



**UNIVERSITÄT PADERBORN**  
*Die Universität der Informationsgesellschaft*

Institut für Informatik  
Warburger Straße 100  
33098 Paderborn

# **Reverse Engineering von im Bytecode vorliegenden Java-Bibliotheken**

**Studienarbeit**

zur Erlangen des Grades

**Bachelor of Computer Science**

für den integrierten Studiengang Informatik

von

Markus Knoop  
Reinherstrasse 10  
33100 Paderborn

Eingereicht bei

Prof. Dr. Wilhelm Schäfer

und

Prof. Dr. Wilfried Hauenschild



## Erklärung

Ich versichere hiermit, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzen anderer als der angegebenen Quellen angefertigt habe und das die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

---

Ort, Datum

---

Unterschrift

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung .....</b>	<b>1</b>
1.1	Reverse-Engineering .....	1
1.2	FUJABA .....	4
1.3	Problem .....	5
1.4	Zielsetzung .....	6
1.5	Struktur der Studienarbeit .....	9
<b>2</b>	<b>Reverse Engineering mittels vergleichbarer CASE-Tools .....</b>	<b>11</b>
2.1	Together .....	11
2.2	EclipseUML .....	12
2.3	Poseidon for UML.....	13
<b>3</b>	<b>Konzept zur Analyse des Java Byte Codes .....</b>	<b>15</b>
3.1	Erstellen eines Klassendiagramms .....	15
3.1.1	Direkte Untersuchung der kompilierten Dateien.....	15
3.1.2	Informationsgewinnung mittels der Java-Reflection-API.....	16
3.2	Assoziationserkennung.....	17
3.2.1	Erstellen eines Musterkataloges .....	17
3.3	Mustersuche innerhalb des Klassendiagramms.....	19
<b>4</b>	<b>Technische Realisierung .....</b>	<b>21</b>
4.1	Laden der Klassen mittels des UPBClassLoaders.....	21
4.2	Erstellen des Klassendiagramms .....	23
4.2.1	Analyse der Class-Objekte .....	24
4.2.2	Analyse der Attribute .....	25
4.2.3	Analyse der Methoden .....	25
4.3	Muster für die Assoziationserkennung.....	26
4.3.1	Konventionen für den Standardmusterkatalog.....	27
4.3.2	Konventionen für benutzerdefinierte Muster .....	28
4.4	Erkannte Muster durch Assoziationen ersetzen .....	28
<b>5</b>	<b>Beispiel einer Java Byte Code Analyse.....</b>	<b>33</b>
5.1	Auswahl der zu analysierenden Dateien .....	33
5.2	Ergebnis der Analyse .....	35
5.3	Einstellungsmöglichkeiten des JBCAnalyser.....	36
<b>6</b>	<b>Zusammenfassung und Ausblick .....</b>	<b>41</b>
	<b>Literaturverzeichnis .....</b>	<b>43</b>
	<b>Anhang.....</b>	<b>44</b>
	Standardmusterkatalog des JBCAnalyser: .....	44

# Abbildungsverzeichnis

Abbildung 01) Einteilung der Softwareentwicklungsschritte [ChCr90].....	1
Abbildung 02) Kostenverteilung im Software-Lebenszyklus [ZSG79].....	2
Abbildung 03) Aktivitäten in der Software-Wartung [Fra92] .....	3
Abbildung 04) Beispiel eines Problems der Assoziationserkennung.....	6
Abbildung 05) Ablauf der Objekterstellung aus Klassendateien .....	7
Abbildung 06) Erkanntes Muster durch Annotation gekennzeichnet .....	8
Abbildung 07) Assoziation aus Annotation erstellt .....	8
Abbildung 08) Auszug aus einer kompilierten Klassendatei .....	16
Abbildung 09) Beispiel einer Musterregel .....	17
Abbildung 10) Überführen von UMLAnnotationen in Assoziationen innerhalb des ASG .....	19
Abbildung 11) Für die Analyse von Class-Objekten zuständige Dateien.....	23
Abbildung 12) Beispiel einer Musterregel .....	26
Abbildung 13) Assoziationen definiert durch zwei Referenzen .....	30
Abbildung 14) Beispiel des Auswahldialoges des JBCAnalyser.....	34
Abbildung 15) Ergebnisklassendiagramm des Beispielles .....	35
Abbildung 16) Einstellungsmöglichkeiten zum Deklarieren der primitiven Klassen.....	36
Abbildung 17) Einstellungsmöglichkeiten zum Deklarieren der Container Klassen .....	37
Abbildung 18) Einstellungsmöglichkeiten zum Deklarieren von SET/GET-Methoden .....	38
Abbildung 19) Einstellungsmöglichkeiten für die Anzeige, Classpath und Musterkatalog ....	40
Abbildung 20) Muster für Referenz mit Kardinalität "1" .....	44
Abbildung 21) Muster für Referenz mit Kardinalität "n" durch Array .....	45
Abbildung 22) Muster für Referenz mit Kardinalität "n" durch Containerklasse.....	45
Abbildung 23) Muster für die Erkennung der Set-Methoden .....	46
Abbildung 24) Muster für Assoziation mit Kardinalität "1" zu "1" .....	47
Abbildung 25) Muster für Assoziation mit Kardinalität "1" zu "n" .....	47
Abbildung 26) Muster für Assoziation mit Kardinalität "1" zu "n" durch Array .....	48
Abbildung 27) Muster für Assoziation mit Kardinalität "n" zu "n" .....	48
Abbildung 28) Muster für Assoziation mit Kardinalität "n" zu "n" durch Array .....	49



# Kapitel 1

## 1 Einleitung

### 1.1 Reverse-Engineering

Unter dem Begriff „Softwareentwicklung“ wird häufig nur der Weg von einem Konzept bis hin zu einer fertigen Implementierung verstanden. Hierbei wird etwas Neues geschaffen, was vorher so noch nicht existiert hat. Doch dieses Forward Engineering ist nur eine Technik, die innerhalb des Lebenszyklus einer Applikation eingesetzt wird.

Mit dem Reverse Engineering bezeichnet man die Analyse eines fertigen Produktes auf Basis des Programmcodes. Es wird hierbei versucht, möglichst viele Informationen über das Produkt zu erhalten und diese in einer abstrakten, leicht verständlichen Form darzustellen. Das Ergebnis gibt einen Einblick in die Struktur der Software und die Beziehung der einzelnen Komponenten zueinander. Es hilft beim „Verstehen einer Software“.

Beim Re-Engineering wird ein fertiges Produkt überarbeitet, um es in seinen Funktionalitäten zu erweitern oder noch vorhandene Fehler zu beheben. Die Entwickler müssen auch hier das Programm zunächst erstmal verstehen, um die Änderungen vornehmen zu können. Anschließend kann mit der Überarbeitung begonnen werden. Daher ist das Re-Engineering eine Mischung aus dem Reverse- und Forward Engineering.

Das folgende Diagramm verdeutlicht die Abläufe der einzelnen Vorgehensarten innerhalb der Entwicklungsschritte.

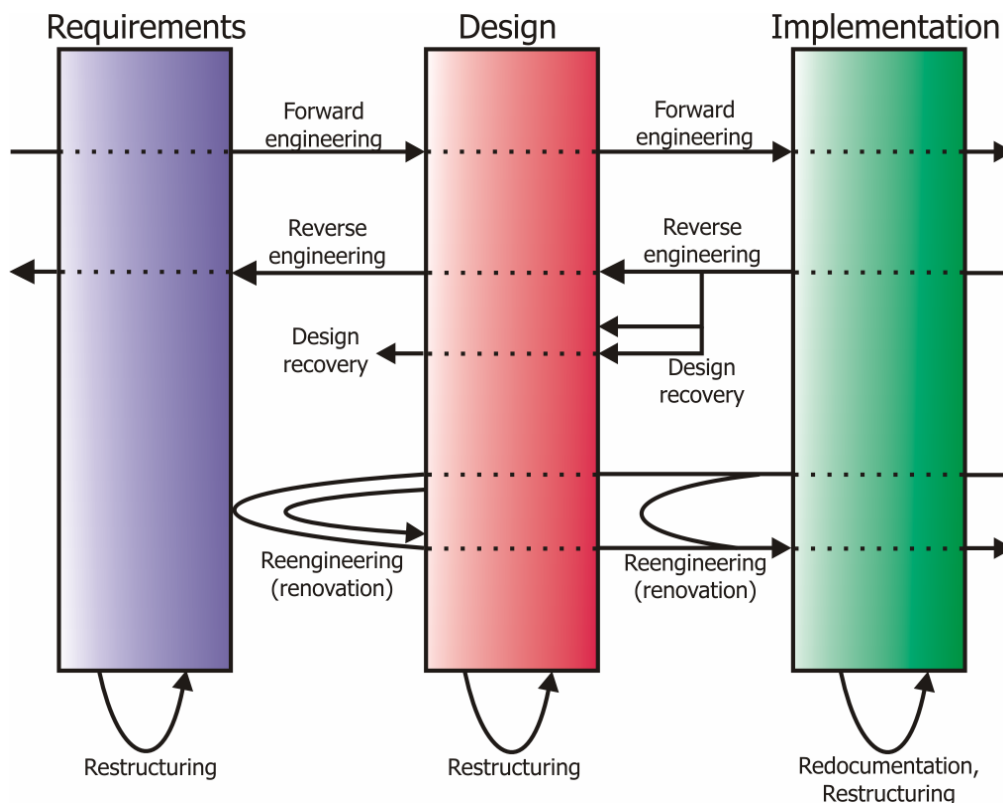


Abbildung 01) Einteilung der Softwareentwicklungsschritte [ChCr90]

