Bachelor-Thesis

im Studiengang “Angewandte Informatik”


Hochschule Offenburg

Generierung von Java-Quellcode aus UML-Zustandsdiagrammen

Autor:
Jan Klass
Bellenwaldweg 1
77652 Offenburg

Betreuer:
Prof. Dr. Klaus Dorer

Zweitbetreuer:
Prof. Dr. Daniel Fischer

Bearbeitungszeitraum: 01.09.2010 bis 28.02.2011

Kurzfassung

Diese Bachelorarbeit erläutert zunächst wieso UML im Allgemeinen, und Zustandsdiagramme im Speziellen, so ein wichtiges Werkzeug in der Software-Entwicklung sind. Anschließend werden die UML-, XMI- und MOF-Standards vorgestellt und deren Verbindungen zueinander beschrieben. Darüber hinaus werden die Ergebnisse einer Recherche nach Generatoren erläutert. Die Generatoren sollen hierbei aus UML-Zustandsmodellen, die im XMI-Format gespeichert wurden, Quellcode erzeugen. Anschließend wird das Software-Projekt vorgestellt, welches im Rahmen dieser Bachelorarbeit durchgeführt wurde. Aus dem Projekt geht ein Quellcodegenerator, mit den eben genannten Funktionen, als Ergebnis hervor. Die Bachelorarbeit begleitet hierbei die komplette Entwicklung. In der Analysephase wird auf die Bibliotheksevaluation, den XMI-Import und Template-Engines eingegangen. Außerdem werden verschiedene Ansätze zum Mergen des generierten Quellcodes vorgestellt und das Markieren mit Javadoc-Tags als hierfür am geeignetsten ermittelt. Anschließend werden die in UML möglichen, hierarchischen Zustände vorgestellt und es wird elaboriert, wie man diese im Programm ebenfalls umsetzen könnte. Für Zustandsaktivitäten wird eine alternative Implementierungsmöglichkeit dargestellt und bewertet.

In der, auf die Analyse folgenden, Designphase wird zunächst auf die verwendete Wizard-Bibliothek CJWizard eingegangen. Anschließend wird der Gedankengang bei der Festlegung der Datei- und Paketstruktur des Projektes dargestellt. Im darauf folgenden Kapitel wird erläutert, wie eine Template-Engine in das Projekt eingeplant wird und was dies für Vorteile bringt.

In der Implementierungsphase wird erläutert, wie die zuvor vorgestellte Wizard-Bibliothek im Projekt konkret verwendet wird. Bezüglich des XMI-Imports werden die aufgetretenen Herausforderungen dargestellt, und wie diese bewältigt wurden. Anschließend wird auf das *Ant*-Buildsystem sowie die *Ivy*-Abhängigkeitsauflösung eingegangen und wie diese etwa zur Generierung von einer allein lauffähigen Jar-Datei verwendet werden.

Zum Ende steht als Ergebnis ein funktionstüchtiges und eigenständiges Programm, mit welchem man aus in XMI gespeicherten UML-Zustandsdiagrammen Java-Quellcode generieren kann. Einige Erweiterungsmöglichkeiten, wie das Projekt fortgeführt werden könnte, werden ebenfalls vorgestellt.

Vorwort

Ich möchte mich herzlich bei meinen Betreuern Prof. Dr. Klaus Dorer und Prof. Dr. Daniel Fischer bedanken. Nicht nur bei dieser Bachelorarbeit, sondern auch im Verlauf des Studiums waren sie immer hilfsbereit. Ich möchte mich außerdem bei allen Dozenten bedanken, deren Vorlesungen ich besuchen durfte und mit deren Hilfe ich mein Wissen verbreitern und vertiefen konnte. Dank gilt auch meiner Familie und Verwandtschaft, die mich immer unterstützt haben. Im Rahmen dieser Bachelorarbeit gilt dabei besonderer Dank Kerstin Bülow für das Korrekturlesen dieser.

Vielen Dank.

Hiermit versichere ich eidesstattlich, dass die vorliegende Bachelor-Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

_____

Inhaltsverzeichnis

1. Einleitung

Zustandsdiagramme sind ein wichtiges Werkzeug zur Beschreibung und Modellierung von Software. Insbesondere im Bereich der Software-Entwicklung von eingebetteten Systemen ist das Modellieren nach Objekten, welche verschiedene Zustände annehmen und wechseln können, eine sehr gängige Praxis. Aber auch bei der Entwicklung von Desktop-Anwendungen können Zustandsdiagramme und die Modellierung und Planung in Zuständen sehr hilfreich sein. Der UML-Standard ist die gängigste Modellierungssprache der Software-Entwicklung und definiert unter anderem auch Repräsentationsmöglichkeiten für solche Zustandsmaschinen. XMI ist ein Standard der definiert, wie UML-Modelle serialisiert werden können.

Bei der Umsetzung der Modelle zu Quellcode ergibt sich jedoch ein Bruch. Nach der Planung in Modellen und Diagrammen müssen diese abstrakten Repräsentationen zu Quellcode, geschrieben in einer Programmiersprache, umgesetzt werden. Dies ist mit Aufwand verbunden und ergibt die Gefahr von Fehlern, die bei der Umsetzung eventuell gemacht und übersehen werden. Seien es Programm- oder Umsetzungsfehler (im Quellcode ist beispielsweise nicht genau das vorhanden, was im Modell definiert wurde). Hauptsächlich aus diesen beiden Gründen heraus, für Zeitersparnis und zur Fehlervermeidung, versucht man häufig, mittels Quellcode-Generatoren, aus Modellen verwendbaren Quellcode zu generieren.

Diese Bachelor-Thesis beschäftigt sich mit der Quellcode-Generierung aus UML-Zustandsdiagrammen. Hierfür wird zunächst ein Überblick über die aktuelle Situation, mit vorhandenen Generatoren, beleuchtet und anschließend ein Software-Projekt vorgestellt, in dem ein neuer Quellcodegenerator im Rahmen dieser Bachelorarbeit umgesetzt wird. Dieser liest UML-Zustandsdiagramme, die in das XMI-Dateiformat exportiert wurden, ein und generiert aus diesen Java-Quellcode.

2. Hintergrund

In diesem Kapitel wird zunächst der Hintergrund dieser Bachelorarbeit genauer beleuchtet. Das heißt. welche Standards wichtig sind und was in diesen Standards definiert wird. Anschließend werden die Ergebnisse der Recherche nach vorhandenen Quellcodegeneratoren vorgestellt.

Der UML-Standard 2.3 definiert die Modellierungssprache *UML* als objektorientierte Klassenstruktur und als Instanziierung einer *Meta-Object Facility* (MOF). MOF wiederum ist ein Standard zur Beschreibung von Metamodellen. Es ist in zwei Teilen definiert, der EMOF (Essential MOF) und der CMOF (Complete MOF). *XMI* ist ein Standard zur Serialisierung von MOF-Metamodellen und -Modellinstanzen. All diese genannten Standards sind von der *Object Management Group* (OMG), einem Konsortium aus mehreren Firmen, spezifiziert und veröffentlicht. Tabelle 1 zeigt die Metamodell- und Modellstruktur von MOF und UML. Ein Modell kann immer auch ein Metamodell für ein weiteres Modell sein. MOF etwa ist ein Metamodell für das Metamodell UML. Deshalb auch die Bezeichnung „Meta-Metamodell" für MOF. MOF ist über sich selbst definiert, ist also auch ein MOF-Modell.

Meta-Metamodell	MOF
Metamodell	UML
Modell	konkrete UML-Modelle/-Diagramme
Modelliertes	Quellcode

Tabelle 1: Einbettung von UML in die hierarchische Modell- und Metamodell-Struktur bei typischen Software-Projekten

2.1 UML-Standard

Seit UML 2.0 ist UML in zwei Standards definiert. Der *UML Superstructure* Standard ([UMLS10]) basiert auf dem *UML Infrastructure* Standard ([UMLI10]) und erweitert dessen Modellstruktur.

UML ist in sogenannten *language units*, also Spracheinheiten, definiert. Diese Strukturierung des Standards erlaubt es, sich nur bestimmte Teilaspekte von UML ansehen zu können, an denen gerade Interesse besteht. Diese Einheiten bauen dann inkrementell auf anderen Einheiten auf,

wodurch die Modellierungsfähigkeiten im Verlauf des Standards fortlaufend zunehmen.

Der UML Standard selbst nennt folgende Designaspekte als Richtprinzipien bei der Erstellung des selbigen (vgl. [UMLI10]):

- Modularität
- Schichtung
- Partitionierung
- Erweiterbarkeit
- Wiederverwendbarkeit

Der UML Infrastructure Standard definiert eine grundlegende Infrastruktur, mit derer sowohl das weitere UML (im UML Superstructure Standard), wie auch MOF und andere Metamodelle definiert werden.

Definiert wird das Paket *InfrastructureLibrary*, welches die Pakete *Core* und *Profiles* enthält. Im Paket *Core* sind die bereits erwähnten grundlegenden Klassen und Definitionen spezifiziert. Über die so genannten Profile kann diese Grund-Infrastruktur, dieses Metamodell, dann erweitert werden. Konkret verwenden, sowohl der MOF-Standard wie auch der UML-Superstructure-Standard das Core-Paket als Basis. Das UML-Modell basiert dann auf dem Metamodell MOF.

Der Zweck von MOF ist, dass es als Metamodell beispielsweise definiert, wie ein Modell in das XMI-Format umgewandelt werden kann. Dies wird im XMI-Standard für MOF bestimmt und kann auch auf UML angewandt werden, da UML auf MOF aufbaut. Ein weiteres Beispiel ist, dass MOF reflektive Interfaces definiert, welche eine Introspektion sowohl für MOF, wie auch für auf diesem basierenden Modellierungssprachen erlauben.

Im UML Superstructure Standard wird dann schließlich das UML-Paket spezifiziert. In ihm enthalten sind weitere Pakete, die sich etwa mit der Struktur- oder Verhaltensmodellierung beschäftigen. Das zentralste Paket dabei ist das *Kernel*-Paket, welches in allen anderen Paketen verwendet wird.

Zwischen Diagramm und Modell ist zu unterscheiden. Bei der UML-Modellierung von Software repräsentiert das Modell den Quellcode, die Architektur, das Projekt. Optimalerweise gibt es daher nur ein in UML spezifiziertes Modell für ein Software Projekt. Diagramme sind dann verschiedene Darstellungen dieses Modells. So wären zwei Klassendiagramme zu einem Modell

denkbar, bei dem das eine die Klassen eher abstrakt, ohne Attribute und Methoden darstellt, das zweite aber im Detail mit Attributen und Methoden. Ersteres wäre etwa eher für die Analyse geeignet, letzteres für die Designphase. Oder ein Zustandsdiagramm zeigt Zustände, die eine bestimmte Klasse aus dem Modell, beziehungsweise ein Objekt dieser Klasse des Modells, annehmen kann.

2.2 XMI-Format

Wie bereits dargestellt wurde ist XMI („XML Metadata Interchange") ein von der OMG spezifiziertes Format und Standard zur Serialisierung von Modellen und Modellinformationen. Der Standard legt fest, wie MOF-Modelle, und damit auch alle auf MOF basierenden Modelle, in das XMI-Format serialisiert werden können. Umgekehrt lassen sich die serialisierten Daten auch wieder deserialisieren. Die serialisierten Daten sind XML-basiert. XMI ist also vor allem ein Format für die Übertragung und zum plattformunabhängigen und anbieterneutralen Speichern von Metamodellen und Modellen.

Eine Anwendung von XMI ist der Export von Modellen aus UML-Modellierungswerkzeugen in XMI-Dateien. Dies ermöglicht, sofern die Werkzeuge XMI unterstützen, eine von einzelnen Werkzeugen unabhängige Erstellung, Modifizierung und Begutachtung von Software-Modellen.

Die UML-XMI-Dateien speichern zunächst nur Modelle, aber keine Diagramme. Das heißt sie speichern die Repräsentation, aber nicht wie die Elemente im Diagramm, also einer Ansicht des Modells, angeordnet sind und welche Modellelemente angezeigt werden und welche nicht. Ein Speichern auch dieser Daten kann jedoch über Erweiterungen der XMI-Daten erreicht werden. So fügt beispielsweise das Modellierungswerkzeug Visual Paradigm diese Zusatzinformationen ein. Bei einem Import der XMI-Datei können dann nicht nur die Modelle, sondern auch die Diagramme wiederhergestellt und angezeigt werden.

2.3 Vorhandene Codegeneratoren

Dieses Kapitel behandelt die Recherche und Rechercheergebnisse der Evaluierung von UML-Zustandsdiagramm-Quellcodegeneratoren. Besondere Aufmerksamkeit wurde, neben dem Design des generierten Quellcodes, der Benutzerführung des Tools und der Möglichkeit eines XMI-Imports gewidmet.

Visual Paradigm for UML bietet volle UML2 Unterstützung und zahlreiche begleitende Funktionen. Es wird in verschiedenen Editionen angeboten, welche zu unterschiedlichen Preisen verfügbar sind und einen angepassten Funktionsumfang bieten. Zur Zeit (Stand: Okt. 2010) ist ein Import von und Export nach XMI ab der dritten (gezählt von der günstigsten, funktionsärmsten) Edition, der „Modeler-Edition", möglich. Die Generierung von Quellcode aus Zustandsdiagrammen ist ab der fünften Edition, der „Professional Edition", möglich. Insgesamt gibt es 6 Editionen. Die Benutzerführung ist sehr gut umgesetzt und macht das Verwalten von Diagrammen und Modellen einfach. Die Möglichkeiten und die in Diagrammen festlegbaren Eigenschaften machen durchweg den Eindruck, nahe am UML-Standard zu sein und diesen durchweg konform umzusetzen. Bei dem für ein Zustandsmodell generierten Quellcode wird eine Klassenstruktur erstellt, welche eine ausführbare Zustandsmaschine umsetzt. Abbildung 2

