Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Milena Rötting

Konzeption von Codegenerierung aus UML 2 Komponenten- und Kompositionsstrukturdiagrammen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Milena Rötting

Konzeption von Codegenerierung aus UML 2 Komponenten- und Kompositionsstrukturdiagrammen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte 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 15. August 2011

Milena Rötting

Thema der Bachelorarbeit

Konzeption von Codegenerierung aus UML 2 Komponenten- und Kompositionsstrukturdiagrammen

Stichworte

UML 2, Codegenerierung, Modellgetriebene Softwareentwicklung, Komponentediagramm, Kompositionsstrukturdiagramm, Visual C# 2010

Kurzzusammenfassung

Codegenerierung erhöht die Softwarequalität und erleichtert die Arbeit von Softwareentwicklern. Darum wird in dieser Arbeit die Generierung von Code aus UML 2 Komponenten- und Kompositionsstrukturdiagrammen in Bezug auf Komponentenmodellierung konzipiert und analysiert. Es wird der Prototyp eines Codegenerators unter den Gesichtspunkten der Erweiterbarkeit und Modularität entworfen und entwickelt.

Milena Rötting

Title of the paper

Code generation of UML 2 Component- and Compositestructurediagrams

Keywords

UML 2, code generation, model driven software development, component diagram, composite structure diagram, Visual C# 2010

Abstract

Code generation rises the quality of software and supports software developers in their work. For that reason the generation of code from UML 2 component- and composite structure diagrams is designed and analyzed focusing on the design of components. A prototype of a code generator is developed in terms of modularity and expandability.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einführung

1.1. Motivation

Codegenerierung ist eine Entwicklungsmethode der Softwareentwicklung bei der Quellcode, anstatt durch manuelle Implementierung durch einen Entwickler, automatisch aus einem Modell generiert wird. Dadurch sind Modell und Code konform zueinander und der Code aller vom Codegenerator übersetzten Modelle hat einen einheitlichen Programmierstil. Wie Stahl u. a. (2007) schreiben bringt dies vor allem in größeren, gewachsenen Softwareprojekten bei der Nutzung eines Generators, der ausführbaren Code erzeugt, einen qualitativen und mittelfristig auch zeitlichen Vorteil. Bei Anpassungen und Erweiterungen über einen längeren Zeitraum ist der Code bei konsequenter Nutzung des Generators einheitlich aufgebaut und Entwickler müssen sich nicht an verschiedene Eigenheiten von anderen Entwicklern anpassen. Im Idealfall wird zusammen mit dem Code auch ein Teil seiner Dokumentation generiert (Stahl u. a. (2007)).

Code aus Modellen generieren zu lassen bietet des Weiteren den Vorteil der Unabhängigkeit und Wiederverwendbarkeit. Codegeneratoren unterstützten unter Umständen mehrere Programmiersprachen. Gerade bei Systemen, die für verschiedene heterogene Plattformen konzipiert werden und unterschiedliche Implementierungssprachen benötigen, bietet sich hier die Generierung des Codes an, da das Modell des Systems nur einmal entworfen werden muss und für die Implementierung nicht so viel Zeit benötigt wird.

Entwickler machen darüber hinaus Fehler bei der Übersetzung des Modells in Code. Bei der Nutzung eines Generators hingegen kann man, vorausgesetzt, er ist korrekt konzipiert und entwickelt, davon ausgehen, dass keine Fehler bei dieser Übersetzung passieren.

Für die Entwickler kann die Nutzung eines Codegenerators zudem subjektive Vorteile bringen. Gerade die Erstellung von grundlegenden Komponenten- und Klassenrümpfen ist eine Arbeit, die zwar häufig erledigt werden muss und unumgänglich ist, die aber nicht sonderlich anspruchsvoll ist. Diese Arbeit kann als lästig und eintönig empfunden werden. Ein Codegenerator kann einem Entwickler diese Arbeit abnehmen und diesem mehr Zeit für anspruchsvollere Aufgaben geben. Diese Arbeiten können dann idealerweise intensiver bearbeitet werden und führen zu qualitativ besseren Ergebnissen.

Um die Rümpfe einer Anwendung zu beschreiben, eignen sich Komponentendiagramme der UML. Komponenten bieten den Vorteil der Wiederverwendbarkeit und unterstützen damit die Prinzipien der Modularität, Abstraktion und Kapselung. Komponenten können durch diese Prinzipien einfach ausgetauscht und angepasst werden, sofern sie ihre äußere Schnittstelle einhalten (Taylor u. a. (2010)).

Die verschiedenen Diagramme der UML beschreiben verschiedene Sichten auf ein System. Um Entwicklern die oben erwähnte wiederkehrende Tätigkeit des Implementierens von Komponenten- und Klassenrümpfen abzunehmen, benötigt man Diagramme, welche Komponenten, ihre Beziehungen zwischen einander und ihre internen Strukturen, Klassen und Interfaces, modellieren. All diese Aspekte werden entweder durch Komponenten- oder durch Kompositionsstrukturdiagramme abgedeckt. Darum soll nun untersucht werden, ob beide Diagrammarten zusammen dafür geeignet sind, Komponenten- und Klassen- sowie Interfacerümpfe zu generieren und modellierte Beziehungen zwischen diesen Elementen im Code herzustellen.

1.2. Ziel der Arbeit

In dieser Arbeit soll die direkte Transformation von UML 2 Komponenten- und Kompositionsstrukturdiagrammen zu C#-Code konzipiert werden. Hierfür wird zunächst festgelegt, welche Diagrammelemente in die Codegenerierung eingehen sollen. Nachfolgend wird dann untersucht, welche Codeelemente sich aus diesen Diagrammen generieren lassen.

Anschließend soll der Prototyp eines Codegenerators entwickelt werden, der die UML-Modelle im XML-Format einliest, den entsprechenden Code generiert und an einem ausgewählten Ort im Dateisystem speichert. Die eingegebenen Modelle müssen dabei einigen Kriterien genügen damit der Generator ausreichend Informationen zur Codeerzeugung erhält. Die UML-Modelle sollen mit dem Enterprise Architect 8.0 (SparxSystems (2011)) erstellt und in das XMI-Format 2.1 (OMG (2007)) exportiert werden.
Es ist darauf zu achten, dass der generierte Code für Menschen lesbar ist, da der Code manuell ergänzt werden muss, damit er lauffähig wird. Die Architektur des Codegenerators soll modular aufgebaut sein, um Änderbarkeit und Erweiterbarkeit zu garantieren.

1.3. Aufbau der Arbeit

Die Einführung in das Thema und das Ziel dieser Arbeit wurden bereits in diesem ersten Kapitel erbracht. Diese Arbeit beschäftigt sich im nachfolgenden Kapitel 2 mit der Einführung eines Fallbeispiels und den Grundlagen verschiedener Technologien zur Codegenerierung.

Auch eine Einführung in die UML sowie in die in dieser Arbeit betrachteten Diagrammelemente finden sich im zweiten Kapitel. Das dritte Kapitel widmet sich der Analyse der Diagramme und beschreibt die Transformation vom Modell zum Code anhand von C#. Im Anschluss wird in Kapitel 4 der zu entwickelnde Codegenerator entworfen. Es werden Anwendungsfälle beschrieben und Anforderungen aufgestellt anhand derer im Anschluss die Architektur des Codegenerators erstellt wird. Kapitel 5 beschreibt zum einen, wie weit die Realisierung des Codegenerators voran geschritten ist, zum anderen wird das Testkonzept desselben beschrieben. Das letzte Kapitel 6 schließt die Arbeit mit einem Fazit und möglichen Erweiterungen ab.

2. Grundlagen

Nachfolgend werden die nötigen Grundlagen für den weiteren Verlauf dieser Arbeit erläutert. Zunächst wird ein Fallbeispiel entworfen anhand dessen der Codegenerator entwickelt werden soll. Es werden eine Einführung in die UML gegeben und die verwendeten Diagrammelemente von Komponenten- und Kompositionsstrukturdiagrammen näher beschrieben. Das Kapitel schließt mit der Einführung einiger Codegenerierungsansätze, die zur Auswahl stehen und aus welchen später in Abschnitt 4.4 der Passendste ausgewählt werden soll.

2.1. Fallbeispiel

Im Folgenden wird das Fallbeispiel eingeführt, welches im weiteren Verlauf der Arbeit dazu genutzt wird, die Grundlagen und die Analyse zu erklären und den zu entwickelnden Codegenerator zu beschreiben und zu testen.

Als Beispiel wird ein einfaches Transportverwaltungssystem entworfen und modelliert. Das System nimmt einen Transportauftrag an, plant die Strecke anhand des angegebenen Start- und Zielortes und wählt ein zur Ladung passendes Transportmittel aus.

Das System besteht aus den nachfolgenden fachlichen Entitäten:

- Transportauftrag
- Transportmittel
- Transportstrecke
- Lokation
- Transportbeziehung

Die komplette Liste der fachlichen Anforderungen ist in Anhang B zu finden. An dieser Stelle werden nur einige Anforderungen erläutert, welche die zentralen Abhängigkeiten zwischen den Entitäten beschreiben.

Das System verwaltet Transportmittel und Transportbeziehungen, die als Grundlage für die Transportplanung verwendet werden. Die zentrale Entität zur Transportplanung ist der Transportauftrag (Abbildung A.1). In ihm werden alle Informationen gespeichert, die für die Planung relevant sind.

Eine Transportbeziehung besteht aus zwei Lokationen, wie Anforderung F-4.1 beschreibt:

F-4.1 Eine Transportbeziehung stellt eine gerichtete Verbindung von einer bestimmten Länge zwischen zwei Lokationen in dem Transportsystem dar und kann von einem Transportmittel befahren werden.

Wenn einem Transportauftrag der Start- und der Zielort des Transports bekannt ist, so wird eine Transportstrecke wie in F-5.7 beschrieben, berechnet.

F-5.7 Die Strecke einer Transportstrecke ist eine geordnete Liste von Referenzen auf Transportbeziehungen des Transportsystems, welche den Startort der Transportstrecke mit ihrem Zielort verbinden. Diese Liste umfasst in dem Fall, dass es eine direkte Transportbeziehung zwischen Start- und Ziellokation gibt, genau ein Element. In allen anderen Fällen werden Transportbeziehungen gesucht, die die Startlokation mit einer Zwischenlokation verbinden, welche wiederum mit der Ziellokation oder weiteren Zwischenlokationen verbunden ist.

Ein Transportmittel hat einen Fahrzeugtyp und einen Ladungstyp. Der Ladungstyp kann laut Anforderung F-1.7 (siehe Anhang B) „Person“, „TEU“ oder „FEU“ sein. „TEU“ und „FEU“ sind Bezeichnungen für nach Iso (1999) standardisierte Container von 20 („TEU“, Twenty-foot Equivalent Unit) bzw. 40-Fuß („FEU“, Forty-foot Equvalent Unit) Größe. Das Transportunternehmen kann also Container und Personen transportieren. Je nach angegebenem Ladungstyp ist die Menge in der Anzahl zu transportierender Personen oder in der Anzahl zu transportierender „TEUs“ bzw. „FEUs“ anzugeben. Aus den Anforderungen F-2.7 und F-2.8 ergibt sich die Anforderung F-6.12, welche die Zuordnung des Transportmittels zu einem Transportauftrag erläutert.

F-2.7 Ein Transportmittel vom Fahrzeugtyp „Bus“ kann nur Ladung vom Typ „Person“ transportieren.

F-2.8 Ein Transportmittel vom Fahrzeugtyp „LKW“ kann entweder Ladung vom Typ „TEU“ oder Ladung vom Typ „FEU“, nicht jedoch von beiden, transportieren.

F-6.12 Das System erstellt die Referenz auf ein Transportmittel, indem ein Transportmittel gesucht wird, welches den im Transportauftrag angegebenen Ladungstyp transportieren kann und Kapazitäten entsprechend der angegebenen Ladungsanzahl besitzt.

Zur Berechnung der Transportkosten und der Länge der Transportstrecke greifen nun die Referenzen von einem Transportmittel und einer Transportstrecke bei einem Transportauftrag ineinander. Die Länge einer Transportstrecke wird zunächst folgendermaßen berechnet:

F-6.16 Die Länge eines Transportauftrages wird vom System aus der Summe der Streckenlängen der Transportbeziehungen, die in der zugehörigen Transportstrecke enthalten sind, berechnet.

Das Ergebnis dieser Berechnung wird in Anforderung F-6.18 verwendet um die Transportkosten zu berechnen:

F-6.18 Die Kosten eines Transportauftrages werden vom System aus dem Produkt der Länge dieses Transportauftrages und den Kosten pro Kilometer dieses Transportmittels berechnet.

Da dieses Beispiel nur zu Demonstrationszwecken dient, ist der Schwerpunkt der Modellierung auf die Vollständigkeit des Kompositionsstrukturdiagrammes gesetzt. Der Fokus des Beispiels liegt nicht auf der realitätsnahen Abbildung des Workflows zur Transportplanung. Dies spiegelt sich beispielsweise in Ausgrenzung F-7.1 wieder:

F-7.1 Es gibt keine zeitliche Kapazitätsbeschränkung von Transportmitteln.

Dieser fachlich Punkt bleibt in der Modellierung offen um die Komplexität des Beispielsystems gering zu halten. Dennoch ist das Beispiel mit den modellierten Komponenten ausreichend komplex gestaltet um den Codegenerator experimentell testen zu können.

Aus den Anforderungen ergibt sich zunächst das fachliche Datenmodell. Aus dem fachlichen Datenmodell lassen sich wiederum Komponenten bilden. Das fachliche Datenmodell und der Komponentenschnitt sind in Anhang A zu finden.

Es werden drei Komponenten gebildet. Durch diese kann gezeigt werden, wie UML Komponenten- und Kompositionsstrukturdiagramme verwendet werden können, um die internen Strukturen einer Komponente und ihre Interaktion mit ihrer Umgebung zu modellieren.

Die Komponenten sind nach dem Prinzip der Quasar Referenzarchitektur aufgebaut (vgl. Siedersleben (2003a) und Siedersleben (2003b)). Da das Beispiel überschaubar sein soll, ist dieses System ohne technische Laufzeitumgebung und graphische Benutzerschnittstelle konzipiert. Es wird also nur der Anwendungskern modelliert.

Ein Anwendungskern nach Quasar besteht aus Komponenten die wiederum folgende Typen von Klassen enthalten können:

- Entitäten

- Datentypen
- Verwalter
- Anwendungsfälle

Entitäten stellen die Repräsentation der fachlichen Daten des Systems dar. Sie sind über eine Id eindeutig referenzierbar und besitzen einen Lebenszyklus. Man kann sie also anlegen, speichern, ändern und löschen (vgl. Siedersleben (2004)).
Datentypen stehen im Gegensatz dazu. Sie repräsentieren Daten, deren Speicherung als eigenständiges Objekt unsinnig wäre. Beispielhaft ist hier ein Datum. Der Wert eines Datums ist relevant, aber eine Speicherung von mehrere Daten mit dem gleichen Wert aber unterschiedlicher Id wäre nicht nötig. Darum besitzen Datentypen keine eindeutige Id. Sie werden durch Feldgleichheit anstelle von Referenzgleichheit verglichen (vgl. Siedersleben (2004)).
Verwalter implementieren Operationen zum Verwalten von Entitäten. Im vorliegenden Beispiel benutzen sie die Schnittstelle der Persistenz, welche die konkrete Persistenztechnik vom Anwendungskern kapselt, und realisieren das Anlegen, Speichern, Löschen und Bearbeiten von Entitäten. Des Weiteren dienen sie dazu, Entitäten nach bestimmten Kriterien zu suchen.
Anwendungsfälle spiegeln die Anwendungsfälle des Systems wieder und bilden somit die fachlichen Kernfunktionen. Sie greifen auf die von der Komponente angeforderten Schnittstellen zu und implementieren die Schnittstellen, welche die Komponente anbietet.
Zusätzlich zu den vier oben genannten Klassentypen gibt es in diesem Beispiel einen Konfigurator pro Komponente. Dieser ist dafür zuständig, die Klassen bei der Initialisierung der Komponente nach dem Muster der Dependency Injection richtig zusammen zu setzen, so dass die Komponente die Anwendungsfälle kennt und die Anwendungsfälle wiederum mit dem Verwalter kommunizieren können. Durch diesen Konfigurator wird eine weitere Abstraktionsschicht eingebaut, die die Abhängigkeiten innerhalb der Komponente nochmals strikt voneinander trennt und an einem Ort bündelt. Dadurch sind eventuelle komponenteninterne Umstrukturierungen einfacher zu realisieren und produzieren weniger Seiteneffekte.

Im Folgenden wird jeweils die fachliche Sicht auf die Komponenten und die Anwendungsfälle des Systems beschrieben, um dem Leser ein Gefühl für das Transportsystem zu vermitteln.

Transportkomponente

Die Transportkomponente enthält die fachliche Entität *Transportauftrag*. Ein Transportauftrag bündelt alle Informationen, die für die erfolgreiche Durchführung eines Transports notwendig sind. Er wird, wie auch alle anderen fachlichen Entitäten, sowohl über einen Datentypen als auch über eine technische Id eindeutig referenziert. Dies ist der Fall, da der Datentyp die

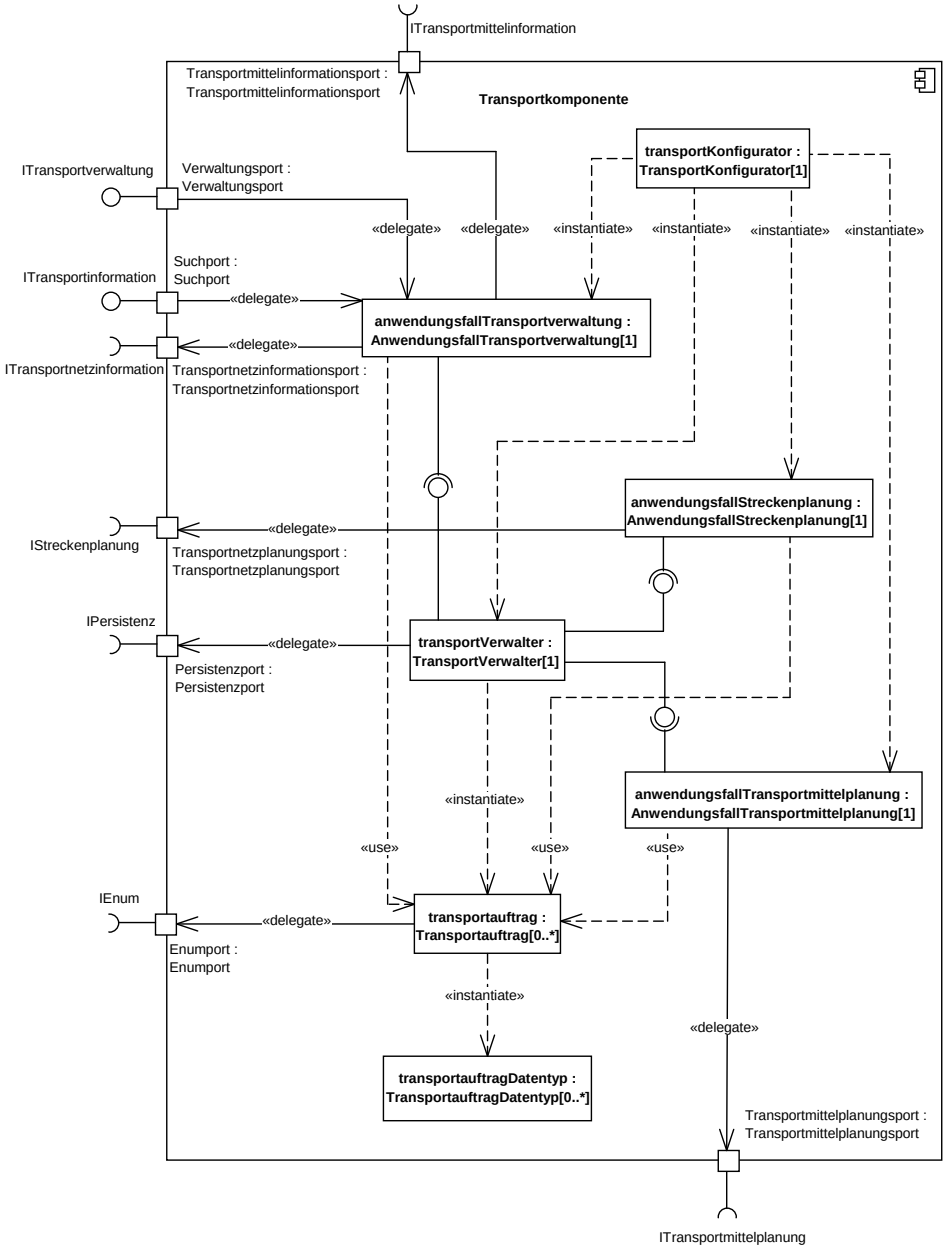


Abbildung 2.1.: Kompositionsstrukturdiagramm der Transportkomponente

fachliche Id zur Benutzerschnittstelle und zur Repräsentation im Anwendungskern darstellt. Die technische Id wird für die Repräsentation in der Persistenzschicht benötigt und nur dort verwendet.
Zusätzlich hält ein Transportauftrag Informationen über Start- und Zielort des Transports, die gesamte Transportstrecke und über ein zugeordnetes Transportmittel. Diese Informationen werden jeweils in Form einer fachlichen Id gespeichert, die die Referenz auf die jeweilige Entität darstellt. Des Weiteren werden im Transportauftrag die Menge an zu transportierender Ladung, der Ladungstyp, der Name des Kunden, das Transportdatum und die Gesamtkosten des Transports festgehalten.
Um die Transportstrecke und ein passendes Transportmittel zu finden, hat die Komponente Abhängigkeiten zur Transportmittel- und zur Transportnetzkomponente. Der Ladungstyp wird in der nicht dargestellten Datentypenkomponente definiert, da die Transportmittelkomponente diesen ebenfalls verwendet und es darum sinnvoll ist, ihn in einer eigenen Komponente zu kapseln. Diese Abhängigkeit wird durch einen Zugriff des Parts *Transportauftrag* auf die Schnittstelle *IEnum* mittels des *Enumports* dargestellt.

AnwendungsfallStreckenplanung Dieser Anwendungsfall stellt die Strecke eines Transports zusammen. Dazu übergibt der Anwendungsfall die fachliche Id der Start- und Ziellokation aus dem Transportauftrag mithilfe des *Transportnetzplanungsports* an die Transportnetzkomponente. Im Erfolgsfall gibt diese die fachliche Id einer Transportstrecke zurück, die die Startlokation durch im System vorhandene Transportbeziehungen mit der Ziellokation verbindet. Außerdem wird in diesem Anwendungsfall die Länge der gesamten Transportstrecke in Kilometern berechnet.

AnwendungsfallTransportmittelplanung Dieser Anwendungsfall sucht ein zum Transportauftrag passendes Transportmittel. Der Anwendungsfall übergibt der Transportmittelkomponente Informationen über den Ladungstyp und die Ladungsmenge aus dem Transportauftrag durch den *Transportmittelplanungsport*. Im Erfolgsfall erhält der Anwendungsfall die fachliche Id eines passenden Transportmittels zurück. Zusätzlich berechnet dieser Anwendungsfall die Gesamtkosten eines Transports. Um die Komplexität des Beispiels überschaubar zu halten wurde an dieser Stelle darauf verzichtet, eine Verfügbarkeitsprüfung des Transportmittels zu modellieren. Es ist somit möglich, dass zeitlich gesehen unbegrenzt viele Transportmittel zur Verfügung stehen.

AnwendungsfallTransportverwaltung Mithilfe dieses Anwendungsfalles können Transportaufträge angelegt, gespeichert, bearbeitet und gelöscht werden. Die Eingabe der Grunddaten wie Kundenname, Transportdatum, Start- und Ziellokation sowie Ladungstyp und -menge geschieht über die Schnittstelle *ITransportverwaltung* des *Verwaltungsports*. Alle weiteren Felder des Transportauftrages werden berechnet indem dieser Anwendungsfall die Anwendungsfälle *AnwendungsfallStreckenplanung* und *AnwendungsfallTransportmittelplanung* aufruft. Neben der Bereitstellung dieser Funktionen für die Schnittstelle *ITransportverwaltung* implementiert der Anwendungsfall

die Schnittstelle *ITransportinformation* des *Suchports* durch die es möglich sein soll, Transportaufträge nach bestimmten Kriterien zu suchen. Um Informationen zu den Transportstrecken und Transportmitteln des jeweiligen Transportauftrages zu erhalten, muss dieser Anwendungsfall auf die Transportmittel- und Transportnetzkomponente zugreifen. Dazu verwendet er die Schnittstellen *ITransportnetzinformation* des *Transportnetzinformationsports* und *ITransportmittelinformation* des *Transportmittelinformationsports*.

Transportmittelkomponente

Die Transportmittelkomponente enthält die fachliche Entität *Transportmittel*. Ein Transportmittel wird mithilfe des Transportmitteldatentyps oder einer technischen Id eindeutig referenziert. Des Weiteren verwendet die Entität den *FahrzeugtypDatentyp* und sie kann über den *Enumport* auf den Ladungstyp zugreifen. Der *FahrzeugtypDatentyp* kann einen der beiden Werte „Bus“ und „LKW“ annehmen. Dieser *FahrzeugtypDatentyp* schränkt den möglichen Ladungstyp des Transportmittels ein. Die Menge der zu transportierenden Güter kann dann als Anzahl von Einheiten des Ladungstyps gelesen werden. Weiterhin enthält ein Transportmittel Informationen zu seinen Kosten, angegeben in € pro Kilometer.

AnwendungsfallTransportZuordnung Dieser Anwendungsfall erhält Informationen zu Ladungstyp und -menge eines Transportauftrags über die Schnittstelle *ITransportmittelplanung* des *Transportports* und gibt die fachliche Id eines passenden Transportmittels zurück. Dieses Transportmittel wird nach der Anzahl von Gütern des übergebenen Ladungstyps, die es transportieren kann, ausgewählt.

AnwendungsfallTransportmittelverwaltung Dieser Anwendungsfall implementiert die Schnittstellen *ITransportmittelverwaltung* des *Verwaltungsports* und *ITransportmittelinformation* des *Suchports*. Über die Schnittstelle *ITransportmittelverwaltung* werden die CRUD-Methoden[1] für Transportmittel bereitgestellt. *ITransportmittelinformation* bietet Suchoperationen nach verschiedenen Kriterien für Transportmittel an. Zum Beispiel könnte es für die Darstellung in einer graphischen Benutzeroberfläche nötig sein, Transportmittel anhand ihrer fachlichen Id oder ihres *FahrzeugtypDatentyps* zu suchen.