Definition nach Chikofsky/Cross [ChCr90]:

Unter Reverse Engineering versteht man den Prozess der Analyse eines bestehenden Systems, mit dem Zweck:

- der Identifikation von Systemkomponenten und Beziehungen untereinander, sowie
- der Erzeugung von Darstellungen des untersuchten Systems auf unterschiedlichen, höheren Abstraktionsstufen

Wie die Definition schon ausdrückt, wird beim Reverse Engineering ein bereits implementiertes Programm verwendet, von dem die innere Struktur der Komponenten, sowie deren Beziehung zueinander unbekannt sind. Die Analyse einer solchen Applikation dient zur Informationsgewinnung, die zum Verstehen einer Software beiträgt, um evtl. Wartungsarbeiten durchzuführen, die Qualität zu ermitteln oder bei Designentscheidungen eigener Projekte unterstützt. Auch für das Re-Engineering können die Ergebnisse einer solchen Analyse wichtig sein, da auch hier zunächst das „Verstehen der Software“ im Vordergrund steht. Dieser Teil der Software-Entwicklung ist nicht zu unterschätzen, da allein die Wartung eines Produktes einen großen Zeit- und Kostenfaktor darstellt.

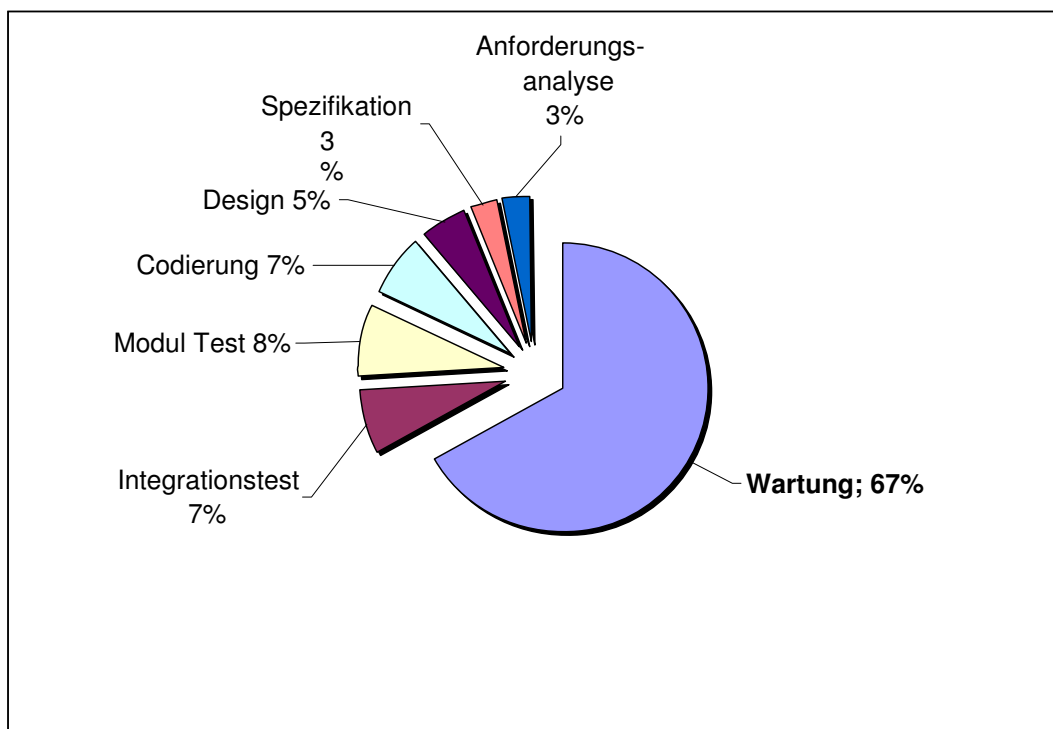


Abbildung 02) Kostenverteilung im Software-Lebenszyklus [ZSG79]

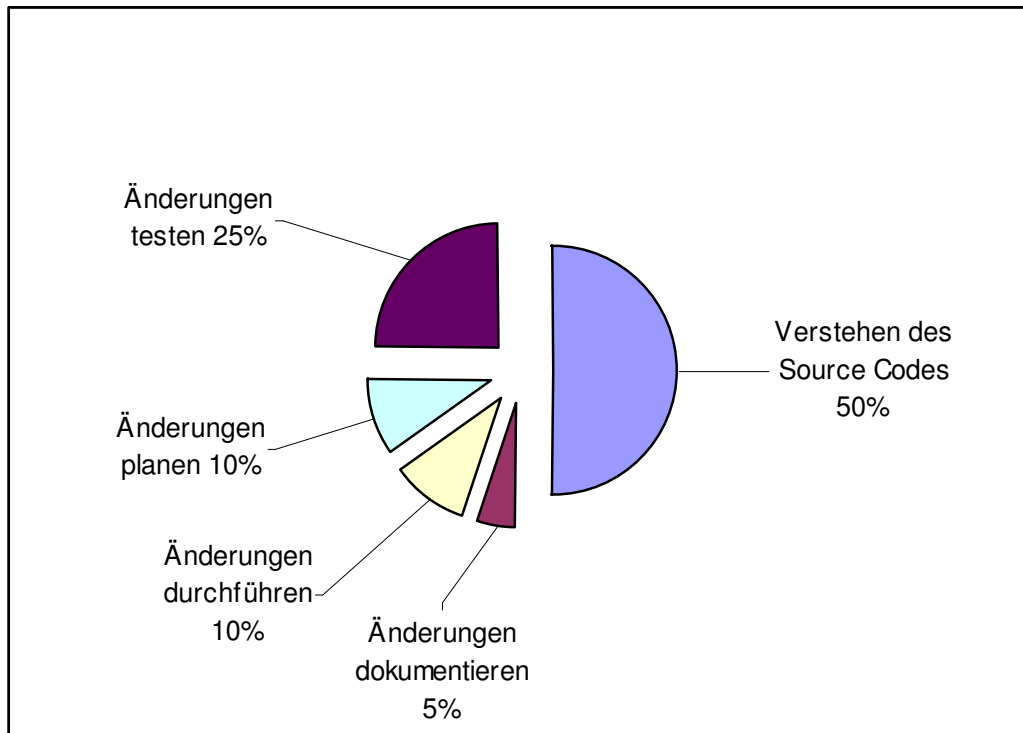


Abbildung 03) Aktivitäten in der Software-Wartung [Fra92]

Wie in Abbildung 2 veranschaulicht wird, spielt der Bereich der Wartung eine große Rolle im Software-Lebenszyklus, da auch nach der Veröffentlichung eines Produktes, viel Zeit und Geld in Verbesserungen und die Fehlerbeseitigung investiert werden muss. In Abbildung 3 wird dieser Bereich zusätzlich unterteilt in die einzelnen Aktivitäten der Entwickler. Dabei liegt der größte Zeit- und Kostenfaktor bei dem „Verstehen einer Software“. Um diese zu reduzieren, müssen Lösungen gefunden werden, um die Informationen über die Struktur, Aufbau und Abhängigkeiten möglichst einfach zu erhalten und dem Menschen auf leicht verständliche Art zugänglich zu machen.

Schon in den frühen Jahren der Softwareentwicklung wurde nach einer geeigneten Form zur Darstellung der Programmstruktur gesucht. Doch erst 1995 durch den Zusammenschluss von Grady Booch, James Rumbaugh und Ivar Jacobson konnte sich ihre Notation durchsetzen und wurde 1997 als Unified Modeling Language Version 1.1 zur Standardisierung bei der Object Management Group eingereicht und akzeptiert [Oe01]. Diese wurde bis heute weiterentwickelt und die momentan aktuellste Version ist 1.5, wobei 2.0 bereits in Vorbereitung ist. Diese bietet mehrere definierte Diagramme um den Aufbau, Zustände, Funktionen und Beziehungen in abstrakter Form grafisch wiederzugeben und somit die Entwickler zu unterstützen. Folgende Diagramme sind in der UML enthalten:

- Anwendungsfalldiagramme
- Klassendiagramme
- Aktivitätsdiagramm, Objektflussdiagramm
- Kollaborationsdiagramm
- Sequenzdiagramm
- Zustandsdiagramm
- Komponentendiagramm

Diese werden in allen Arten der Softwareentwicklung verwendet, da diese Abstrakte Darstellung für die Entwickler leichter zu verstehen ist. Beim Forward Engineering wird somit der genaue Programmaufbau, die Funktionen und Zusammenhänge, sowie die Zustände genau definiert und dient bei der Implementierung als Grundlage. Alle getroffenen Designentscheidungen sind in ihnen wieder zu finden. Da beim Reverse Engineering häufig nur der Programmcode zur Verfügung steht, können nicht alle diese Informationen ermittelt werden. Eine Möglichkeit besteht, den Aufbau einer Bibliothek heraus zu finden und in ein Klassendiagramm mit den Abhängigkeiten (Assoziationen) der einzelnen Komponenten untereinander zu überführen. Durch die Ermittlung der Attribute und Methoden kann ein Rückschluss auf die Aufgaben der einzelnen Programmelemente geführt werden. Es können durch die Ermittlung der Methodennamen und deren Parameter die Zugriffsmöglichkeiten auf die Funktionen der Bibliothek ermittelt werden.

