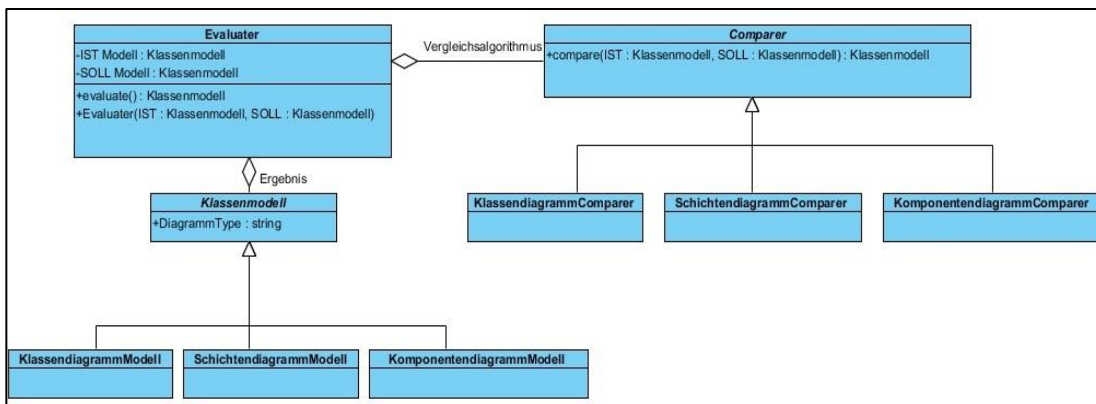

Abbildung 23: Innerer Aufbau der Vergleicherkomponente

Der innere Aufbau des Vergleichers implementiert das Strategie-Entwurfsmuster (Gamma, et al., 2004). Für die angebotenen und unterstützen Zwischenmodelle werden unterschiedliche Vergleichsalgorithmen benötigt. Bei der Erstellung einer Instanz von der Klasse „Evaluater" werden über den Konstruktor die zu untersuchenden Zwischenmodelle der Ist- und auch Soll-Architektur geladen. Diese sind typgleich und müssen somit dasselbe Zwischenmodell darstellen. In der Methode „evaluate()" der Klasse Evaluater wird dann entweder automatisiert oder durch den Anwender der entsprechende Vergleichsalgorithmus gewählt. Die Implementierung des Vergleichsalgorithmus ist in einer der Unterklassen von Comparer enthalten und die gewählte Variante wird im „`Evaluater.evaluate() -> Comparer.compare()`" aufgerufen. Somit ist eine Austauschbarkeit und Flexibilität in der Wahl des Vergleichsalgorithmus hergestellt.

Die Ergebnisse des Vergleichs werden erneut in einem typgleichen Zwischenmodell der Eingabemodelle gespeichert zuzüglich der Vergleichsinformationen. Es muss somit für jede Zwischenmodell-Diagrammart-Kombination, die unterstützt werden soll, eine Strategie-Unterklasse mit den Vergleichsregeln implementiert werden.

7.2.6 Ergebnisersteller

Ausgehende Schnittstellen: Das aus dem Vergleichsergebnis transformierte Diagramm, in Form eines XML-Dokuments

Eingehende Schnittstellen: Das Ergebnis der Vergleichsanalyse aus dem Vergleicher in Form eines Klassenmodells inkl. angereicherter Informationen

Verhalten: Der Ergebnisersteller nutzt die Abbildungsregeln für den eingegebenen Diagrammtyp, um aus dem Ergebnis des Vergleichers ein Diagramm in Form eines XML-Dokuments zu erstellen

Interne Repräsentation:

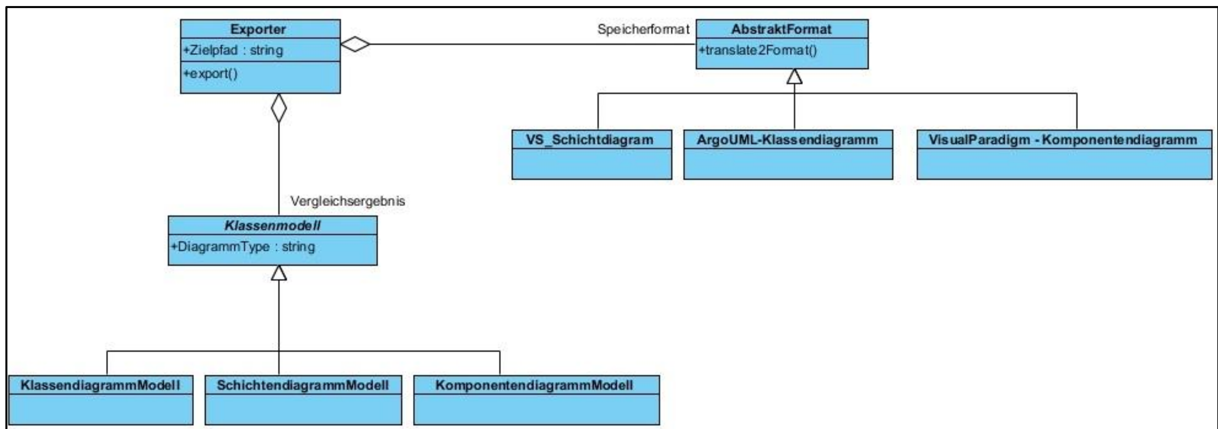

Abbildung 24: Innerer Aufbau der Ergebnisersteller-Komponente

Auch bei dem Ergebnisersteller wird das Strategie-Entwurfsmuster eingesetzt, um die verschiedenen Algorithmen zum Export des XML-Format bzw. der Abbildung des Ergebnis-Zwischenformats hin zu einem Diagramm flexibel und erweiterbar zu halten. Hierbei bietet die abstrakte Klasse „Exporter“ die Methode „export()“ und ein Attribut für den Zielspeicherpfad an. Ebenfalls hält es die Ergebnisse des Vergleichs in Form eines Zwischenformats (Klassenmodell im Diagramm in Abbildung 24) fest.

Die abstrakte Strategie „AbstraktFormat“ bietet die Methode „translate2Format()“ an, die in den abgeleiteten Klassen je nach Anbieterformat und Diagrammtyp die Abbildungsregeln enthält. Es muss somit für jede Anbieter-Zwischenmodell-Kombination, die unterstützt werden soll, eine Strategie-Unterklasse mit den Abbildungsregeln implementiert werden.

7.2.7 Ergebnis-Diagramm

Ausgehende Schnittstellen: keine

Eingehende Schnittstellen: Das XML-Format des Ergebniserstellers

Verhalten: Dies ist die Repräsentation der Ergebnisse, die dann mittels des Anbieterwerkzeuges geladen werden können.

8 Implementierung

In diesem Kapitel werden die für die Realisierung der aufgestellten Ansätze und der aufgestellten Systemarchitektur nötigen Implementierungsschritte beschrieben. Neben einem allgemeinen Überblick werden die besonders schwierigen Stellen der Implementierung hervorgehoben und beschrieben. Die Struktur dieses Kapitels orientiert sich an den Komponenten aus der Systemarchitektur. Das gesamte Werkzeug zur modellbasierten Softwarearchitekturanalyse ist in C# in der Entwicklungsumgebung Visual Studio implementiert. Somit stehen eine Vielzahl an Bibliotheken zur Verfügung.

8.1 Zwischenmodell-Implementierung

Wie in Kapitel 7.1 beschrieben, sind die Zwischenmodelle, in welche die Soll- und Ist-Architektur abgebildet werden sollen, als Klassendiagramme definiert. Somit können diese Zwischenmodelldefinitionen direkt in Programmierklassen überführt werden, da diese kein Verhalten in Form von Methoden enthalten. Abbildung 18 zeigt das für diese Arbeit definierte Klassenmodell für ein Klassendiagramm, der beschriebene Quelltext weiter unten zeigt die Implementierung dieser:

```
    class ClassDiagrammClass: IKlassenmodell
    {

        public bool isClass { get; set; }
        public bool isInterface { get; set; }
        public bool isAbstract { get; set; }
        public string Name { get; set; }
        public List<ClassDiagramAttribute> Attributes { get; set; }
        public List<ClassDiagramGeneralisations> Genralisations { get; set; }
        public List<ClassDiagramImplementedInterfaces> Interfaces { get; set;
}
        public List<ClassdiagramMethod> Methoden { get; set; }
     }
```

Die enthaltenen Klassen

- „ClassDiagramAttribute“
- „ClassDiagramGeneralisation“
- „ClassDiagramImplementedInterfaces“
- „ClassDiagramMethod“

sind ebenfalls direkt aus der Zwischenmodell-Struktur des Klassendiagramms übernommen und implementiert.

8.2 Diagramm-Abbilder

Der Diagramm-Abbilder stellt die Komponente dar, die den Import eines in XML vorliegenden Diagramms, die Erkennung des untersuchten Diagrammtyps und die Abbildung auf das entsprechende Klassenmodell des Diagramms vornimmt. Sowohl für den Zugriff auf XML-Strukturen als auch für das Parsen dieser XML-Struktur ist ein Import der „`System.XML`“-Bibliothek nötig. Das im XML-Format vorliegende Diagramm kann dann über einen entsprechenden Aufruf geladen werden.

```
XmlDocument doc = new XmlDocument();
 doc.Load(diagramPath);
```

Die Variable „`diagramPath`“ repräsentiert dabei den Pfad zu dem Diagramm als String.

Nun muss zunächst erkannt werden, um welchen Diagrammtyp es sich handelt. Dazu muss für jeden Anbieter eines Diagramms ermittelt werden, wie ein Diagrammtyp im XML des Diagramms dargestellt wird. In allen VisualStudio-Diagrammen ist der Knotenname des ersten Tags der Hinweis, um welchen Diagrammtyp es sich handelt und kann wie folgt über den ersten Knoten im XML-Dokument ermittelt werden:

```
doc.FirstChild.Name.Equals("layerModel");                  //Schichtendiagramm
doc.FirstChild.Name.Equals("logicalClassdesignerModel");//Klassendiagramm
doc.FirstChild.Name.Equals("componentModel");             //Komponentendiagramm
```

Nachdem nun festgestellt wurde, welchem Diagrammtyp das Eingabediagramm entspricht, wird das entsprechende Klassenmodell geladen. Mit dem Zugriff auf das XML-Dokument des Diagramms und das entsprechende Zwischenmodell, kann mit der Transformation des Diagramms in das Zwischenmodell begonnen werden. Dazu muss für jede Diagramm-Anbieter-Zwischenmodell-Kombination die Methode „transform()“ implementiert werden. In dieser Ausarbeitung ist als Eingabediagramm-Anbieter ausschließlich Microsofts VisualStudio 2010 gewählt und Klassen-, Schichten- und Komponentendiagramme hieraus modelliert. Ebenfalls wurde für diese Diagrammtypen je eine Realisierung des Zwischenmodells erstellt, vgl. Kapitel 8.1. Somit müssen drei Implementierungen der Klassen „`ComponentDiagramMapper`“, „`ClassDiagramMapper`“ und „`LayerDiagramMapper`“ aus Abbildung 21 erstellt werden, die diese Abbildungsregeln in der Methode „`transform()`“ enthalten. Aufgrund der Strukturähnlichkeit der Formate (XML-Datei und Zwischenmodell) kann hier über einfaches XML-Parsing die entsprechenden Informationen ermittelt werden.

Am Beispiel des Schichtendiagramms sollen alle enthaltenen Schichten des Diagramms ermittelt und deren Namen in dem Zwischenmodell gespeichert werden:

```
XmlNodeList layers = doc.GetElementsByTagName("layer");
foreach (XmlNode node in layers)
{
    string name = node.Attributes.GetNamedItem("name").Value;
    Layer tmp = new Layer(name);

    zwischenmodell.Add(tmp);
}
```

Dieses Codebeispiel zeigt auf, dass alle XML-Knoten, deren Tag-Name „layer“ heißt, in eine „XMLNodeList“ überführt werden. Nach diesem Aufruf sind hier alle Schichtenrepräsentanten als XML-Knoten enthalten. Eine Iteration über diese Liste ermöglicht das Extrahieren des Namens und das Speichern in die Zwischenmodellstruktur „Layer“. Solch ein Layer wird anschließend in der Zwischenmodellrepräsentation „zwischenmodell“ hinzugefügt, in der am Ende der Überführung die Ergebnisse der Abbildung enthalten sind. Um nun noch für die Schichten die Verbindungen zu ermitteln, muss zunächst im XML-Dokument erkannt werden, dass ein Tag mit Namen „layerMoniker“ eine ausgehende Verbindung zu einer anderen Schicht darstellt. Dann kann ähnlich wie im vorherigen Codebeispiel an diese Stelle im XML „navigiert“ und der Name der referenzierten Schicht in das Zwischenmodell aufgenommen werden.

Bei Klassen- und Komponentendiagrammen ist der Ablauf identisch, nur dass mehr Informationen extrahiert werden als die Namen. Hier sind viele Informationen als Attribute in XML-Tags enthalten. Diese XML-Dokumente müssen auf die Struktur hin untersucht werden, damit einzelne architekturbeschreibende Elemente identifiziert und ermittelt werden können. Anschließend kann mit dem XML-Parser an diese Stelle im XML-Dokument navigiert und die Information extrahiert werden.

8.3 Ist-Architektur-Abbilder

Die Komponente des Ist-Architektur-Abbilders umfasst den größten Aufwand in der Implementierung. Hierbei werden fünf Schritte benötigt, um alle Informationen sammeln zu können: das Erkennen von fachlichen Klassen, der Umgang mit der Reflections API, die Erkennung von Verbindungen in C#, das automatische Laden des benötigten Zwischenmodells und die Implementierung der Abbildungsregeln für die genutzten Diagrammtypen. Die Folgenden Unterkapitel geben einen Überblick, wie diese Schritte umgesetzt wurden.

8.3.1 Fachliche Klassenerkennung

Diagramme beschreiben meist einen fachlichen Kontext und stellen dazu fachliche Elemente in den Vordergrund. Bei der Analyse von Programmiersprachen und bereits implementierten Softwarelösungen steht zwar zum einen auch die fachliche Lösung im Vordergrund, zum anderen werden zur Erreichung der fachlichen Lösung aber zusätzliche Klassen eingesetzt oder von der Entwicklungsumgebung definiert, die eine technische Motivation haben. So ist in jedem C#-Forms-Programm eine Klasse namens „Program.cs“ enthalten, die den Start einer Applikation steuert, z.B. das erste Fenster/Form lädt. Solche wiederkehrenden Klassen können ebenfalls fachliche Inhalte beinhalten. Jedoch wird selbst durch die Hersteller empfohlen, dass einige solcher Klassen nicht geändert, sondern so belassen werden sollen, wie die Entwicklungsumgebung sie erstellt hat. Vergleiche (IComponent, 2011). In einem Vergleich mit der Soll-Architektur würde es kaum einen Repräsentanten geben, da solche

Konstrukte Programmiersprachen-abhängig sind und im fachlichen Kontext keine Rolle spielen. Daher ist es ein weiteres Ziel, solche technischen Klassen aus der Analyse herauszuhalten. Dazu soll in Form einer Blacklist definiert werden, welche Klassen prinzipiell nicht in den Vergleich der Architektur einfließen. Diese Liste umfasst in Rahmen dieser Arbeit folgende Klassen:

- Program.cs
- Alle Klassen, die mit *.resx enden (Ressourcen-Dateien)

Darüber hinaus sollen ebenfalls keine Abhängigkeiten zu Bibliotheken untersucht werden. So wird für den Zugriff auf XML-Strukturen die Bibliothek namens „`System.XML`“, genutzt. Auch diese ist technisch motiviert und soll daher nicht in die Analyse mit einfließen, da es sehr unwahrscheinlich ist, dass alle genutzten Bibliotheken in einem Diagramm enthalten sind. Im Umkehrschluss bedeutet dieses Vorgehen, dass nur durch den Entwickler implementierte Klassen untersucht werden sollen. Falls hier ebenfalls technische Klassen enthalten sind, die nicht analysiert werden sollen, können diese über die Ausschlussliste exkludiert werden. Die Ermittlung der zu analysierenden Klassen, im Folgenden „fachliche Klassen“ genannt, ist einer der ersten Schritte in der Analyse des Quelltextes.

In dieser Arbeit wird ausschließlich die Programmiersprache C# als Ist-Architektur-Repräsentant untersucht. Da, wie in Kapitel 6.2 beschrieben, der Einsatz eines Parsers nicht ohne sehr hohen Aufwand möglich ist, wird die Technologie der Reflections API genutzt. Hierzu muss das zu untersuchende Softwaresystem als kompilierte ausführbare Datei (*.exe) oder als Bibliothek (.dll) vorliegen, im Folgenden Assembly genannt. Über folgenden Aufruf kann das Assembly in den Programmablauf geladen werden:

```
Assembly assembly = System.Reflection.Assembly.LoadFile(assemblyPath);
Type[] types = assembly.GetTypes();
```

Mittels des Aufrufs „`assembly.GetTypes()`“ werden alle im Assembly enthaltenen und durch Entwickler implementierte Klassen, im folgenden Typen genannt, aufgerufen und in ein Array geladen. Die Zeichenkette „`assemblyPath`“ stellt hierbei den Pfad des zu ladenden Assembly dar. Damit ist die Extraktion der zu untersuchenden Typen möglich. Aus diesem Array mit Typen muss nun über den Namen ermittelt werden, ob Typen enthalten sind, die sich in der Ausschlussliste befinden. Ist dies der Fall, sind diese Typen aus dem Array zu löschen. Dies ist für alle Analyse-Prototypen unabhängig vom Diagrammtyp notwendig. An dieser Stelle kann davon ausgegangen werden, dass in dem Array „`types`“ nur noch fachliche Klassen enthalten sind, die im Folgenden analysiert und in das entsprechende Zwischenformat überführt werden sollen.

8.3.2 Grundlegender Umgang mit Reflections und Extraktion der Information

Reflections ermöglicht den Zugriff auf die statischen Informationen zu einer programmierten Klasse. Hierzu gehören allgemeine Informationen, z.B. ob es sich um eine statische oder abstrakte Klasse handelt oder um ein Interface. Dazu werden Informationen über die Felder,Properties (entsprechen den Attributen in UML-Klassendiagrammen) und Methoden vermittelt. Bei der Repräsentation von Reflections

handelt es sich um eine rekursive Darstellung. Über folgenden Aufruf wird ein Assembly geladen:

```
Assembly assembly = System.Reflection.Assembly.LoadFile(assemblyPath);
Type[] types = assembly.GetTypes();
```

Mittels einer Iteration über das Array „types“ können nun die einzelnen Typen untersucht werden.
In der untersuchten Ist-Architektur ist der 27te Typ in dem Array „`types`“ die programmierte Klasse „`Kunde`“. Über den Aufruf „`assembly.getTypes()[27].getProperties()`“ erhält man nun den Zugriff auf die implementierten Properties des Typen Kunde. Folgende Abbildung zeigt eine grafische Anzeige über den eben genannten Aufruf:

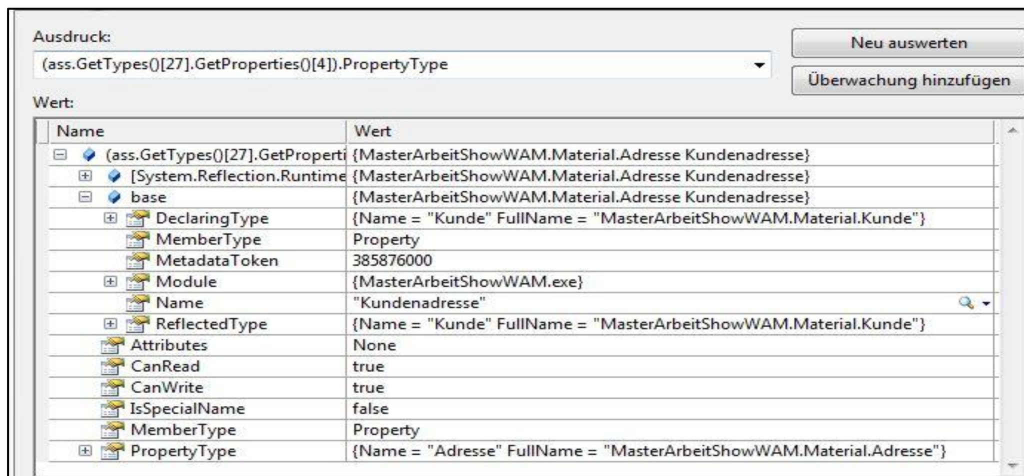

Abbildung 25: Schnellübersicht Reflections in VisualStudio 2010

Über das Feld „`PropertyType`“ kann nun auf den Typ der Kunden-Property „Kundenadresse“ zugegriffen werden. Der Name der Property ist über den Getter „`.Name`“ erreichbar. Über den folgenden Aufruf lässt sich der Typ der Kundenadresse ermitteln:

```
string prop_Name =
assembly.GetTypes()[27].GetProperties()[4].PropertyType.Name;
```

Das Ergebnis dieses Aufrufes ist ein String, der den Namen des Typs enthält, der wiederum Kundenadresse eines Kunden implementiert. Hier ist die Schachtelungstiefe der Methoden recht hoch, was jedoch an dem Aufbau und der internen rekursiven Struktur von Reflections liegt.