Transportnetzkomponente

Die Transportnetzkomponente modelliert das Transportnetz, welches von Transportmitteln des Unternehmens befahren werden kann. Die fachlichen Entitäten sind *Lokation*, *Trans-*

[1] CRUD = Create, Read, Update, Delete

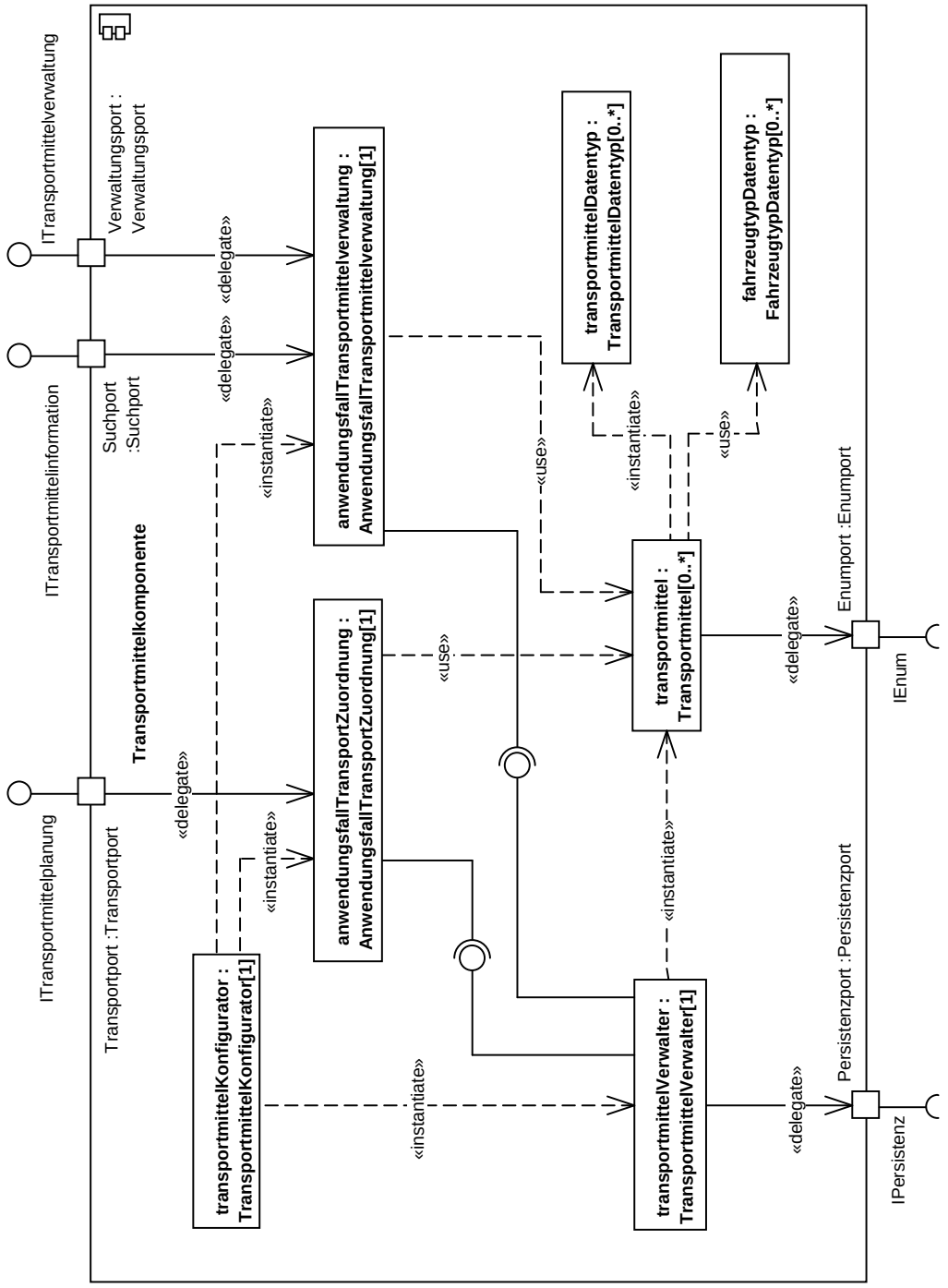


Abbildung 2.2.: Kompositionsstrukturdiagramm der Transportmittelkomponente

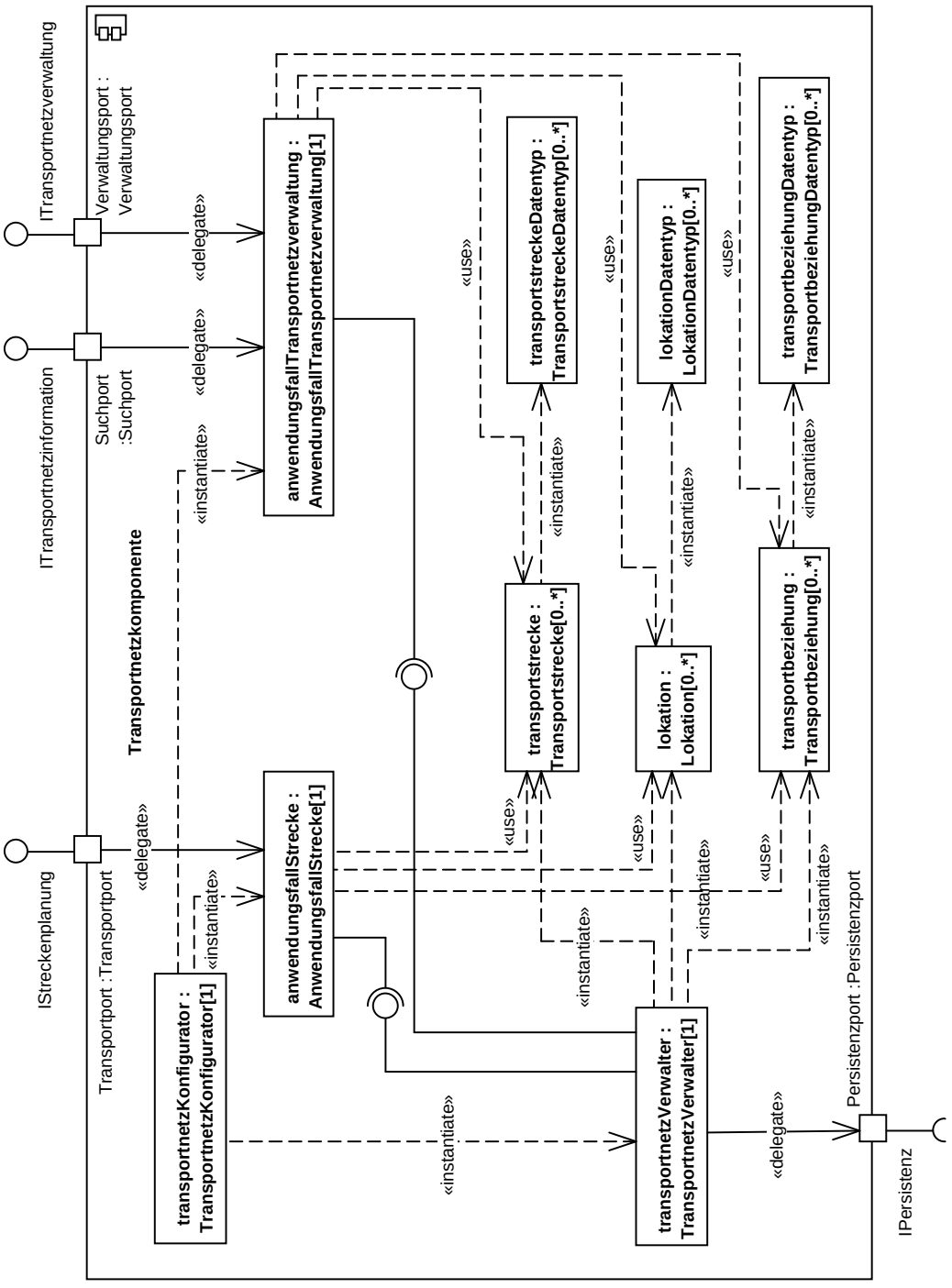


Abbildung 2.3.: Kompositionsstrukturdiagramm der Transportnetzkomponente

portbeziehung und *Transportstrecke*.
Eine Lokation besteht aus einem Namen, einer Id vom Typ *LokationDatentyp* sowie einer technischen Id. Die Lokation ist ebenfalls bewusst simpel gehalten um das System nicht zu komplex werden zu lassen. Darum werden an dieser Stelle keine Adresse oder Geodaten zum genauen Spezifizieren der Position eingeführt. Es ist angedacht, dass der Ort einer Lokation in diesem Rahmen durch ihren Namen beschrieben wird.
Eine Transportbeziehung besteht aus zwei Lokationen, einer Id vom Typ *TransportbeziehungDatentyp*, einer technischen Id und der Länge der Strecke zwischen den beiden Lokationen in Kilometer. Eine der beiden Lokationen stellt den Startort der Transportbeziehung dar, die andere den Zielort. Daraus ergibt sich, dass eine Transportbeziehung gerichtet ist. Für den Hin- und Rückweg zwischen zwei Lokationen muss es also zwei Transportbeziehungen geben.
Zu jedem Transportauftrag wird eine Transportstrecke erstellt. Sie besteht aus einer Id vom Typ *TransportstreckeDatentyp*, einer technischen Id sowie einer Liste von Transportbeziehungen und der Start- und Ziellokation dieser Strecke.

AnwendungsfallStrecke Dieser Anwendungsfall implementiert die Schnittstelle *IStreckenplanung* des *Transportports*. Über diese Schnittstelle werden Referenzen auf eine Start- und eine Ziellokation einer Transportstrecke angegeben. Der Anwendungsfall sucht nun Transportbeziehungen heraus, die diese referenzierten Lokationen miteinander verbinden. Dabei gibt es im Idealfall bereits eine direkte Transportbeziehung zwischen den Lokationen. Ansonsten müssen solche Transportbeziehungen gefunden werden, die die Start- und Ziellokation transitiv, also über eine oder mehrere Zwischenlokationen, miteinander verbinden. Die geordnete Menge der gefundenen Transportbeziehungen stellt eine Transportstrecke dar, deren fachliche Id als Ergebnis an die Schnittstelle zurück gegeben wird.

AnwendungsfallTransportnetzverwaltung Dieser Anwendungsfall implementiert die Schnittstellen der Ports *Verwaltungsport* und *Suchport*. Für die Schnittstelle *ITransportnetzverwaltung* des *Verwaltungsports* werden die CRUD-Operationen für Lokationen und Transportbeziehungen zur Verfügung gestellt. Da Transportstrecken berechnet werden, existieren für diese Entität keine CRUD-Operationen. Über die Schnittstelle *ITransportnetzinformation* des *Suchports* können Lokationen, Transportbeziehungen und Transportstrecken nach verschiedenen Kriterien gesucht werden.

2.2. UML

Die UML (Unified Modeling Language) ist eine von der Object Management Group (OMG) standardisierte Modellierungssprache mit 14 unterschiedlichen Arten von Diagrammen. Die nachfolgende Beschreibung der Notationselemente in dieser Arbeit bezieht sich auf die Version 2.3, welche im Mai 2010 erschienen ist (OMG (2010a) und OMG (2010b)).

Sowohl in der Spezifikation OMG (2010b) als auch in Hitz u. a. (2005) wird vorgeschlagen, verschiedene Diagrammarten miteinander zu kombinieren. Das soll in dieser Arbeit geschehen. Ziel ist es, dass sowohl die Interna einer Komponente in Form eines Kompositionsstrukturdiagrammes als auch die Abhängigkeiten einer Komponente zu anderen Komponenten des Systems mittels Komponentendiagramm dargestellt werden können. Dazu ist die Kombination von Komponenten- und Kompositionsstrukturdiagrammen gut geeignet. Sie gehören beide zur Gruppe der Strukturdiagramme, welche die statischen Elemente eines Systems beschreiben (vgl. Kecher (2009)).

Komponentendiagramme dienen dazu, ein System in Komponenten zu strukturieren und Abhängigkeiten zwischen diesen Komponenten darzustellen. Die Komponenten eines Diagramms haben eine Außenansicht und eine Innenansicht. Die Außenansicht zeigt die Schnittstellen einer Komponente, die nach außen sichtbar sind sowie die Zusammenhänge von verschiedenen Komponenten zwischen einander. Durch die Außenansicht können die öffentlichen Operationen und das öffentliche Verhalten der Komponente beschrieben werden. Die Innenansicht beschreibt, wie die Komponente intern aufgebaut ist. Sie zeigt die zugehörigen Klassen und Abhängigkeiten zwischen diesen. Hierzu bietet sich zunächst ein Klassendiagramm an. Mit diesem ist es aber nicht möglich, zu zeigen, welcher Bestandteil einer Komponente eine bestimmte Funktionalität der Komponente realisiert. Dies ist möglich, wenn die interne Sicht auf eine Komponente durch ein Kompositionsstrukturdiagramm beschrieben wird.

Kompositionsstrukturdiagramme modellieren die interne Struktur eines Klassifizierers und dessen Abhängigkeiten zu seiner Umgebung. Ein Klassifizierer kann eine Klasse sein, aber auch eine Komponente. Mit Hilfe des Kompositionsstrukturdiagrammes kann die Verbindung zwischen der bereitgestellten bzw. benötigten Funktionalität dieses Klassifizierers und dessen internen Elementen, die zur Implementierung bzw. Verwendung der Funktionalität verwendet werden, hergestellt werden. Die internen Strukturen eines Klassifizierers bestehen wiederum aus Klassifizierern und deren Abhängigkeiten untereinander.
Im Gegensatz zum Klassendiagramm beschreibt ein Kompositionsstrukturdiagramm diese Abhängigkeiten aber nur im Kontext des umgebenen Klassifizierer. Dadurch wird eine bestimmte Sicht auf das Klassendiagramm beschrieben (Hitz u. a. (2005)) und es können so kontextsensitive Einschränkungen dargestellt werden, die nicht durch ein Klassendiagramm modelliert werden können. So besitzt z. B. die Klasse *Transportbeziehung* des in

Abschnitt 2.1 beschriebenen Fallbeispiels zwei Attribute vom Typ *Lokation*, einen Start- und einen Zielort. Obwohl Start- und Zielort vom selben Typ sind, kann man den Unterschied zwischen ihnen im Kontext *Transportbeziehung* deutlich erkennen. Die Lokation, die die Rolle des Startortes einnimmt, wird beispielsweise verwendet um Anfragen nach dem Startort einer Transportbeziehung zu beantworten. Die Lokation, welche die Rolle des Zielortes einnimmt, würde in diesem Moment nicht verwendet werden, dafür aber analog bei einem Aufruf, welcher nach dem Zielort fragt. Durch ein Klassendiagramm wäre es nur möglich gewesen, auszudrücken, dass eine Transportbeziehung aus zwei Lokationen besteht, nicht aber, dass eine als Startort und die andere als Zielort arbeitet und sie für unterschiedliche Funktionen von Transportbeziehungen zuständig sind. Ein ausführlicheres Beispiel zu diesem Zusammenhang kann in Oestereich und Bremer (2009) nachgelesen werden.

Im Folgenden werden die in dieser Arbeit verwendeten Elemente von Komponenten- und Kompositionsstrukturdiagrammen beschrieben. Eine genaue Analyse, wie die einzelnen Elemente zur Codegenerierung verwendet werden können, folgt in Kapitel 3.

Komponente

Komponenten sind Softwareeinheiten, die eine in sich möglichst abgeschlossene Einheit bilden und deren Verhalten durch ihre angebotenen und benötigten Schnittstellen definiert wird. Die Schnittstellen kapseln das Verhalten einer Komponente und ermöglichen so ihre Wiederverwendung oder ihren Austausch gegen eine andere Komponente, die mindestens die gleichen Schnittstellen bereitstellt. Die Kapselung des Verhaltens einer Komponente in Schnittstellen ermöglicht es zudem, die einzelnen Komponenten unabhängig voneinander zu entwickeln und am Ende zum Gesamtsystem zusammenzusetzen. Eine noch stärkere Kapselung ist durch Ports möglich. Dadurch wird einen weitere Ebene zwischen der Komponente und ihrer Umgebung eingezogen und sie ist vollständig abgeschirmt. Komponenten können ineinander geschachtelt sein. Die inneren Komponenten realisieren in dem Fall einen Teil oder die gesamte Funktionalität der umgebenden Komponente. Wird die Innenansicht einer Komponente modelliert, so wird sie grafisch als Rechteck dargestellt mit einem Komponentensymbol in der rechten oberen Ecke und einem Namen am oberen Rand des Rechtecks (siehe Abbildung 2.1). Wenn die Innenansicht, wie in Abbildung A.2 dargestellt, nicht modelliert wird, so kann der Name in die Mitte des Rechtecks gesetzt werden.

Part

Parts sind Objekte eines Typs, die in einem Klassifizierer, in dieser Arbeit auch in einer Komponente, enthalten sein können. Sie dienen dazu, Informationen zu transportieren, die aus Assoziationen im Klassendiagramm nicht deutlich werden. Durch die Verwendung von Parts

können Multiplizitäten stärker eingeschränkt werden, als dies im Klassendiagramm der Fall ist. Somit können Informationen für einzelne Instanzen des umgebenen Klassifizierers transportiert werden, die nicht aus einem entsprechenden Klassendiagramm hervorgehen. Diese Kontextsensitivität wurde bereits in der Einleitung dieses Abschnitts im Absatz über Kompositionsstrukturdiagramme erläutert. Weitere Beispiele hierzu finden sich in Kecher (2009) (S. 126ff.) und OMG (2010b) (S. 190f.). Auf Grund der Kompositionsbeziehung zwischen Part und umgebenen Klassifizierer, wird ein Part gelöscht, sobald der umgebene Klassifizierer gelöscht wird. Hingegen wird die Instanziierung eines Parts nicht an die Instanziierung des Klassifizierers gekoppelt. Ein Part darf nicht vor der Instanziierung des umgebenen Klassifizierers erzeugt werden, sehr wohl aber danach. Wenn der Typ eines Parts im Klassendiagramm in Komposition mit dem den Part umgebenen Klassifizierer steht, so wird der Part durch ein Rechteck dargestellt (vgl. *transportauftrag* in Abbildung 2.1). Stehen der Typ des Parts und der Klassifizierer in einer Assoziationsbeziehung, wird der Part durch ein Rechteck mit gestrichelter Linie dargestellt. Unabhängig von der Linie wird bei beiden Darstellungsformen der Name eines Parts mit einem Doppelpunkt vom Typ des Parts abgetrennt. Hinter dem Typ steht die Multiplizität in eckigen Klammern. Diese darf nie größer sein als die Multiplizität des Typs.

Klassen, Datentypen und Enumerationen

Klassen, Datentypen und Enumerationen sind in dem Zusammenhang der Komponenten- und Kompositionsstrukturdiagramme nur als Typen von Parts und Ports wichtig. Sie werden nicht als graphische Elemente in den Diagramme dargestellt, sollten aber in den zugehörigen Modellen der Diagramme auftauchen, um den Typ von Parts und Ports zu spezifizieren. Datentypen ähneln Klassen insofern, als dass sie wie Klassen Attribute und Opperationen enthalten können. Der Unterschied zu Klassen liegt aber darin, dass seine Instanzen nicht durch Referenzgleichheit sondern durch Wertgleichheit verglichen werden. So sind beispielsweise zwei Instanzen vom Datentyp *Uhrzeit* mit der Stundenanzahl „4“ und der Minutenanzahl „37“ gleich. Dagegen wären zwei Objekte der Klasse *Uhrzeit* ungleich, weil die Objekte unterschiedliche Ids besitzen würden.
Enumerationen stellen laut der Spezifikation der UML (OMG (2010b)) spezielle Datentypen dar. Enumerationen sind Aufzählungen, die nur einen bestimmten Wertebereich haben, welcher schon im Diagramm festgelegt sein kann.

Port

Ein Port gehört zu einem Element und stellt einen Interaktionspunkt zwischen dem Inneren dieses Elements und seiner Umgebung dar. Ein Element kann eine Klasse, eine Komponente oder ein Part sein. Der Port kapselt das jeweilige Element von seiner Umgebung und

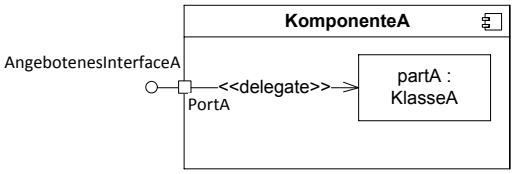
unterstützt hierdurch die Wiederverwendbarkeit des Elements. Ein Port kann Schnittstellen anbieten oder fordern. Diese spezifizieren die gesamten Interaktionsmöglichkeiten, die durch diesen Port möglich sind. Ebenso können sie den Typ des Ports spezifizieren (vgl. Bruel und Ober (2006)).

Ein Port leitet Aufrufe, die an angebotenen Schnittstellen eingehen, über Konnektoren oder Delegationskonnektoren, die auf der der Schnittstelle gegenüberliegenden Seite des Ports wegführen, an die entsprechenden Parts weiter. Ebenso können Parts Aufrufe, die an benötigte Schnittstellen weiter geleitet werden sollen, mittels Konnektoren oder Delegationskonnektoren an einen Port herantragen. Die Aufrufe werden von dem Port dann an die benötigte Schnittstelle weiter geleitet. Diese Möglichkeiten sind in den Abbildungen Abbildung 2.4(a) und Abbildung 2.4(b) dargestellt. An die der Schnittstelle gegenüberliegenden Seite eines Ports können mehrere Konnektoren angebunden werden, wie in Abbildung 2.4(c) dargestellt. In diesem Fall ist die Weiterleitung eines Aufrufs nicht spezifiziert (vgl. OMG (2010b) (S. 187) und Hitz u. a. (2005) (S. 161)). Sollten nur an einer Seite eines Ports Konnektoren oder Schnittstellen anbinden, wie Abbildung 2.4(d) und Abbildung 2.4(e) zeigen, so terminieren die Aufrufe am Port.
Dargestellt wird ein Port durch ein Rechteck auf der Außenkante seines Klassifizierers. Beispiele für Ports lassen sich in allen Diagrammen in Abschnitt 2.1 finden. Konkret sei hier der Port *Verwaltungsport* der *Transportkomponente* (Abbildung 2.1) genannt. Der Name eines Ports wird durch einen Doppelpunkt von seinem Typ abgetrennt, dahinter folgt optional die Multiplizität in eckigen Klammern.

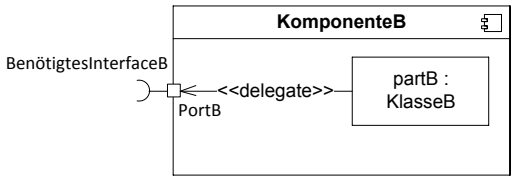
Für eine detaillierte Analyse der semantischen Variationen und offenen Punkte sei außerdem auf Cuccuru u. a. (2008) verwiesen.

Schnittstellen

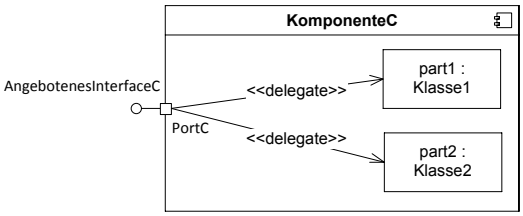
Schnittstellen spezifizieren die Operationen, die eine Klasse, Komponente oder ein Part anbietet oder verwendet. Eine angebotene Schnittstelle beschreibt die Funktionen, die das Element, zu welchem die Schnittstelle gehört, seiner Umgebung bereitstellt. Eine benötigte Schnittstelle beschreibt die Funktionen, die das Element von seiner Umgebung erwartet um korrekt arbeiten zu können. In dieser Arbeit wird ausschließlich die so genannte Ball-and-Socket-Notation verwendet. Bei dieser Notation wird eine angebotene Schnittstelle wie in Abbildung 2.1 die Schnittstelle *ITransportverwaltung* des *Verwaltungsports* mit einem Ballsymbol gezeichnet. Eine benötigte Schnittstelle wird durch ein Socketsymbol dargestellt (vgl. die Schnittstelle *ITransportmittelplanung* des *Transportmittelplanungsports* in Abbildung 2.1).



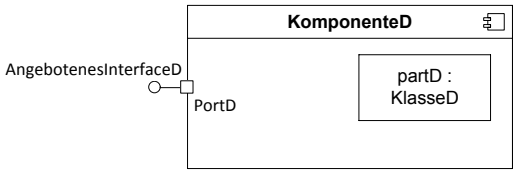
(a) Port mit einer angebotenen Schnittstelle und Delegation



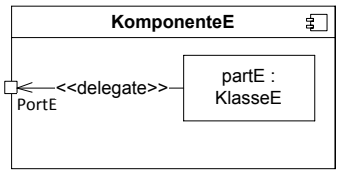
(b) Port mit einer benötigten Schnittstelle und Delegation



(c) Port mit einer angebotenen Schnittstelle und zwei Delegationen



(d) Port mit einer angebotenen Schnittstelle ohne Delegation



(e) Port mit einer Delegation ohne Schnittstelle

Abbildung 2.4.: Mögliche Weiterleitungen zwischen Ports, Schnittstellen und Parts

Konnektor

Ein Konnektor ist eine Verbindung zwischen zwei Elementen, die miteinander kommunizieren können. Er kann Klassen, Ports und Parts in allen Kombinationsvarianten miteinander verbinden und ist im Unterschied zu Assoziationen flexibler einzusetzen. Er spezifiziert nur, dass die miteinander verbundenen Elemente kommunizieren. Es lassen sich keine Aussagen darüber treffen, wie die verbundenen Elemente miteinander kommunizieren. Sie können also wie bei einer Assoziation im Klassendiagramm statisch miteinander verbunden sein. Es kann aber auch sein, dass dynamische, temporäre Verbindungen bestehen, wie beispielsweise durch Methodenaufrufe. Ein Konnektor wird durch einen Pfeil dargestellt, dessen Richtung die Kommunikationsrichtung angibt. Optional kann einem Konnektor ein Name gegeben werden. Dieser wird durch einen Doppelpunkt von einem optionalen Typnamen abgetrennt. An den Enden eines Konnektors können Multiziplitäten stehen, die die Anzahl von mit dem anderen Ende verbindbaren Instanzen begrenzen.

Abhängigkeit

Abhängigkeiten verbinden einen Klienten mit einem Anbieter. Der Klient ist semantisch abhängig vom Anbieter und muss bei einer Änderung des Anbieters wahrscheinlich ebenso angepasst werden. Die Form der Abhängigkeit kann durch die Verwendung von Stereotypen näher beschrieben werden (vgl. Kecher (2009)). In dieser Arbeit werden nur Abhängigkeiten mit den Stereotypen „use“ und „instantiate“ verwendet. Assoziationen können grundsätzlich zwischen verschiedenen Elementen der Diagramme bestehen. Doch auch hier wird in dieser Arbeit die Einschränkung vorgenommen, dass sie nur zwischen Parts existieren dürfen.
Laut der Spezifikation OMG (2010b) beschreibt eine „use“-Abhängigkeit, dass der Klient den Anbieter benötigt. Die genau Art und Weise der Verwendung ist nicht spezifiziert. Es kann sich also um eine statische oder dynamische Verbindung handeln. In Kecher (2009) wird die „use“-Abhängigkeit unter Anderem und beispielsweise verwendet, um deutlich zu machen, dass eine Klasse ein angebotenes Interface benötigt um korrekt arbeiten zu können. Hitz u. a. (2005) verwendet die „use“-Abhängigkeit ebenfalls in diesem Zusammenhang aber auch in einem Beispiel in welchem eine Klasse eine andere Klasse verwendet. Die Semantik der Abhängigkeit ist also vielseitig verwendbar.
Die „instantiate“-Abhängigkeit werden in der Spezifikation der UML (OMG (2010b)) nur beispielsweise genannt. Sowohl nach Hitz u. a. (2005) als auch nach Kecher (2009) besagt diese Art der Abhängigkeit, dass der Klient durch den Anbieter instanziiert wird. Hitz u. a. (2005) stellt die „instantiate“-Abhängigkeit zudem als eine Unterart der „use“-Abhängigkeit dar, da sie beide besagen, dass zwischen Klient und Anbieter eine Verwendungsbeziehung vorliegt.
Abhängigkeiten werden generell durch einen gestrichelten Pfeil vom Klienten zum Anbieter

dargestellt. Wenn ein Stereotyp spezifiziert ist, so wird er in doppelten Spitzen Klammern („«“, „»“) an die Linie geschrieben. Beispielsweise existiert eine „use“-Abhängigkeit zwischen *anwendungsfallTransportverwaltung* als Klienten und *transportauftrag* als Anbieter in Abbildung 2.1.

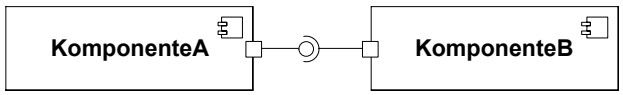
Delegationskonnektor

Ein Delegationskonnektor verbindet Schnittstellen mit Elementen, die diese Schnittstellen verwenden oder realisieren. Er stellt die bereits beschriebene Verbindung zwischen einem Klassifizierer und einem seiner Parts dar. Der Part realisiert oder verwendet eine Funktionalität, die durch ein angebotenes bzw. benötigtes Interface des Klassifizierers spezifiziert wird. Die konkrete Implementierung der Komponente kann also durch diese Konnektorart spezifiziert werden. Optional kann ein Delegationskonnektor auch einen Port, der Schnittstellen verwendet oder bereitstellt und entsprechende Parts verbinden. Dies ist beispielsweise in Abbildung 2.1 zwischen dem *Verwaltungsport* mit der angebotenen Schnittstelle *ITransportverwaltung* und dem Part *anwendungsfallTransportverwaltung* zu sehen. Da die Schnittstelle in diesem Fall angeboten wird, bedeutet der Delegationskonnektor, dass der Part die Funktionalität der Schnittstelle bereitstellt. Im Gegensatz dazu bedeutet die Delegation von dem Part *transportVerwalter* zum *Persistenzport* mit der benötigten Schnittstelle *IPersistenz* in Abbildung 2.1, dass der Part die Schnittstelle verwendet. Wie in den Abbildungen zu sehen ist, wird ein Delegationskonnektor durch einen gestrichelten Pfeil mit dem Stereotyp «delegate» dargestellt.

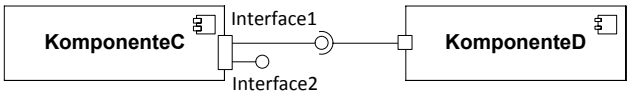
Kompositionskonnektor

Ein Kompositionskonnektor verbindet Parts mit Parts oder Ports mit Ports. Dadurch wird spezifiziert welcher Part bzw. Port Funktionen für einen anderen Part bzw. Port bereitstellt. Üblicherweise geschieht dies durch die Ball-and-Socket-Notation, wie zum Beispiel in Abbildung 2.1 zwischen den Parts *anwendungsfallTransportmittelplanung* und *verwalterTransport* dargestellt ist. Der Part *verwalterTransport* stellt in diesem Fall eine Schnittstelle bereit, die von *anwendungsfallTransportmittelplanung* benötigt und verwendete wird.

Laut der Spezifikation der UML OMG (2010b) dürfen Kompositionskonnektoren in der Ball-and-Socket-Notation nur an einfachen Ports, d. h. Ports, die nur eine Schnittstelle anbieten oder benötigen, verwendet werden. Dies hat den Grund, dass die Semantik eines Kompositionskonnektors nicht den expliziten Namen der Schnittstelle des Ports enthält, auf die sich der Konnektor bezieht. Stattdessen wird nur der Part bzw. Port durch den Konnektor verbunden.



(a) Kompositionskonnektor an einfachem Port



(b) Kompositionskonnektor an komplexem Port

Abbildung 2.5.: Kompositionskonnektoren an einfachen und komplexen Ports

Der Kompositionskonnektor in Abbildung 2.5(a) sagt aus, dass *KomponenteA* eine Funktionalität von *KomponenteB* benötigt. In dem Diagramm ist erkennbar, dass hier nur durch die Schnittstelle *Interface1* auf die Komponente zugegriffen werden kann. In Abbildung 2.5(b) dagegen ist nur im Diagramm ersichtlich, dass die *KomponenteD* über das *Interface1* auf die *KomponenteC* zugreift. Bei der Übersetzung dieses Diagramms in XML geht diese Information verloren und es ist nicht mehr klar, welche Schnittstelle des Ports der *KomponenteC* durch den Kompositionskonnektor genutzt werden soll. Im Fall von Abbildung 2.5(a) kann, da nur eine Schnittstelle zur Verfügung steht, nur diese für die Nutzung in Frage kommen und darum auf diese zugegriffen werden.

2.3. Codegenerierung

Zur Codegenerierung gibt es mehrere Möglichkeiten. Vier Ansätze sollen zunächst beschrieben werden. Da bereits zu Beginn der Arbeit festgelegt wurde, dass die Implementierungssprache des Codegenerators Visual C# 2010 sein soll, wurden nur Ansätze betrachtet, die sich generell mit dieser Sprache umsetzen lassen. Ein Vergleich und eine Entscheidung für einen dieser Ansätze befinden sich in Abschnitt 4.4.