### 1.2 FUJABA

Um die Erzeugung dieser Diagramme möglichst einfach zu gestalten, wurden, aufbauend auf die UML-Notationen, so genannte CASE-Tools (Computer Aided Software Engineering) entwickelt. Diese Programme sollen bei der Modellierung eines Software-Projektes vor der eigentlichen Implementierung helfen, um die Struktur und die Zusammenarbeit der Komponenten darzustellen. Teilweise bieten sie auch weitere Funktionalitäten um direkt aus den Diagrammen den Quellcode zu erzeugen, wobei die Semantik der einzelnen Methoden noch nachträglich selber implementiert werden muss.

Ein solches CASE-Tool mit dem Namen FUJABA (Akronym für „From UML to Java and back again“) wurde 1997 im Rahmen einer Diplomarbeit von T. Fischer, J. Niere und Torunski an der Universität Paderborn entwickelt [FNT98] und bezieht sich auf die objektorientierte Programmiersprache Java. Es unterstützt bei der Erstellung von den oben genannten UML Klassen-, Aktivitäten-, sowie Storydiagrammen (eine Mischung aus dem Aktivitäts- und Kollaborationsdiagramm). Diese können direkt in den Java-Quellcode übersetzt werden, wobei sich FUJABA dabei besonders hervorhebt. Durch das Storydiagramm werden nicht nur die Methodenköpfe, sondern auch die dahinter stehende Semantik in die Java-Dateien überführt.

Seit der Erstellung dieses CASE-Tools wurden weitere Funktionen hinzugefügt, die nicht nur dem Bereich des Forward Engineering zuzuordnen sind. Mittlerweile existiert eine Möglichkeit den Quellcode von Java-Dateien einzulesen und in ein entsprechendes Aktivitäten- und Klassendiagramm zu überführen. Somit kann FUJABA auch zum Reverse Engineering eingesetzt werden, um schon erstellte Programme zu untersuchen und grafisch die Informationen in abstrakter Weise darzustellen. Anhand dieser Diagramme bekommt man einen Einblick in die Programmstruktur und kann Abläufe und Zusammenarbeit der einzelnen Komponenten leichter verstehen als bei der Untersuchung des reinen Quelltextes.

FUJABA stellt ein eigenes Meta-Modell zur Verfügung in Form eines „Abstract Syntax Graph“ (ASG). Der Aufbau erfolgt wie ein Baukastensystem, wobei einzelne Elemente verbunden oder zusammengefügt werden. Die Informationen aus dem Reverse Engineering können verwendet werden, um aus dem ASG ein eigenes logisches Modell zu erstellen. FUJABA ist anschließend in der Lage, dieses grafisch in Form von UML-Diagrammen wieder zu geben. Auf ihnen können weitere Analysen folgen.

### 1.3 Problem

Die im vorherigen Kapitel genannte Analyse für das Reverse Engineering in FUJABA bezieht sich zur Zeit ausschließlich auf die Untersuchung von Java-Quellcode. Mittels einer Grammatikanalyse (Parser) werden die nötigen Informationen erhoben, in das ASG-Modell überführt und als Klassendiagramm innerhalb von FUJABA dargestellt. Doch gerade bei eingekaufter Software besteht das Problem, dass diese nur in kompilierter Form vorliegt und keine Designdokumentation beigefügt wurde. Eine Analyse ist zur Zeit mit FUJABA somit nicht möglich und der Mensch alleine kann den Java-Byte-Code nur schwer interpretieren.

Allerdings wäre die Untersuchung von kompilierten Dateien eben so wichtig. Es hilft beim Verstehen der Software und deren Abläufe, Struktur und Abhängigkeiten. Mittels der Suche nach Entwurfsmustern (engl. Designpattern) kann ein Rückschluss auf die Qualität getroffen werden. Dieser Aspekt ist besonders wichtig vor dem Kauf einer Bibliothek, um die Arbeit der Entwickler überprüfen zu können und nicht Geld für ein minderwertiges Produkt auszugeben. Der eigene Qualitätsstandard kann somit gesichert werden. Des Weiteren kann diese Art der Analyse bei der Verwendung von Fremd-Bibliotheken innerhalb des eigenen Projektes helfen. Schon bei der Entwicklung mittels FUJABA können die analysierten Klassen in die Diagramme des eigenen Projektes eingebunden werden und zur Komplettierung der Dokumentation dienen. Dieses Vorgehen spart zusätzlich Zeit und Geld, da bei der automatischen Quellcodeerstellung in FUJABA die Fremdbibliothek mit berücksichtigt wird. Die Zugriffe auf die Komponenten müssen nicht von den Entwicklern nachträglich hinzugefügt werden.

Die Untersuchung von Java-Byte-Code birgt allerdings Probleme. Der Inhalt von Klassendateien ist für den Menschen nur schwer interpretierbar und das direkte Analysieren des Inhaltes der Dateien erfordert einen aufwendigen Algorithmus. Um an die Informationen zu gelangen, muss ein anderer Lösungsweg gefunden werden, um diese anschließend in ein UML-Klassendiagramm überführen zu können.

Ein weiteres Problem besteht bei der Untersuchung der Assoziationen. Mittels ihnen wird die Verbindung zwischen den einzelnen Klassen gekennzeichnet. Es muss unterschieden werden zwischen Referenzen für eine unidirektionale Verbindung oder Assoziationen für eine bidirektionale Verbindung. Die Kardinalitäten zeigen deren Häufigkeit an. Attribute besitzen häufig nur einfache Verbindungen an. Arrays hingegen geben einen Hinweis auf eine höhere Kardinalität. Ein weiteres Problem ist die Erkennung von Containerklassen und der darin gespeicherten Objekttyps. Auch hierdurch wird eine höhere Kardinalität signalisiert. Folgende Abbildung stellt einen solchen Problemfall dar.

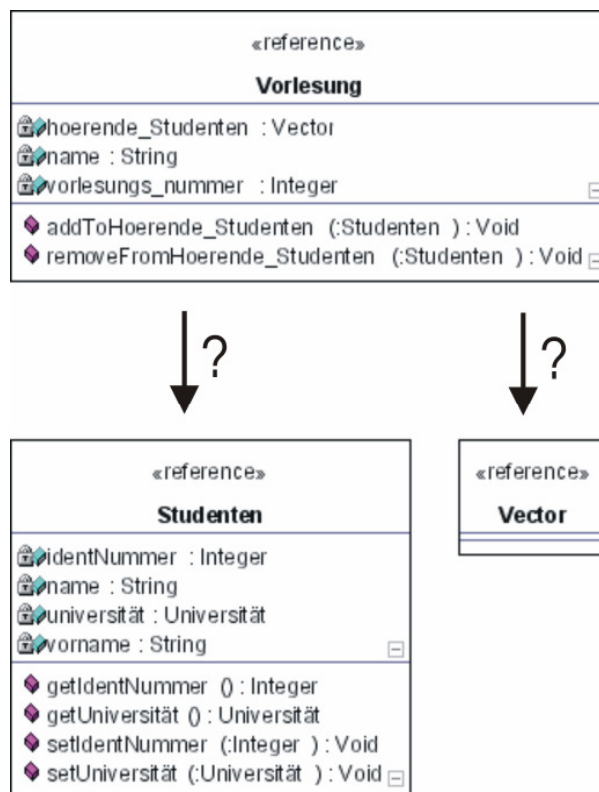


Abbildung 04) Beispiel eines Problems der Assoziationserkennung

Das Problem besteht bei dem Erstellen eines aussagekräftigen Diagramms. Wie in Abbildung 4 gezeigt wird, wird innerhalb der Klasse „Vorlesung“ ein Attribut „hoerende\_Studenten“ vom Typ „Vector“ deklariert. Eine einfache Lösung wäre nun eine Referenz mit der Kardinalität „1“ zwischen der Klasse „Vorlesung“ zur Klasse „Vektor“ einzufügen. Aber diese Art der Assoziationsfindung ist nicht aussagekräftig genug. Der Vektor dient nur als ein Hilfskonstrukt zur Speicherung von Objekten des Typs „Studenten“. Daher wäre eine Referenz mit der Kardinalität „0..n“ zu dieser Klasse sinnvoller, um die Abhängigkeiten zu verdeutlichen. Es muss ein Weg gefunden werden, die Assoziationen zu den inneren gespeicherten Objekten herauszufinden. Java (Version 1.4.2) stellt allerdings zur Zeit noch keine Methode zur Verfügung, den Typ innerhalb solcher Containerklassen zu ermitteln. Daher ist ein Algorithmus zu entwerfen, der anhand der Zugriffsmethoden auf Objekte, die zur Speicherung anderer dienen, und deren Parametern diese Assoziationen ermittelt und entsprechende Kardinalitäten herausfindet.