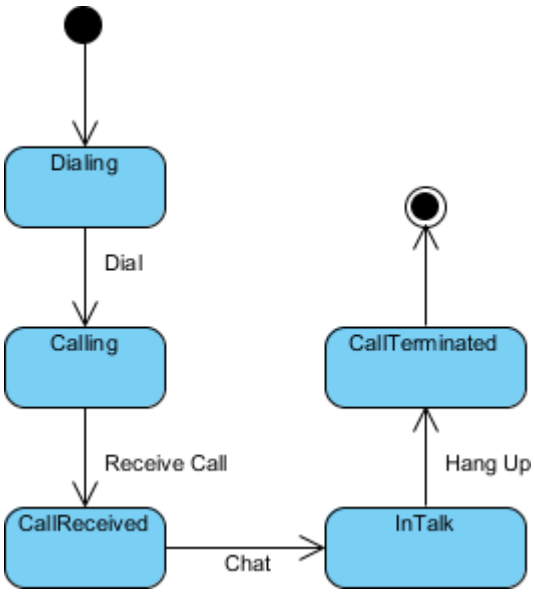

Abbildung 1: Beispielhaftes Zustandsdiagramm, aus dem Quellcode generiert wird

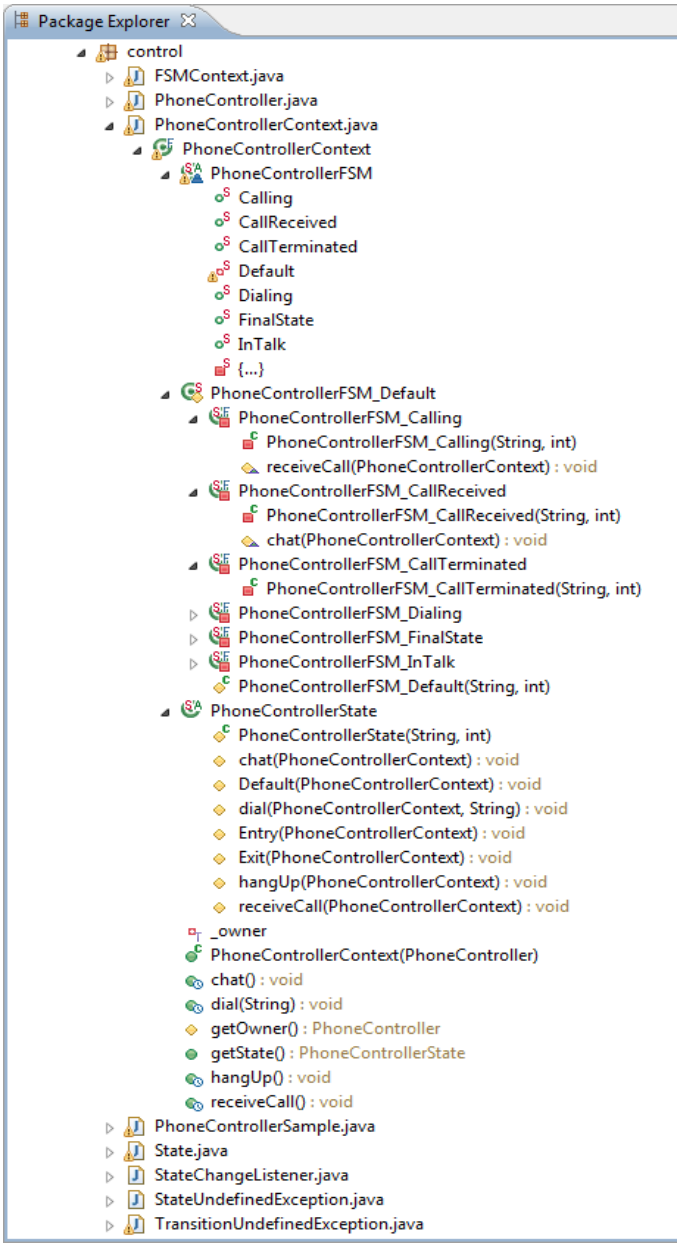

Abbildung 2: Klassen- und Methodenstruktur des von Visual Paradigm erstellten Quellcodes

zeigt die Klassen- und Methodenstruktur solch einer generierten Zustandsmaschine, welche aus einem in Abbildung 1 dargestellten Zustandsdiagramm generiert wurde. Die Zustände werden als innere Klassen in 2 Ebenen konkretisiert. Dafür ist eine Basisklasse für Zustände und eine für die Zustandsmaschine definiert. Beide sind innere Klassen, abstrakt und statisch. Die Trigger für die Zustandsübergänge werden als Methoden umgesetzt, die Zustandsaktivitäten ebenfalls. Die abstrakte, statische Zustandsklasse definiert die Methodenköpfe der Trigger- und Aktivitätsmethoden. In einer weiteren statischen Klasse werden die einzelnen Zustände mit ihren Trigger-Methoden und Aktivitätsmethoden als innere Klassen implementiert. Der Quellcode für den Aufruf der Aktivitätsmethoden und den Zustandswechsel wird redundant für jeden Zustand implementiert. In der oben erwähnten, abstrakten, statischen Zustandsmaschinenklasse werden die Zustände statisch initialisiert. Wird in einem Zustand eine Trigger-Methode aufgerufen, welche einen für diesen Zustand nicht definierten Trigger repräsentiert, so wird eine *TransitionUndefinedException* geworfen. Als Bewertung lässt sich folgendes festhalten: Obwohl der Quellcode funktioniert und über eine Kontextklasse sogar einen Zustandsstack verwaltet und Zustandswechsel-Listener unterstützt, ist er für die eigentlichen Zustände und Zustands-übergänge eher unübersichtlich und zu komplex. Ein Entwickler, welcher Quellcode mit einem guten, schnell erkenn- und durchschaubaren Design erwartet, welches auch leicht erweitert werden kann, wird hier enttäuscht.

Die Modeling-Suite *Enterprise Architect* von Sparx Systems kann Quellcode in mehreren Sprachen, auch aus Zustandsdiagrammen, generieren. Hierzu wird ein Zustandsdiagramm einer Klasse hierarchisch untergeordnet erstellt. Dies bedeutet im Umkehrschluss jedoch auch, dass eine Klasse zwingend notwendig ist, um Quellcode für ein Zustandsdiagramm erzeugen zu können. Ein alleinstehendes Zustandsdiagramm ist nicht ausreichend. Abbildung 3 zeigt ein beispielhaftes, in Enterprise Architect erstelltes Zustandsdiagramm. Enterprise Architect verwendet für das Generieren die durch UML gegebenen Möglichkeiten, in Zuständen Aktivitäten (entry, do, exit) und Trigger angeben zu können (siehe [UMLS10]), und setzt diese in Methoden (Aktivitäten) und Enum-Werte (Trigger/Events) um. Die Zustände selbst werden ebenfalls als Enum-Werte umgesetzt.

Der aktuelle, sowie der nächste Zustand wird in der Klasse als Attribut gehalten. Aus jedem Zustand des Modells mit einer Aktion wird eine Methode der Klasse generiert, welche als Parameter einen Aktions-Enum-Wert erhält. In dieser Methode werden zunächst zustandsspezifische Methodenaufrufe durchgeführt und im Anschluss die Zustandsübergangslogik abgearbeitet. Hierbei werden gegebenen-falls die Guard-Bedingungen geprüft und anschließend der passende Übergang initia-lisiert, indem der nächste Zustand im Attribut als aktueller Zustand gesetzt wird. In der Methode, welche die Zustandsmaschine ausführt, werden dann die Methoden für die Eingangs- und Ausgangsaktivitäten (jene mit entsprechenden Aktions-Enum-Parametern) aufgerufen.

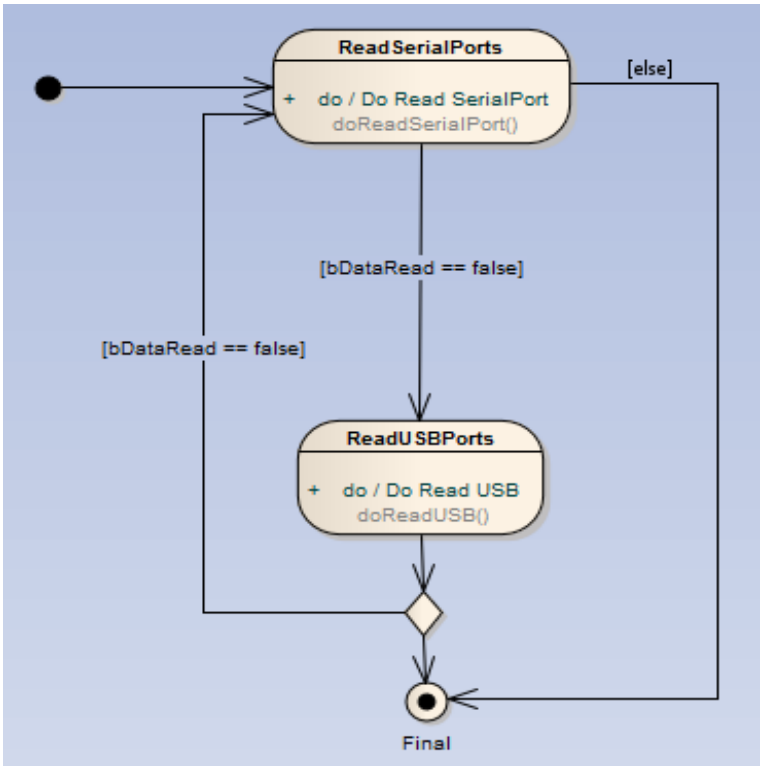

Abbildung 3: Beispiel Zustandsdiagramm mit Zustandsevents und internen Aktionen

Aufgrund der Beschränkungen, wann Quellcode aus einem Zustandsdiagramm generiert werden kann, und aufgrund der schlechten Dokumentation dieser Funktionalität in Verbindung mit einer schlechten Benutzerführung im Programm, ist Enterprise Architect für diesen Zwecke eher nicht geeignet. Der generierte Quellcode zeigt sich dann, durch ein starres Design mit Enum Werten für die Zustände, als schwer erweiterbar.

Das kommerzielle UML-Werkzeug *WithClass* von MicroGold erlaubt die Generierung von Quellcode aus Klassen- und Zustandsdiagrammen. Der Quellcode wird hierbei durch ein einfaches Skript erstellt und kann, in einem gewissen Rahmen, etwa um den Code-Stil (Formatierung, Einrückung) zu ändern, angepasst werden. Der Java-Code für ein Zustandsdiagramm besteht am Ende jedoch nur aus einer einzigen Klasse. Diese Klasse hält als Attribut den aktuellen Zustand. Für den Zustandswechsel wird die *process*-Methode, mit dem

Namen eines Übergangs als String-Parameter, aufgerufen. Das Zustandsdiagramm wird also lediglich über eine Klasse, ein Zustandsattribut und eine Übergangsmethode realisiert. Die weiteren möglichen UML-Modellierungen, wie Trigger, Aktivitäten oder Effekte, werden nicht umgesetzt. Ein Verarbeiten von XMI-Dateien oder dem XMI-Format ist ebenfalls nicht möglich.

Die kommerzielle Anwendung *Artisan Studio* kann keinen Quellcode aus Zustandsdiagrammen erzeugen. Auf Rückfrage erklärte man, eine eben solche Funktion sei aber für die nächste Version 7.3 geplant (Stand: Oktober 2010). Diese soll Quellcode für Java 1.5 und höher generieren können.

3. Analyse

Aus dem im Rahmen dieser Bachelorarbeit durchgeführten Softwareprojekt soll ein Tool für die Java-Quellcodegenerierung, aus UML-Zustandsdiagrammen, hervorgehen. Dieses Tool wurde *StateXMI2Java* genannt. Das Zustandsdiagramm, beziehungsweise das dem Diagramm zugrunde liegende Modell, liegt dem Generator hierbei stets im XMI-Format als Datei vor.

Zu den Anforderungen an das Programm gehört ein Grunddesign des generierten Quellcodes. Dieser Zielcode sich durch ein gutes, objekt-orientiertes Design auszeichnen und einfach verwendbar sein. Dies bringt die üblichen Vorteile von gut durchgeführter Objektorientierung mit sich, wie beispielsweise gute Anpassbarkeit und Erweiterbarkeit. So können Aktivitäten und Guards durch einfache Methodenüberschreibung angepasst werden. Ist die Methode bereits vorhanden, so muss nur deren Logik geändert werden. Eine Erweiterung um einen Zustand oder einen Zustandsübergang ist jeweils über das Hinzufügen einer weiteren Klasse möglich, welche von einer passenden Klasse erbt (von *ObjectStateConcrete* oder *StateTransitionConcrete*). Ein vereinfachtes Klassendiagramm des Zielcodes ist in Abbildung 4 zu sehen. Ein Diagramm mit mehr Details ist in Abbildung 5 dargestellt.

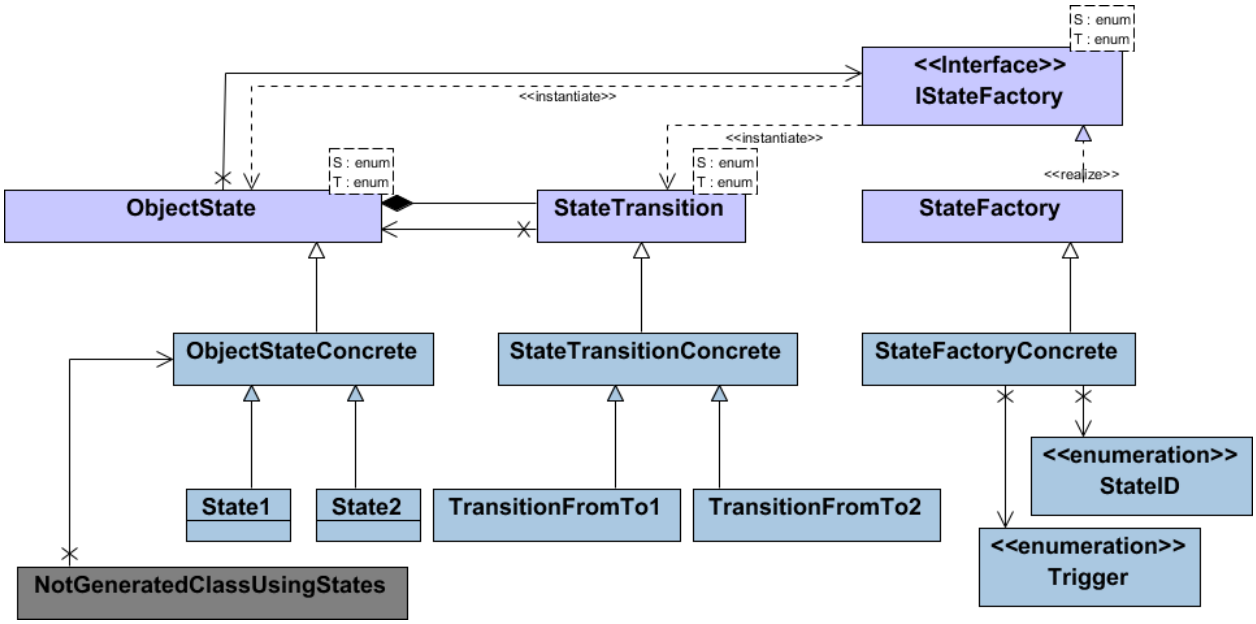

Abbildung 4: vereinfachtes, schematisches Klassendiagramm des Zielquellcodes

Bei der Generierung werden modellpezifische und modellunabhängige Klassen erstellt. In der

oberen Hälfte der Abbildung 4 befinden sich die modellunabhängigen Klassen. Hierzu gehören *ObjectState*, die einen Zustand repräsentiert, *StateTransition*, die einen Zustandsübergang repräsentiert, und *IStateFactory* sowie *StateFactory*. *StateFactory* implementiert dabei das *IStateFactory*-Interface und kennt und erzeugt als Fabrik die Zustände und Zustandsübergänge, sowie die Identifikation dieser Zustände und die Trigger als Enumeration.

In der unteren Hälfte befinden sich die modellspezifischen Klassen, die von den allgemeineren Klassen erben und diese spezialisieren. So enthalten die konkreten, von *StateTransition* erbenden Klassen gegebenenfalls eine überschriebene Guard-Methode, die den Guard des Übergangs realisiert und prüft. In den von *ObjectState* erbenden Klassen werden die Methoden für Aktivitäten beim Eintritt, beim Austritt und im Zustand gegebenenfalls überschrieben.

Die *StateFactory* wird ebenfalls konkretisiert. Sie enthält die Logik zur Erstellung sämtlicher konkreten Zustände und Zustandsübergänge, und definiert ZustandsIDs (*StateID*) und *Trigger* als Enumerationen. Die Enumerationen werden zum Identifizieren der Zustände, beziehungsweise Trigger verwendet. Ein Trigger-Enum-Wert wird später vom Entwickler verwendet, um aus einem Zustand heraus, über diesen Trigger, einen Zustandsübergang auszulösen.