Beispielhaft wird jedem der hier aufgeführten Ansätze eine Liste mit den drei modellierten Komponentennamen des Fallbeispiels übergeben, zu deren Namen jeweils eine Klasse mit dem leeren Standardkonstruktor in C# erstellt werden soll. Eine solche zu generierende Ausgabedatei ist für die Transportkomponente in Listing 2.1 zu sehen. Dieses Beispiel wird jeweils dazu genutzt, um das Verfahren zu erklären und später auch zu bewerten.

```
1  // Dieser Code wurde durch ein Tool generiert.
2  // Manuelle Änderungen werden bei erneuter Generierung
       überschrieben!
3  using System;
4
5  namespace Example.Transportkomponente
6  {
7    public class Transportkomponente
8    {
9      public Transportkomponente() { }
10   }
11 }
```

Listing 2.1: Templatevorlage für eine Klasse mit leerem Standardkonstruktor in C#

2.3.1. Konkatenation von Strings in C#

Der erste Ansatz zur Codegenerierung besteht darin, den Code ohne weitere Hilfsmittel durch Stringkonkatenationen in der Implementierungssprache des Codegenerators zusammen zu setzen. Die grundsätzliche Idee dieser Methode ist, dass Codeblöcke, die Schlüsselwörter und Satzzeichen der zu generierenden Programmiersprache enthalten, als Strings mit variablen Inhalten aus den Eingabedaten, wie beispielsweise Klassennamen und Typen, konkateniert werden.

Für jeden Komponentennamen des Beispiels wird sukzessive in mehreren Methoden ein String erzeugt, der die zu generierende Klasse enthält. Um beispielsweise den Namensraum der Klasse anzulegen (Zeilen 4, 5 und 10 in Listing 2.1), wird die in Listing 2.2 zu sehende Methode *GeneriereNamensraum(string namensraumName)* aufgerufen.

```
1  private string GeneriereNamensraum(string namensraumName) {
2    return "\nnamespace " + namensraumName + "\n{";
3  }
```

Listing 2.2: Methode zur Generierung des Namensraumes einer Klasse

Ihr Rückgabewert wird dann mit den Rückgabewerten anderer Methoden konkateniert und in einer .cs-Datei gespeichert. Wie in Listing 2.2 zu sehen ist, erzeugt diese Methode nur die Zeilen 4 und 5 des Listings 2.1. Die Zeile 10 mit der schließenden Klammer des Namensraumes wird durch die Methode *GeneriereSchliessendeKlammer(int tiefe)*, in Listing 2.3 zu sehen, erzeugt. Da diese Methode für mehrere schließende Elemente verantwortlich ist, wird

mit Hilfe des Parameters *depth* die Anzahl der Einrückungsschritte angegeben. Dadurch wird der generierte Code lesbarer. Die Lesbarkeit ist besonders wichtig, da keine ausführbare Klasse erzeugt wird, sondern das generierte Ergebnis noch manuell erweitert werden muss.

```
1 private string GeneriereSchliessendeKlammer(int tiefe) {
2   string ergebnis = "\n";
3   for (int i = 0; i < tiefe; i++) {
4     ergebnis += "\t";
5   }
6   return ergebnis + "}";
7 }
```

Listing 2.3: Methode zur Erzeugung einer schließenden Klammer mit Beachtung der Einrückungstiefe

Eine Möglichkeit, die Methode *GeneriereSchliessendeKlammer(int tiefe)* zu umgehen, wäre, die schließenden Elemente mit in der Methode zu setzen, in der auch das öffnende Element erzeugt wird. Die dazwischen liegendenen zu generierenden Teile würden dann durch einen verschachtelten Aufruf von anderen Methoden erzeugt, wie Listing 2.4 andeutungsweise zeigt. Die Methode *GeneriereKlasse(string klassenname)* wird geschachtelt innerhalb des Aufbaus des umgebenden Namensraumes aufgerufen. In ihr werden weitere, hier nicht gezeigte, Methoden aufgerufen, die für das Generieren der inneren Elemente einer Klasse, wie Felder oder Methodendefinitionen, zuständig sind.

```
1 private string GeneriereNamensraum(string namensraumName, string
    klassenname) {
2   return "\nnamespace " + namensraumName + "\n{" +
3          GeneriereKlasse(klassenname) +
4          "\n}";
5 }
```

Listing 2.4: Verschachtelter Aufruf von Methoden zur Stringkonkatenation

2.3.2. Text Template Transformation Toolkit

Das Text Template Transformation Toolkit (T4) ist ein Framework zur Code Generierung von Microsoft, das bereits in Visual Studio integriert ist (Msdn (2011a)). Darum ist grundsätzlich keine weitere Installation notwendig. Es wird jedoch empfohlen, eine Erweiterung für das Visual Studio zu installieren, welche Syntax-Highlighting und Intelli-Sense-Unterstützung bietet. In dieser Arbeit wurde der tangible T4 Editor (Tangible (2011)) verwendet.

Wie in Msdn (2011a) beschrieben, unterscheidet T4 zwischen Templates, die zur Entwurfszeit im Visual Studio ausgeführt werden und Templates, die zur Laufzeit einer Anwendung ausgeführt werden. Die Templates, die bereits zur Entwurfszeit ausgeführt werden, generieren dabei einen Teil des Quelltextes für die zu entwickelnde Anwendung. Für den Codegenerator werden jedoch Laufzeit-Templates, sogenannte Preprocessed Text Templates, benötigt. Diese werden in die Anwendung eingebunden und erhalten Daten aus den Eingabedateien. Zu jedem Template wird automatisch eine Klasse mit dem Namen dieses Templates generiert und bei jeder Templateänderung aktualisiert. Die Datenübergabe an ein Template erfolgt durch eine partielle Klasse, welche die dem Template zugeordnete Klasse erweitert. Die partielle Klasse kann Properties enthalten, auf die dann aus dem Template heraus zugegriffen werden kann.

Eine partielle Klasse stellt beispielsweise eine String-Property bereit, die durch den Konstruktoraufruf dieser partiellen Klasse oder durch einen Setter mit einem Wert initialisiert werden kann. In dem Template kann dann auf die Properties dieser Klasse zugegriffen werden und der Wert in die zu generierende Klasse eingetragen werden. Dies geschieht beispielsweise in Listing 2.5 in Zeile 9.

Um das Template auszuführen wird ein Objekt der Templateklasse über den Konstruktor der partiellen Klasse angelegt und mit variablen Werten gefüllt. Die generierte Templateklasse stellt eine Methode *TransformText()* bereit, die einen String mit dem generierten Inhalt des Templates zurück gibt. Dieser String kann dann in eine .cs-Datei gespeichert werden.

Der Code in den Templates wird in C# oder Visual Basic geschrieben. Zusätzlich zur Auswertung von Feldern kann ein Template auch durch die Kontrollstrukturen von C# bzw. Visual Basic beeinflusst werden. Dazu gibt es verschiedene Escape-Blöcke, die unterschiedliche Wirkungen haben.

<# #> Dieser Block enthält Kontrollstrukturen, die ausgeführt werden, um die Generierung der Ausgabe zu steuern.

<#@ #> Dieser Block setzt Eigenschaften, die für das Template gelten sollen. In jedem Template muss z. B. die verwendete Sprache gesetzt werden. In Listing 2.5 wird dies in Zeile 1 für C# gemacht.

<#= #> Dieser Block dient dazu, den Wert der angegebenen Variablen auszulesen oder die Anweisung auszuwerten und deren Ergebnis in den generierten Text zu schreiben.

<#+ #> Innerhalb dieser Blöcke können Hilfsmethoden geschrieben werden, die dann innerhalb des Templates verwendet werden können. Dies bietet den Vorteil, Elemente der Templates wiederzuverwenden. Darum wird in Msdn (2011a) auch empfohlen, solche Hilfsmethoden in eigene Templates auszulagern und bei Bedarf in das Template zu importieren.

```
1  <#@ template debug="false" hostspecific="false" language="C#" #>
2  <#@ output extension=".cs" #>
3  // Dieser Code wurde durch ein Tool generiert.
4  // Manuelle Änderungen werden bei erneuter Generierung
       überschrieben!
5  using System;
6
7  namespace TFourBeispiel.<#= this.Klassenname #>
8  {
9    public class <#= this.Klassenname #>
10   {
11       public <#= this.Klassenname #>() { }
12   }
13 }
```

Listing 2.5: Beispiel eines T4 Templates

2.3.3. NVelocity

NVelocity (NVelocity (2011)) ist eine Portierung des Apache Velocity Projektes (Velocity (2011)) auf die .NET-Plattform. Velocity ist eine Java-basierte Template-Engine, die es erlaubt, Textdateien aller Art und damit auch Codedateien zu generieren. Das Framework unterstützt die Trennung von Templates und Daten, die in diese Templates einfließen, durch das Model-View-Controller-Pattern (MVC). In den Templates von NVelocity kann mit Hilfe einer eigenen Markup-Sprache auf die Daten eines sogenannten „Contextes“ zugegriffen werden. Dieser bietet Getter und Setter um Daten in das Template zu schreiben oder aus ihm heraus zu lesen. Die Markup-Sprache bietet bedingte Anweisungen sowie Kontrollstrukturen zur Iteration über Collections an.
Das Initialisieren des Contexts ist beispielhaft in Listing 2.6 in den Zeilen 3 und 5 dargestellt. In Zeile 7 wird dieser Context dann mit dem Template, welches in Zeile 1 initialisiert wird, vereint und die Ausgabe in einen Stringwriter geschrieben. Mit dessen Hilfe kann dann die .cs-Ausgabedatei geschrieben werden.

```
1  Template template = velocity.GetTemplate("pfad\\zum\\template\\
       NVelocityTemplate.vm");
2  //...
3  VelocityContext context = new VelocityContext();
4  //...
5  context.Put("name", componentName);
```

```
6 //...
7 template.Merge(context, writer);
```

Listing 2.6: Initialisierung und Aufruf der Codegenerierung mit NVelocity

2.3.4. StringTemplate

StringTemplate (StringTemplate (2011a)) ist eine Java-basierte Template-Engine, die eine Portierung für C# bereitstellt. Sie wird im Rahmen des ANTLR Projektes (Antlr (2011)) entwickelt. Da StringTemplate ursprünglich für die Webentwicklung entwickelt wurde, zeichnet es sich durch die Trennung von Templates und Logik aus. Die Templates sollen die Beschreibung einer Webseite beinhalten, die Logik die eigentliche Anwendung hinter der Webseite. Diese Trennung ist deutlich durch die wenig umfangreiche Templatesprache sichtbar. Die Templatesprache kommt mit Attributreferenzen, Referenzen auf andere Templates, bedingten Anweisungen und Listenoperationen aus. Um die Logik der eigentlichen Anwendung hinter der Webseite zu programmieren werden meistens mehr Kontrollstrukturen benötigt. Somit wird die Trennung zwischen der Beschreibung der Webseite und ihrer Logik unterstützt.
Des Weiteren unterstützt StringTemplate Gruppen von Templates, wie in Listing 2.7 in Zeile 2 verwendet. Diese gruppieren die Templates eines Ordners, sodass man aus dieser Gruppe verschiedene Templates je nach Bedarf laden kann. Die variablen Daten werden mit Hilfe von Settern an das Template übergeben, wie in Zeile 5 in Listing 2.7 zu sehen ist.

```
1 //...
2 StringTemplateGroup stg = new StringTemplateGroup("eineGruppe","
    pfad\\zum\\template\\gruppe");
3 StringTemplate st = stg.GetInstanceOf("StringTemplateTemplate");
4 //...
5 st.SetAttribute("name", name);
6 //...
```

Listing 2.7: Initialisierung und Aufruf der Codegenerierung mit StringTemplate

2.4. Verwandte Arbeiten

Es existieren bereits verschiedene Lösungen zur Codegenerierung. Diese lassen sich grob in ganze, meist kommerzielle, Modellierungssuiten und in kleinere, auf spezielle Diagramme oder Probleme konzentrierte Anwendungen unterteilen.

Auf Seiten der großen Systeme sind beispielsweise der Enterprise Architect (SparxSystems (2011)), hier in der Version 8.0 vorliegend, und Visual Paradigm (VisualParadigm (2011)), hier in Version 8.1 betrachtet, zu erwähnen. Beide Anwendungen bieten einen Editor zur graphischen Modellierung der verschiedenen Diagrammarten der UML. Aus diesen Diagrammen kann Code in verschiedenen gängigen Programmiersprachen generiert werden. Dabei ist aber zu beachten, dass vor allem die Unterstützung von Klassendiagrammen und teilweise Sequenzdiagrammen gegeben ist. Die Unterstützung der hier betrachteten Komponenten- und Kompositionsstrukturdiagrammen konnte nicht festgestellt werden. Hinzu kommt, dass diese Diagramme in dieser Arbeit gemischt verwendet werden, um speziell Komponenten ausführlich beschreiben zu können. Solche speziellen Aspekte lassen sich durch diese großen Anwendungen nicht umsetzen, da sie eine breit gefächerte Zielgruppe ansprechen soll, welche unterschiedlichste Funktionen benötigt.

Um spezielle Probleme durch einen Codegenerator zu beschreiben und zu lösen, eignet sich die zweite erwähnte Gruppe der Codegeneratoren. Hier ist beispielhaft der Entwurf eines Codegenerators zur Generierung von ausführbarem Code aus Klassen-, Sequenz- und Aktivitätsdiagrammen von Usman und Nadeem zu nennen (Usman u. a. (2008) und Usman und Nadeem (2009).

3. Analyse der Komponenten- und Kompositionsstrukturdiagramme

Die in Abschnitt 2.2 beschriebenen UML-Elemente sollen in diesem Kapitel analysiert werden und ihre Abbildung von XML auf Code beschrieben werden. Der Code, der mit dem Generator erzeugt werden soll, ist C#-Code für das .NET Framework 4. Darum bezieht sich die Analyse auf Code dieser Sprache. Eine Analyse für weitere Sprachen führt an dieser Stelle zu weit und ist nicht Teil dieser Arbeit. Da beispielsweise Java und C# aber syntaktisch ähnlich sind, kann davon ausgegangen werden, dass eine Abbildung in die Syntax von Java mit wenig Aufwand vollführt werden könnte. Des Weiteren soll diese Analyse möglichst auf konzeptueller Ebene geschehen, sodass auch eine Portierung auf weitere Programmiersprachen, die syntaktisch nicht so ähnlich sind, wie Java und C#, möglich ist.

Die Auswahl der Diagrammelemente erfolgte unter dem Gesichtspunkt der Darstellung und Modellierung von Komponenten. Es sollten zum einen möglichst genaue Aussagen über die Komponenten selbst getroffen werden können. Dies wird vor allem durch die Verwendung der Komponentendiagramme möglich. Zum Anderen sollen aber nicht nur die Referenzen der Komponente selbst aussagekräftig sein. Viel mehr sollen auch die Zusammenhänge der Interna einer Komponente eigenständig und unabhängig von ihrem umgebenen Klassifizierer möglichst genau beschrieben werden. Hierzu wurden vor allem die verschiedenen Konnektorarten mit deren teilweise semantischen Änderungen, die im folgenden Kapitel beschrieben werden, in die Menge der Diagrammelemente hinzugefügt.

3.1. Analyse und Übersetzung der Diagrammelemente

Aus den Diagrammen lassen sich grundsätzlich die Klassenrümpfe mit ihren Verbindungen zu einander erzeugen. Auch die Gruppierungen von Klassen, die durch Komponenten abgebildet werden, lassen sich durch eine geeignete Ordnerstruktur darstellen.

3.1.1. Statische Elemente

Komponente

Komponenten bilden eine größere Einheit, die mehrere Klassen strukturieren und kapseln. Viele Programmiersprachen, darunter auch C#, bieten keine konkrete Struktur, um Komponenten direkt auf den Code abzubilden. Es ist jedoch möglich, das Konzept einer Komponente durch die Ordnerstrukutur und, zumindest im Fall von C#, durch verschiedene Namensräume und Projekte innerhalb einer Solution[1] darzustellen.

Im Fall der konkret verwendeten Programmiersprache C# wäre es denkbar, eine Solution für das ganze Modell zu erzeugen. Für jede Komponente des Gesamtmodells kann ein Projekt innerhalb dieser Solution angelegt werden. Die Projekte erhalten unterschiedliche Namensräume, sodass damit auch im Code selber, nicht nur in seiner Ordnerstruktur, die Komponenten sichtbar werden. Um Solutions und Projekte anzulegen, bedarf es einiger Konfigurationsdateien, die die Einstellungen der einzelnen Projekte wie auch ihre Abhängigkeiten untereinander beschreiben. Da der Entwurf der Generierung dieser Dokumente sehr viel Zeit in Anspruch nehmen würde und dieses Thema nicht Fokus dieser Arbeit ist, wurde darauf verzichtet.

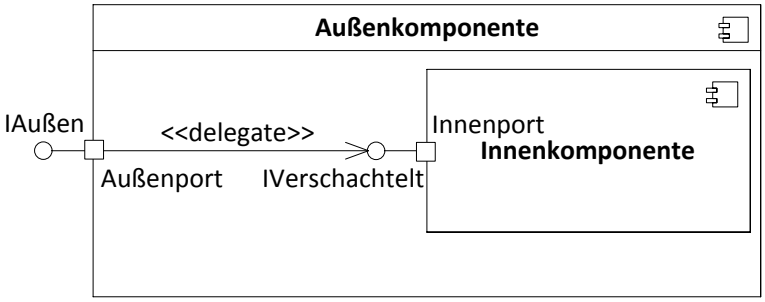
Stattdessen wird für jede Komponente in diesem Codegenerator ein Ordner im Dateisystem angelegt. Dieser Ordner enthält alle Klassen, die der Komponente zugeordnet werden. Somit ist die, unter Umständen große, Anzahl von Klassen strukturiert und die Komponenten können auch von mehreren Entwicklern gleichzeitig und unabhängig voneinander entwickelt werden. Der Entwickler kann bei Bedarf manuell eine Solution im Visual Studio mit entsprechenden Projekten pro Komponente anlegen und die generierten Klassen und Interfaces importieren. Dieser Schritt kostet sehr viel weniger Zeit, als wenn der Entwickler die Klassen- und Interfacerümpfe manuell selbst anlegen müsste.

Um die internen und externen Details einer Komponente, wie beispielsweise ihre Parts, Ports und Schnittstellen, in Code abzubilden und an einem Punkt zu sammeln, bedarf es eines Einstiegspunktes auf Codeebene. Darum wird für jede Komponente des Modells eine Klasse mit dem Komponentennamen angelegt. Diese Klasse dient als Ausgangspunkt um die Instanziierung der Bestandteile der Komponente anzustoßen. Dies bedeutet nicht zwingend, dass diese Klasse die Instanziierung selbst vornehmen muss. Viel mehr kann, wie im Fallbeispiel vorgesehen, z. B. ein Konfigurator initialisiert werden, der dann die anderen Teile der Komponente richtig zusammensetzt und initialisiert. Der Aufbau der Komponentenklasse erfolgt analog zu dem aller Partklassen und wird im nächsten Abschnitt näher erläutert.

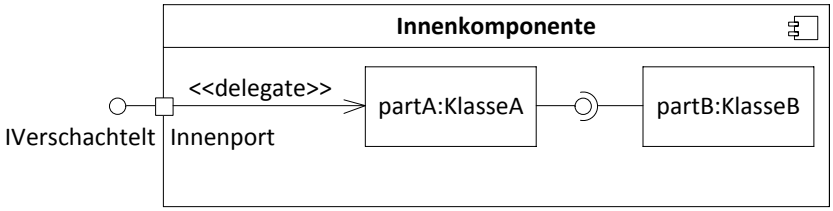
[1] Gruppierung von einem oder mehreren Projekten, die zusammen eine Anwendung bilden (Msdn (2011b)).

Für jede Komponente wird ein Namensraum mit dem Namen der Komponente angelegt und alle Klassen, die der Komponente zugeordnet sind, erhalten diesen. Auf diese Weise kann man bereits im Code erkennen, zu welcher Komponente eine Klasse gehört.

Komponenten können verschachtelt sein, wie Abbildung 3.1(a) zeigt. Eine verschachtelte Komponente kann nicht weiter detailliert oder untergliedert werden, allerdings kann diese Untergliederung in einem anderen Diagramm, hier Abbildung 3.1(b) spezifiziert sein. Existiert solch eine Konstruktion, so wird die innere Komponente wie oben beschrieben aufgebaut. Die äußere Komponente erhält Referenzen auf die angebotenen Schnittstellen der inneren Komponente.



(a) Außenansicht einer verschachtelten Komponente



(b) Innenansicht einer verschachtelten Komponente

Abbildung 3.1.: Beispiel zur Innen- und Außenansicht einer verschachtelten Komponente

Komponenten benötigen eindeutige Namen über Diagrammgrenzen hinweg, damit Informationen zu einer Komponente in verschiedenen Diagrammen jeweils der richtigen Komponente zugeordnet werden können. Ansonsten wäre es beispielsweise nicht möglich, die einzelnen Komponentendiagramme des Fallbeispiels sowie sein Komponentenübersichtsdiagramm einzulesen und die Verbindung zwischen der Innenansicht einer Komponente in seinem jeweiligen Komponentendiagramm sowie der Außenansicht der Komponente im Komponentenübersichtsdiagramm herzustellen.

Datentyp

Datentypen werden von C# durch das Konstrukt *struct* umgesetzt und können demnach auch von dem Codegenerator direkt generiert werden. Da die Datentypen nur als Typ eines Parts auftreten können, sind diese generierten *structs* leer und müssen von dem Entwickler manuell angepasst und erweitert werden. Pro Datentyp wird eine eigene Datei angelegt, die den Namen des Datentyps trägt und in den Ordner der zugehörigen Komponente abgelegt.

Enumeration

C# unterstützt das Konzept der Enumeratioens durch die *enum*-Struktur. Darum kann eine Enumeration der UML direkt in diese Struktur übersetzt werden. Aus dem Diagramm bekannte Elemente der Enumeration können in die Elementliste der enum-Struktur eingetragen werden. Da nicht bekannt ist, wie viele Enumerations in einer Komponente existieren, wird pro Enumeration eine eigene Datei angelegt, die den Namen dieser Enumeration trägt. Es wäre auch denkbar, alle Enumerationen einer Komponente in eine einzige Datei zu schreiben, allerdings könnte diese schnell unübersichtlich werden. Eine nun möglicherweise große Menge an Enumeration-Dateien wird dadurch strukturiert, dass die Dateien in einen Ordner „Enumerations“ der zugehörigen Komponente gespeichert werden. Klassen, die eine Referenz auf die Enumeration benötigen, erhalten durch einen Import des Namensraums der Enumeration die Möglichkeit, diese zu verwenden. Da die Elemente einer Enumeration bereits im Diagramm angegebene werden können, ist diese Struktur die einzige, die vollständig durch den Generator generiert werden kann. Sie muss zur Verwendung nicht mehr durch einen Entwickler angepasst werden.

Part

Ein Part wird durch eine Referenz auf die Klasse seines Typs umgesetzt. Sein Name stellt den Namen dieser Referenz dar, seine Sichtbarkeit die Sichtbarkeit der Referenz.

Die Multiplizität des Parts wird durch die Art der Referenz umgesetzt. Wenn die Multiplizität genau „1“ beträgt, so enthält die Klasse ein Feld vom Typ des Parts. Dann kann genau eine Instanz des Parts genutzt werden. Bei mehr erlaubten Instanzen wird eine Liste mit Elementen vom Typ des Parts als Feld gespeichert. Durch diese sind Parts mit einer Multiplizität zwischen „0“ und „*“ umgesetzt. Wenn die Multiplizität eine bestimmte Höchstanzahl, beispielsweise „3“, vorgibt, werden genau so viele Felder für dieses Element generiert, wie die Höchstanzahl vorgibt. Der Standardwert der Multiplizität, der verwendet wird, wenn diese nicht explizit angegeben wurde, beträgt „1“. Die den Part besitzende Klasse erhält also ein Feld vom Typ des Parts.

Da ein Part eine oder mehrere Instanzen einer Klasse darstellt, ist der Typ des Parts genau diese Klasse. Dementsprechend muss für jeden Typ eines Parts eine Klasse generiert werden. Der Typ eines Parts muss angegeben werden, sonst wäre es nicht möglich, der Komponente, welche den Part besitzt, eine Referenz auf diesen zu geben. Darum ist dies eine der wenigen Anforderungen an das Diagramm, die gegeben sein müssen, um den Code korrekt zu generieren.

Wenn der Part in einer echten Kompositionsbeziehung zu einer Komponente steht, so gehört die Klasse seines Typs zu dieser Komponente und wird in deren Ordner gespeichert und in deren Namensraum angelegt. Die Unterscheidung zwischen Kompositions- und Aggrgationsbeziehungen wird vom Enterprise Architect 8.0 nicht unterstützt. Darum kann die Abbildung einer Aggregationsbeziehung zwischen Part und Komponente an dieser Stelle nur hypothetisch erfolgen, sie soll aber dennoch beschrieben werden. Der zu entwickelnde Codegenerator wird diesen Aspekt dann nicht umsetzen können.
Wenn der Part in einer Aggregationsbeziehung zu der Komponente steht, so gibt es entweder in einer anderen Komponente eine Klasse dieses Namens, auf die dieser Part verweist und deren Namensraum dann importiert wird. Wenn dies nicht der Fall ist, dann wird eine Klasse vom Typ des Parts erstellt und in den Hauptausgabeordner gespeichert. Der Anwender des Codegenerators muss dann entscheiden, zu welcher Komponente die Klasse gehört, sie entsprechend verschieben und, bei Referenzen auf diese Klasse in anderen Komponenten, den Namensraum der Komponente importieren.

Um die Instanziierung eines Parts zu ermöglichen, erhält seine Klasse einen privaten leeren Konstruktor sowie einen privaten vollständigen Konstruktor, durch den alle Felder der Klasse initialisiert werden können. Die konkrete Erzeugung einer Instanz geschieht über den Aufruf einer statischen Methode, die den privaten vollständigen Konstruktor aufruft. Diese Methode wird verwendet, um dem Entwickler die Möglichkeit zu geben, das Singleton-Muster nach Gamma u. a. (1995), einfach anzuwenden. In dem Fallbeispiel wäre es z. B. sinnvoll, die Anwendungsfälle, Verwalter, Konfiguratoren und Komponentenklassen als Singletons zu implementieren. Da die Semantik eines Parts mit der Multiplizität „1“ jedoch nicht ausschließt, dass seine Klasse an anderer Stelle mit einer höheren Multiplizität verwendet wird, kann das Singleton-Muster nicht direkt generiert werden.

Port

Ein Port kapselt das Element, zu dem es gehört, stärker von der Umwelt ab, als es die bloße Nutzung von Schnittstellen tun würde. Der Port leitet Aufrufe an die Schnittstellen, die er besitzt, weiter. Um Aufrufe der Umwelt an die internen Parts weiterleiten zu können, benötigt der Port Referenzen auf die entsprechenden Parts. Die Referenzen auf die internen Parts ergeben sich durch Delegationskonnektoren, die an den Port heranführen. Andersherum,

um Aufrufe der Parts an angebotene Schnittstellen anderer Komponenten weiterleiten zu können, benötigt der Port Referenzen auf die entsprechenden angebotenen Schnittstellen. Diese Referenzen ergeben sich aus den benötigten Schnittstellen, die der Port besitzt. Dabei kann es sein, dass die Schnittstelle von den Interna der Komponente nicht verwendet wird, da kein Delegationskonnektor von einem Part zu dem Port führt, der auf diese Schnittstelle zugreift. Dennoch muss der Port die Funktionalität der Schnittstelle anbieten, damit dessen Nutzung zu einem späteren Zeitpunkt möglich ist.

Ein Port muss seine angebotenen Schnittstellen implementieren, damit die Umwelt der Komponente diese nutzen kann. Diese Implementierung kann in der Weiterleitung an einen Part bestehen, der die eigentliche Funktionalität implementiert. Dennoch müssen die Aufrufe aus der Umwelt zunächst an den Port geleitet werden, um die Kapselung zu erhalten.

Der Typ eines Ports ist entweder direkt angegeben oder er wird durch seine Schnittstellen bestimmt. Da der Port die Funktionalität der Schnittstellen bereitstellen muss, bietet der Port eine einheitliche Schnittstelle für die internen Bestandteile der Komponente. Die internen Bestandteile sehen nur den Port und wissen nicht, dass dieser Aufrufe an die Umgebung der Komponente weiter leitet. Genauso merken die internen Bestandteile nicht, dass ein Aufruf, der vom Port zu ihnen herangetragen wird, aus der Umgebung der Komponente stammt. Somit umfasst der Typ des Ports genau die Schnittstellen, die er anbietet und benötigt.

Die Multiplizität eines Ports wird wie bei den Parts umgesetzt. Bei nicht angegebener Multiplizität ist der Standardwert „1“, sodass die besitzende Komponente genau eine Referenz auf den Port besitzen darf. Bei mehreren möglichen Instanzen erhält die Komponentenklasse eine Liste vom Typ des Ports, um beliebig viele oder auch eine minimale oder maximale Anzahl speichern zu können. Damit die Komponentenklasse dies tun kann, benötigt die Klasse des Ports einen privaten Konstruktor sowie eine Factorymethode, in welcher die Prüfung der Multiplizität stattfindet, bevor diese Methode den Konstruktor aufruft.

Schnittstelle

Wie in Abschnitt 2.2 beschrieben, bietet das Konzept der Ports eine verstärkte Kapselung der Komponente von ihrer Umwelt. Da die Kapselung aus genannten Gründen von Vorteil ist, soll die Verwendung von Ports unterstützt werden. Darum dürfen benötigte und angebotene Schnittstellen nur an Ports und nicht direkt an Komponenten definiert werden.

Eine angebotene Schnittstelle kann direkt auf ein Interface in C# abgebildet werden, weil die Sprache das Konzept von Schnittstellen unterstützt. Da der Codegenerator bislang nur die Ball-and-Socket-Notation von Schnittstellen unterstützt, können keine Methodendeklarationen spezifiziert und generiert werden. Die Unterstützung von Schnittstellen mit Methodendeklarationen sollte in einer zukünftigen Ausbaustufe des Codegenerators hinzugefügt

werden. Damit wäre es möglich, komplette Interfaces zu generieren sowie den implementierenden Klassen bereits die entsprechenden Methodenrümpfe hinzuzufügen. Der Entwickler hätte so eine noch größere Arbeitserleichterung und es wäre eine weitere Fehlerquelle eliminiert.