- **<u>Rekursiver Type-Aufbau:</u>**
  - Allgemeine Informationen zu einem Typen
  - FieldInfo
    - Allg. Informationen zu einem Field
    - FieldType
      - Type
  - Properties
    - Allg. Informationen zu einer Property
    - PropertyType
      - Type
  - MethodInfo
    - Allg. Informationen zu einer Methode
    - ReturnParameter
      - ReturnParameterType
        - Type
    - ParameterInfo
      - ParameterType
        - Type
    - getMethodBody()
      - LocalVariables
        - Type
  - BaseType
    - Type
  - getInterfaces()
    - Type[]
  - getConstructors()
    - ConstructorInfo
      - Type

Über diese aufgezeigte Struktur kann mittels Reflections navigiert und somit die darin enthaltene Information extrahiert werden. Auf dieser Basis sind auch die Abbildungsregeln auf die Zwischenmodelle erstellt.

8.3.3 Grundlegender Umgang mit der Verbindungserkennungen bei C#-Klassen

Verbindungen zwischen Klassen werden in Reflections nicht explizit dargestellt, es gibt keinen direkten Repräsentanten. Wie bereits diskutiert, sollen allgemein nur die ausgehenden Verbindungen eines Typs untersucht werden. Diese entsprechen dem Zugriff auf andere Klassen. Somit ist die Möglichkeit zur Erkennung von ausgehenden Verbindungen mittels Reflections über die Erkennung und Nutzung anderer fachlicher Typen gegeben. Am Beispiel des zu untersuchenden Shop-Systems besteht eine Verbindung zwischen der Klasse Kunde und Adresse, da eine Property, und zwar die Kundenadresse in Kunde, vom Typ „Adresse“ ist. Die Multiplizität lässt sich darüber ermitteln, ob es sich bei dem Typ um eine „Collection“ oder ein Array handelt. Somit lassen sich alle Verbindungen auflösen, die in Reflections erneut auf einen Typ referenzieren, wie in dem Listing in Abschnitt 8.3.2 dargestellt.

Über diesen Ansatz können jedoch nicht nur Verbindungen zwischen Klassen allgemein ermittelt werden, sondern es kann auch zwischen der Art einer Verbindung unterschieden werden, also ob es sich um eine Vererbungs-, Realisations- oder Benutzt-Beziehung handelt. Der über die Reflections API zugreifbare BaseType stellt die Vaterklasse einer eventuell vorhandenen Vererbungsbeziehung dar. Das Typen-Array aus dem Methodenaufruf „`getInterfaces()`" eines Typs ist die Sammlung aller realisierten Schnittstellen und ist somit eine Realisierungsbeziehung. Alle anderen Verbindungsarten beschreiben die Benutzt-Beziehung.

Bei der Untersuchung der Benutzt-Beziehung kann Reflections aber nicht die Art der Variableninstanziierung betrachten, es werden nur die Ergebnisse einer Instanziierung in Form von Typen dargestellt. So können in den Konstruktoren von Typen weitere fachliche Typen instanziiert werden. Folgendes Beispiel der Klasse „`service_Registrierung`" aus dem Shop-System soll dies verdeutlichen:

```
1   class service_Registrierung
2     {
3         public service_Registrierung()
4         {
5             regKunde = new Kunde("","","","",
6             new Adresse("","",0,0),"",
7             new Land("",0,0));
8
9             …
10
11        }
12        public Kunde regKunde { get; private set; }
13         { …
14         }
15      }
```

Die Klasse „`service_Registrierung`" beinhaltet eine Verbindung zu einer Instanz einer Kunden-Klasse in Form einer Property namens „`regKunde`" (Zeile 5), welches über den Konstruktor „`new Kunde (string, string, string, string, Adresse,String,Land)`" instanziiert wird.

In der Schnellübersicht aus Visual Studio, die ebenfalls auf die Reflections API zugreift, ist ersichtlich, dass der Zugriff bzw. die Erstellung einer Adresse und eines Landes nicht in den lokalen Variablen des Konstruktors enthalten ist. Vielmehr ist nur der Zugriff auf eine Shop-Klasse, die in dem Listing (siehe Zeile 1-15) nicht dargestellt wurde, vorhanden.

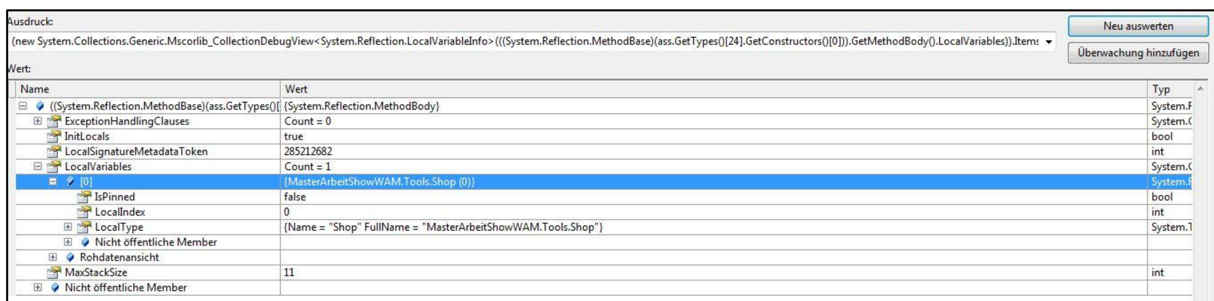

Abbildung 26: Schnellübersicht Reflections aus VisualStudio 2010

Im Konstruktor der Klasse „service_Registrierung" wird die Property „regKunde" instanziiert. Dabei wird ein Konstruktor der Klasse Kunde aufgerufen, dessen Signatur neben String-Werten auch die fachlichen Klassen „Adresse" und „Land" fordert. In diesem Fall werden die Klassen direkt in dem Konstruktor-Aufruf von „new Kunde(…)" erzeugt und nicht in einer Variablen deklariert. Hierdurch entzieht sich dieser Aufruf von „new Adresse(…)" und „new Land(…)" der Analyse durch Reflections, da diese nur die statischen Aspekte eines Typs untersuchen kann und nicht das Verhalten von Methoden interpretieren kann. Bei dieser Art der Instanziierung einer Kunden-Klasse hält die Klasse „service_Registrierung" ebenfalls Benutzt-Beziehungen zu den Klassen „Adresse" und „Land", diese Verbindungen können aber mit den bisher genutzten Mitteln nicht aufgedeckt werden. Dies gilt für alle Verbindungen, die nicht zuerst durch eine Typdeklaration in einer Klasse, genauer in einem Methodenrumpf, definiert wurden. Um diese Verbindungen dennoch aufdecken zu können, reichen die Grundmittel der Reflections API nicht aus. Da es sich jedoch nur um Probleme im Umfeld von Methoden-Rümpfen handelt, stellt die Reflections API hier einen Zugang zur Common Intermediate Language (vgl. Kapitel 4.1.2) bereit. Über den Aufruf „.getMethodBody().getILAsByteArray()" wird die Common Intermediate Language (CIL) des Methoden-Rumpfes als ByteArray geladen. Dieses kann dann über Transformation in ein String-Array Zeile für Zeile untersucht werden. An dieser Stelle sind auch automatisch angelegte, temporäre Variablen, die bei der Instanziierung z.B. einer Adresse, wie im oberen Beispiel angelegt werden, sichtbar. Über diesen Weg sind die bisher nicht erreichbaren Verbindungen sichtbar zu machen:

```
1       mbr = new ILMethodBodyReader.MethodBodyReader((MethodInfo)info);
2             string ilBodyCode = mbr.GetBodyCode();
3             //Umwandlung von Byte[] in String
4             //String mit \n als Trennzeichen zwischen den Zeilen
5             //interessante Operatoren: call, callvirt, newobj
6             string[] strings = ilBodyCode.Split('\n');
7             foreach (string line in strings)
8            {
9             //hier die Erkennung der interessanten Operatoren durchführen
10            }
```

In dem oben gezeigten Code-Beispiel wird ein „ILMethodBodyReader" mit der Methodeninformation aus den Reflections-Zugriffen instanziiert. Der „ILMethodBodyReader" übernimmt die Transformation der ByteCodes, aus denen die Common Intermediate Language nativ besteht, hin zu einem String durch den Aufruf „mbr.GetBodyCode()". Über das Trennzeichen „\n" kann der String, welcher den gesamten Body einer Methode beinhaltet, zeilenweise getrennt werden, das Ergebnis einer Zeile als String sieht dann z.B. so aus:

```
call instance
System.VoidMasterArbeitShowWAM.Material.Land::InitializeComponent()
```

Anschließend wird über eine Iteration über alle Zeilen nach Schlagwörtern wie „call", „callvirt" oder „newobj" über String-Operationen wie „String.contains(„newobj)" gesucht. Folgend können die enthaltenen Verbindungen über einfache String-Operationen und -Vergleiche aufgedeckt werden, indem ermittelt wird, ob der Name der genutzten Instanz, hier „Land" (gelb), eine fachliche Klasse ist. Wenn dies zutrifft, kann diese in die

Analyse mit einbezogen und als Benutzt-Beziehung der Klasse „service_Registrierung" interpretiert werden.

Da keine Unterscheidung zwischen den Benutzt-Beziehungen in C# getroffen werden können, werden für alle Analysen und Abbildungen auf die Zwischenformate die Verbindungen zwischen Typen in C# auf dieselbe Art und Weise ermittelt. Daher wurde diese Funktionalität der Ermittlung der Verbindung zwischen C#-Typen extrahiert. Eine Klasse namens „CSharpReflectionAssemblyConnectionChecker" analysiert hierzu folgende Aspekte:

- Eigenschaften
- Felder
- Methodensignaturen
- Methodenrümpfe
- Vererbung
- Schnittstellenrealisierung
- Konstruktoren

Zur Wiederverwendung in den folgenden Analyseapparaten wurde der „CSharpReflectionAssemblyConnectionChecker" wie folgt implementiert (Auszug aus dem Quelltext):

```
1  namespace Analyst1stTry.ConnectionCheck
2  {
3      class CSharpReflectionAssemblyConnectionChecker
4      {
5          private string _assemblyPath;
6          private Dictionary<string, string> Generalisations;
7          private Dictionary<string, HashSet<string>> InterfaceRealisation;
8          private Dictionary<string, Dictionary<string,bool>> UsesConnections;
9          private List<string> fachlicheKlassen;
10
11         CSharpReflectionAssemblyConnectionChecker(string assemblyPath)
12         {
13             …
14         }
15
16         public Dictionary<string, string> getGeneralisation()
17         {
18             Assembly ass = System.Reflection.Assembly.LoadFile(_assemblyPath);
19             Type[] types = ass.GetTypes();
20             foreach (Type t in types)
21             {
22                 if (t.BaseType != null)
23                 {
24                     Generalisations.Add(t.Name, t.BaseType.Name);
25                 }
26             }
27
28             return Generalisations;
29         }
30
31         public Dictionary<string, HashSet<string>> getInterfaceRealisations()
32         {
33             …
34         }
35
```

```
36          public Dictionary<string, Dictionary<string,bool>> getUsesConnections()
37          {
38              Assembly ass = System.Reflection.Assembly.LoadFile(_assemblyPath);
39              Type[] types = ass.GetTypes();
40              foreach (Type t in types)
41              {
42                  getFieldConnections(t);
43                  getPropertyConnections(t);
44                  getMethodConnections(t);
45                  getContructorConnections(t);
46              }
47
48              return UsesConnection;
49          }
50
51          private void getFieldConnections(Type t)
52          {
53              …
54          }
55
56          private void getPropertyConnections(Type t)
57          {
58              …
59          }
60
61          private void getMethodConnections(Type t)
62          {
63              …
64          }
65
66          private void getContructorConnections(Type t)
67          {
68              …
69          }
70      }
71  }
```

Die Verbindungsinformationen werden in den drei Feldern „`Generalisations`“, „`InferfaceRealisations`“ und „`UsesConnections`“ gehalten. Da eine Vererbung in C# nur von einer Klasse geschehen kann (Einfachvererbung), wurde als Datentyp für die „`Generalisation`“ ein Dictionary gewählt, welches als Key den Namen des derzeit untersuchten Typs und als Value den Namen der Vaterklasse hält, von dem der jeweils untersuchte Typ geerbt hat. „`InterfaceRealisation`“ hat als Datentyp ebenfalls ein Dictionary, jedoch als Value ein HashSet, welches die Namen der implementierten Schnittstellen hält. Es ist möglich, dass mehrere Schnittstellen implementiert wurden, daher die Ergänzung um ein HashSet. Bei den „`UsesConnection`“ ist ebenfalls ein Dictionary eingesetzt worden, jedoch als Value mit einem weiteren Dictionary. Hier werden als Key in dem enthaltenen Dictionary der Name des referenzierten Typs und als boolescher Wert der Value gespeichert, wobei der boolesche Wert die Mulitplizität repräsentiert („true“ bei Mehrfach- und „false“ bei Einfachverwendung). Das Feld „`fachlicheKlassen`“ aus Zeile 9 stellt das Ergebnis des fachlichen Klassenchecks, wie unter Kapitel 8.3.1 beschrieben, dar.

Die „`CSharpReflectionAssemblyConnectionChecker`“-Klasse stellt drei öffentliche Methoden zur Verfügung:

- `getGeneralisations`()
- `getIterfaceRealisations`()

- getUsesConnections()

Zeile 16 -29 zeigt die Implementierung der „getGeneralisations()"-Methode. Nach dem Laden des Assembly werden die einzelnen enthaltenen Typen per Iteration untersucht. Dabei wird zu jedem Typ der jeweilige BaseType abgefragt. Dieser stellt, wenn ein Wert enthalten ist, die Informationen des Typs dar, von dem der untersuchte Typ geerbt hat. Der Name der erbenden und vererbten Klasse wird in Zeile 24 in das „Generalisation"-Dictionary eingefügt. Nachdem alle Typen iterativ verarbeitet wurden, wird dieses „Generalisation"-Dictionary als Rückgabewert der Methode zurückgegeben (Zeile 28). Der Aufbau der „getInferfaceRealisations()"-Methode ist analog zu der „getGeneralistions()"-Methode, nur wird hier nicht der BaseType sondern das Type-Array, welches der Aufruf „t.getInferfaces()" zurückgibt, ausgewertet und in einem HashSet gespeichert.

Die „getUsesConnections()"-Methode sammelt alle Informationen zu den Benutzt-Beziehungen in C#. Zusammenfassend werden hier alle referenzierten Typen einer Klasse, die ebenfalls in den fachlichen Klassen enthalten sind, gesammelt. Dazu müssen sowohl die Felder, Properties und Methoden, als auch Konstruktoren analysiert werden. Mehr Varianten sind in C# nicht vorhanden, um Typen zu repräsentierten.

Die Zeilen 42 – 45 zeigen die Aufrufe von nicht-öffentlichen Methoden, die genau diese Informationen sammeln sollen. Folgendes Listing aus dem Quelltext zeigt die „getFieldInformation()"-Methode auf:

```
73      private void getFieldConnections(Type t)
74          {
75              foreach (FieldInfo fi in t.GetFields())
76              {
77                  string fieldTypeName = fi.FieldType.Name;
78                  if (fachlicheKlassen.Contains(fieldTypeName))
79                  {
80                      if (UsesConnection.ContainsKey(t.Name))
81                      {
82
83                          UsesConnection[t.Name].Add(fieldTypeName,false);
84                      }
85                      else
86                      {
87                           UsesConnection.Add(t.Name,new
88                           Dictionary<string,bool>());
89                           UsesConnection[t.Name].Add(fieldTypeName,false);
90                      }
91                  }
92                  else
93                  {
94                     if (fi.FieldType.BaseType.IsGenericType &&
95                     fi.FieldType.Namespace.Contains("Collection"))
96                      {
97                           …
98                      }
99                  }
100             }
101         }
```

In Zeile 75 wird über jedes „FieldInfo"-Attribut des Typs „t" iteriert und in Zeile 77 für jeden „FieldInfo"-Typ der zugrundeliegende Klassenname zwischengespeichert. Anschließend

erfolgt in Zeile 78 eine Prüfung, ob es sich um eine fachliche Klasse handelt. Ist dies der Fall, wird zwischen Zeile 80 und 89 der jeweilige Typname in das „UsesConnections“-Dictionary eingefügt. In diesem Fall handelt es sich immer um eine Multiplizität von eins, somit ist der boolesche Wert stets mit „false“ angegeben. Handelt es sich bei dem untersuchten „FieldInfo“-Attribut nicht um eine fachliche Klasse, kann es sich dennoch um eine Collection handeln, die eindeutig fachliche Klassen halten kann. Diese Eindeutigkeit ist jedoch nur durch Collections, die über generische Attribute (Generics) verfügen, eindeutig festzustellen. Collections wie die ArrayList, die keine Typeinschränkungen über Generics besitzen, können nicht auf einen definierten enthaltenen Typ zurückgeführt werden. In dem Falle, dass es sich um eine generische Collection handelt, können weitere Verbindungen zu fachlichen Klassen festgestellt werden (Zeile 93 – 94). Hierzu ist aus Darstellungsgründen ab Zeile 96 die Darstellung abgebrochen worden. Über die Methode „getGenericArguments()“ lässt sich eine Liste von Typen erstellen, die analog zu den oben genannten Schritten aus Verbindungen zu anderen Typen analysiert werden und anschließend in dem „UsesConnections“-Dictionary gespeichert werden können. Der Aufbau der „getPropertyConnections()“-Methode ist analog zu der „getFieldConnections()“-Methode.
In der „getMethodConnections()“- und „getConstructorConnections“-Methode werden mit dem bereits vorgestellten Vorgehen Informationen zu den Methodensignaturen und –rümpfen gesammelt. Hier kommt jedoch noch die Auswertung der CIL, wie in diesem Kapitel bereits diskutiert, hinzu. Nachdem die nicht-öffentlichen Methoden allesamt im Ablauf der „getUsesConnections()“-Methode aufgerufen wurden, wird als Rückgabewert das „UsesConnections“-Dictionary als Ergebnis zurückgeben.

Die hier gesammelten Informationen können in den folgenden Algorithmen genutzt werden, um die Erkennung von Verbindungen zwischen den architekturbeschreibenden Elementen zu ermöglichen. Da alle architekturbeschreibenden Elemente auf Typen in Reflections, zumindest in dieser Arbeit, zurückgeführt werden, können diese Abbildungsinformationen genutzt werden, um die Ergebnisse der „CSharpReflectionAssemblyConnectionChecker“-Klasse in den Analyseablauf zu integrieren und die Ergebnisse entsprechend abzubilden.