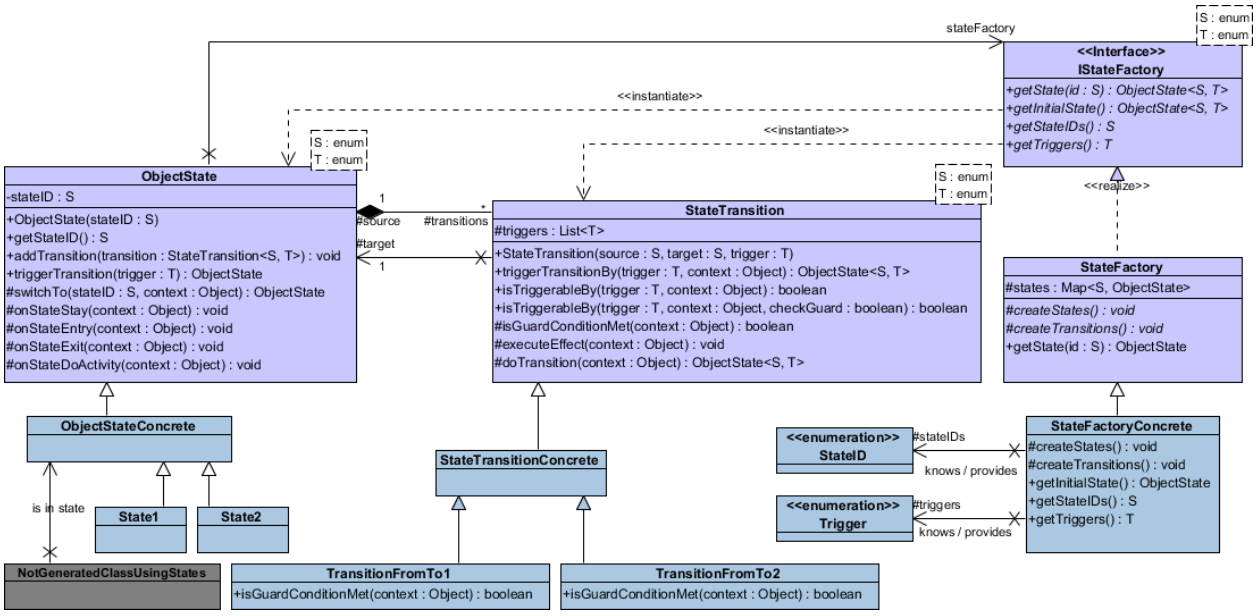

Abbildung 5: Detailliertes Klassendiagramm des generierten Quellcodes

In der unteren Hälfte findet sich außerdem die grau hinterlegte Klasse *NotGeneratedClassUsingStates*. Sie repräsentiert jene Klasse, die letztlich die Zustände

annehmen kann und den generierten Quellcode verwendet. Das Einfügen einer allgemeinen konkretisierten Klasse für Zustände und Übergänge (*ObjectStateConcrete* und *StateTransitionComplete*), von welcher dann die einzelnen entsprechenden Klassen erben, ermöglicht es dem Entwickler allgemeine, für das Projekt spezifische, aber auf alle Zustände oder Übergänge zutreffenden Methoden oder Attribute umzusetzen.

Diese vorgestellte Quellcode-Struktur soll aus einem UML-Zustandsdiagramm, welches in das XMI-Format in eine Datei serialisiert wurde, generiert werden. Zunächst sollen aus Visual Paradigm exportierte XMI-Dateien unterstützt werden. Aufgrund der Interoperabilität des XMI-Formats sollten anschließend aber auch XMI-Dateien anderer Modellierungswerkzeuge unterstützt werden.

3.1 XMI-Dateiimport

Für das Einlesen von XMI-Dateien ist eine Bibliothek wünschenswert, um den Implementierungsaufwand gering zu halten. Bereits bei der ausführlichen Recherche nach vorhandenen UML-Werkzeugen (siehe oben) fanden sich zwei Bibliotheken und eine Spezifikation mit solchen Funktionen.

Das *Java Metadata Interface* (JMI) ist ein im *Java Community Process* (JCP) entstandener Spezifikationsvorschlag („*Java Specification Request*") für das Erstellen, Speichern, Zugreifen auf und den Austausch von Metadaten. Die Metadaten basieren hierbei auf MOF. Für den Informationsaustausch verwendet JMI XMI. Die finale Version des Spezifikationsvorschlags ([JMI02]) wurde im Juni 2002 veröffentlicht.

JMI spezifiziert ein Interface. Daher sind verschiedene Implementierungen dieses Interfaces möglich. Die auf der von Sun gehosteten JMI-Webseite genannten Implementierungen dieser Spezifikation führen jedoch zu keiner verwendbaren Bibliothek. So führt der Weblink zur Referenz-Implementierung von Unisys auf eine nicht existierende Webseite.

Im Rahmen des Netbeans IDE Projektes wurde die quelloffene JMI-Implementierung *MDR* („Metadata Repository“) entwickelt. Diese wurde jedoch ab der Netbeans Version 6 wieder fallen gelassen und nicht mehr weiter entwickelt. Die letzte MDR-Version bietet keine UML 2 Unterstützung.

Eine andere Java-Bibliothek zum Verarbeiten von Metadaten und -modellen sind die *UCL MDA Tools*. Diese sind quelloffen und frei unter der Mozilla Public License 1.1 verfügbar. Sie wurden von der University College London entwickelt. Sie sollen, so eine Designentscheidung, möglichst standardkonform zu MOF, XMI, OCL2 und JMI sein. Auch diese Bibliothek unterstützt jedoch kein UML 2.

Im Rahmen des *Eclipse Modeling Framework* (EMF) Projektes wurde ebenfalls eine Bibliothek für die Verarbeitung von Metamodellen erstellt. Diese verwendet allerdings ein von MOF abgeleitetes Modell. Das abgeleitete Modell, beziehungsweise dessen Bibliothek, trägt die Bezeichnung *ECore*. Die Bibliothek für den XMI-Import hat außerdem einige Abhängigkeiten. So werden durch die Verwendung der Jar-Bibliothek 10 bis 15 weitere Jar-Bibliotheken aus dem EMF notwendig. Die Dokumentation, speziell für den Umgang mit XMI, ist nicht intuitiv und eher unzureichend ausgeführt.

Warum finden sich also keine ausgereiften XMI-Bibliotheken, wo XMI doch so einen wichtigen, plattformunabhängigen und anbieterneutralen Standard zur De- und Serialisierung von UML-Modellen darstellt, und UML die wichtigste Modellierungssprache in der Software-Entwicklung ist?

Dies mag mehrere Gründe haben. Die Firmen, die ausgereifte Modellierungstools, welche oft XMI-Funktionalität haben, anbieten und verkaufen, möchten diese Implementierung oft für sich alleine beanspruchen, um ihre Marktstellung zu behalten und nicht zu schwächen. Eine Offenlegung des Quellcodes, womöglich unter einer freien Lizenz, würde es Konkurrenten wesentlich einfacher machen, ebenfalls XMI-Unterstützung einzubauen und anzubieten. Dies kann ein wichtiges Verkaufsfeature sein. Aus Benutzersicht wäre gerade dies natürlich wünschenswert, dass man seine Modelle im XMI-Format, zwischen möglichst vielen Modellierungstools, austauschen kann. Für die Hersteller aber wiederum, aus genannten Gründen, ist dies nicht unbedingt wünschenswert. Eventuell auch in Hinblick auf die Kundenbindung scheuen sich Firmen Bibliotheken zu veröffentlichen, oder an quelloffenen

mitzuarbeiten. Hat man keine Möglichkeit seine Modelle in ein neues, anderes Modellierungstool zu übertragen, so bleibt einem gegebenenfalls keine Möglichkeit, als mit dem alten Tool weiter zu arbeiten. Auch im Bereich der FOSS (Free Open-Source Software) finden sich leider keine aktuellen und ausgereiften Bibliotheken. Lediglich ECore des EMFs scheint ausgereift, aber zu eng an das EMF gebunden (siehe Erläuterungen oben).

Aufgrund der fehlenden Möglichkeiten, eine vorhandene Bibliothek für das StateXMI2Java-Projekt zu verwenden, wird daher ein eigener XMI-Import implementiert. Da XMI auf XML basiert, lässt sich dies relativ einfach und mit etablierten XML-Frameworks bewerkstelligen.

Von den in Frage kommenden Java-XML-Frameworks wurde *JAXB* („Java Architecture for XML Binding“) gewählt. JAXB ist in der Java Standard Edition ab Version 6 bereits enthalten und erlaubt es, über Annotationen, ein Mapping zwischen XML und Java herzustellen. Genauer wird das Mapping zwischen den XML-Elementen und den Java-Klassen, -Attributen und -Methoden hergestellt.

3.2 Template-Engines

Zur Generierung des Quellcodes bietet es sich an, Templates zu verwenden. Eine Template-Engine erlaubt es, festgelegten Platzhaltern (Variablen) innerhalb eines Textgerüstes, Werte zuzuweisen. Somit kann angepasster, auf dem Gerüst basierender Text erzeugt werden. Der zu generierende Quellcode kann so in Dateien ausgelagert werden. Die für die konkrete Quellcodeerzeugung einmaligen Werte (Klassennamen, etc.) werden in den Templates durch Platzhalter ersetzt. Bei einer konkreten Quellcodeerzeugung werden die Platzhalter dann durch die einmaligen Werte ersetzt. Über die Template-Dateien kann dann auch ein etwa geltender Coding Style umgesetzt werden, oder das Grundgerüst der zu generierenden Klassen und des zu generierenden Quellcodes angepasst werden. Für die Quellcode-Generierung ist eine möglichst hohe Dichte an Templates wünschenswert, um eine möglichst hohe Anpassbarkeit des generierten Quellcodes zu gewährleisten, ohne das StateXMI2Java-Tool selbst ändern und neu kompilieren zu müssen.

Durch eine Recherche nach für das Projekt in Frage kommenden Template-Bibliotheken, stellten sich drei zur Auswahl: *Apache Velocity*, *FreeMarker* und *StringTemplate*.

Apache Velocity ist unter der Apache Software License veröffentlicht und wird unter dem Dach

der Apache Software Foundation entwickelt. Es ist minimalistisch in seinen Funktionen. Durch die große Community an Benutzern gibt es jedoch auch zahlreiche Erweiterungen, die zusätzliche Funktionalitäten ermöglichen.

FreeMarker steht unter einer BSD-artigen Lizenz und ist somit ebenfalls freie Software. Es hat keine so große Community, wie etwa Apache Velocity, verfügt aber von Haus aus über mehr Funktionen. Die Dokumentation von FreeMarker ist außerdem etwas eingängiger und übersichtlicher.

Die dritte genannte Template-Engine ist StringTemplate. Sie steht unter einer BSD-Lizenz und arbeitet ausschließlich mit Strings. StringTemplate ist die simpelste der drei Engines.

Letztlich wird für das StateXMI2Java-Projekt FreeMarker verwendet, hauptsächlich wegen den geringen Abhängigkeiten und der guten Dokumentation, die einen schnellen Einstieg erleichtern.

3.3 Wizard-Bibliothek

Für die Erstellung eines Wizards ist eine Wizard-Bibliothek wünschenswert. Diese soll Funktionen bereitstellen, um eine Wizard-Oberfläche und -Steuerung, sowie eine Datenhaltung, mit möglichst wenig Aufwand umsetzen zu können. Hierzu wurden einige Wizard-Bibliotheken evaluiert.

Als sehr gute und simple Lösung stellte sich CJWizard heraus. Die CJWizard-Bibliothek steht unter der Apache-Lizenz und hat keine weiteren Abhängigkeiten. Die Verwendung ist sehr einfach und intuitiv. Für die einzelnen Wizard-Seiten werden Klassen implementiert (die von einer Page-Klasse erben) und von einer Seiten-Fabrikklasse wird die Steuerung übernommen. Zusätzlich bietet CJWizard noch eine Datenhaltung, um Daten zwischen Wizard-Seiten zu sichern und zu übertragen. CJWizard wurde für das Projekt als Bibliothek gewählt.

3.4 Quellcode-Merging

Bei einem Quellcode-Generator stellt sich gegebenenfalls ein Problem. Wenn der Benutzer Quellcode auch noch generieren können soll, nachdem am generierten Code bereits Änderungen vorgenommen worden sind, so muss eine Strategie festgelegt werden, wie mit Änderungen zwischen altem und neuem generierten Code, sowie mit Änderungen des Benutzers umgegangen wird.

Die für die Entwicklung des Generators einfachste, aber für den verwendenden Entwickler nutzloseste Möglichkeit ist, den Quellcode einfach zu überschreiben. Zur bestmöglichen Verwendung müsste der Entwickler seine Versionierungssoftware nutzen, um manuell seine überschriebenen Änderungen wieder einzupflegen und die neu generierten Daten dabei nicht zu verlieren.

Auf dieser Idee aufbauend gibt es die Möglichkeit im Tool selbst einen Mechanismus zu integrieren, der Änderungen zur vorigen Version erkennt und beim Einpflegen bei gegebenenfalls auftretenden Konflikten den Benutzer fragt, wie mit diesen umgegangen werden soll. Hierzu müsste generierter Quellcode zwischengespeichert werden, damit bei einer später folgenden, erneuten Generierung die Änderungen am Modell und am generierten Code bestimmt werden können. Hierbei bestünde jedoch die Gefahr des Verlustes dieser Daten. Sei es weil die Daten auf der Festplatte oder im Repository ansonsten nutzlos sind und daher entfernt werden, oder weil diese lokal auf dem Rechner gespeichert werden und die erneute Quellcodegenerierung aber auf einem anderen Rechner durchgeführt wird. Die Implementierung eines Änderungserkennungs- und eines Mergingsystems, für die Verwebung der erkannten Änderungen mit dem aktuellen Entwicklercode, wäre sehr aufwendig. Das Verwebungssystem müsste Konflikte zwischen durchzuführenden Änderungen und Änderungen des Entwicklers gegenüber dem vorher generierten Quellcode erkennen, und dem Benutzer ein Lösen dieser Konflikte ermöglichen.

Eine weitere Möglichkeit ist das Markieren von editierbaren und nicht editierbaren Bereichen im generierten Quellcode. Dies kann bestenfalls über spezifische Kommentare oder Kommentarblöcke erfolgen. Dem Entwickler würde etwa erlaubt, innerhalb des Methodenkörpers der Guard-Methoden eigenen Code zu implementieren und an anderen Stellen

der Zustandsklassen eigene Methoden zu definieren. Ohne einen Editor, der Änderungen an anderen Stellen unterbindet, wäre die Gefahr jedoch groß, dass der Entwickler den generierten Quellcode verändert. Sei es nur um die Formatierung anzupassen oder um die Einrückung zu wechseln (etwa von Leerzeichen zu Tabs, oder umgekehrt). Selbst mit einem solchen funktionstüchtigen Editor könnte man den Entwickler nicht dazu zwingen, diesen zu verwenden. Der Entwickler möchte seine Toolchain und IDE sicherlich nicht verlassen. Das Erkennen der generierten Änderungen ist ebenfalls nicht trivial. Entweder muss wieder, wie oben bei der Diff-Merge-Strategie beschrieben, ein Mechanismus zur Erkennung der Änderungen mit dem gespeicherten, vorig generierten Quellcode implementiert werden, oder es muss zwischen aktuellem Quellcode und neu generiertem Quellcode verglichen werden. Letzteres würde schon durch Änderungen des Entwicklers außerhalb der erlaubten Blöcke, sei es Formatierung, Whitespace oder Methoden-Umsortierung, erschwert. Ein robuster Mechanismus wäre sehr komplex und müsste mögliche Änderungen des Entwicklers außerhalb der markierten Bereiche berücksichtigen.