Damit die Klasse, welche die Schnittstelle verwendet, auf deren Methoden zugreifen kann, wird eine benötigte Schnittstelle im Code als eine Referenz in dieser Klasse dargestellt. In C# wird dazu ein Feld erzeugt. Um das Feld zu initialisieren werden dem vollständigen Konstruktor und der Factorymethode der Klasse ein Parameter hinzugefügt, der vom Typ der Schnittstelle ist. Damit die Referenz auf die Schnittstelle verwendet werden kann, wird auch der Namensraum der Komponente des angebotenen Interfaces importiert. Andernfalls würde der Compiler bei dem Versuch, diese Klasse zu verwenden, einen Fehler werfen, da ihm das Interface nicht bekannt ist.

Es kann durch Unachtsamkeit des Architekten passieren, dass in der Modellierung des Systems eine benötigte Schnittstelle spezifiziert wird, aber keine entsprechende angebotene Schnittstelle existiert. In diesem Fall wird trotzdem eine Referenz in der verwendenden Klasse auf die nicht vorhandene Schnittstelle gesetzt und es wird ein Interface mit dem Namen der Schnittstelle angelegt. Da in diesem Fall eine Komponentenzuordnung nicht möglich ist, wird das Interface in den Ausgabeordner direkt gespeichert. Der Anwender wird über dieses potenzielle Problem informiert, sodass er das erzeugte Interface manuell in die richtige Komponente verschieben und diese in der Klasse, welche das Interface verwendet, durch einen Import des Namensraumes der Komponente bekannt machen kann.

3.1.2. Verbindungen zwischen statischen Elementen

Die Tabelle 3.2 gibt eine Übersicht über alle möglichen Verbindungen zwischen den verschiedenene Elementen in einem Diagramm, aufgeteilt in die verschiedenen Konnektorarten.

	Part einer Komponente	Part einer Klasse	Port einer Komponente	Port einer Klasse	Benötigte Schnittstelle an Port	Benötigte Schnittstelle an Part	Angebotene Schnittstelle an Port	Angebotene Schnittstelle an Part
Part einer Komponente	U I		D					
Part einer Klasse		U I		D				
Port einer Komponente	D							
Port einer Klasse		D						
Benötigte Schnittstelle an Port							A	
Benötigte Schnittstelle an Part								A
Angebotene Schnittstelle an Port					A			
Angebotene Schnittstelle an Part						A		

Legende: D = Delegationskonnektor, A = Kompositionskonnektor, U = „use“-Abhängigkeit, I = „instantiate“-Abhängigkeit

Tabelle 3.2.: Mögliche umsetzbare Konnektoren zwischen den Elementen eines Diagrammes

Konnektor

Konnektoren sind zwar, wie bereits in Abschnitt 2.2 beschrieben, ein Bestandteil von Kompositionsstrukturdiagrammen, lassen sich aber nicht in Code umsetzen, da ihre Semantik auch dynamische Verbindungen vorsieht und diese nicht dargestellt werden können. Aus diesem Grund sind auch keinerlei Einträge zu Konnektoren in Tabelle 3.2 zu finden. Statt einer direkten Umsetzung in Code, wie es beispielsweise eine Referenz von einer Klasse auf die andere wäre, wird für jeden Konnektor, welcher dennoch im Diagramm modelliert wurde, ein Kommentar in den Code der beteiligten Klassen eingefügt. Dieser Kommentar beinhaltet die Nachricht, dass ein Konnektor modelliert wurde und dieser im Code noch manuell realisiert werden sollte.

Abhängigkeiten

In Abschnitt 2.2 wurde die semantische Bedeutung von „use“- und „instantiate“-Abhängigkeiten nach OMG (2010b) eingeführt. In diesem Kapitel soll ihre semantische Bedeutung erweitert werden, mit dem Ziel, die Aussagekraft der Kompositionsstruktur- und Komponentendiagramme zu erhöhen. Ohne die Verwendung der „use“- und „instantiate“-Abhängigkeiten im Rahmen von diesen Diagrammen ergibt sich nachfolgendes Problem. Ein Konnektor der Kompositionsstrukturdiagramme stellt eine Verbindung zwischen zwei Parts einer Komponente bzw. einer Klasse her. Abbildung 3.2(a) beschreibt, dass *Konfigurator A* und *Verwalter A* in irgendeiner Form zusammenhängen. Diese Verbindung kann aber, wie auch in Abschnitt 2.2 beschrieben, jeglicher Art sein. Das bedeutet, dass diese Verbindung unter Umständen nur dynamisch, durch beispielsweise einen Methodenaufruf, besteht und statisch im Code nicht direkt sichtbar ist. Wenn es die Möglichkeit gibt, explizite und für das Grundgerüst einer Klasse wichtige, statische Verbindungen darzustellen, würde die Mächtigkeit der hier genutzten Diagrammarten steigen.

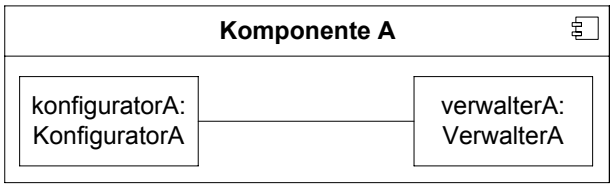
Dazu soll die Semantik der use- und instantiate-Abhängigkeiten erweitert werden. Wie die Tabelle 3.2 zeigt, können „use“- und „instantiate“-Abhängigkeiten zwischen den Parts einer Komponente bzw. Klasse bestehen. Sie werden in der Tabelle mit den Buchstaben „U“ („use“) und „I“ („instantiate“) gekennzeichnet.

Eine „use“-Abhängigkeit zwischen zwei Parts wird dahingehend verschärft, dass der eine Part den anderen Part statisch verwendet. Die Klasse des Klienten erhält hierbei eine Referenz auf die Klasse des Anbieters und kann diesen somit durch Zugriffe auf das Feld verwenden.

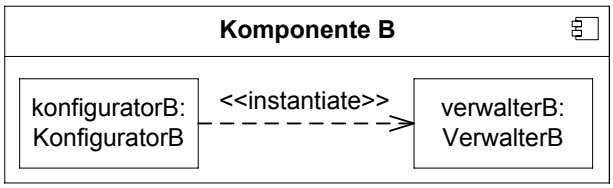
Eine „instantiate“-Abhängigkeit zwischen zwei Parts stellt, wie schon in Hitz u. a. (2005) beschrieben, einen Spezialfall der „use“-Abhängigkeit dar. Er sagt aus, dass ein Part den anderen ínstanziiert. Konkret bedeutet dies, dass der Klient die statische Factorymethode des Anbieters in einer eigenen Methode, welche eine Instanz des Anbieters zurück gibt, aufruft. Um die statische Factorymethode des Anbieters aufrufen zu können, muss der Klient den Namensraum des Anbieters importieren.
Die „instantiate“-Abhängigkeit ist wichtig, um die innere Struktur einer Komponente in Bezug auf ihre Initialisierung zu vervollständigen. Aus einem standardisierten Kompositionsstrukturdiagramm geht nicht hervor, welcher Part einer Komponente einen anderen instanziiert. In Abbildung 3.2(a) ist nicht erkennbar, dass der *konfiguratorA* den *verwalterA* instanziiert. Diese Information könnte nicht in die Codegenerierung eingehen und würde wiederum mehr Arbeit und eine Fehlerquelle für die Entwickler bedeuten. Die Verwendung der „instantiate“-Abhängigkeit in Abbildung 3.2(b) hingegen zeigt, dass der *konfiguratorB* den *verwalterB* instanziiert. Dadurch ist eine genauere Beschreibung der Komponente und ihrer internen Abhängigkeiten möglich. In Abbildung 3.2(a) würde die Komponentenklasse alle Bestandteile instanziieren müssen. In Abbildung 3.2(b) mit der Abhängigkeit kann die Instanziierung

gekapselt werden und der Code der eigentlichen Komponentenklasse schlank gehalten werden.

Wenn in einem Diagramm für einen Part keine „instantiate“-Abhängigkeit angegeben wurden, so wird standardmäßig davon ausgegangen, dass der umgebene Klassifizierer den Part instanziiert. Im Beispiel in Abbildung 3.2 würde in Teil a) also die Komponentenklasse die Instanziierung übernehmen.



(a) Verbindung ohne eine instantiate-Abhängigkeit



(b) Verbindung durch eine instantiate-Abhängigkeit

Abbildung 3.2.: Beispiel zum Nutzen von instantiate-Abhängigkeiten

Delegationskonnektor

Delegationskonnektoren können zwischen den Parts einer Komponente bzw. Klasse und deren Ports auftreten. Sie werden in der Tabelle 3.2 mit dem Buchstaben „D“ gekennzeichnet. Je nach Richtung des Konnektors implementiert oder verwendet ein Part die Schnittstellen des Ports. Um dies umzusetzen bedarf es Referenzen zwischen den Elementen.

Wenn der Delegationskonnektor vom Part zum Port führt, benötigt der Part eine Referenz auf den Port. Der Part kann dann die benötigten Schnittstellen des Ports verwenden.

Wenn der Delegationskonnektor vom Port zum Part führt, bedeutet dies, dass der Part die Funktionalität der am Port angebotenen Schnittstellen realisiert. Darum benötigt in diesem

Fall der Port eine Referenz auf den Part, um die Schnittstellenaufrufe von außerhalb der Komponente an den Part weiterleiten zu können.

Wie im Absatz über Schnittstellen bereits beschrieben, wäre es an dieser Stelle möglich, die Methodenrümpfe der angebotenen Schnittstellen auch in den entsprechenden implementierenden Parts zu generieren, sobald die Unterstützung von Schnittstellenmodellierungen mit Methodendeklarationen vorhanden ist. In der Portklasse könnten die implementierenden Methoden dann vollständig generiert werden, indem diese pro Methode der Schnittstelle eine Methode implementiert, welche den Aufruf an den Part weiter leitet. Dies funktioniert unter der Voraussetzung, dass der Port keine eigenen Berechnungen vornimmt. Außerdem müsste überlegt werden, wie der Port mit der Weiterleitung umgeht, wenn mehrere Parts diesen Port implementieren.

Kompositionskonnektor

Diese Konnektorart ist in Tabelle 3.2 mit dem Buchstaben „A“ gekennzeichnet. Kompositionskonnektoren kann es zwischen zwei Parts oder zwei Ports geben, wenn ein Part bzw. ein Port dem anderen Funktionen über eine Schnittstelle bereitstellt.

Die Schnittstelle eines Parts, der keine expliziten Schnittstellen bereit stellt, ist die Menge der sichtbaren Methoden dieses Parts. Stellt ein Part keine Schnittstellen bereit, so kann der Kompositionskonnektor dennoch durch eine Referenz in dem die Funktionalität benötigenden Part auf den die Funktionalität bereitstellenden Part übersetzt werden.

Ein Kompositionskonnektor zwischen einfachen Ports wird umgesetzt, indem der die Funktionalität der Schnittstelle benötigendem Port eine Referenz auf die Schnittstelle erhält. Diese Referenz kann erstellt werden, da der Port, der diese Schnittstelle anbietet, nur eben diese eine Schnittstelle besitzt. Somit wird davon ausgegangen, dass der Port, welcher die Funktionalität benötigt, diese Schnittstelle meint. Durch diese Umsetzung des Kompositionskonnektors bleibt die Kapselung des Ports durch die Schnittstelle weiter erhalten.

3.2. Anforderungen und Standardwerte der Diagramme

Die Diagramme, die als Grundlage der Codegenerierung genutzt werden, müssen, wie im vorigen Kapitel teilweise bereits erwähnt, einige Anforderungen erfüllen, damit der Code fehlerfrei und problemlos generiert werden kann. Es ist darauf geachtet worden, möglichst wenige Anforderungen zu entwickeln, da diese die Modellierung des Diagramms einschränken.

1. **Ports**

D-1.1 Alle Ports müssen mit einem Namen versehen sein.

D-1.2 Alle Ports müssen mit einem Typ versehen sein.

D-1.3 Der Standardwert der Multiplizität eines Ports beträgt „1“.

2. Komponenten

D-2.1 Alle Komponenten müssen mit einem Namen versehen sein.

D-2.2 Der Name einer Komponente muss über die Grenzen eines einzelnen Eingabedokuments hinweg eindeutig sein.

3. Klassen

D-3.1 Alle Klassen müssen mit einem Namen versehen sein.

D-3.2 Der Name einer Klasse muss über die Grenzen eines einzelnen Eingabedokuments hinweg eindeutig sein.

4. Datentypen

D-4.1 Alle Datentypen müssen mit einem Namen versehen sein.

D-4.2 Der Name eines Datentyps muss über die Grenzen eines einzelnen Eingabedokuments hinweg eindeutig sein.

5. Enumerationen

D-5.1 Alle Enumerationen müssen mit einem Namen versehen sein.

D-5.2 Der Name einer Enumeration muss über die Grenzen eines einzelnen Eingabedokuments hinweg eindeutig sein.

6. Parts

D-6.1 Alle Parts müssen mit einem Namen versehen sein.

D-6.2 Alle Parts müssen mit einem Typ versehen sein.

D-6.3 Der Typ eines Parts kann eine Klasse, ein Datentyp oder eine Enumeration sein.

D-6.4 Der Standardwert der Multiplizität eines Parts beträgt „1“.

7. Schnittstellen

D-7.1 Alle Schnittstellen müssen mit einem Namen versehen sein.

D-7.2 Alle Schnittstellen müssen über einen Port an Komponenten angebunden sein. Schnittstellen zwischen Parts müssen nicht, können aber durch Ports gekapselt sein.

Die Anforderungen, welche den Namen der Elemente vorschreiben (D-1.1, D-2.1, D-3.1, D-4.1, D-5.1 und D-6.1), begründen sich dadurch, dass ohne diese Namen keine Referenzen im Code gesetzt werden können. Es wäre zwar möglich, künstliche Namen zu erzeugen, allerdings wären diese dann aber wohl nicht mehr sonderlich lesbar. Da der Codegenerator aber nur Klassenrümpfe erstellen kann, muss der generierte Code von den Entwicklern noch manuell angepasst werden. Um für diese die Lesbarkeit und Verständlichkeit des Codes zu gewährleisten, müssen Namen angegeben werden.

Die Eindeutigkeit der Namen von Klassen, Komponenten, Datentypen und Enumerationen (D-2.2, D-3.2, D-4.2 und D-5.2) wurde bereits in Komponente erklärt. Ohne eindeutige Namen wäre es nicht möglich, eine einzelnes Element durch mehrere Diagramme zu beschreiben. Die Referenzen auf gleiche Elemente in verschiedenen Dokumenten geschehen über den Namen des Elements. Dies folgt aus der Tatsache, dass der Enterprise Architect für jedes Element eines Modells zwar auch eine Id generiert, diese aber nur innerhalb des Modells eindeutig ist. Sobald in einem zweiten Modell dasselbe Element beschrieben werden soll, erhält dieses Element eine andere Id und ist demnach nicht mehr dem Element im anderen Modell zuordbar.

Die Anforderung D-6.2 und D-1.2 ergeben sich, da Parts und Ports als Referenzen im Code wieder gespiegelt werden. Um diese Referenzen zu erstellen, ist das Vorhandensein eines Typs unumgänglich.

4. Analyse und Entwurf des Codegenerators

Dieses Kapitel beschäftigt sich mit der Analyse und dem Entwurf des Codegenerators. Es werden zunächst die Anwendungsfälle und Anforderungen an den Codegenerator entwickelt. Darauf aufbauend wird eine Architektur entworfen und die technischen Entwurfsentscheidungen erläutert.

Der zu entwickelnde Codegenerator soll Entwickler bei ihrer täglichen Arbeit unterstützen. Es wird von folgendem Szenario als Basis für die Entwicklung des Codegenerators ausgegangen. Es wird angenommen, dass ein Entwickler die Modelle einer komponentenorientierten Architektur von einem Softwarearchitekten erhält. Diese Architektur wurde mit Hilfe des Enterprise Architects 8.0 erstellt und in XML-Dokumente exportiert. Der Entwickler startet die Codegeneratoranwendung und gibt die Dokumente, welche Komponenten- und Kompositionsstrukturdiagramme beschreiben, als Eingabedaten an. Für die Ausgabe muss der Entwickler einen Pfad, der das Basisverzeichnis für die zu generierenden Dateien ist, sowie die Programmiersprache, in der der Code generiert werden soll, angeben. Die Anwendung generiert die Klassenrümpfe und deren Beziehungen zueinander und legt diese in einer Ordnerstruktur, die die Komponenten wieder spiegelt, ab. Die erzeugten Dateien fügt der Entwickler dann manuell einem Projekt der generierten Programmiersprache hinzu. Danach kann der Entwickler das Projekt weiter ausbauen und ergänzen. Optional kann es natürlich auch sein, dass der Architekt die Codegenerierung durchführt und erst die erzeugten Dateien zur manuellen Vervollständigung an den Entwickler gibt.

4.1. Anwendungsfälle

Die Anwendungsfälle des Codegenerators wurden unter Zuhilfenahme des Fallbeispiels entwickelt. Auch das kurze Szenario im Einleitungsabschnitt dieses Kapitels wurde verwendet, um einen typischen Ablauf des Programms zu entwerfen. Da in beiden Anwendungsfällen die selben Akteure vorkommen, sollen diese zunächst kurz beschrieben werden.

Anwender Ein Anwender ist der menschliche Benutzer des Systems, der dieses über eine Benutzeroberfläche bedienen kann. Der Anwender ist typischerweise ein Entwickler oder ein Architekt.

System Das System ist die Codegenerator-Anwendung. Von den Aktionen und Tätigkeiten des Systems, die im folgenden beschrieben werden, bekommt der Anwender in der Regel nichts mit.

Codegenerierung vorbereiten

Der Anwendungsfall „Codegenerierung vorbereiten“ beschreibt die Tätigkeiten, die der Anwender des Codegenerators sowie das System vornehmen müssen, um die Codegenerierung zu starten.

Der Anwender muss die Ausgangsdaten, welche in die Codegenerierung einfließen sollen, den Pfad zu dem Ordner, in den die generierten Dateien gespeichert werden sollen, sowie die zu generierende Programmiersprache angeben. Das System prüft diese Angaben auf Vollständigkeit. So dürfen nur XML-Dateien als Eingabe dienen. Sobald eine der Angaben des Anwenders ungültig ist oder fehlt und der Anwender versucht, die Codegenerierung zu starten, wird er darüber informiert, das noch Daten fehlen und dazu aufgefordert, diese anzugeben.

Die Prüfung der Eingabedaten umfasst an dieser Stelle nur eine Prüfung des Dateityps. Eine tiefergehende Prüfung, die auch semantische Fehler in den Dateien sucht, findet erst im Verlauf der Codegenerierung selbst statt. Vor solchen Fehlern wird der Anwender an dieser Stelle also nicht gewarnt.

Codegenerierung vorbereiten	
Akteur	Anwender, System
Ziel	Anwendungsfall „Code erzeugen“ wird gestartet.
Auslöser	Anwender möchte aus einem Komponenten- und Kompositionsstrukturdiagramm Code erzeugen.
Vorbedingungen	Die Codegeneratoranwendung ist gestartet. Der Anwender hat die Dateien des Modells/ der Modelle in Enterprise Architect 8.0 als XML exportiert.
Nachbedingungen	Der Anwendungsfall „Code generieren“ ist erfolgreich gestartet.

Tabelle 4.1 – Fortsetzung auf der nächsten Seite

Tabelle 4.1 – Fortsetzung

Codegenerierung vorbereiten	
Erfolgsszenario	1. Der Anwender wählt die XML-Datei des Modells. 2. Der Anwender wählt den Ordner, der als Wurzelordner für die erzeugten Codedateien dienen soll. 3. Der Anwender wählt die zu generierende Programmiersprache aus. 4. Der Anwender startet die Codegenerierung. 5. Das System prüft, ob alle erforderlichen Eingabedaten des Anwenders vorhanden sind. 6. Das System startet die Codegenerierung.
Erweiterungen	1a., 2a. und 3a: Der Anwender gibt die Eingabedateien, den Ausgabeordner und die Programmiersprache in anderer Reihenfolge an. 1b.: Der Anwender wiederholt diesen Schritt, da mehrere XML-Dateien vorliegen.
Fehlerfälle	1a.: Die ausgewählten Dateien haben keine .xml-Endung: Der Anwender wird darüber informiert und zur erneuten Eingabe aufgefordert (zurück zu 1). 5a.: Der Anwender hat den Pfad zum Ausgabeordner nicht angegeben: Der Anwender wird darüber informiert und zur Eingabe aufgefordert, das System startet die Codegenerierung nicht (zurück zu 2). 5b.: Der Anwender hat keine Eingabedateien angegeben: Der Anwender wird darüber informiert und zur Eingabe aufgefordert, das System startet die Codegenerierung nicht (zurück zu 1).

Tabelle 4.1.: Anwendungsfall Codegenerierung vorbereiten

Code generieren

Der Anwendungsfall „Code generieren“ beschreibt, wie das System vorgeht, um den Code zu erzeugen. Dies geschieht unter der Voraussetzung, dass der Anwendungsfall „Codegenerierung vorbereiten“ erfolgreich ausgeführt wurde und somit alle Voraussetzungen geschaffen sind, um diesen Anwendungsfall erfolgreich zu durchlaufen.

Das System liest zunächst die angegebenen XML-Dateien ein und erzeugt eine interne Repräsentation des Modells. Diese wird an einigen Stellen für die weitere Verarbeitung optimiert. Nachfolgend wird das Modell semantisch auf die in Abschnitt 3.2 beschriebenen Anforderungen geprüft. Wenn diese Validierung positiv erfolgt ist, wird der Code generiert und in dem angegebenen Ausgabeordner gespeichert.

Wenn die Validierung Fehler aufzeigt, wird der Anwender vom System darüber informiert und die Codegenerierung wird vorzeitig abgebrochen. Ein Abbruch der Codegenerierung kann auch bereits während der Phase des Parsens der XML-Dateien passieren, wenn die Dateien Elemente enthalten, die der Codegenerator nicht unterstützt. Auch in diesem Fall wird der Anwender darüber informiert, sodass er die Möglichkeit hat, die Modelle den Anforderungen entsprechend anzupassen.

Code generieren	
Akteur	System, Anwender
Ziel	Es wurden erfolgreich Codedateien in der angegebenen Programmiersprache passend zu den angegebenen Modellen erzeugt und gespeichert.
Auslöser	Der Anwender hat die Codegenerierung gestartet.
Vorbedingungen	Der Anwender hat die Eingabedateien, den Ausgabeordner und die Programmiersprache angegeben.
Nachbedingungen	Es liegen Codedateien der angegebenen Programmiersprache, die den Modellen entsprechen, an dem angegebenen Ort im Dateisystem.
Erfolgsszenario	1. Das System liest die Eingabedateien ein. 2. Das System parst die Eingabedateien. 3. Das System validiert die Eingabedateien in Bezug auf die Anforderungen an die Diagramme. 4. Das System generiert die Ausgabedateien. 5. Das System speichert die erzeugten Dateien in den angegebenen Zielordner. 6. Der Anwender erhält vom System die Rückmeldung, dass die Codegenerierung erfolgreich war.
Erweiterungen	

Tabelle 4.2 – Fortsetzung auf der nächsten Seite

Tabelle 4.2 – Fortsetzung

Code generieren	
Fehlerfälle	2a.: Das Parsen schlägt fehl: Die Codegenerierung wird abgebrochen und der Anwender darüber informiert. 3a.: Die Validierung schlägt fehl: Die Codegenerierung wird abgebrochen und der Anwender darüber informiert.

Tabelle 4.2.: Anwendungsfall Code generieren

4.2. Anforderungen an den Codegenerator

Die Anforderungen an den Codegenerator wurden aus den Anwendungsfällen entwickelt. Die vollständige Liste der Anforderungen ist in Anhang C zu finden. An dieser Stelle sollen nur einige wichtige Anforderungen erläutert werden.

Die Anforderung A-1.1 beschreibt den grundlegenden Aufbau der Anwendung, welcher teilweise bereits aus dem Anwendungsfall „Code generieren“ erkennbar ist.

A-1.1 Die Codegenerierung wird in vier Schritte eingeteilt:

- Parsen
- Optimieren
- Validieren
- Generieren

Auch die Anforderungen A-2.1 bis A-2.6 bezüglich der Benutzerschnittstelle sind aus den Anwendungsfällen erschließbar. Der Anwendungsfall „Codegenerierung vorbereiten“ beschreibt verschiedene Eingaben, die der Anwender zu erledigen hat, bevor die eigentliche Codegenerierung gestartet werden kann. Da die Bedienung einer Anwendung für die meisten Anwender über eine graphische Benutzeroberfläche am komfortabelsten ist, wird diese durch die genannten Anforderungen funktional spezifiziert.

Die Validierung der Eingabedaten wird in den Anforderungen A-3.1 und A-3.2 spezifiziert. Da die genauen Validierungspunkte bereits in Kapitel 3 spezifiziert wurden, wird an dieser Stelle nur auf diese Anforderungsliste verwiesen, anstatt dass die Anforderungen nochmal an dieser Stelle spezifiziert werden.

Aus einem ähnlichen Grund wurde darauf verzichtet, die genaue Übersetzung der Diagramme in Code zu spezifizieren. In Kapitel 3 wurde diese Übersetzung bereits ausführlich dargestellt. Nach dieser Analyse wird der Generator vorgehen und den Code generieren.

Die Anzahl an Eingabedateien in der nichtfunktionalen Anforderung A-6.3 wurde auf Grund fehlender Erfahrungswerte willkürlich gewählt. Da eine Beschränkung der garantiert möglichen Anzahl von Eingaben für spätere Tests nötig ist, wurde diese Anforderung dennoch aufgestellt:

A-6.3 Das System muss mindestens 5 Eingabedateien in einem Codegenerierungsprozess verarbeiten können.

4.3. Architektur

Die Architektur des Codegenerators wurde aus den Anforderungen und den Anwendungsfällen entwickelt. Besonders aus dem Erfolgsszenario des Anwendungsfalles „Code generieren“ sind vier Komponenten zur Kapselung der eigentlichen Logik des Codegenerators erkennbar. Insgesamt werden die zehn folgenden Komponenten entworfen.

- BaG
  - Parser
  - Optimierer
  - Validierer
  - Generator
    - Templates
  - Modellbaum
  - Ergebnisdatentypen
  - Exceptions
- Benutzeroberfläche

Die Zusammenhänge zwischen den Komponenten sind in Abbildung 4.1 zu sehen.

Diese Architektur ist nach dem Repository-Stil (Buschmann u. a. (1996)), einer Variation des Blackboard-Stils, aufgebaut. Taylor u. a. (2010) schlägt vor, den Blackboard-Stil bei Problemen anzuwenden, die auf einer zentralen, sich ändernden Datenquelle beruhen. Die zentrale Datenquelle ist in diesem Fall der *Modellbaum*, die Subkomponenten *Parser*, *Optimierer*,

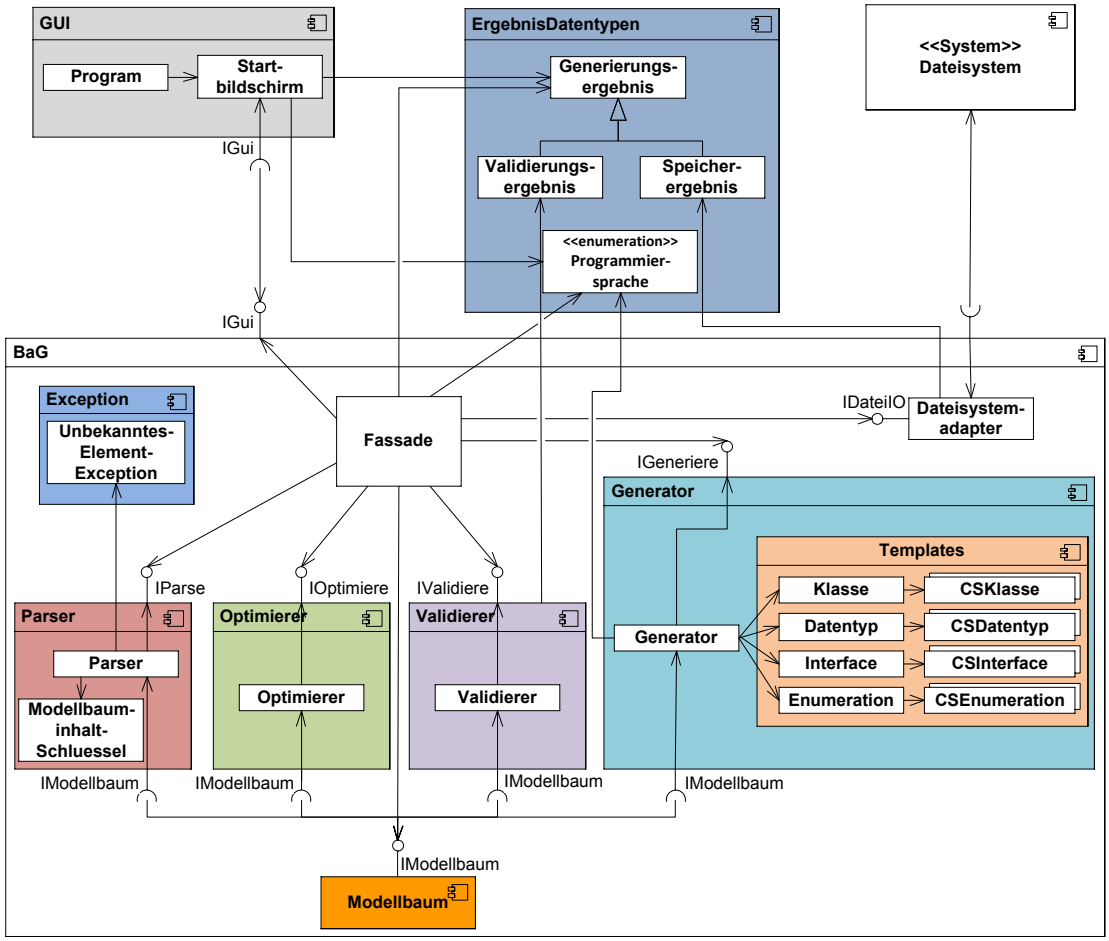


Abbildung 4.1.: Komponentenzusammensetzung des Codegenerators

Validierer und *Generator* sind die unabhängigen Komponenten, welche die Datenquelle verwenden und ändern. Beim Repository-Stil wird die Auswahl derjenigen Subkomponente, die als nächstes auf die Datenquelle zugreifen darf, durch eine weitere externe Instanz gesteuert. Beim reinen Blackboard-Stil würde die Auswahl durch den Zustand der Daten in der Datenquelle selbst geschehen. Die externe Auswahlinstanz ist hier im Fall des Codegenerators die Klasse *Fassade* der BaG-Komponente. Die *Fassade* erhält die Eingabedaten der Benutzerschnittstelle und koordiniert den Aufruf der Subkomponenten *Parser*, *Optimierer*, *Validierer* und *Generator*.