## 1.4 Zielsetzung

Um die Funktionalitäten von FUJABA im Bereich des Reverse Engineering zu erweitern, soll ein Plug-in entwickelt werden, das Klassendiagramme aus in Java-Byte-Code vorliegenden Bibliotheken erstellt und mittels einer Mustererkennung die Assoziationen ermittelt. Diese Datenerhebung soll mittels der Java-Reflection-API geschehen. Dazu müssen jeweils von jeder Klassendatei eigene Objekte erzeugt werden.

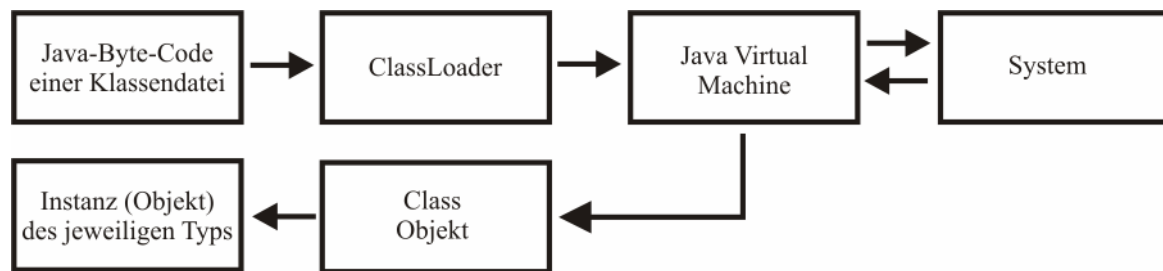


Abbildung 05) Ablauf der Objekterstellung aus Klassendateien

Es existiert die Möglichkeit von Klassendateien Meta-Objekte zu erzeugen und hieraus die nötigen Informationen zu erhalten. Abbildung 5 zeigt den Ablauf um an diese zu gelangen. Dabei wird der Java Byte Code von einem ClassLoader geladen und anschließend an die Java-Virtual-Machine (JVM) übergeben. Die JVM übersetzt diesen in den Systemabhängigen Code und erstellt Meta-Objekte vom Typ „Class“, die weiter analysiert werden können. In ihnen ist der Aufbau eines Objektes dieser Klasse genau beschrieben und dient zum Erstellen von Instanzen dieses Typs.

Java bietet für die Analyse die Reflection-API (ab dem JDK 1.1 enthalten). Es ist dabei nicht nötig eine direkte Instanz des zu untersuchenden Objektes zu erstellen. Die Reflection-API kann auch schon auf die Meta-Objekte (vom Typ „Class“) angewendet werden. Mit Hilfe dieser Bibliothek ist es möglich die wichtigsten Daten, wie z.B. Attribute, Methoden und Konstruktoren mit deren Parametern und Sichtbarkeit, auszulesen.

Diese Informationen können verwendet werden, um aus dem Meta-Modell von FUJABA ein eigenes Modell zu erstellen und grafisch als Klassendiagramm wiederzugeben.

Eine weitere Problematik besteht nun in der Analyse der Assoziationen und Referenzen zwischen den einzelnen Klassenobjekten. Ein Teil wird schon über die deklarierten Attribute bestimmt, jedoch kann somit schlecht eine Aussage über die Kardinalitäten getroffen werden. Dies gilt besonders, wenn Objekte nur innerhalb von Containerklassen gespeichert werden. Da Java keine Möglichkeit bietet diesen inneren Typ der gespeicherten Objekte zu ermitteln, muss eine andere Technik angewendet werden, um die Verbindungen zu analysieren. Eine Möglichkeit bieten die Zugriffsmethoden, die Objekte in den Containern speichern und auslesen. Das zu erstellende Plug-in muss diese identifizieren und bei der Erstellung des Klassendiagramms berücksichtigen. Da für diese Methoden häufig Konventionen für den Namen bestehen, soll der Benutzer durch entsprechende Einstellungen die Suche anpassen können.

Eine Lösung für die Assoziationserkennung besteht darin, einen Algorithmus zu entwerfen, der danach sucht. Jedoch würde dadurch die Flexibilität nicht gewährleistet. Eine Änderung durch den Benutzer nach anderen Eigenschaften innerhalb des Klassendiagramms kann nur schwer in die feste Implementierung übernommen werden. Eine andere und sinnvoller Lösung ist es, die Suche nicht durch einen fest implementierten Algorithmus durchzuführen, sondern unabhängig alleine auf dem erstellten Modell des Klassendiagramm. FUJABA bietet bereits die Möglichkeit mittels zweier schon bestehender Plug-ins eine Suche nach vorher definierten Mustern durchzuführen. Diese können für die Assoziationsanalyse genutzt werden. Das Plug-in „PatternSpecification“ [NSWW02] bietet die Möglichkeit Musterregeln (engl. Patternrules) zu erstellen, die die Struktur einer Assoziation wieder spiegeln. Somit ist die Suche durch den Benutzer leicht veränderbar. Die erstellten Muster können zu einem Katalog zusammengestellt werden, den ein weiteres Plug-in verwendet. Die „InferenceEngine“ [Wen01] sucht anhand dieser vordefinierten Muster nach entsprechenden Vorkommen im logischen Modell des Klassendiagramms und fügt bei zutreffendem Vergleich zusätzlich Annotations-Objekte ein. Diese beinhalten alle wichtigen Daten und

können anschließend, nachdem die Mustersuche beendet wurde, durch die eigentlichen Assoziationen ersetzt werden.

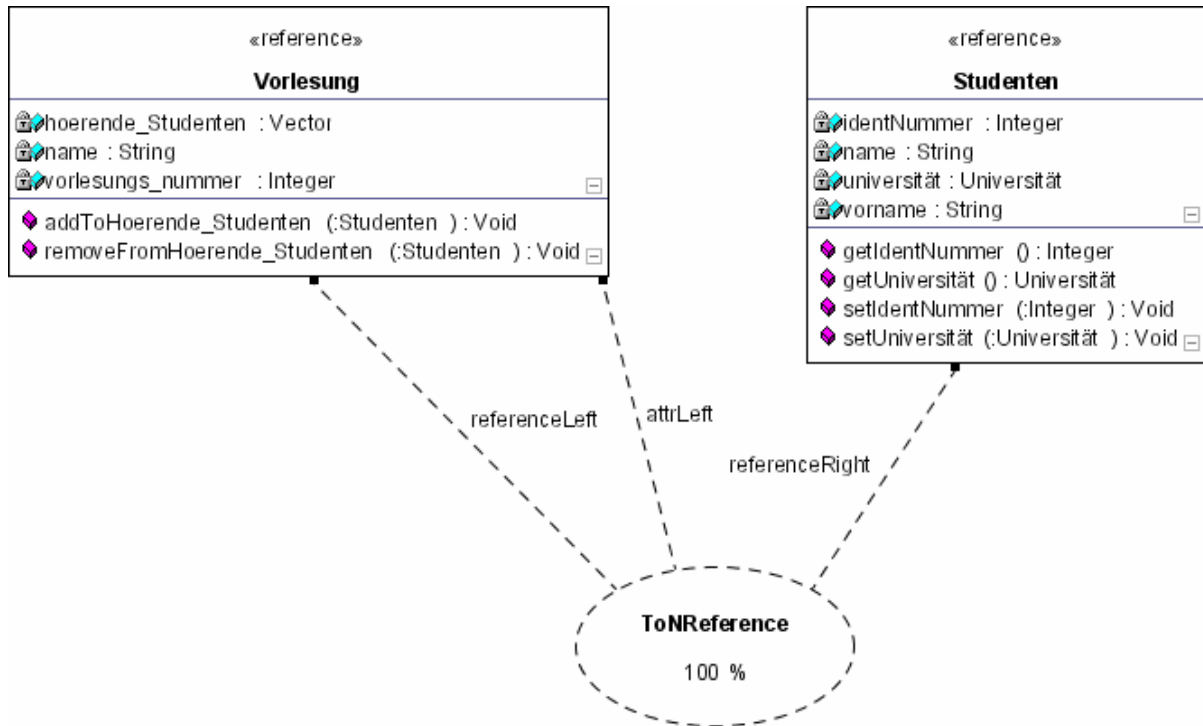


Abbildung 06) Erkanntes Muster durch Annotation gekennzeichnet

Wie in Abbildung 6 veranschaulicht, wird in das Klassendiagramm eine Markierung eingefügt, falls ein entsprechendes Muster zutrifft. Diese Annotation (gestricheltes Oval) besitzt alle wichtigen Informationen und Referenzen, die genutzt werden können, um eine Assoziation zu erstellen.