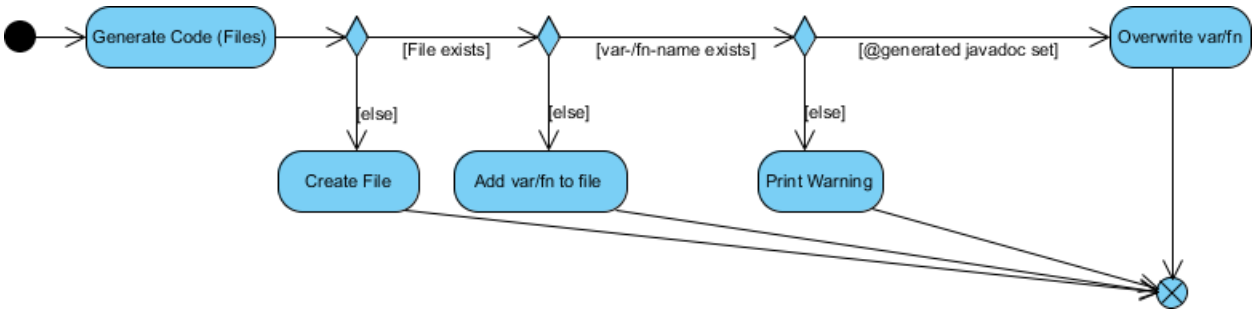

Abbildung 6: Schematischer Ablauf des Mergings mit @generated Javadoc-Tag

Eine weitere Möglichkeit ist das Markieren von generierten Methoden mit einem Javadoc-Tag (beispielsweise *@generated*). Abbildung 6 zeigt den schematischen Ablauf dieser Verwebungsstrategie (den Merge). Methodenköpfe sind innerhalb einer Klasse einmalig, womit zu einer neu generierten Methode stets die zuvor generierte, aktuell eventuell geänderte und implementierte Methode gefunden werden kann. Ist der spezifizierte Javadoc-Tag noch vorhanden, so wird die Methode einfach überschrieben. Implementiert der Entwickler die Methode, etwa den Guard eines Zustandsübergangs, so entfernt er den Javadoc-Tag und signalisiert damit, dass diese Methode nicht mehr überschrieben werden soll. Bei der erneuten Quellcodegenerierung wird nun für die neu generierte Methode die alte (mit selbem Methodenkopf) gefunden und es wird erkannt, dass der Tag entfernt wurde und die Methode

damit nicht mehr überschrieben werden darf. Da jene Methoden, die vom Entwickler angepasst werden, in der Regel keine sich durch Zustandsmodelländerungen ändernden Quellcode besitzen, entsteht hieraus normalerweise auch kein Problem. Ein Nachteil ergibt sich lediglich, falls dies doch einmal der Fall ist. Nach Entfernung des Tags werden die Änderungen am Modell eventuell nicht in den Quellcode übernommen. Diesem Umstand hat der Entwickler mit dem Entfernen des Tags aber explizit zugestimmt, beziehungsweise ist sich dessen bewusst. Sollte dieser Fall eintreten, so ist sich der Entwickler über die Änderungen am Modell meist im klaren und kann diese hinterher gegebenenfalls manuell einpflegen.

Diese zuletzt genannte Möglichkeit, mit Javadoc-Tags, ist die beste Strategie. Zunächst wird jedoch kein Quellcode-Merging unterstützt. Eine Implementierung der Strategie findet nicht statt.

3.5 Hierarchische Zustände

UML ermöglicht es, in Zustandsdiagrammen den sogenannten hierarchischen Zuständen (auch zusammengesetzte Zustände, von englisch „composite states“, oder Untermaschinen-Zustände, von englisch „submachine states“, vgl. [UMLS10]) weitere, untergeordnete Zustandsmodelle zuzuweisen. Dies erlaubt es, Zustandsdiagramme bei der Erstellung abstrakter und übersichtlicher zu gestalten, für einen enthaltenen Zustand aber trotzdem noch die detaillierte Funktionsweise, in einem weiteren Diagramm, beschreiben zu können.

Aus dieser Verschachtelung ergibt sich jedoch, beim Einlesen von XMI-Dateien, ein potenzielles Problem. Entweder, weil gegebenenfalls nur untergeordnete Zustandsmodelle in Quellcode umgewandelt werden sollen oder, weil bei der Generierung von Quellcode aus dem übergeordneten Zustandsmodell auch die untergeordneten, eingebetteten Zustandsmodelle verarbeitet werden sollen.

Für die Umsetzung hierarchischer Zustände zu Java-Quellcode, und damit als Lösungen für dieses Problem, fanden sich zunächst drei mögliche Ansätze:

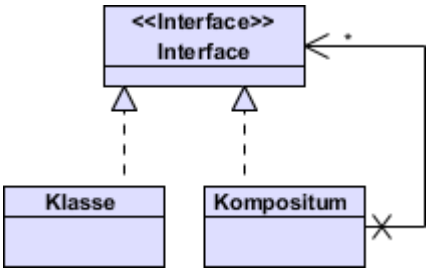

Abbildung 7: Kompositum-Entwurfsmuster

Ein Kompositum, beziehungsweise das Kompositum-Entwurfsmuster (engl. „composite pattern“, vergleiche Abbildung 6), definiert zunächst ein abstraktes Interface, von

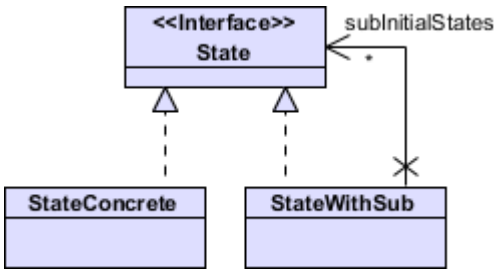

Abbildung 8: Kompositum auf Zustände angewandt

welchem einmal eine konkrete Klasse erbt, und zum anderen eine zweite Klasse, die das Interface wiederum selbst enthalten kann. Man könnte über ein Kompositum also, unter das gleiche einen Zustand repräsentierende Interface, einerseits normale Zustandsklassen und andererseits hierarchische Zustandsklassen zusammenfassen und verwenden. Dies wird in Abbildung 7 als Diagramm dargestellt.

Diesen Ansatz kann man jedoch fortführen und auf den konkreten, im Kapitel 12 dargestellten, Zielcode anwenden. Abbildung 9 zeigt, dass lediglich die allgemeine *ObjectState*-Klasse angepasst werden muss. Sie hat fortan eine zusätzliche Assoziation zu einer Liste untergeordneter Initialzustände, welche ebenfalls vom Typ *ObjectState* sind. Initialisiert wird die Assoziation über den ebenfalls neu hinzugefügten Konstruktor, welcher eine Liste von Initialzuständen übergeben bekommt. Hat ein Zustand also untergeordnete Zustandsmodelle, so erhält der übergeordnete ObjectState-Zustand eine Assoziation zum Initialzustand des untergeordneten Modells. Dieser reicht aus um in die untergeordnete Zustandsmaschine einzusteigen und diese zu identifizieren, da der Initialzustand einmalig ist, und es für jedes Zustandsmodell nur einen geben kann. In den Transitionsmethoden muss nun nur noch eine passende Fallunterscheidung implementiert werden, um die untergeordneten Zustandsmodelle zu betreten und wieder aus ihnen zurück zu kommen.

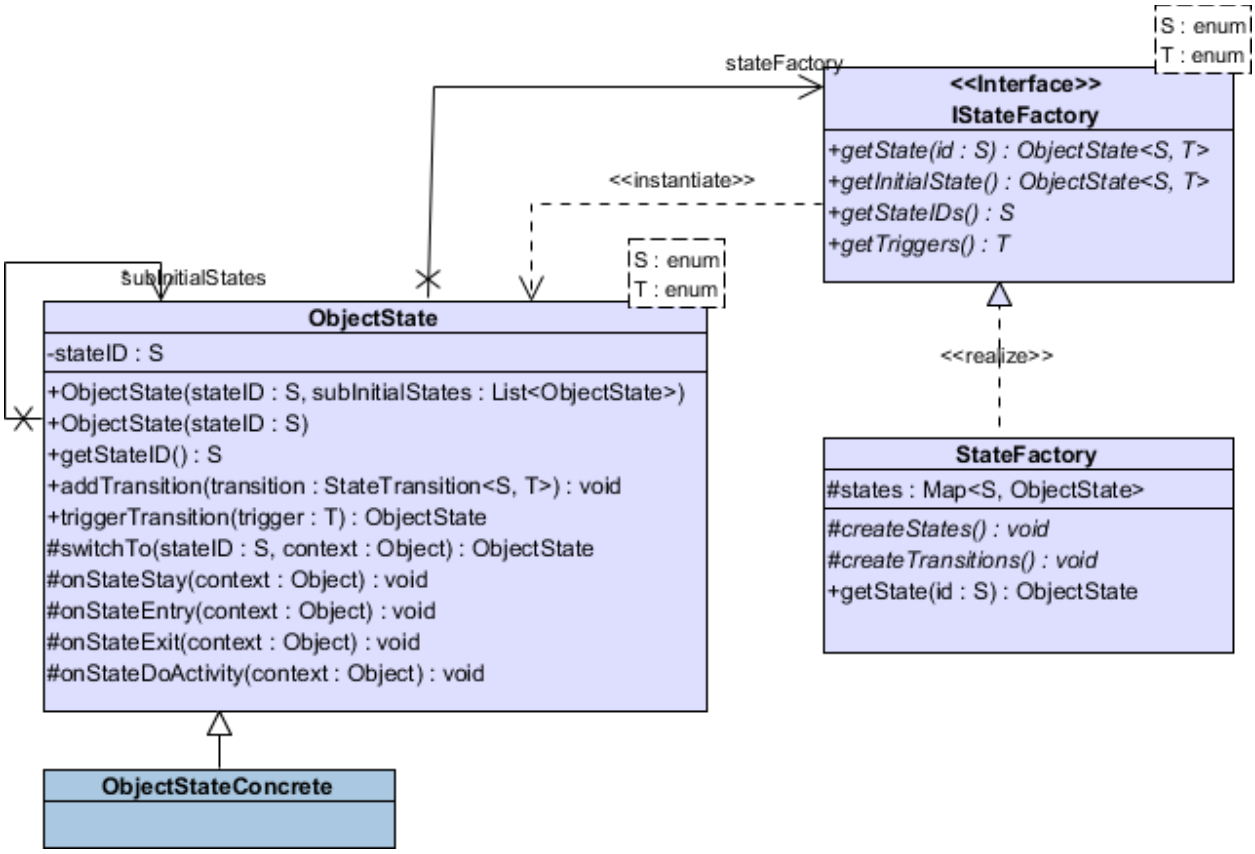

Abbildung 9: Adaption des Kompositum and konkreten Zielcode

Eine zweite Möglichkeit hierarchische Zustände umzusetzen ist, dass die entsprechenden Zustände einfach das bereits vorhandene *IStateFactory*-Interface implementieren (vergleiche hierzu das später folgende Kapitel Design). Beim Wechsel in den Zustand übergibt dieser übergeordnete Zustand an den untergeordneten Zustand (das heißt, liefert als neuen Zustand bei einem Zustandsübergang diesen untergeordneten zurück). Der finale Zustand des untergeordneten Zustandsmodells muss dann nur noch an den passenden übergeordneten Zustand übergeben, um die Kontrolle wieder an das übergeordnete Zustandsmodell weiterzugeben. Abbildung 10 verbildlicht diese Möglichkeit als Diagramm.

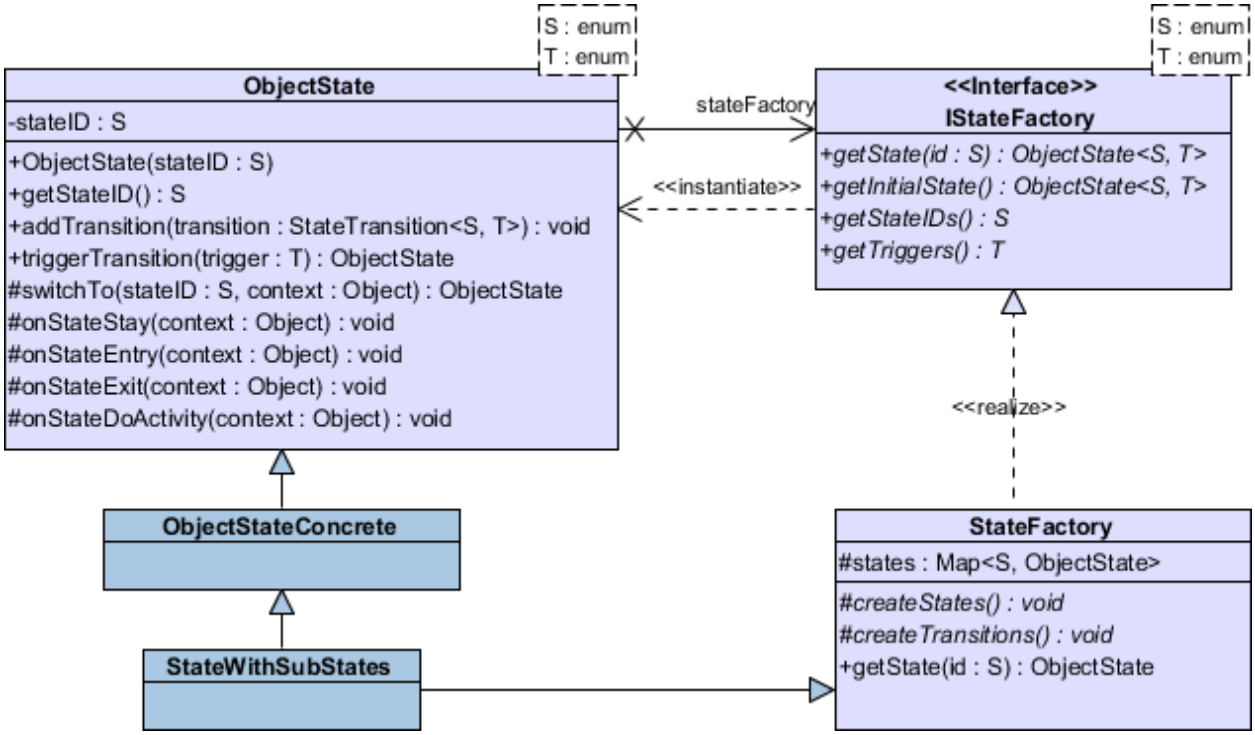

Abbildung 10: Umsetzung hierarchischer Zustände über Factory-Vererbung und -Kaskadierung

Der UML Superstructure Standard ([UMLS10]) definiert, dass in einer Region (in zusammengesetzten Zuständen, in „composite-states“) nur ein einziger Startzustand sein kann. Der Zustandsübergang, der von diesem ausgeht, darf außerdem keinen Guard und keinen Trigger besitzen. Das heißt, immer wenn der übergeordnete Zustand erreicht wird und damit automatisch das untergeordnete, detailliertere Zustandsmodell in Kraft tritt, so tut es dies stets über den einen definierten Startzustand und die von ihm ausgehende Transition. Diese Transition darf Verhalten definiert haben, welche noch beachtet werden müssen. Ansonsten aber kann man den durch das Modellieren in mehreren Ebenen geschaffenen Bruch auflösen, indem man eine direkte Verbindung herstellt. Diese Verbindung besteht zwischen dem übergeordneten, vorangegangen Zustand und dem untergeordneten, dem Startzustand folgenden Zustand (vgl. auch Ähnlichkeit zum ersten der drei hier Vorgestellten Möglichkeiten). Bei der Verbindung, also dem Zustandsübergang, muss der Trigger und der Guard des übergeordneten Übergangs übernommen werden und die Effekte von beiden Übergängen müssen kombiniert werden. Ein finaler Zustand (*FinalState*), welcher im untergeordneten Modell das Ende des Modells markiert, beendet die Ausführung des untergeordneten Modells und kehrt zum übergeordneten Modell zurück. Auch dieser Bruch, beim Beenden des untergeordneten Modells, ist auflösbar. Man kann auch hier

einen Zustandsübergang, zwischen den letzten, untergeordneten Zuständen und dem nächsten übergeordneten Zustand, herstellen. Dies geschieht analog zum Eintreten in das untergeordnete Modell. Von Endzuständen, und natürlich auch den ihnen vorgelagerten Zuständen, kann es, im Gegensatz zum Eintrittsknoten, mehrere geben. Diese müssen alle angepasst werden und eine Verbindung in das übergeordnete Modell herstellen.

Auf mögliche Unterschiede, zwischen zusammengesetzten Zuständen und Zuständen mit untergeordneten Zustandsmaschinen („composite states“ und „sub-statemachine states“), wäre vor einer Anwendung, einer dieser dargestellten Möglichkeiten, noch zu prüfen. Ebenso wäre auf die Spezifika von Eintritts- und Austrittspunkten zu prüfen („Entry point” und „Exit point”, im Gegensatz zu Startknoten und Endknoten in Regionen bei zusammengesetzten Zuständen). Im Software-Projekt StateXMI2Java wurde ein Verarbeiten hierarchischer Zustände zunächst nicht umgesetzt. Die Möglichkeiten wurden aber, wie in diesem Kapitel dargestellt, geprüft und bewertet.