Der Repository-Architekturstil hat den Vorteil, die Subkomponenten voneinander zu trennen aber dennoch gemeinsame Daten allen Komponenten zur Verfügung zu stellen. Des Weiteren sind die Subkomponenten nur lose gekoppelt und können somit leicht einzeln ausgetauscht oder erweitert werden.

Zudem wird in Buschmann u. a. (1996) vorgeschlagen, diesen Architekturstil für Compiler zu verwenden. Der hier entworfene Codegenerator hat eine sehr ähnlich Aufgabe und Struktur wie ein Compiler. Darum lässt sich dessen Architektur auch auf diesen Codegenerator anwenden.

4.3.1. BaG

Die Komponente BaG bildet den Kern der Anwendung und kapselt die Anwendungslogik von der Benutzerschnittstelle. In der BaG-Komponente sind weitere Komponenten gekapselt, welche die eigentliche Funktionalität des Codegenerators bereit stellen.

Die BaG-Komponente enthält eine Klasse *Fassade*, die den Vorteil des oben beschriebenen Repository-Architekutrstiles noch verstärkt. Ihre Position im Gesamtsystem als eine Art Koordinator und Schnittstelle zur Benutzeroberfläche sowie zum Dateisystemadapter ist an das Entwurfsmuster „Fassade“ nach Gamma u. a. (1995) angelehnt. Durch die Nutzung dieses Musters sind zum einen die Benutzerschnittstelle und die Logik der Anwendung von einander getrennt und können separat ausgetauscht werden. Zum anderen bietet die Fassade die Koordination der Subkomponenten an, sodass auch diese nur lose gekoppelt sind und unabhängig voneinander agieren können. Dadurch wird die Austauschbarkeit und Erweiterbarkeit des gesamten Systems gewährleistet und Anforderung A-6.1 somit erfüllt. Es kann beispielsweise die Parserkomponente ausgetauscht werden, etwa weil ein neues Eingabeformat zur Verfügung steht. Solange dieser neue Parser die Schnittstelle *IParse* implementiert, können die anderen Subkomponenten problemlos weiter genutzt werden.

Das Sequenzdiagramm in Abbildung 4.2 verdeutlicht nochmals die Aufgabe der Fassade. Sie nimmt die Eingabedaten der Benutzeroberfläche an und reicht sie nacheinander an die Subkomponenten weiter.

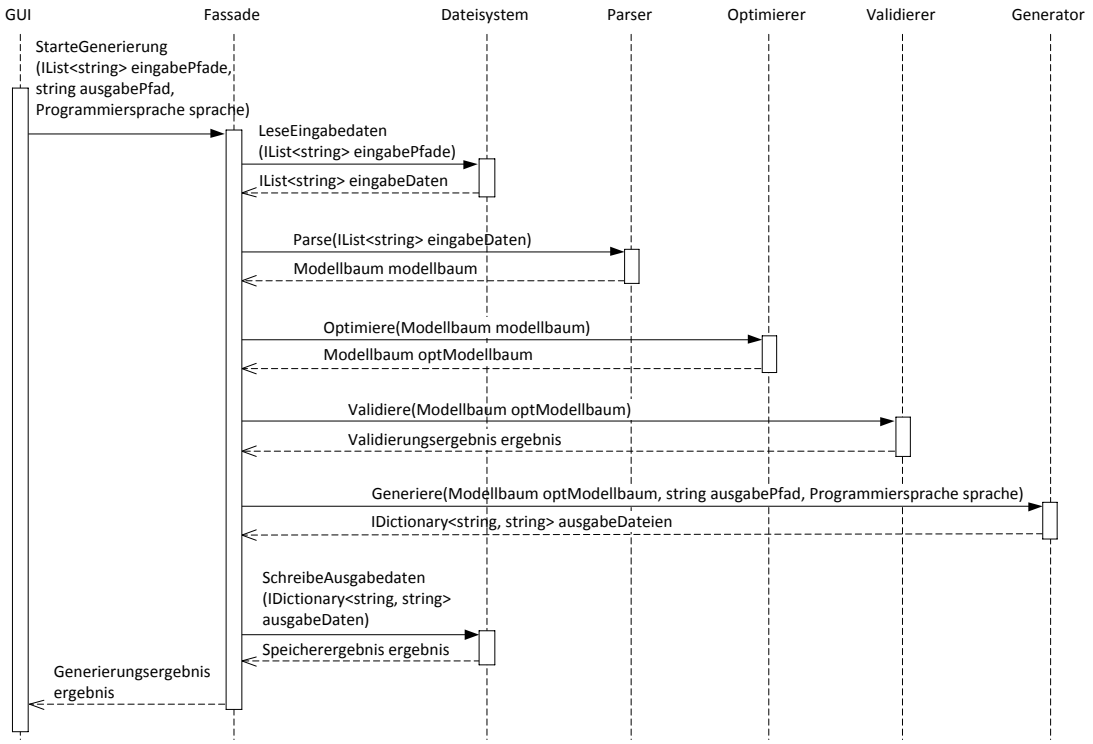


Abbildung 4.2.: Sequenzdiagramm der Methodenaufrufe der Interfaces

Die Klasse *Datenspeicheradapter* bildet die Schnittstelle zum Dateisystem. Sie übernimmt die Aufgaben des Einlesens und Schreibens der Ein- bzw. Ausgabedaten. Da die BaG-Komponente, wie auch graphisch in Abbildung 4.1 erkennbar ist, eine Art Rahmen um die internen Komponenten des Codegenerators legt, ist diese Funktionalität nicht als eigenständige äußere Komponente gekapselt. Alle Komponenten, die nicht unbedingt selbstständig von der Benutzerschnittstelle angesprochen werden müssen, sollten durch die BaG-Komponente und insbesondere die *Fassade* gekapselt werden. Der Datenspeicheradapter würde als Komponente außerhalb der BaG-Komponente liegen, da er eine Schnittstelle an das, den Codegenerator umgebende, System bietet. Diese Art von Schnittstellen an Nachbarsysteme sollen durch die BaG-Komponente umgesetzt werden, um einen einheitlichen funktionalen Rahmen zu bilden.

4.3.2. Modellbaum

Der Modellbaum bildet die zentrale Datenstruktur des Codegenerators, mit welcher die weiteren Komponenten arbeiten, um Code zu generieren. Die Modellbaumkomponente ist in dem Klassendiagramm in Abbildung 4.3 detailliert aufgezeigt. Neue Instanzen der einzelnen Knoten können über das Interface *IModellbaum* durch Factorymethoden erzeugt werden. Für jeden Knoten des Modellbaums leitet das Interface bei der Erzeugung eines neuen Knoten den Aufruf an das Interface dieses zu erstellenden Knotens weiter.
Die jeweiligen Interfaces der Knoten kapseln diesen nach außen hin ab. Dies ist wichtig, da unter Umständen bei einer Erweiterung des Codegenerators auch die Datenstruktur geändert werden muss. Somit wäre es möglich, einzelne Knoten zu erweitern ohne an ihrer äußeren Repräsentation etwas zu verändern. Die anderen Komponenten des Codegenerators könnten dann noch immer fehlerfrei arbeiten und würden nicht beeinträchtigt.

4.3.3. Parser

Der Parser erhält die XML-Eingabedaten in Form von Strings von der *Fassade* und wandelt sie in den Modellbaum um. Dabei passiert an dieser Stelle nur eine rudimentäre Validierung. Der Parser bricht seine Arbeit ab, wenn er in den XML-Dateien Elemente findet, die nicht vom Codegenerator unterstützt werden. In so einem Fall wirft der Parser eine *UnbekanntesElementException* und beendet seine Arbeit. Die Fassade muss diese Exception abfangen und eine sinnvolle Fehlermeldung an die Benutzerschnittstelle übergeben. Es ist darauf geachtet worden, dass die Anwendung an dieser Stelle nicht beendet wird, sondern der Anwender die Möglichkeit hat, seinen Fehler zu korrigieren und die Codegenerierung erneut zu starten.

Um während des Parsens die Referenzen zwischen gleichen Elementen in verschiedenen Modellen herstellen zu können, führt der Parser eine Art Modellbaum-Inhaltsverzeichnis mit.

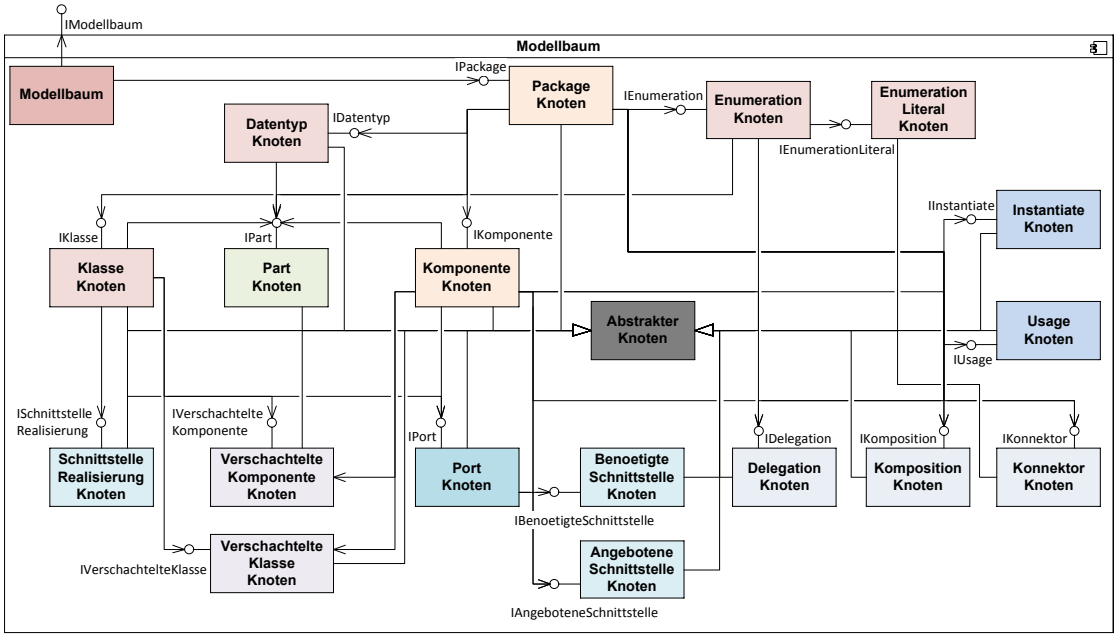


Abbildung 4.3.: Modellbaumkomponente

Der Schlüssel eines Elements in diesem Verzeichnis besteht aus einem *Modellbauminhalts-Schlüssel*.

4.3.4. Optimierer

Der Optimierer erhält den vom Parser eingelesenen Modellbaum und hat die Aufgabe, ihn zu optimieren. Das Optimieren umfasst an dieser Stelle das Austauschen von Referenzen durch konkrete Namen. Der Enterprise Architect erzeugt in jedem Modell Ids für die verschiedenen Elemente. Bei Verknüpfungen zwischen den Elementen, beispielsweise Konnektoren, wird nicht der Name sondern eben diese Id gespeichert. Im Modellbaum stehen nun alle Verknüpfungen aus sämtlichen Modellen in einer gemeinsamen Datenstruktur. Es ist die Aufgabe des Optimierers, die Ids den jeweiligen Elementen zuzuordnen und statt der Id den Namen des Elements einzutragen. Dadurch können die Validierungs- und die Generatorkomponente später beim Verarbeiten einer Verknüpfung alle relevanten Daten direkt aus der Verknüpfung lesen und müssen nicht erst noch Referenzen finden und setzen.

4.3.5. Validierer

Der Validierer erhält den vom Optimierer bearbeiteten Modellbaum und überprüft ihn auf semantische Fehler. Diese Fehler werden anhand der Anforderungen in Abschnitt 3.2 festgestellt. Die aufgeführten Anforderungen müssen alle erfüllt sein, damit die Codegenerierung starten kann. Andernfalls gibt der Validierer eine Meldung mit dem Problem an die Fassade zurück. Diese sorgt dann dafür, dass die Codegenerierung abgebrochen wird und der Anwender über das Problem informiert wird.

4.3.6. Generator und Templates

Diese Komponenten sind für die eigentliche Codegenerierung zuständig. Die Templatekomponente enthält die konkreten Templates aller angebotenen Programmiersprachen sowie pro Templatetyp eine Oberklasse. Der Templatetyp kann *Klasse*, *Interface*, *Datentyp* oder *Enumeration* sein. Die Generatorkomponente erhält über die Schnittstelle *IGeneriere* von der Fassade den validierten Modellbaum, die gewünschte Programmiersprache sowie den Pfad zu dem Basisausgabeordner. Aus dem Modellbaum werden in der Klasse *Generator* die Informationen der einzelnen Knoten ausgelesen und zusammen mit der Programmiersprache an einem den extrahierten Informationen entsprechende Oberklasse der Templatekomponente übergeben. Diese wählt dann je nach Programmiersprache das passende Template ihres Typs aus und generiert den Code. Die Generatorkomponente erzeugt aus dem übergebenen Basisausgabepfad und den Informationen aus dem Modellbaum eine passende Ordnerstruktur und ordnet den von den Templates generierten Code einem Dateipfad zu. Diese Pfad-Datei-Tupel werden an die Fassade zurück gegeben.

4.3.7. Benutzerschnittstelle

Über die Benutzerschnittstelle kann der Anwender die Anwendung bedienen und die benötigten Daten angeben (vgl. Anforderung A-2.1, A-2.2 und A-2.4). In der Benutzerschnittstelle wird lediglich geprüft, ob alle Eingabedaten angegegeben wurden, danach werden diese dann direkt an die Fassade weiter geleitet, welche die weitere Prüfung und Verarbeitung übernimmt. Die Benutzerschnittstelle bietet also über die bloße Darstellung und Weiterleitung von Informationen keine Funktionalität an.

4.3.8. Exception

Der Parser verwendet eine *UnbekanntesElementException*, welche geworfen wird, wenn in den Eingabedaten XML-Elemente stehen, die er nicht verarbeiten kann, weil diese nicht vom Codegenerator unterstützt und damit eingelesen werden können. Diese Exception ist in einer extra Komponente gekapselt, damit bei Bedarf weitere Exceptions an einem zentralen Ort definiert werden können. Momentan verwenden nur der Parser und die Klasse *Fassade* diese Exception. Die Exception soll nicht die ganze Anwendung zum Absturz bringen. Vielmehr soll die Fassade die Nachricht auslesen und an die Benutzerschnittstelle weiter leiten, um den Anwender darüber zu informieren und einen ordentlichen Abbruch des Codegenerierungsprozesses zu veranlassen.

4.3.9. Ergebnisdatentypen

Um aussagekräftige Nachrichten über die Codegenerierung an den Anwender übergeben zu können, wurde die Komponente *Ergebnisdatentypen* entworfen. Diese enthält verschiedene Datentypen, die Nachrichten und den booleschen Wert des Gesamtausgangs der Codegenerierung speichern und transportieren können. So gibt die Validierungskomponente ein *Validierungsergebnis* zurück. Wenn die Validierung fehl schlägt, werden in dem *Validierungsergebnis* die genaue Problemnachricht und ein *false* gespeichert. Die Fassade stellt an Hand des booleschen Wertes fest, dass die Validierung nicht korrekt war und kann die Nachricht der Validierungskomponente an die Benutzeroberfläche weiterleiten. Analog verfährt die Klasse *Datenspeicheradapter* mit dem *Speicherergebnis* bei der Speicherung der erzeugten Dateien.

Neben diesen Datentypen enthält diese Komponente eine Enumeration *Programmiersprachen*, die eine Liste aller vom Generator unterstützten Programmiersprachen darstellt. Diese Enumeration wird in der Benutzeroberfläche, der *Fassade* und den Komponenten *Generator* und *Templates* verwendet, um die Programmiersprache eindeutig zu referenzieren.

4.4. Entwurfsentscheidungen

Nachfolgend werden einige Entscheidungen, die vor dem Beginn der Implementierung des Codegenerators getroffen werden müssen, begründet.

Die Entscheidung für die UML-Modellierungsanwendung Enterprise Architect 8.0 als Exportanwendung für die XML-Dateien, die die Eingabe des zu entwickelnden Codegenerators bilden, wurde auf Grund mehrerer Faktoren getroffen. Es musste zunächst eine Anwendung

ausgewählt werden, die den XMI-Export von UML-Modellen unterstützt. Des Weiteren musste diese Anwendung zur Verfügung stehen. Diese beiden Faktoren trafen sowohl auf Enterprise Architect 8.0 als auch auf Visual Paradigm for UML 8.1 (VisualParadigm (2011)) zu. Der Enterprise Architect wurde schließlich auf Grund von Erfahrungswerten mit der Anwendung seitens der Autorin gewählt.

Der Codegenerator wird in Visual C# 2010 implementiert. Diese Entscheidung begründet sich vor allem aus dem Vorhandensein einer eingebauten Template Engine (vgl. Abschnitt 2.3), welche zur Codegenerierung gut geeignet ist.

Die Eingabedaten liegen im XML-Format vor, darum muss dieses eingelesen und in eine Struktur gebracht werden, mit deren Hilfe dann der Code generiert werden kann. Das Einlesen des XML soll mit Hilfe der von der .NET-Plattform bereitgestellten API *Linq-to-XML* geschehen. Diese bietet die für diese Arbeit relevanten Fähigkeiten zum Iterieren durch einen XML-Baum und Auslesen von Knoten und Attributen.

4.4.1. Templatesprache

Zur Codegenerierung gibt es mehrere Möglichkeiten. Vier mögliche Ansätze wurden bereits in Abschnitt 2.3 beschrieben. Diese Ansätze haben alle Vor- und Nachteile, die nachfolgend verglichen werden, sodass am Ende eine Entscheidung über den optimalsten Ansatz erfolgen kann.

Da Anforderung A-6.1 des Codegenerators besagt, dass er wartbar und erweiterbar sein soll, wurde dieser Aspekt bei der Wahl des Generierungsansatzes besonders beachtet. Daraus schließt sich, dass die Codegenerierung durch eine Methode geschehen sollte, die leicht zu erlernen und anzuwenden ist. Der Code, der geschrieben wird, sollte übersichtlich und gut lesbar sein. Außerdem sollte die Methode so einsetzbar sein, dass eine extra Komponente für die Templates angelegt werden kann. Diese Trennung von der Logik der Generierung und den Templates selbst unterstützt die Erweiterung des Generators um Templates für andere Programmiersprachen.

Um eine spätere Änderung des Codegenerators möglich zu machen, wird außerdem darauf geachtet, wie aktuell die Generierungsansätze sind und wie hoch die Wahrscheinlichkeit ist, dass sie weiter gepflegt und entwickelt werden.

Konkatenation von Strings in C#

Der erste in Abschnitt 2.3 genannte Ansatz ist die Codegenerierung durch Stringkonkatenationen in C#. Dieser Ansatz hat den Vorteil, dass der Entwickler keine neue Templatesprache

lernen muss, um die Templates zu erzeugen und zu ändern. Auch muss keine zusätzliche Bibliothek in die Anwendung eingebunden oder eine Erweiterung installiert werden.

Dieses Verfahren birgt aber einige nachfolgend beschriebenen Probleme, die in Templatesprachen bereits gelöst wurden und ist damit ungeeignet für diese Aufgabe. Die Probleme liegen in der Aufrufreihenfolge von Methoden zur Konstruktion von verschiedenen Bestandteilen eines typischen Codedokuments. Das beschriebene Beispiel in Abschnitt 2.3 besteht jeweils aus einer Methode zur Generierung öffnender Codeelemente sowie einer Methode zur Generierung der entsprechenden schließenden Elemente. Es müssen also, nur um beispielsweise einen Klassenrumpf zu erstellen, bereits zwei Methoden aufgerufen werden. Der Aufruf der Methoden wird schnell unübersichtlich. Der Entwickler der Templates darf nicht vergessen, die schließende Klammer zu generieren, sonst ist der erzeugte Code fehlerhaft. Sobald die zu generierenden Dateien ein wenig komplexer werden und damit mehrere Methoden aufgerufen werden müssen, ist dies eine große Fehlerquelle.

Eine Lösung des Problems wäre der verschachtelte Aufruf der Methoden. Beispielsweise wird innerhalb einer Methode, die den Namensraum sowohl mit öffnendem als auch mit schließendem Teil generiert, die Methode zur Generierung des Klassenrumpfes aufgerufen (Listing 2.4). Auch dieser Ansatz ist aber sehr unflexibel und schlecht wartbar. Sobald eine Datei beispielsweise eine Enumeration statt einer Klasse enthalten soll, muss die Namensraummethode entweder erweitert oder dupliziert werden, was zu unnötiger Redundanz führen würde. Von diesem Problem abgesehen, muss der Entwickler, der den Generator anpassen möchte, wissen, welche Verschachtelungen existieren, damit er den richtigen Ansatzpunkt für die Anpassungen findet. Die Methoden zur Generierung einzelner Bausteine wären also nicht mehr unabhängig voneinander aufrufbar und würden große Abhängigkeiten untereinander aufweisen.

Die Nutzung dieses Ansatzes ist auch aus dem Grund ungeeignet, als dass dieser Ansatz in der eben beschriebenen Form kaum die Unterstützung von mehreren Programmiersprachen fördert. Für jede Programmiersprache, in welcher der Code erzeugt werden soll, müssten die Methoden neu geschrieben werden.

Abgesehen von gut dokumentiertem Quellcode ist dem Entwickler bei diesem Ansatz keine Hilfe gegeben. Dies erhöht nochmals die Fehleranfälligkeit der Codegenerierung und trägt nicht zur Qualität des Ergebnisses bei.

Existierende Templatesprachen lösen bereits die geschilderten Probleme und werden darum eher in Betracht gezogen.

Text Template Transformation Toolkit

Das Text Template Transformation Toolkit (T4) ist bereits in Visual Studio 2010 intergriert. Darum ist keine Installation notwendig. Die Unterstützung von Intelli-Sense und Syntaxhighlighting, welche durch die Installation eines zusätzlichen Editors erreicht werden kann, erleichtert die Arbeit mit der Template-Engine.

Generell muss keine neue Syntax erlernt werden, da der Kontrollcode innerhalb der Templates in C# oder Visual Basic geschrieben wird. Die Syntax wird lediglich um einige Tags und deren Bedeutung (vgl. Msdn (2011a) sowie Unterabschnitt 2.3.2) erweitert.

Ein Nachteil daran, dass C# als Templatesprache verwendet wird, ist, dass komplexer Code innerhalb der Templates schreibbar ist. Dies führt zu unübersichtlichen Templates und es liegt am Entwickler, sich dementsprechend einzuschränken und auf möglichst wenig Code im Template zu achten. Die Anbindung an den Codegenerator erfolgt über eine partielle Klasse, die das Template, welches intern in eine .cs-Datei übersetzt wird, erweitert. Diese Anbindung ist einerseits komfortabel, da die Initialisierung der Codegenerierung leicht gemacht wird, andererseits müssen so zwei Dateien pro Template gepflegt werden (das Template selbst und die partielle Klasse).

Das T4 ist ausreichend unter Msdn (2011a) dokumentiert. Zusätzlich existiert weitere Dokumentation online. Hierbei ist der Blog Sych (2011) besonders empfehlenswert und informativ.

NVelocity

Obwohl das NVElocity-Projekt mittlerweile von den ursprünglichen Entwicklern bei der Apache Software Foundation losgelöst und von CastleProject übernommen wurde, ist die Aktualität und das weitere Bestehen des Projekts nicht sicher. Die letzten Releases stammen von Ende Dezember 2010. Die anderen hier beschriebenen Ansätze sind, soweit man dies sagen kann, aktueller. Auch die Community, welche bei einem solchen Projekt im Rahmen der Dokumentation eine große Rolle spielt, ist, der Webseite nach zu urteilen, unaktiv. Aus diesen Punkten ergibt sich, dass die Template-Engine ungeeignet für die Nutzung in dieser Arbeit ist. Es kann nicht sicher gestellt werden, dass die Unterstützung der Template-Engine und Dokumentation noch langfristig gegeben sind.

Dennoch sei hier erwähnt, dass die einfache und kompakte Syntax der Templatesprache vorteilhaft für den Codegenerator sind. So bleiben die Templates kurz und enthalten nur wenige Kontrollstrukturen.

StringTemplate

StringTemplate ist wie NVelocity ebenfalls eine Portierung von Java nach C#. Im Gegensatz dazu wird die Portierung allerdings noch aktiv entwickelt, wie man auf der Webseite StringTemplate (2011a) sehen kann.

Allerdings ist die Dokumentation der Templatesprache noch nicht ausreichend. Die Webseite bietet einige Informationen für die Entwicklung von StringTemplate in Java, die teilweise auch auf die Entwicklung in C# übertragen werden können. Es werden, wie man anhand der Daten in dem Wiki der Webseite sehen kann, noch kleine Änderungen vorgenommen (StringTemplate (2011b)). Dennoch wäre eine Ergänzung der Dokumentation sinnvoll.

Nachteilig an StringTemplate ist das Aufsetzen der Entwicklungsumgebung. Es müssen aus mehreren Bibliotheken die passenden ausgewählt und herunter geladen werden. Dazu muss der Entwickler zunächst wissen, welche Bibliotheken er benötigt. Das Aufsetzen der Entwicklungsumgebung ist damit bei StringTemplate komplizierter als bei den anderen hier verglichenen Ansätzen.

Zusammenfassung

Die nachfolgende Tabelle 4.3 fasst die im Text aufgezählten Vor- und Nachteile noch einmal zusammen und bewertet sie anhand einer einfachen Skala, die die Einträge zwischen einander vergleicht und nach ihrer Nützlichkeit für diese Arbeit bewertet. ⊕ ist dabei der beste Wert, ⊖ der Schlechteste.

Auch in dieser Tabelle erkennt man den Vorteil des Text Template Transformation Toolkits gegenüber den anderen Ansätzen. Der einzige nicht positive Punkt ist die Übersichtlichkeit des Codes. Wie oben beschrieben, kann diese dennoch durch Disziplin des Entwicklers erreicht werden.

Die Aktualität der Webseite kann bei der Stringkonkatenation nicht betrachtet werden, da es zu diesem Verfahren keine bekannte Webseite gibt.

Das NVelocity-Projekt wurde, wie oben beschrieben, bereits von einem anderen Entwicklungsteam übernommen, dennoch finden sich auch auf der neuen Webseite (NVelocity (2011)) nur wenige Informationen. Auf der Webseite von StringTemplate gibt es verschiedene aktuellere Einträge, viele davon allerdings auch zu der Java-Version von StringTemplate.

	Stringkonkatenation	T4	NVelocity	StringTemplate
Aktualität der Webseite		⊕	⊖	○
Dokumentation	⊖	⊕	○	○
Integration	⊕	⊕	⊖	⊖
Übersichtlichkeit des Codes	⊖	○	⊕	⊕
Neue Syntax	⊕	⊕	⊖	⊖

Tabelle 4.3.: Vergleich der verschiedenen Codegenerierungsansätze

5. Realisierung

Dieses Kapitel beschäftigt sich mit der Realisierung des Codegenerators und seinen Tests. Da im Verlauf dieser Arbeit nur ein Prototyp erstellt werden soll, wurde die in Kapitel 4 entworfene Architektur nicht vollständig umgesetzt, sondern teilweise gekürzt. Diese Abweichungen werden beschrieben und begründet. Außerdem wird die Implementierung der anderen Komponenten dargestellt, wie die Benutzeroberfläche umgesetzt wurde sowie welche Testkonzepte geplant und umgesetzt wurden.

5.1. Abweichungen der Implementierung

In diesem Abschnitt sollen die Abweichungen der Implementierung vom Entwurf in Kapitel 4 beschrieben werden. Diese Abweichungen wurden auf Grund des zu großen Umfangs des modellierten Codegenerators nötig.

Modellbaum

Es wurde darauf verzichtet, die Schnittstellen des Modellbaumes, die in Abbildung 4.3 dargestellt sind, explizit zu implementieren. In der prototypischen Implementierung greifen die Komponenten, die den Modellbaum verwenden, stattdessen direkt auf die Klassen der einzelnen Knoten zu. Diese tatsächliche Implementierung ist in Abbildung 5.1 zu sehen.

Um die Schnittstellenverwendung zu implementieren müsste pro Knoten des Modellbaumes ein Interface erzeugt werden, das eine Methode zur Erzeugung dieses Knotens bereitstellt, sowie Methoden, die den Zugriff auf die Felder des Knotens bereitstellen. Dessen Verwendung müsste in den anderen Komponenten, die momentan direkt auf den Knoten und seine Felder zugreifen, sicher gestellt werden.

