Automatisierte Analyse und Visualisierung des Java-TX Quellcodes

Studienarbeit

des Studienganges IT-Automotive 2014
an der Dualen Hochschule Baden Württemberg

von

Jan-Elric Neumann

Stand 02.06.2017

Bearbeitungszeitraum	8 Monate
Kurs / Matrikelnummer	STG-TINF14ITA / 4997183
Gutachter	Dr. Martin Plümicke

Selbstständigkeitserklärung

Thema: „Automatisierte Analyse und Visualisierung des Java-TX Quellcodes“

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Falls sowohl eine gedruckte als auch elektronische Fassung abgegeben wurde, versichere ich zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Hildesheim, 02.06.2017

Jan-Elric Neumann

Abstract

Java-TX is a Java-Compiler currently in development at the Duale Hochschule Baden-Württemberg. For approximately 10 years have around 20 students and professors worked on the implementation of this compiler.

Aim of Java-TX is to provide the ability to automatically deduct the type of a variable, the return type of a method or any other piece of code which usually needs to explicitly declare a type.

The code of Java-TX builds on UML-Models which were created with a deprecated UML-Tool.

In this study we use another tool, ObjectiF by microTOOL GmbH, to parse the existing code to create new and up-to-date UML-models. ObjectiF doesn't only support the creation of simple UML-Models but gives the ability to conduct the full scale of model-driven-development and round-trip-engineering. In the theoretical part of this study

Is explained what model-driven-development means and how it can be used to speed up the development process of software.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AC-MDSD	Architecture Centric Model Driven Softwaredevelopment
AJAX	Asynchronous JavaScript and XML
AST	Abstract Syntax Tree
BPMN	Business Process Model and Notation
CIM	Computation Independent Model
CORBA	Common Object Request Broker Architecture
DIE	Integrated Development Environment
EJB	Enterprise Java Beans
ISM	Implementation Specific Model
MDD	Model Driven Development
MDSD	Model Driven Softwaredevelopment
OMG	Object Modelling Group
PIM	Platform Independent Model
PSM	Platform Specific Model
UML	Unified Modelling Language
WLS	Oracle WebLogic Server
XML	Extensible Markup Language

Abbildungsverzeichnis

Listings

1. EINLEITUNG

Seit ca. 10 Jahren wird an der DHBW der Java-Compiler Java-TX entwickelt. Hauptfunktion von Java-TX ist es, Java um die automatische Berechnung von Variablentypen zu erweitern. Durch bisher 20 Studenten der DHBW wurde dieser Compiler entwickelt und in die Entwicklungsumgebung Eclipse eingebunden. Die Implementierung des Java-TX-Compilers basiert auf veralteten UML-Modellen, die mit einem UML-Tool erzeugt wurden, welches stark veraltet ist.

1.1 Aufgabenstellung

Aufgabe dieser Studienarbeit ist es, aus dem bestehenden Quellcode des Compilers durch die Verwendung des Tool ObjectiF der microTOOL GmbH, neue UML-Modelle zu generieren. Die Erzeugung der Modelle soll möglichst automatisiert stattfinden.

Zusätzlich soll in einem theoretischen Teil der Arbeit das Softwareentwicklungs-Paradigma „Model-Driven-Development“ erschlossen werden.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst die Grundlagen des Model-Driven-Development und den Unterschieden zum Round-Trip-Engineering erläutert.

Kapitel 3 schließt mit der Vorstellung eines Softwaretools an, welches Entwickler bei der Anwendung von Model-Driven-Development und Round-Trip-Engineering unterstützt.

Der Umfang des Java-TX Projekts wird im Kapitel 4 erläutert.

Im Kapitel 5 wird eine Case Study durchgeführt, welche die Verwendung von ObjectiF (siehe Kapitel 3) für das Round-Trip-Engineering erörtert.

Die Vorbereitung des Java-TX Projekts für das Round-Trip-Engineering wird in Kapitel 6 behandelt.

Abschließend wird ein Ausblick für eine mögliche weitere Studienarbeit im Kapitel 7 gegeben.

2. MODEL-DRIVEN-DEVELOPMENT

Das Model-Driven-Development (kurz: „MDD“), zu Deutsch „Modell getriebene Entwicklung“, oft auch als Model-Driven-Software-Development (kurz: „MDSD“), „Modell getriebene Softwareentwicklung“, ist eine Methode der Softwareentwicklung, die sich stark auf Modelle als Beschreibungswerkzeug stützt (1 S. 1). Eine formale Definition von MDD existiert nicht, stattdessen ist MDD ein Überbegriff von Softwareentwicklungs-Prozessen, welche über verschiedene Abstraktionsebenen hinweg Modelle für den Entwicklungsprozess einsetzen.

Generell kann zwischen zwei Arten des MDD unterschieden werden: interpretatives und generatives MDD (2).

Interpretatives MDD ist dadurch bestimmt, dass am Ende des Softwareentwicklungs-Prozess ausschließlich Modelle stehen. Diese Modelle werden von eine Laufzeitumgebung interpretiert und ausgeführt. Eine Codegenerierung findet nicht statt.
Das generative MDD stellt im Gegensatz dazu an das Ende des Entwicklungsprozesses Quellcode, der durch die Modelle (teilweise) generiert ist. Im weiteren Verlauf wird nur generatives MDD berücksichtigt.

2.1 Motivation

Die Softwareentwicklung befindet sich in einem stätigen Evolutionsprozess hinzu effektiveren Methoden. Anfänglich standen Programmieren Methoden mit einem engen Bezug zu der Verwendeten Hardware zur Verfügung, beispielsweise das direkte Schreiben von Maschinencode oder Assemblersprachen. Durch prozedurale Sprachen (z.B. C) und Compilern wurde die Hardware ein einem ersten Schritt für den Entwickler abstrahiert. Heute stehen Objekt-Orientierte Programmiersprachen zur Verfügung, die Teilweise in Runtime-Umgebungen (siehe JAVA) ausgeführt werden, mit dem Ziel, die Software soweit wie möglich von der ausführenden Hardware zu abstrahieren.
Als ein weiterer Schritt in dieser Evolution kann das MDD interpretiert werden (3 S. 14). Ziel des Model-Driven-Development ist es, ein Prozess zu schaffen, der es unter anderem ermöglicht, die Entwicklungszeit und Qualität einer Software zu erhöhen.

Schwer zu verwaltende Redundanzen sollen vermieden werden, Teile einer Software sollen für andere, ähnliche Software verwendet werden können und die Komplexität soll durch Abstraktion, unterstützt durch Modelle, vereinfacht werden (3 S. 13). Durch die Verwendung verschiedener Abstraktionsebenen soll die Kopplung zwischen einer allgemeinen Problemlösung und der Umsetzung auf einer Zielplattform entfernt werden.

Jede Anwendung beinhaltet eine Struktur bzw. Architektur, ausgedrückt in ihrem Quellcode. Wie deutlich diese Architektur im Quellcode erkennbar ist, entscheidet maßgeblich über die Qualität, Performance, Erweiterbarkeit und Portabilität der Anwendung (3 S. 14). Diese Architektur jedoch ausschließlich aus dem Programmcode heraus zu erfassen wird mit dem Umfang des Quellcodes immer schwerer. Die Verwendung von Modellen, um diese Architektur festzuhalten ist mittlerweile weit verbreitet. Der Entwickler ist verantwortlich dafür, die Software anhand der Modelle umzusetzen und Modelle bei Änderungen im Code anzupassen. Mit steigendem Zeitdruck entfallen jedoch oft diese Aktualisierungen. Die führt dazu, dass der Code und die Modelle nicht mehr konsistent sind. Als Abhilfe werden Methoden wie das Reverse- oder Round-Trip-Engineering genutzt. Hierbei werden durch automatisierte Prozesse Model und Code immer auf einem Konsistenten Stand gehalten. Ein großer Nachteil dieser Methoden ist jedoch, dass Code und Modell konsequenterweise auf derselben Abstraktionsebene stehen.

Im Gegensatz hierzu führt das MDD an, Anpassungen an dem Code der Software stets am Modell durchzuführen. Durch verschiedene Schritte wird aus den Modellen dann der Code erzeugt. Der Vorteil hieraus besteht darin, dass die Architektur der Software an einer Stelle definiert ist. Durch die Generierung des Codes aus den Modellen kann gewährleistet werden, dass die Architektur in allen Teilen der Software umgesetzt wird.

Generell lässt sich der Code einer Software in drei elementare Teile unterteilen (3 S. 16):Generischer Code, der für alle Anwendungen gleich ist. Hierrunter fallen alle Frameworks (intern oder extern) die für jede Anwendung, die diese nutzt, gleich ist. Schematischer Code, der sich zwar in Teilen unterscheiden wird, aber einem gleichen Muster folgt. Dies können z.B. gängige Designmuster sein. Diese beiden Teile können unter dem Begriff „Infrastrukturcode“ zusammengefasst werden. Sie erfüllen primär

keine der funktionalen Anforderungen einer Software, sind jedoch elementar für die Umsetzung dieser. Der Anwendungsspezifischer Teil ist je Anwendung individuell und setzt die funktionalen Anforderungen um.

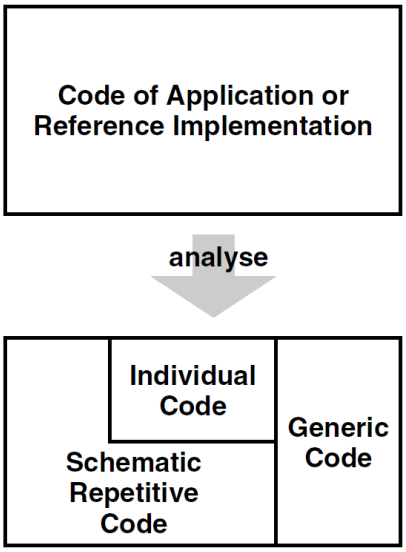

Abbildung 2.1 Unterteilung von Anwendungscode (3 S. 15)

Abbildung 2.1 zeigt schematisch diese Abhängigkeit. Im allgemeinen kann jeder dieser drei Teile leer sein, Untersuchungen zeigen jedoch, dass typischer Weise der Anteil von generischem und schematischem Code 60% - 70% einer Anwendung ausmachen (3 S. 22). Diese Teile einer Software sind dadurch geprägt, dass sie sehr oft repetitiv sind und daher durch „Copy & Paste“ Arbeit erzeugt werden. Hierdurch sind dies auch die fehleranfälligsten Teile der Software.

Einige Varianten des MDD zielen darauf ab, diesen Teil der Software durch Modelle zu abstrahieren und automatisch zu erzeugen. Dies hat den Vorteil, dass die Entwicklungszeit stark verringert werden kann und der wichtigste Teil, der individuelle Code, im Fokus der Entwicklung stehen kann. ^

2.2 **Terminologie**

Auch wenn das MDD nicht formal Definiert ist, gibt es viele Begriffe und Konzepte, die Varianten übergreifend verwendet werden.

2.2.1 Plattform

Eine Plattform beschreibt eine oder eine Menge an Ressourcen, die genutzt werden, um ein System zu realisieren (4 S. 9). Eine Plattform kann viele verschiedene Formen annehmen. Beispielsweise kann eine Plattform eine Middleware („CORBA“, „.NET“), eine Technologie wie „AJAX“, eine Programmiersprache (C, C++, Java, Python, etc.) oder die Hardware selbst sein. Eine einzelne Plattform wiederum kann auf anderen Plattformen basieren, z.B. kann für „AJAX“ die Formatierung der gesendeten Daten durch XML oder JSON (beide Stellen eine eigene Plattform da) erfolgen.

2.2.2 Modell

Das Modell stellt die wichtigste Einheit für das MDD dar. Ein Modell ist eine abstrakte Darstellung einer Struktur, Funktion oder Verhalten eines Systems (3 S. 18). Im Kontext der Modelle im MDD zu beachten, dass Abstrakt keine Mehrdeutigkeit impliziert. Ein Modell, unabhängig der Abstraktionsebene, hat dieselbe Bedeutung wie der Quellcode einer Anwendung.

Generell gibt es im MDD drei Abstraktionsebenen, dass Platform-Independent-Model (kurz: PIM), zu Deutsch „Plattform unabhängiges Modell“, dass Platform-Specific-Model (kurz: PSM), zu Deutsch „Plattform spezifisches Modell) und das Implementation-Specific-Model (kurz: ISM), zu Deutsch „Implementationsspezifisches Modell. Einige Varianten des MDD führen zusätzlich noch das Computation-Independent-Model (kurz: CIM) als höchste Abstraktionsebene ein. Oft ist diese Ebene jedoch in der PIM-Ebene mit einbegriffen.

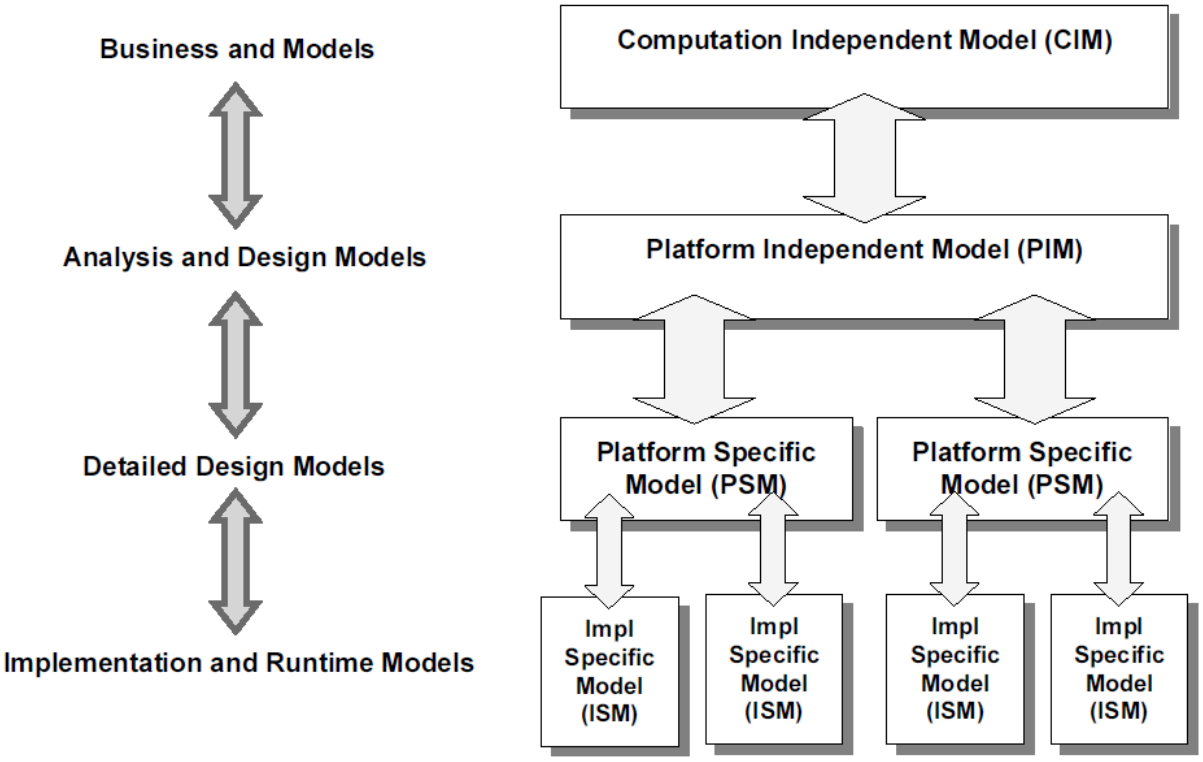

Abbildung 2.2 Zusammenhang der Abstraktionsebenen (1 S. 7)

Abbildung 2.2 zeigt die Abstraktionshierarchie zwischen den einzelnen Modellebenen auf.

Das CIM, als höchste Abstraktionsebene, modelliert das zu entwickelnde System komplett unabhängig von einer möglichen späteren Umsetzung. Es „ist unabhängig von technischen Aspekten der Implementierung, d.h. es beschreibt keine internen Strukturen oder das innere Verhalten des Softwaresystems.“ (5) Das CIM wird in einer Sprache verfasst, die von der Domäne der Anwendung abhängig ist. Zur Modellierung kommen hier Aktivitäts-, Interaktions- und Use-Case-Diagramme zum Einsatz (5).

Das PIM modelliert das System unabhängig von einer späteren Plattform, die verwendet wird, um das System zu realisieren. Auch hier werden keine Konzepte einer bestimmten Plattform zur Beschreibung einzelner Elemente des Systems verwendet, sondern Konzepte, die sich aus der Domäne der Anwendung ableiten. Im Gegensatz zum CIM kann hier jedoch bereits eine gewisse Architektur der Software festgelegt

werden. CIM und PIM stehen in sehr enger Verbindung zueinander, weshalb diese in vielen Fällen zusammengefasst werden.