3.6 Aktivitätsobserver statt Aktivitätsmethoden

Es gibt drei Aktivitätsarten, die einem Zustand zugeordnet sein können. So kann dem Eintritt in einen Zustand eine Aktivität zugeordnet werden. Die Aktivität wird dann vor jeglichen anderen zustandsinternen Berechnungen oder weiteren Übergängen durchgeführt. Analog kann dem Verlassen eines Zustandes ebenfalls eine Aktivität zugeordnet werden. Zusätzlich gibt es noch die Möglichkeit, eine Aktivität zur Ausführung innerhalb eines Zustands anzugeben, die *doActivity*. Sie wird beim Eintritt in den Zustand gestartet und beendet sich entweder von selbst, oder wird beim Austritt aus dem Zustand beendet. Sie ist, gegenüber den anderen beiden Arten, nicht blockierend.

Ursprünglich, durch ein Missverständnis begonnen, wurde eine alternative Möglichkeit analysiert und evaluiert, um diese Aktivitätsarten in den generierten Quellcode umzusetzen. Abbildung 11 zeigt ein Diagramm des zu generierenden Quellcodes. Es ist bereits aus den Abbildungen 4 und 5 aus dem Kapitel Analyse bekannt, wurde jedoch angepasst. Abbildung 12 zeigt die Änderungen als vergrößerten Ausschnitt.

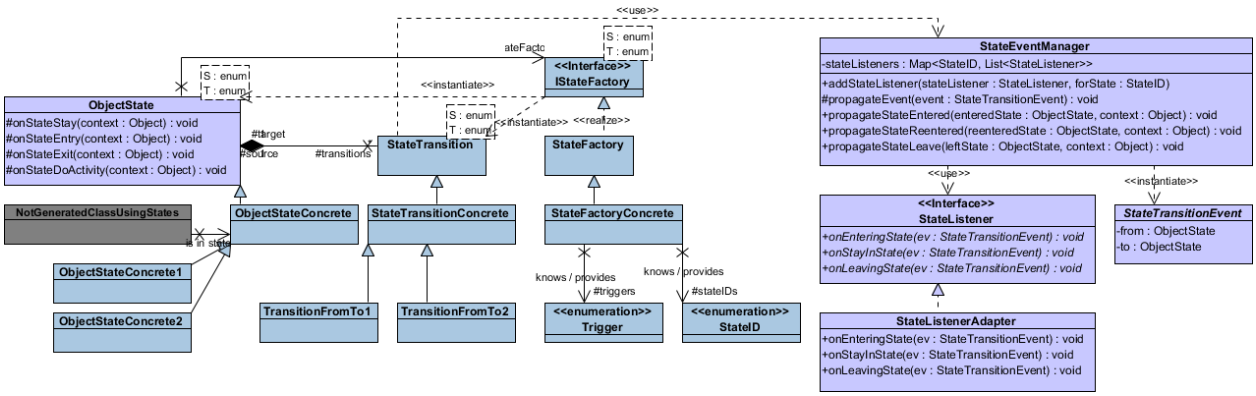

Abbildung 11: Diagramm mit Unterschieden im generierten Code mit Aktivitätsobserver

Der Großteil der Klassen bleibt gegenüber dem bisherigen Zielcodedesign gleich (dunkles Grau). Die Zustandsklasse *ObjectState* verliert jedoch die Aktivitätsmethoden zu Gunsten der neuen Klassen *StateEventManager*, *StateListener*, *StateListenerAdapter* und *StateTransitionEvent*. Diese realisieren ein Event-Listener-, beziehungsweise Observer-System für Zustandsaktivitäten.

Hierfür können beim *StateEventManager* für bestimmte Zustände Zustandslistener registriert werden, deren Methoden dann beim Auftreten des Events aufgerufen werden. Die Events werden

in der *doTransition*-Methode von Transitionen ausgelöst, wo im ursprünglichen Design auch die Aktitivätsmethoden der Zustände aufgerufen werden (siehe *use*-Beziehung von *StateTransition* zu *SateEventManager*). Statt der bisherigen direkten Funktionsaufrufe wird nun, jedoch über den Manager, eine unbekannt große Liste an Listenern benachrichtigt.

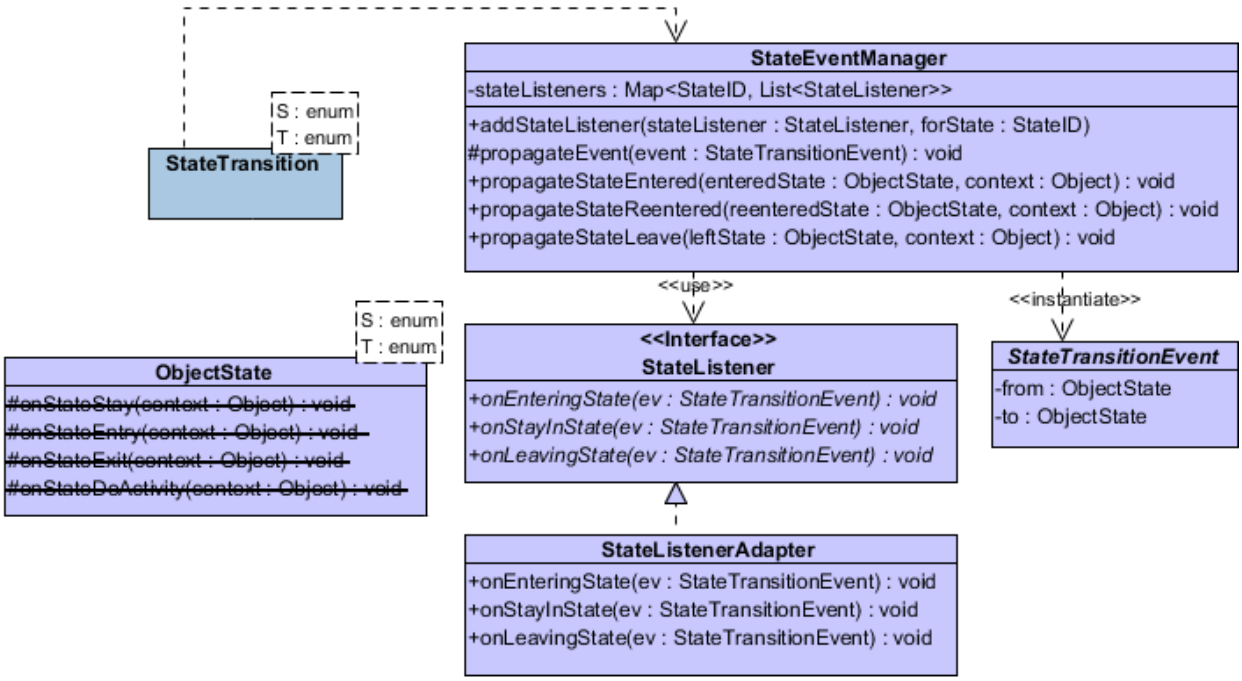

Abbildung 12: Detaildarstellung der Änderungen im generierten Code mit Aktivitätsobserver

Bei einer solchen Erweiterung des Designs stellt sich natürlich die Frage nach der Nützlichkeit dieses Ansatzes. Vorteile sind vor allem darin zu sehen, wenn man als Entwickler oft die selbe Logik für Zustandsein- und -austrittsaktivitäten verwendet. Bei einem Observersystem, wie nun vorgestellt, müsste man nur einen Listener implementieren und könnte diesen auf Events von mehreren Zuständen registrieren. Ohne Observerfunktionalität müsste man die Aktivitätsmethoden der Zustände einzeln implementieren, und um Redundanz zu vermeiden, gegebenenfalls mehrfach verwendete Logik in Helferklassen oder -methoden verlagern. Dies würde, vor allem gegenüber der Verwendung von Listenern, die Logik schlechter kapseln und die Übersichtlichkeit beeinträchtigen.

Die Verwendung des Observer-Patterns hätte aber auch klare Nachteile. Der generierte Quellcode wäre wesentlich komplexer (zusätzliche Klassen, Klassenabhängigkeiten und dynamische Assoziationen). Durch die gesteigerte Komplexität wäre auch die Verwendung des

Codes durch einen Entwickler weniger eingängig, vor allem, wenn nur einzelne Zustände einzelne Aktivitäten ausführen sollen. Zusätzlich würde der Implementierungsaufwand für die Generierung dieser zusätzlichen Observerlogik steigen. Aus diesen Gründen, und weil die Vorteile wohl doch eher selten zum Tragen kommen würden, ist die Möglichkeit, dieses Observersystems zu nutzen, nur eine mögliche Erweiterung. Eine Implementierung findet zunächst nicht statt.

4. Design

StateXMI2Java ist als Wizard konzipiert. Zunächst wurden die typischerweise notwendigen Schritte, aus der Sicht des Benutzers, für die Erzeugung des Java Quellcodes aus XMI Dateien ermittelt.

- XMI-Datei und Zielverzeichnis wählen
- falls mehrere Zustandsmodelle in XMI-Datei vorhanden: Zustandsmodell wählen
- Generierung des Quellcodes
- Abschlussfenster, mit Option das Zielverzeichnis zu öffnen

Diese Schritte wurde in einem GUI-Prototypen umgesetzt, welcher in der Netbeans IDE erstellt wurde.

4.1 Datei- und Paketstruktur

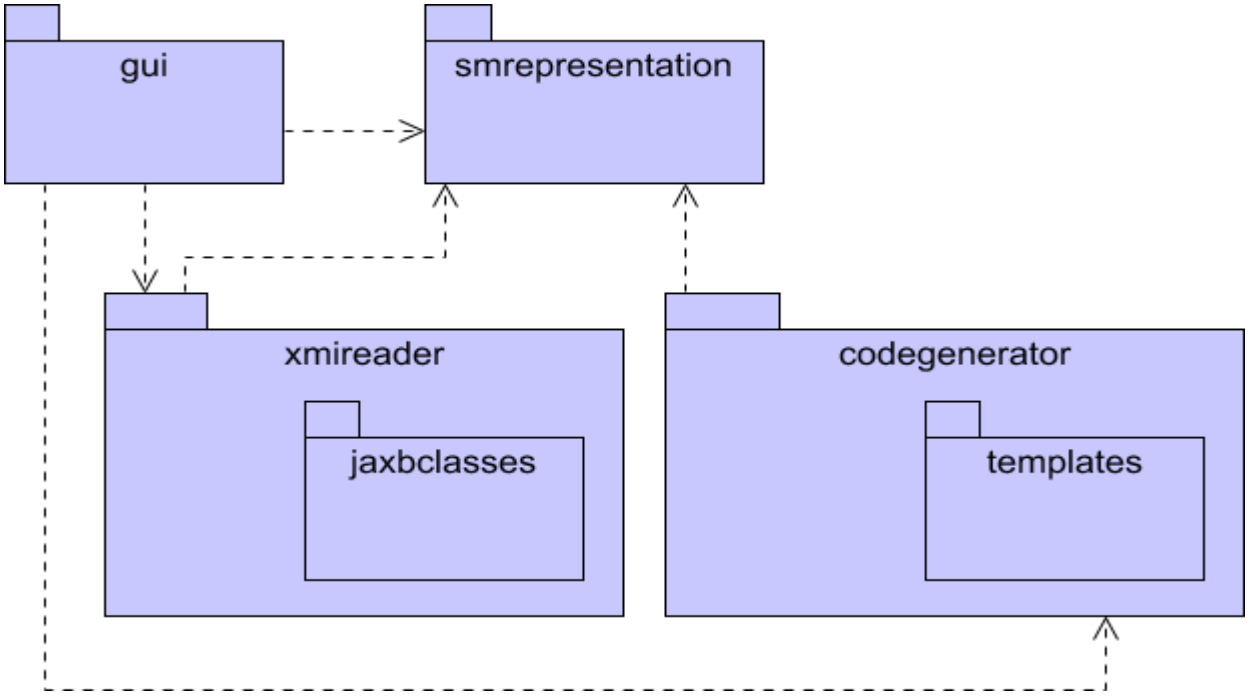

Abbildung 13: StateXMI2Java Paketdiagramm mit den Hauptpaketen

Das Programm besteht aus den Wizard-GUI-Komponenten, dem XMI-Import und dem Quellcode-Export. Außerdem sind für die interne und Java-native Repräsentation des

eingelesenen Zustandsmodells noch entsprechende Klassen notwendig. Wie in Abbildung 13 zu sehen, wurden für diese Komponenten vier Hauptpakete definiert. Die Abbildungen 14 und 15 zeigen die Pakete detailliert.

Im Paket *gui* befinden sich die Klassen des Wizards, also die Klassen für die einzelnen Seiten des Wizards, sowie für die Steuerung des Wizards. Die Klassen im *gui*-Paket sind allerdings noch nicht CJWizard-spezifisch, sondern stellen nur dar, welche Wizard-Seiten notwendig sind und was in der Datenhaltung des Wizards über die Schritte hinweg gehalten werden muss. Bei der Implementierung wurden diese an die CJWizard-Bibliothek angepasst (siehe hierzu auch das Kapitel Implementierung unten).

Im *smrepresentation*-Paket befinden sich die oben erwähnten, zur Repräsentation von Zustandsmodellen notwendigen, Klassen. Wie Abbildung 15 zeigt, sind die Klassen *StateMachine*, *State*, *Transition* und *Behavior* für die Quellcodegenerierung ausreichend, um ein Zustandsmaschinenmodell zu repräsentierten. Diese Klassen wurden anhand des UML-Standards definiert.

Die für das Einlesen von XMI-Dateien notwendigen Klassen sind im *xmireader*-Paket enthalten. Neben dem *IXMIReader*-Interface befinden sich hier zwei weitere Klassen. Der *XMIReader* implementiert allgemeine Funktionen, die für Klassen, die XMI-Dateien parsen, generisch sind. Der JAXB-spezifische Reader *XMIReaderJAXB* dagegen enthält die konkrete Parse-Logik. Hierzu nutzt dieser die JAXB-annotierten Klassen, welche im Unterpaket *jaxbclasses* enthalten sind. Diese annotierten Klassen stellen ein Mapping zu der XML-Struktur, in einer XMI-Datei, her.

Ein *IXMIReader* bekommt als Parameter den Pfad zur einzulesenden XMI-Datei und kann anschließend, nach den in dieser XMI-Datei enthaltenen Zustandsmaschinenmodellen abgefragt werden. Er liefert dann einen Objektgraphen, der aus Instanzen der oben erwähnten Repräsentationsklassen besteht.

Im *codegenerator* Paket befinden sich die Klassen zur Quellcodegenerierung. Ein *CodeGenerator* erzeugt in einem übergebenen Zielverzeichnis den Quellcode für eine ebenfalls übergebene Zustandsmaschinenrepräsentation. Typischerweise ist dies jene Repräsentation, die vom *IXMIReader* nach dem Einlesen einer XMI-Datei geliefert wird.