8.3.4 Laden des entsprechenden Zwischenmodells

Beim Starten des Softwarearchitekturüberprüfungs-Werkzeug muss herausgefunden werden, welche Kombination von Diagramm und Zwischenmodell miteinander verglichen werden sollen. Dies ist vom eingegebenen Diagrammtyp abhängig. Hierbei gibt es zwei Varianten, die umgesetzt werden können:

- Manuelle Auswahl des Zwischenmodells durch den Anwender
- Automatische Auswahl des Zwischenmodells anhand des Eingabediagramms

In der prototypischen Umsetzung wird die automatische Auswahl nicht implementiert, sondern der Anwender wählt das entsprechende Werkzeug, welches definiert, was für ein Eingabediagramm und Zwischenmodell genutzt werden soll. Hierdurch wird ebenfalls definiert, welche Transformations-, Vergleichs und Exportalgorithmen verwendet werden

sollen. Eine Erweiterung um automatische Auswahl eines Zwischenmodells oder die Auswahl mehrerer Zwischenmodelle wäre in einem späteren Prototypen zu realisieren. Somit sind drei Werkzeuge erstellt worden, die jeweils in Visual Studio erstellte Diagramme vom Typ Klassen-, Schichten- und Komponentendiagramme als Eingabediagramm akzeptieren. Für jeden Diagrammtyp wurde ein Zwischenmodell, vgl. Kapitel 7.1, erstellt und implementiert. Über die Schnittstelle „`IKlassenmodell`“ ist eine Austauschbarkeit der Implementierung gegeben und kann in späteren Ausbaustufen realisiert werden.

8.3.5 Umsetzung der Abbildungsregeln für die unterstützen Diagramme

In diesem Unterkapitel werden die implementierten Transformationsalgorithmen für die Umwandlung der Ist-Architektur in ein entsprechendes Zwischenmodell vorgestellt. Dazu wird zu jedem in dieser Arbeit unterstützten Diagrammtyp und dem dazugehörigen Zwischenmodell die Implementierung der Transformationsmethode „transform()“ vorgestellt, welche die Abbildungsregeln beinhaltet.

8.3.5.1 Zwischenmodell basiert auf einem Schichtendiagramm

Im Falle der Implementierung für Schichtenmodelle ist die Transformationsmethode „transform()“ wie folgt aufgebaut:

```
1   public void tranform()
2   {
3          // fachliche Klassen ermitteln
4          List<string> classes = new List<string>();
5          fachlicheKlassenCheck.FachlicheKlassenChecker checker = new
6          fachlicheKlassenCheck.FachlicheKlassenChecker();
7          checker.initFachKlassenChecker(_assemblyPath);
8           classes = checker.getClasses();
9          //Namen und Namespace für Typen aus Assembly extrahieren
10         //KVP: <Typname>,<Namespace>
11         Dictionary<string, string> classesAndNamesspaces =
12         new Dictionary<string, string>();
13         classesAndNamesspaces = getNamespacesForClasses(classes);
14         //Namespaces der Typen als Layer darstellen
15         HashSet<string> layerNames = new HashSet<string>();
16         //Reduktion der Mehrfachnennung eines Namespaces über HashSet
17         foreach(string oneNamespace in
18         classesAndNamesspaces.Values.ToList<string>())
19                {
20                    layerNames.Add(oneNamespace);
21                }
22          //Layer nun in das Zwischenmodell einfügen inkl. Layernamen
23          foreach (string layerName in layerNames)
24                {
25                    Zwischenmodell.Add(new Layer(layerName));
26                }
27          //Connections der Typen zueinander aus Assembly extrahieren
28                //KVP: <Typenname><Hashset mit den Namen der Typen, auf die
29                Zugegriffen wird>
30         Dictionary<string, Dictionary<string,bool>> assemblyConnections = new
31         Dictionary<string,Dictionary<string,bool>>();
32          CSharpReflectionAssemblyConnectionChecker cSharpChecker = new
33         CSharpReflectionAssemblyConnectionChecker();
```

```
34         assemblyConnections = cSharpChecker.getUsesConnections();
35         foreach (KeyValuePair<string, Dictionary<string,bool>> kvp in
36         assemblyConnections)
37               {
38               // KVP<Key>: Layer des Typen, der eine Verbindung zu anderen Typen
39               hat, herausfinden
40                Layer currentLayer = getLayerByName(kvp.Key);
41               // jedem Typen, den kvp(Key) als Typen referenziert auf fachliche
42               Klasse prüfen
43                   foreach (string type in kvp.Value.Keys.ToList<string>())
44                   {
45                       if (classes.Contains(type))
46                       {
47                            //Wenn der referenzierte Typ fachlich ist, den
48                            dazugehörigen Namespace herausfinden
49                           string connectedlayerName =
50                           classesAndNamesspaces[type];
51                           //NamespaceName == Layername, Referenz in Layer
52                           eintragen
53
54                            currentLayer.outgoingEdgeTo.Add(currentLayer,
55                            connectedlayerName);
56                       }
57                   }
58               }
59 }
```

Zunächst werden die fachlichen Klassen ermittelt (Zeile 3-7), hierzu wird die Klasse „`FachlicherKlassenchecker`“ benutzt, die das Assembly durchläuft und die Typennamen in einer Liste speichert, die zu den fachlichen Klassen unter Berücksichtigung der Ausschlussregeln gehören. Anschließend soll herausgefunden werden, welche fachlichen Klassen welchem Namespace angehören, da diese Namespaces als Indikator für die Schichtzugehörigkeit definiert wurden. Dazu wird eine Methode namens „`getNamespacesForClasses(List<string> classes)`“ aufgerufen. Das Listing zu dieser Methode ist folgendes:

```
60 public Dictionary<string, string> getNamespacesForClasses(List<string> classes)
61         {
62               Assembly assembly =
63               System.Reflection.Assembly.LoadFile(assemblyPath);
64              Type[] types = assembly.GetTypes();
65              //KVP: <Typname>,<Namespace>
66               Dictionary<string, string> classesAndNamesspaces = new
67               Dictionary<string, string>();
68              foreach (Type t in types)
69              {
70                  if (classes.Contains(t.Name))
71                  {
72                      string nameSpace = t.Namespace;
73                  //relevanten Teil des Namespaces identifizieren &  extrahieren
74                      int idx = nameSpace.LastIndexOf(".");
75                      classesAndNamesspaces.Add(t.Name,
76                      nameSpace.Substring(idx + 1));
77                  }
78              }
79              return classesAndNamesspaces;
80         }
```

Zeile 54 und 55 zeigen das Laden des Assembly und die Filterung der enthaltenen Typen. Eine Iteration über diese Typenliste (Zeile 59 ff.) enthält die programmatische Extraktion des Namespaces. In Zeile 61 wird zunächst geprüft, ob es sich bei dem aktuellen Typ „t“ um eine fachliche Klasse handelt. Ist dies der Fall, wird der Namespace ab Zeile 63 extrahiert und um den relevanten Teil gekürzt. Als Rückgabewert wird ein „`KeyValuePair`“ in Form eines Dictionary geliefert, dass als Key den Klassennamen und als Value den dazugehörigen Namespace-Namen eines Typs enthält.

Nun können diese erhaltenen Informationen in das Zwischenmodell überführt werden. Da eine Menge an Klassen einem einzigen Namespace zugehören kann, kommt es vor, dass Namespaces doppelt genannt werden. Die Reduktion dieser Information geschieht, indem wie in Zeile 14 ff. alle Values aus dem Dictionary „`classesAndNamespaces`“ in ein HashSet überführt werden. HashSets enthalten keine Mehrfachnennungen, sodass dieses HashSet mit Schichtennamen anschließend in das Zwischenformat überführt werden kann. Zeile 21 – 24 zeigen diesen Schritt, in dem ein Typ namens „Layer“ (aus dem Zwischenmodell) mit den Namespaces-Namen der Typen aus dem HashSet „`layerNames`“ erstellt und im Zwischenformat gespeichert wird. Das Zwischenformat ist als „List<Layer>“ deklariert.

Zu diesem Zeitpunkt sind alle Schichten der Quelltextrepräsentation in das Zwischenmodell überführt worden. An dieser Stelle fehlen nun die Informationen über die Verbindungen zwischen den Schichten. Hierzu werden, wie in Kapitel 8.3.3 beschrieben, die Benutzt-Verbindungen mittels der „`CSharpReflectionAssemblyConnectionChecker`“-Klasse ermittelt. Zeile 32 beschreibt eine Iteration über die Ergebnisse der Verbindungsanalyse. Hierbei wird nun ab Zeile 40 für jede Verbindung die Schichtenzugehörigkeit hergestellt und zu dem entsprechenden Layer als ausgehende Verbindung hinzugefügt. Nach diesem Schritt ist die Transformation des Quelltextes, in diesem Fall durch ein Assembly repräsentiert, auf ein Zwischenmodell für Schichtendiagramme abgeschlossen. Alle relevanten Informationen sind gesammelt und entsprechend überführt.

8.3.5.2 Zwischenmodell basiert auf einem Klassendiagramm

Die Abbildung von C# auf Klassendiagramm-Elemente kann direkt umgesetzt werden, da sehr ähnliche Elemente in beiden Darstellungsformen enthalten sind. Im Zentrum beider Betrachtungen steht eine Klasse. Es folgt das Listing für die „transform()“-Methode:

```
1   public void transform()
2           {
3               Assembly assembly =
4               System.Reflection.Assembly.LoadFile(_assemblyPath);
5               Type[] types = assembly.GetTypes();
6               FachlicheKlassenChecker checker = new FachlicheKlassenChecker();
7               checker.initFachKlassenChecker(_assemblyPath);
8               fachlicheKlassen = checker.getClasses();
9               foreach (Type t in types)
10              {
11                  if (fachlicheKlassen.Contains(t.Name))
12                  {
13                      ClassDiagramClass currentClass = new ClassDiagramClass();
14                      currentClass.Name = t.Name;
```

```
15                      currentClass.isAbstract = t.IsAbstract;
16                      currentClass.isClass = t.IsClass;
17                      currentClass.isInterface = t.IsInterface;
18                      //Unterbestandteile analysieren
19                      getAttributeInfo(t, currentClass);
20                      getMethodInfo(t, currentClass);
21                      getGeneralisations(t, currentClass);
22                      getImplementedInterfaces(t, currentClass);
23                      getClassConnections(t, currentClass);
24                     zwischenmodell.Add(currentClass);
25                  }
26              }
27          }
```

Nach dem Laden und der Erkennung der fachlichen Klassen (Zeile 3-7) erfolgt eine Iteration über alle geladenen Typen. Wenn der untersuchte Typ „t" eine fachliche Klasse ist (Zeile 10), dann wird eine neue „ClassDiagramClass"-Instanz aus dem Zwischenmodell erstellt. Anschließend werden in Zeile 13 – 16 alle allgemeinen Informationen der Klasse, die auch im Zwischenmodell eine Relevanz haben, aus den Reflections extrahiert. Ab Zeile 18 – 22 werden erneut nicht-öffentliche Methoden genutzt, um die Details der Attribute, Methoden, genutzten Schnittstellen und Generalisationen zu extrahieren.

Im Folgenden wird als Beispiel für solch eine Implementierung die Methode „getMethodInfo()" näher aufgeführt:

```
27  private void getMethodInfo(Type t, ClassDiagrammClass oneClass)
28          {
29              MethodInfo[] minfos = t.GetMethods();
30              foreach (MethodInfo mi in minfos)
31              {
32                  ClassdiagramMethod currentMethod = new ClassdiagramMethod();
33                  currentMethod.Name = mi.Name;
34                  currentMethod.isAbstract = mi.IsAbstract;
35                  currentMethod.isStatic = mi.IsStatic;
36                  if (mi.IsPublic)
37                  {
38                      currentMethod.Visibility = "public";
39                  }
40                  else
41                  {
42                      if (mi.IsPrivate)
43                      {
44                          currentMethod.Visibility = "private";
45                      }
46                      else
47                      {
48                          currentMethod.Visibility = "protected";
49                      }
50                  }
51                  //Rückgabewert:
52                  ParameterInfo returnValue = mi.ReturnParameter;
53                  string typeName = returnValue.ParameterType.Name;
54                  if (fachlicheKlassen.Contains(typeName))
55                  {
56                     ClassDiagramMethodParameter returnPara = new
57                     ClassDiagramMethodParameter();
58                      returnPara.direction = "return";
59                      returnPara.Name = "";
60                      returnPara.Type = typeName;
61                      returnPara.Multiplizität = "1";
62                  }
```

```
63                    else
64                    {
65                       if
66                       (returnValue.ParameterType.Namespace.Contains("Collection"))
67                        {
68                            //Generics abfragen und diese als Typen prüfen
69                             …
70                        }
71                    }
72                    //Parameter
73                    ParameterInfo[] pinfos = mi.GetParameters();
74                    foreach (ParameterInfo pi in pinfos)
75                    {
76                       ClassDiagramMethodParameter currentParameter = new
77                       ClassDiagramMethodParameter();
78                        currentParameter.Name = pi.Name;
79                       …
80                    }
81                oneClass.Methoden.Add(currentMethod);
82                }
83           }
```

Hier wiederholen sich die beschriebenen Strukturen und der Umgang mit der Reflections API aus DotNet wie in den anderen beschriebenen Algorithmen. Die Abbildungsregeln für C# sind immer auf die vorhandenen und über Reflections erreichbaren Informationen zurückgeführt. Es wird über Reflections an die entsprechende Stelle navigiert (die Iterationen über die Typen) und die entsprechende Information in das Zwischenmodell gespeichert (z.B. Zeile 32 – 35). Bei dem Umgang mit Methoden ist zu beachten, dass C#-Reflections anders als die Diagramme den Rückgabewert einer Methode gesondert behandeln und diesen nicht in dem Parametersatz direkt enthalten, daher muss der Rückgabewert wie in Zeile 33-43 behandelt werden. Sobald Multiplizitäten eine Rolle spielen, muss geprüft werden, ob in einem Typ-Namespace das Wort „Collection“ oder das Tag „`isArray`“ enthalten ist, hierüber lassen sich dann über die Untersuchung der generischen Typen der Collection erweiterte Informationen extrahieren.

Die Abbildung der Verbindungen erfolgt erneut über die `CSharpReflectionAssemblyConnectionChecker`“-Klasse in der nicht-öffentlichen „`getClassConnections`“-Methode. Das Ergebnis der Analyse dieser Klasse muss in diesem Fall nur noch dahingehend geprüft werden, ob die ermittelten Klassen fachlicher Natur sind. Ist dies der Fall, ist die Klasse, die diese Verbindung hält, über den Namen aus dem Zwischenmodell zu ermitteln und die Verbindung einzutragen. Die Multiplizität ist dem booleschen Attribut des Value-Dictionary zu entnehmen.

8.3.5.3 Zwischenmodell basiert auf einem Komponentendiagramm

Komponentendiagramme zeichnen sich durch die höchste Komplexität aller untersuchten Diagrammtypen aus. Dies liegt zum einen an der höheren Anzahl der zu untersuchenden und abzubildenden architekturbeschreibenden Elemente und zum anderen daran, dass Komponenten keine direkte Repräsentation in der Klassenstruktur in C# haben. Dazu kommt, dass eine Komponente sowohl eine externe Sicht als auch eine interne Sicht über den inneren Aufbau einer Komponente aufweisen. Diese sind in der Analyse ebenfalls zu

unterscheiden und es gilt festzustellen, ob eine Klasse oder ein Typ, der mittels Reflections analysiert wird, selbst eine Komponente oder ein Bestandteil einer Komponente ist. In den Abbildungsregeln für Komponentendiagramme wurde definiert, dass hier eine Unterscheidung nach externer und interner Sicht stattfindet. Die interne Sicht einer Komponente besteht wiederum aus Komponenten (Rekursion) oder Klassen (analog zu Klassendiagrammen).

Daher wird in dieser Ausarbeitung eine nähere Betrachtung der internen Struktur einer Komponente nicht weiter bearbeitet, da es sich hierbei um eine Adaption der Klassendiagrammanalyse für die interne Struktur handelt, erweitert um den rekursiven Aspekt der Komponenten. Aus diesem Grund wird im folgenden Listing die Analyse der externen Sicht, also die Darstellung der Generalisation, die Darstellung der angebotenen und benötigten Schnittstellen und auch die Darstellung der allgemeinen Verbindungen zwischen Komponenten, aufgeführt:

```
1       public void transform()
2           {
3               Assembly assembly =
4               System.Reflection.Assembly.LoadFile(_assemblyPath);
5               Type[] types = assembly.GetTypes();
6               FachlicheKlassenChecker checker = new FachlicheKlassenChecker();
7               checker.initFachKlassenChecker(_assemblyPath);
8               fachlicheKlassen = checker.getClasses();
9               foreach (Type t in types)
10              {
11                  Type[] interfaces = t.GetInterfaces();
12                  Komponente currentComponent = new Komponente();
13                  foreach (Type inter in interfaces)
14                  {
15                      if (inter.Name.ToLower().Equals("icomponent"))
16                      {
17                          //!!! Dann haben wir eine Komponente
18                          currentComponent.Name = t.Name;
19                          currentComponent.isAbstract = t.IsAbstract;
20
21                          getExternalGeneralisation(t, currentComponent);
22                          getExternalViewInterfaces(t, currentComponent);
23                          getExternaViewConnections(t, currentComponent);
24
25                        //Interne Sicht wie bei KLassendiagrammen abzubilden
26                        zzgl. Parts und internen Komponenten
27
28                          zwischenmodell.Add(currentComponent);
29                      }
30                  }
31              }
32          }
```

Die Zeilen 3 – 7 beschreiben erneut das Laden des Assembly als auch die Selektion der fachlichen Klassen. Wie in den Abbildungsregeln zu Komponentendiagrammen beschrieben, wurde für diese Arbeit definiert, dass eine Komponentenrepräsentation in C# eine Schnittstelle namens „IComponent" implementieren muss. Daher muss jeder Typ in Reflections dahingehend untersucht werden, ob die Schnittstelle „IComponent" realisiert wird (Zeile 8-15). Ist dies der Fall, handelt es sich bei dem untersuchten Typ um eine Komponentenrepräsentation. In den Zeilen 17 und 18 werden die allgemeinen Informationen, der Name und ob die Komponente abstrakt ist, gesammelt. Die nicht-öffentlichen Methoden

„getExternalGeneralisations()", „getExternalViewInterfaces()" und „getExternalViewConnections" sammeln die weiteren Informationen, die für die Repräsentation einer Komponente benötigt wird. Komponenten sind geprägt durch die angebotenen und benötigten Schnittstellen, daher soll in diesem Abschnitt die „getExternalViewInterfaces()"-Methode beschrieben werden:

```
33    private void getExternalGeneralisation(Type t, Komponente currentComp)
34          {
35              //angebotene Schnittstellen: implementierte Schnittstellen
36              analysieren
37              Type[] interfaces = t.GetInterfaces();
38              foreach (Type currentInferface in interfaces)
39              {
40              ExternalViewInterfaces currentZwischenmodellInterface = new
41              ExternalViewInterfaces();
42                  if (!currentInferface.Name.Contains("IComponent"))
43                  {
44                     currentZwischenmodellInterface.interfacName =
45                     currentInferface.Name;
46                      currentZwischenmodellInterface.interfaceType = "provided";
47
48                     currentComp.externalInterfaces.Add(currentZwischenmodellInte
49                     rface);
50                  }
51              }
52              //benötigte Schnittstellen: über Konstruktoren filtern
53              ConstructorInfo[] constructors = t.GetConstructors();
54              foreach (ConstructorInfo ci in constructors)
55              {
56                  foreach (ParameterInfo pi in ci.GetParameters())
57                  {
58                      if (pi.ParameterType.IsInterface)
59                      {
60                          ExternalViewInterfaces currentZwischenmodellInterface
61                          = new ExternalViewInterfaces();
62                          currentZwischenmodellInterface.interfaceType =
63                          "required";
64                          currentZwischenmodellInterface.interfacName =
65                          pi.ParameterType.Name;
66                          currentComp.externalInterfaces.Add
67                          (currentZwischenmodellInterface);
68                      }
69                  }
70              }
71          }
```

Die angebotenen Schnittstellen werden in den Abbildungsregeln darüber definiert, dass alle implementierten Schnittstellen, außer die „IComponent"-Schnittstelle, die ausschließlich eine Komponentenrepräsentation festlegt, als solche zu identifizieren sind. In den Zeilen 36 – 50 wird diese Erkennung implementiert, indem alle Schnittstellen über „getInferfaces()" selektiert, deren Typennamen in das Zwischenmodell überführt und mit „provided" beschrieben werden.

Benötigte Schnittstellen sind definiert als Schnittstellen, die in den Konstruktoren einer Komponentenfassade (siehe Kapitel 7.1.3) enthalten sind. In den Zeilen 52- 65 werden alle Konstruktoren iteriert und es wird festgestellt, welche Parameter als Typ eine Schnittstelle sind. Diese werden erneut in das Zwischenmodell überführt und mit „required" beschrieben.