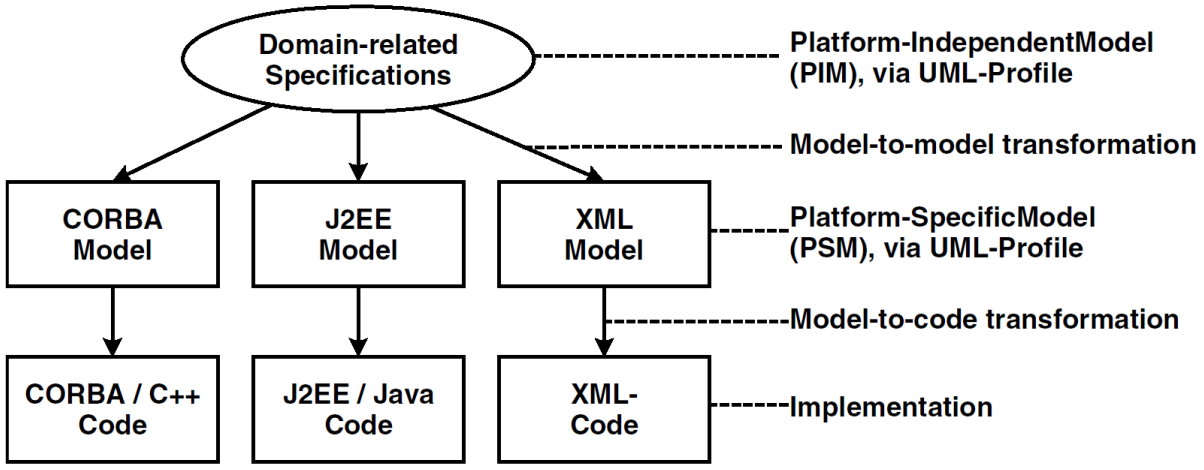

Abbildung 2.3 Zusammenhang PIM, PSM und ISM (3 S. 16)

Jedes System ist im Regelfall durch ein PIM modelliert. Im Gegensatz dazu können aus einem PIM mehrere PSMs abgeleitet werden. Die Modellierung eines PSM findet nicht mehr durch Konzepte der Domäne der Anwendung statt, sondern durch Konzepte der entsprechenden Plattform. Da eine Plattform, wie bereits beschrieben, sehr vielfältig sein kann, sind die Konzepte von PSM zu PSM sehr verschieden. Beispielsweise kann ein PIM gleichzeitig in einer Objekt-Orientierten Sprache wie C++ oder prozeduralen Sprache wie C transformiert werden. Das PSM von C++ kann dann Konzepte dieser Sprache nutzt, wie Klassen, Methoden-Overloading oder Templates nutzen. Das PSM von C muss jedoch auf diese Konzepte verzichten, da diese dort nicht existieren (siehe Abbildung 2.3).

Zu beachten ist, dass im PSM keine Implementierungen festgelegt werden. Im Beispiel von C++ wird ein PSM festlegen, dass es eine Klasse X geben wird, jedoch wird hierfür nicht der entsprechende Code erzeugt, sondern diese Klasse beispielsweise von UML modelliert.

Die Codegenerierung findet in der Transformation von PSM zu ISM statt. Hier wird aus den Modellen und Konzepten des PSM lauffähiger Code erzeugt, der die Software darstellt. Je nach Interpretation kann dieser entstandene Code ebenfalls als ein Modell

angesehen werden, jedoch mit einer sehr geringen Abstraktion. Das ISM ist das gewünschte Endprodukt des generativen MDD.

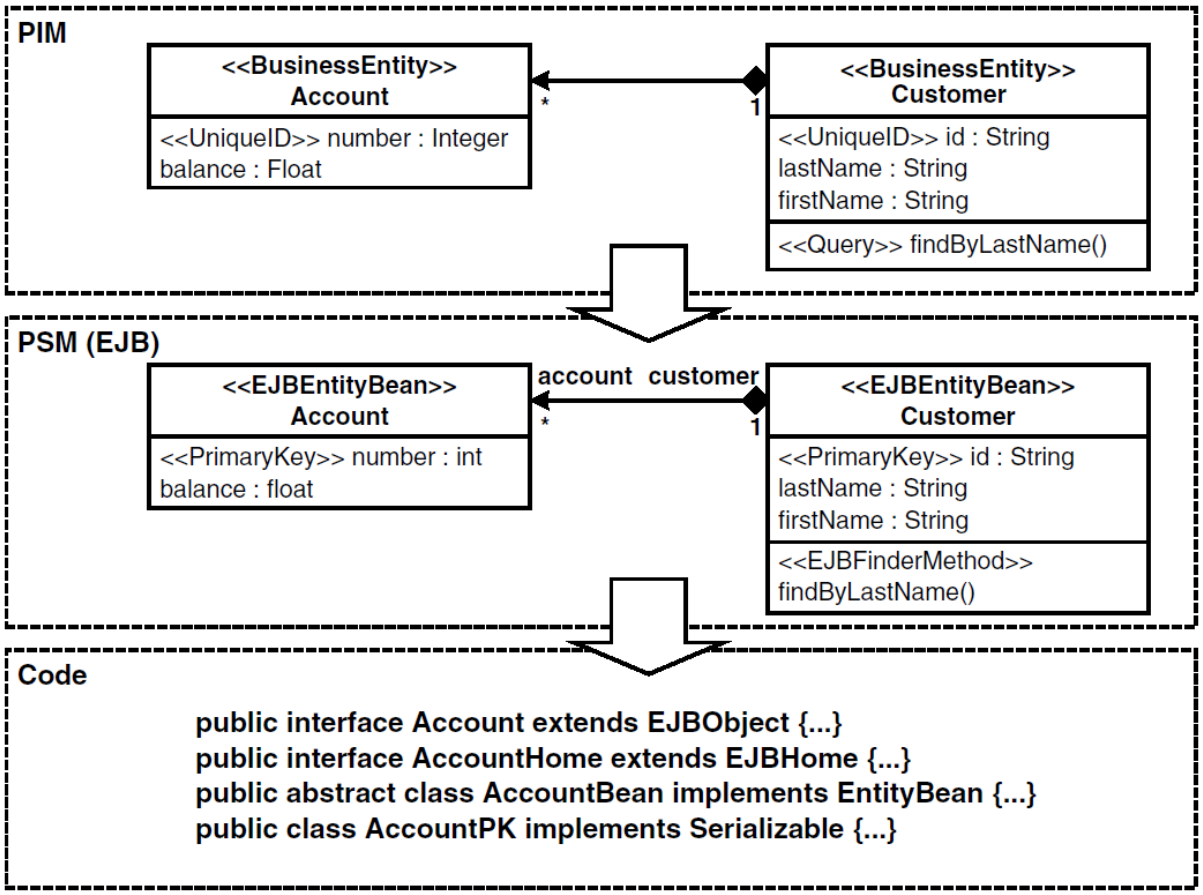

Abbildung 2.4 Konzepte des PIM, PSM und ISM (3 S. 17)

Anhand von Abbildung 2.4 soll der Begriff „Konzept“ im Kontext von MDD noch einmal erläutert werden. Die Abbildung zeigt die „Evolution“ einiger Konzepte vom Kontext des PIM über ein PSM bis hin zu einem ISM (Code). An dieser Stelle ist es für das Beispiel irrelevant, welche Bedeutung diese einzelnen Konzepte im konkreten besitzen.

Das PIM definiert unabhängig von dem späteren PSM zwei Einheiten, „Account“ und „Customer“, die das PIM-Konzept „<<BusinessEntity>>“ besitzen. Beide Einheiten besitzen eine „<<UniqueID>>“ und die Einheit „<<Customer>>“ besitzt zusätzlich eine „<<Query>>“. Diese Konzepte sind denkbar auf verschiedene Arten umsetzbar

(mehrere PSMs), eine davon ist auf der zweiten Ebene dargestellt. Das PIM-Konzept „“<<BusinessEntity>>“ wird auf das PSM-Konzept „<<EJBEntityBean>>“, „<<UniqueID>>“ auf „<<PrimaryKey>>“ und „<<Query>>“ auf „<<EJBFinderMethod>>“ abgebildet. „<<EJBEntity>>“, „<<PrimaryKey>>“ und „<<EJBFinderMethod>>“ sind Konzepte des Ziel-PSMs EJB (Enterprise JavaBeans). Im letzten Schritt findet eine Abbildung der Konzepte des PSM (EJB) auf das ISM (Java-Code) statt.

2.2.3 Transformation

MDD wird erst dann hilfreich, wenn aus dem PIM die PSMs, und aus den PSMs wiederum die ISMs generiert werden können. Dieser Aspekt ist zentral für das MDD. Im Allgemeinen nimmt ein Transformator als Eingabe ein bestehendes Modell und erzeugt daraus ein anderes Modell oder Code. Es gibt drei Arten von Transformationen: Refactoring, Modell-zu-Modell und Modell-zu-Code (1 S. 11).

Refactoring-Transformatoren arbeiten innerhalb einer Ebene (Bsp. PIM) um dort Veränderungen vorzunehmen.

Modell-zu-Modell-Transformatoren setzen ein Modell in ein anderes Modell um. Diese Transformatoren werden genutzt, beispielsweise das PIM in ein PSM umzusetzen. Generell gibt es je PSM ein oder mehrere Transformatoren, die das PIM in dieses eine PSM umwandeln können. Die Transformatoren sind also spezialisiert auf dieses PSM. Modell-zu-Code-Transformatoren setzen ein PSM in ein ISM um.

Transformatoren müssen von einem Anwender Programmiert oder Konfiguriert werden. Es existieren nur in trivialen Fällen allgemeingültige Transformatoren. Mittels der Anpassung der Transformatoren kann eine bestimmte Umsetzung eines Modells erreicht werden. Vorteil der Anpassung am Transformator, anstatt am erzeugten Modell bzw. Code, liegt darin, dass sich die Änderungen universell auf allen entsprechenden Modellen auswirken. Änderungen gehen auch dann nicht verloren, wenn das hierarchisch höher liegende Modell verändert wird und die Transformation erneut durchgeführt wird.

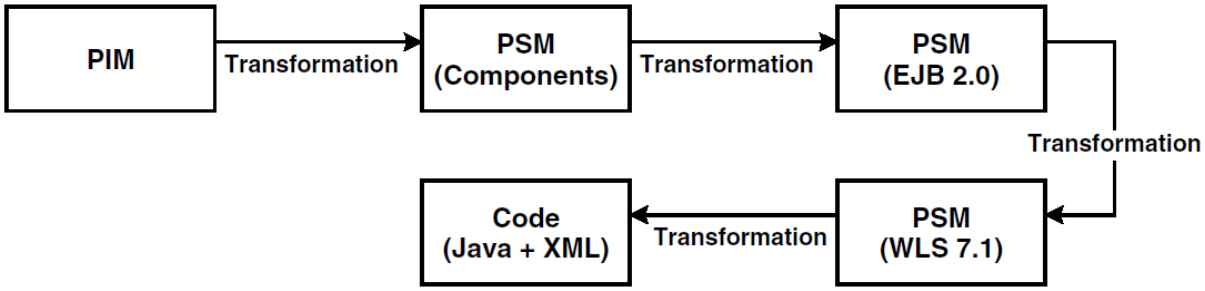

Abbildung 2.5 Transformationskette (3 S. 17)

Abbildung 2.5zeigt eine Kette von Transformationen vom PIM ausgehend bis hin zum Code (ISM). Deutlich wird hier noch einmal der Aspekt, dass eine Plattform ebenfalls auf anderen Plattformen aufsetzen kann. Beispielsweise ist die Umsetzung des PIM mittels EJB 2.0 unabhängig von der verwendeten Anwendungsserverumsetzung (WLS 7.1), welche als Plattform für EJB 2.0 dient (3 S. 17).

2.2.4 UML-Profile

Ein UML-Profil (Stereotyp) ist ein Mechanismus, um die UML um Konzepte zu erweitern (3 S. 19). In Abbildung 2.4 wurden diese bereits eingeführt („<<BusinessEntity>>“, etc.). Mithilfe dieser Profile ist es auf der einen Seite möglich, einzelne UML-Klassen um Konzepte zu erweitern, um so das Modell zu definieren, auf der anderen Seite können diese Profile von den Transformatoren genutzt werden. Die Transformatoren enthalten Regeln, um eine UML-Klasse eines bestimmten UML-Profils auf eine UML-Klasse eines anderen UML-Profils abzubilden, um Modell-zu-Modell Transformationen durchzuführen.

2.2.5 Model Markings und Decision Criteria

Sind die Informationen im PIM nicht spezifisch genug ist es oft möglich, dieses auf der Zielebene auf verschiedene Arten umzusetzen. Es gibt vier Möglichkeiten, dieses Problem zu lösen: Im Transformator ist fest codiert, welche Alternative zu bevorzugen

ist, der Entwickler wählt die gewünschte Alternative manuell aus, das Modell wird um „Model markings“ oder um „Decision Criteria“ erweitert.

Model markings sind zusätzliche Informationen, die dem Quellmodell der Transformation angeheftet werden (3 S. 248). Wichtig hierbei ist, dass das Quellmodell nicht direkt verändert wird. Stattdessen wird ein weiteres Modell erzeugt, welches sich auf das Quellmodell bezieht und dies um die „model markings“ erweitert. In Abbildung 2.6 wird eine Einheit eines bestehenden PIM (Account) um eine Markierung erweitert.

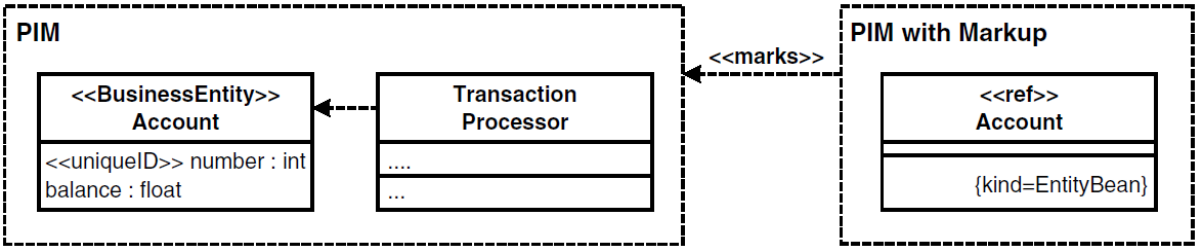

Abbildung 2.6 Beziehung zwischen Quellmodell und Marking (3 S. 249)

Alternativ dazu ist es möglich, einer UML-Klasse im Quellmodell um Kriterien zu erweitern, welche die Abbildung im Zielmodell erfüllen muss.

2.3 Architecture-Centric MSDS

Ein Beispiel für eine MDD Variante ist das Architecture-Centric Model-Driven-Software-Development (kurz: AC-MDSD).

Ziel des AC-MDSD ist es, die Entwicklungseffizienz, Software-Qualität und –Wiederverwendbarkeit zu steigern (3 S. 21). Dies wird hauptsächlich dadurch erzielt, denn Entwickler von routinierter und fehleranfälliger Arbeit zu befreien (3 S. 21). Viele Anwendungen bestehen heutzutage aus einer Menge an Teilsystemen, wie z.B. Servern, Datenbanken, Kommunikationsprotokollen, etc. Oft sind die Programm-Schnittstellen zu diesen Teilsystemen unabhängig von der konkret entwickelten Anwendung identisch bzw. schematisch gleich. Wie bereits erwähnt, macht der Anteil dieses Codes zwischen 60% - 70% der gesamten Software aus. AC-MDSD wird dazu genutzt, diesen Teil automatisiert zu generieren.

Mittels MDD wird diese Infrastruktur im PIM definiert. Durch Transformatoren werden die PSMs erzeugt. Durch weitere Transformationen wird anschließend der Infrastrukturcode generiert. Dieses Vorgehen befreit den Entwickler davon, für eine neue Anwendung diesen Code selbst zu schreiben. Aufgabe ist es nun, die eigentliche Funktionalität der Anwendung umzusetzen.

AC-MDSD setzt für diese, „per Hand“ geschriebene Teile der Software, Markierungen im Code ein (3 S. 23). Die Transformatoren analysieren den vorhandenen Code vor einer Neugenerierung und verhindern so, dass die markierten Abschnitte überschrieben werden. Änderungen am Infrastrukturteil der Software finden also auf der Ebene des PIMs (bzw. durch Anpassung der Transformatoren) statt. Die Vorteile hierdurch wurden bereits erläutert.

2.4 Unterschiede zu Reverse- und Round-Trip-Engineering

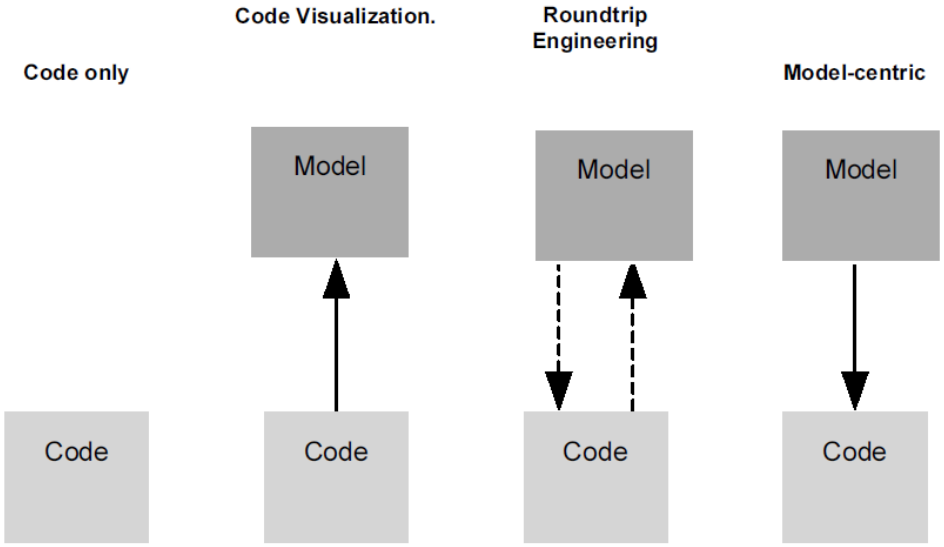

Abbildung 2.7 Modellierungsarten (1 S. 4)

Abbildung 2.7 zeigt vier verschiedene Arten der Beziehung zwischen Modellen und Quellcode.