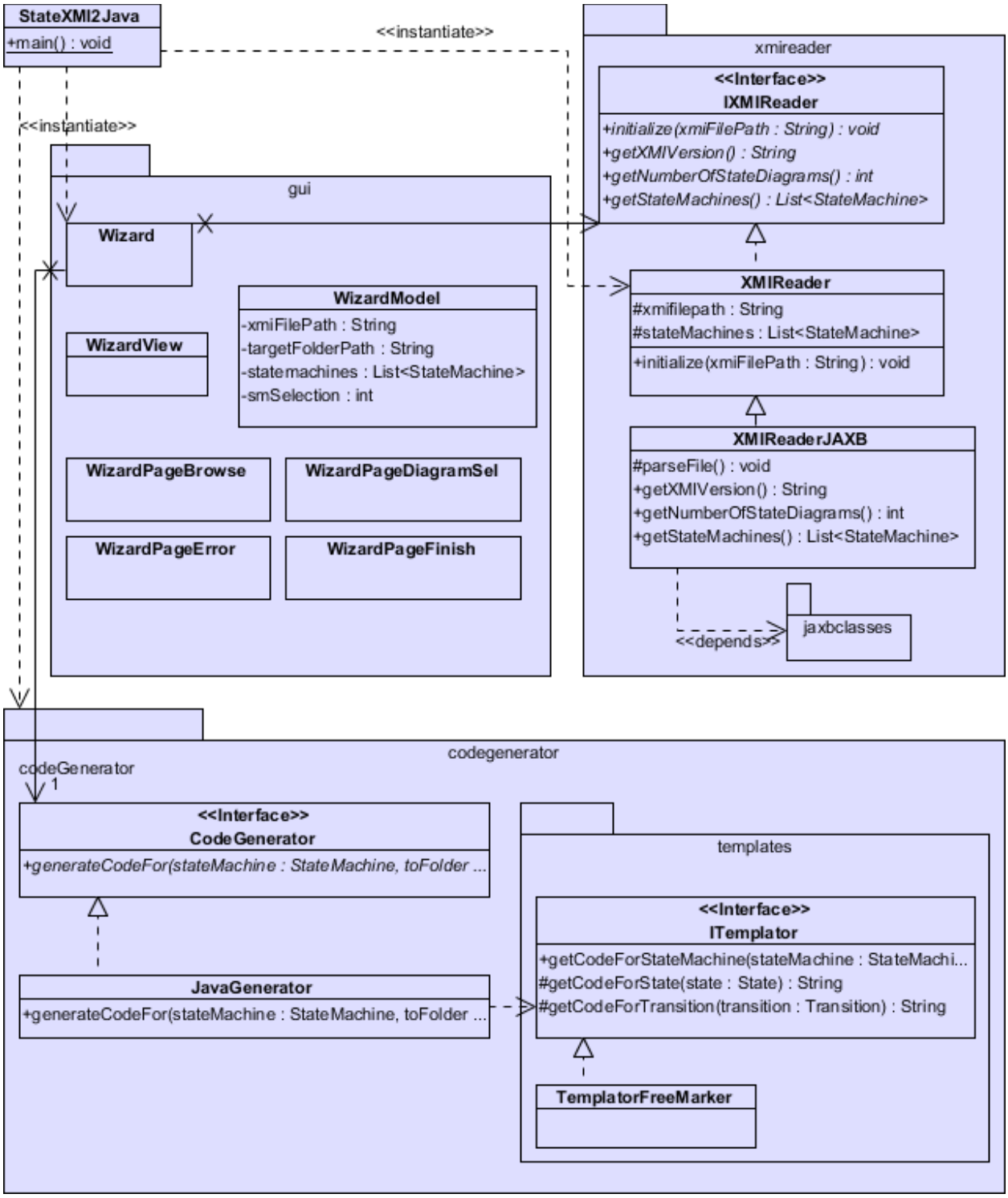

Abbildung 14: StateXMI2Java Klassendiagramm, ohne Paket smrepresentation

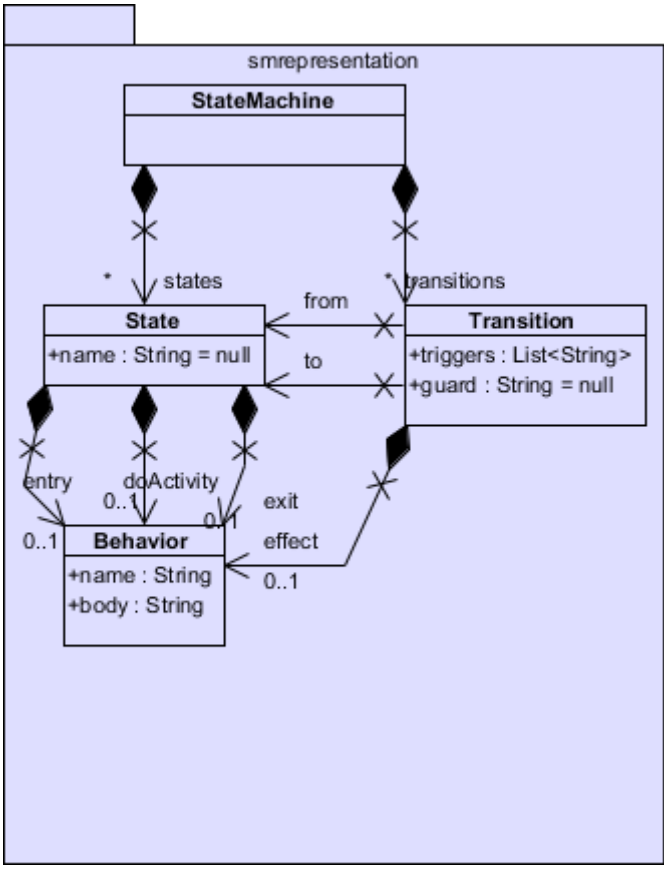

Abbildung 15: StateXMI2Java Klassendiagramm, Paket smrepresentation

4.2 CJWizard

Im vorigen Kapitel Datei- und Paketstruktur ist bereits der Grundaufbau des Wizards erläutert. Der Inhalt des *gui*-Pakets, aus dem Programmdesign, ist nochmals in der unten folgenden Abbildung 16 gezeigt. Die Implementierung der GUI wurde anhand des Designs und des Prototypen durchgeführt, aber auf die CJWizard-spezifischen Klassen und Methoden umgesetzt. Das Resultat ist als abstraktes Klassendiagramm (ohne Methoden und Attribute) in Abbildung 17, und detailliert (mit Methoden und Attributen) in Abbildung 18 zu sehen.

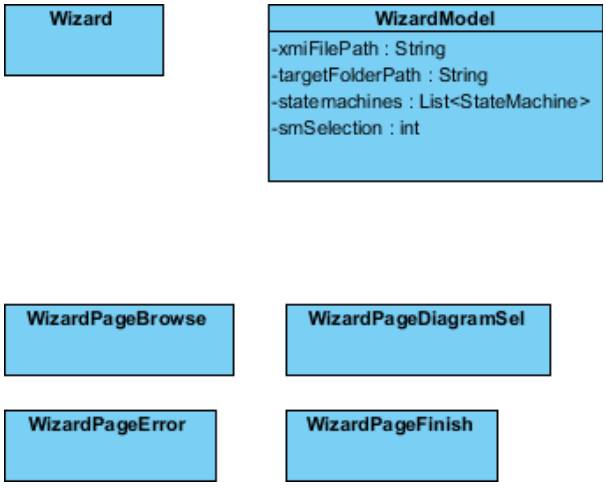

Abbildung 16: Klassendesign im gui-Paket

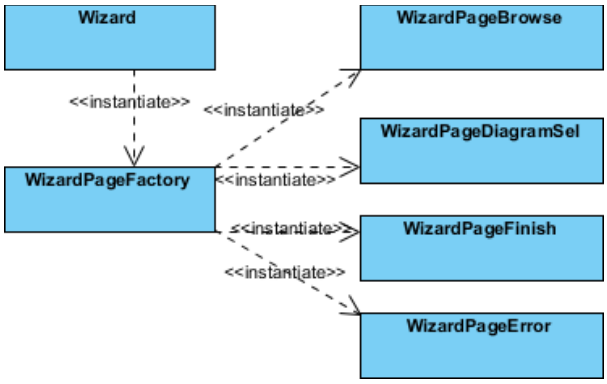

Abbildung 17: Klassendesign im gui-Paket bei Implementierung, angepasst an CJWizard

Zunächst wird die *Wizard*-Klasse erstellt, welche ein *JFrame*-Fenster erzeugt und ein *WizardContainer-JPanel* als Inhalt festlegt. Der *WizardContainer* ist eine von CJWizard bereit gestellte Klasse, welche über eine ihr übergebene *PageFactory* die passenden Wizard-Fenster erzeugt und anzeigt. *WizardPageFactory* ist eine solche von *PageFactory* erbende Klasse, welche dem Wizard-Container letztendlich übergeben wird. Sie entscheidet in der Methode *createPage,* welche Wizard-Seite angezeigt werden soll und instanziiert diese. Diese Wizard-Seiten, auch Fenster oder Schritte genannt, sind rechts in den Diagrammen der Abbildungen 17 und 18 als Klassen zu finden. *WizardPageBrowse*, *WizardPageDiagramSel*, *WizardPageFinish* und *WizardPageError* erben hierzu von der von CJWizard bereitgestellten Klasse *WizardPage*. Die von *WizardPage* bereitgestellte und überschreibbare Methode *updateSettings* erlaubt es, beim Verlassen einer Wizard-Seite etwaige Formulareingaben oder Berechnungen in den *Settings*, einer Daten-*HashMap*, zu speichern. So werden am Ende der Browse-Seite die Formulareingaben zu Quell-XMI-Datei und Zielverzeichnis in dieser Datenschicht gesichert. Zusätzlich wird die XMI-Datei eingelesen und verarbeitet, und die resultierenden, eingelesenen Zustandsmodelle werden ebenfalls gespeichert. Wurde mehr als ein Zustandsmodell gesichert, so wird über die Seitenfabrik die *DiagramSel*-Seite angezeigt. Diese speichert beim Verlassen den Index des vom Benutzer gewählten Zustandsmodells. Die Quellcodegenerierung wird durch die Fabrik angestoßen nachdem feststeht, welches Zustandsmodell in Code umgewandelt werden soll. Nach einer erfolgreichen Generierung wird die Finish-Seite angezeigt, welche auch das Öffnen des Zielverzeichnisses anbietet.

Tritt während einer der genannten Schritte und Seiten ein Fehler auf, so wird dem Benutzer dies als Meldung auf einer *WizardPageError*-Seite angezeigt.

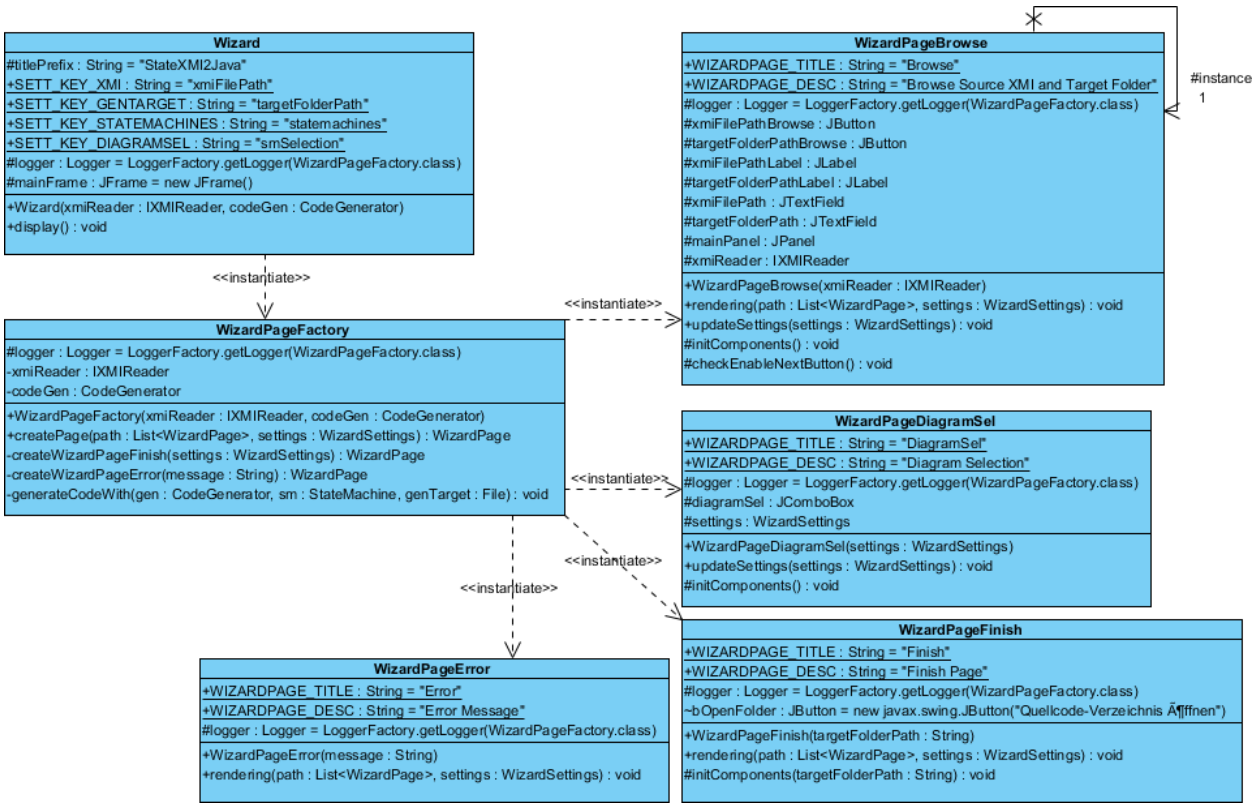

Abbildung 18: Detailliertes Klassendesign des GUI-Pakets bei Implementierung, angepasst an CJWizard

Die Abhängigkeiten werden durch Konstruktor-Dependency-Injection aufgelöst. Das heißt die Wizard-Instanz übergibt der Wizard-Seitenfabrik die von ihr benötigten Abhängigkeiten, welche wiederum die Abhängigkeiten der jeweiligen Seiten an diese übergibt. So wird eine strenge Abhängigkeitshierarchie „von oben nach unten" eingehalten.

4.3 Code-Templates

Im Abschnitt Template-Engines wurden bereits die Vorzüge einer Template-Engine dargestellt und warum FreeMarker für dieses Projekt verwendet wird. Eine möglichst große Abdeckung des generierten Quellcodes mit Templates ist wünschenswert, aber als erster Schritt nicht naheliegend. Stattdessen werden die wichtigsten und größten Codestücke, wie die Gerüste der Klassen als Templates umgesetzt und dynamisch generierte Methoden und Codeschnipsel

zunächst im Programmcode als String erzeugt und erst später als Templates umgesetzt. Ein Beispiel für solch dynamischen Code ist etwa der Konstruktur-Code eines Zustandsübergangs. Dieser enthält die Instanziierung der Start- und Zielzustände, sowie das Hinzufügen der Übergänge zu den Zuständen.

Dieser Ansatz erleichtert die erste Implementierung durch weniger Templates und die Codeschnipselgenerierung an Ort und Stelle. Er führt aber zu enger Kopplung und ist daher schon auf mittlere Sicht nicht wünschenswert. Deshalb werden auch bald folgend Templates verwendet.

Die *JavaGenerator*-Klasse verwendet die zu implementierende, für die Templates zuständige Klasse *TemplatorFreeMarker* als *ITemplator*. Abbildung 19 zeigt den Mechanismus. Die Templating-Klasse liest die Template-Datei ein und ersetzt die Platzhalter durch ihm übergebene Werte. Sie gibt den so verwobenen Quellcode anschließend zurück an den Aufrufer, den *JavaGenerator*. Die Generator-Klasse schreibt den Quellcode dann als Datei in das Zielverzeichnis.