8.4 Vergleiche der Abbildungen

Die Implementierung der Vergleichsalgorithmen nutzt ausschließlich die angereicherten Zwischenmodelle aus den vorhergegangen Unterkapiteln und führt den Vergleich auf diesen aus. Hierbei ist noch von Relevanz, dass das Zwischenmodell der Soll-Architektur, sprich dem eingegebenen Diagramm, als korrekt gilt. Die Abbildung der Ist-Architektur somit gegen die Soll-Architektur verglichen, da Übereinstimmungen und auch Abweichungen der Ist-Architektur herausgefunden werden sollen.

```
1       public List<Layer> compare()
2           {
3               List<Layer> result = new List<Layer>();
4               foreach (Layer inSoll in soll)
5               {
6                   foreach (Layer inIst in ist)
7                   {
8                      if
9                      (inSoll.LayerName.ToLower().Equals(inIst.LayerName.ToLower()
10                     ))
11                       {
12                           //Übereinstimmung gefunden
13                           Layer resLayer = new Layer(inSoll.LayerName);
14                           resLayer.isOk = true;
15                           result.Add(resLayer);
16                           //Connections prüfen)
17                            foreach (LayerConnection toLayerNameSoll in
18                            inSoll.verbindungen)
19                           {
20                                  foreach (LayerConnection toLayerNameIst in
21                                  inIst.verbindungen)
22                               {
23
24                                  if(toLayerNameSoll.toLayerName.ToLower().Equals(
25                                  toLayerNameIst.toLayerName.ToLower())
26                                   {
27                                       //Übereinstimung gefunden
28                                  LayerConnection resCon = new
29                                  LayerConnection(toLayerNameSoll.toLayerName);
30                                       resCon.isOk = true;
31                                       resLayer.verbindungen.Add(resCon);
32                                   }
33                               }
34                           }
35                           addConnectionsFromIstNotInSoll();
36                           addConnectionsFromSollNotInIst();
37
38                       }
39                   }
40               }
41               addLayerFromIstNotInSoll();
42               addLayerFromSollNotInIst();
43               return result;
44           }
```

Das oben gezeigte Listing der „`compare`()“-Methode ist für den Vergleich von Schichtendiagramm-Zwischenmodellen entwickelt worden. In einer geschachtelten Schleife (Zeile 4 + 6) werden die Schichten der Zwischenmodelle durchlaufen und gegeneinander

geprüft. Zeile 8 beschreibt den Vergleich bezüglich Namensgleichheit zweier Schichten. Ist dieser Vergleich korrekt und erfolgreich, so wird eine Schicht erstellt, die diesen Namen trägt und dem Ergebniszwischenmodell („result“) zuzüglich der „isOK“-Information hinzugefügt wird (Zeile 11-13). Anschließend werden erneut zwei ineinander geschachtelte Iterationen über die Verbindungen der beiden Schichtenzwischenmodelle gestartet (Zeile 15-20). An dieser Stelle (Zeile 22-23) wird per String-Operation die Namensgleichheit der Verbindungen geprüft. Im Falle einer Übereinstimmung wird eine neue „LayerConnection“-Instanz erstellt und mit „isOk“ und in der Ist-Architektur als erfolgreich und korrekt implementiert gekennzeichnet.

Über dieses Verfahren werden nur direkte Übereinstimmungen, d.h. Schichten, die in der Ist-Architektur korrekt nach dem Vorbild der Soll-Architektur implementiert wurden, im Vergleich erkannt. Layer, die nur im Soll oder nur im Ist vorzufinden sind, werden dabei jedoch nicht erkannt. Selbiges gilt für fehlende oder überschüssige Verbindungen in der Ist-Architektur. Um diese Restmengen zu identifizieren, werden nach den Durchläufen der geschachtelten Iteration, sowohl bei der Betrachtung der Schichten als auch der Verbindungen, nicht-öffentliche Hilfsmethoden gestartet, die genau diese Restmengen filtern. Hierin entdeckte Schichten und Verbindungen werden ebenfalls dem Ergebniszwischenmodell „result“ hinzugefügt, jedoch mit dem „isOk“-Wert „false“ und einer textuellen Beschreibung, welche Schicht oder Verbindung nicht korrekt umgesetzt wurde. Am Beispiel einer Schicht, die im Soll definiert aber im Ist nicht vorhanden ist, wäre eine Fehlerbeschreibung: „Schicht: <Schichtname> ist im Soll aber nicht im Ist vorhanden“.

Dieser Vorgang des Vergleichs ist in den Implementierungen der Klassen- und Komponentenzwischenmodelle ähnlich angewandt. Hier ist jedoch eine größere Vielfalt an Informationen zu vergleichen. Der Ablauf und Abstieg innerhalb der Strukturen ist analog dieses hier gezeigten Beispiels anhand von Schichtenzwischenmodellen zu verstehen. Die Navigation durch die Strukturen eines Klassenmodells geschieht in diesem Ansatz über geschachtelte Iterationen, die Vergleiche der darin enthaltenen Informationen beruhen auf String- oder Boolean-Vergleichen.

8.5 Export der Ergebnisse

Ebenso wie beim Einlesen einer XML-Struktur können über die in DotNet enthaltenen Bibliotheken aus dem Namespace System.XML auch XML-Dokumente erstellt und in Dateien abgelegt werden. Da verschiedene Anbieter verschiedene Formate zur Speicherung der Diagramme nutzen, ist eine Analyse dieser Struktur nötig, um den Aufbau der XML-Struktur in einem Export-Werkzeug anwenden zu können. Da diese Arbeit nur XML-basierte Formate der Diagramme nutzt, soll als Stellvertreter für das gewählte Vorgehen des Exports das Beispiel des XMI-Exports von Komponentendiagrammen genutzt werden. XMI-Dokumente nutzen einen standardisierten Aufbau, dabei hat jedes Element eine eindeutige ID, über die das jeweilige Element referenziert werden kann. Über einen Pseudozufallszahlengenerator wurden entsprechende IDs generiert. Viele Werkzeuge bieten bei dem Einlesen die Option an, die vorhandenen IDs an interne IDs anzupassen, sodass ein anbieterabhängiger Umgang mit IDs nicht nötig ist.

```
1  public void export()
2          {
3              XmlDocument doc = new XmlDocument();
4
5              XmlNode docNode = doc.CreateXmlDeclaration("1.0", "UTF-8", null);
6              doc.AppendChild(docNode);
7
8              XmlNode xmiMainNode = doc.CreateElement("xmi:XMI", "xmi");
9              doc.AppendChild(xmiMainNode);
10
11             xmiMainNode.Attributes.Append(gma("xmi:version", "xmi", "2.1", doc));
12              xmiMainNode.Attributes.Append(gma("xmlns:uml",
13              "http://schema.omg.org/spec/UML/2.1", doc));
14             xmiMainNode.Attributes.Append(gma("xmlns:xmi",
15             "http://schema.omg.org/spec/XMI/2.1", doc));
16
17             XmlNode model = doc.CreateElement("uml:Model", "uml");
18             xmiMainNode.AppendChild(model);
19
20             model.Attributes.Append(gma("xmi:type", "xmi", "uml:Model", doc));
21             model.Attributes.Append(gma("xmi:id", "xmi", OXD.getId(), doc));
22             model.Attributes.Append(gma("name", "OutputDiagram", doc));
23
24             XmlNode package = doc.CreateElement("packagedElement");
25             model.AppendChild(package);
26
27             package.Attributes.Append(gma("xmi:type", "xmi", "uml:Package", doc));
28             package.Attributes.Append(gma("xmi:id", "xmi", OXD.getId(), doc));
29             package.Attributes.Append(gma("name", "OutputPackage", doc));
30              …
```

Das oben gezeigte Listing stellt den Umgang mit den allgemeinen Informationen dar, die eine XMI-kompatible XML-Struktur fordert. Zunächst wird in Zeile 3 ein neues leeres XML-Dokument erstellt, in das die Ergebnisse gespeichert werden sollen. Eine Deklaration des Zeichenformats und der XML-Version geschieht in Zeile 5. Zeile 6 zeigt auf, wie einem XML-Dokument ein XML-Knoten über die Methode „appendChild()“ hinzugefügt werden kann. Ein XML-Knoten, der so einem anderen Knoten hinzugefügt wird, ist automatisch ein Kindknoten des Knotens, an dessen Stelle diese Methode aufgerufen wird. So kann programmatisch eine verschachtelte XML-Struktur erstellt werden. Zeile 8 zeigt, wie ein einzelner XML-Knoten erstellt wird, in dem von dem zu schreibenden XML-Dokument ein Kindknoten generiert wird. Der Name und Namespace des Knotens wird per String-Parameter eingefügt. Anschließend kann per „appendChild()“-Aufruf dieser erstellte Knoten mit Informationen an die benötigte Stelle im XML-Dokument eingefügt werden. Ein ähnliches Vorgehen wird im Umgang mit XML-Attributen genutzt. Bis einschließlich Zeile 30 wird in diesem Listing ein XMI-Dokument geschrieben, das einem Komponentendiagramm entspricht, jedoch noch ohne die Informationen über Komponenten oder ähnliches. Der Typ, also ein Komponentendiagramm wird in Zeile 27 über den XML-Knoten mit der Bezeichnungen „uml:Package“ definiert.

Anschließend müssen die Informationen des Vergleichsergebnisses in dieses XML-Format überführt werden. Dazu wird über das Ergebnis-Zwischenmodell iteriert:

```
31 foreach (Komponente kr in Result)
32             {
33                     XmlNode curKr = doc.CreateElement("ownedMember");
34                     curKr.Attributes.Append(gma("xmi:type", "xmi", "uml:Component",
35                     doc));
```

```
36                          string krID = OXD.getId();
37                          curKr.Attributes.Append(gma("xmi:id", "xmi", krID, doc));
38                      curKr.Attributes.Append(gma("name", kr.Name, doc));
39                      package.AppendChild(curKr);
40                          …
41                  }
```

In Zeile 31 wird auf dem Ergebniszwischenmodell „`Result`“, der Export der Vergleicher-Komponente, die Iteration über die Komponenten gestartet. Für jede Komponente wird ein neuer XML-Knoten erstellt (Zeile 33) und mit den entsprechenden Attributen ausgestattet (Zeile 34). Zeile 37 zeigt, dass dem soeben erstellten XML-Knoten nun Attribute hinzugefügt werden, die der aktuellen iterierten Komponente aus dem Ergebniszwischenmodell entnommen werden, in diesem Fall über „`kr.Name`“ (Zeile 37) der Name der aktuellen Komponente. Anschließend wird dieser XML-Knoten an der benötigten Stelle dem XML-Dokument hinzugefügt (Zeile 39).

Neben dem Export der inhaltlichen Daten aus dem Zwischenmodell ist es notwendig, die Ergebnisse des Vergleichs auch graphisch darzustellen. Hierzu sollen die architekturabbildenden Elemente je nach Analyseergebnis eingefärbt und bei Abweichungen die Fehlerbeschreibung als Kommentar angezeigt werden. Im Beispiel des Exportes eines Komponentendiagramms via XMI 2.1 müssen nach diesem Standard die beschreibenden Inhalte des Zwischenmodells und die darstellenden Inhalte getrennt werden. So wird für jedes zu exportierende Element ein weiterer XML-Knoten benötigt, der die Darstellung im Diagramm steuert. Diese Elemente sind innerhalb von XMI 2.1 durch den Tag „`uml:Diagram`“ gekennzeichnet. Innerhalb dieses Tags gibt es eine Sammlung aller Elemente unter dem Tag „`uml:Diagram.element`“, indem alle darzustellenden Elemente aufgeführt sind. Diese Struktur wird in der Implementierung eines Export-Automats erneut durch die oben beschriebenen Operationen eines XML-Dokumentes (vgl. Zeile 3-6) erstellt. Durch die Iteration über das Ergebnis-Zwischenmodell können an dieser Stelle die darstellenden Informationen für jedes Element hinzugefügt werden:

```
40      <uml:DiagramElement geometry="0,0,0,0" preferredShapeType="Component"
41      subject="tLVxAvSGAqAADn0Z" xmi:id="tLVxAvSGAqAADn0Y">
42              <captionBounds value="609,160,20,0"/>
43              <captionVisible value="true"/>
44              <captionSide value="1"/>
45                      <properties>
46                              <abackground value="Cr:0,255,0,255"/>
47                              <aforeground value="Cr:0,0,0,255"/>
48                                      …
49      </uml:DiagramElement>
```

Die Verknüpfung von Inhalten und Darstellung einer Komponente wird über die gegebene ID gesteuert. Daher muss bei der Erstellung dieser Knoten dieselbe ermittelte ID genutzt werden, wie sie die Komponente oder andere Elemente in der vorhergehenden Darstellung erhalten haben. Über die Eigenschaft „abackground“ und „aforeground“ (Zeile 47 + 48) wird die farbliche Darstellung beschrieben. An dieser Stelle wird für jedes Element ermittelt, ob der boolesche Wert „isOK“ aus dem Zwischenmodell wahr oder falsch ist. Im Falle des Wahrheitswertes „wahr“ wird hier der entsprechende Farbcode für eine grüne Darstellung und im Falle des Wahrheitswertes „falsch“ rot als Farbe gewählt. So wird dem Anwender graphisch dargestellt, ob das jeweilige Element korrekt, im Sinne der Soll-Architektur, in der Ist-Architektur implementiert wurde. Bei Elementen, die wiederum geschachtelte Elemente

enthalten können, wie z.B. Komponenten, wurde sich dafür entschieden, einen weiteren Farbcode einzuführen, die Farbe Gelb. Diese repräsentiert, dass das jeweilige Element zwar vorhanden ist, aber nicht alle Unterelemente korrekt implementiert wurden. Bei der Darstellung einer Komponente beispielsweise dann, wenn eine Komponente vorhanden, aber eine angebotenen Schnittstelle nicht vorhanden ist.

Hierzu müssen zur Vergabe des richtiges Farbcodes zunächst alle Unterelemente ausgewertet werden, nur wenn alle den Wahrheitswert bei „`isOk`" mit wahr tragen, hat auch die diese Unterelemente haltende Komponente den Wahrheitswert „wahr" bei „`isOk`". Hierdurch soll es dem Anwender möglich sein, sich auf Problemfelder zu fokussieren. Die fehlende Schnittstelle wiederum wird mit rot markiert, da diese keine architekturbeschreibenden Unterelemente aufweist.

Neben der farblichen Darstellung sollen auch die entsprechenden Fehlermeldungen in das Diagramm eingebunden werden. Dazu werden in diesem Fall die Kommentarfunktionen in XMI genutzt. Diese ist ebenfalls als Tag unter einem architekturelementbeschreibenden XML-Knoten in folgender Form einzufügen:

```
50      <comments>
51          <comment author="doccy" date="03.05.2011 22:51:49" documentation="Inhalte des
52          Kommentars" summary="Hier eine Zusammenfassung" xmi:id="M_GF7fSGAqAADs0P"/>
53      </comments>
```

Im Attribut „`documentation`" können nun alle Fehlermeldungen aus dem Zwischenmodell, die in der Sammlung der „`FailureDescriptons`" enthalten sind, als Wert eingefügt werden. Das Attribut „`summary`" stellt eine Überschrift zur Kategorisierung dar, die in diesem Fall mit dem Wert „Fehlermeldungen" bezeichnet wird. Wenn also ein Element den Wahrheitswert „false" bei „`isOk`" gesetzt hat, ist dies für den Export-Automaten das Steuersignal, dass solch ein Kommentar zu diesem Element hinzugefügt werden muss, welches die entsprechenden Attribute aus den Fehlermeldungen aus dem Zwischenmodell füllt. Über dieses Vorgehen ist es möglich, die Vergleichsergebnisse sowohl graphisch als auch textuell in ein Diagramm zu integrieren, sodass dem Anwender eine Auswertung der Ergebnisse ermöglicht wird.

Nachdem das komplette XML-Dokument vorliegt und alle Strukturen des Ergebnismodells abgebildet wurden, wird das Dokument über „`doc.Save(path);`" gespeichert. Der Pfad des Exportes der Datei wird bei dem Anwender über einen Dialog erfragt.

Die Schwierigkeit beim Export liegt darin, das Ergebnisformat in XML korrekt zu repräsentieren. Es erfordert somit eine eingehende Analyse und Strukturierung. Der Umgang mit der programmatischen Umsetzung durch die bidirektionalen Abbildungsregeln der architekturbeschreibenden Elemente ist einfach umzusetzen.

Das Vorgehen bei dem Export der anderen untersuchten Diagrammtypen ist analog zu dem hier vorgestellten Vorgehen implementiert worden. In allen Fällen wurden Farben und Kommentare eingesetzt um in der Analyse erkannte Übereinstimmungen und Abweichungen hervorzuheben.

9 Auswertung und Optimierung

9.1 Untersuchung der Schichtendiagramme

An dieser Stelle soll die Anwendung des implementierten Werkzeugs zur Schichtendiagramm-Analyse erläutert werden. Hierzu wird das unter Kapitel 5.2 beschriebene Shop-Softwaresystem untersucht, wobei eine Schichtenarchitektur mit vier Schichten implementiert wurde. Das Ergebnisdiagramm wurde in das Visual Studio-spezifische XML-Format exportiert und über Visual Studio eingelesen.

9.1.1 Anwendung und Auswertung der Analyse auf Schichtendiagramme

Als Soll-Architektur ist in dieser Analyse ein Schichtendiagramm mit vier Schichten eingegeben worden (Abbildung 29). Das Diagramm wurde in Visual Studio 2010 erstellt. Das Analysewerkzeug nutzt einen Dialog, sodass der Anwender alle für die Analyse relevanten Daten eingeben muss und anschließend die Analyse starten kann:

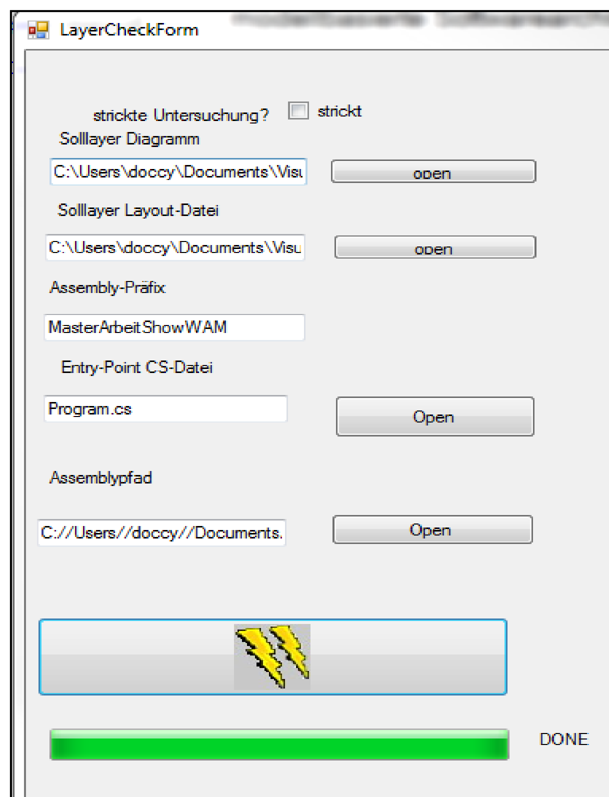

Abbildung 27: Werkzeugoberfläche für Schichtendiagrammanalysen

Bevor die Analyse gestartet werden kann, muss der Diagramm-Pfad zur dem Diagramm, inklusive Layout-Datei, der Präfix der Namespaces und der Pfad zur Assembly angegeben werden. Der Durchlauf der Analyse dauerte in diesem Fall ca. 3 Sekunden. Nach Abschluss der Analyse bestimmt der Anwender, wohin das Ausgabediagramm gespeichert wird. Dieses ist dann erneut über Visual Studio zu öffnen. Das Ergebnis der Analyse des vorgestellten Shop-Systems ist in der Abbildung 28 vorgestellt.