Der Schwerpunkt, unter dem der Codegenerator entwickelt wurde, ist die Klärung der Frage, ob und wie es möglich ist, aus Komponenten- und Kompositionsstrukturdiagrammen Code zu generieren. Die Zeitersparnis brachte an dieser Stelle den Vorteil, sich stärker auf die Implementierung der Funktionen des Generators und weniger auf die korrekte Umsetzung

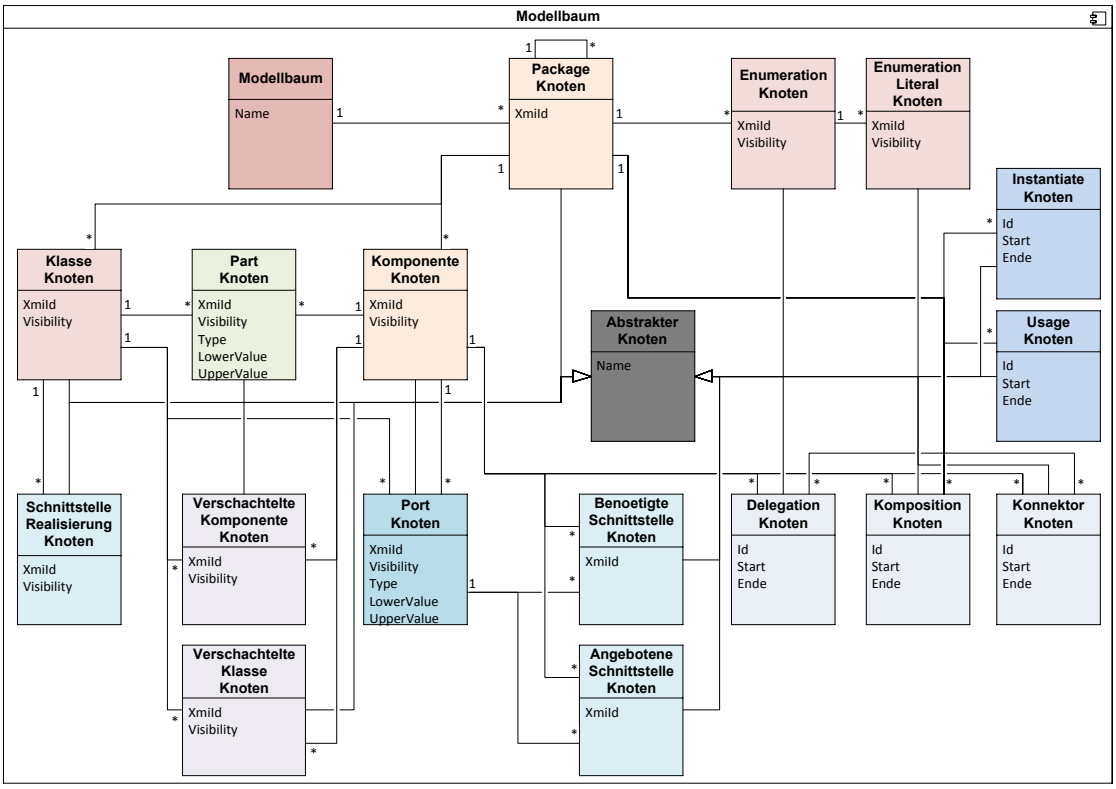


Abbildung 5.1.: Umgesetzte Modellbaumkomponente

der Architektur konzentrieren zu können. Die Funktionen des Generators helfen, die gestellte Frage zu beantworten und sind damit zu priorisieren.

Templates

Eine weitere Einschränkung des Prototypen gegenüber seiner Konzipierung besteht in der Anzahl unterstützter Programmiersprachen. Die Benutzeroberfläche bietet die Möglichkeit, neben C# auch Java als zu generierende Programmiersprache auszuwählen. Zu dieser Sprache wurden aber keine Templates erstellt. Darum kann momentan nur Code der Sprache C# generiert werden. Wenn ein Anwender die Sprache Java auswählt, so wird ihm direkt mitgeteilt, dass diese Sprache noch nicht unterstützt ist. Die Erstellung weiterer Templates für Java ist eine der möglichen Erweiterungen, die auf Grund der Ähnlichkeit zwischen der Syntax von Java und C# voraussichtlich einfach zu implementieren ist.

5.2. Implementierungsdetails

Die Umsetzung des Codegenerators erfolgte inkrementell. Zunächst wurde das Grundgerüst des Codegenerators mit der umfassenden BaG-Komponente und dem Modellbaum erstellt. Es folgte die Implementierung einer Benutzeroberfläche, die zwar rudimentär, aber dennoch voll funktionsfähig ist. Anschließend wurden nacheinander die Komponenten der eigentlichen Funktionalität des Codegenerators implementiert.

Nachfolgend sollen einige interessantere Implementierungsdetails vorgestellt werden. Da der Generator recht umfangreich ist, kann an dieser Stelle nicht auf jede Komponente im Einzelnen eingegangen werden.

Modellbaum

Der Modellbaum enthält Klassen, nachfolgend auch „Knoten“ genannt, welche die relevanten Daten der eingelesenen XML-Elemente speichern. Diese Klassen enthalten einige Felder für die eingelesenen Attribute sowie Listen, die die Kindelemente des aktuellen Knotens enthalten. Dadurch bildet sich ein Baum, welcher die Struktur der eingelesenen XML-Datei wiederspiegelt.

Da jeder Knoten, außer dem Wurzelknoten, auch eine Referenz auf seinen Elternknoten hält, ist der Baum bidirektional aufgebaut und kann in alle Richtungen durchlaufen werden. Diese Referenz auf das Elternteil wurde nötig, da beispielsweise mehrere Komponenten einen

Port mit dem Namen *Verwaltungsport* besitzen können (vgl. alle modellierten Komponenten des Fallbeispiels). Durch die Referenz auf den Elternknoten kann zu jedem Zeitpunkt erkannt werden, um welchen *Verwaltungsport* es sich handelt, indem sowohl der Name des aktuellen Knotens als auch der Name seines Elternknotens in der *equals()*-Methode des aktuellen Knotens verglichen werden können. Unterschiedliche *Verwaltungsports* besitzen unterschiedliche Komponenten als Elternteile und sind so voneinander zu unterscheiden.

Die Referenz auf den Elternknoten eines Knotens begründet auch die Verwendung einer abstrakten Oberklasse, die alle Knoten implementieren. Da einige Knotenarten, wie beispielsweise Portknoten, verschiedene Typen von Elternknoten haben können, wurde es notwendig, eine gemeinsame Oberklasse zu schaffen. Diese Oberklasse *AbstrakterKnoten* bietet nur das Feld „Name“, welches alle Knoten gemeinsam haben. Dieses Feld muss bei allen Knoten gesetzt sein, da sonst der Vergleich zweier Knoten durch die *equals()*-Methode fehl schlagen würde.

Parser

Der Parser durchläuft mit Hilfe der Linq-to-XML-API rekursiv die eingelesenen XML-Dateien und baut aus den eingelesenen Informationen den Modellbaum auf. Dazu wurde pro XML-Element eine private Methode implementiert, welche die relevanten Attribute des XML-Elements ausliest und in eine, dem XML-Element entsprechende, Instanz einer Knotenklasse des Modellbaums einfügt. Für die Kindelemente wird deren entsprechende Parsermethode aufgerufen, welche wiederum den entsprechenden Kindknoten erzeugt und in den Modellbaum einfügt.

Parallel wird des Weiteren ein Dictionary aufgebaut, welche eine Art Inhaltsverzeichnis für den Modellbaum darstellt. Dieses wurde nötig um die Daten aus mehreren Modellen in eine einzige Modellbaumstruktur zusammen fügen zu können. Dieses Inhaltsverzeichnis existiert nur in der Parserkomponente, da es in den anderen Komponenten nicht mehr benötigt wird. In diesen liegen alle Modelle bereits in einer einzigen Modellbaumstruktur vor. In dem Dictionary wird als Schlüssel ein Element des Typs *ModellbauminhaltSchluessel* und als Wert der entsprechende Knoten abgelegt. Der *ModellbauminhaltSchluessel* besteht aus dem Namen des jeweiligen Elements sowie dem Namen seines Elternknotens. Mit Hilfe des Inhaltsverzeichnisses kann bei doppelt auftretenden Elementen schnell überprüft werden, ob das aktuelle Element neu in den Modellbaum einzufügen ist oder bereits vorhanden ist und ergänzt werden muss. Es muss nicht durch den ganzen, bereits bestehenden, Baum iteriert werden.

Der Vorgang des Parsens ist beispielhaft in Listing 5.1 dargestellt. Das Beispiel zeigt die Methode *ParseEnumerationKnoten(. . .)*, welche für das Einlesen eines Enumerationselements zuständig ist. Zunächst werden in den Zeilen 3-5 die Attribute der Enumeration ausgelesen.

In den Zeilen 8-19 wird überprüft, ob es dieses Enumerationselement bereits als Knoten in dem Modellbaum gibt. Dazu wird in Zeile 7 ein *ModellbauminhaltSchluessel* angelegt, welcher aus dem Namen der Enumeration und dem Namen ihres Elternteils besteht. Ist dieser Schlüssel bereits in dem Inhaltsverzeichnis enthalten, so wird dieser Knoten ausgelesen und um die Id des aktuellen Elements ergänzt (Zeile 13). Andernsfalls wird ein neuer *EnumerationKnoten* angelegt und in das Inhaltsverzeichnis eingetragen.

In den Zeilen 21 - 38 werden die Kindelemente des Enumerationselements durchlaufen und für diese jeweils die eigene Methode des Parsers aufgerufen. Da ein *EnumerationKnoten* nur *EnumerationLiteralKnoten* als Kindelemente besitzen kann, ist diese Art die einzige, die als Kindelemente auftreten dürfen. Diese Tatsache wird in Zeile 25 geprüft. Wenn die Enumeration ein solches Kindelement besitzt, so wird dessen Parser-Methode aufgerufen und der daraus resultierende *EnumerationLiteralKnoten* wird in die entsprechende Liste des *EnumerationKnoten* eingefügt. Dadurch ist die Eltern-Kind-Beziehung hergestellt und der *EnumerationLiteralKnoten* in den Modellbaum eingefügt. Sollte die Enumeration noch andere Kindelemente besitzen, die keine Literale sind, so wird eine *UnbekanntesElementException* geworfen und die Codegenerierung bereits an dieser Stelle beendet. Wenn das Einlesen der Kindelemente keine Fehler wirft, wird der neu erstellte *EnumerationKnoten* zurück gegeben und von der aufrufenden, hier nicht abgebildeten, Methode in den Modellbaum eingefügt.

```
1  private EnumerationKnoten ParseEnumerationKnoten(XElement
       enumeration, XNamespace uml, XNamespace xmi, AbstrakterKnoten
       elternteil)
2  {
3    string name = enumeration.Attribute("name").Value;
4    string xmiId = enumeration.Attribute(xmi + "id").Value;
5    string visibility = enumeration.Attribute("visibility").Value;
6    EnumerationKnoten enumerationNode;
7    ModellbauminhaltSchluessel schluessel = new
       ModellbauminhaltSchluessel(name, elternteil.Name);
8    if (InhaltModellbaum.ContainsKey(schluessel))
9    {
10     AbstrakterKnoten abstractNode;
11     InhaltModellbaum.TryGetValue(schluessel, out abstractNode);
12     enumerationNode = (EnumerationKnoten)abstractNode;
13     enumerationNode.XmiId.Add(xmiId);
14   }
15   else
16   {
17     enumerationNode = new EnumerationKnoten(name, xmiId, visibility
         , elternteil);
```

```
18      InhaltModellbaum.Add(schluessel, enumerationNode);
19    }
20
21    if (enumeration.HasElements)
22    {
23      foreach (XElement child in enumeration.Elements())
24      {
25        if (child.Name.LocalName == "ownedLiteral" && (string)child.
              Attribute(xmi + "type") == "uml:EnumerationLiteral")
26        {
27          EnumerationLiteralKnoten tmp =
                ParseEnumerationLiteralKnoten(child, uml, xmi,
                enumerationNode);
28          if (!enumerationNode.Literals.Contains(tmp))
29          {
30            enumerationNode.Literals.Add(tmp);
31          }
32        }
33        else
34        {
35          throw new UnbekanntesElementException(child.Name + " mit
                dem Typ " + child.Attribute(xmi + "type") + " ist in dem
                 Kontext von " + enumerationNode.Name + " unbekannt!");
36        }
37      }
38    }
39    return enumerationNode;
40  }
```

Listing 5.1: Parse-Methode eines Enumerationknotens

Optimierer und Validierer

In den Klassen des Optimierers und des Validierers wird der Modellbaum durchlaufen und an den entsprechenden Stellen optimiert bzw. validiert. Da die Eltern-Kind-Beziehungen des Modellbaums vielfach über Listen implementiert wurden, bestehen diese Klassen vor allem aus Iterationen durch Listen.

Generator

Der Generator durchsucht den Modellbaum gezielt nach Informationen für einzelne Codeelemente. Für jedes generierbare Element wird eine gesonderte Methode aufgerufen, die alle nötigen Informationen aus dem Modellbaum an das dem Element zugehörige Template übergibt. Dieser Vorgang ist in Listing 5.2 für angebotene Schnittstellen dargestellt.

```
1  private IDictionary<string, string> GeneriereInterfaces(
       PackageKnoten package, string basisAusgabePfad,
       Programmiersprache sprache)
2  {
3    IDictionary<string, string> ergebnis = new Dictionary<string,
       string>();
4    foreach (var komponente in package.Komponenten)
5    {
6      foreach (var port in komponente.Ports)
7      {
8        foreach (var angeboteneSchnittstelle in port.
           AngeboteneSchnittstellen)
9        {
10         Interface schnittstelle = new Interface(
             angeboteneSchnittstelle.Name, komponente.Name);
11         string schnittstelleString = schnittstelle.TransformText(
             sprache);
12         string ausgabePfad = basisAusgabePfad + "\\" + komponente.
             Name + "\\Interfaces\\" + angeboteneSchnittstelle.Name +
              ".cs";
13         ergebnis.Add(ausgabePfad, schnittstelleString);
14       }
15     }
16   }
17   return ergebnis;
18 }
```

Listing 5.2: Generierungsmethode eines Interfaces

Die Methode *GeneriereInterface(...)* erhält über ihre Parameterliste einen *PackageKnoten* und sucht in diesem alle *AngeboteneSchnittstelleKnoten*. Aus jedem dieser Knoten wird in Zeile 10 eine Instanz eines *Interfaces* angelegt. Die Klasse *Interface* ist eine der Oberklassen in der Templatekomponente und für die Auswahl eines konkreten Templates zuständig. Das *Interface* erhält den Namen der Schnittstelle sowie den Namen der Komponente, zu welcher

die Schnittstelle gehört. Der Komponentenname wird zur Generierung des Namensraumes des Interfaces benötigt.

Der Parameter *ausgabePfad* wird benötigt um den absoluten Pfad zu der generierten Datei zusammen zu setzen. Dieser Parameter enthält den vom Benutzer angegebenen Pfad zum Basisausgabeordner. Er wird mit dem Namen der Komponente, dem Wort „Interfaces“ und dem Namen des zu generierenden Interface zum Dateipfad für die generierte Datei zusammengesetzt.

Der Parameter *sprache* wird verwendet, wenn alle Informationen aus dem übergebenen Knoten extrahiert und in ein Objekt des Typs *Interface* gespeichert wurden. Die Klasse *Interface* bietet einen Aufruf zur Transformation der Templates in Codedateien. Dieser Aufruf geschieht in Zeile 11 des Listing 5.2. Der Methode *TransformText(...)* in Listing 5.3 wird der Parameter *sprache* übergeben.

```
1 public string TransformText(Programmiersprache sprache)
2 {
3   if (sprache == Programmiersprache.CSharp)
4   {
5     return new CSInterface(InterfaceName, KomponenteName).
        TransformText();
6   }
7   //...
8   else return "Die Codegenerierung hat beim Generieren des Codes
      eines Interfaces Fehler gemacht!";
9 }
```

Listing 5.3: Methode *TransformText(...)* zur Generierung von Interfaces

Mit Hilfe des Parameters wird entschieden, welches Template aufgerufen und transformiert wird. In Zeile 5 wird die *TransformText()*-Methode des Templates aufgerufen, welche den konkreten Code für die Programmiersprache C# generiert. Der so erzeugte String wird an die Generierungsmethode in Listing 5.2 zurück gegeben und zusammen mit dem erstellten Ausgabepfad der Datei in das Rückgabe-Dictionary eingefügt. Mit Hilfe solcher Methoden werden nach und nach alle Dateien generiert und in ein Dictionary geschrieben. Dieses wird dann an die Fassade und von dort aus an den Dateisystemadapter weiter geleitet, welcher dann die Speicherung der Dateien an den jeweiligen Pfaden übernimmt.

Benutzeroberfläche

Die Benutzeroberfläche wurde, wie in Abbildung 5.2 abgebildet, umgesetzt. Sie ist sehr einfach gehalten, bietet aber alle relevanten Funktionen an, um die Codegenerierung vorzu-

bereiten. Über den Button „Eingabedateien wählen...“ öffnet sich ein Dateidialog, über den eine oder mehrere Eingabedateien angegeben werden können. Dieser Dialog prüft direkt, dass die angegebenen Dateien nur XML-Dateien sind, sonst werden sie nicht eingelesen. Über die Schaltfläche „Eingabedateien löschen“ können die Eingabedaten wieder gelöscht werden, sollten diese fehlerhaft gewesen sein. Dabei ist zu beachten, dass das Löschen alle angegebenen Dateien löscht, nicht nur die aktuell markierte.

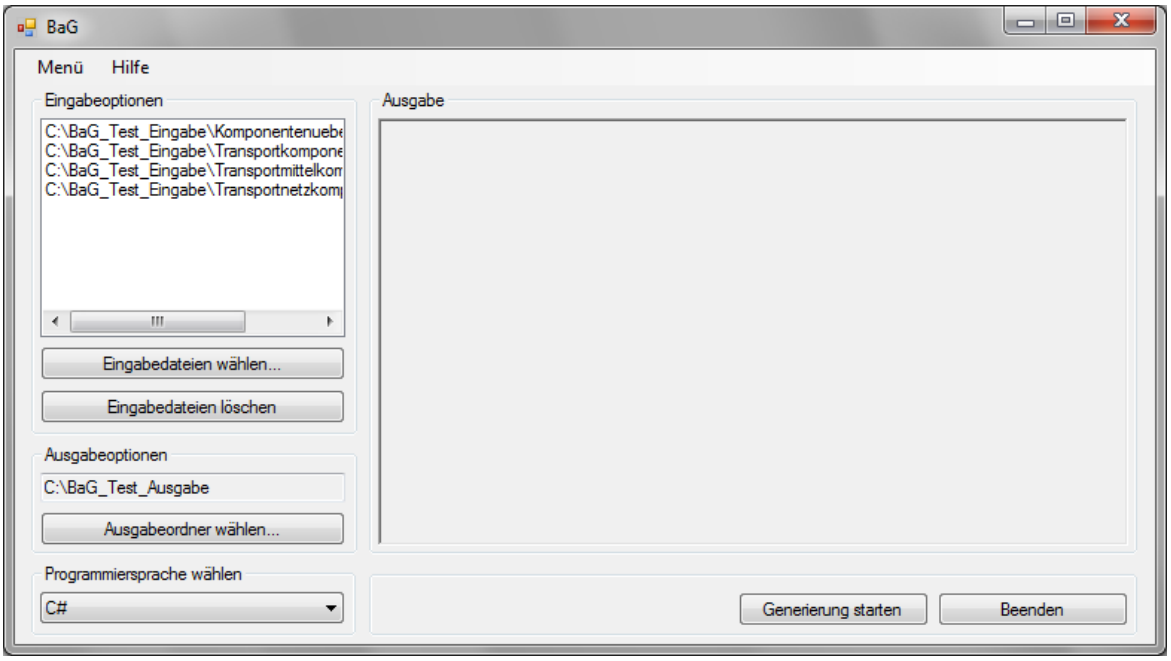


Abbildung 5.2.: Benutzeroberfläche des Codegnerators

Die Schaltfläche “Ausgabeordner wählen...“ öffnet einen weiteren Dialog, durch den der Ausgabeordner angegeben werden kann. Dieser Dialog lässt nur die Auswahl eines Ordners, nicht aber einer Datei zu und setzt somit die Anforderung A-2.3 um. Schließlich kann durch die Auswahlliste die zu generierende Programmiersprache ausgewählt werden. Über den Button „Codegenerierung starten“ kann dann die Generierung gestartet werden, sofern alle Eingabedaten angegeben wurden (vgl. Anforderung A-2.5). Der Button „Beenden“ beendet die gesamte Anwendung.

Der Aufbau der Benutzeroberfläche wurde so konzipiert, dass zunächst die linke Spalte mit den Eingabedaten abgearbeitet werden sollte, bevor in der rechten Spalte die Generierung des Codes und die Beendigung des Programms ausgeführt und Nachrichten angezeigt werden können.

Die linke Seite folgt dabei den intuitiven Eingaben - zuerst müssen die Eingabedateien angegeben werden, danach folgt der Basisausgabeordner. Zum Schluss bleibt die Eingabe der Programmiersprache übrig. Da auf die explizite Angabe der Programmiersprache aber

verzichtet werden kann, da als Standard bereits C# eingestellt ist, wurde die Auswahl der Programmiersprache nicht in der Fokus der Oberfläche gelegt.

Die rechte Seite der Benutzeroberfläche bietet Platz für die Ausgabe des Codegenerators. Diese Ausgabe sollte generell, wie auch in Abschnitt 6.1 beschrieben wird, noch ausgearbeitet und informativer gestaltet werden. So sind Statusinformationen, wie beispielsweise die für die Codegenerierung benötigte Zeit sowie die Anzahl an erzeugten Dateien und Zeilen von Code, denkbar.

5.3. Tests

Der entwickelte Codegenerator wurde mit Hilfe verschiedener Verfahren getestet, die nachfolgend vorgestellt werden. Die Testdaten für sämtliche Tests wurden auf Basis des Fallbeispiels entwickelt, weisen an vielen Stellen jedoch kürzungsbedingte Abweichungen auf. Eine vollständige Liste aller Testspezifikationen findet sich in Anhang D.

Unittest

Die implementierten Schnittstellen der Anwendung wurden durch Unittests getestet. Diese lassen sich auf der Testebene der Komponententests ansiedeln und testen einzelne Komponenten unabhängig von ihrer Umgebung. Es ging in diesem Testschritt weniger darum, funktionale Fehler der Anwendung zu finden, sondern vielmehr um Robustheitstests der Schnittstellen gegenüber fehlerhafter Datenformate. Darum wurden vor allem die Parameter der öffentlichen Schnittstellenmethoden getestet.

Obwohl das Interface *IGui* zum jetztigen Stand der Implementierung alle anderen Interfaces kapselt und nur über dieses Interface der Zugriff auf die BaG-Komponente und deren Kinder realisiert ist, wurden dennoch auch die inneren Schnittstellen getestet. Ein Test der inneren Schnittstellen ist wichtig, da das Fassade-Entwurfsmuster nach Gamma u. a. (1995) eine Kapselung von Komponenten nur anbietet, nicht jedoch erzwingt. Dementsprechend wäre es möglich, auch direkt Aufrufe an Schnittstellen, die zur Zeit von der *IGui*-Schnittstelle gekapselt werden, zu senden.

Die Umsetzung dieser Tests erfolgte mit Hilfe des im Visual Studio integrierten Unit Test Frameworks. Pro implementierter Schnittstelle wurde eine Unittestklasse angelegt. Diese wurde in mehrere Testmethoden aufgeteilt, welche jeweils einen Parameter der zu testenden Methoden überprüft.

Des Weiteren existiert für jede Methode eine Testmethode, welche den positiven Durchlauf der Methode überprüft. Dadurch wurde gezeigt, dass die Funktionalität der einzelnen Komponente funktionsfähig ist. Weitere funktionale Tests werden in dem nachfolgenden Abschnitt dieses Kapitels erläutert.

Funktionaler Test

Die funktionalen Tests des Codegenerators lassen sich auf der Ebene der Systemtests ansiedeln. Ein Systemtest testet das Zusammenspiel aller Komponenten, wenn das System komplett implementiert ist.

Diese Anwendung wird experimentell zunächst mit Hilfe des in Kapitel 2 vorgestellten Fallbeispiels getestet. Dieser Test wird manuell ausgeführt, indem der Anwendung über die Benutzerschnittstelle die benötigten Daten angegeben und die resultierenden generierten Dateien überprüft werden. Dieses Vorgehen hat den Vorteil, dass auch die Benutzerschnittstelle zumindest rudimentär auf ihre Funktionalität überprüft wird. Es reicht jedoch nicht aus, um die angestrebte nachfolgend beschriebene Testüberdeckung zu erreichen.

Es ist das Ziel, alle, in Kapitel 3 vorgestellten, Anforderungen und Übersetzungsmöglichkeiten des Modells in Code, abzudecken. Dazu wurden aus den Modellen des Fallbeispiels für jedes Diagrammelement XML-Dokumente erzeugt, die in den Codegenerator eingelesen werden. Der aus diesen Dokumenten erzeugte Code wird erneut eingelesen und mit einem zuvor gespeicherten und eingelesenen erwarteten Ergebnis verglichen.

Pro Element existiert ein Test, welcher prüft, ob das Generieren von Code aus einem solchen Element möglich ist. Dieser Test soll also zeigen, dass der Codegenerator funktioniert und den Code korrekt erzeugt. Zusätzlich sollen die Funktionen der einzelnen Generierungskomponenten *Parser*, *Optimierer*, *Validierer* und *Generator* getestet werden. Der positive Durchlauf wird bereits mit dem beschriebenen XML-Dokument überprüft. Zusätzlich muss es Testfälle geben, welche die Ausnahme- und Fehlersituationen überprüfen.

Hierzu wurden für die Parserkomponente bereits die Testfälle F-1a bis F-18a erstellt. Jede Methode, die für das Einlesen eines XElements zuständig ist (vgl. Abschnitt 5.2), kann eine *UnbekanntesElementException* werfen, wenn das aktuelle Element einen nicht bekannten Kindknoten enthält. Um diese Fälle zu prüfen wurden XML-Dokumente erstellt, die eben diese Exception hervorrufen sollen. Diese Dokumente besitzen das jeweils zu testende XML-Element mit einem unbekannten Kindelement.

Ein ebensolches Vorgehen ist auch für die weiteren Komponenten *Optimierer*, *Validierer* und *Generator* notwendig. Diese Testfälle sind aber noch nicht umgesetzt.

In Bezug auf den Optimierer sollte vor allem überprüft werden, was geschieht, wenn ein Element mit falschen oder unkonsistenten Referenzen eingegeben wird.

Der Validierer sollte auf die Anforderungen aus Abschnitt 3.2 getestet werden. Um beispielsweise die Anforderung D-1.2 zu testen sollte ein XML-Dokument erstellt werden, welches ein Portelement ohne angegebenem Typ enthält.

Wenn die Daten der Modelle in der Generatorkomponente ankommen, sind sie bereits vom Parser, vom Optimierer und vom Validierer verarbeitet und geprüft worden. Wenn keine dieser drei Komponenten Fehler in den Eingabedaten gefunden hat, kann davon ausgegangen werden, dass die Daten valide sind. Darum liegt die größte Fehlerquelle des Generators darin, dass er falsch konzipiert wurde und dementsprechend falschen Code generiert. Darum sollten für den funktionalen Test des Generators viele XML-Dokumente erzeugt werden, die eine große Bandbreite von verschiedenen Diagrammelementen und ihren Kombinationen enthalten.

Die funktionalen Tests wurden automatisiert. Dazu wurde wie schon bei den Unittests das Unit Test Framework von Visual Studio verwendet. Es wurde pro Komponente eine Testklasse angelegt. Pro Diagrammelement existiert eine Testmethode für das Element. Die Testmethode ruft jeweils die Methode *StarteGenerierung(. . .)* der *IGui*-Schnittstelle auf und übergibt ihr das XML-Dokument des zu testenden Knotens. Die Methode *StarteGenerierung(. . .)* umfasst das Speichern der generierten Dateien im Dateisystem. Diese werden darum nach ihrer Generierung durch eine Hilfsmethode eingelesen und mit weiteren eingelesenen erwarteten Codedateien verglichen.

Zusätzlich zu den funktionalen Tests sollte ein Lasttest angedacht werden, der prüft, wie viele Dateien der Codegenerator in einem Durchlauf verarbeiten kann. Dazu wurde die Anforderung A-6.3 aufgestellt. Die aktuelle in dieser Anforderung genannte Zahl ist willkürlich gewählt und dient vor allem dazu, eine Ziffer für die Testfallspezifikation zu besitzen.

6. Fazit

In dieser Arbeit wurden Komponenten- und Kompositionsstrukturdiagramme der UML daraufhin untersucht, ob es sinnvoll ist, sie zur Codegenerierung zu verwenden.

Die Analyse der Komponenten- und Kompositionsstrukturdiagramme hat gezeigt, dass sie eine gute Basis zur Codegenerierung bieten. Sie liefern, wie in Kapitel 2 erläutert, kontextspezifische Informationen, welche aus reinen Klassendiagrammen nicht hervorgehen.
Es wäre allerdings sinnvoll, sowohl Klassendiagramme als auch Komponenten- und Kompositionsstrukturdiagramme, in die Codegenerierung einfließen zu lassen. Dies würde ermöglichen, klassenspezifische Strukturen, wie Methodenrümpfe und Klassenvariablen, und Referenzen auf andere Klassen sowie die Anordnung der Klassen untereinander, darzustellen. Wie in den nachfolgenden Erweiterungsmöglichkeiten beschrieben, wäre es sinnvoll, auch die Darstellung von Methoden in den Diagrammen zu unterstützten. Diese würden einen deutlichen Mehrwert an generierbarem Code bringen und weitere Abhängigkeiten, insbesondere im Bereich der Konnektoren verdeutlichen und generierbar machen.

Mit Hilfe des erstellten Prototypen konnte gezeigt werden, dass die Codegenerierung aus den Komponenten- und Kompositionsstrukturdiagrammen der UML grundsätzlich funktioniert und umsetzbar ist.

Insgesamt hat sich die entworfene Architektur des Codegenerators im Einsatz im Prototypen bewährt. Die tatsächliche Architektur des Codegenerators muss, wie bereits in Abschnitt 5.2 angedeutet, noch an die in Abschnitt 4.3 beschriebene Architektur angepasst werden, um die volle Austauschbarkeit und Erweiterbarkeit, wie sie die Anforderungen A-6.1 und A-6.2 vorschreiben, zu gewährleisten. Über diese Anforderungen hinaus können durch die Modularität der Architektur aber auch die Benutzerschnittstelle, die Optimierungs- oder die Validierungskomponente ausgetauscht werden, ohne dass die anderen Komponenten davon etwas mit bekommen.