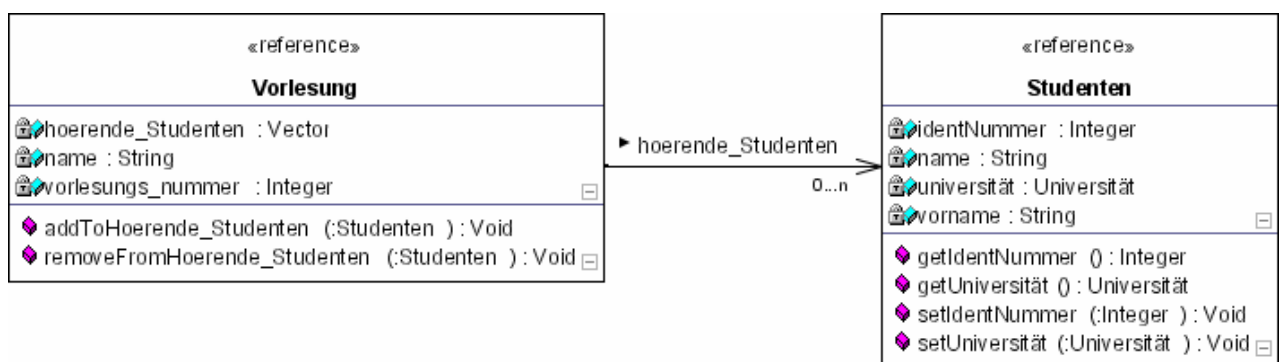


Abbildung 07) Assoziation aus Annotation erstellt

Abbildung 7 zeigt die erstellte Assoziation innerhalb des Klassendiagramms. Alle für die Erstellung notwendigen Daten wurden der zuvor eingefügten Annotation entnommen. Um das Diagramm möglichst übersichtlich zu halten, kann diese anschließend gelöscht werden. Die Elemente, die für die Assoziation notwendig sind, können ausgeblendet werden (in Abbildung 7 noch dargestellt). Das würde in diesem Fall das Attribut „hoerende\_Studenten“ vom Typ „Vector“ betreffen, so wie die Zugriffsmethoden auf diese Containerklasse.

## 1.5 Struktur der Studienarbeit

Im 2. Kapitel wird ein Vergleich zu anderen CASE-Tools erstellt und deren Funktionalitäten im Bereich des Reverse Engineering erläutert, besonders in Hinblick auf die Erkennung von Assoziationen.

Im 3. Kapitel wird das Konzept vorgestellt, mit dem versucht wird, die an das Plug-in gestellten Aufgaben zu lösen. Hierbei wird näher auf den ClassLoader und die Java-Reflection-API eingegangen und von welchem Nutzen ihre Funktionalitäten für die Byte-Code-Analyse sind. Ein weiterer Schwerpunkt liegt auf den zusätzlich verwendeten Plug-ins „InferenceEngine“ und „PatternSpecification“, deren Verwendung bei der Untersuchung der Assoziationen zwischen den einzelnen Programmkomponenten notwendig ist.

Das 4. Kapitel befasst sich mit der technischen Realisierung und erläutert genauer, wie das im 3. Kapitel erstellte Konzept umgesetzt wird. Es liefert einen tieferen Einblick in die einzelnen Klassendateien des Plug-in und deren Funktionen, sowie in das Vorgehen bei der Analyse der kompilierten Dateien, der Erstellung eines Klassendiagramms und der Erkennung von Assoziationen.

Im 5. Kapitel soll anhand eines Beispieldurchlaufes der Ablauf einer Analyse veranschaulicht werden, um den Umgang mit dem Plug-in zu erklären. Hierbei wird auch näher auf die Einstellungsmöglichkeiten des JBCAnalyser eingegangen.

Im 6. Kapitel wird eine kurze Zusammenfassung gegeben über die Möglichkeiten der Analyse, sowie die weiteren Verwendungsmöglichkeiten der Ergebnisse.



## Kapitel 2

# 2 Reverse Engineering mittels vergleichbarer CASE-Tools

Neben FUJABA existieren noch weitere CASE-Tools, die auch Funktionalitäten im Bereich des Reverse Engineering bieten. Um eine Vergleichsmöglichkeit gegenüber dem im Rahmen dieser Studienarbeit zu erstellenden Plug-in zu ermöglichen, wird im Folgenden eine Auswahl dieser vorgestellt.

### 2.1 Together

Entwickelt wurde dieses Tool von Borland Software Corporation. Es existieren mehrere Versionen, die zum Teil als Plug-in in weiteren Entwicklungsprogrammen eingebunden werden können. Für diesen Test wurde die aktuelle Version Together Control Center Version 6.1 verwendet, das eine eigenständige Applikation ist.

In Bezug auf die Analyse von kompilierten Klassendateien in Java-Byte-Code besitzt dieses Programm die Möglichkeit, Jar-Archivdateien einzulesen, zu analysieren und in ein Klassendiagramm zu überführen. Dabei werden folgende Daten erhoben:

- Klassennamen
- Package
- Attribute
- Methoden
- Oberklassen
- Implementierte Interfaces

Jedoch bietet Together keine Möglichkeiten zur Assoziationserkennung. Nur die Verbindungen zu den Oberklassen und implementierten Interfaces werden erfasst und grafisch wiedergegeben. Der Benutzer muss somit selber durch deklarierte Attribute auf die Verbindungen schließen. Methoden werden innerhalb des Klassendiagramms nur mit ihrem Namen dargestellt. Den Parametertyp und Rückgabewert kann der Benutzer sich durch Auswahl der entsprechenden Operation in einem „Properties“-Feld anzeigen lassen, was die selbständige Ermittlung der Assoziationen erschwert.

Ein weiterer Nachteil ergibt sich durch die Beschränkung der Auswahl auf Archivdateien. Die Analyse von einzelnen Klassendateien oder Verzeichnissen ist somit schwer möglich. Der Benutzer muss diese erst in ein entsprechendes Format überführen.

Together bietet für die Analyse keine Einstellungsmöglichkeiten und kann somit nicht an eigene Anforderungen angepasst werden.

## 2.2 EclipseUML

Omondo entwickelte dieses Plug-in für Eclipse<sup>1</sup> und ist somit keine eigenständige Applikation. Es existieren mehrere Versionen, die alle zum Reverse Engineering eingesetzt werden können, jedoch zum Teil mit Einschränkungen. Für den Test wurde die Version EclipseUML Studio 1.0.1 verwendet. Für das Diagramm werden folgende Daten erhoben:

- Klassenname
- Package
- Implementierte Interfaces
- Oberklassen
- Attribute
- Methoden

Es wurden implementierte Interfaces und Oberklassen erkannt und innerhalb des Diagramms durch eine spezielle Verbindung gekennzeichnet. Des Weiteren konnte dieses Plug-in vorhandene Assoziationen und Referenzen mit deren Kardinalitäten ermitteln. Auch Getter- und Settermethoden, die zum Speichern von Objekten in Containerklassen verwendet werden, wurden hierfür berücksichtigt. Es besitzt Einstellmöglichkeiten, um die Suche an seine Bedingungen anzupassen.

Jedoch konnte EclipseUML während der Testphase keine Analyse alleine auf dem Java-Byte-Code durchführen. So musste für die Analyse immer der zugehörige Quellcode mit beigefügt sein.

1. Eclipse ist eine Entwicklungsumgebung für Java-Applikationen und wurde in Zusammenarbeit vieler renommierter Unternehmen erstellt. Nähere Informationen unter [www.eclipse.org](http://www.eclipse.org)

## 2.3 Poseidon for UML

Dieses Programm wurde von Gentleware AG entwickelt und ist eine allein stehende Applikation für UML-Diagramme. Auch diese steht in mehreren Versionen zur Verfügung. Die aktuelle Community Edition ist kostenlos, jedoch mit Einschränkungen, so dass die Möglichkeit zur Analyse von Bibliotheken in Java-Byte-Code nicht möglich ist. Daher wurde für diesen Test die aktuelle Version Poseidon for UML Professional Edition 2.5 verwendet. Die Analyse von in Java-Byte-Code vorliegenden Bibliotheken ist durch die Auswahl nur auf Jar-Archivdateien beschränkt. Die innerhalb der Untersuchung erhobenen Daten:

- Klassenname
- Package
- Implementierte Interfaces
- Oberklassen
- Attribute
- Methoden

Innerhalb des Klassendiagramms werden Abhängigkeiten zu den implementierten Interfaces und Oberklassen grafisch dargestellt, jedoch keine sonstigen Assoziationen ermittelt. Für jede untersuchte Klasse wird ein eigenes Objekt mit dem zugehörigen Namen in das Diagramm eingefügt, aber keine Attribute und Methoden. Diese werden dem Benutzer erst bei Auswahl eines Elementes in einem zusätzlichen Feld angezeigt. Dabei werden für Attribute der Namen und der Typ ausgegeben mit einer Anzahl ihres Vorkommens. Für einzelne Variablen steht hierbei eine „[1]“ und für Arrays eine „[1..\*]“. Bei Methoden wird der Name, Typ, Sichtbarkeit und Rückgabewert angegeben.