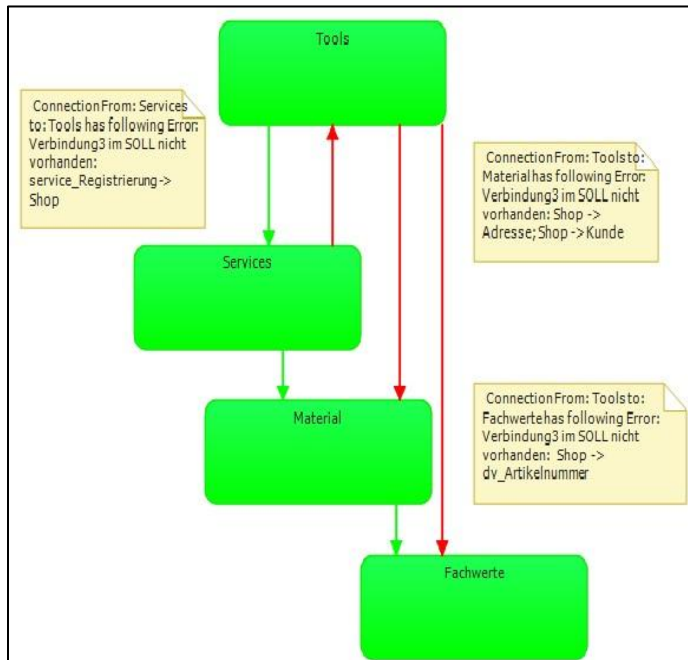

Abbildung 28: Erster Ergebnislauf

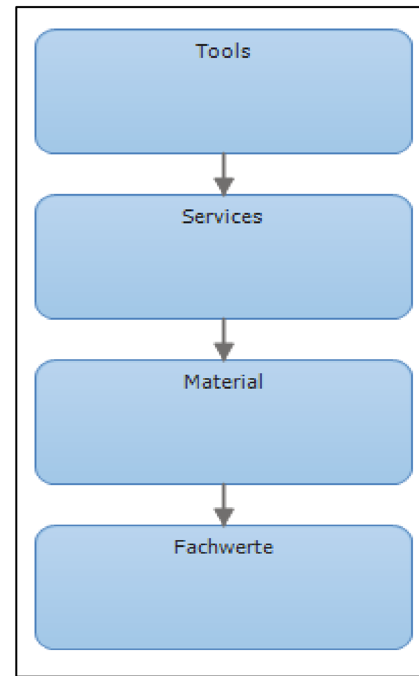

Abbildung 29: Eingabe Schichtendiagramm

Hierbei ist ersichtlich, dass die in Abbildung 29 beschriebenen Schichten aus dem Soll-Diagramm allesamt in der Ist-Architektur nach dem implementierten Vorgehen gefunden werden konnten. Dargestellt durch die grüne Färbung. Ebenfalls ist der grundlegende Zugriff der Schichten wie im Soll-Diagramm implementiert, dies bedeutet, dass Klassen im Quelltext weitestgehend nur auf Klassen zugreifen, die in der Architekturbeschreibung erlaubt sind. Eine Ausnahme bildet die rot markierte Verbindung zwischen der Schicht „Services“ und „Tools“ und die Zugriffe der Schicht „Tools“ auf „Material“ und „Fachwerte“. Die z.B. fehlerhaft implementierte Klasse „`service_Registrierung`“ ist in der Fehlerbeschreibung (Kommentarfeld) aufgeführt. Zudem wird angegeben, auf welche Klasse fehlerhaft zugegriffen wurde. Im Falle der Verbindung zwischen „Tools“ und „Material“ wurden mehrere Abweichungen aufgedeckt, die allesamt in dem Kommentar dargestellt werden. Mit dieser Information kann der Anwender nun prüfen und mit anderen Mitgliedern des Projektteams diskutieren, wie mit diesem Verstoß gegen die Soll-Architektur umgegangen werden soll. Weitere Testläufe mit konstruierten Fehlersituationen, indem Schichten zu der Software hinzugefügt wurden, um zu prüfen, ob nicht oder zu viel implementierte Schichten erkannt werden, wurden ebenfalls ermittelt, was in folgendem Ergebnisdiagramm aufgezeigt werden soll:

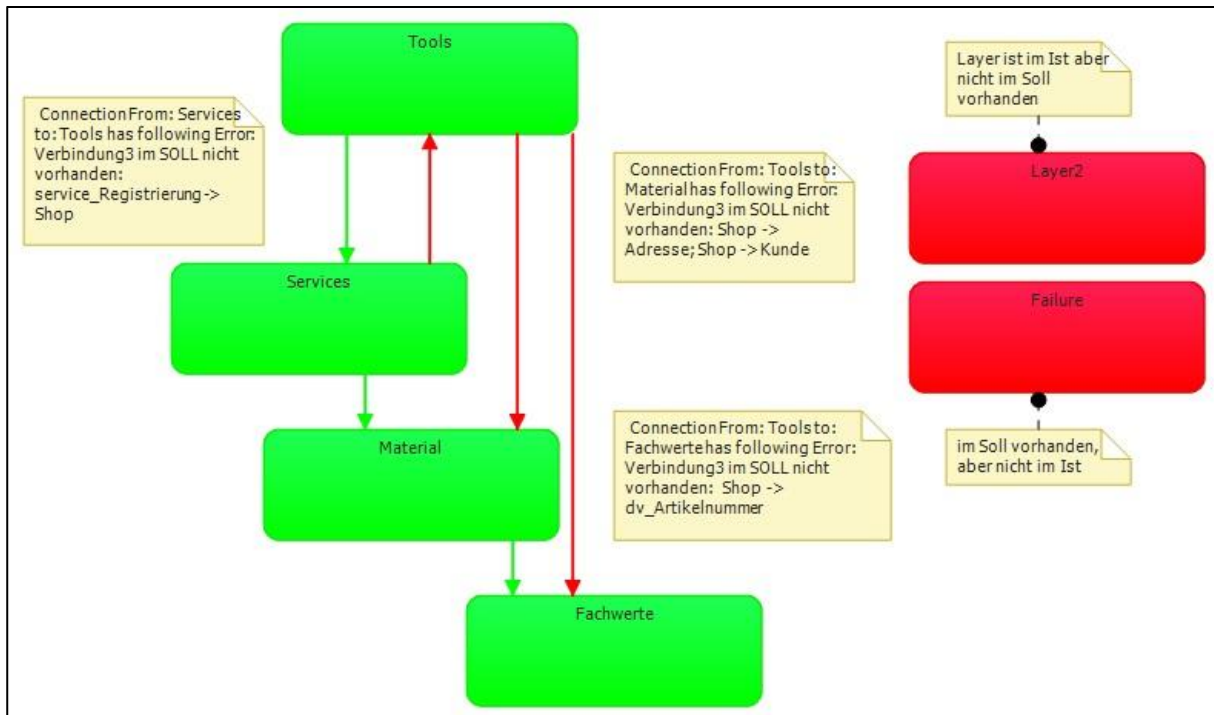

Abbildung 30: Weiteres Analysergebnis für Schichtendiagramme

In diesem Fall ist dem Eingabediagramm (vgl. Abbildung 29) eine weitere Schicht namens „Layer2“ hinzugefügt worden, welche aber nicht in der Ist-Architektur implementiert ist. Des Weiteren ist in der Ist-Architektur eine Schicht namens „`Failure`“ implementiert, welche nicht in der Soll-Architektur spezifiziert ist. Ein weiterer Durchlauf des Analysewerkzeugs stellt das in Abbildung 30 gezeigte Ergebnisdiagramm dar, in dem die fehlerhaften Schichten „`Failure`“ und „Layer“ erkannt und der Fehler ebenfalls korrekt beschrieben wurde.

9.1.2 Optimierung des Prototypen für die Schichtendiagrammanalyse

Schichtendiagramme können zwei Varianten in der angewandten Semantik nutzen. Zum Einen kann der Zugriff exakt so wie modelliert zu verstehen sein, d.h. strikt die ausgehenden Verbindungen. Im Beispiel an Abbildung 29 darf in diesem Fall die Schicht „Tools“ nur auf „Services“ zugreifen, und auf keine andere Schicht. Alle anderen Zugriffe sind als fehlerhafte Verbindungen gekennzeichnet. Zum anderen gibt es eine weichere Sichtweise der Semantik, wobei der Zugriff auf alle unterhalb gelagerten Schichten ermöglicht wird, sofern ein Pfad über die ausgehenden Verbindungen der darunter liegenden Schichten existiert (nicht strikte Semantik). Am Beispiel der in Abbildung 29 vorgestellten Schichten dürfte hier die Schicht „Tools“ auf alle anderen Schichten zugreifen, auch wenn diese nicht explizit modelliert sind, da ein Pfad über die ausgehenden Verbindungen bis zur „Fachwerte“-Schicht ermöglicht ist. Damit der Anwender zwischen den hier vorgestellten Semantiken wählen kann, wird das Werkzeug so erweitert, dass die Auswahl der strikten und nicht strikten Semantik bezüglich der ausgehenden Verbindungen im Analysewerkzeug konfiguriert werden kann.

An der Implementierung der Algorithmen für die Abbildung und des Vergleichs hat sich an dieser Stelle nichts geändert. Es wird bei dem Vergleich der Zwischenmodelle für Schichtendiagramme eine private Methode namens „`optimizeResults`()“ eingeführt, die je nach Anwenderauswahl ausgeführt werden kann. Diese Methode untersucht alle fehlerhaften ausgehenden Verbindungen einer Schicht dahingehend, ob es auch andere Wege über korrekt erkannte ausgehenden Verbindungen gibt. Prinzipiell ist eine Verbindung von der Schicht „Tools“ auf „Material“ (vgl. Abbildung 24) nicht zulässig, da nicht diese nicht modelliert ist. Daher müssen nur die fehlerhaft erkannten Verbindungen untersucht werden, für den Fall, dass der Anwender sich für eine nicht strikte Analyse entschieden hat.

Grundlegend iteriert der genutzte Algorithmus über alle fehlerhaften Verbindungen und prüft zusätzlich, ob ein alternativer Pfad über eine Iteration der korrekt implementierten Verbindungen möglich ist. Dazu werden aus den korrekt (grünen) implementierten Verbindungen die zugehörigen Schichten darauf untersucht, ob diese wiederum korrekt implementierte Schichtenverbindungen haben oder selbst schon die gesuchte Schicht sind. Wird ein Pfad gefunden, so wird die ursprünglich fehlerhaft erkannte Verbindung gelöscht.

Folgende Abbildung zeigt den Ablauf der Analyse auf das Shop-Softwaresystem inklusive der oben beschriebenen implementierten Optimierung. In diesem Durchlauf wurde durch den Anwender entschieden, dass eine nicht strikte Analyse durchzuführen ist, die strikte Analyse wurde bereits vor der Erweiterung (vgl. Abbildung 28 und 29) ermöglicht.

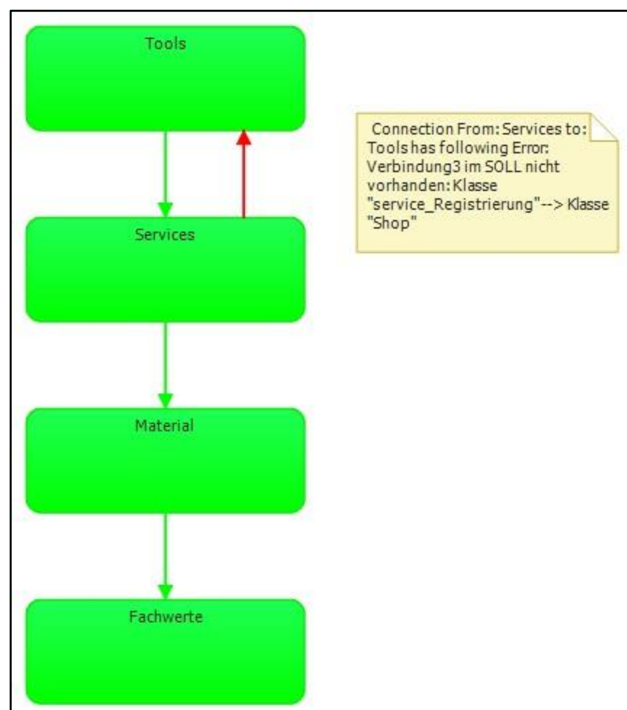

Abbildung 31: Ergebnis der nicht strikten Schichtendiagrammanalyse

Abbildung 31 zeigt nun das Ergebnis der nicht strikten Analyse. Im Vergleich zur strikten Analyse und deren Ergebnissen aus Abbildung 28 konnte nun die fehlerhaften Verbindungen der Schicht „Tools“ hin zu „Material“ und „Fachwerte“ gelöscht werden, da ein alternativer Pfad (in diesem Fall einfach den ausgehenden Verbindungen über die Schichten folgend)

vorhanden ist. Die fehlerhafte Verbindung von der Schicht „Services“ zu „Tools“ bleibt bestehen, da hier kein alternativer Pfad möglich ist.

9.1.3 Bewertung der Ergebnisse bezüglich der Schichtendiagrammanalyse

Die Durchläufe für die Analyse der Schichtendiagramme und die Prüfung, ob diese auch in der Ist-Architektur wiederzufinden sind, können mit dem hier vorgestellten Ansatz erfolgreich durchgeführt werden. Die Abbildung des Eingabediagramms und der Softwarerepräsentation, in diesem Fall über ein kompiliertes Assembly, können dabei wie erwartet angewandt werden. Die Wahl der Variante für die Abbildungsregeln des Assembly auf das Zwischenmodell fällt auf eine Identifizierung über die eingesetzten Namespaces. Jedoch ist die Komplexität der Schichtendiagramme im Vergleich zu den anderen hier genutzten Diagrammarten nicht hoch. Ebenfalls ist die Repräsentation und Diagrammerstellung nicht standardisiert, sodass hier für verschiedenen Anbieter gänzlich andere Repräsentationen gewählt werden können. Des Weiteren werden in diesem hier vorgestellten Ansatz nur horizontale Schichten beschrieben, die nicht geschachtelt sind. So ist es jedoch möglich, die Schichtendiagrammsemantik um vertikale und geschachtelte Schichten zu erweitern. Hierzu müssten sowohl die Zwischenmodelle als auch die Abbildungsregeln entsprechend erweitert oder ausgetauscht werden. Der hier entwickelte Prototyp ermittelt jedoch die erwarteten Ergebnisse, eine Prüfung fand durch die manuelle Sichtung des Quelltextes statt.

Die Erweiterung um die strikte und nicht strikte Semantik-Konfiguration ermöglicht es dem Anwender, die in dem jeweiligen Projekt eingesetzte Semantik analysieren zu können. Weitere Varianten können ebenfalls implementiert werden, indem neue Zwischenmodelle und Regeln gesucht und umgesetzt werden.

9.2 Untersuchung der Klassendiagramme

Die Analyse auf Klassendiagrammebene ist wie in Kapitel 8 beschrieben implementiert. Zur Prüfung, ob der entwickelte Ansatz dieser Arbeit und die Implementierung auf reale Softwareprojekte funktioniert, soll in diesem Abschnitt die Durchführung einer Analyse für Klassendiagramme durchgeführt werden. Hierbei werden neben der Darstellung der Durchführung auch noch weitere Schritte der Optimierung und eine Bewertung der Ergebnisse beschrieben. Der Export des Ergebnisdiagramms wurde für das Modellierungswerkzeug ArgoUML erstellt.

9.2.1 Anwendung der Analyse mit Klassendiagrammen

Für das Shop-Softwaresystem wurde ein Klassendiagramm entworfen, welches als Eingabediagramm für die Analyse dient.

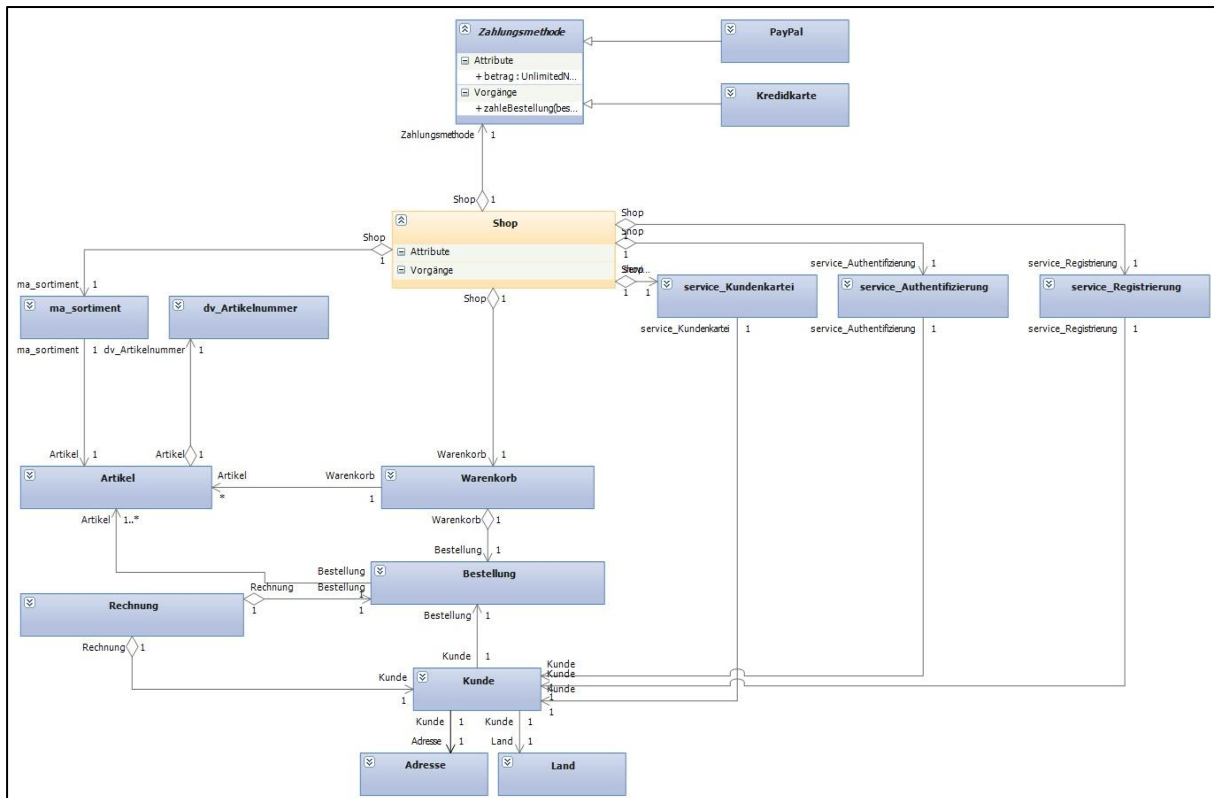

Abbildung 32: Übersicht Klassendiagrammeingabe

Die Abbildung 32 zeigt dieses Klassendiagramm als Übersicht, die Attribute und Methoden sind aus Platzgründen ausgeblendet. Grundsätzlich beschreibt der linke Teil des Diagramms alle Klassen rund um die Artikel in dem Shop-System. In der Mitte sind sowohl die Kundendaten als auch der Warenkorb und die Bestellung modelliert. Auf der rechten Seite sind die Klassen rund um die Kundenbehandlung, wie Registrierung etc. beschrieben. Oberhalb der gelblich dargestellten Shop-Klasse sind noch verschiedene Zahlungsmethoden modelliert.