In vielen Projekten ist der Quellcode komplett separiert von Modellen, die diesen Beschreiben sollen. Diese Modelle liegen oft in Form von Präsentationen,

Diagrammen oder als Gedanken im Kopf des Entwicklers vor. Dies führt dazu, dass die gesamte Logik und Struktur eines Programmes aus dem Quellcode abgeleitet werden muss.

Durch Reverse-Engineering (in Abbildung 2.7 „Code Visualization“) ist es möglich, aus dem Quellcode Modelle zu entwickeln. Diese Modelle helfen dem Verständnis der Programmstruktur durch die visuelle Darstellung etwas, basieren jedoch auf derselben Abstraktionsebene wie der Code selbst. Nach Änderungen am Code müssen die Modelle stets erneuert werden.

Im Round-Trip-Engineering ist die Aktualisierung der Modelle bei Codeänderungen automatisiert. Zusätzlich ist es möglich, Änderungen am Modell vorzunehmen, welche dann ebenfalls automatisch auf den Code übertragen werden.

Durch MDD Methoden (in Abbildung 2.7 „Model-centric“) wird der Code aus den Modellen generiert. Der Entwicklungsaufwand findet auf der Modellebene statt. Hierdurch ist es möglich, den Code weiter zu abstrahieren als in den vorangegangenen Methoden, da der Code eine Abbildung aus den Modellen ist.

Anhand dieser Beschreibung wird der Unterschied zwischen Reverse-, Round-Trip-Engineering und Model Driven Development ersichtlich.

Das Model Driven Development bietet gegenüber den anderen beiden Methoden den Vorteil, dass die genutzten Modelle höhere Abstraktionsebenen einnehmen können als der Code.

Dennoch bieten Reverse-, und Round-Trip-Engineering eigene Vorteile.
In Situationen, in denen eine Software bereits zu weiten Teilen besteht, ist es ohne ein umfassendes Verständnis dieser, fast unmöglich Abstraktionsebenen vergleichbar zu denen im MDD herauszustellen. Dennoch bietet die Visualisierung der Programmstruktur eine andere und oft intuitivere Darstellungsart, welches dem Verständnis einer Software für den Entwickler vereinfacht.

3. ObjectiF

ObjectiF ist ein Softwaretool der microTOOL GmbH, welches für die modellgetriebene Entwicklung von Softwareprodukten eingesetzt werden kann. Es unterstützt die Modellierungssprachen Business Process Modeling Notation (kurz: BPMN) und UML, zwei Standards verwaltet durch die OMG (4 S. 1), und die Programmiersprachen Java, C++, C# und .NET (6 S. 13).Um die Einbettung von ObjectiF in bestehende Entwicklungsumgebungen und –Prozesse zu erleichtern, unterstützt das Tool eine Integration in die IDEs „Eclipse“ und „Microsoft Visual Studio“ (6 S. 14).

ObjectiF unterstützt fast alle Formen der Code-Modell-Bindung wie sie in Abbildung 2.6 dargestellt sind. Mit dem Tool kann bestehender Code durch das Reverse-Engineering in Modelle umgewandelt werden um darauf basierend das Round-Trip-Engineering anzuwenden. Auch ist es möglich, Formen des MDD mit ObjectiF durchzuführen oder ObjectiF als reines UML-Tool zu verwenden, um das Softwaresystem zu modellieren und die Implementation unabhängig von ObjectiF durchzuführen. Modelle, Diagramme und Code sind in ObjectiF stark gekoppelt. Veränderungen an einer Stelle wirken sich auf alle anderen Bereiche aus. Wird beispielsweise eine Java-Klasse im Code durch eine Methode erweitert, wird diese Methode der UML-Repräsentation dieser Klasse hinzugefügt, und die Darstellung der UML-Repräsentation in allen Diagrammen aktualisiert. Wird eine Vererbungshierarchie einer Klasse durch eine Aktion in einem Klassendiagramm hinzugefügt, wird diese im Code automatisch übernommen.

ObjectiF unterstützt eine Auswahl an Diagrammtypen definiert durch die UML: Anwendungsfall-, Aktivitäts-, Sequenz-, Zustands-, Paket- und Klassendiagramme lassen sich mittels ObjectiF anlegen. Zudem sind Geschäftsprozessdiagrammen aus der BPMN unterstützt.

3.1 Bedienung

Das wichtigste Bedienelement in ObjectiF ist das Kontextmenü. Dies wird mit der rechten Maustaste aufgerufen. Die aufgelisteten Optionen sind namensgebend vom den Kotext abhängig, in dem das Menü geöffnet wurde. Der Kontext ist bestimmt durch

das Element, welches sich zum Zeitpunkt des Rechtsklicks unter dem Cursor befindet. Abbildung 3.1 zeigt das Kontextmenü eines ObjectiF-Package.

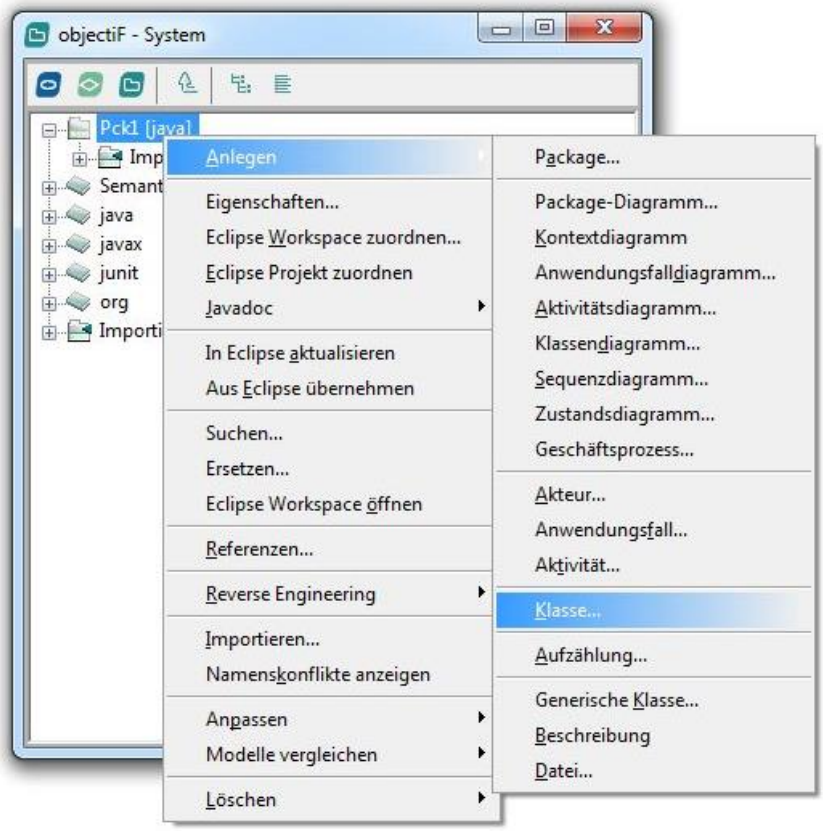

Abbildung 3.1 Kontextmenü eines Package

3.1.1 ObjectiF-System

Jedes Projekt wird in ObjectiF von weiteren Projekten das das Anlegen eines individuellen ObjectiF-Systems getrennt. Ein ObjectiF-System bündelt alle Informationen, Modelle und Konfigurationen eines Projektes. Beim Start von ObjectiF bietet das Tool eine Auswahl zwischen dem Anlegen eines neuen Systems oder das Öffnen eines bestehenden.

Beim Anlegen eines neuen Systems muss eine Systemvorlage gewählt werden. Diese Vorlage bestimmt, welche Programmiersprache für das neue Projekt verwendet werden soll. Durch die Vorlage werden dem ObjectiF-System automatisch die verfügbaren Elemente der Programmiersprache (z.B. C++ class, template, struct, etc.), ihre Basistypen (z.B. Java bool, int, string, etc.) und die Standardbibliotheken der Sprache hinzugefügt. Ebenso werden, sofern die Vorlage MDD unterstützt,

entsprechende Standardtransformatoren hinzugefügt (6 S. 38). Bei Vorlagen, die keine Programmiersprache zu Beginn festlegen, kann diese auch an einem späteren Zeitpunkt gewählt werden (6 S. 46) (siehe Abbildung 8.1, Abbildung 8.2, Abbildung 8.3, Abbildung 8.4, Abbildung 8.5).

Ein ObjectiF-System gibt drei Sichten vor, welche unterschiedliche Elemente des Projektes sichtbar machen: Fachliches Modell, Technisches Modell und System. Die Sicht System enthält alle Elemente des ObjectiF-Systems, die ersten zwei Sichten bekommen erst mit der Anwendung von MDD eine Bedeutung. Im weiteren Verlauf wird davon ausgegangen, dass die System-Sicht gewählt ist.

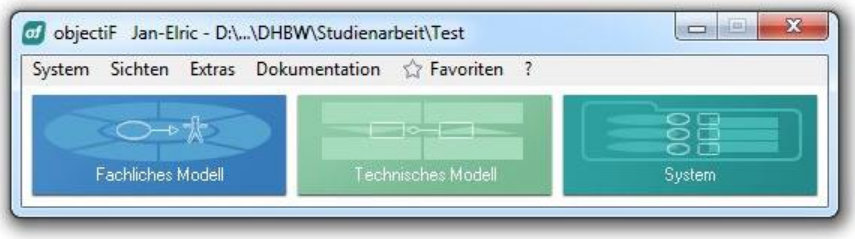

Abbildung 3.2 ObjectiF Sichten

3.1.2 Package

ObjectiF bietet die Möglichkeit, dass modellierte System in Packages aufzuteilen. Dies bietet die Möglichkeit, zusammengehörende Modelle und Diagramme in Modulen zu bündeln. Ein Package kann weitere Packages enthalten. Die Notation der Packages entspricht der UML.

Ein Package definiert in ObjectiF einen eigenen Namensraum. Elemente eines Packages haben Kenntnis von einander, sind diese Elemente zudem öffentlich, können sie auch von Elementen außerhalb des Package referenziert werden. Die öffentlichen Elemente eines Packages definieren dadurch seine Schnittstellen (Ports).

Besitzt die Programmiersprache des ObjectiF-Systems ebenfalls ein Konzept, welches mit einem Package vergleichbar ist, wird der generierte Code ebenfalls in solchen unterteilt.

Ein Package kann mit einem Stereotypen versehen werden. Im Kontext des MDD könnten diese Stereotypen „<<Platform Independent Model>>“ und „<<Platform Specific Model>>“ lauten. Sollte das ObjectiF-System keine Programmiersprache

definiert haben, oder möchte man die Programmiersprache innerhalb eines Package ändern, kann einem Package eine Zielsprache zugewiesen werden. Im Kontext des MDD ermöglicht dies, mehrere PSMs mit potentiell unterschiedlichen Zielsprachen innerhalb eines ObjectiF-Systems zu definieren.

Ein Package wird durch das Aufrufen des Kontextmenüs in dem Sichtfenster von ObjectiF möglich. Befindet sich kein Element unter dem Cursor, wird das Package auf der höchsten Ebene angelegt, befindet sich unter dem Cursor ein Package, so wird das angelegte Package diesem untergeordnet (siehe Abbildung 8.6Abbildung 8.7).

Beziehungen zwischen einzelnen Packages werden mit Packagediagrammen modelliert und dargestellt. Diese Beziehungen umfassen Importbeziehungen, Hierarchiebeziehungen, Subpackage-Beziehungen. Zusätzlich können hier auch die Schnittstellen (Ports) eines Package modelliert und visualisiert werden. Die Visualisierung entspricht der UML-Notation (siehe Abbildung 8.8).

3.1.3 Klassen

Klassen bilden die zentrale Einheit aller Modellierungen von (Software-) Systemen. Sie werden für das Systemdesign eingesetzt und bilden die Basis für die Implementation des Systems. Klassen werden dazu eingesetzt, um die fachlichen und technischen Anforderungen an eine Software zu erfüllen (6 S. 98). ObjectiF unterstützt den vollen Umfang der Klasse im Sinne der UML.

Ähnlich den ObjectiF-Packages bilden Klassen einen eigenen Namensraum. Öffentliche Elemente in den Klassen bilden ihre Schnittstelle zu weiteren Klassen. Abhängig von der Sprache des umschließenden Package können einer Klasse Elemente zugeordnet werden, die der Zielsprache der Klasse entsprechen. Ist die Klasse teil eines technischen Modells, sind diese Elemente abhängig von der Definierten Programmiersprache, ist die Klasse teil eines fachlichen Modells, entsprechen die Elemente der fachlichen Domäne. Von der Zielsprache hängt auch die Auswahl der Stereotypen ab, die einer Klasse zugewiesen werden können. Erlaubt die Programmiersprache generische Klassen (Java, C++ (template)), können diese ebenfalls angelegt werden.

Klassen werden durch das Aufrufen des Kontextmenüs angelegt. Ebenso wie bei Packages hängt die Zuordnung der neuen Klasse in eine Hierarchie von dem Element unterhalb des Cursors ab. Einer Klasse können Elemente (Methoden, Attribute, Konstruktoren, etc.) durch das Aufrufen des Kontextmenüs der entsprechenden Klasse hinzugefügt werden. Bei der Definition von Methoden, etc. stehen automatisch die Typen der Zielsprache zur Verfügung. Zusätzlich können als Typen weitere, für die Klasse sichtbare andere Klassen verwendet werden (siehe Abbildung 8.9, Abbildung 8.10).

Beziehungen zwischen einzelnen Klassen werden innerhalb von Klassendiagrammen modelliert und visualisiert. Klassendiagramme können, wie alle Elemente, in eine Hierarchie eingeordnet werden. Innerhalb des Klassendiagramms ist es möglich, bestehende Klassen einzufügen oder neue Klassen anzulegen.

Klassendiagramme unterstützen vier Arten der Beziehung zwischen den einzelnen Klassen. Diese vier Arten entsprechen denen der UML: Generalisierungen, Assoziationen, Kompositionen und Abhängigkeiten. Die Beziehungstypen unterstützen alle, durch die UML definierten, Eigenschaften. Es können Multiplizitäten, Rollennamen, Navigierbarkeiten, Stereotypen, etc. den Beziehungen hinzugefügt werden.

Neben Klassen und Klassenbeziehungen können dem Klassendiagramm auch Notizen hinzugefügt werden.

Klassendiagramme werden über das Kontextmenü angelegt. Zunächst ist jedes Klassendiagramm leer.

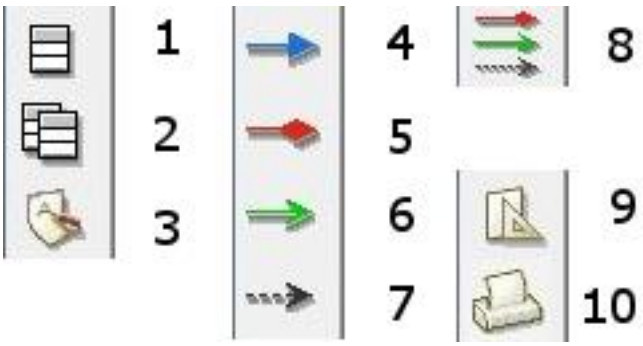

Abbildung 3.3 Klassendiagramm Menü

Abbildung 3.3 zeigt eine Auswahl an Menüpunkten des Klassendiagrammes.

Mit den Menüpunkten 1 und 2 ist es möglich, eine neue Klasse anzulegen, oder eine bereits existierende Klasse dem Diagramm hinzuzufügen. Menüpunkt 3 fügt dem Diagramm eine neue Notiz hinzu.

Beziehungen werden mit den Menüpunkten 4 bis 7 angelegt. Der blaue Pfeil legt Generalisierungen, der rote Pfeil Kompositionen, der grüne Pfeil Assoziationen und der schwarze Pfeil Abhängigkeiten zwischen zwei Klassen an.

Wird verwendet, um bestehende, zurzeit nicht visualisierte, Beziehungen zwischen Klassen sichtbar zu machen. Für eine automatische Anordnung der Beziehungspfeile im Diagramm wird der Menüpunkt 9 verwendet, um das Diagramm zu drucken, Menüpunkt 10.

Beziehungen zwischen zwei Klassen können durch das Kontextmenü wieder gelöscht, oder im Diagramm nur verborgen werden (siehe Abbildung 8.11).

Es ist möglich, alle Klassendefinitionen zuzuklappen, wodurch nur ihr Name angezeigt wird, oder alle aufzuklappen, um auch Methoden und Attribute aller Klassen anzuzeigen. Dies ist auch für individuelle Klassen möglich. Im Kontextmenü des Klassendiagramms kann zudem gewählt werden, ob der einfache Name der Klasse oder der gesamte Name, also inklusive Package-Zugehörigkeit, angezeigt werden soll.

3.2 **MDD mit ObjectiF**

ObjectiF unterstützt für das MDD die bereits angeführte Variante des AC-MDSD. Die Architektur des Systems wird durch ein fachliches Modell definiert, anschließend wird dieses durch Transformatoren in ein oder mehrere technische Modelle umgewandelt. Die Umwandlung von fachlichen zu technischen Modell wird durch die Anpassung der Transformatoren verändert. Codegeneratoren erzeugen anschließend aus dem technischen Modell den modellierten Infrastrukturcode. Markierungen im Quellcode werden eingesetzt, um Bereiche zu markieren, die bei der erneuten Generation von Quellcode nicht überschrieben werden. Innerhalb dieser Bereiche wird die Funktionalität der Anwendung programmiert.