Für das Reverse Engineering von kompiliertem Java-Code ist dieses Programm nur bedingt einsetzbar. Obwohl in einem zusätzlichen Feld ein eigener Bereich mit dem Namen „Associations“ enthalten ist, wurden keine ermittelt. Nur zu den Oberklassen und Interfaces wurden Abhängigkeiten grafisch dargestellt. Arrays innerhalb der Attribute werden durch einen Zahlenwert [1..\*] speziell gekennzeichnet. Der Benutzer muss die Assoziationen selbstständig ermitteln.



## Kapitel 3

### 3 Konzept zur Analyse des Java Byte Codes

Zur Zeit ist FUJABA im Bereich des Reverse Engineering auf die Analyse von Quellcode beschränkt, um diese in UML-Klassendiagramme darzustellen. Diese Funktionalität soll durch das Plug-in „Java-Byte-Code Analyser“ (kurz JBCAnalyser) erweitert werden. Der Ablauf geschieht in zwei Stufen. Zunächst müssen alle relevanten Informationen aus den kompilierten Dateien erhoben werden, um diese in ein Klassendiagramm innerhalb von FUJABA zu überführen. Anschließend kann dieses als Grundlage für eine Assoziationserkennung genutzt werden, um durch das zusätzliche Einfügen der Verbindungen zwischen den einzelnen Komponenten einen verständlicheren Einblick in die Abläufe innerhalb einer Bibliothek zu erhalten.

#### 3.1 Erstellen eines Klassendiagramms

Um ein Klassendiagramm erstellen zu können, müssen zunächst alle notwendigen Informationen aus den kompilierten Dateien analysiert werden. Mittels dieser Daten kann ein logisches Modell (ASG) aus dem Meta-Modell von FUJABA erstellt werden. Da es sich bei dieser Analyse ausschließlich um die Erstellung von Klassendiagrammen handelt, werden nur die UML-Elemente benötigt. Dieses logische Modell kann anschließend grafisch innerhalb von FUJABA als Klassendiagramm dargestellt werden. Um dieses möglichst aussagekräftig zu gestalten sollten mindestens diese Informationen ermittelt werden:

- Klassenname
- Attribute
- Methoden
- Oberklassen
- Interfaces

Für die folgende Assoziationserkennung sollten noch zusätzlich die Parameter und der Rückgabebetyp der Methoden ermittelt werden. Zwei mögliche Techniken, um an die nötigen Informationen zu gelangen, werden in den folgenden Kapiteln vorgestellt.

##### 3.1.1 Direkte Untersuchung der kompilierten Dateien

Dateien in Java-Byte-Code beinhalten keinen Maschinencode der direkt ausgeführt werden kann. Vielmehr ist der Inhalt eine Zwischensprache, die erst durch den Javainterpreten in den hardwareabhängigen Code übersetzt wird. Hiermit wird die Plattformunabhängigkeit gewährleistet. Es besteht die Möglichkeit, diese Zwischensprache direkt auszulesen und alle notwendigen Informationen herauszufiltern.

```
!de/upb/lib/plugins/AbstractPlugin
PLUGIN_KEYLjava/lang/String; class$0Ljava/lang/Class;
Synthetic<clinit>()VCodejava/lang/StringBuffer
de.upb.jbcanalyser.JBCAnalyserjava/lang/Class
forName%(Ljava/lang/String;)Ljava/lang/Class;
java/lang/NoClassDefFoundError java/lang/Throwable
getMessage()Ljava/lang/String;
<init>(Ljava/lang/String;)V!"
```

Abbildung 08) Auszug aus einer kompilierten Klassendatei

Wie der oben in Abbildung 8 gezeigte Auszug aus einer kompilierten Datei zeigt, würde sich die direkte Informationsgewinnung aus dem Code als sehr schwierig gestalten.

### 3.1.2 Informationsgewinnung mittels der Java-Reflection-API

Die Java-Reflection-API wurde ab dem JDK 1.1 eingeführt, um eine Schwachstelle der vorherigen Versionen auszubessern. Vorher mussten zur Kompile-Zeit alle verwendeten Klassen und Bibliotheken bekannt sein. Die Java-Reflection-API bietet die Möglichkeit von unbekanntem Objekten zur Laufzeit Meta-Daten zu erhalten und diese anschließend zu verwenden. Mittels dieser Schnittstelle können unter anderem folgende Werte ausgelesen werden [Krü02]:

- Objekttyp -> Klasse
- Deklarierte Attribute mit Namen, Typ und Sichtbarkeit
- Definierte Methoden mit Namen, Parameter, Rückgabewert und Sichtbarkeit
- Oberklasse
- Implementierte Interfaces

Diese Eigenschaft kann auch für die Analyse von kompilierten Dateien verwendet werden. Dazu muss zunächst jeweils ein Objekt erstellt werden, auf das die Java-Reflection-API zugreifen kann. Java bietet hierfür die Klasse `ClassLoader`. Dieser kann den Inhalt von kompilierten Javodateien auslesen und an die Java-Virtual-Machine übergeben. Die Daten werden von ihr interpretiert und ein Objekt vom Typ „Class“ erstellt. Dieses stellt keine Instanz der Klasse dar, sondern dient als ein Meta-Modell, das alle relevanten Daten zum Erstellen eines Objektes von diesem Typ beinhaltet. Die Java-Reflection-API kann auf diese Objekte angewendet werden und man erhält somit alle wichtigen Informationen, die zum Erstellen eines aussagekräftigen Klassendiagramms benötigt werden.

Da sich diese Technik einfacher realisieren lässt, als die Analyse des Inhalts von kompilierten Dateien, wird das Plug-in „JBCAnalyser“ auf dieser Art der Informationsgewinnung basieren. Mittels der somit gewonnenen Daten kann aus dem Meta-Modell von FUJABA ein eigenes Modell erstellt und innerhalb eines Klassendiagramms grafisch dargestellt werden.

## 3.2 Assoziationserkennung

In Hinblick auf die Abhängigkeiten ist dieses alleine jedoch noch nicht ausreichend. Es sollen Assoziationen zwischen den einzelnen Klassenobjekten innerhalb des Diagramms eingezeichnet werden. Um diese herauszufinden kann ein fest implementierter Suchalgorithmus innerhalb des Plug-in „JBCAnalyser“ eingebunden werden, jedoch würde auf diese Weise die Flexibilität eingeschränkt werden, da Änderungen nur schwer übernommen werden könnten. Diese aber sind zwingend notwendig, um die Analyse anzupassen. Insbesondere bei der Erkennung von Assoziationen zu Objekten, die innerhalb von Container-Klassen gespeichert werden. Dieses kann nur über die Zugriffsmethoden geschehen, die häufig speziellen Namenskonventionen unterliegen. Die Parameter und Rückgabewerte geben dabei Aufschluss, zu welcher Klasse eine Assoziation erstellt werden muss. Die Identifizierung der Zugriffsmethoden geschieht über einen Vergleich derer Namen mit den Einträgen einer Liste. Diese soll entsprechend durch den Benutzer veränderbar sein. Für das Herausfinden der Assoziationen besitzt FUJABA bereits die Funktionalität der Mustererkennung [NSWW02]. Dabei wird nach vorher definierten Strukturen innerhalb des ASG (hier basierend auf dem Klassendiagramm) gesucht. Durch Auffinden solcher Musterregeln (engl. Patternrules) wird das Diagramm durch Annotationen erweitert, die in den folgenden Kapiteln noch genauer erläutert werden. Diese markieren die Elemente innerhalb des Diagramms, auf die die Struktur eines Musters zutrifft. Diese Art der Suche nach Assoziationen kann der Benutzer durch die Veränderung der Muster und Angabe der Zugriffsmethoden auf Container-Klassen an die eigenen Bedürfnisse anpassen und es wird somit eine hohe Flexibilität gewährleistet.

### 3.2.1 Erstellen eines Musterkataloges

Die Muster können mit dem schon genannten Plug-in „PatternSpecification“ erstellt werden. Sie beschreiben die genaue Struktur innerhalb des ASG, die auf eine Assoziation hinweisen. Diese sollen durch den Benutzer modifiziert werden können, um eine angepasste Suche zu ermöglichen. Es ist eine Standardkatalog zu erstellen, der möglichst viele Assoziationen definiert. Der Benutzer kann diesen wieder verwenden und verändern, oder einen neuen eigenen erstellen. Ein Beispiel für ein solches Muster wird in der folgenden Abbildung veranschaulicht.