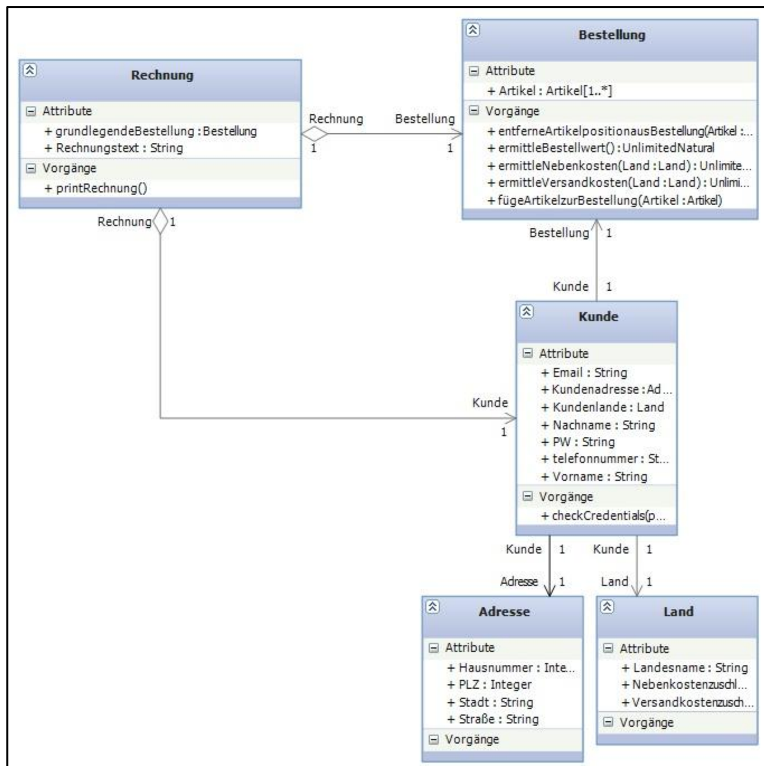

Abbildung 33: Klassendiagammauszug der Detailsicht

Abbildung 33 zeigt fünf einzelne Klassen, bei denen die Attribute und Methoden nicht ausgeblendet sind. Dieses Diagramm dient als Eingabe für die Analyse. Das zu untersuchende Softwaresystem ist dasselbe wie in Abschnitt 9.1.. Das Eingabediagramm ist ebenfalls unter Visual Studio 2010 erstellt worden und der entsprechende Abbilder für diese Anbieter-Diagrammart-Kombination genutzt. Das Ergebnis des Vergleichs, einsehbar in Abbildung 34, ist zunächst sehr unübersichtlich, da sehr viele Verbindungen bestehen und die Inhalte der einzelnen Klassen viel umfangreicher sind als das Soll-Diagramm.

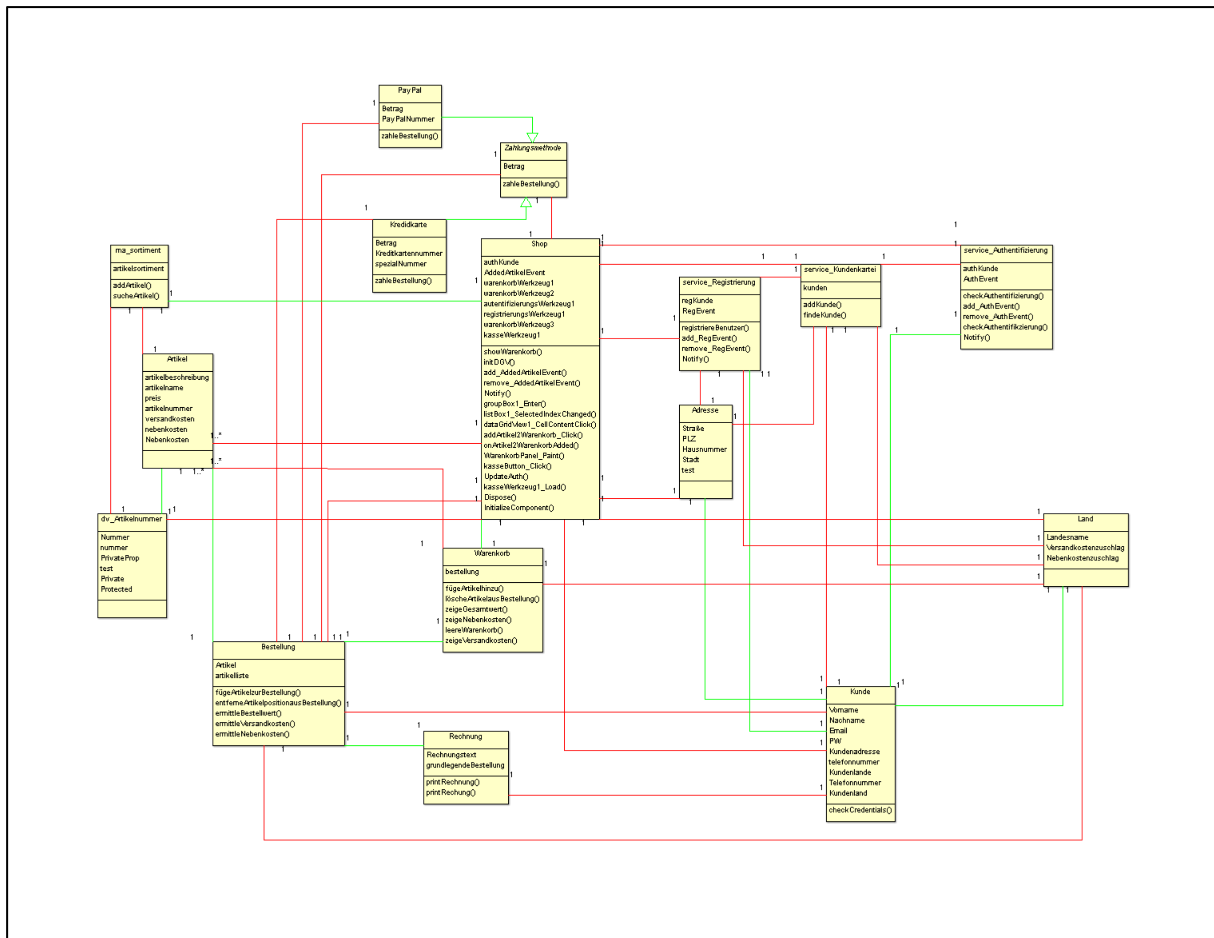

Abbildung 34: Ergebnis der ersten Klassendiagrammanalyse

9.2.2 Auswertung der Analyse mit Klassendiagrammen

Zunächst fällt auf, dass alle Klassen gelb markiert sind, was bedeutet, dass diese Klassen sowohl im Soll als auch im Ist repräsentiert sind und aufeinander abgebildet werden konnten, aber Details, bzw. der exakte Aufbau an Attributen und Methoden nicht übereinstimmen. Mit anderen Worten: Attribute und/oder Methoden der einzelnen Klassen sind nicht wie im Soll definiert umgesetzt worden. Des Weiteren fallen sehr viele rote Verbindungen auf, viel mehr als es grüne Verbindungen gibt. Hier sind, erkenntlich an den Fehlermeldungen, Verbindungen enthalten, die nur im Soll oder nur im Ist existieren. Die Fehlermeldung ist im Dokumentationstext des jeweiligen Elements nachzulesen:

Dokumentation:	Connection ist nicht im IST vorhanden

Abbildung 35: Fehlermeldung einer Verbindung zwischen zwei Klassen

Die Vererbung wurden wie erwartet gefunden und aufgelöst, dies ist im oberen Bereich um die Zahlungsmethoden in Abbildung 34 zu erkennen. Hier sind die implementierten Generalisierungen korrekt eingezeichnet und als korrekt umgesetzt eingestuft worden. Bei

genauerer Betrachtung fällt auf, dass auch technische Methoden, wie Observer-Events, Form-spezifische-Events und auch Getter- und Setter-Methoden aufgeführt werden. Weiterhin fällt auf, dass der benötigte Platz ca. doppelt so groß ist, wie der des Eingabediagramms. Die Übersicht in dieser Form ist begrenzt und ebenso die Handhabung der Verbindungsdarstellung; eine überschneidungsfreie Darstellung war nicht möglich. Die Abbildung der architekturbeschreibenden Elemente auf die Ist-Architektur ist durch die Strukturähnlichkeit wie erwartet durchgeführt worden.

9.2.3 Optimierung des Prototypen für die Klassendiagrammanalyse

Um die Darstellung der Ergebnisse auch fokussierter analysieren zu können und für den Anwender einfacher und verständlicher darzustellen, wird an dieser Stelle ebenfalls versucht, eine Konfigurierbarkeit des Analysewerkzeuges herzustellen. Dazu werden vier Sichten für Klassendiagramme eingeführt (vgl. Sichten unter 3.3.1):

1. Domain Specific View (DSV): nur die Klassennamen und die Verbindungen untereinander werden dargestellt
2. statische Sicht: die Klassennamen und Attribute werden dargestellt, ebenso die Verbindungen untereinander
3. Verhaltenssicht: die Klassennamen und Methoden werden dargestellt, ebenso die Verbindungen untereinander
4. Alles: Alle verfügbaren Informationen werden angezeigt, wie unter 9.2.1 bereits umgesetzt

Diese Sichten sollen es dem Anwender ermöglichen, architekturrelevante Informationen selektiv analysieren zu können, um die Ergebnisse handhabbar zu halten.

Des Weiteren ist in der Analyse der Ergebnisse festgestellt worden, dass neben den fachlichen Inhalten, trotz der bisherigen Filter, technische Informationen dargestellt werden, wie z.B. Getter- und Setter-Methoden und auch Events. Die Kategorisierung in technische und fachliche Inhalte ist an dieser Stelle jedoch nicht trivial, da auch Events fachlicher Natur sein können bzw. fachliches Verhalten beinhalten können. Ebenso ist es anwenderspezifisch, ob Getter- und Setter-Methoden angezeigt werden sollen oder nicht. Aus diesem Grund werden aktivierbare Filter implementiert, die jeweils steuern, ob Getter- und Setter-Methoden und auch Events angezeigt werden sollen oder nicht.

Als letzte Optimierung zum Umgang mit Klassendiagrammen werden noch einzeln aktivierbare Kommentarfunktionen implementiert, mit denen der Benutzer auswählen kann, von welchen Elementen er die Fehlermeldungen angezeigt bekommen möchte und von welchen nicht.

Nach der Implementierung dieser Erweiterungen stellt sich der Benutzungsdialog für den Klassendiagramm-Analysierer wie folgt dar:

Abbildung 36: Klassendiagrammanalysierer Dialog inkl. Optimierungen

Im Folgenden wird ein Durchlauf mit aktivierter statischer-Sicht ausgeführt:

Abbildung 37: Statische Sicht- Fokus Klassendiagrammvergleich

Zum Vergleich hier ein Ergebnis des Durchlaufs mit DSV-Sicht:

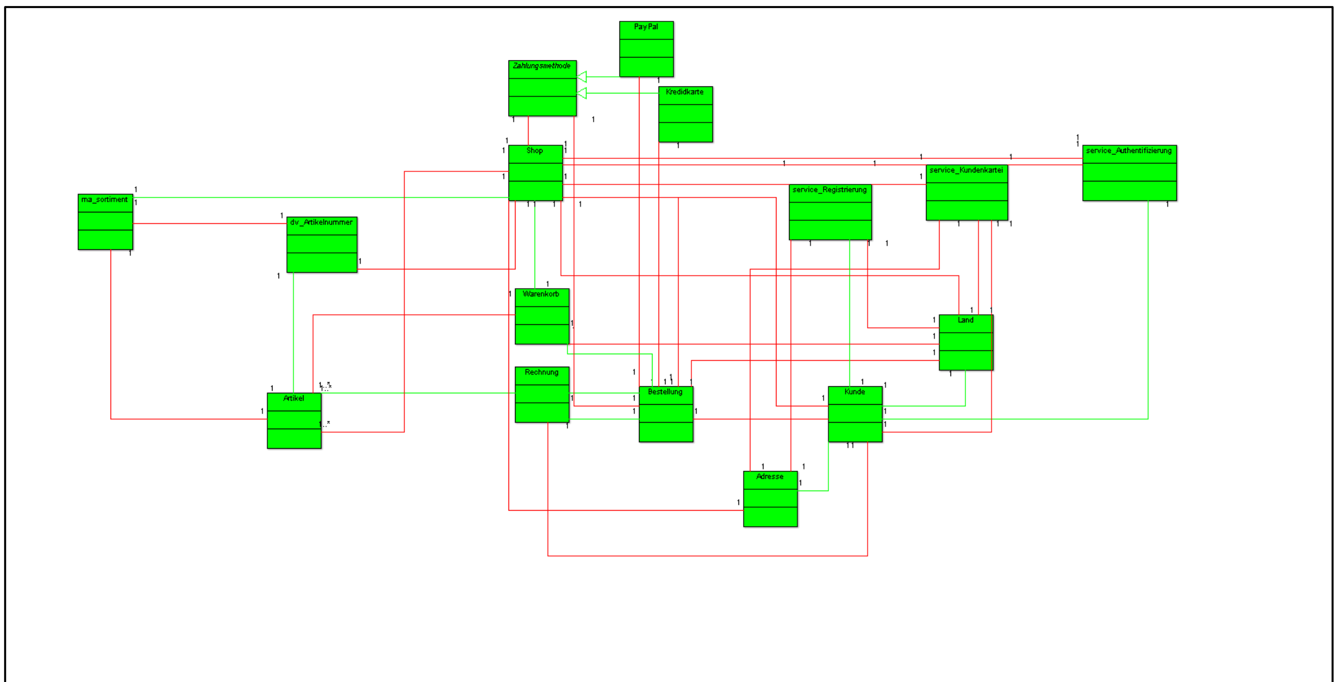

Abbildung 38: DSV-Fokus Klassendiagrammvergleich

Die Analyse der Verbindungen bleibt immer gleich, lediglich die Klassen-bezogene Interpretation der Ergebnisse ändert sich. So ist aus der Analyse der DSV-Sicht (Abbildung 38) zu entnehmen, dass alle modellierten Typen auch implementiert wurden, es wurden keine ausgelassen oder hinzugefügt. In diesem Fall wären die Klassen rot markiert und somit sofort für den Anwender ersichtlich. In der Abbildung 37 ist die statische Sicht als Fokus eingeschaltet, wobei die Attribute ebenfalls untersucht werden. Hier ändert sich die farbliche Darstellung im Gegensatz zur DSV-Sicht sehr stark. Dies liegt daran, dass von den 16 dargestellten Klassen nur sechs alle Attribute wie im Soll-Diagramm beschrieben implementiert haben. Abweichungen, seien es auch nur Sichtbarkeiten oder Datentypen, werden als Abweichung registriert und entsprechend farblich dargestellt.

Eine weitere Optimierung stellt die Erkennung von bidirektionalen Verbindungen dar. Da in der Analyse nur die ausgehenden Verbindungen einer Klasse analysiert werden, kann es sein, dass Klasse A eine Verbindung zu Klasse B aufweist, Klasse B jedoch auch eine Verbindung zu Klasse A. Dies würde derzeit anhand zweier Verbindungspfeile in das Ergebnisdiagramm eingezeichnet werden. Klassendiagramme erlauben aber die Darstellung von bidirektionalen Verbindungen, wobei die gegenseitige Benutzung aufzeigt. Hierzu müssen zu jeder zugegriffenen Klasse die ausgehenden Verbindungen überprüft und die Klassennamen der verbundenen Klassen festgehalten werden. In einem zweiten Schritt werden nun von diesen Klassen, deren Namen festgehalten wurden,die ausgehenden Verbindungen überprüft, ob eine Verbindung zu der untersuchten Klasse vorhanden ist. Ist dies der Fall, kann eine Verbindung gelöscht und die Multiplizität in der nicht gelöschten Verbindung unter „`targetMultiplicity`“ eingetragen werden. Hierdurch können überflüssige Verbindungen gelöscht und die Übersichtlichkeit erhöht werden.

Hier der Auszug aus dem Quelltext, der dieses Vorgehen beschreibt:

```
1         foreach (ClassDiagrammClass idt in Result)
2             {
3                 foreach (Connection sourceAsso in idt.connections)
4                 {
5                   ClassDiagrammClass target =
6                   getTypeFromRESULTByName(sourceAsso.toTypeName);
7                    List<Connections> temp = new List<Connections>();
8                    foreach (Connections targetAsso in target.connections)
9                    {
10                       if (targetAsso.toTypeName.Equals(idt.Name))
11                       {
12                         sourceAsso.sourceMultiplizity =
13                        targetAsso.targetMultiplizity;
14                           temp.Add(targetAsso);
15                       }
16                   }
17
18                   if (temp.Count != 0)
19                   {
20                       foreach (Connections temoasso in temp)
21                       {
22                           target.connections.Remove(temoasso);
23                           counter++;
24                       }
25                   }
26               }
27           }
```

Durch dieses Vorgehen konnten in diesem Testlauf zwei Verbindungen aufgelöst und in eine bidirektionale Verbindung umgewandelt werden.

9.2.4 Bewertung der Ergebnisse bezüglich der Klassendiagrammanalyse

Der aktuelle Stand der Realisierung der Untersuchung von Klassendiagrammen an dem gewählten Beispiel des Shop-Systems hat den Stand, dass in den meisten Fällen nur ein einzelnes Attribut abweicht, oder ein wiederkehrender Fehler, wie z.B. die Erkennung des Datentypens „double" als Abweichung registriert wird. Somit ist mittels der Analyse eine Bewertung der Schwere der aufgetretenen Fehler möglich und Fehlerbehebungen können geplant und begonnen werden.

Bei der Analyse der Klassendiagramme wird deutlich, dass während der Modellierung die fachlichen Aspekte im Fokus liegen, bei der Programmierung hingegen neben den fachlichen auch die technischen Aspekte eine Rolle spielen. Dies schlägt sich in der Analyse insofern nieder, dass technische Aspekte aufgedeckt werden, und eine Unterscheidung zwischen fachlichen und technischen Elementen nicht immer eindeutig möglich ist, wie z.B. bei dem Umgang mit Events oder Setter- und Getter-Methoden.

In der Implementierung des Analysewerkzeugs war das Klassendiagramm durch die Strukturähnlichkeit mit der Ist-Architektur-Repräsentation am einfachsten von allen untersuchten Diagrammarten abzubilden. Das Fehlen der Unterscheidbarkeit zwischen den

Verbindungsarten in Klassendiagrammen, welche nicht ohne Programmierstil-Konventionen im Quelltext abgebildet werden können, ist im Ergebnis deutlich bemerkbar. Die Aussagekraft der verschiedenen Verbindungsarten fehlt gänzlich. An dieser Stelle wäre eine Unterscheidung der Verbindungsarten wünschenswert, um die Modellierungstiefe auch in der Analyse darstellen und in der Ist-Architektur aufdecken zu können.

Leider ist es nicht möglich, einzelne Methoden oder Attribute farblich zu markieren, dies würde die Übersicht über die Abweichungen erhöhen, da nur dann sofort erkennbar wäre, welches Element fehlerhaft implementiert ist.

In dieser Arbeit wird die Frage der Skalierbarkeit nicht untersucht. Bei sehr großen Softwaresystemen und eventuell vorhandenen Modellierungen in Form von Klassendiagrammen, zumindest für Teile des Softwaresystems, kann durch die geschachtelten Vergleiche ein Laufzeitproblem auftreten. Jedoch kann dies durch eine Optimierung der Implementierung zumindest minimiert werden. Performanz-Messungen sind nicht erstellt worden, da keine empirischen Auswertungen gemacht wurden. Im Vordergrund dieser Arbeit stehen die Machbarkeitsanalyse und die Erstellung.

Neben den genannten Problemfeldern werden die modellierten Klassen, Attribute und Methoden erkannt und mit dem Quelltext verglichen. Der Vergleich funktioniert mit den angegebenen Regeln erwartungskonform. Ebenso ist die Erkennung der Verbindungen wie definiert durchgeführt worden. Eine manuelle Analyse der Software ergab dieselben Ergebnisse, sodass für das vorliegende Softwaresystem nach den hier aufgestellten Regeln die Analyse korrekt gelaufen ist. Testläufe neben der hier dargestellten und untersuchten Software ergaben auch die korrekte Darstellung von Schnittstellenimplementierungen. Diese werden erkannt und ebenfalls eingezeichnet. Ebenfalls werden nicht spezifizierte und nicht implementierte Klassen erkannt, als fehlerhaft gekennzeichnet und mit einer entsprechenden Fehlermeldung versehen.

9.3 Untersuchung der Komponentendiagramme

In diesem Abschnitt wird das entwickelte Analysewerkzeug für Komponentendiagramme angewendet und die Benutzung und die Ergebnisse werden analysiert. Das Ergebnisdiagramm wurde im XMI-Standard exportiert und über Visual Paradigm eingelesen.