Um die Übersicht des ObjectiF-Systems zu verbessern, bietet ObjectiF neben der System-Sicht noch die Sichten „Fachliches Modell“ und „Technisches Modell“ an. Die „Fachliches Modell“-Sicht blendet alle Teile des Systems aus, die nicht Teil des PIMs sind, die „Technisches Modell“-Sicht entsprechend alle Teile, die nicht zu einem PSM gehören.

Fachliche und technische Modelle sind durch Packages voneinander getrennt. Packages der fachlichen Modelle werden durch den Stereotyp „<<Platform Independent Model>>“ gekennzeichnet, technische durch den Stereotyp „<<Platform Specific Model>>“.

Für das fachliche Modell stellt ObjectiF die BPMN bereit. BPMN umfasst Stereotypen (vgl. MDD: UML-Profile), welche für die fachliche Beschreibung der meisten Systeme ausreichend ist. Diese Stereotypen können durch selbst definierte erweitert werden. Für das technische Modell wird die entsprechende Programmiersprache eingesetzt.

Die Transformatoren von ObjectiF setzen auf den jeweiligen Metamodellen der eingesetzten Sprachen an. Ein Transformator wird durch ein Klassendiagramm beschrieben. Die Klassen in diesen Diagrammen spiegeln die Elemente der Quell- und Zielsprachen wieder. Beispielsweise ist ein „Property“, also ein Attribute einer Klasse in der Sprache C#, durch eine Klasse im Transformator repräsentiert. Die einzelnen Klassen selbst besitzen Stereotypen ihrer Metamodelle. In der Abbildung 3.4 besitzt die Klasse „Class“ den Stereotyp „<<Element>>“, da diese ein Element des Metamodells von BPMN repräsentiert.

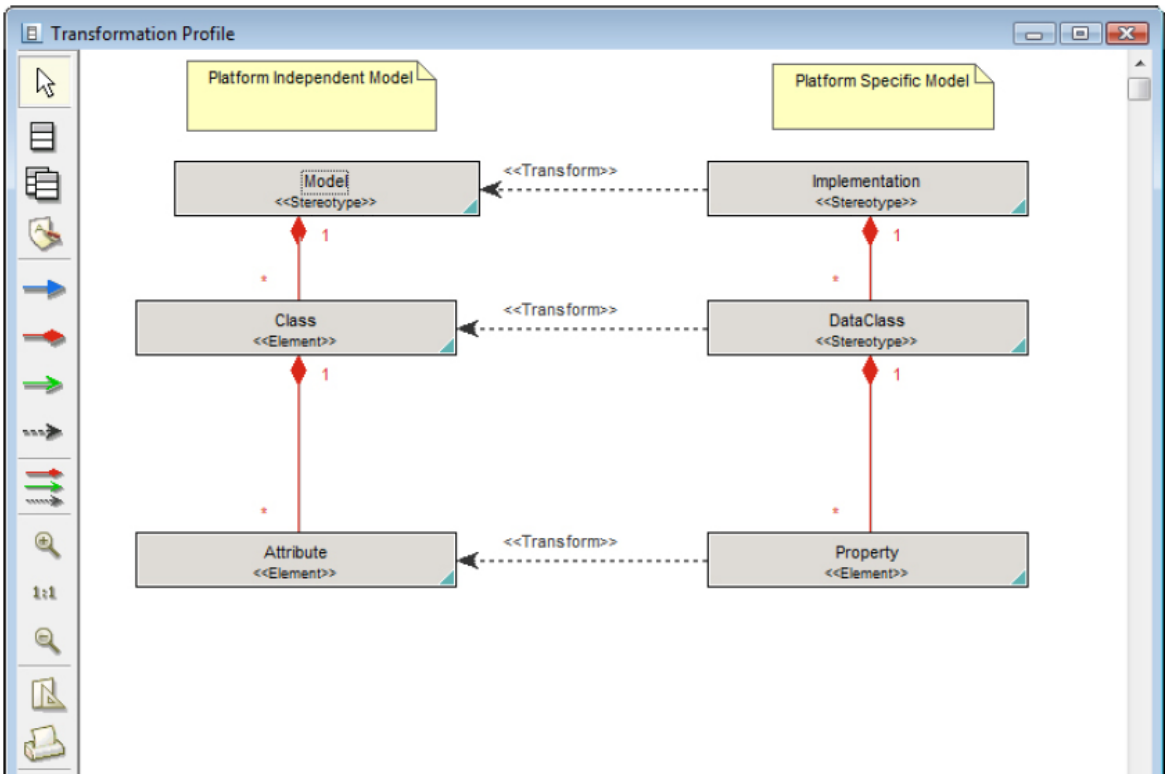

Abbildung 3.4 Transformatoren in ObjectiF (7 S. 7)

Zwischen den Elementen (im Transformator repräsentiert durch eine Klasse) der Quellsprache und denen der Zielsprache besteht eine Abhängigkeit der Art „Transform“. Diese Abhängigkeit sorgt dafür, dass die Elemente aus dem PIM auf diese Elemente des PSM abgebildet werden.

3.3 Round-Trip Engineering mit ObjectiF und Eclipse

ObjectiF bietet die Möglichkeit der Integration mit der Java IDE Eclipse. Dies ermöglicht es, Software Projekte auf Basis von Java mittels Round-Trip-Engineering zu entwickeln.

Für das Round-Trip-Engineering ist es entscheidend, dass Code und Modell konsistent miteinander gehalten werden. Dies Verlangt, dass aus bestehendem Quellcode

Modelle extrahiert und aus Modellen korrekter Quellcode generiert werden kann. ObjectiF deckt beide Anforderungen ab.

Eclipse setzt für die Organisation von Softwareprojekten workspaces ein. Ein workspace kann genutzt werden, um dort zusammenhängende Projekte zu speichern. Der erste Schritt der Integration von Eclipse und einem ObjectiF-System ist es, dem ObjectiF-System einem Eclipse workspace zuzuordnen. Hierfür gibt es zwei Varianten. Standardmäßig legt ObjectiF für ein ObjectiF-System, welches die Programmiersprache Java nutzt, in dem ObjectiF-System-Verzeichnis einen separaten Eclipse workspace an. Ein ObjectiF-System kann entweder mit diesem, oder mit einem anderen bereits bestehendem workspace verbunden werden. Dazu wird im ObjectiF-System der Kontextmenüpunkt „Eclipse Workspace zuordnen."

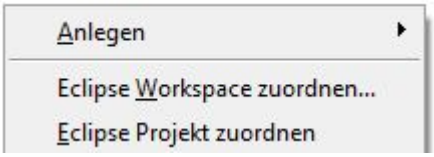

Abbildung 3.5 Eclipse Workspace zuordnen

Anschließend muss dem ObjectiF-System mit dem Kontextmenüpunkt „Eclipse Projekt zuordnen" ein Eclipse Projekt zugeordnet werden. ObjectiF liest automatisch das Eclipse Projekt ein und führt ein anfängliches Reverse-Engineering durch. Hierbei wird aus dem Code in dem Eclipse Projekt Packages und Klassen in ObjectiF angelegt. Dies kann bei großen Eclipse Projekten einige Zeit in Anspruch nehmen. Ein Nachteil hierbei besteht darin, dass ObjectiF keine Fehlermeldungen ausgibt, wenn das automatische Reverse-Engineering fehlschlägt. Daher ist es ratsam, dass Reverse-Engineering manuell durch den Kontextmenüpunkt „aus Eclipse übernehmen" auszulösen.

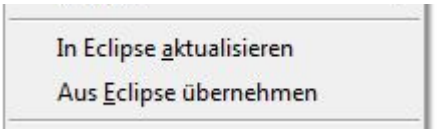

Abbildung 3.6 Round-Trip-Menüpunkte

Mit dem Menüpunkt „Aus Eclipse übernehmen“ wird der Quellcode aus Eclipse zurückgeführt, werden automatisch Packages und Klassen, die in Eclipse nicht mehr existieren, in ObjectiF ebenfalls gelöscht.

Die Übernahme von Änderungen innerhalb von ObjectiF in Eclipse findet im Gegensatz nicht automatisch statt. Der Vorgang der Codegenerierung wird in ObjectiF durch den Kontextmenüpunkt „In Eclipse übernehmen“ ausgelöst. Auch hierbei werden Packages und Klasse, die in ObjectiF nicht mehr existieren in Eclipse entfernt.

Beide Aktionen können entweder für das gesamte System, oder für eine bestimme Klasse, durchgeführt werden. Soll nur eine einzelne Klasse, oder ein Package, hierbei ist der Vorgang der Selbe, muss sich diese unter dem Cursor befinden, bevor das Kontextmenü aufgerufen wird.

Eclipse muss für das Round-Trip-Engineering stets aus ObjectiF heraus gestartet werden. Dies wird durch den Kontextmenüpunkt „Eclipse Workspace öffnen“ ausgelöst. Gegeben falls muss ObjectiF im Administratorrechte besitzen, um Eclipse starten zu können.

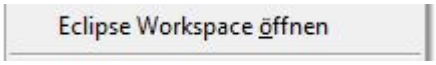

Abbildung 3.7 Eclipse durch ObjectiF starten

4. JAVA-TX

Heutzutage existiert eine große Anzahl an unterschiedlichen Programmiersprachen. Diese Sprachen unterscheiden sich an vielen Punkten voneinander. Ein Unterschied liegt beispielsweise in dem Programmparadigma, welches sie umsetzen. Funktionale Programmiersprachen (z.B. LISP oder Haskell), prozedurale Programmiersprachen (z.B. C) oder objekt-orientierte Programmiersprachen (z.B. C++, Java) bieten verschiedene Ansätze für die Lösung von Problemen. Dabei folgen nicht alle Sprachen streng einem einzigen dieser Paradigma, sondern können Konzepte aller Typen beinhalten (z.B. Python).

Programmiersprachen unterscheiden sich nicht nur an dieser Stelle. Ein weiterer, sehr wichtiger Unterschied zwischen einzelnen Programmiersprachen liegt in ihrem Typsystem. Dynamisch typisierte Programmiersprachen (z.B. Python) überprüfen Variablentypen erst zur Laufzeit und erzeugen gegebenenfalls Laufzeitfehler, wenn Variablen einen inkompatiblen Typen besitzen. Im Gegensatz existieren statisch typisierte Programmiersprachen. Bei diesen muss der Typ einer Variable zur kompilierzeit feststehen. Der Compiler kann vor Ausführung des Programms überprüfen, dass jede Variable den korrekten Typ besitzt, um so Laufzeitfehler zu vermeiden, die durch falsche Typen ausgelöst werden. Auch hier können Sprachen eine gewisse Mischung aus beiden Typisierungsarten umsetzen.
Bei der Programmiersprache Java handelt es sich weitestgehend um eine statisch typisierte Sprache.

4.1 Ziel des Projektes

Im Rahmen des Java-TX Projekt soll ein Java Compiler entstehen, der in der Lage ist, „anhand von Methoden und Eigenschaften eines Objektes die Typen von Variablen [zu] rekonstruieren“ (8). Dies hat zur Folge, dass ein Programmierer von Java-Code nicht mehr zwingend die Typen von Variablen explizit deklarieren muss (8).

4.2 Projekt Aufbau

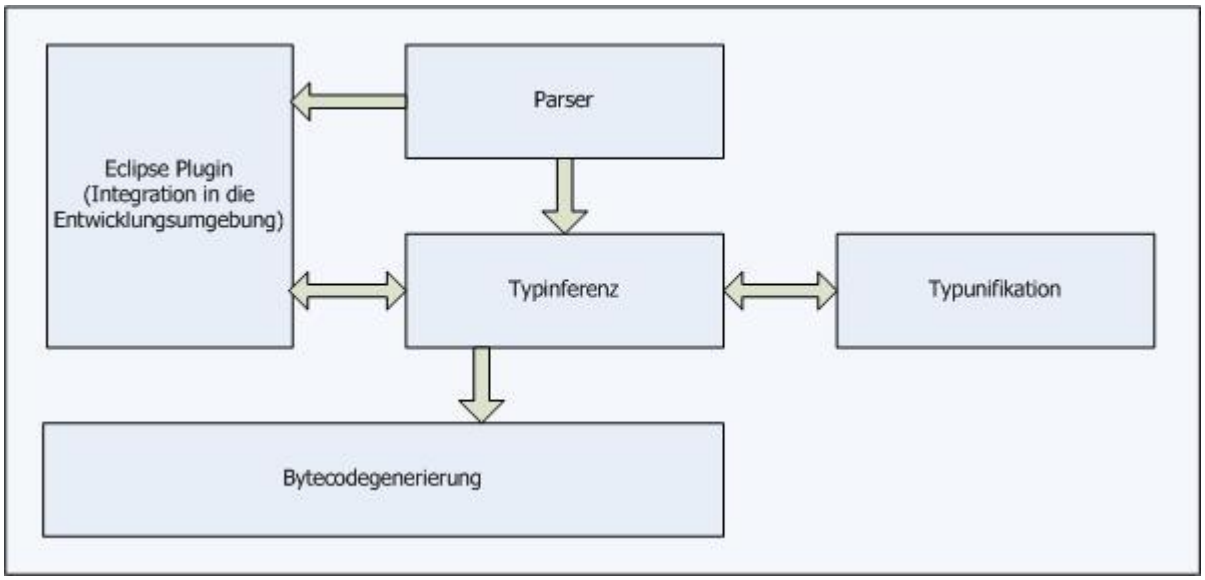

Abbildung 4.1 Schematischer Aufbau des Java-TX Projekts (8)

Der Kern des Java-TX Projekts besteht aus einem Parser, der Typinferenz und –Unifikation und einem Bytecode-Generator. Zusätzlich besitzt das Projekt eine Schnittstelle zu einem Eclipse-Plug-In, was es erlaubt, die Funktionalitäten in Eclipse einzubinden.

4.2.1 Parser

Der Parser hat die Aufgabe, den Quellcode in eine andere Repräsentation zu transformieren. Bei dieser Repräsentation handelt es sich oft um einen Abstract-Syntax-Tree (kurz: AST). Ein AST ist eine Datenstruktur, welche die Struktur eines Programmes effizient darstellen kann. Jeder Knoten des Baumes repräsentiert ein Konstrukt in dem Quellcode. Je Knoten besitzt, entsprechend dem Konstrukt, beliebig viele Unterknoten. Eine Binäroperation beispielsweise besitzt genau zwei Unterknoten. Ein „if-then-else“ Konstrukt besitzt drei Unterknoten: Die Bedingung, der if-Zweig und der else-Zweig.

Die Übersetzung von Quellcode in den AST findet in zwei Phasen statt (9). Zuerst wird mittels einem L*exer* der Quellcode gescannt und in *Tokens* (Schlüsselwörter der Programmiersprache, Zahlen, Variablennamen, etc.) (9). Diese *Tokens* dienen als Input für den Parser (9). Der Parser setzt die generierten *Tokens* zu syntaktischen

Einheiten zusammen (z.B. Variablendeklaration, Wertzuweisungen, etc.), überprüft dabei die syntaktische Korrektheit des Quellcodes und baut den AST auf.

Der entstandene AST dient den anderen Teilen des Projekts aus die Grundlage für ihre Aufgaben.

4.2.2 Typinferenz und –Unifikation

Aufgabe der Typinferenz und –Unifikation ist es, alle nicht explizit deklarierten Typen zu berechnen (10). Mit Hilfe des AST werden zunächst alle bekannten Typen ermittelt. Auf diese Typen werden Inferenz- und Unifikationsregeln angewandt um eine Menge von gültigen Typannahmen zu erhalten (10). Mittels der Unifikation wird Belegung korrekte Typenbelegung berechnet (10). In der Arbeit (11)ist dieser Vorgang im Detail erläutert.

Ein Problem besteht darin, dass die mögliche Belegung in Fällen nicht eindeutig sein muss (8). Daher wurde das Eclipse-Plug-In entwickelt, welches dem Programmierer die Wahl des korrekten Typs erlaubt (siehe Abbildung 4.2, Punkt 12).

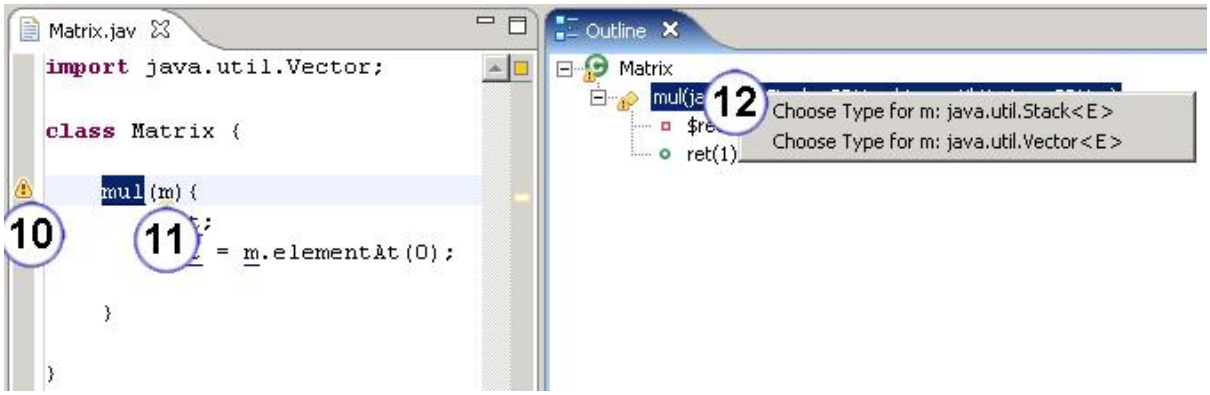

Abbildung 4.2 Typenauswahl im Eclipse Plug-In

4.2.3 Bytecode Generation

Auf Basis der Ergebnisse der Typinferenz kann anschließend der AST in Java Bytecode übersetzt werden. Dieser Vorgang ist im Detail in der Arbeit (10) beschrieben.