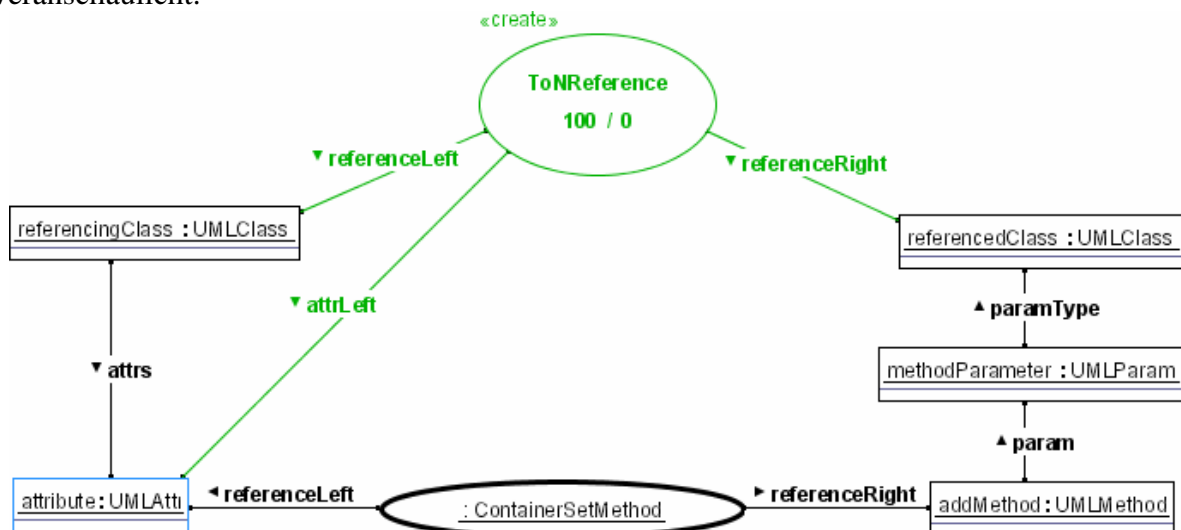


Abbildung 09) Beispiel einer Musterregel

In der Abbildung 9 wird ein Beispielmuster gezeigt für eine Referenz zwischen zwei Klassen mit der Kardinalität „n“. Die vergebenen Namen für die ASG-Elementen (Rechtecke) sind dabei nicht von Relevanz. Dieses Beispiel zeigt eine Besonderheit. Eine andere Regel (im Oval „ContainerSetMethod“) wird hierbei wieder verwendet. Erst wenn beide übereinstimmen, gilt dieses Muster der „ToNReference“ als erkannt. Es können somit Regeln in andere mit eingebunden werden, um Zeit bei der Erstellung zu sparen und die Übersicht zu wahren.

Die ASG-Elemente (in den Rechtecken) definieren hierbei die Struktur des Musters nach dem innerhalb des ASG gesucht werden soll. Das Oval, das mit <<create>> gekennzeichnet ist, bildet die Annotation, die bei Zutreffen des Musters in das Klassendiagramm eingefügt wird.

Die wichtigen Element und eine Erklärung zu dessen Aufgabe soll folgende Tabelle erläutern:

Regelname (unterliegt Konvention)	Befindet sich innerhalb des Ovals, das mit <<create>> gekennzeichnet ist (hier: ToNReference). Der Name dient später zur Identifizierung der Assoziationen oder Referenz, sowie deren Kardinalitäten.
annotatedElement (unterliegt Konvention)	Verbindung von dem Oval zu den UML-Elementen (hier: referenceLeft, referenceRight, attrLeft). Mittels der vergebenen Namen kann in der späteren Analyse die Elemente im Klassendiagramm herausgefunden werden, die zu diesem Muster gehören.
ASG-Elemente	Rechtecke (hier: UMLClass, UMLAttr). Beschreiben die Struktur einer Assoziation, die später mit den Elementen aus dem Klassendiagramm verglichen wird.
Pattern-Link	Verbindungen zwischen UML-Elementen (hier: attr, attrType). Dienen ebenfalls zur Strukturbeschreibung des Musters.
Trigger	Objekt mit fetter Umrandung (hier: ContainerSetMethod). Dient als Auslöser. Erst durch das Auffinden eines solchen Elementes im logischen Modell des Klassendiagramms, wird das komplette Muster verglichen. In Jedem Muster muss ein Trigger definiert werden, wobei jedes Element des ASG dafür genutzt werden kann.
Eingebundenes Muster	Oval (hier: ContainerSetMethod). Bindet eine andere Regel in die aktuelle ein. Die Struktur des eingebundenen Musters und die des einbindenden müssen zutreffen.
Constraints	In diesem Beispiel nicht vorhanden. Dient zur Erweiterung des Musters, falls dieses alleine nicht durch die UML-Elemente komplett definiert werden kann. Rückgabewert muss vom Typ „boolean“ sein.

Der Name der Regel und die Namen der von ihr ausgehenden Verbindungen (annotatedElement) müssen einer Konvention unterliegen, da sie bei der späteren Erstellung der Assoziationen die Referenzierung zu den Elementen im Klassendiagramm, auf die ein Muster zutrifft, ermöglichen. Dieses geschieht durch die vergebenen Namen.

Es lassen sich somit mehrere Muster definieren, die die jeweiligen Assoziationen beschreiben. Anschließend wird von ihnen ein Katalog erstellt, der für die anschließende Suche nach den Assoziationen benötigt wird. Die Muster lassen sich leicht durch den Benutzer verändern. Es können eigene definiert werden, um die Suche an seine Anforderungen anzupassen. Diese unterliegen auch den Konventionen für Musterregeln.

Eine genauere Beschreibung zu der Erstellung von Mustern und die Erkennung innerhalb des ASG ist in der Diplomarbeit [Wen01] genauer beschrieben.

### 3.3 Mustersuche innerhalb des Klassendiagramms

Mittels des erstellten Musterkataloges kann nach deren Strukturen innerhalb des Diagramms, bzw. des erstellten Modell aus dem ASG gesucht werden. Aus Effizienzgründen werden diese nicht komplett verglichen, sondern zunächst nur nach den gesetzten Triggern gesucht. Erst wenn ein entsprechender gefunden wurde, wird das komplette Muster in die Analyse mit einbezogen. Für jede Übereinstimmung wird eine Markierung vom Typ „UMLAnnotation“ in das Klassendiagramm eingefügt, die die beiden Klassen referenziert, auf die eine Regel zutrifft und somit eine Assoziation besteht. Hierfür wird das Meta-Modell von FUJABA erweitert um die Klasse GFRNAnnotation, die als Basis jeder Annotation dient [Wen01] und der Klasse UMLAnnotation [Wen01], die für die Visualisierung notwendig ist.