6.1. Erweiterungsmöglichkeiten

Der Prototyp des Codegenerators, der Umfang der unterstützten Diagramme und die Analyse derselben bieten viele Möglichkeiten, weitere Untersuchungen anzustellen und Konzepte

zu entwerfen. Einige wurden bereits in den vorangegangenen Kapiteln angesprochen, andere Erweiterungen bieten sich an, hängen aber entfernter mit den hier behandelten Fragestellungen zusammen. Die Erweiterungsmöglichkeiten lassen sich grob in Erweiterungen betreffend den Codegenerator und seine Architektur unterteilen. Zum anderen gibt es Erweiterungsmöglichkeiten,die die Unterstützung und Umsetzung von zusätzlichen Diagrammelementen betreffen.

Erweiterungen des Codegenerators

Zunächst sollte die Architektur des Codegenerators auf den modellierten Stand gebracht werden. Hierzu ist die Verwendung von Interfaces in der Modellbaumkomponente sowie die Ergänzung der Templates für die Programmiersprache Java nötig.

Zusätzlich sollten die fehlenden Testfälle spezifiziert und implementiert werden, um die Funktionalität und Robustheit des Systems zu überprüfen.

Als funktionale Erweiterung des Codegenerators wird die Generierung von ganzen Projekten einer bestimmten Programmiersprache empfohlen. So könnte beispielsweise eine komplette Solution für C# angelegt werden, um ein Modell direkt in Visual Studio bearbeiten zu können. Die einzelnen Komponenten werden durch Projekte dargestellt. Wie bereits in Kapitel 3 beschrieben ist das Anlegen einer Solution recht umfangreich, da verschiedene Einstellungen und Konfigurationsdateien je Projekt und pro Solution generiert werden müssen. Um auch für weitere vom Codegenerator unterstützte Programmiersprachen Projekte anzulegen, bedarf es unter Umständen einer größeren Erweiterung der Generatorkomponente. Daher sollte dieser Schritt gut geplant werden, um dem Anspruch der Modularität weiterhin zu genügen.

Optional kann der Codegenerator um die Templates anderer Programmiersprachen erweitert werden. Es wäre interessant zu untersuchen, inwiefern sich auch Programmiersprachen, die konzeptuell und syntaktisch weit von C# entfernt sind, für diesen Codegenerator eignen. Dabei müsste einerseits beachtet werden, ob die Programmiersprache die für die Abbildung der Diagramme in Code verwendeten Konzepte unterstützt. Andererseits muss auch beachtet werden, ob die vorhandene Struktur der Oberklassen in den Templates weiterhin geeignet ist. Unter Umständen müssen hier weitere Oberklassen für die Templates entwickelt werden, die speziell für die Generierung dieser neuen Programmiersprache zuständig wären.

Weiterhin sollte darüber nachgedacht werden, ob die Entwicklung einer Komponente zur graphischen Modellierung der Diagramme sinnvoll wäre. Dies würde den Codegenerator unabhängig von anderen Modellierungstools machen und damit seine Verwendung stärken.

Auch würde eine eigene Modellierungskomponente Roundtrip Engineering leichter machen. Roundtrip Engineering bietet die Möglichkeit, aus einem Modell Code zu generieren, diesen

manuell zu verändern und das Modell automatisch an diese Änderungen anzupassen. Dieses Thema bietet sich vor allem unter dem Aspekt an, dass ein Projekt, erst mit der Zeit um neue Komponenten erweitert wird. Der Code von neuen zusätzlichen Modellen könnte erst später generiert werden. Dennoch haben neue Komponenten unter Umständen Abhängigkeiten und Referenzen zu bestehenden Komponenten, welche dann in diese eingefügt werden müssen. Wenn ältere Komponenten bereits manuell ergänzt wurden, würde eine erneute Generierung diese manuellen Änderungen überschreiben. Es muss also ein Mechanismus entwickelt werden, der existierenden Code einliest, zwischen manuell und automatisch generiertem Code unterscheidet, den manuellen Code speichert und in neu generierten Code wieder einfügt.

Unterstützung von weiteren Diagrammelementen

Um den Anwendern des Codegenerators noch größere Arbeitserleichterung zu verschaffen, sollte die Abbildung von weiteren Diagrammelementen in Code konzipiert und der Codegenerator für diese Elemente erweitert werden.

Zusätzlich zur Ball-and-Socket-Notation für Schnittstellen sollte auch die Klassennotation für Schnittstellen unterstützt werden. Diese bietet den Vorteil, auch Methodendeklarationen darstellen zu können. Mit diesen ist es dann möglich, Delegationsabhängigkeiten detaillierter im Code abzubilden und bereits in den Klassen, die diese Schnittstellen implementieren, die Methodenrümpfe zu generieren.

Diese Generierung von Methodendeklarationen würde weiterführend die Generierung von Testfällen für die Schnittstellen ermöglichen. Es könnten Tests für die Schnittstellen generiert werden, die von dem Entwickler dann nur noch mit passenden Daten gefüllt werden müssen. Mit Hilfe dieser Tests könnte die spezifizierte Funktionalität der Schnittstellen sowie ihre Robustheit gegenüber falschen Eingabedaten überprüft werden.

Des Weiteren kann die Menge der unterstützten Diagrammelemente um Assoziationen und n-äre Konntektoren ergänzt werden. Assoziationen würden die Möglichkeit bieten, statische Verbindungen zwischen den Elementen des Modells darzustellen. N-äre Konnektoren werden von der Spezifikation OMG (2010b) vorgesehen, können aber nicht vom Enterprise Architect 8.0 umgesetzt werden und wurden darum bislang nicht betrachtet.

Die Generierung von objektrelationalen Mappings, wie beispielsweise für Hibernate, könnte ermöglicht werden, wenn die Menge der Diagrammelemente um den Stereotyp „Entität“ erweitert wird. Dieser könnte verwendet werden, wenn ein Kompositionsstrukturdiagramm zur Spezifikation der Struktur einer Klasse verwendet wird. In dem momentan unterstütztem Umfang von Diagrammelemente ist es nicht möglich, eine Klasse als Entität zu kennzeichen. Es wäre durch die Konzentration auf Komponentenmodellierung in dieser Arbeit nicht sinnvoll,

pauschal für jede Klasse ein objektrelationales Mapping zu generieren, da viele Klassen, wie z. B. Verwalter und Konfigurator eher für die technische oder funktionale Umsetzung der Komponente zuständig sind und nicht in einer Datenbank gespeichert werden sollen.

Die Liste der Erweiterungen lässt sich auf Grund der Fülle von UML-Diagrammen beliebig fortsetzen. An dieser Stelle wurden vor allem solche Erweiterungen vorgeschlagen, die den Aspekt der Komponentenmodellierung unterstützen.

Literaturverzeichnis

[Antlr 2011] *Antlr*. 2011. – URL http://www.antlr.org/. – Zugriff: 14.08.2011

[Bruel und Ober 2006] BRUEL, J.-M. ; OBER, I.: Components Modeling in UML 2. In: *Studia Universitatis Babes-Bolyai Series Informatica* LI (2006), S. 79–90. – URL http://www.cs.ubbcluj.ro/~studia-i/2006-1/09-BruelOber.pdf

[Buschmann u. a. 1996] BUSCHMANN, F. ; MEUNIER, R. ; ROHNERT, H. ; SOMMERLAD, P. ; STAL, M.: *Pattern-Oriented Software Architecture - A System of Patterns*. Bd. Volume 1. John Wiley & Sons Ltd., 1996. – ISBN 978-0-471-95869-7

[Cuccuru u. a. 2008] CUCCURU, A. ; GÉRARD, S. ; RADERMACHER, A.: Meaningful Composite Structures - On the Semantics of Ports in UML2. In: CZARNECKI, K. (Hrsg.) ; OBER, I. (Hrsg.) ; BRUEL, J.-M. (Hrsg.) ; UHL, A. (Hrsg.) ; VÖLTER, M. (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 5301. Springer Berlin / Heidelberg, 2008, S. 828–842. – URL http://www.springerlink.com/content/357h846tt71838v1/

[Gamma u. a. 1995] GAMMA, E. ; HELM, R. ; JOHNSON, R. ; VLISSIDES, J.: *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. – ISBN 0-201-63361-2

[Hitz u. a. 2005] HITZ, M. ; KAPPEL, G. ; KAPSAMMER, E. ; RETSCHITZEGGER, W.: *UML @ Work*. 3., aktualisierte und überarbeitete Auflage. Heidelberg : dpunkt.verlag GmbH, 2005. – ISBN 3-89864-261-5

[Iso 1999] : *ISO-Container der Reihe 1 - Klassifikation, Maße, Gesamtgewichte (ISO 668:1995)*. October 1999

[Kecher 2009] KECHER, C.: *UML 2: Das umfassende Handbuch*. 3. Auflage. Galileo Computing, May 2009. – ISBN 978-3836214193

[Msdn 2011a] *MSDN - Code Generation and T4 Text Templates*. 2011. – URL http://msdn.microsoft.com/de-de/library/bb126445.aspx. – Zugriff: 04.06.2011

[Msdn 2011b] *MSDN - Solution*. 2011. – URL http://msdn.microsoft.com/de-de/library/bb165327.aspx. – Zugriff: 14.08.2011

[NVelocity 2011] *NVelocity*. 2011. – URL http://www.castleproject.org/monorail/documentation/v20/viewengines/nvelocity/index.html. – Zugriff: 29.05.2011

[Oestereich und Bremer 2009] OESTEREICH, B. ; BREMER, S.: *Analyse und Design mit UML 2.3*. 9. Auflage. München : Oldenbourg Wissenschaftsverlag GmbH, 2009. – ISBN 978-3-486-58855-2

[OMG 2007] OMG: *MOF 2.0/XMI Mapping, Version 2.1.1*. December 2007. – URL http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm

[OMG 2010a] OMG: *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.3*. May 2010. – URL http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/

[OMG 2010b] OMG: *OMG Unified Modeling LanguageTM (OMG UML), Superstructure Version 2.3*. May 2010. – URL http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

[Siedersleben 2003a] SIEDERSLEBEN, J.: Quasar: Die sd&m Standardarchitektur Teil 1 / sd&m Research. München, April 2003. – Forschungsbericht. – URL https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-_Quasar_1__sd_m_Brosch_re_.pdf

[Siedersleben 2003b] SIEDERSLEBEN, J.: Quasar: Die sd&m Standardarchitektur Teil 2 / sd&m Research. München, May 2003. – Forschungsbericht. – URL https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-_Quasar_2__sd_m_Brosch_re_.pdf

[Siedersleben 2004] SIEDERSLEBEN, J.: *Moderne Softwarearchitektur - Umsichtig planen, robust bauen mit Quasar*. Heidelberg : dpunkt.verlag GmbH, 2004. – ISBN 3-89864-292-5

[SparxSystems 2011] *Enterprise Architect 8.0*. 2011. – URL http://www.sparxsystems.de/. – Zugriff: 29.03.2011

[Stahl u. a. 2007] STAHL, T. ; VÖLTER, M. ; EFFTINGE, S. ; HAASE, A.: *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. 2. Auflage. Heidelberg : dpunkt.verlag GmbH, 2007. – ISBN 978-3-89864-448-8

[StringTemplate 2011a] *StringTemplate*. 2011. – URL http://www.stringtemplate.org/. – Zugriff: 05.06.2011

[StringTemplate 2011b] *Using StringTemplate with CSharp*. March 2011. – URL http://www.antlr.org/wiki/x/_wKGAQ. – Zugriff: 29.07.2011

[Sych 2011] SYCH, O.: *Text Template Transformation Toolkit Blog*. 2011. – URL `http://www.olegsych.com/`. – Zugriff: 27.07.2011

[Tangible 2011] *Tangible T4 Editor*. 2011. – URL `http://t4-editor.tangible-engineering.com/T4-Editor-Visual-T4-Editing.html`. – Zugriff: 04.06.2011

[Taylor u. a. 2010] TAYLOR, R. ; MEDVIDOVIC, N. ; DASHOFY, E.: *Software Architecture - Foundations, Theory, and Practice*. John Wiley & Sons, Inc., 2010. – ISBN 978-0470-16774-8

[Usman und Nadeem 2009] USMAN, M. ; NADEEM, A.: Automatic Generation of Java Code from UML Diagrams using UJECTOR. In: *International Journal of Software Engineering and Its Applications* 3 (2009), April, Nr. 2, S. 21–38. – URL `http://www.sersc.org/journals/IJSEIA/vol3_no2_2009/3.pdf`. – Zugriff: 14.08.2011

[Usman u. a. 2008] USMAN, M. ; NADEEM, A. ; KIM, T.: UJECTOR: A tool for Executable Code Generation from UML Models. In: *Advanced Software Engineering and Its Applications* 0 (2008), December, S. 165–170. – Zugriff: 14.08.2011

[Velocity 2011] *Apache Velocity*. 2011. – URL `http://velocity.apache.org/engine/releases/velocity-1.7/`. – Zugriff: 29.05.2011

[VisualParadigm 2011] *Visual Paradigm for UML 8.1*. 2011. – URL `http://www.visual-paradigm.com/`. – Zugriff: 29.03.2011

A. Diagramme zum Fallbeispiel

Nachfolgend befinden sich das fachliche Datenmodell (Abbildung A.1) und das Übersichtsdiagramm der Komponenten (Abbildung A.2) des Fallbeispiels.

Die Farben des fachlichen Datenmodells korrespondieren zu den Komponenten des Fallbeispiels und verdeutlichen, welche Entität in welcher Komponente umgesetzt wurde.

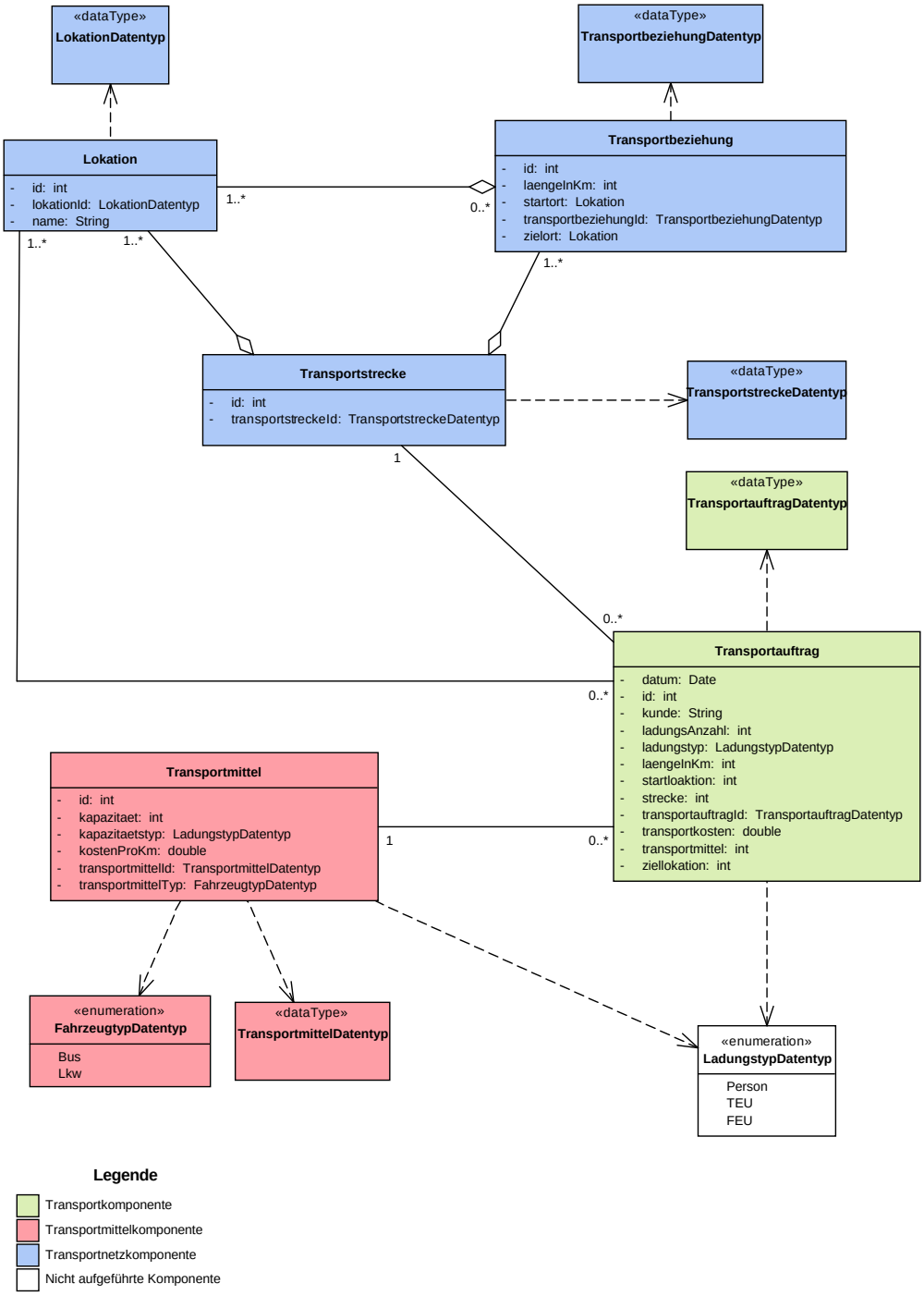


Abbildung A.1.: Fachliches Datenmodell des Fallbeispiels

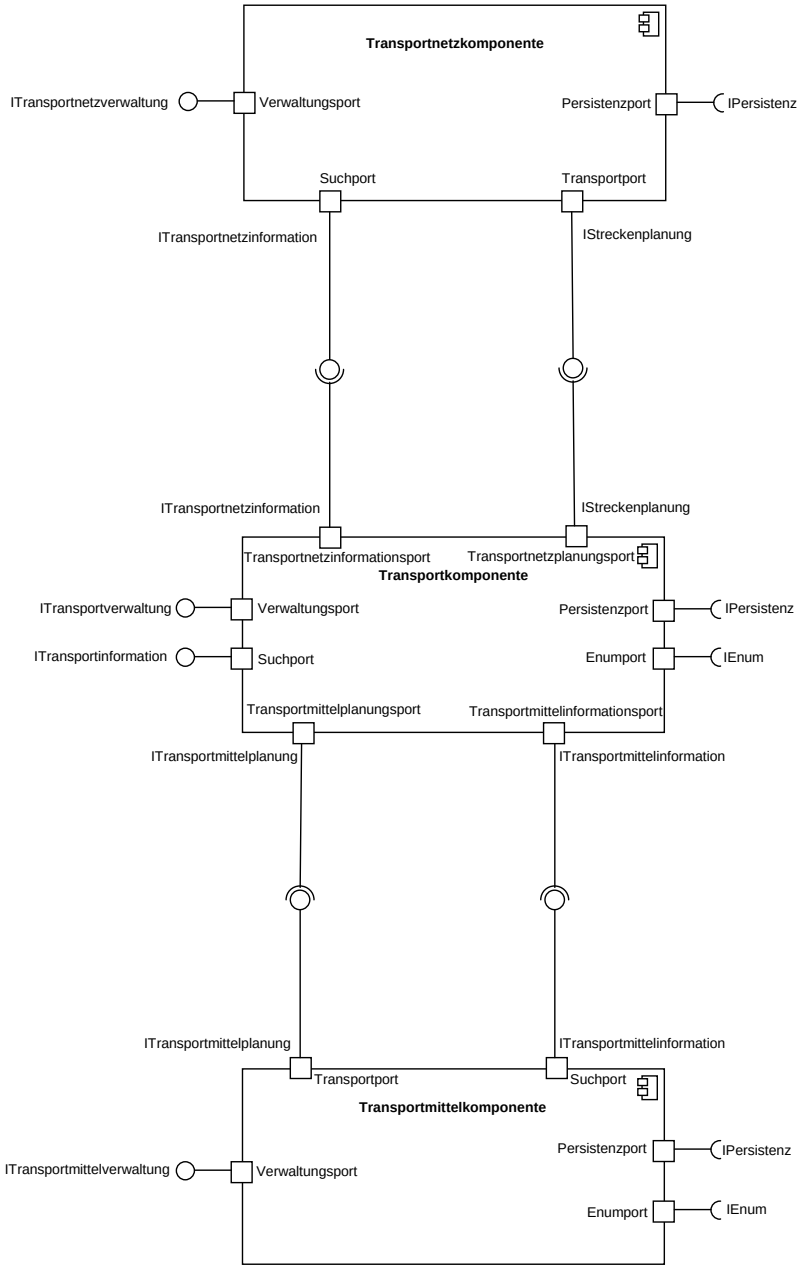


Abbildung A.2.: Komponentenzusammensetzung des Fallbeispiels

B. Anforderungen des Fallbeispiels

1. Datentypen

F-1.1 Ein Transportmitteldatentyp ist eine alphanumerische Zeichenfolge der Form „TM-<Nr>“, wobei <Nr> durch eine natürliche Zahl ersetzt werden muss.

F-1.2 Ein Lokationsdatentyp ist eine alphanumerische Zeichenfolge der Form „Lok-<Nr>“, wobei <Nr> durch eine natürliche Zahl ersetzt werden muss.

F-1.3 Ein Transportbeziehungsdatentyp ist eine alphanumerische Zeichenfolge der Form „TB-<Nr>“, wobei <Nr> durch eine natürliche Zahl ersetzt werden muss.

F-1.4 Ein Transportstreckendatentyp ist eine alphanumerische Zeichenfolge der Form „TS-<Nr>“, wobei <Nr> durch eine natürliche Zahl ersetzt werden muss.

F-1.5 Ein Transportauftragdatentyp ist eine alphanumerische Zeichenfolge der Form „TA-<Nr>“, wobei <Nr> durch eine natürliche Zahl ersetzt werden muss.

F-1.6 Der Fahrzeugtyp beeinflusst den Typ der Transportgüter, die ein Transportmittel transportieren kann. Es gibt die Fahrzeugtypen „Bus“ und LKW“.

F-1.7 Ein Ladungstyp kategorisiert die zu transportierenden Güter. Es gibt die Ladungstypen „Person“, „TEU“ und „FEU“.

2. Transportmittel

F-2.1 Ein Transportmittel repräsentiert ein Fahrzeug von einem bestimmten Typ im Transportsystem, das ein begrenzte Kapazität hat und nur bestimmte Güter transportieren kann.

F-2.2 Das System soll Transportmittel erstellen, speichern, bearbeiten und löschen können.

F-2.3 Die technische Id eines Transportmittels ist eine eindeutige numerische Referenz auf dieses und wird bei seiner ersten Speicherung in der Persistenz generiert.

F-2.4 Die fachliche Id eines Transportmittels ist ein Transportmitteldatentyp, der eine eindeutige Referenz auf das Transportmittel darstellt und beim ersten Anlegen des Transportmittels generiert wird.

F-2.5 Ein Transportmittel hat genau einen Fahrzeugtyp, der beim Ersten Erstellen des Transportmittels vom Benutzer angegeben wird und nicht mehr änderbar ist.

F-2.6 Ein Transportmittel kann Güter genau eines Ladungstypen transportieren. Dieser wird beim Ersten Erstellen des Transportmittels vom Benutzer angegeben und kann nicht mehr geändert werden.

F-2.7 Ein Transportmittel vom Fahrzeugtyp „Bus“ kann nur Ladung vom Typ „Person“ transportieren.

F-2.8 Ein Transportmittel vom Fahrzeugtyp „LKW“ kann entweder Ladung vom Typ „TEU“ oder Ladung vom Typ „FEU“, nicht jedoch von beiden, transportieren.

F-2.9 Die Kapazitätsanzahl gibt die Anzahl der Einheiten des jeweiligen Ladungstyps an, die maximal auf einmal mit einem Transportmittel transportiert werden können.

F-2.10 Die Kosten eines Transportmittels pro Kilometer werden in € angegeben. Diese werden von dem Benutzer beim Ersten Erstellen des Transportmittels angegeben und können bei Bedarf bearbeitet werden.

3. Lokation

F-3.1 Eine Lokation repräsentiert einen Ort im Transportsystem, der Start- oder Zielort eines Transportes, einer Transportbeziehung oder einer Transportstrecke sein kann.

F-3.2 Das System soll Lokationen erstellen, speichern und löschen können.

F-3.3 Die technische Id einer Lokation ist eine eindeutige numerische Referenz auf diese und wird bei ihrer ersten Speicherung in der Persistenz generiert.

F-3.4 Die fachliche Id einer Lokation ist ein Lokationsdatentyp, der eine eindeutige Referenz auf die Lokation darstellt und beim ersten Anlegen der Lokation generiert wird.

F-3.5 Der Name einer Lokation beschreibt diese und gibt implizit den Ort an, z. B. „Hamburg“ oder „Berliner Tor 7“.

F-3.6 Der Name einer Lokation ist durch den Benutzer bei der Erzeugung der Lokation anzugeben. Er ist nicht änderbar.

4. Transportbeziehung

F-4.1 Eine Transportbeziehung stellt eine gerichtete Verbindung von einer bestimmten Länge zwischen zwei Lokationen in dem Transportsystem dar und kann von einem Transportmittel befahren werden.

F-4.2 Das System soll Transportbeziehungen erstellen, speichern, bearbeiten und löschen können.

F-4.3 Die technische Id einer Transportbeziehung ist eine eindeutige numerische Referenz auf diese und wird bei ihrer ersten Speicherung in der Persistenz generiert.

F-4.4 Die fachliche Id einer Transportbeziehung ist ein Transportbeziehungsdatentyp, der eine eindeutige Referenz auf die Transportbeziehung darstellt und beim ersten Anlegen der Transportbeziehung generiert wird.

F-4.5 Der Startort einer Transportbeziehung ist eine Referenz auf eine Lokation des Transportsystems, von der aus die Transportbeziehung wegführt. Der Startort wird bei der ersten Erstellung der Transportbeziehung vom Benutzer angegeben und kann dann nicht mehr bearbeitet werden.

F-4.6 Der Zielort einer Transportbeziehung ist eine Referenz auf eine Lokation des Transportsystems, zu der die Transportbeziehung hinführt. Der Zielort wird bei der ersten Erstellung der Transportbeziehung vom Benutzer angegeben und kann dann nicht mehr bearbeitet werden.

F-4.7 Die Länge einer Transportbeziehung wird in Kilometern angegeben. Sie wird vom Benutzer bei der ersten Erstellung der Transportbeziehung angegeben und kann bei Bedarf bearbeitet werden.

5. Transportstrecke

F-5.1 Eine Transportstrecke stellt eine Abstraktion einer Liste von Transportbeziehungen dar, die einen Startort mit einem Zielort verbinden und für einen Transportauftrag erstellt wird.

F-5.2 Das System soll Transportstrecken erstellen, speichern und löschen können.

F-5.3 Die technische Id einer Transportstrecke ist eine eindeutige numerische Referenz auf diese und wird bei ihrer ersten Speicherung in der Persistenz generiert.

F-5.4 Die fachliche Id einer Transportstrecke ist ein Transportstreckendatentyp, der eine eindeutige Referenz auf die Transportstrecke darstellt und beim ersten Anlegen der Transportstrecke generiert wird.

F-5.5 Der Startort einer Transportstrecke ist eine Referenz auf eine Lokation des Transportsystems, von der aus die Transportstrecke wegführt. Der Startort wird bei der ersten Erstellung eines Transportauftrages vom Benutzer angegeben und an die Transportstrecke weitergereicht. Bei einer Transportstrecke kann der Startort nicht mehr bearbeitet werden. Gibt der Benutzer einen neuen Startort für den Transportauftrag an, so wird eine neue Transportstrecke erstellt.

F-5.6 Der Zielort einer Transportstrecke ist eine Referenz auf eine Lokation des Transportsystems, zu der die Transportstrecke hinführt. Der Zielort wird bei der ersten Erstellung eines Transportauftrages vom Benutzer angegeben und an die Transportstrecke weitergereicht. Bei einer Transportstrecke kann der Startort nicht mehr bearbeitet werden. Gibt der Benutzer einen neuen Zielort für den Transportauftrag an, so wird eine neue Transportstrecke erstellt.

F-5.7 Die Strecke einer Transportstrecke ist eine geordnete Liste von Referenzen auf Transportbeziehungen des Transportsystems, welche den Startort der Transportstrecke mit ihrem Zielort verbinden. Diese Liste umfasst in dem Fall, dass es eine direkte Transportbeziehung zwischen Start- und Ziellokation gibt, genau ein Element. In allen anderen Fällen werden Transportbeziehungen gesucht, die die Startlokation mit einer Zwischenlokation verbinden, welche wiederum mit der Ziellokation oder weiteren Zwischenlokationen verbunden ist.

F-5.8 Eine Transportstrecke wird automatisch erstellt, wenn ein neuer Transportauftrag in dem System gespeichert wird oder der Start- oder Zielort eines bestehenden Transportauftrages geändert werden.

6. Transportauftrag

F-6.1 Ein Transportauftrag bündelt alle relevanten Daten eines Transportvorganges und stößt automatisch die Transportplanung, wie die Auswahl eines passenden Transportmittels und die Routenplanung, an.

F-6.2 Das System soll Transportaufträge erstellen, speichern, bearbeiten und löschen können.

F-6.3 Die technische Id eines Transportauftrags ist eine eindeutige numerische Referenz auf diesen und wird bei seiner ersten Speicherung in der Persistenz generiert.

F-6.4 Die fachliche Id eines Transportauftrags ist ein Transportauftragdatentyp, der eine eindeutige Referenz auf den Transportauftrag darstellt und beim ersten Anlegen des Transportauftrags generiert wird.

F-6.5 Der Startort eines Transportauftrags ist eine Referenz auf eine Lokation und wird bei der Erstellung des Transportauftrags vom Benutzer angegeben. Er kann bei Bedarf bearbeitet werden.

F-6.6 Der Zielort eines Transportauftrags ist eine Referenz auf eine Lokation und wird bei der Erstellung des Transportauftrags vom Benutzer angegeben. Er kann bei Bedarf bearbeitet werden.

F-6.7 Das Datum eines Transportauftrags beschreibt das Datum an dem der Transport durchgeführt werden soll. Es kann bei der ersten Erstellung des Transportauftrags angegeben und nachträglich bearbeitet werden.

F-6.8 Der Kunde eines Transportauftrages stellt den Namen des Auftraggebers des Transports dar. Er wird bei der ersten Erstellung des Transportauftrags angegeben und kann nicht mehr geändert werden. Wenn sich der Auftraggeber ändert, muss ein neuer Transportauftrag angegeben werden.

F-6.9 Einem Transportauftrag können zu transportierende Güter von genau einem Ladungstyp zugeordnet werden.