5. CASE STUDY

Diese Case Study soll das Round-Trip-Engineering mit ObjectiF und Eclipse verdeutlichen.

In der Case Study soll ein Java Package „datastructures“ entstehen. In diesem Package wird die Java-Klasse MyList und die Klasse MyListMember implementiert. Zusätzlich wird ein Package „datastructures.interfaces“ angelegt, welches das Java-Interface „IMyList“ beinhaltet.

Bei den Klassen handelt es sich um generische Klassen und bei dem Interface um ein generisches Interface mit dem Typparameter T. Das Interface gibt zwei Funktionen vor, um der Liste ein neues Element anzuhängen und um das letzte Element in der Liste zu löschen.

Die Klasse MyList implementiert das Interface IMyList. Die Klasse MyListMember wird verwendet, um innerhalb der Klasse MyList eine doppelt verkettete Liste aufzubauen.

Nachdem die Funktionalität erfüllt wurde, werden die Klassen und das Interface entsprechend umbenannt nach MyStack, MyStackMember und IMyStack.

5.1 ObjectiF-System konfigurieren

Zuerst muss ein neues ObjectiF-System angelegt werden. Als Systemvorlage wird „UML with Java“ ausgewählt.

ObjectiF erstellt für diese Vorlage automatisch zwei Packages: „MyApplication“ und „MyCompany“. Diese werden mit dem Kontextmenü aus dem System gelöscht (siehe Abbildung 8.12, Abbildung 8.13)

Nachdem das ObjectiF-System bereit ist, muss dieses mit einem Eclipse Workspace verbunden werden. Hierzu wird der Kontextmenüpunkt „Eclipse Workspace zuordnen“ verwendet. Für dieses Projekt wird der Workspace genutzt, der von ObjectiF automatisch für das System angelegt wurde (siehe Abbildung 8.14, Abbildung 8.15).

Ist das System mit dem Workspace verbunden, startet ObjectiF automatisch Eclipse.

Der verwendete Workspace enthält immer ein Projekt mit dem Namen „MyProject“. Dieses wird in Eclipse zu „Case Study“ umbenannt.

Nachdem das Projekt umbenannt wurde, muss dem ObjectiF-System dieses Eclipse-Projekt durch den Kontextmenüpunkt „Eclipse Projekt zuordnen“ zugewiesen werden (siehe Abbildung 8.16). Bei Auswahl dieser Option öffnet sich ein Fenster, welche alle verfügbaren Eclipse Projekte innerhalb des verbundenen Workspace anzeigt. Hier kann das Projekt „Case Study“ ausgewählt werden.

Nach Beendigung dieser Schritte ist das ObjectiF-System für das Round-Trip-Engineering konfiguriert.

5.2 **Struktur erstellen**

Nachdem das ObjectiF-System konfiguriert ist, kann die Struktur des Projektes definiert werden. Hierzu werden zwei Packages angelegt: „datastructures“ und „interfaces“. Das Package „interfaces“ ist ein Subpackage von „datastructures“.

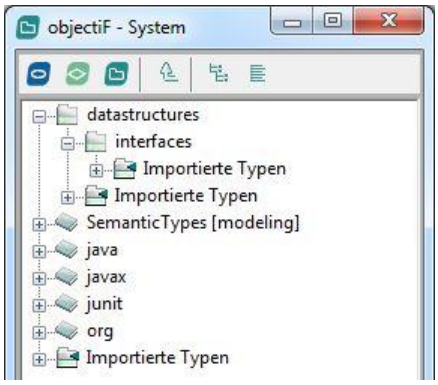

Abbildung 5.1 ObjectiF – Packages

Da die Packages keine Klassen enthalten, werden diese während der Codegenerierung nicht erstellt.
Durch das Aufrufen des Kontextmenüs der entsprechenden Packages kann die generischen Klassen MyList und das generische Interface IMyList erstellt werden. Wichtig an dieser Stelle ist, dass der Punkt „Generische Klasse“ gewählt wird. In ObjectiF gibt es keine Möglichkeit eine Klasse in eine generische Klasse umzuwandeln. Um eine Klasse zu einer generischen Klasse umzuwandeln kann dies innerhalb von Eclipse durchgeführt werden. Werden die Änderungen aus Eclipse in ObjectiF übernommen, werden die nicht generischen Klassen durch die generischen ersetzt.

Um ein Interface in ObjectiF anzulegen wird ebenfalls zuerst eine Klasse angelegt. In den Eigenschaften dieser Klasse muss der Stereotyp von „Class“ zu „Interface“ geändert.

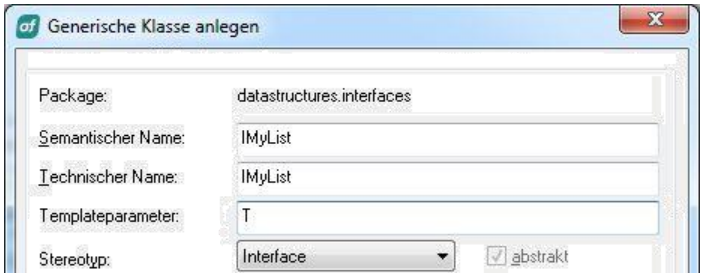

Abbildung 5.2 ObjectiF - Interface anlegen

Nach der Codegenerierung („In Eclipse übernehmen“) erscheint die Klasse und das Interface auch in Eclipse.

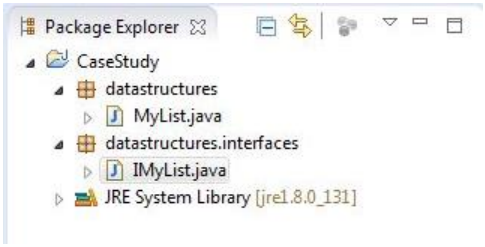

Abbildung 5.3 ObjectiF - Klassen in Eclipse

```
1 public interface IMyList<T > {
2
3 }
```

Listing 5.1 Eclipse - Initiale Implementation IMyList

```
1 public class MyList<T >{
2
3 }
```

Listing 5.2 Eclipse - Initiale Implementation MyList

Die Klasse MyList soll das Interface IMyInterface implementieren. In ObjectiF wird dieser Zusammenhang innerhalb von einem Klassendiagramm modelliert. Hierzu wird eine Generalisierung zwischen dem Interface und der Klasse im Klassendiagramm hergestellt. Durch die erneute Codegenerierung wird in Eclipse diese Beziehung ebenfalls hergestellt.

```
1 public class MyList<T >
2 implements datastructures.interfaces.IMyList<T >{
3
4 }
```

Listing 5.3 Eclipse - MyList implements IMyList

Nun kann das Interface um die Methoden „add“ und „remove“ erweitert werden.

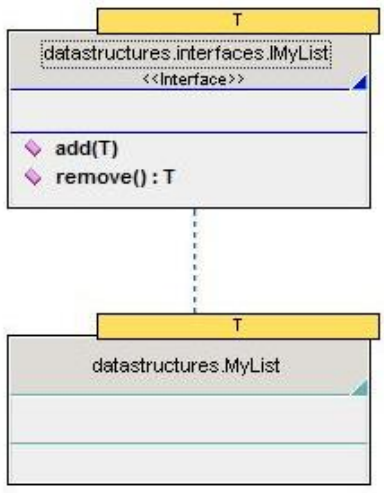

Abbildung 5.4 ObjectiF - Interface in ObjectiF

```
1 public interface IMyStack<T  > {
2
3    public void add(T Member);
4    public T remove();
5
6 }
```

Listing 5.4 Eclipse - IMyList um Methoden erweitert

ObjectiF fügt den Klassen, die ein Interface implementieren, die Methoden dieses Interface nicht automatisch hinzu. Hierfür kann die Funktion von Eclipse verwendet werden. Nachdem innerhalb von Eclipse die Klasse MyList durch die Methoden erweitert wurde, können diese Änderungen wieder in ObjectiF übernommen werden.

```
1 public class MyList<T >
2 implements datastructures.interfaces.IMyList<T >{
3
4     public void add(T Member){}
5
6     public T remove(){}
7 }
```

Listing 5.5 Eclipse - MyList um Methoden erweitert

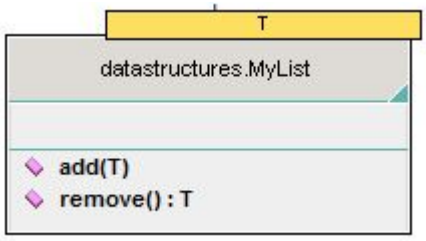

Abbildung 5.5 ObjectiF - MyList um Methoden erweitern

5.3 Implementation

Für die Funktionalität der Klasse MyList wird dieser in Eclipse die Attribute „Tail“, und „length“ hinzugefügt. Zudem wird die Klasse MyListMember in Eclipse erstellt und implementiert. MyListMember besitzt die Attribute „prev“, „next“ und „content“ und entsprechende getter- und setter-Methoden.

```
01 public class MyList<T >
02 implements datastructures.interfaces.IMyList<T >{
03
04     private MyListMember<T > Tail;
05     private int length;
06     public MyList() {
07         this.Tail = null;
08         this.length = 0;
09     }
10
11     public void add(T Member){}
12
13     public T remove(){}
14 }
```

Listing 5.6 Eclipse - MyList um Attribute erweitert

```
01 class MyListMember<T  > {
02
03    protected MyListMember <T > prev;
04    protected MyListMember <T > next;
05    protected T content;
06    public MyListMember (MyListMember <T > prev, MyListMember <T > next,
T content) {
07        this.prev = prev;
08        this.next = next;
09        this.content = content;
10    }
11
12    public void setprev(MyListMember <T > prev) { this.prev = prev; }
13
14    public MyListMember <T > getprev() { return this.prev; }
15
16    public void setnext(MyListMember <T > next) { this.next = next; }
17
18    public MyListMember <T > getnext() { return this.next; }
19
20    public T getcontent() { return this.content; }
21 }
```

Listing 5.7 Eclipse - Implementation MyListMember

Die Änderungen innerhalb von Eclipse werden anschließend wieder in ObjectiF übernommen.

Das nötige Gerüst für die gesamte Implementation der Klasse MyList ist nun fertig gestellt. Innerhalb von Eclipse können nun die Methoden add und remove implementiert werden.

```
01 public void add(T Member) {
02     if(this.length == 0)
03     {
04         MyListMember<T > newMember = new MyListMember<T >(null, null,
Member);
05         this.Tail = newMember;
06     }
07     else
08     {
09         MyListMember<T > newMember = new MyListMember<T >(this.Tail,
null, Member);
10         this.Tail.setnext(newMember);
11         this.Tail = newMember;
12     }
13
14     this.length = this.length + 1;
15 }
```

Listing 5.8 Eclipse - MyList.add Implementation

```
01 public T remove() {
02     if(this.length > 0)
03     {
04         MyStackMember<T > tailPrev = this.Tail.getprev();
05         T tailContent = this.Tail.getcontent();
06 
07         tailPrev.setnext(null);
08         this.Tail = tailPrev;
09 
10         this.length = this.length - 1;
11 
12         return tailContent;
13     }
14     else
15     {
16         return null;
17     }
18 }
```

Listing 5.9 Eclipse - MyList.remove Implementation

Durch die Implementation entsteht zwischen der Klasse MyList und der Klasse MyListMember eine Assoziation. Zudem enthält die Klasse MyListMember eine Assoziation auf sich selbst. ObjectiF erkennt diese Beziehung und fügt diese dem Klassendiagramm hinzu.

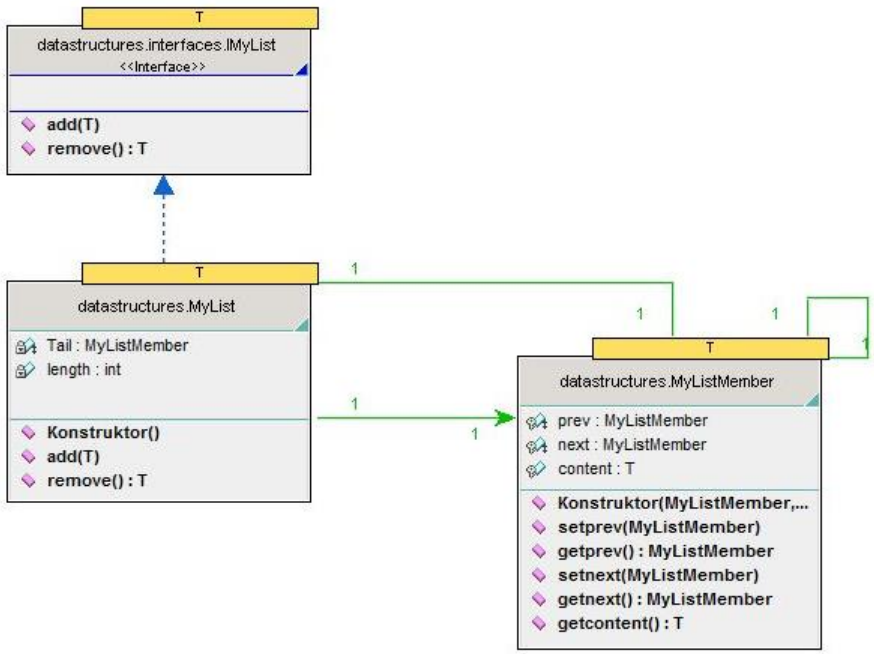

Abbildung 5.6 ObjectiF - Klassenbeziehungen:

5.4 **Refactoring**

Nachdem die Klasse MyList implementiert ist, stellt man fest, dass das Interface IMyList und die Klasse MyList einen Stack und keine Liste implementieren. Durch die Refactoring Möglichkeiten in ObjectiF können nun die Namen der Klassen angepasst werden: IMyStack, MyStack und MyStackMember.

Hierzu werden die Eigenschaften der einzelnen Klassen aufgerufen (per Doppelklick oder Kontextmenü) und der Name der Klasse verändert.

6. Round-Trip-Engineering für Java-TX

6.1 Vorbereitung

Die erfolgreiche Integration von Eclipse in ObjectiF war die größte Hürde dieser Studienarbeit. Das korrekte Vorgehen ist in der Dokumentation von ObjectiF nur sehr oberflächlich beschrieben und für einige Punkte nicht korrekt. Schlägt das Aufrufen von Eclipse aus ObjectiF heraus fehl, würd keine aufschlussreiche Fehlermeldung von ObjectiF ausgeben, sondern lediglich die Meldung „Eclipse konnte nicht gestartet werden“.

ObjectiF ist nur in der Lage, Eclipse-Versionen zu nutzen, welche für x86-Architekturen (32 Bit) ausgelegt sind. Zudem ist die neuste Version von Eclipse (zu diesem Zeitpunkt Eclipse 4.6 „Neon“) nicht kompatible mit ObjectiF.

Zum Zeitpunkt der Installation ist es nötig, bereits eine kompatible Installation von Eclipse installiert zu haben. Während der Installation von ObjectiF kann auf diese Installation verwiesen werden.

Auf einigen Systemen benötigt ObjectiF für das Starten von Eclipse zusätzlich Administrationsrechte. Die Ursache hierfür bleibt ungeklärt.

6.2 Umgebung

Um einen zentralen Punkt bereitzustellen, an dem die UML-Modelle des Java-TX Quellcodes liegen, existiert ein dedizierter Remote Computer im DHBW Campus-Horb Netz. Dieser Computer besitzt eine korrekt konfigurierte ObjectiF Installation mit funktionierender Eclipse Integration.

Da das Java-TX Projekt durch GIT verwaltet wird, existiert ein lokales GIT-Repository welches den Quellcode enthält. Durch GIT kann der Quellcode des Java-TX Projekts flexibel über verteilte Entwickler hinweg auf dem aktuellen Stand gehalten werden.

Die ObjectiF besitzt ein einzelnes ObjectiF-System. Der Quellcode des Java-TX Projekts ist einem Eclipse Workspace zugeordnet, welcher dem entsprechenden ObjectiF-System zugewiesen ist.

6.3 **Klassendiagramme**

Alle erstellten Klassendiagramme sind im ObjectiF-Package *de.dhbwstuttgart* zu finden.

Die Diagramme sind nach den Packages benannt, die sie enthalten. Sofern ein Subpackage nicht aufgeführt ist, sind die Klassen in dem hierarchisch darüber liegenden Diagramm mit enthalten. So sind beispielsweise die Klassen des Package *de.dhbw-stuttgart.syntaxtree.statement.literal* in dem Klassendiagramm *dia_syntaxtree_statement* enthalten.

Der Großteil der Diagramme hat die Beziehungen des Typs *dependency* ausgeblendet, um die Übersichtlichkeit zu wahren.