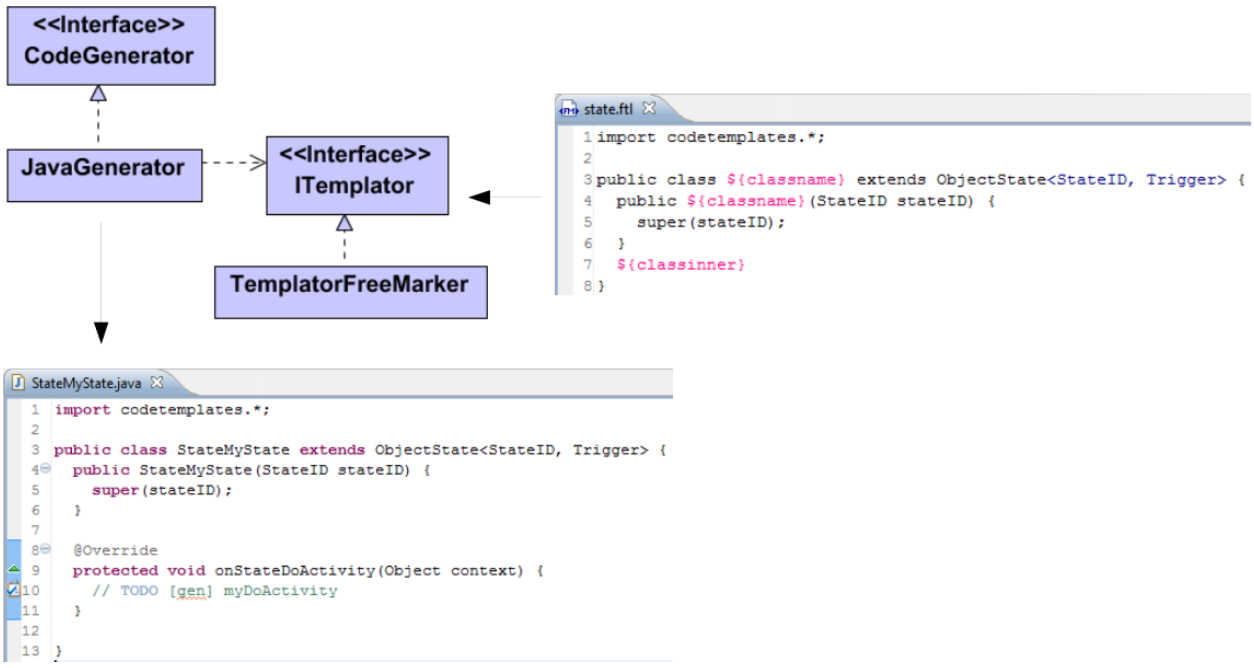

Abbildung 19: Mechanismus des Template-Parsings zu Quellcode

Abbildung 4 zeigt den zu generierenden Quellcode. Die grundlegenden Klassen in der oberen Hälfte werden für jede Generierung unverändert übernommen und müssen daher nicht als

Templates mit Platzhaltern realisiert werden. Denn es reicht aus, diese als fertig implementierte Templates beziehungsweise Dateien, zu erstellen. Bei der Quellcodegenerierung werden diese Dateien dann nur noch in das Zielverzeichnis kopiert.

Von den modellspezifisch generierten Klassen (siehe untere Hälfte des Diagramms) wird für jede Klasse ein Klassentemplate angelegt. Die Zustandsübergangsklassen (Transition-Klassen) beispielsweise verwenden zusätzlich ein Template für Guard-Methoden. Diese sind in einem eigenen Template umgesetzt, da nicht alle Transitionen eine Guard haben müssen, und damit die Methode auch nicht überschreiben. Der Inhalt dieses Templates ist im Folgenden dargestellt:

```
/**
 * @generated
 */
public boolean isGuardConditionMet(Object context) {
  // TODO [gen] implement guard ${guard}
  return false;
}
```

Bei den Guard-Methoden ergibt sich hier die Notwendigkeit einer Überlegung, was die Standard-Rückgabe sein soll. Die Guard-Methoden prüfen, ob der Übergang durchgeführt werden kann, oder ob der Guard nicht zutrifft und der Übergang damit verhindert wird. Geben die konkreten, überschriebenen Guard-Methoden standardmäßig *false* zurück, so ist der Entwickler gezwungen, zunächst die Guard-Prüfung, welche im Quellcode als *TODO* eingebaut wird, zu implementieren, bevor die Transition durchgeführt werden kann. Eine fälschliche Transition wäre potenziell wesentlich schlimmer und verwirrender, als eine nicht durchgeführte Transition aufgrund eines nicht implementierten Guards, bei dem das *TODO* sogar noch vorhanden ist. Hierbei ist zusätzlich zu beachten, dass die Guard-Methode der Basisklasse *true* zurückliefern muss, damit jene Transitionen ohne Guard, und damit ohne Guard-Methoden-Überschreibung, validieren und die Transition möglich ist.

Eine weitere Herausforderung bei der Generierung stellt sich mit dem Paket, in welches der generierte Java-Code, beziehungsweise die Klassen, platziert werden. Da die Basisklassen des generierten Quellcodes bei jeder Generierung gleich sind, ist es wünschenswert diese in ein Unterpaket zu platzieren. Würden in einem Projekt mehrere Zustandsmodelle zu Javacode generiert, so könnte man das Unterpaket einer dieser Generierungen verwenden und die anderen entfernen, um Redundanz zu vermeiden. Ein weiteres mögliches Problem entsteht hierbei dadurch, dass der Entwickler den generierten Quellcode womöglich in einem Unterpaket seines

Projektes verwenden möchte. Es ergeben sich folgende Möglichkeiten:

Die Basisklassen enthalten keine Angabe zum Paket und werden einfach in einem Unterordner des Zielordners erstellt. Hiermit müsste der Entwickler aber nach der Generierung zunächst Paketnamen vergeben und sich um den Import des Paketes kümmern.

Eine andere Möglichkeit ist, den Benutzer nach dem Paketnamen zu fragen und bei der Generierung die Paketnamen für die konkreten Klassen, sowie für die Basisklassen passend zu setzen.

Oder aber die Basisklassen erhalten einen Paketnamen passend zum Unterverzeichnis im Zielverzeichnis, in das sie gespeichert werden. Der Entwickler muss das Paket gegebenenfalls anschließend nur noch umbenennen, um der Pakethierarchie in seinem Projekt zu entsprechen. Der generierte Quellcode an sich ist aber in sich geschlossen bereits funktionstüchtig, weil in den konkreten, abgeleiteten Klassen bereits das passende Import-Statement für die Basisklassen angegeben wird. Dies ist auch die Möglichkeit, welche für dieses Projekt verwendet wird.

5. Implementierung

In diesem Kapitel werden verschiedene Aspekte der Implementierungsphase des Projektes beschrieben. So wird zunächst auf die Entwicklungsumgebung eingegangen. Anschließend werden die Implementierung des XMI-Imports, sowie die dort aufgetretenen Herausforderungen beschrieben. Danach folgt eine Beschreibung des Hinzufügens eines Build-Systems, mit der Einrichtung einer Jar-Generierung und Abhängigkeitsauflösung.

5.1 Entwicklungs- und Testumgebung

Zur Versionsverwaltung wird Git verwendet. In der Toolchain kommt als IDE Eclipse mit zahlreichen Erweiterungen zum Einsatz. Zu den Erweiterungen zählen unter anderem eine erweiterte XML-Darstellung, Freemarker-Syntaxhervorhebung und EGit als Git Integration. Als Buildsystem wird, neben Eclipse selbst, Ant, sowie Ivy zur Abhängigkeitsauflösung verwendet.

Zum Testen wird JUnit und EasyMock verwendet. Ergänzend wird zur statischen Quellcodeanalyse FindBugs genutzt.

5.2 XMI-Import

Die Umsetzung der JAXB-annotierten Klassen wurde in mehreren Schritten durchgeführt. Zunächst wurden, mittels eines manuell erstellten XML-Schemas, die JAXB-Klassen generiert. Diese generierten Klassen wurden anschließend als Basis verwendet und weiter angepasst.

Die noch notwendigen Änderungen wurden zunächst, anhand des UML-Standards, identifiziert und ausgeführt. Der UML-Standard spezifiziert eine Klassen- und Vererbungshierarchie der UML-Elemente, welche im XMI-Format, als hierarchische XML-Struktur, wiederzufinden sind.

Um die Änderungen an den JAXB-Klassen auf Korrektheit verifizieren zu können, wurden einige Beispiel-XMI-Dateien aus Visual Paradigm exportiert und Unit-Tests für den Import von diesen geschrieben. Die XMI-Dateien enthalten eine Reihe von verschiedenen und unterschiedlich vielen UML-Elementen. Die Dateien reichen von einer XMI-Datei ohne Zustandsmodell bis zu einer solchen mit einem Zustandsmodell mit Triggern, Guards, Effekten und Aktivitäten.

Zu beachten ist auch, dass XMI-Dateien, beziehungsweise das XMI-Format, *idref* und *XLink* Verknüpfungen erlaubt. Mit diesen können XML-Elemente über eine ID referenziert werden. Das heißt, das Element wird an einer Stelle mit ID (womöglich mit weiteren Attributen und Unterelementen) definiert, und kann an einer anderen Stelle im XML über die ID referenziert werden. Beim Parsing müssen also gegebenenfalls die Eigenschaften eines referenzierten XML Elements an einer anderen Stelle übernommen werden. JAXB kennt auch für ID-Referenzen Annotationen (*@XmlId* und *@XmlIDREF*), welche hierfür verwendet werden.

Wie zuvor erwähnt fand die Implementierung zunächst mit Unit-Tests statt, welche aus Visual Paradigm exportierte Zustandsdiagramme einlesen. Mit Fortschreiten der Implementierung wurden immer komplexere XMI-Daten eingelesen. Als jedoch versucht wurde eine XMI-Datei einzulesen, welche aus einem Visual Paradigm Projekt mit mehreren Zustandsdiagrammen erstellt wurde stellte sich heraus, dass sich die XMI-Daten nicht wie vermutet verhielten, also nicht wie erwartet um ein neues Modell erweitert wurden.

Nach Begutachtung der XMI-Dateien wurde davon ausgegangen, dass Zustandsmaschinen immer als *Model*-Unterelemente (*ownedMember* vom Typ *StateMachine*) gespeichert würden. Diagramme sind nur eine bestimmte Sichtweise auf Modelle, was im ersten Blick in XMI-Dateien auch bestätigt wird. So finden sich in den *Diagram*-Elementen detaillierte Anzeigeeigenschaften (wie Positions-, Geometrie- und Farbangaben) und Referenzen auf die Modellelemente die im *Model*-Element definiert sind. Wie der folgende, vereinfachte Codeausschnitt zeigt sind im *Model*-Element Untermodelle definiert, etwa auch für Zustandsmaschinen, sobald dieses erste Zustandsmodell in Visual Paradigm erstellt wurde.

```
<xmi:XMI xmi:version="2.1">
  <uml:Model>
    <ownedMember xmi:type="uml:StateMachine">
      <region xmi:type="uml:Region">
        <subvertex xmi:type="uml:State" />
        <transition xmi:type="uml:Transition" />
      </region>
    </ownedMember>
  </uml:Model>
  <uml:Diagram />
</xmi:XMI>
```

Erstellt man nun in Visual Paradigm jedoch ein zweites Zustandsdiagramm und exportiert das Projekt in eine XMI-Datei, so findet sich nicht, wie zunächst vermutet, ein weiteres

ownedMember-Element vom Typ Zustandsmaschine im *Model*-Element, jeweils mit den eigenen untergeordneten Elementen, sondern die neuen Elemente des Zustandsdiagramms werden dem vorhandenen *ownedMember* untergeordnet. Eine Recherche in allen vier betroffenen Standards UML-Infrastructure, UML-Superstructure, MOF und XMI ergab, dass in keinem der Standards die Model und Diagram-Elemente definiert zu sein scheinen. Lediglich der erlaubte Inhalt eines Model-Pakets wird angegeben, aber nicht genauer auf die genauere Repräsentation mehrerer (Zustands-)Modelle eingegangen.

Um die in Visual Paradigm im selben Projekt befindlichen, aber getrennt erstellten Diagramme auch in unterschiedliche Zustandsmaschinen parsen zu können, gibt es verschiedene Möglichkeiten. Die erste wäre auch, die Diagramm-Elemente in der XMI-Datei zu verarbeiten. Diese sind aber nicht standardisiert, auch nicht deren Inhalt. Es ist also keine Konsistenz von Visual Paradigm garantiert, geschweige denn sind die Elemente dokumentiert. Andere UML- und XMI-Tools können die Diagramm-Elemente außerdem ebenfalls nicht verarbeiten und speichern sie gegebenenfalls gar nicht. Beim Verwenden von XMI-Dateien anderer Quellen ist die Funktionstüchtigkeit des Generators damit höchst fragwürdig, beziehungsweise nicht gegeben. Aufgrund dieser gravierenden Nachteile wird eine andere Möglichkeit umgesetzt.

Eine weitere Möglichkeit ist es, die Startzustände der Zustandsdiagramme als Ausgangspunkte der Diagramme zu verwenden und die Zustandsmaschinen von diesen aus aufzubauen. In Visual Paradigm werden untergeordnete Elemente auf verschiedene Weise in einer XMI-Datei gespeichert. So kann ein Zustand direkt enthaltene Zustände beinhalten, oder ein eine Region, welche wiederum Subelemente enthält. Ersteres ist der Fall bei einem Submachine-State, letzteres bei zusammengesetzten Zuständen (es sind potentiell auch mehrere Regionen in einem Zustand möglich). Dieser hierarchische Aufbau gilt jedoch nur für Zustände. Die *Transition*-Elemente, auch von untergeordneten Regionen und eingebetteten Zustandsmodellen (zusammengesetzte Zustände), sind immer auf der gleichen Ebene in der obersten *StateMachine-Region* (siehe Programmcode oben). Vor allem aufgrund der gravierenden Nachteile der ersten dargestellten Möglichkeit werden die Zustandsmaschinen, wie gerade vorgestellt, anhand ihrer Startzustände identifiziert.

5.3 Ant-Buildsystem und Jar-Generierung

Wie bereits im Kapitel Entwicklungs- und Testumgebung erwähnt, wird neben dem Eclipse-Buildsystem auch Ant verwendet. Während der Implementierung wurde zunächst ausschließlich Eclipse verwendet, bis das Programm im sogenannten *Happy-Flow*, dem Standardfall (hier die Generierung des Quellcodes aus einer XMI-Datei mit einem Zustandsdiagramm), funktionierte. Anschließend wurde in einer Ant-Builddatei das Buildziel zur Jar-Generierung erstellt. Dieses kopiert die kompilierten Klassen in eine Jar-Datei und platziert diese, zusammen mit Kopien der notwendigen Bibliotheken und Template-Dateien, im Distributionsordner. Darauf aufbauend wurde noch ein weiteres Buildziel hinzugefügt, welches auch die Bibliotheken und Templatedateien in das generierte Jar-Archiv hinzufügt. Folglich ist für die Distribution der Anwendung nur noch die Verteilung dieser einen Jar-Datei notwendig. Ein weiteres Buildziel für das Leeren der *Bin*- und Distributionsordner wurde ebenfalls hinzugefügt, sowie eines für das Kopieren der Templatedateien in den *Bin*-Ordner.

Bei der Generierung der Jar-Distributionsdateien stellte sich eine Herausforderung bezüglich der Templatedateien. Zuvor wurde die Template-Engine FreeMarker so verwendet, dass diese einen Ordner übergeben bekommt, in welchem die später jeweils zu parsenden Templatedateien zu finden sind. Dieser Ordnerpfad wurde über das aktuelle Arbeitsverzeichnis, in Kombination mit dem Unterpfad zu den Template Dateien, ermittelt. Das Arbeitsverzeichnis wurde hierbei über das System Property *user.dir* (*System.getProperty("user.dir");*) ermittelt.

Wird das Programm jedoch aus der Jar-Datei heraus gestartet, so ist das Arbeitsverzeichnis nicht mehr in einem Verzeichnis mit den Unterordnern und -dateien. Außerdem kann ein Ordner innerhalb einer Jar-Datei ohnehin nicht als ein *File*-Object referenziert werden, was notwendig wäre. Deshalb musste eine andere Möglichkeit gefunden werden, die Template Engine mit den Templatedateien aus der Jar-Datei arbeiten zu lassen.

Dateien in Jar-Archiven können über einen *ClassLoader* als Ressourcen geladen werden. Eine programmatische Lösung wäre das Herauskopieren der Templatedateien in einen temporären Ordner gewesen, welcher dann der Template-Engine übergeben wird. Es stellte sich jedoch heraus, dass FreeMarker nicht nur das Laden aus Ordnern unterstützt, sondern auch mit Ressourcen arbeiten kann. Hierzu wird, wie im diesem Absatz folgenden Codestück zu sehen ist, für die Konfiguration die passende Methode *setClassForTemplateLoading* aufgerufen.