F-6.10 Die Anzahl der zu transportierenden Güter wird mit der Anzahl von Einheiten des Ladungstyp beschrieben.

F-6.11 Ein Transportauftrag erhält eine Referenz auf ein Transportmittel, sobald Ladungstyp und -anzahl im Transportauftrag gespeichert werden.

F-6.12 Das System erstellt die Referenz auf ein Transportmittel, indem ein Transportmittel gesucht wird, welches den im Transportauftrag angegebenen Ladungstyp transportieren kann und Kapazitäten entsprechend der angegebenen Ladungsanzahl besitzt.

F-6.13 Ein Transportauftrag erhält eine Referenz auf eine Transportstrecke, sobald Start- und Ziellokation im Transportauftrag gespeichert werden.

F-6.14 Das System erstellt die Referenz auf eine Transportstrecke indem eine Liste von Transportbeziehungen gesucht wird, die die Startlokation mit der Ziellokation verbinden.

F-6.15 Die Länge eines Transportauftrages wird in Kilometern angegeben.

F-6.16 Die Länge eines Transportauftrages wird vom System aus der Summe der Streckenlängen der Transportbeziehungen, die in der zugehörigen Transportstrecke enthalten sind, berechnet.

F-6.17 Die Kosten eines Transportauftrages werden in € angegeben.

F-6.18 Die Kosten eines Transportauftrages werden vom System aus dem Produkt der Länge dieses Transportauftrages und den Kosten pro Kilometer dieses Transportmittels berechnet.

7. Ausgrenzungen

F-7.1 Es gibt keine zeitliche Kapazitätsbeschränkung von Transportmitteln.

C. Anforderungen des Codegenerators

1. Allgemein

A-1.1 Die Codegenerierung wird in vier Schritte eingeteilt:

- Parsen
- Optimieren
- Validieren
- Generieren

A-1.2 Wenn die Codegenerierung abgebrochen wird, wird nicht die gesamte Applikation beendet. Stattdessen beendet sich nur der Generierungsvorgang und der Anwender hat die Möglichkeit, seine Eingabedaten nochmal zu setzen und die Codegenerierung erneut zu starten.

A-1.3 Fehlermeldungen und Informationen an den Anwender enthalten immer den Namen des Schrittes der Codegenerierung, in der der Fehler aufgetreten ist, sowie wo möglich eine Angabe über das Element, das den Fehler verursacht hat.

2. Benutzerschnittstelle

A-2.1 Der Anwender muss mehrere Eingabedateien aus dem lokalen Dateisystem über die Benutzerschnittstelle angeben können.

A-2.2 Der Basisausgabeordner, in dem die generierten Dateien in einer Ordnerstruktur gespeichert werden, wird vom Anwender über die Benutzerschnittstelle angegeben.

A-2.3 Der angegebene Pfad zum Basisausgabeordner muss ein Ordner sein.

A-2.4 Der Anwender kann die zu generierende Programmiersprache über eine Liste in der Benutzerschnittstelle auswählen. Zur Verfügung stehen Visual C# 2010 und Java 1.6.

A-2.5 Die Eingabedaten sind ausreichend, wenn mindestens eine Eingabedatei, der Basisausgabeordner und die Programmiersprache angegeben wurden. Liegen die Eingabedaten nicht ausreichend vor, so wird der Anwender darüber informiert.

A-2.6 Die Codegenerierung wird durch den Anwender gestartet, wenn Anforderung A-2.5 erfüllt ist.

3. Validierung

A-3.1 Das System muss prüfen, dass die eingegebenen Diagramme in der in Abschnitt 3.2 spezifizierten Form vorliegen. Liegen die Daten in dieser Form vor, so sind sie semantisch valide.

A-3.2 Das System darf bei semantisch nicht validen Eingabedaten nicht abstürzen sondern muss die Codegenerierung abbrechen und den Anwender darüber informieren.

4. Unterstützte Eingabedaten

A-4.1 Die Eingabedateien müssen als XML-Dateien aus dem Enterprise Architect 8.0 exportiert werden und mit folgenden verwendeten XML-Namensräumen vorliegen:

- http://schema.omg.org/spec/UML/2.1
- http://schema.omg.org/spec/XMI/2.1
- http://www.sparxsystems.com/profiles/EAUML/1.0

A-4.2 Das System muss die folgenden statischen Elemente von Komponenten- und Kompositionsstrukturdiagrammen einlesen und in der Generierung verarbeiten können:

- Komponente
- Part
- Port
- Klasse
- Datentyp
- Enumeration

A-4.3 Das System muss die folgenden Verbindungselemente von Komponenten- und Kompositionsstrukturdiagrammen einlesen und in der Generierung verarbeiten können:

- „use“-Abhängigkeit
- „instantiate“-Abhängigkeit
- Delegationskonnektor
- Kompositionskonnektor
- Konnektor

5. Generierung

A-5.1 Der Code kann in einer der folgenden Programmiersprachen generiert werden:

- Visual C# 2010
- Java 1.6

A-5.2 Die vom Codegenerator voreingestellte Programmiersprache zur Codegenerierung ist Visual C# 2010. Sie wird verwendet, falls der Anwender keine andere Auswahl trifft.

A-5.3 Der generierte Code muss in der folgenden Ordnerstruktur abgelegt werden:

- pro Komponente ein Ordner mit dem Namen der Komponente direkt im Basisausgabeordner.
- in jedem Komponentenordner ein Ordner *Interfaces* für die generierten Dateien, die Interfaces darstellen.
- in jedem Komponentenordner ein Ordner *Enumerations* für die generierten Dateien, die Enumerationen darstellen.
- Alle anderen generierten Dateien, die eindeutig einer Komponente zugeordnet werden können, werden direkt in den Komponentenordner gelegt.
- Dateien, die nicht eindeutig einer Komponente zugeordnet werden können, werden in den Basisausgabeordner gelegt.

A-5.4 Alle generierten Dateien erhalten einen Kommentar, der besagt, dass die Datei automatisch generiert wurde und darauf hinweist, dass eine erneute Ausführung manuell getätigte Änderungen überschreibt.

A-5.5 Wenn die Eingabedaten nicht in der in A-4.1 unterstützten Form vorliegen, so wird die Codegenerierung abgebrochen und der Anwender informiert, aus welchem Grund die Generierung abgebrochen wurde.

6. Nichtfunktionale Anforderungen

A-6.1 Das System soll so modular konzipiert sein, dass ein Austausch des Codegenerators und des Parsers möglich ist.

A-6.2 Die Templates der zu generierenden Programmiersprache sollen austauschbar sein, ohne dass neben der Generatorkomponente eine andere Komponente geändert werden muss.

A-6.3 Das System muss mindestens 5 Eingabedateien in einem Codegenerierungsprozess verarbeiten können.

D. Testfallspezifikationen

D.1. Unittest

Die Variable *basepath* enthält den Basispfad zu der ausgeführten Generatoranwendung. Da der Basispfad von der Ausführungsumgebung abhängig ist, kann dieser hier nicht näher spezifiziert werden.

Testfall Gui-1	
Methode	IGui.**StarteGenerierung**(IList<string> eingabePfade, string ausgabe-Pfad, Programmiersprache sprache)
Eingabe	**eingabePfade** = `basepath\Unittests\ressources\testinput\TestGui.xml` **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	Generierungsergebnis: *true*, Codegenerierung erfolgreich! In den Ordner `basepath\Testausgaben` wurden 1 Dateien aus folgenden Eingaben generiert: `basepath\Unittests\ressources\testinput\TestGui.xml`
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.1.: Testfall Gui-1

Testfall Gui-2a	
Methode	IGui.**StarteGenerierung**(IList<string> eingabePfade, string ausgabe-Pfad, Programmiersprache sprache)
Eingabe	**eingabePfade** = null **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.CSharp

Tabelle D.2 – Fortsetzung auf der nächsten Seite

Tabelle D.2 – Fortsetzung

Testfall Gui-2a	
Erwartetes Ergebnis	Generierungsergebnis: *false*, Eingabedateien dürfen nicht leer sein! Bitte korrigieren Sie diese Angaben.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.2.: Testfall Gui-2a

Testfall Gui-2b	
Methode	IGui.**StarteGenerierung**(IList<string> eingabePfade, string ausgabe-Pfad, Programmiersprache sprache)
Eingabe	**eingabePfade** = new List<string>() **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	Generierungsergebnis: *false*, Eingabedateien dürfen nicht leer sein! Bitte korrigieren Sie diese Angaben.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.3.: Testfall Gui-2b

Testfall Gui-3a	
Methode	IGui.**StarteGenerierung**(IList<string> eingabePfade, string ausgabe-Pfad, Programmiersprache sprache)
Eingabe	**eingabePfade** = `basepath\Unittests\ressources\testinput\TestGui.xml` **ausgabePfad** = null **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	Generierungsergebnis: *false*, Ausgabedateien dürfen nicht leer sein! Bitte korrigieren Sie diese Angaben.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.4.: Testfall Gui-3a

Testfall Gui-3b	
Methode	IGui.**StarteGenerierung**(IList<string> eingabePfade, string ausgabe-Pfad, Programmiersprache sprache)
Eingabe	**eingabePfade** = `basepath\Unittests\ressources\testinput\TestGui.xml` **ausgabePfad** = ““ **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	Generierungsergebnis: *false*, Ausgabedateien dürfen nicht leer sein! Bitte korrigieren Sie diese Angaben.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.5.: Testfall Gui-3b

Testfall Gui-4	
Methode	IGui.**StarteGenerierung**(IList<string> eingabePfade, string ausgabe-Pfad, Programmiersprache sprache)
Eingabe	**eingabePfade** = `basepath\Unittests\ressources\testinput\TestGui.xml` **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	Generierungsergebnis: *false*, Java wird leider noch nicht unterstützt! Bitte korrigieren Sie diese Angaben.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.6.: Testfall Gui-4

Testfall De-1	
Methode	IDateiIO.**LeseEingabedaten**(IList<string> eingabePfade)
Eingabe	**eingabePfade** = `basepath\Unittests\ressources\testinput\TestDatei.xml`
Erwartetes Ergebnis	new List<string>() mit Eintrag: "<?xml version=\"1.0\" encoding=\"windows−1252\"?>\r\n<xmi:XMI xmi:version=\"2.1\" xmlns:uml=\"http://schema.omg.org/spec/UML/2.1\" xmlns:xmi=\"http://schema.omg.org/spec/XMI/2.1\">\r\n</xmi:XMI>\r\n"
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.7 – Fortsetzung auf der nächsten Seite

Tabelle D.7 – Fortsetzung

Testfall De-1

Tabelle D.7.: Testfall De-1

Testfall De-2a	
Methode	IDateiIO.**LeseEingabedaten**(IList<string> eingabePfade)
Eingabe	**eingabePfade** = null
Erwartetes Ergebnis	new List<string>()
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.8.: Testfall De-2a

Testfall De-2b	
Methode	IDateiIO.**LeseEingabedaten**(IList<string> eingabePfade)
Eingabe	**eingabePfade** = new List<string>()
Erwartetes Ergebnis	new List<string>()
VerifVerifizie-rungizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.9.: Testfall De-2b

Testfall Da-1	
Methode	IDateiIO.**SchreibeAusgabedaten**(IDictionary<string, string> ausgabedaten)
Eingabe	**ausgabedaten** = new Dictionary<string, string>(); mit Tupel: Key = `basepath\Unittests\ressources\testoutput\TestDateiOut.xml` Value = <?xml version=\"1.0\" encoding=\"windows−1252\"?>
Erwartetes Ergebnis	Speicherergebnis: *true*,
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.10.: Testfall Da-1

Testfall Da-2a	
Methode	IDateiIO.**SchreibeAusgabedaten**(IDictionary<string, string> ausgabedaten)
Eingabe	**ausgabedaten** = null
Erwartetes Ergebnis	Speicherergebnis: *false*, Die Eingabedaten in die Speicherkomponente sind leer.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.11.: Testfall Da-2a

Testfall Da-2b	
Methode	IDateiIO.**SchreibeAusgabedaten**(IDictionary<string, string> ausgabedaten)
Eingabe	**ausgabedaten** = new Dictionary<string, string>()
Erwartetes Ergebnis	Speicherergebnis: *false*, Die Eingabedaten in die Speicherkomponente sind leer.
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.12.: Testfall Da-2b

Testfall P-1	
Methode	IParse.**Parse**(IList<string> eingabeDaten)
Eingabe	**eingabeDaten** = new List<string>() mit Eintrag: <?xml version=\"1.0\" encoding=\"windows−1252\"?>\r\n<xmi:XMI xmi:version=\"2.1\" xmlns:uml=\"http://schema.omg.org/spec/UML/2.1\" xmlns:xmi=\"http://schema.omg.org/spec/XMI/2.1\">\r\n<xmi:Documentation exporter=\"Enterprise Architect\" exporterVersion=\"6.5\"/>\r\n<uml:Model xmi:type=\"uml:Model\" name=\"EA_Model\" visibility=\"public\">\r\n<packagedElement xmi:type=\"uml:Package\" xmi:id=\"EAPK_3DE4D1EF_8043_4023_AFE4_7BEF2D39376C\" name=\"Model\" visibility=\"public\">\r\n</packagedElement>\r\n</uml:Model>\r\n</xmi:XMI>\r\n
Erwartetes Ergebnis	modellbaum = new Modellbaum("Modellbaumname"); modellbaum.Packages.Add(new PackageKnoten(ËAPK_3DE4D1EF_8043_4023_AFE4_7BEF2D39376C", "Model"))
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.13.: Testfall P-1

Testfall P-2a	
Methode	IParse.**Parse**(IList<string> eingabeDaten)
Eingabe	**eingabeDaten** = null
Erwartetes Ergebnis	null
Verifizierung	Überprüfung des Ergebnisses auf *null*.

Tabelle D.14.: Testfall P-2a

Testfall P-2b	
Methode	IParse.**Parse**(IList<string> eingabeDaten)
Eingabe	**eingabeDaten** = new List<string>()
Erwartetes Ergebnis	null
Verifizierung	Überprüfung des Ergebnisses auf *null*.

Tabelle D.15.: Testfall P-2b

Testfall O-1	
Methode	IOptimiere.**Optimiere**(BaG.Modellbaum.Modellbaum modellbaum)

Eingabe	Modellbaum **modellbaum** = new Modellbaum("Modellbaumname");
	PackageKnoten package = new PackageKnoten(ËinePackageId", "Model");
	KlasseKnoten klasse = new KlasseKnoten(ËineKlasse", ËineKlassenId", "public", package);
	package.Klassen.Add(klasse);
	KomponenteKnoten komponente = new KomponenteKnoten(ËineKomponente", ËineKomponentenId", "public", package);
	PartKnoten part = new PartKnoten(ËinPart", ËinePartId", "public", komponente);
	part.TypId = ËineKlassenId";
	komponente.Parts.Add(part);
	package.Komponenten.Add(komponente);
	modellbaum.Packages.Add(package);

Tabelle D.16 – Fortsetzung auf der nächsten Seite

Tabelle D.16 – Fortsetzung

Testfall O-1	

Erwartetes Ergebnis	Modellbaum expected = new Modellbaum("Modellbaumname");
	PackageKnoten expectedPackage = new PackageKnoten(ËinePackageId", "Model");
	KlasseKnoten expectedKlasse = new KlasseKnoten(ËineKlasse", ËineKlassenId", "public", expectedPackage);
	expectedPackage.Klassen.Add(expectedKlasse);
	KomponenteKnoten expectedKomponente = new KomponenteKnoten(ËineKomponente", ËineKomponentenId", "public", expectedPackage);
	PartKnoten expectedPart = new PartKnoten(ËinPart", ËinePartId", "public", expectedKomponente);
	expectedPart.TypId = ËineKlasse";
	expectedPart.Typ = expectedKlasse;
	expectedKomponente.Parts.Add(expectedPart);
	expectedPackage.Komponenten.Add(expectedKomponente);
	expected.Packages.Add(expectedPackage);

Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.16.: Testfall O-1

Testfall O-2	
Methode	IOptimiere.**Optimiere**(BaG.Modellbaum.Modellbaum modellbaum)
Eingabe	**modellbaum** = null
Erwartetes Ergebnis	null
Verifizierung	Überprüfung des Ergebnisses auf *null*.

Tabelle D.17.: Testfall O-2

Testfall V-1	
Methode	IValidiere.**Validiere**(Modellbaum.Modellbaum modellbaum)

Tabelle D.18 – Fortsetzung auf der nächsten Seite

Tabelle D.18 – Fortsetzung

Testfall V-1	

Eingabe	Modellbaum **modellbaum** = new Modellbaum("Modellbaumname");
	PackageKnoten package = new PackageKnoten(ËinePackageId", "Model");
	KlasseKnoten klasse = new KlasseKnoten(ËineKlasse", ËineKlassenId", "public", package);
	package.Klassen.Add(klasse);
	KomponenteKnoten komponente = new KomponenteKnoten(ËineKomponente", ËineKomponentenId", "public", package);
	PartKnoten part = new PartKnoten(ËinPart", ËinePartId", "public", komponente);
	part.TypId = ËineKlasse";
	part.Typ = klasse;
	komponente.Parts.Add(part);
	package.Komponenten.Add(komponente);
	modellbaum.Packages.Add(package);

Erwartetes Ergebnis	Validierungsergebnis: *true*,
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.18.: Testfall V-1

Testfall V-2	
Methode	IValidiere.**Validiere**(Modellbaum.Modellbaum modellbaum)
Eingabe	**modellbaum** = null
Erwartetes Ergebnis	Validierungsergebnis: *false*, Der Modellbaum darf nicht null sein!
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.19.: Testfall V-2

Testfall G-1	
Methode	IGeneriere.**Generiere**(Modellbaum.Modellbaum modellbaum, string ausgabePfad, Programmiersprache sprache)

Tabelle D.20 – Fortsetzung auf der nächsten Seite

Tabelle D.20 – Fortsetzung

Testfall G-1	
Eingabe	Modellbaum **modellbaum** = new Modellbaum("ModellbaumGeneratortest"); PackageKnoten package = new PackageKnoten("PackageId", "Package"); KomponenteKnoten komponnte = new KomponenteKnoten("Komponente", "KomponenteId", "public", package); package.Komponenten.Add(komponnte); modellbaum.Packages.Add(package); **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	new Dictionary<string, string>() mit Tupel: Key: `basepath\Unittests\ressources\testoutput\Komponente\Komponente.cs` Value: *// Dieser Code wurde durch ein Tool generiert.\r\n // Manuelle Änderungen werden bei erneuter Generierung überschrieben! \r\nusing System; \r\n\r\nnamespace Komponente \r\n{ \r\n\tpublic class Komponente \t\r\n\t{\r\n\t\t\r\n\t\t\t public Komponente(){ }\r\n\t\t\r\n\t \tpublic static Komponente createKomponente()\r\n\t\t{\r\n\t\t\t return new Komponente();\r\n\t\t}\r\n\t\t\r\n\t\t\r\n\t\r\n\t\t\r\n\t\r\n }\r\n}*
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.20.: Testfall G-1

Testfall G-2	
Methode	IGeneriere.**Generiere**(Modellbaum.Modellbaum modellbaum, string ausgabePfad, Programmiersprache sprache)
Eingabe	**modellbaum** = null **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	new Dictionary<string, string>()
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.21.: Testfall G-2

Testfall G-3a	
Methode	IGeneriere.**Generiere**(Modellbaum.Modellbaum modellbaum, string ausgabePfad, Programmiersprache sprache)
Eingabe	**modellbaum** = new Modellbaum("ModellbaumGeneratorTest") **ausgabePfad** = null **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	new Dictionary<string, string>()
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.22.: Testfall G-3a

Testfall G-3a	
Methode	IGeneriere.**Generiere**(Modellbaum.Modellbaum modellbaum, string ausgabePfad, Programmiersprache sprache)
Eingabe	**modellbaum** = new Modellbaum("ModellbaumGeneratorTest") **ausgabePfad** = ““ **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	new Dictionary<string, string>()
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.23.: Testfall G-3b

Testfall G-4	
Methode	IGeneriere.**Generiere**(Modellbaum.Modellbaum modellbaum, string ausgabePfad, Programmiersprache sprache)
Eingabe	Modellbaum **modellbaum** = new Modellbaum("ModellbaumGeneratortest") **ausgabePfad** = `basepath\Unittests\ressources\testoutput` **sprache** = Programmiersprache.Java
Erwartetes Ergebnis	new Dictionary<string, string>()
Verifizierung	Vergleich des erwarteten Ergebnisses mit der tatsächlichen Ausgabe.

Tabelle D.24.: Testfall G-4

D.2. Funktionaler Test

Die funktionalen Tests lesen Dateien ein, generieren Code aus diesen und speichern die generierten Dateien an einem anderen Ort wieder ab. Diese generierten Dateien werden daraufhin nochmals eingelesen und mit andenen, ebenfalls eingelesenen Dateien verglichen. Es gibt drei Basisordner, in denen die Eingaben, erwarteten Ausgaben und tatsächlichen Ausgaben liegen:

basisEingabePfad `basepath\FunktionaleTests\ressources\input`

basisExpectedPfad `basepath\FunktionaleTests\ressources\expected`

basisAusgabePfad `basepath\FunktionaleTests\ressources\output`

Der *basepath* enthält, wie bereits bei den Unittests (vgl. Abschnitt D.1) den Pfad zu der ausgeführten Generatoranwendung.

Die Verifizierung der Tests beruht, soweit nicht anders angegeben, stets auf dem Vergleich der eingelesenen erwarteten Ergebnisse aus dem Verzeichnis *basisExpectedPfad* und der eingelesenen tatsächlichen Ergebnisse aus dem Verzeichnis *basisAusgabePfad*. Da dies für alle nachfolgenden Tests gilt, ist die Verifizierung nicht mehr für jeden Testfall einzeln aufgeführt.

Die Nummerierung der Testfälle bezieht sich auf den getesteten Diagrammeelemente. Es wird versucht, alle Testfälle eines Elements zu gruppieren, indem dieses eine Nummer, beispielsweise „F-1“ für angebotene Schnittstellen, erhält und dann pro Testfall, welcher einen Fehler in diesem Knoten testet einen Buchstaben an die Nummer anhängt. So erhält der Testfall, welcher den Parser mit einer angebotenen Schnittstelle testet und einen Fehler werfen soll, die Nummer „F-1a“. Die Testfallnummer ohne Buchstaben, beispielsweise F-1 bezieht sich jeweils auf den positiven Test, welcher keinen Fehler produzieren soll.

Testfall F-1	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\AngeboteneSchnittstelleKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\AngeboteneSchnittstelle` **sprache** = Programmiersprache.CSharp

Tabelle D.25 – Fortsetzung auf der nächsten Seite

Tabelle D.25 – Fortsetzung

Testfall F-1	
Erwartetes Ergebnis	2 generierte Dateien: - `basisExpectedPfad\AngeboteneSchnittstelle\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\AngeboteneSchnittstelle\Transportport.cs`

Tabelle D.25.: Testfall F-1

Testfall F-2	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\BenoetigteSchnittstelleKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\BenoetigteSchnittstelle` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	2 generierte Dateien: - `basisExpectedPfad\BenoetigteSchnittstelle\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\BenoetigteSchnittstelle\Enumport.cs`

Tabelle D.26.: Testfall F-2

Testfall F-3	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\DatentypKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Datentyp` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	2 generierte Dateien: - `basisExpectedPfad\Datentyp\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Datentyp\Transportmittelkomponente\Datentypen\TransportmittelDatentyp.cs`

Tabelle D.27 – Fortsetzung auf der nächsten Seite

Tabelle D.27 – Fortsetzung

Testfall F-3

Tabelle D.27.: Testfall F-3

Testfall F-4	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\DelegationKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Delegation` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	3 generierte Dateien: - `basisExpectedPfad\Delegation\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Delegation\Transportmittelkomponente\AnwendungsfallTransportZuordnung.cs` - `basisExpectedPfad\Delegation\Transportport.cs`

Tabelle D.28.: Testfall F-4

Testfall F-5	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\EnumerationKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Enumeration` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	2 generierte Dateien: - `basisExpectedPfad\Enumeration\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Enumeration\Transportmittelkomponente\Enumerationen\FahrzeugtypDatentyp.cs`

Tabelle D.29.: Testfall F-5

Testfall F-6	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\EnumerationLiteralKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\EnumerationLiteral` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\EnumerationLiteral\FahrzeugtypDatentyp.cs`

Tabelle D.30.: Testfall F-6

Testfall F-7	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\InstantiateKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Instantiate` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	3 generierte Dateien: - `basisExpectedPfad\Instantiate\Transportmittelkomponente\Transportmittel.cs` - `basisExpectedPfad\Instantiate\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Instantiate\Transportmittelkomponente\Datentypen\TransportmittelDatentyp.cs`

Tabelle D.31.: Testfall F-7

Testfall F-8	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KlasseKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Klasse` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\Klasse\AnwendungsfallTransportZuordnung.cs`

Tabelle D.32.: Testfall F-8

Testfall F-9	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KomponenteKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Komponente` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\Komponente\Transportmittelkomponente\Transportmittelkomponente.cs`

Tabelle D.33.: Testfall F-9

Testfall F-10	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KompositionKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Komposition` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	3 generierte Dateien: - `basisExpectedPfad\Komposition\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Komposition\Transportmittelkomponente\Transportmittelverwalter.cs` - `basisExpectedPfad\Komposition\Transportmittelkomponente\AnwendungsfallTransportZuordnung.cs`

Tabelle D.34.: Testfall F-10

Testfall F-11	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KonnektorKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Konnektor` **sprache** = Programmiersprache.CSharp

Tabelle D.35 – Fortsetzung auf der nächsten Seite

Tabelle D.35 – Fortsetzung

Testfall F-11	
Erwartetes Ergebnis	3 generierte Dateien: - `basisExpectedPfad\Konnektor\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Konnektor\Transportmittelkomponente\Transportmittelverwalter.cs` - `basisExpectedPfad\Konnektor\Transportmittelkomponente\AnwendungsfallTransportZuordnung.cs`

Tabelle D.35.: Testfall F-11

Testfall F-12	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\PackageKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Package` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	0 generierte Dateien

Tabelle D.36.: Testfall F-12

Testfall F-13	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\PartKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Part` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	3 generierte Dateien: - `basisExpectedPfad\Part\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Part\Transportmittelkomponente\AnwendungsfallTransportZuordnung.cs`

Tabelle D.37.: Testfall F-13

Testfall F-14	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\PortKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Port` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\Port\Transportmittelkomponente\Transportmittelkomponente.cs`

Tabelle D.38.: Testfall F-14

Testfall F-15	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\SchnittstelleRealisierungKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\SchnittstelleRealisierung` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\SchnittstelleRealisierung\AnwendungsfallTransportZuordnung.cs`

Tabelle D.39.: Testfall F-15

Testfall F-16	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\UsageKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\Usage` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	3 generierte Dateien: - `basisExpectedPfad\Usage\Transportmittelkomponente\Transportmittelkomponente.cs` - `basisExpectedPfad\Usage\Transportmittelkomponente\Transportmittel.cs` - `basisExpectedPfad\Usage\Transportmittelkomponente\AnwendungsfallTransportZuordnung.cs`

Tabelle D.40.: Testfall F-16

Testfall F-17	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\VerschachtelteKlasseKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\VerschachtelteKlasse` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\VerschachtelteKlasse\AnwendungsfallTransportZuordnung.cs`

Tabelle D.41.: Testfall F-17

Testfall F-18	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\VerschachtelteKomponenteKnotenTrue.xml` **ausgabePfad** = `basisAusgabePfad\VerschachtelteKomponente` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	1 generierte Datei: - `basisExpectedPfad\VerschachtelteKomponente\Transportmittelkomponente\Transportmittelkomponente.cs`

Tabelle D.42.: Testfall F-18

Testfall F-1a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\AngeboteneSchnittstelleKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\AngeboteneSchnittstelle` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.43.: Testfall F-1a

Testfall F-2a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\BenoetigteSchnittstelleKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\BenoetigteSchnittstelle` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.44.: Testfall F-2a

Testfall F-3a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\DatentypKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Datentyp` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.45.: Testfall F-3a

Testfall F-4a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\DelegationKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Delegation` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.46.: Testfall F-4a

Testfall F-5a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\EnumerationKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Enumeration` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.47.: Testfall F-5a

Testfall F-6a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\EnumerationLiteralKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\EnumerationLiteral` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.48.: Testfall F-6a

Testfall F-7a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\InstantiateKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Instantiate` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.49.: Testfall F-7a

Testfall F-8a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KlasseKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Klasse` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.50.: Testfall F-8a

Testfall F-9a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KomponenteKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Komponente` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.51.: Testfall F-9a

Testfall F-10a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KompositionKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Komposition` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.52.: Testfall F-10a

Testfall F-11a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\KonnektorKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Konnektor` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.53.: Testfall F-11a

Testfall F-12a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\PackageKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Package` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.54.: Testfall F-12a

Testfall F-13a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\PartKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Part` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.55.: Testfall F-13a

Testfall F-14a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\PortKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Port` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.56.: Testfall F-14a

Testfall F-15a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\SchnittstelleRealisierungKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\SchnittstelleRealisierung` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.57.: Testfall F-15a

Testfall F-16a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\UsageKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\Usage` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.58.: Testfall F-16a

Testfall F-17a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\VerschachtelteKlasseKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\VerschachtelteKlasse` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.59.: Testfall F-17a

Testfall F-18a	
Eingabe	**eingabePfade** = new List<string>() mit einem Eintrag: `basisEingabePfad\VerschachtelteKomponenteKnotenFalse.xml` **ausgabePfad** = `basisAusgabePfad\VerschachtelteKomponente` **sprache** = Programmiersprache.CSharp
Erwartetes Ergebnis	UnbekanntesElementException
Verifizierung	Abfangen der Exception

Tabelle D.60.: Testfall F-18a

E. Inhalt der beigefügten CD

Auf der beiliegenden CD-Rom befinden sich folgende Daten:

- Der im Rahmen dieser Arbeit erstellte Quellcode für den Prototypen des Codegenerators.
- Die Diagramme des Fallbeispiels im PDF-Format und deren XML-Dokumente.
- Diese Ausarbeitung im PDF-Format

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. August 2011	
Ort, Datum	Unterschrift