Daher ist empfohlen, auf die Funktion, alle Beziehungen anzuzeigen, zu verzichten (siehe Abbildung 3.3, Punkt 8). Die Beziehungen einer einzelnen Klasse können im Klassenmenü durch den Kontextmenüpunkt dieser Klasse „Beziehungen darstellen“ hinzugefügt werden.

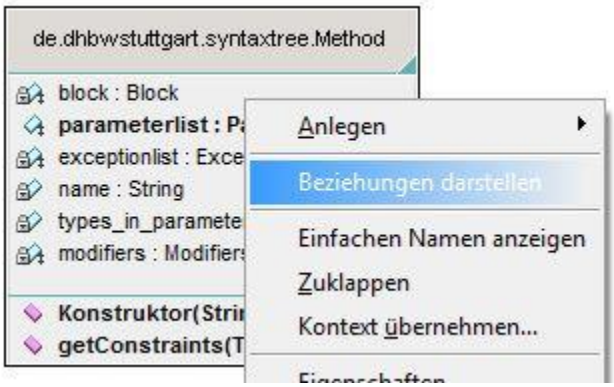

Abbildung 6.1 Beziehung einzelner Klasse darstellen

Da die Beziehungspfeilen beim Verschieben einer Klasse stets neu angeordnet werden, wurde die Anordnung der Pfeile durch die automatische Anordnungsfunktion von ObjectiF durchgeführt (siehe Abbildung 3.3 Punkt 9). Dies bietet den Vorteil, dass die Beziehungen neu in ein Diagramm hinzugefügter Klassen ebenfalls durch diese Funktion einfach angeordnet werden können.

Es kann vorkommen, dass die Größe der UML-Klasse im Diagramm zu klein ist, um hinzugefügte Attribute und Methoden anzuzeigen. ObjectiF vergrößert die

Klassendarstellung nicht automatisch. Um sicherzustellen, dass alle Elemente der Klassen angezeigt werden, können im Klassendiagramm alle Klassensymbole zu- und wieder aufgeklappt werden. Anschließend sind alle Elemente garantiert sichtbar. Hiernach können die Beziehungen wieder entsprechend automatisch angeordnet werden.

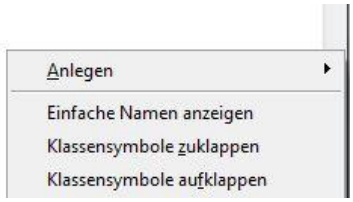

Abbildung 6.2 Klassensymbole auf- und zuklappen

6.4 **Fehlerbehebung**

Ein Schwachpunkt an ObjectiF ist, dass es keine aufschlussreichen Fehlermeldungen ausgibt. Dies ist vor allem für ein Projekt mit großem Umfang, so wie das Java-TX Projekt, ein Hindernis.

Im folgenden Abschnitt sollen Fehler im Quellcode erläutert werden, die während dieser Studienarbeit aufgetreten sind, und Ansatzpunkte, wie diese behoben werden können.

Wichtigster Ansatzpunkt für die Fehlerbehebung sind die Report-Dateien, die ObjectiF bei jedem zurückführen von Code aus Eclipse, anlegt.

Report-Dateien enthalten jede Aktion, die ObjectiF beim Parsen einer Datei, ausführt.

```
--------------------------------------------------------------------------------

Reverse engineering a file [FILE]...

Class 'ConsoleInterface' ('class ConsoleInterface' )  created

Attribute 'directory' ('static final String directory =
system.getProperty("user.dir")' )    created

Reverse engineering of a file [FILE]

completed with 0 error(s).

................................................................................
```

Listing 6.1 Eintrag einer Report-Datei

Ein fehlerfreier Eintrag in einer Report-Datei ist in Listing 6.1 dargestellt. An dieser Stelle wird eine Klasse ‚ConsoleInterface' geparst und in ObjectiF angelegt, die ein Attribute ‚directory' besitzt.

Report-Dateien werden von ObjectiF im „Temp"-Verzeichnis des aktuellen Windows-Benutzers angelegt (*%userprofile%/AppData/Local/Temp/*). Die Report-Dateien sind nach dem Schema repXXXX.tmp benannt. Da ObjectiF veraltete Report-Dateien nicht löscht, muss die aktuelle Report-Datei anhand des Erstellungsdatums gewählt werden.

6.4.1 Nicht fatale Fehler

Bei der Behandlung vieler Fehler reagiert ObjectiF sehr robust. Syntaxfehler im Quellcode verursacht oft eine Fehlermeldung, diese führt jedoch nicht zum Abbruch des Zurückführens.

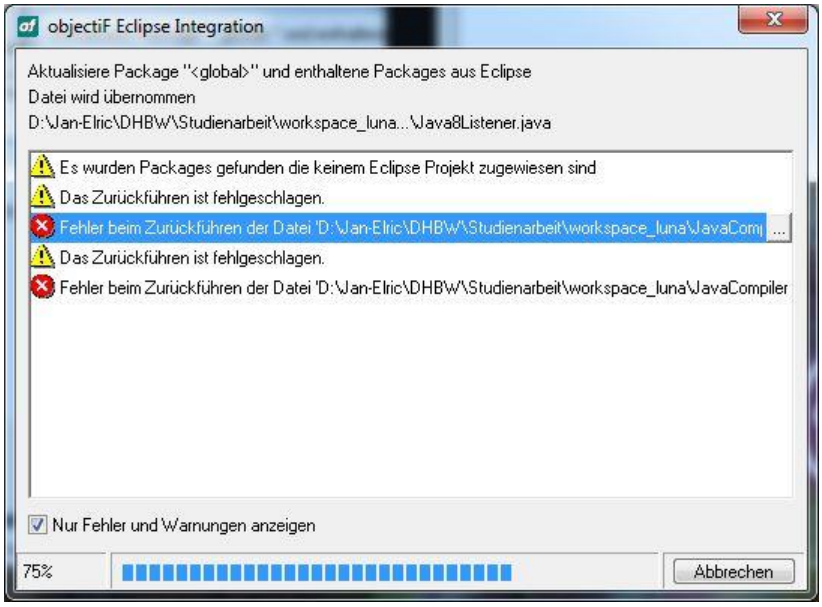

Abbildung 6.3 Nicht fatale Fehler in ObjectiF

Die häufigste Quelle für diese Fehler ist die Instanziierung von generischen Klassen.

```
--------------------------------------------------------------------------------

Reverse engineering a file [FILE]...

Class 'JavaTXCompiler' ('class JavaTXCompiler' )       created

(23):ERROR: unexpected `>`; expected { QUESTIONMARK JAVA_BOOLEAN JAVA_BYTE
CHAR_ DOUBLE_ FLOAT_ INT_ LONG_ SHORT_ VOID_ ID }

Attribute 'sourceFiles' ('List<SourceFile> sourceFiles = new ArrayList<>()'
)       created

Method 'typeInference' ('void typeInference()' )       created

Method 'parse' ('void parse(File sourceFile) throws IOException,
ClassNotFoundException' )       created

Reverse engineering of a file [FILE]

 completed with 1 error(s).

................................................................................
```

Listing 6.2 Fehlermeldung einer Report-Datei eines nicht-fatalen Fehlers

```
private List<SourceFile> sourceFiles = new ArrayList<>();
```

Listing 6.3 nicht-fatale Fehlerursache

Listing 6.2 zeigt den Eintrag in einer Report-Datei, der einen Fehler enthält. Die konkrete Fehlermeldung ist in roter Schrift hervorgehoben. ObjectiF hat beim Parsen einen vermeintlichen Syntaxfehler im Quellcode gefunden. Listing 6.3 zeigt den Code, der diese Fehlermeldung verursacht hat.

Die Fehlerquelle ist die Deklaration eines privaten, generischen Klassenattributes. Java erlaubt es, für die Instanziierung einer generischen Klasse, die Typparameter entfallen zu lassen, wenn diese aus dem Kontext erschließbar sind. ObjectiF erzeugt für dieses valide Konstrukt jedoch einen Syntaxfehler. Würde man an der entsprechenden Stelle den Typen konkret angeben, entfällt die Fehlermeldung.

```
private List<SourceFile> sourceFiles = new ArrayList<SourceFile>();
```

Listing 6.4 Behebung des nicht-fatalen Fehlers

Listing 6.4 zeigt die optionale Änderung am Code, um diese Fehlermeldung zu beheben. Da alle generischen Klassen wie in Listing 6.3 instanziiert werden, ist davon abzusehen, den Code dahingehend zu ändern.

Trotz der Fehlermeldung bricht ObjectiF das parsen der Datei und allen restlichen Dateien nicht ab. Ebenso wird der entsprechenden Klasse das Klassenattribut hinzugefügt.

Die Fehlermeldung „Es wurden Packages gefunden, die keinen Eclipse Projekt zugeordnet sind“ (siehe Abbildung 6.3) kann gänzlich ignoriert werden. Die nicht zugeordneten Packages werden von ObjectiF trotz Fehlermeldung vollständig geparst.

6.4.2 Fatale Fehler

Im Gegensatz zu den nicht fatalen Fehlern verursachen fatale Fehler den Stillstand von ObjectiF und es wird keine Fehlermeldung generiert. Dies kann daran erkannt werden, dass sich ObjectiF unverhältnismäßig lange mit dem Parsen einer einzelnen Datei aufhält. Im Kontext des Java-TX Parsers liegt diese Zeitspanne bei ca. 5 Minuten.

Die Datei, die aktuell von ObjectiF geparst wird, ist in dem Fenster abgebildet in Abbildung 6.3 abzulesen.

Stellt man fest, dass ObjectiF sich im Stillstand befindet, muss das Zurückführen abgebrochen werden. In der Report-Datei findet sich dann der Grund für den fatalen Fehler. Nachdem der Fehler aus dem Quellcode entfernt wurde, muss das Zurückführen erneut gestartet werden.

Ausgelöst wird dieses Verhalten unter anderem durch validen Java-Code, der ein Konstrukt wie Listing 6.5 enthält.

```
01 import examplePackage.* //Enthält die Klasse ExampleClass
02 [...]
03 public ExampleClass Method1(){ [...] }
04 [...]
05 public examplePackage.ExampleClass Method2(){ [...] }
```

Listing 6.5 fatale Fehlerursache

Dieses Konstrukt, in dem ein Typ über verschiedene Import-Pfade genutzt wird, erzeugt eine Fehlermeldung nach Listing 6.6.

```
[…]
Method 'Method1' ('ExampleClass Method1()' )     created
Method ‘Method2’ (‘examplePackage.ExampleClass Method2()’) created
(525):ERROR:` ExampleClass`is ambigous.
[…]
```

Listing 6.6 Fehlermeldung einer Report-Datei eines fatalen Fehlers

```
01 import examplePackage.* //Enthält die Klasse ExampleClass
02 [...]
03 public ExampleClass Method1(){ [...] }
04 [...]
05 public ExampleClass Method2(){ [...] }
```

Listing 6.7 Behebung des fatalen Fehlers

Durch die Codeänderungen in Listing 6.7 wird dieser fatale Fehler behoben.

Aktuell ist der Java-TX Quellcode frei von fatalen Fehlern.

7. AUSBLICK

Diese Arbeit hat die Grundlagen für die Anwendung des Round-Trip-Engineerings für das Java-TX Projekt gelegt.

Es besteht eine korrekt konfiguriertes ObjectiF-System, welches den Java-TX Code erfolgreich einlesen und visualisieren kann. Die Codegenerierung aus ObjectiF und das Zurückführen aus Eclipse heraus kann angewendet werden.

Der Code wurde in sinnvollen Gruppen durch Klassendiagramme dargestellt welche mit Hilfe von ObjectiF weitestgehend automatisch aktualisiert werden können.

Darüber hinaus ist es mit ObjectiF möglich, dass Round-Trip-Engineering für das Java-TX Projekt anzuwenden.

Für fortführende Studienarbeiten ist eine solide Basis gebildet wurden. Eine Möglichkeit, diese Arbeit fortzusetzen, bestünde in der Untersuchung, inwieweit das Vorgestellte Paradigma des Model-Driven-Development für das Projekt vom Vorteil ist, und wie es für dieses Projekt konkret mit ObjectiF umsetzbar ist.

Eine Funktionalität von ObjectiF, die in dieser Arbeit nicht betrachtet wurde, ist die Möglichkeit, Algorithmen durch Sequenzdiagramme zu modellieren. In wie weit ObjectiF hierfür einsetzbar ist, und ob der Aufwand, bestehende Algorithmen durch Sequenzdiagramme zu modellieren, lohnenswert ist, könnte ebenso ein Gegenstand einer fortführenden Arbeit sein.

8. ANHANG

8.1 Software

Auf dem Remote Computer ist folgende Software, die für diese Studienarbeit relevant ist, installiert:

- ObjectiF Version 7.2.24 (18.06.2015)
- Eclipse „Luna“ Version 4.4.2, 32 Bit
- Java 8 Version 1.8.0_131, 32 Bit
- Git Version 2.12.2.windows.2

8.2 Verzeichnisstruktur

Alle relevanten Dateien dieser Studienarbeit liegen auf dem Remote Computer im Verzeichnis *C:\Users\jcc\Desktop\jccObjectIF*.

Dieses Verzeichnis enthält folgende Unterverzeichnisse:

- *_installer* (Installationsdateien der aufgeführten Software)
- *eclipse* (Eclipse-Installation für ObjectiF)
- ObjectIF_*Systems* (Alle ObjectiF-Systeme)

Das Verzeichnis *C:\Users\jcc\Desktop\jccObjectIF\ObjectIF_Systems* enthält das ObjectiF-System *JavaCompilerCore*.

Der Java-TX Code (und das GIT-Repository) liegen in dem Verzeichnis *C:\Users\jcc\Desktop\jccObjectIF\ObjectIF_Systems\JavaCompilerCore\workspace\JavaCompilerCore*