Dieser wird als erster Parameter eine Laufzeitklasse als Objekt übergeben. Über diese ließe sich der *ClassLoader* auch manuell aufrufen. Als zweiter Parameter wird ein Präfix für die Ressourcen, beziehungsweise den Ressourcenpfad, angegeben. Damit der Pfad absolut zum Arbeitsverzeichnis, also dem Hauptverzeichnis im Jar-Paket ist, beginnt der Präfix mit einem Schrägstrich. Da diese Angabe der Ressource also absolut erfolgt, ist auch unwesentlich, welches Laufzeitklassenobjekt konkret übergeben wird, so lange es von der aktuellen Programminstanz aus der Jar-Datei ist.

```
Configuration tmplCfg = new Configuration();
tmplCfg.setClassForTemplateLoading(this.getClass(), "/templates/");
```

Die Template-Engine kann nun problemlos, sowohl im Falle einer Ausführung vom Dateisystem wie auch komplett aus einer Jar-Datei heraus, auf die Templatedateien zugreifen. Die Templateverarbeitung selbst wird im Programmcode über die Funktionen *getTemplate* und *process* angestoßen, wie der folgende Beispielprogrammcode zeigt.

```
Template tmpl = tmplCfg.getTemplate("transitionfromto.ftl");
Map<Object, Object> tmplData = new HashMap<Object, Object>();
tmplData.put("classname", classname);
Writer out = …;
String result = process(tmplData, out);
```

Mit der Funktion *getTemplate* wird die Templaterepräsentation geholt. Anschließend wird, über die Funktion *process* und einer ihr übergebenen *HashMap*, die Templatedatei verarbeitet. Das Ergebnis kann über einen *Writer* ausgelesen werden. Die übergebene *HashMap* enthält die Werte für die Platzhalter im Template. Zum einfacheren Parsen wurde in der *TemplatorFreeMarker* Klasse (siehe Kapitel Datei- und Paketstruktur) eine Hilfsmethode umgesetzt, welcher man nur noch das *Template* und die Parameter-*HashMap* übergibt. Das Ergebnis wird als String zurückgeliefert. Ein Hantieren mit *Writer*-Objekten ist nicht mehr notwendig.

Neben der Template-Engine, die auf die Templates als Ressourcen zugreifen muss, müssen die statischen Java-Dateien der allgemeinen generierten Klassen ebenfalls, als Ressourcen aus der Jar-Datei, in das Zielverzeichnis kopiert werden. Hierzu wird, analog zum Vorgehen der Template-Engine, über das Laufzeitklassenobjekt auf die Dateien als *InputStream* der Ressourcen zugegriffen und deren Inhalt in den Zielordner, in die entsprechenden, angelegten Dateien kopiert.

5.4 Ivy-Abhängigkeitsauflösung

Ivy ist ein Unterprojekt des Ant-Projektes. Entwickelt wird es unter der Leitung der Apache Software Foundation. Es erweitert Ant um Funktionen zur Abhängigkeitsauflösung. In einer XML-Datei (typischerweise *ivy.xml* benannt) werden hierzu die Bibliotheken angegeben, von welchen das Projekt abhängt. Über ein Ant-Buildtarget kann dann eine Abhängigkeitsauflösung angestoßen werden, wodurch die notwendigen Bibliotheken von einem, meist öffentlichen, Repository in ein vorher angegebenes Verzeichnis heruntergeladen werden. Neben den Bytecode-Bibliotheken können auch Jar-Pakete mit Quellcode oder Javadoc heruntergeladen werden, um diese in der IDE anschließend mit den Bytecode-Bibliotheken verknüpfen zu können. Für das StateXMI2Java-Projekt wurden solche Bibliotheksabhängigkeiten ebenfalls definiert. So kann der Quellcode des Projektes ohne Bibliotheken kopiert werden und die Bibliotheken können später mit dem Ausführen des Buildtargets heruntergeladen werden. Somit müssen die Bibliotheken auch nicht in das Versionsverwaltungsrepository hinzugefügt werden, sondern diese Abhängigkeiten können auf einem Entwicklungsrechner einfach aufgelöst werden.

6. Ergebnis, Ausblick und Fazit

In diesem Kapitel wird der aktuelle Stand der Entwicklung, des im Rahmen dieser Bachelorarbeit erstellten Programms, aufgezeigt, sowie mögliche Weiterentwicklungen des Programms aufgeführt. Abschließend wird ein kurzes Fazit gezogen.

6.1 Stand der Entwicklung

Das StateXMI2Java-Projekt ist, mit den im Kapitel Analyse dargestellten Anforderungen, umgesetzt. Vorgegangen wurde nach der im Kapitel Design vorgestellten Planung, sowie nach den im Kapitel Implementierung erläuterten Schritten und Anpassungen. Der Wizard ist aus einer auslieferbaren Jar-Datei ausführbar. Er liest XMI-Dateien der Version 2.1, welche UML-Modelldaten der Version 2.3, aber vor allem UML-Zustandsmodelle, enthält. Sind mehrere Zustandsmaschinen in der XMI-Datei definiert, so kann der Benutzer eines der Modelle als Quelle auswählen. Aus dem Quell-Zustandsmodell wird dann, unter anderem mit Hilfe von Templatedateien, Java-Quellcode generiert. Dieser ist objektorientiert und generisch gestaltet. Sowohl der XMI-Import, als auch die Template-Verwendung werden mit Unit-Tests geprüft. Getestet wurde die Anwendung bisher lediglich mit XMI-Dateien, die aus Visual Paradigm exportiert wurden. Aufgrund der Standardisierung von XMI sollte die Verwendung von XMI-Dateien aus anderen Quellen jedoch kein Problem darstellen und wenn, dann nur geringe Probleme verursachen.

Der Wizard besteht aus 3 Seiten, wenn keine Probleme während der Ausführung auftreten und in der XMI-Datei mehrere Zustandsmodelle gespeichert sind. Diese Seiten sind in den Abbildungen 20, 21 und 22 dargestellt. Abbildung 20 zeigt die erste Seite. Sie erlaubt es dem Benutzer die XMI-Datei auszuwählen, aus welcher die Zustandsmaschine oder die Zustandsmaschinen gelesen werden. Wird nur ein Zustandsmodell gefunden, so wird direkt der Quellcode generiert und die Abschlussseite angezeigt, welche in Abbildung 22 zu sehen ist. Werden dagegen mehrere Zustandsmodelle gefunden, so wird die in Abbildung 21 dargestellte, zweite Wizard-Seite angezeigt. Sie erlaubt es dem Benutzer, anhand der benannten Startzustände, das von ihm gewünschte Zustandsmodell zu wählen, aus dem der Quellcode generiert wird. Wird in der XMI-Datei kein Zustandsmodell gefunden, oder tritt beim Lesen der XMI-Datei oder

bei der Generierung des Quellcodes ein Fehler auf, so wird dem Benutzer auf einer Fehlerseite das Problem mitgeteilt. Ist die Generierung des Quellcodes erfolgreich, so wird die in Abbildung 22 dargestellte, letzte Wizard-Seite angezeigt. Diese bietet dem Benutzer noch an, das Zielverzeichnis, in dem der Quellcode erzeugt wurde, zu öffnen.

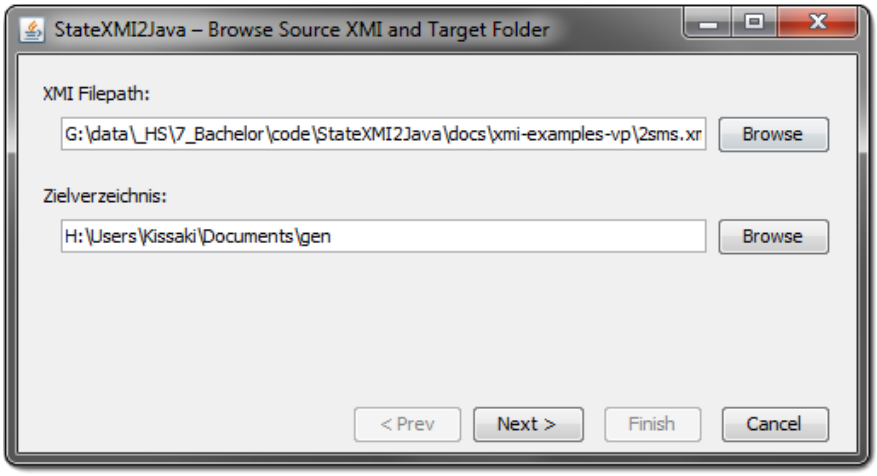

Abbildung 20: Erste Wizard-Seite; Wahl der Quelldatei und des Zielverzeichnisses

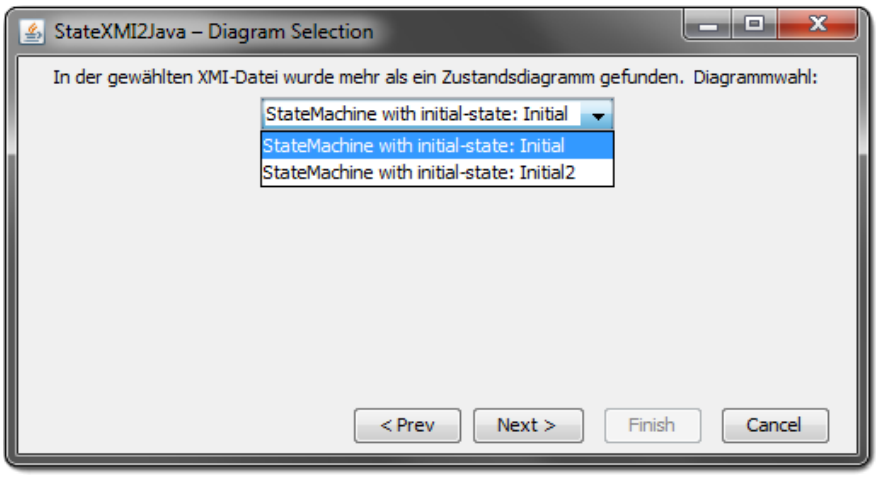

Abbildung 21: Zweite Wizard-Seite; Wahl einer Zustandsmaschine

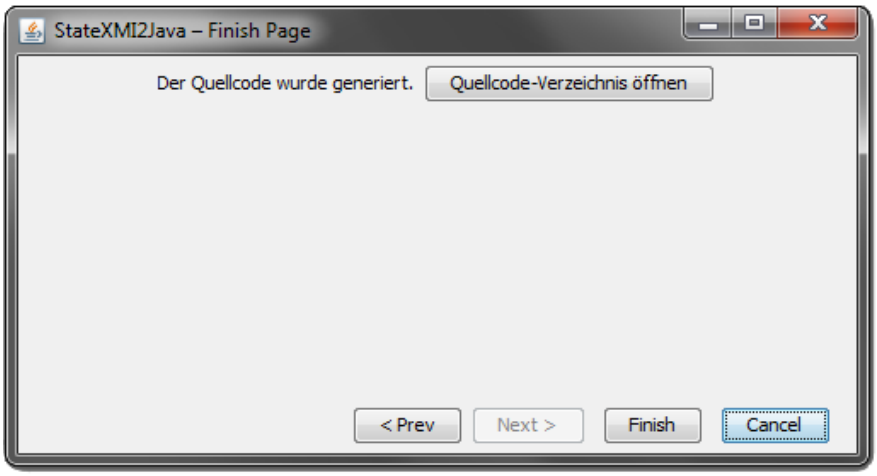

Abbildung 22: Dritte Wizard-Seite; Die Generierung wurde abgeschlossen

Um das Laufzeitverhalten bezüglich Speicher und Zeit zu analysieren, wurde eine große XMI-Datei, für einen Stresstest, erstellt. Hierzu wurde in Visual Paradigm ein Projekt angelegt. Dieses enthält 22 Diagramme und die dazugehörigen Modelle. Darunter vier Zustandsdiagramme, eines davon mit einem hierarchischen Zustand. Die Zustandsdiagramme und das Zustandsmodell haben insgesamt über 100 Zustände. Die aus diesem Projekt exportierte XMI-Datei hat eine Größe von 1,8 MB. Das Parsen dieser XMI-Datei dauert knapp zwei Sekunden. Das Erstellen des Quellcodes für ein Zustandsmodell ist unter einer Sekunde abgeschlossen. Der gesamte Java-Prozess belegt dabei zwischen 40 und 56 Megabyte Arbeitsspeicher. 40 MB nach dem Start des Programms und 56 MB nach dem Parsen der XMI-Datei und dem Generieren des Quellcodes.

6.2 Mögliche Erweiterung und Weiterentwicklung

Das aus dem Projekt hervorgegangene Programm deckt die zu Beginn definierten Anforderungen ab. Es sind jedoch, wie in dieser Thesis auch bereits dargestellt wurde, einige Erweiterungen möglich. Im Kapitel Quellcode-Merging oben wurde ausführlich vorgestellt, welche Möglichkeiten es zum Mergen von wiederholt generiertem Quellcode gibt, und dass das Markieren generierter Methoden mit Javadoc-Tags die beste Strategie ist. Diese Strategie könnte relativ einfach und ohne Änderungen an der GUI umgesetzt werden.

Hierarchische Zustände wurden im Kapitel Hierarchische Zustände vorgestellt und Möglichkeiten dargelegt, wie diese zu verarbeiten sind. Hierarchische Zustände können hierbei

zusammengesetzte Zustände oder Untermaschinen-Zustände sein. Für beide kann ein gemeinsamer Mechanismus genutzt werden.

Die Umsetzung als standalone Wizard ermöglicht das einfache und problemlose Transportieren und Ausführen des Programms. Denkbar wäre jedoch eine Umsetzung des Wizards als Eclipse-Erweiterung. Hierzu würden die CJWizard-Komponenten wegfallen und durch eine Eclipse-spezifische GUI ersetzt. Dies würde eine in den Entwicklungsprozess integriertere Verwendung des Generators ermöglichen, sofern ohnehin Eclipse als IDE verwendet wird.

Im Kapitel Aktivitätsobserver statt Aktivitätsmethoden wurde dargestellt, wie das Observer-Pattern auf die Zustandsaktivitäten angewandt werden kann, um die Redundanz und Komplexität bei der Implementierung einiger weniger und häufig verwendeter Aktivitäten reduzieren zu können. Dies könnte als Fortführung des Projektes ebenfalls eingebaut werden.

Zu guter Letzt wäre die einfachste Fortführung eine Erweiterung des Programms auf weitere Programmiersprachen. Durch die Auslagerung in Template-Dateien ist eine Portierung auf Java-ähnliche, objektorientierte Programmiersprachen sehr einfach. Aber auch über eine weitere Generatorklasse, welche Code*Generator* implementiert, können weitere Programmiersprachen, dann mit mehr Kontrolle, hinzugefügt werden.

6.3 Fazit

Diese Bachelorarbeit hat einen Überblick über die Quellcodegenerierung aus UML-Zustandsdiagrammen aus dem XMI-Dateiformat gegeben und ein Projekt vorgestellt, in welchem ein solcher Quellcodegenerator erstellt wurde. Trotz einiger Herausforderungen und der zusammenspielenden Standards wurden stets gute Lösungen gefunden und evaluiert. Als Ergebnis wurden in dieser Arbeit einerseits die gewonnenen Erkenntnisse vorgestellt und andererseits steht nun ein eigenständiger Wizard zur Verfügung, welcher gut designten Quellcode erzeugt und um weitere Funktionen erweiterbar ist.

Literaturverzeichnis

UMLS10:Object Management Group (OMG). 2010. UML 2.3 Superstructure. OMG

UMLI10:Object Management Group (OMG). 2010. OMG Unified Modeling Language (OMG UML), Infrastructure. OMG

JMI02:Java Community Process; Ravi Dirckze, Unisys Corporation. 2002. Java Metadata Interface (JMI) Specification. Sun / Java Community Process

Abbildungsverzeichnis

Tabellenverzeichnis

Schlussnotiz

Die Software StateXMI2Java wird quelloffen, wahrscheinlich unter einer freien Lizenz, veröffentlicht werden. Informationen hierzu und zum Programm selbst, sollten mit Verfügbarkeit dieser Bachelorarbeit in der Bibliothek (~Mitte März) auf http://kcode.de zu finden sein.