9.3.1 Anwendung der Analyse auf Komponentendiagramme

In diesem Beispiel wird nicht das bisher genutzte Shop-Softwaresystem untersucht, da dieses bereits eine Schichtenarchitektur implementiert und nicht an dem Komponentenmodell orientiert ist. Als Referenzsoftware wurde hierzu ein Beispiel entworfen, welches dahingehend untersucht werden soll, ob folgendes Komponentendiagramm (Abbildung 39) implementiert wurde:

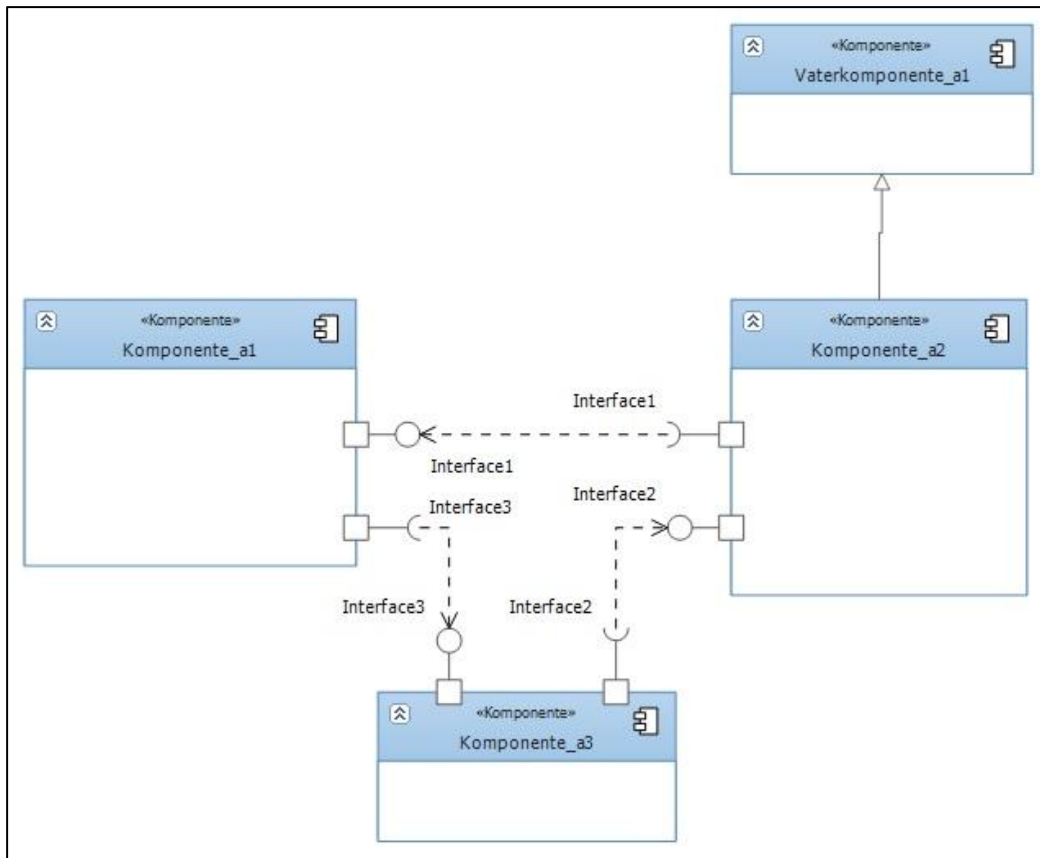

Abbildung 39: Soll-Komponentendiagramm für die Analyse

Hierbei handelt es sich um vier Komponenten, bei denen jede Komponente eine angebotene und eine benötigte Schnittstelle implementieren soll. Eine Komponente, die „`Vaterkomponente_a1`", wurde implementiert, um zu prüfen, ob die Vererbungsbeziehung korrekt erkannt wird. Wie bereits beschrieben, ist in diesem Prototypen nur die äußere Sicht von Komponentendiagrammen beschrieben, da die interne Struktur, bis auf einige Erweiterungen, wie die Delegation von Schnittstellen hin zu internen Klassen, aus weiteren Komponenten und Klassen besteht. Die Untersuchung von Klassen, beziehungsweise Klassendiagrammen, wurde bereits in Kapitel 9.2 beschrieben und wäre hier analog zu implementieren.

Die zu untersuchende Software soll dieses Komponentendiagramm implementieren. Dabei wurde das unter Kapitel 7.1.3 beschriebene Fassadenpattern eingesetzt, indem eine Klasse nach außen hin eine Komponente darstellt. Alle eingehenden und ausgehenden Verbindungen sollen über die Fassadenklasse geleitet und gesteuert werden. Die angebotenen Schnittstellen sind als genutzte Schnittstellen und die benötigten Schnittstellen über die Verwendung in den Konstruktoren der Fassadenklasse definiert.

Die Fassadenklasse „`Komponente_a2`" ist wie folgt (verkürzt) aufgebaut:

```
1  class Komponente_a2:Vaterkomponente_a1, IComponent, Interface2
2      {
3
4          private Interface1 schnittstellenTyp;
5
6          public Komponente_a2(Interface1 if1)
```

```
7          {
8              schnittstellenTyp = if1;
9          }
10
11         #region IComponent Member
12
13         #region IDisposable Member
14
15         #region Interface2 Member
16
17         public string ToString()
18         {
19             return schnittstellenTyp.ToString();
20         }
21
22         #endregion
23     }
```

Zeile 1 beschreibt die Klassendefinition der „Komponente_a2", indem hier definiert ist, dass diese Fassadenklasse von der „Vaterkomponente_a2" erbt und die Schnittstellen „IComponent" und „Interface2" implementieren soll. In Zeile 6 ist der Konstruktor der Fassadenklassen beschrieben. Hier wird die Schnittstellen „Interface1" als Parameter gefordert. Die Zeilen 9 – 14 zeigen die Implementierung der Schnittstelle, beziehungsweise deren beschriebene Methode „ToString()" und deren konkrete Implementierung in der Fassadenklasse. Bei einer Untersuchung der inneren Struktur würde die Implementierung der „ToString()"-Methode an eine geschachtelte Klasse delegiert werden.

Dieser Aufbau einer Komponente ist für alle anderen beschriebenen Komponenten ebenfalls durchgeführt worden und an einigen Stellen sind bewusst Fehler eingebaut worden, um zu prüfen, ob Fehlerfälle gefunden werden und ob die Fehlermeldungen korrekt wiedergegeben werden kann.

Ein Durchlauf der Analyse auf das Assembly der Software hat folgendes Ausgabediagramm ergeben:

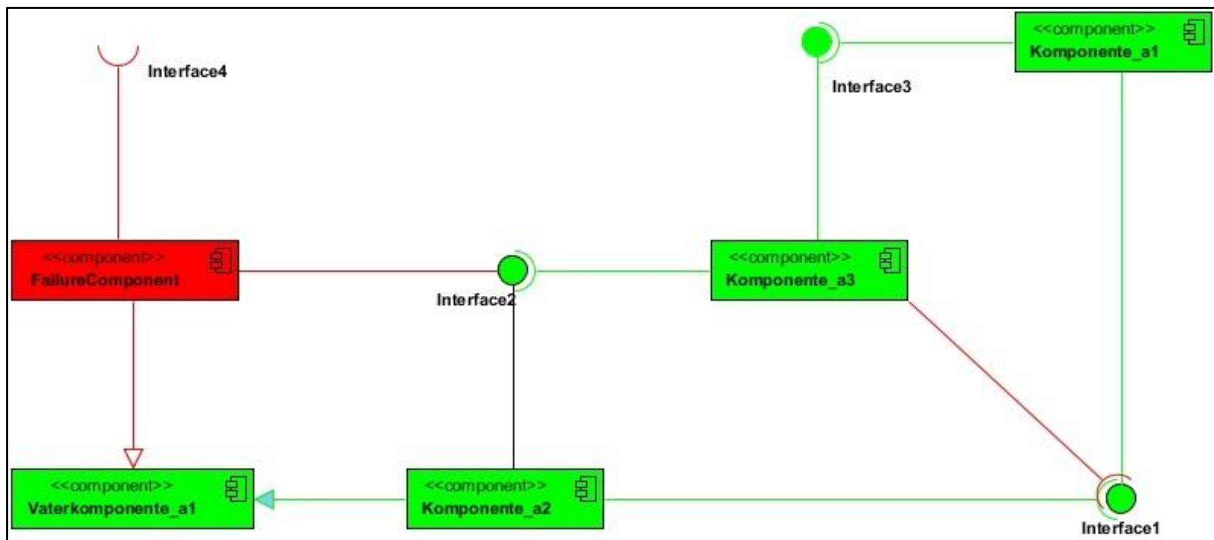

Abbildung 40: Ergebnis-Komponentendiagramm nach der Analyse

Hierbei ist durch die farbliche Markierung zu erkennen, dass alle Grundkomponenten, die im Soll-Diagramm enthalten sind, in der Software realisiert sind. Ebenfalls sind alle Schnittstellen wie beschrieben umgesetzt worden. Lediglich die Nutzung der Schnittstelle „`Interface1`“ als benötigte Schnittstelle der „`Komponente_a3`“ ist nicht spezifiziert und daher rot eingefärbt. Ebenfalls ist eine Komponente namens „`FailureComponent`“ implementiert, die ebenfalls nicht spezifiziert ist, inkl. einer nicht spezifizierten Schnittstelle namens „Interface4“.

9.3.2 Auswertung der Analyse auf Komponentendiagramme

Alle genutzten Komponenten wurden durch das Analysewerkzeug in dem Software-Assembly erkannt und auf das entsprechende Zwischenmodell überführt. So wurde ein Vergleich der beiden Zwischenmodelle möglich und hat das in Abbildung 40 dargestellte Ergebnis erzeugt. Alle bewusst eingefügten Fehlerkonstellationen wurden sowohl korrekt erkannt und farblich hervorgehoben als auch mit den entsprechenden Fehlerkennungen versehen.

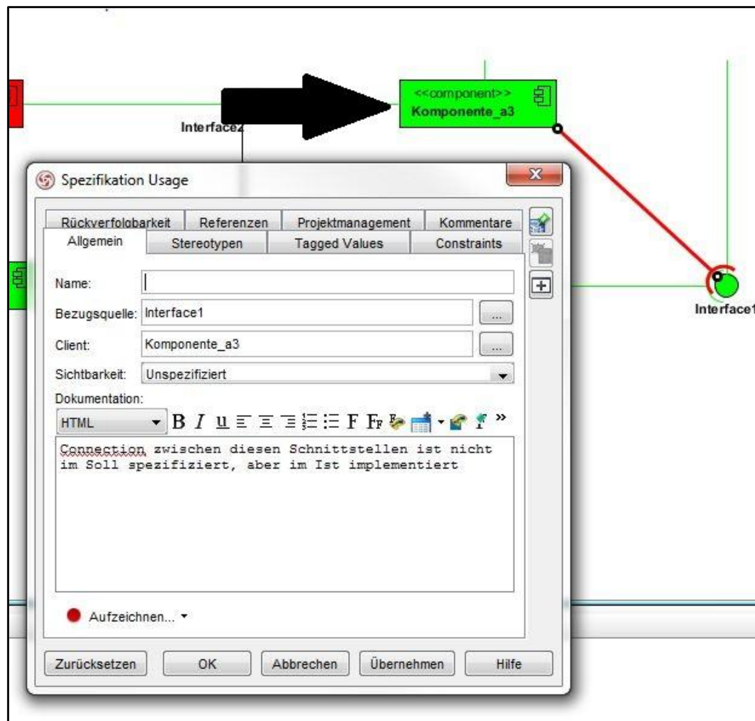

Abbildung 41: Fehlermeldung Darstellung in dem Ergebniskomponentendiagramm

Abbildung 41 zeigt die Darstellung der Fehlermeldung der implementierten Schnittstelle „`Interface1`" von „`Komponenten_a3`", indem beschrieben wird, dass diese Verbindung zu der Schnittstelle nicht im Soll-Diagramm beschrieben ist. Ähnliche Fehlermeldungen wurden bei der Vererbung oder bei Komponenten, welche nicht in der Soll-Architektur spezifiziert sind, erstellt. Das Analyseergebnis ist erwartungskonform und stellt den Vergleich der Soll-Architektur und eine Softwareimplementierung als Repräsentant einer Ist-Architektur für das gewählte Beispiel korrekt dar.

9.3.3 Optimierung des Prototypen für die Komponentendiagrammeanalyse

Neben der Analyse der rein darstellenden Elemente basierend auf der Komponentendiagramm-Notation aus der Soll-Architektur können weitere semantische Architekturregeln für Komponentendiagramme analysiert werden. So stellt eine Komponente eine Blackbox für die auf die Komponente zugreifenden Klassen dar, die nur das Verhalten publik machen soll, welches über die implementierten Schnittstellen spezifiziert wurde. Jede darüber hinaus öffentlich verfügbare Methode einer Komponente würde einen Verstoß gegen diese Regel darstellen. Dies betrifft ebenfalls die äußere Sicht einer Komponente. Als Optimierung des Analyseverfahrens ist eine Erweiterung in die Analyse eingebaut, die prüft, ob alle Methoden oder Attribute der Fassadenklasse, die öffentlich aufrufbar und somit als „public" deklariert wurden, auch in den implementierten Schnittstellen definiert sind. Ist dies nicht der Fall, so muss eine Fehlermeldung erstellt werden und der Status für „`isOk`" und somit die farbliche Repräsentation der Komponente geändert werden. Dies ist nicht explizit in den Komponentendiagrammen modelliert, sondern ist Teil der Semantik der Komponentenmodelle.

Die Erweiterung ist als private Methode namens „`checkIllegalPublicMethods`()“ implementiert, sodass die Abbildungsfunktionen nicht geändert werden mussten. Der Ablauf dieser Methode lässt sich wie folgt darstellen:

1. Alle implementierten Schnittstellen einer Fassadenklasse ermitteln
2. Alle Attribute und Methoden der Schnittstellen in ein Dictionary mit dem Schnittstellennamen als Key und den Methoden und Attributnamen als geschachteltes Dictionary speichern
3. Über alle öffentlichen Felder und Eigenschaften der Fassadenklasse iterieren und prüfen, ob die Namen in dem oben beschriebenen Dictionary enthalten sind
    a. Wenn ja, passiert nichts
    b. Wenn nein, dann Fehlerbeschreibung an das Attribut im Zwischenmodell schreiben und Status „`isOk`“ als falsch setzten
4. Über alle öffentlichen Methoden iterieren und mit dem oben beschriebenen Dictionary vergleichen, ob der Name enthalten ist
    a. Wenn ja, passiert nichts
    b. Wenn nein, dann Fehlerbeschreibung an die Methode im Zwischenmodell schreiben und Status „`isOk`“ als falsch setzten
5. Vorgang abschließen

Mittels dieser Beschreibung wird ein Algorithmus zur Erkennung der nicht spezifizierten öffentlichen Member (Attribute und Methoden) erstellt. Ein erneutes Ausführen des Analysewerkzeugs auf die weiter oben beschriebene Kombination aus Eingabediagramm und Software-Assembly hat folgendes Ergebnis generiert:

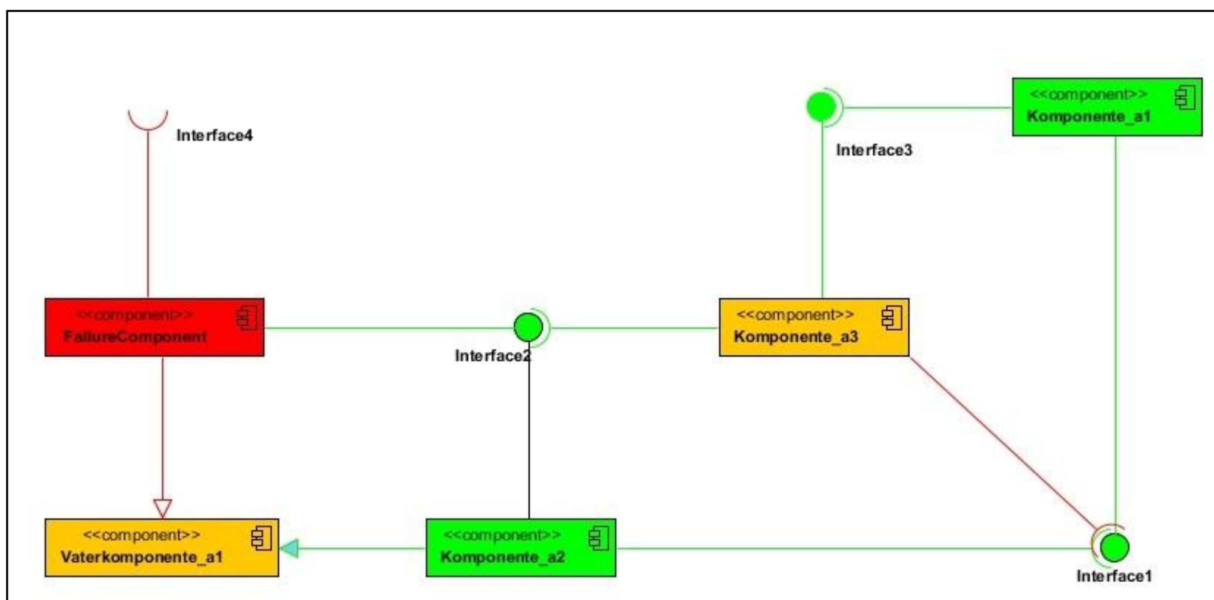

Abbildung 42: Ergebnis des optimierten Komponentdiagrammanalysewerkzeugs

Das Ergebnis zeigt, dass die Komponenten „`Vaterkomponente_a1`“ und „`Komponente_a3`“ nun gelb akzentuiert sind. Dies bedeutet, dass die Komponente prinzipiell korrekt vorhanden und erkannt worden ist, jedoch ein interner Teil nicht korrekt implementiert wurde. Die Fehlermeldung der „`Komponente_a3`“ beschreibt, welche Teile nicht korrekt umgesetzt wurden:

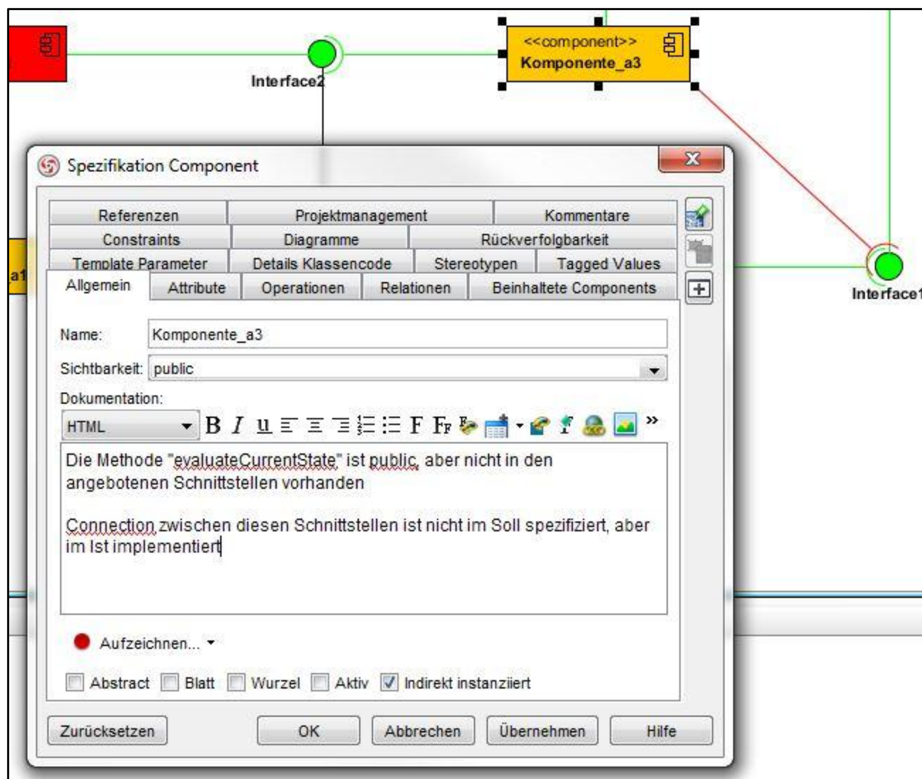

Abbildung 43: Fehlermeldung aus optimierter Komponentenanalyse

Hier wird beschrieben, dass eine Methode namens „`evaluateCurrentState`()“ öffentlich zugänglich ist, aber nicht in einer der implementierten Schnittstellen definiert wurde. Anhand solch einer Fehlermeldung soll es dem Anwender ermöglicht werden, die Fehlersituation aufzulösen.