8.3 Zusätzliche Abbildungen

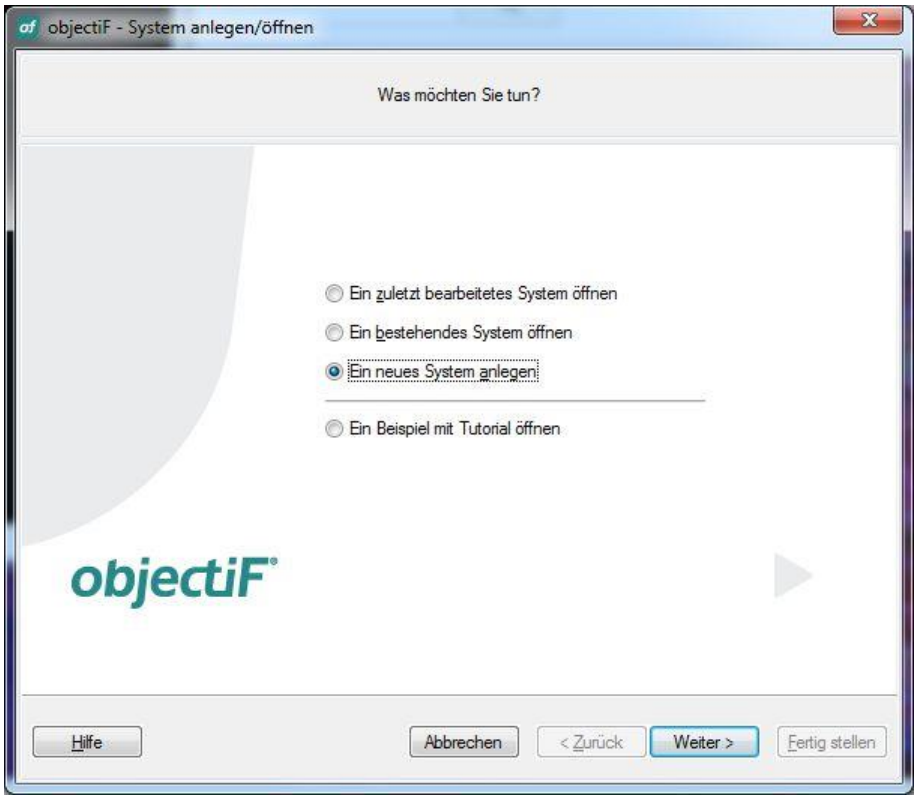

Abbildung 8.1 Neues System anlegen

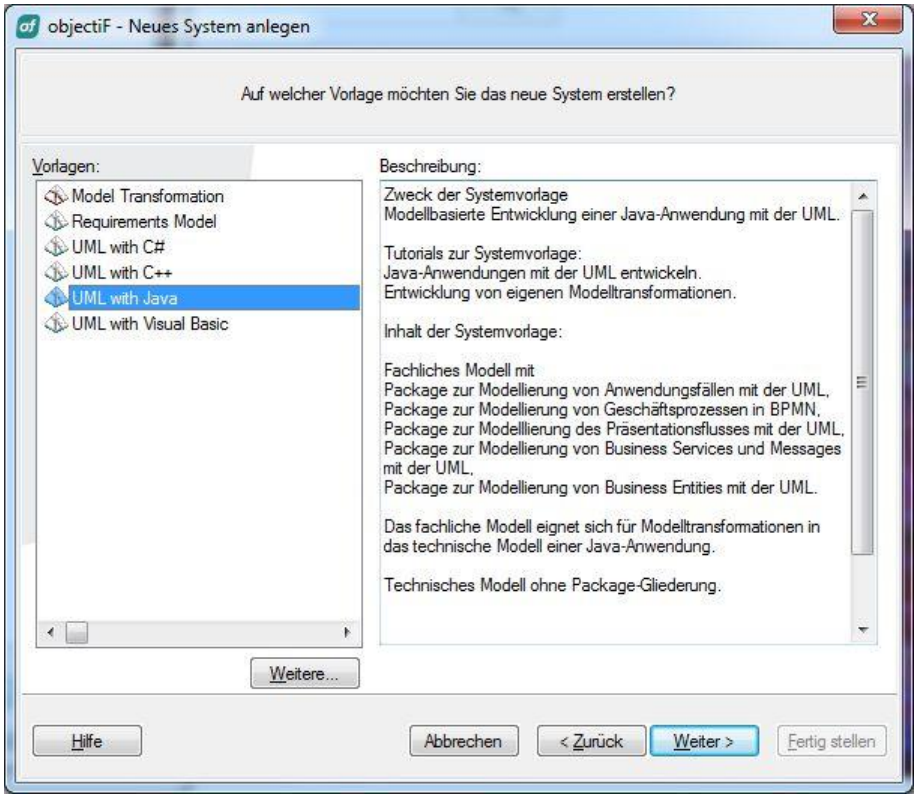

Abbildung 8.2 System-Vorlage wählen

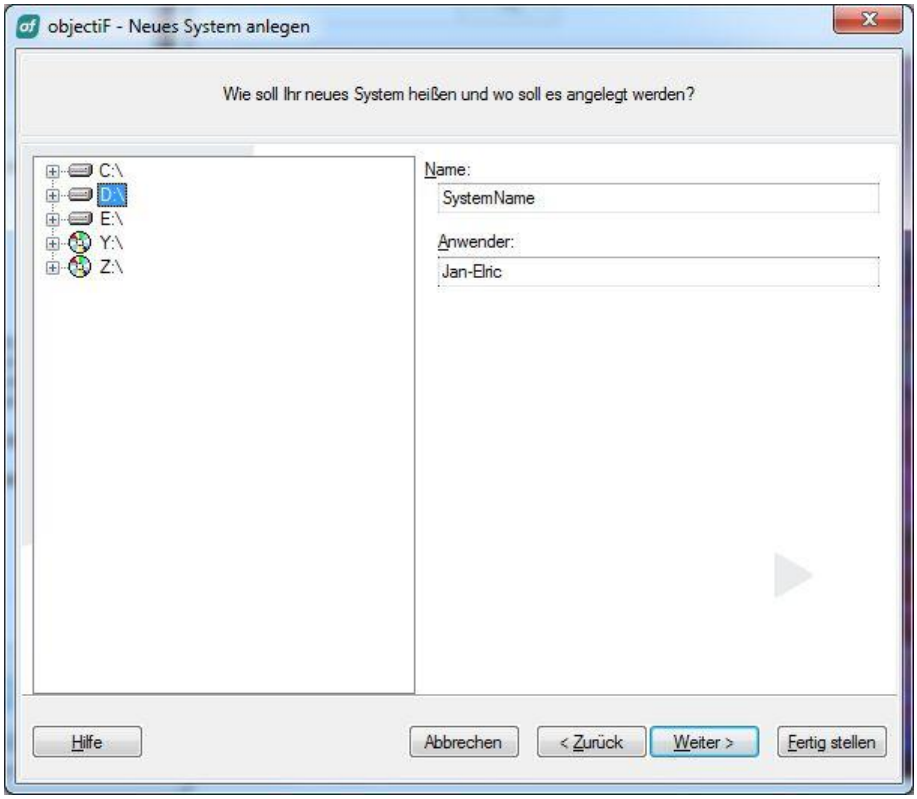

Abbildung 8.3 Systemname und Speicherort setzen

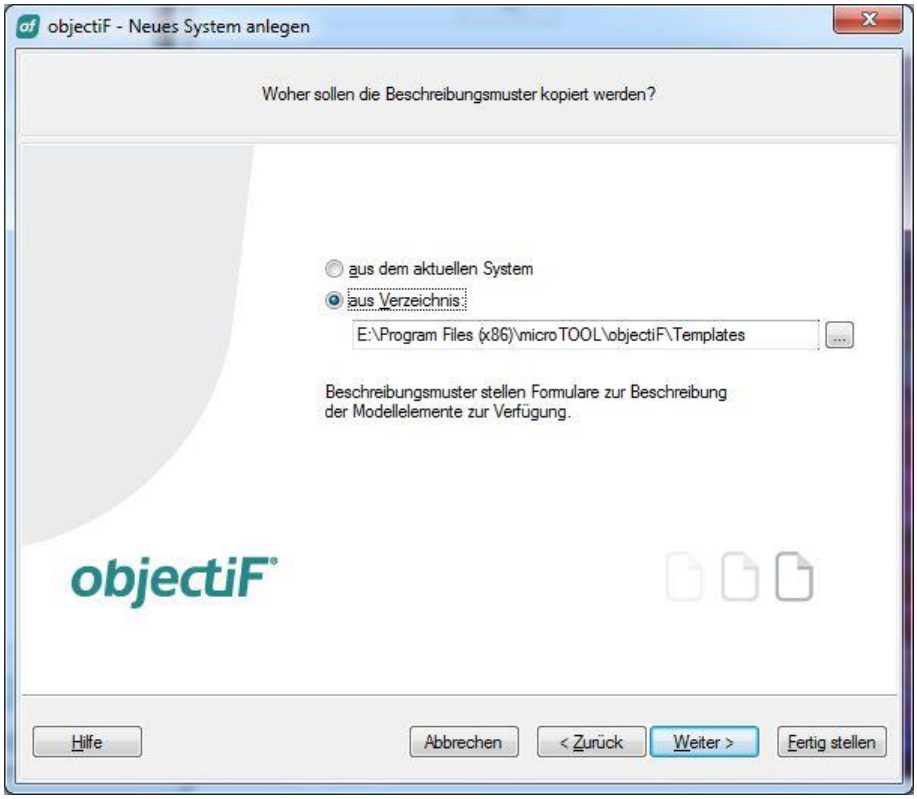

Abbildung 8.4 Systemtemplates wählen

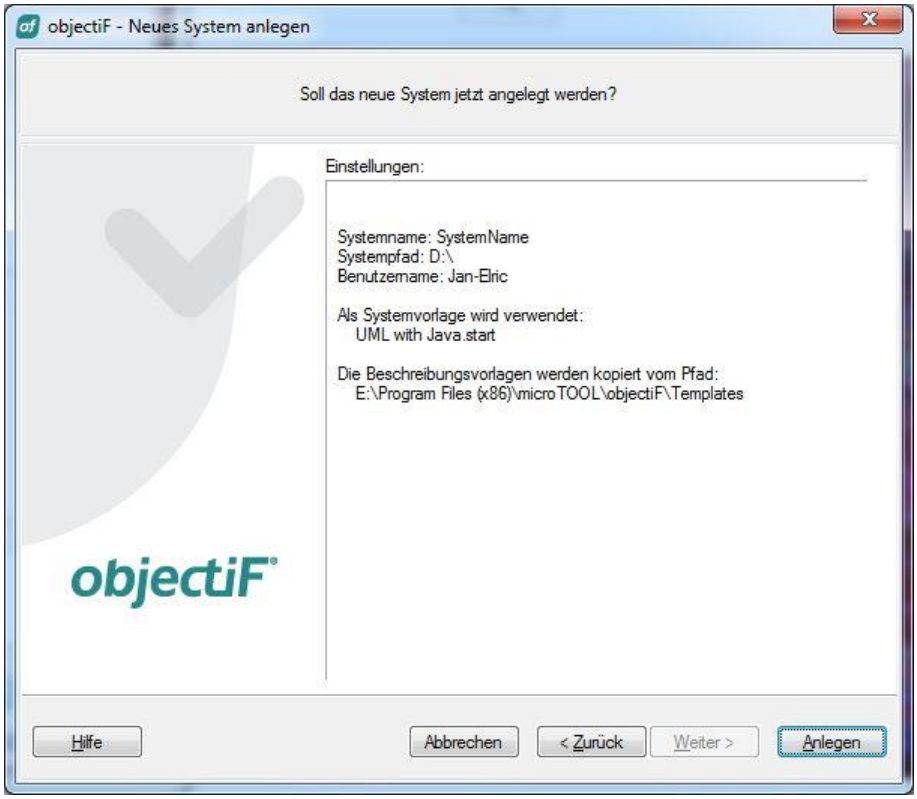

Abbildung 8.5 System Angaben prüfen

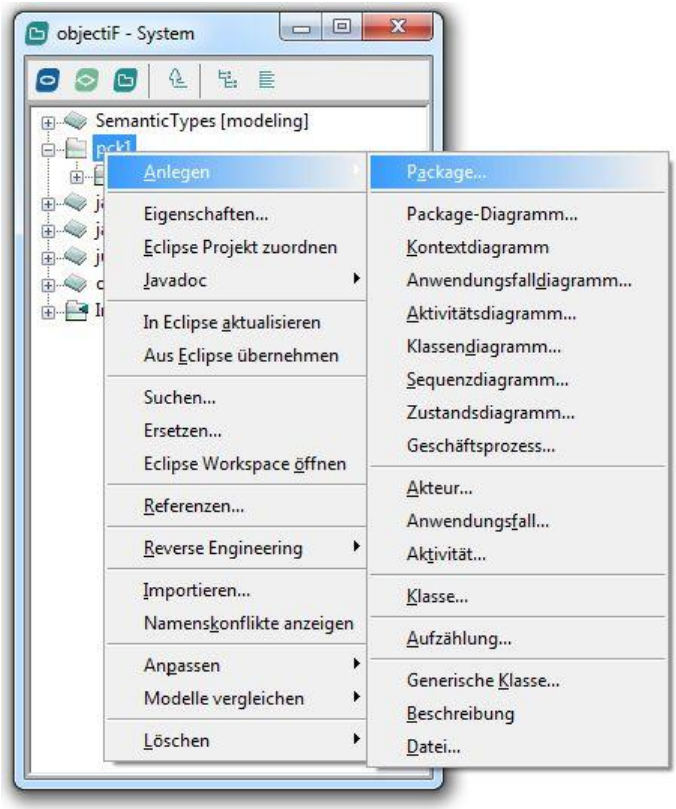

Abbildung 8.6 Subpackage anlegen

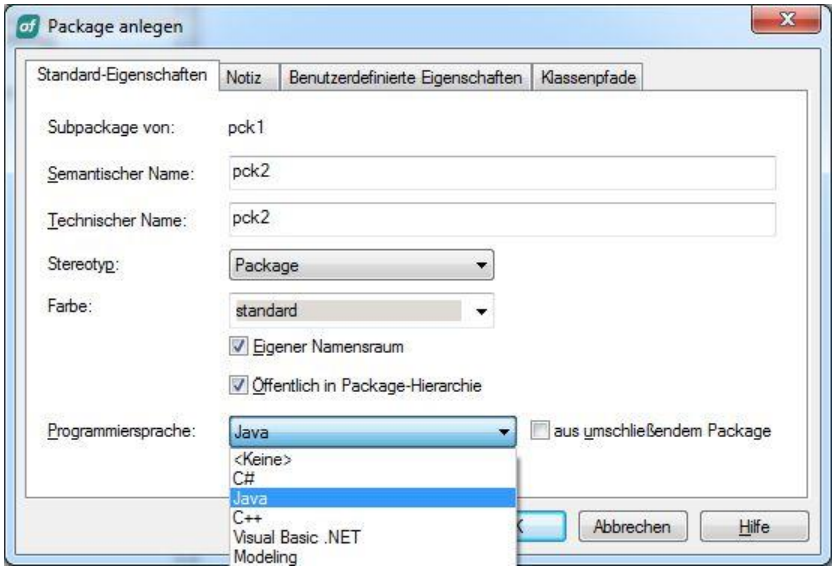

Abbildung 8.7 Package Eigenschaften setzen

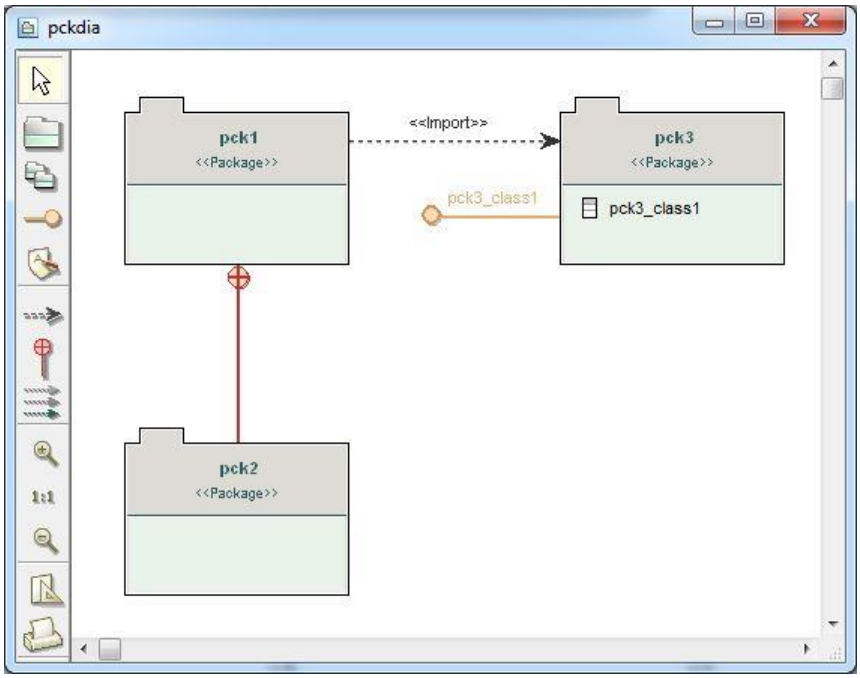

Abbildung 8.8 Package-Diagramm

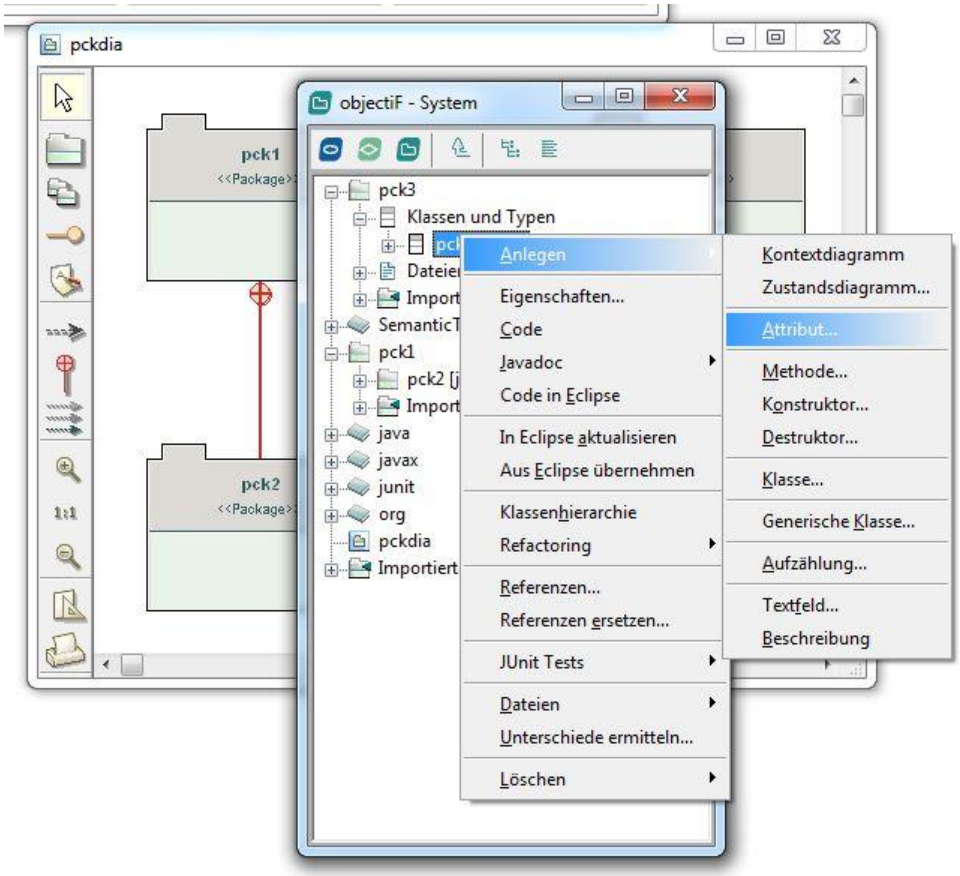

Abbildung 8.9 Klasse anlegen

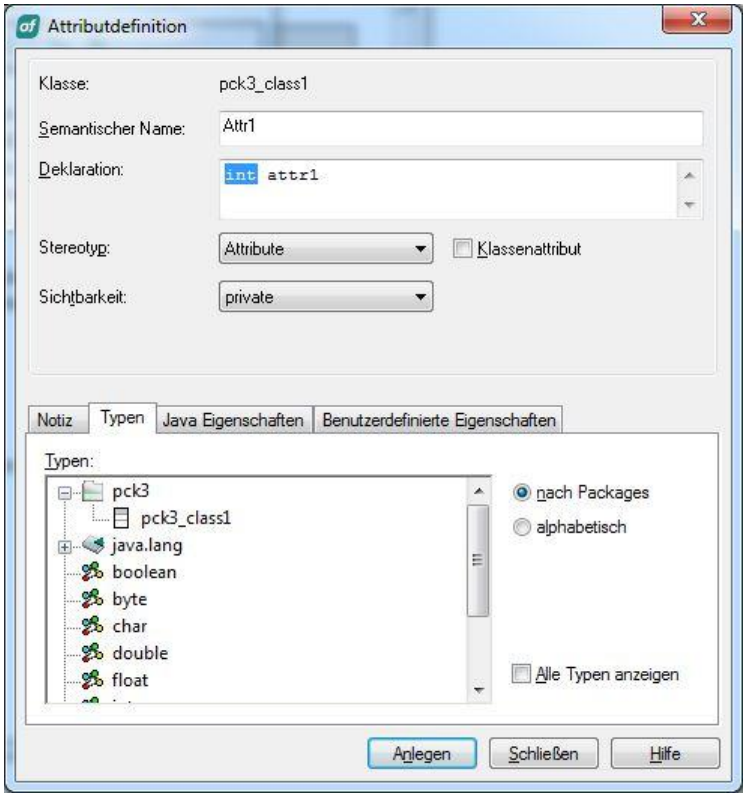

Abbildung 8.10 Attribute hinzufügen

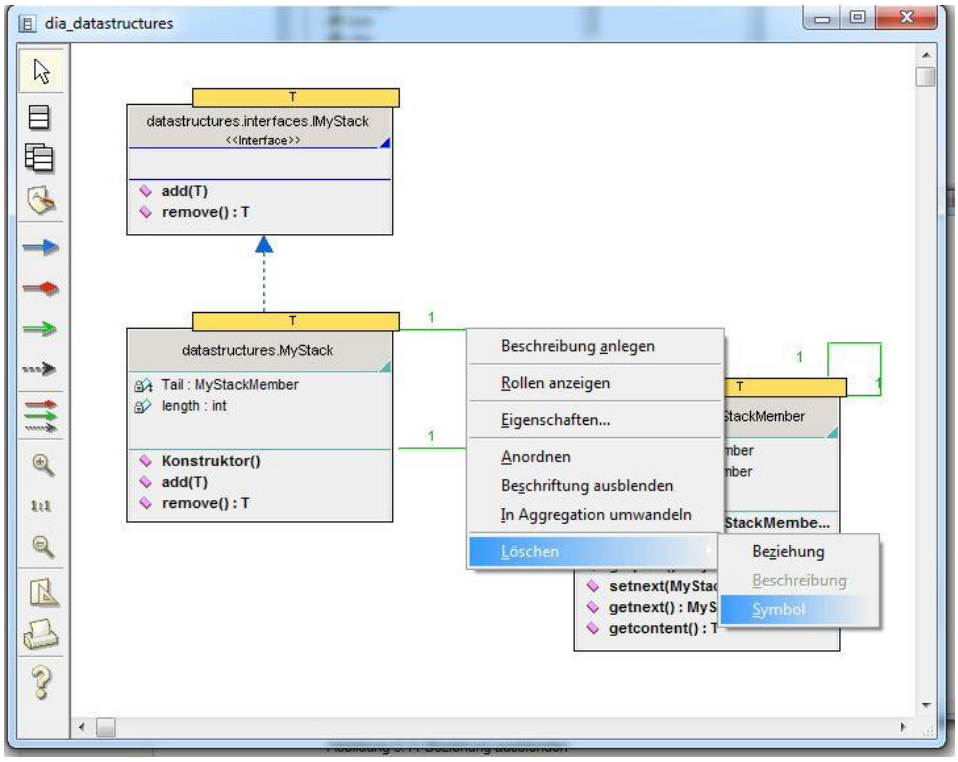

Abbildung 8.11 Beziehung ausblenden

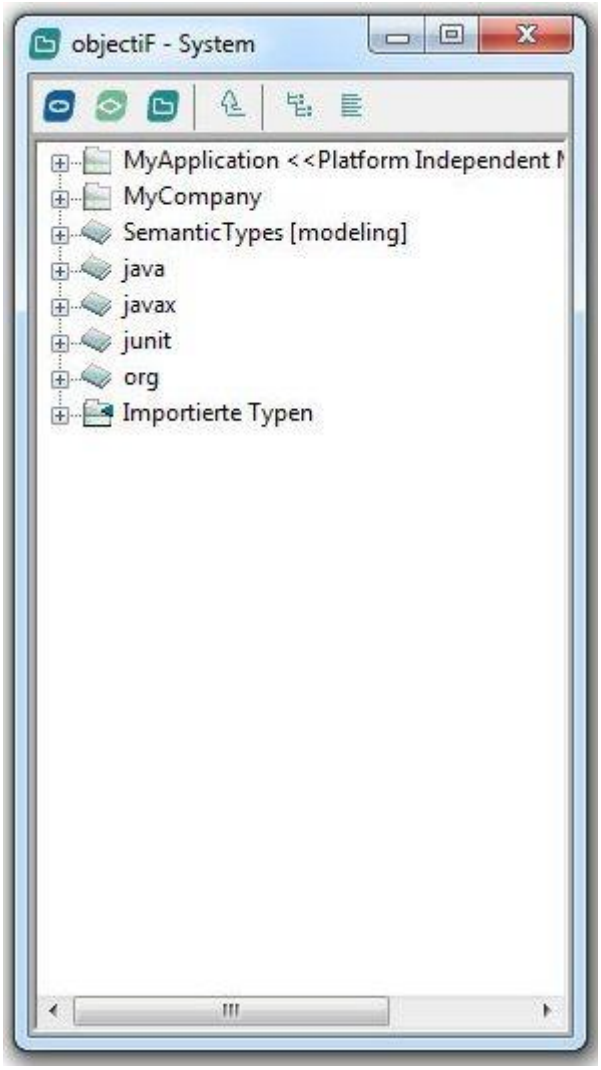

Abbildung 8.12 ObjectiF-System Default Packages

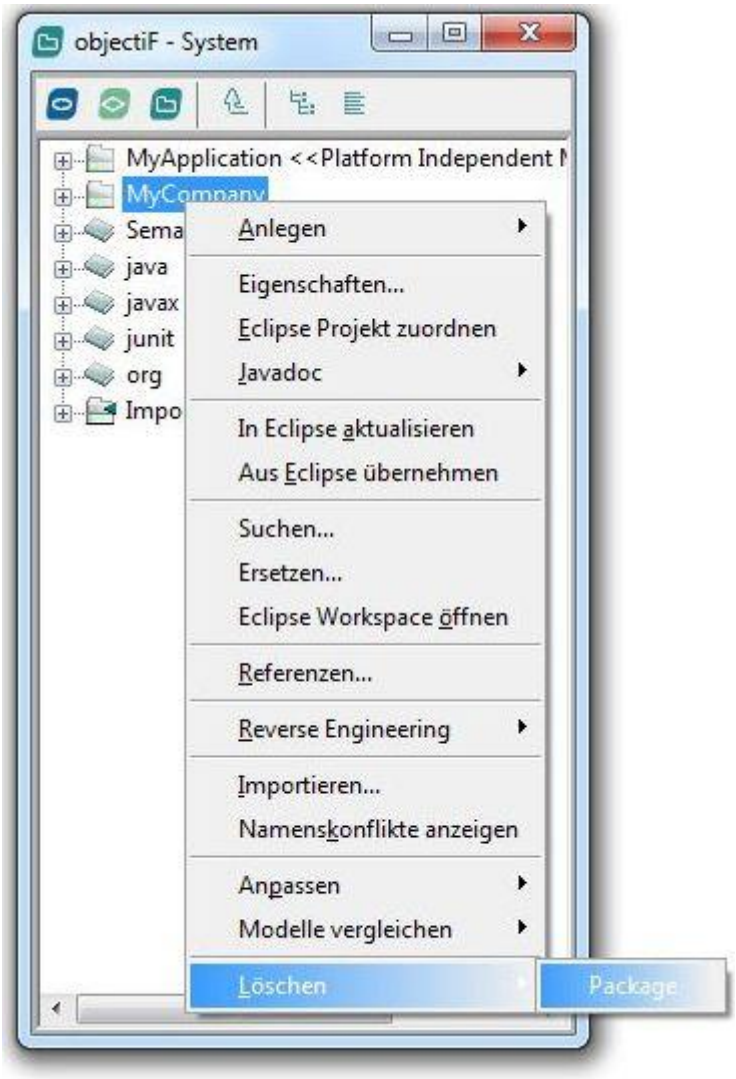

Abbildung 8.13 Package löschen

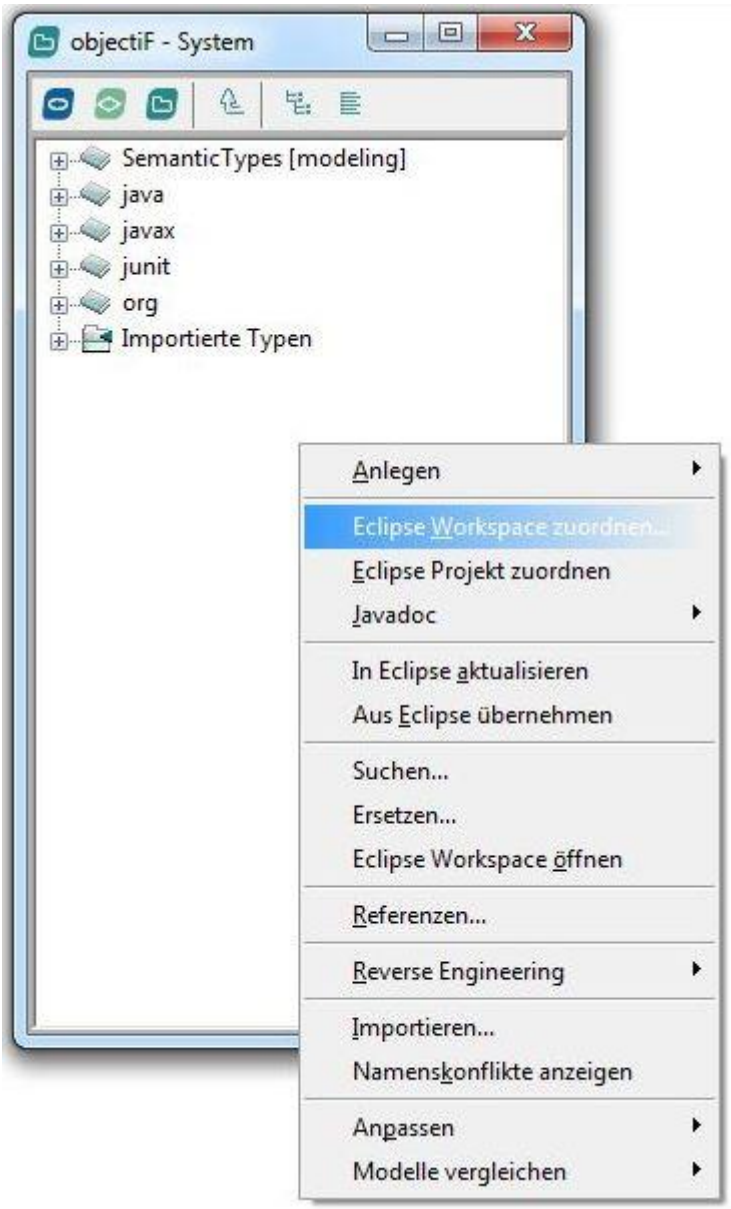

Abbildung 8.14 ObjectiF-Workspace zuordnen

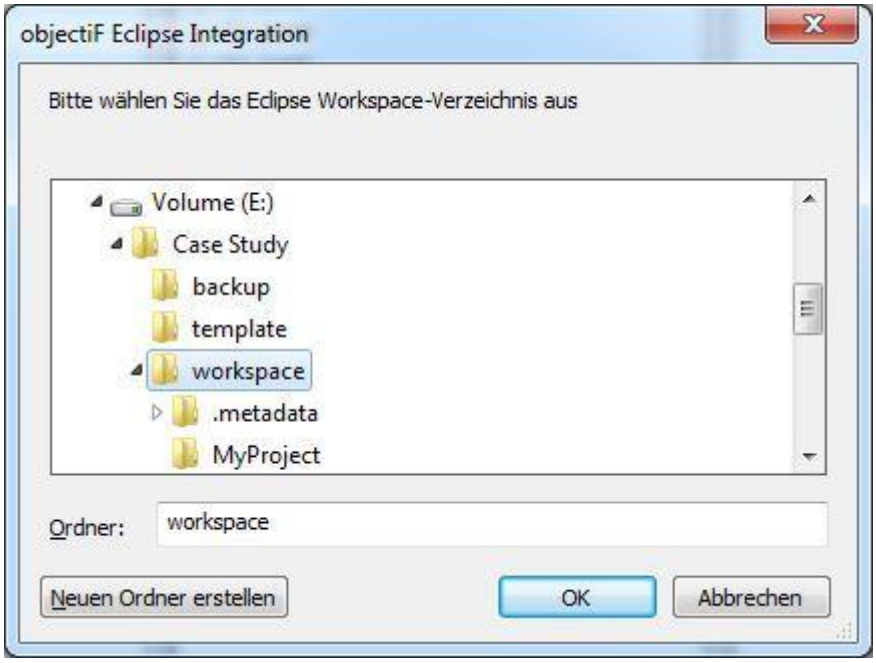

Abbildung 8.15 Workspace auswählen

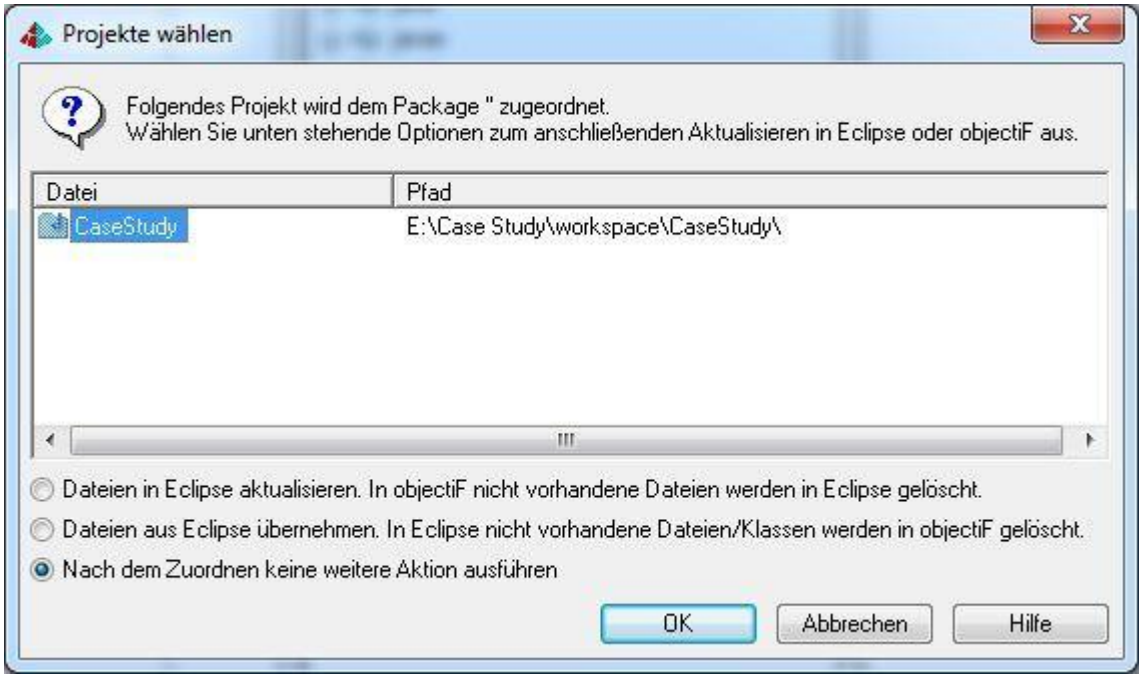

Abbildung 8.16 Eclipse Projekt auswählen

9. LITERATURVERZEICHNIS