9.3.4 Bewertung der Ergebnisse bezüglich der Komponentendiagrammeanalyse

Die äußere Sicht einer Komponente wird in diesem definierten Fall der Abbildungsregeln korrekt erkannt. Und alle Fehler, die bewusst implementiert wurden, werden aufgedeckt und mit einer entsprechenden Fehlermeldung versehen. Die eingebauten Erweiterungen und die Optimierung funktionieren ebenfalls wie erwartet. Zusätzliche Stufen der Optimierung und die Erweiterung um die innere Darstellung sind Punkte, die in weiteren Prototypen implementiert oder in weitergehenden Arbeiten untersucht werden sollen.

Zudem wird in diesem Fall eine definierte Variante der Abbildung von Komponenten in Softwaresystemen angewandt. Es gibt aber kein standardisiertes Vorgehen für die Implementierung von Komponenten, sodass in dieser Anwendung des Analysewerkzeugs nur die Variante der „IComponent“-Schnittstellen-Realisierung und Fassanden-Entwurfsmuster-Implementierung untersucht worden ist. In verschiedenen Softwareprojekten gibt es sehr wahrscheinlich weitere, in dieser Arbeit nicht beschriebene, Vorgehensweisen zur Implementierung von Komponenten, sodass die aufwändige Erstellung eines Komponentenanalyse-Werkzeugs für jede Abbildungsvariante neu implementiert werden muss. Dies ist ein hoher Aufwand, da die Wiederverwendung einzelner Teile, bedingt durch die fehlende direkte Abbildbarkeit von Komponenten auf die Software, wahrscheinlich eingeschränkt ist.

10 Abschließende Betrachtung

In diesem Kapitel wird der erreichte Stand zusammengefasst und ein Ausblick gegeben, welche offenen Anforderungen oder Verbesserungen in weiterführenden Arbeiten behandelt werden können.

10.1 Zusammenfassung

Auf die einleitende Frage, ob ein direkter Vergleich zwischen Diagrammen und einer Softwarerealisation machbar ist, wurden in dieser Arbeit Ansätze entwickelt, die diesen Vergleich ermöglichen.

Dazu wurden verschiedene Problemstellungen erörtert, die die einzelnen Schritte einer automatisierten modellbasierten Softwarearchitekturanalyse beschreiben. Nach der Analyse dieser Problemfelder wurde ein Ansatz entwickelt, der darauf basiert, dass sowohl das eingegebene Diagramm als auch der Repräsentant des Quelltextes als Modelle zu verstehen sind. Um nun eine Vergleichbarkeit und Abbildbarkeit der Modelle aufeinander herstellen zu können, ist ein sogenanntes Zwischenmodell entwickelt worden, in das die Diagramme und Softwarerepräsentanten transformiert werden.

Dieses Zwischenmodell muss zunächst für jeden unterstützen Diagrammtypen entwickelt werden. Dazu werden die architekturbeschreibenden Elemente aus den unterstützen Diagrammen identifiziert, z.B. eine Komponente bei einem Komponentendiagramm oder eine Schicht bei einem Schichtendiagramm. Alle zu untersuchenden Informationen für diese architekturbeschreibenden Elemente werden in dem Zwischenmodell modelliert und anschließend in eine Klassenstruktur programmiert. An die Erstellung des Zwischenmodells schließt die Transformation der Diagramme und Softwarerepräsentanten an. Zudem ist die Abbildung der Eingaben auf das jeweils genutzte Zwischenmodell erforderlich.

Hierzu wurden für Klassen-, Komponenten- und Schichtendiagramme Abbildungsregeln erstellt. Für die Ist-Architekturrepräsentation konnten ebenfalls durch den Zugriff über die Reflections API Abbildungsregeln erstellt werden.

Durch die Anwendung der Abbildungsregeln ist es möglich, die Eingaben in das Zwischenmodell zu transformieren und die gesammelten Informationen, z.B. aus welchen Typen oder Komponenten die Soll- und Ist-Architektur besteht, miteinander zu vergleichen. Da die definierten Zwischenmodelle für die Kombination aus Diagramm und Softwarerepräsentation gleich sind, ist ein nachvollziehbarer und handhabbarer Vergleich möglich.

Für die Ergebnisdarstellung wurden diese Abbildungsregeln auf das Ergebnis des Vergleichs angewandt. Da diese Regeln bidirektional sind, d.h. sowohl vom Diagramm hin zum Zwischenmodell als auch in der inversen Anwendung, ist eine Ableitung eines Ergebnisdiagramms möglich. In diesem Ergebnisdiagramm wurden Informationen über Abweichungen zwischen der Soll- und Ist-Architektur sowohl farblich als auch textuell hinterlegt.

Neben der Entwicklung des soeben beschriebenen Vorgehens zur modellbasierten Softwarearchitekturanalyse ist ein Bestandteil dieser Arbeit eine prototypische Umsetzung und eine Anwendung dieses Vorgehens. Hierzu wurden zwei Softwaresysteme, ein Shop-System und ein komponentenbasiertes System, auf die Einhaltung einer definierten Soll-Architektur, geprüft. Die Soll-Architektur wurde für das Shop-System als Klassen- und Schichtendiagramm, und für das komponentenbasierte System als Komponentendiagramm modelliert.

Bei der Anwendung auf das Shop-System, mit dem Fokus auf die Schichteneinhaltung, wurden alle modellierten Schichten und Verbindungen erkannt, ebenso wie alle bekannten Abweichungen von der Ist- zur Soll-Architektur. Zur Optimierung wurde für das Analysewerkzeug der Schichtendiagramme eine Erweiterung vorgestellt, die es ermöglicht, sowohl eine strikte als auch ein nicht-strikte Schichtensemantik zu unterstützen.

Neben der Untersuchung auf die Schichtenarchitektur ist ein Klassendiagramm, welches die statischen Aspekte des Shop-Systems beschreibt, als weitere Anforderung an die Architektur auf das Shop-System angewandt worden. Hierbei wurden ebenfalls alle beschriebenen Klassen und Abweichungen, innerhalb und auf die Klasse an sich bezogen, erkannt. Da die Verbindungssemantik bzw. die Ausdrucksstärke von Verbindungen in der Programmiersprache C# jedoch nicht auf der Semantik der Verbindungen in Klassendiagrammen abzubilden ist, war es nur möglich, Assoziationen, nicht jedoch Aggregationen und Kompositionen, zu ermitteln.

Die Untersuchung, ob Komponentendiagramme als Repräsentation einer Soll-Architektur direkt mit einem Softwaresystem vergleichbar sind, wurde anhand eines komponentenbasierten Softwaresystems geprüft. Die Erkennung der Komponenten und Verbindungen zueinander, nach den hierfür definierten Abbildungsregeln, funktionierte für die äußere Sicht von Komponenten. Dabei wurden neben den Komponenten die angebotenen und benötigten Schnittstellen als auch Vererbungsbeziehungen erkannt. Die innere Sicht, welche wiederum aus Komponenten oder Klassen und Verbindungen zueinander besteht, wurde zu dem aktuellen Stand der Arbeit nicht behandelt. Die grundlegende Erkennung von Klassen, bzw. der Vergleich mit Klassendiagrammen und die Ermittlung von Komponenten, wurden bereits erfolgreich behandelt und können somit in nachfolgenden Schritten umgesetzt werden.

Grundlegend ist eine Abbildung eines Softwaresystems, z.B. repräsentiert durch den Quelltext, auf ein Diagramm als Soll-Architekturbeschreibung mit dem in dieser Arbeit vorgestellten Ansätzen möglich. Die Schwierigkeit liegt darin, geeignete Repräsentanten von architekturbeschreibenden Elementen wie einer Komponente, im Softwaresystem zu finden, mit dem dieses Element dann verglichen werden kann. Diese Abbildungsregeln können in verschiedenen Softwareprojekten unterschiedlich definiert sein, sodass spezielle Implementierungen des Analysewerkzeugs nötig sein können.

10.2 Weiterführende Arbeiten

Die Hauptprobleme in der Umsetzung des in dieser Arbeit vorgestellten Ansatzes sind der Umgang mit Verbindungen zwischen Klassen in einer Programmiersprache und die Wiederverwendung aufgestellter Abbildungsregeln in verschiedenen Softwareprojekten.

Für die Auflösung von Verbindungen in Softwaresystemen wurde eine eigene Softwarekomponente entwickelt, welche diese Erkennung durchführt. An dieser Stelle erhöht sich die Qualität der Ergebnisse für Klassendiagramme, würden Möglichkeiten entwickelt, die am Quelltext eine Unterscheidung zwischen einer Assoziation, Aggregation und Komposition durchführen könnten.

Eine weiterführende Evaluierung an realen Softwareprojekten sollte durchgeführt werden, um herauszufinden, ob eine Anwenderakzeptanz für dieses vorgestellte Vorgehen erreicht wird. Des Weiteren kann solch eine Evaluation genutzt werden, um Erkenntnisse und Verbesserungen in den Bereichen Skalierbarkeit und Performanz zu erlangen.

Eine Erhöhung der Wiederverwendung von Abbildungsregeln minimiert den Entwicklungsaufwand für die Erstellung eines modellbasierten Softwarearchitekturanalyse-Werkzeugs für Komponenten- und Schichtendiagramme als Soll-Architekturdarstellung. Dazu könnten Best Practices für die Ermittlung und Realisierung von Komponenten und Schichten aus verschiedenen Softwareprojekten ermittelt werden. Mit diesen Best Practices wäre eine Ermittlung definierter und anwendbarer Varianten für die Abbildungsregeln auf die genannten Diagrammtypen und die Implementierung dieser möglich.

Neben den soeben vorgestellten Hauptproblemen und den möglichen Ausblicken auf Lösungsansätze für diese wäre eine Entwicklung hin zu einem Framework für modellbasierte Softwarearchitekturanalysen wünschenswert. In dieser Arbeit wurden für drei Diagrammtypen Prototypen entwickelt und angewandt. Jedoch sind sowohl die Entwicklungsschritte als auch die Hauptbestandteile aus der vorgestellten Systemarchitektur (vgl. Kapitel 7.2) sehr ähnlich, sodass hier ein gemeinsames Werkzeug abstrahiert werden kann. Für die Erweiterbarkeit auf neue Diagrammtypen, Zwischenmodelle und Abbildungsregeln wäre die Entwicklung eines Frameworks förderlich.

Abbildungsverzeichnis

Abbildung 1: Vision einer modellbasierten Softwarearchitekturanaylse 3
Abbildung 2: Beispiel für einen JUnit-Test 4
Abbildung 3: Beispiel der Anwendung der Klassendiagramm-Aufleitung 6
Abbildung 4: Beispiel eines Contraint für die eindeutige Bezeichnung eines Features 6
Abbildung 5: Erweiterung durch Constraints 7
Abbildung 6: Übersicht einer Transformation 8
Abbildung 7: Beispiel für ein Komponentendiagramm 15
Abbildung 8: Auszug aus dem Shop-System 24
Abbildung 9: Genutzte Schichtenarchitektur 24
Abbildung 10: Auszug Klassendiagrammdarstellung 25
Abbildung 11: Ansatz zur Herstellung der Vergleichbarkeit 32
Abbildung 12: Abbildungsproblematik bei direktem XML-Vergleich 36
Abbildung 13: Einführung eines Zwischenformats 37
Abbildung 14: Mengenbeschreibung Vergleichsergebnisse 38
Abbildung 15: Ablauf der Softwarearchitekturanalyse 40
Abbildung 16: Einfaches Schichtendiagramm 43
Abbildung 17: Zwischenmodell für Schichtendiagramme 43
Abbildung 18: Zwischenmodell für Klassendiagramme 45
Abbildung 19: Zwischenmodell eines Komponentendiagramms 48
Abbildung 20: Systemarchitektur 50
Abbildung 21: Innerer Aufbau der Diagramm-Abbilderkomponente 51
Abbildung 22: Innerer Aufbau der Ist-Architektur-Abbilderkomponente 53
Abbildung 23: Innerer Aufbau der Vergleicherkomponente 54
Abbildung 24: Innerer Aufbau der Ergebnisersteller-Komponente 55
Abbildung 25: Schnellübersicht Reflections in VisualStudio 2010 60
Abbildung 26: Schnellübersicht Reflections aus VisualStudio 2010 62
Abbildung 27: Werkzeugoberfläche für Schichtendiagrammanalysen 80
Abbildung 28: Erster Ergebnislauf 81
Abbildung 29: Eingabe Schichtendiagramm 81
Abbildung 30: Weiteres Analysergebnis für Schichtendiagramme 82
Abbildung 31: Ergebnis der nicht strikten Schichtendiagrammanalyse 83
Abbildung 32: Übersicht Klassendiagrammeingabe 85
Abbildung 33: Klassendiagammauszug der Detailsicht 86
Abbildung 34: Ergebnis der ersten Klassendiagrammanalyse 87
Abbildung 35: Fehlermeldung einer Verbindung zwischen zwei Klassen 87
Abbildung 36: Klassendiagrammanalysierer Dialog inkl. Optimierungen 89
Abbildung 37: Statische Sicht- Fokus Klassendiagrammvergleich 89
Abbildung 38: DSV-Fokus Klassendiagrammvergleich 90
Abbildung 39: Soll-Komponentendiagramm für die Analyse 93
Abbildung 40: Ergebnis-Komponentendiagramm nach der Analyse 95
Abbildung 41: Fehlermeldung Darstellung in dem Ergebniskomponentendiagramm 96
Abbildung 42: Ergebnis des optimierten Komponentdiagrammanalysewerkzeugs................97
Abbildung 43: Fehlermeldung aus optimierter Komponentenanalyse 98

Tabellenverzeichnis

Tabelle 1: Abbildungsregeln Schichtendiagramm -> C# 44
Tabelle 2: Klassenabbildungsregeln für Klassendiagramme auf C# 45
Tabelle 3: Auszug Abbildungsregeln für Attribute aus Klassendiagramme 46
Tabelle 4: Auszug Abbildungsregeln für Komponentendiagramme auf C#-Reflections.........49

Literaturverzeichnis

Bass Len [et al.] Evaluating the Software Architecture Competence of Organizations [Konferenz] // WISCA. - [s.l.] : IEEE, 2008.

Bass Len, Kazman Rick und Clements Paul C. Software Architecture in Practice [Buch]. - Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc, 2003.

Becker Peter Grundlagen zu Schnittstellen [Online]. - fh.rhein-sieg, 2008. - http://www2.inf.fh-rhein-sieg.de/~pbecke2m/progjava/interface1.pdf.

Becker-Pechau Petra http://subs.emis.de/LNI/Proceedings/Proceedings154/gi-proc-154-295.pdf [Online]. - 3. 10 2010. - 19. 5 2011. - http://subs.emis.de/LNI/Proceedings/Proceedings154/gi-proc-154-295.pdf.

CIL ECMA C# and Common Language Infrastructure Standards [Online]. - Microsoft, 2005. - 24. 5 2011. - http://msdn.microsoft.com/en-us/netframework/aa569283.aspx.

Clark Tony und Tratt Laurence Language Factories [Konferenz] // Proc. OOPSLA. - 2009.

Fähnrich K.-P. Universität Leipzig - Institut für Informatik Betriebliche Informationssysteme [Online] // Vorlesung Softwaretechnik - Entwufsphase: Einführung-. - 2007. - 2011. 5 24. - http://bis.informatik.uni-leipzig.de/de/Lehre/0708/WS/Softwaretechnik/files?get=2007w_swt_v_11.pdf.

Gamma Erich [et al.] Entwurfsmuster [Buch]. - [s.l.] : Addison Wesley Verlag, 2004.

Group Object Management OMG [Online]. - 13. 03 2011. - http://www.omg.org/.

Group Object Management Unified Modeling Language [Online]. - 2011. - uml.org.

hellow2morrow hello2morrow [Online] // hello2morrow. - 2011. - 13. 03 2011. - http://www.hello2morrow.com/products/sotograph.

Hoffmann Dirk W. Software-Qualität [Buch]. - [s.l.] : Springer, 2008.

Hofmeister Christine, Crnkovic Ivica und Reussner Ralf Quality of Software Architectures, Second International Conference on Quality of Software Architectures [Buchabschnitt] // Lecture Notes in Computer Science. - [s.l.] : Dpunkt, 2006. - Bd. 4214.

IComponent Microsoft - IComponentConnector.InitializeComponent-Methode [Online]. - Microsoft, 2011. - 25. 05 2011. - http://msdn.microsoft.com/de-de/library/system.windows.markup.icomponentconnector.initializecomponent%28v=vs.90%29.aspx.

Kecher Christoph UML 2.0. Das umfassende Handbuch [Buch]. - [s.l.] : Galileo Press, 2005.

Medvidovic Nenad [et al.] Modeling software architectures in the Unified Modeling Language [Artikel] // ACM Transaction on Software Engineering and Methodolgy. - [s.l.] : ACM, 2002. - 1 : Bd. 11.

Microsoft Übersicht über das Visio-Objektmodel [Online]. - Microsoft, 2011. - 15. Mai 2011. - http://msdn.microsoft.com/de-de/library/cc160740.aspx.

Microsoft Visual C# Developer Center [Online]. - Microsoft, 2011. - 24. 5 2011. - http://msdn.microsoft.com/en-us/vcsharp/aa336706.

MSDN Modellieren der Anwendung [Online]. - Microsoft, 2011. - 24. 5 2011. - http://msdn.microsoft.com/de-de/library/57b85fsc%28v=VS.100%29.aspx.

Neuhardt Erwin Modelling Behavior by Activity Diagrams and Complete Code Generation [Bericht]. - [s.l.] : sig-mdse, 2008.

Papadopolous George A. Automatic Code Generation: A Practical Approach [Konferenz] // ITI. - Croatia : [s.n.], 2008. - S. S.6.

Paradigm Visual Products [Online]. - Visual Paradigm, 2011. - 24. 5 2011. - http://www.visual-paradigm.com/product/?favor=vpuml.

Petrasch Roland und Meimberg Oliver Model Driven Architecture [Buch]. - [s.l.] : dPunkt, 2006.

Pires Waldemar, Brunet Jo~ao und Ramalho Franklin UML-based Design Test Generation [Konferenz] // SAC 2008. - Brazilien : [s.n.], 2008.

Reussner Ralf und Hasselbring Wilhelm Handbuch der Software-Architektur [Buch]. - [s.l.] : Dpunkt Verlag, 2006.

Shu Chen, Qing Wu Guo und Jing Xiao Metamodel Approach on Model Conformance and Multiview [Journal] // National High-Tech Research and Development. - 2008. - S. ab S. 4.

Spillner Andreas und Linz Tilo Basiswissen Softwaretest [Buch]. - [s.l.] : Dpunkt Verlag, 2005.

Stachowiak Herbert Allgemeine Modelltheorie [Buch]. - [s.l.] : Springer-Verlag, Wien, 1973.

Stahl Thomas, Völter Markus und Efftinge Sven Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management [Buch]. - [s.l.] : Dpunkt Verlag, 2007.

Sutton Andrew und Maletic Jonathan I. Mappings for Accurately Reverse Engineering UML Class Models from C++ [Konferenz] // WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering. - Washington, DC, USA : IEE Computer Science, 2005.

Tigris Open Source Software Egineering Tools [Online]. - Tigris, 2011. - 14. 5 2011. - http://argouml.tigris.org/.

Transforming Models with ATL [Online]. - Universite De Nantes, 2008. - 25. 5 2011. - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.6117.

Vinita [et al.] On reverse engineering an object-oriented code into UML class diagramms incorporating. - 2008.

Walden Kim und Nerson Jean-Marie Seamless object-oriented software architecture: analysis and design of reliable systems [Buch]. - [s.l.] : Prentice-Hall, Inc., 1995.

Züllighoven Heinz [et al.] Das objektorientierte Konstruktionshandbuch [Buch]. - [s.l.] : Dpunkt, 1998.

Züllighoven Heinz, Lilienthal Carola und Bennicke Marcel Software Architecture Analysis and Evaluation [Konferenz] // QoSA. - 2006.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw. §24(4) bzw. §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

__________________________ ____________________________

Ort, Datum Unterschrift