1. **Beydeda, Sami, Book, Matthias und Gruhn, Volker.** *Model-Driven Software Development.* s.l. : Springer Berlin Heidelberg, 2005. 978-3-540-28554-0.

2. **Generative Software Engineering Zwickau.** Short overview of MDA and MDSD. [Online] [Zitat vom: 20. 05 2017.] http://documentation.genesez.org/en/ch01s01.html#de.genesez.intro.mdsd.

3. **Stahl, Thomas und Völter, Markus.** *Model-Driven Software Development.* s.l. : John Wiley & Sons, Ltd, 2006. 978-0-470-02570-3.

4. **Object Modelling Group.** Business Process Model and Notation (BPMN). *www.omg.org.* [Online] 2011. [Zitat vom: 28. 05 2017.] http://www.omg.org/spec/BPMN/2.0/PDF/.

5. **Fraunhofer IESE.** http://www.software-kompetenz.de/. [Online] VSEK Projektbüro. [Zitat vom: 20. 05 2017.] http://www.software-kompetenz.de/.

6. **microTOOLS GmbH.** *ObjectiF Anwenderhandbuch.* [PDF] 2015.

7. **microTOOL GmbH.** *ObjectiF - Eigene Modelltransformationen entwickeln.* [PDF] 2015.

8. **Plümicke, Martin.** Das Java-TX Projekt . [Online] [Zitat vom: 28. 05 2017.] http://www.hb.dhbw-stuttgart.de/~pl/JCC.html.

9. **Aaby, Anthony A.** *Compiler Construction using Flex and Bison.* 2004.

10. **Schrödter, Enrico.** *Bytecode Generierung von überladenen Methoden.* Stuttgart-Horb : s.n., 2016.

11. **Steurer, Florian.** *Implementierung eines Typunifikationsalgorithmus für Java 8.* Stuttgart-Horb : s.n., 2015.