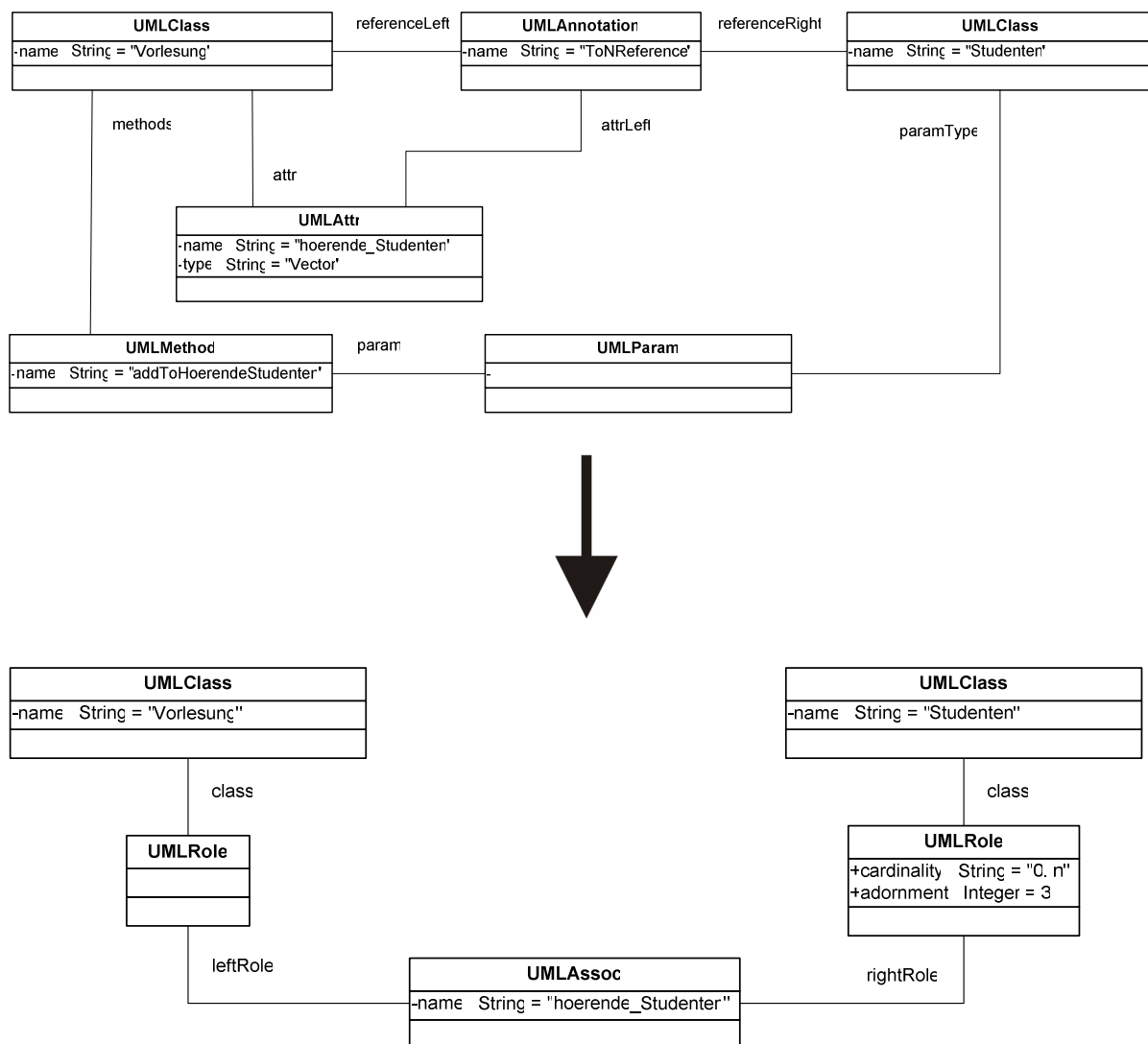


Abbildung 10) Überführen von UMLAnnotationen in Assoziationen innerhalb des ASG

Nachdem die Suche nach den Mustern beendet ist, kann der JBCAnalyser nach Objekten vom Typ „UMLAnnotation“ innerhalb des Klassendiagramms suchen. Die Ansicht in Abbildung 10 zeigt eine Beispielansicht in das Modell des ASG eines Klassendiagramms, wobei zur Übersicht nur die UMLAnnotation eingezeichnet wurde. Diese beinhaltet die zugehörige

GFRNAnnotation. Aus ihnen wird die zu erstellende Assoziation ermittelt. Aus der GFRNAnnotation können alle zusätzlichen Informationen über die Verbindungen (referenceLeft, referenceRight und attrLeft) bezogen werden. Das Attribut dient hierbei als Namensgeber für die Assoziation. Die Kardinalitäten werden durch den Namen der UMLAnnotation (hier: ToNReference) bestimmt und innerhalb der jeweiligen Klasse UMLRole gesetzt. Über ihn wird zusätzlich ermittelt, ob es sich um eine Referenz oder Assoziation handelt, da es hierfür einen Unterschied in der grafischen Wiedergabe gibt. Bei Referenzen wird ein Integerwert in der entsprechenden UMLRole gesetzt (adornment = 3). Bei der grafischen Darstellung des Klassendiagramms wird auf dieser Seite der Assoziation ein Pfeil wieder gegeben, der die einseitige Beziehung der Komponenten signalisiert. Nach der Erstellung der Assoziation kann diese zwischen die beiden Klassen, auf die das Muster zutraf, eingefügt werden. Somit wird die Assoziation, wie schon in Abbildung 7 gezeigt, auch innerhalb des Klassendiagramms sichtbar. Die UMLAnnotation kann anschließend gelöscht werden, da sie keine für den Benutzer relevanten Informationen enthält. Die GFRNAnnotation sollte jedoch erhalten bleiben, damit bei mehrfacher Anwendung des JBCAnalyser und somit der Mustererkennung keine doppelten Verbindungen erstellt werden. Das Vorgehen der Assoziationserstellung soll der folgende abstrakte Algorithmus verdeutlichen.

For each Element in Classdiagramm

If (Element == UMLAnnotation)

```
{
    Create Association of type "UMLAssoc"
    Create leftRole of type "UMLRole"
    Create rightRole of type "UMLRole"
    leftRole.setTarget = get Left Class of UMLAnnotation by ident-string "referenceLeft"
    rightRole.setTarget = get right Class of UMLAnnotation by ident-string "referenceRight"
    leftRole.setCardinality = get Cardinality for the left side by compare the name of UMLAnnotation
    rightRole.setCardinality = get Cardinality for right side by comparing the name of UMLAnnotation
    Association.setLeftRole = leftRole
    Association.setRightRole = rightRole
    Association.setName = get name of UMLAnnotation from additional Annotations
    Add Association to Classdiagramm
    Remove UMLAnnotation
}
```

Nachdem der JBCAnalyser alle Annotationen durch Assoziationen ersetzt hat, sollte ein komplettes Klassendiagramm erstellt worden sein, das dem Benutzer eine Übersicht über den Inhalt einer Bibliothek in Java Byte Code gibt, sowie den Zusammenhang zwischen den einzelnen Dateien zeigt. Durch die Veränderbarkeit der Muster und durch zusätzliche Definitionen von Zugriffsmethoden auf Containerklassen kann die Analyse an kommende Konventionen und an eigene Anforderungen leicht angepasst werden.

## Kapitel 4

### 4 Technische Realisierung

Die Einbindung des JBCAnalysers erfolgt nach der Konvention für FUJABA Plug-ins [LoWen]. Beim Programmstart wird der Menüeintrag „Import Java Byte Code“ innerhalb der Import/Export-Sektion eingefügt, sowie eine Einstellungsmöglichkeit in die Plug-in Preferences (Einstellungen) integriert. Diese Optionen spielen für die Analyse eine wichtige Rolle und werden in dem folgenden Kapitel genauer erläutert.

Durch Aktivierung des Menüeintrages wird die JBCAnalyserAction ausgelöst, die den Hauptdialog des Plug-ins initialisiert. Dieser dient hauptsächlich zum Auswählen der zu analysierenden Dateien und bietet eine Hilfestellung bei eventuell auftretenden Fehlermeldungen.

#### 4.1 Laden der Klassen mittels des UPBClassLoaders

Der Benutzer kann über den Hauptdialog des JBCAnalyser die zu analysierenden Dateien auswählen. Dabei besteht eine Beschränkung auf kompilierte Klassendateien, Verzeichnisse oder Jar-Archive. Deren Pfade werden in eine Baumstruktur innerhalb des Hauptdialoges überführt. Mittels Checkbutton kann der Benutzer seine getroffene Auswahl verfeinern. Von den selektierten Dateien werden die zugehörigen Pfade in der Klasse JBCPathVault zwischengespeichert.

Nachdem eine Auswahl getroffen wurde, kann die Analyse mittels des Button „Create“ gestartet werden. Der JBCAnalyser benötigt einen ClassLoader zum Erstellen von Objekten der jeweiligen Klassendatei, damit in dem späteren Verlauf die Java-Reflection-API angewendet werden kann. Hierfür wird der FUJABA eigene UPBClassLoader verwendet, um Überschneidungen mit anderen Plug-ins zu vermeiden. Eine Instanz ist über die statische Methode get(String key) erhältlich, wobei der Parameter zur Identifizierung dient.

Die Action-Methode des Hauptdialoges startet die Methode „startAnalyse“ in der Klasse JBCAnalyserBackBone. Wie der Name schon ausdrückt, bildet diese das Rückgrad der Analyse und führt alle weiteren Schritte aus. Zum Laden der einzelnen Klassendateien, wird die Methode startClassLoader() der Klasse JBCClassLoader ausgeführt. Diese beinhaltet eine Instanz des UPBClassLoaders. Die Pfade zu den ausgewählten Dateien erhält der JBCClassLoader über eine statische Methode der Klasse JBCPathVault und werden dem ClassPath hinzugefügt. Anschließend kann versucht werden von den Dateien Objekte vom Typ „Class“ zu erzeugen.

Ein Problem entstand beim Laden von Klassenpfaden, da diese durch den UPBClassLoader nur durch Angabe des kompletten Packagepfades mit dem Dateinamen erreichbar sind. Bei Jar-Archiven gestaltete sich die Lösung einfacher, da nur der Pfad zur Archivdatei dem ClassPath hinzugefügt werden musste und der UPBClassLoader die Funktionalität besitzt, diese zu öffnen, um selbstständig an die inneren Dateien zu gelangen. Daher musste eine Unterscheidung getroffen werden. Der Inhalt von Jar-Archiven ist, wie beschrieben somit einfach zu erreichen. Bei Klassendatei wird der komplette Pfad unter der Verwendung des systemeigenen Path-Separator getrennt. Diese Stringstücke werden innerhalb einer Schleife

dem ClassPath des UPBClassLoaders hinzugefügt, wobei vom ersten Element aus, diese wieder zusammengesetzt werden.

Beispiel zur Erweiterung des ClassPath anhand der Klasse JBCAnalyser mit dem Packagepfad „de.upb.jbcanalyser“:

- 1) C:\
- 2) C:\de
- 3) C:\de\upb
- 4) C:\de\upb\jbcanalyser
- 5) C:\de\upb\jbcanalyser\JBCAnalyser.class

Nachdem für eine Datei durch dieses Verfahren, der ClassPath erweitert wurde, versucht der JBCClassLoader mit dem UBPClassLoader die Datei zu laden. Da hierfür die Angabe des Packagepfades nötig ist, wird innerhalb einer weiteren Schleife versucht, eine Instanz des Class-Objektes zu erzeugen. Daher wird vom Klassendateinamen aus durch eine Erweiterung der vorherigen Verzeichnisse anhand der Stringstücke der korrekte Packagepfad ermittelt.

Für das oben genannte Beispiel würde dieses Vorgehen wie folgt aussehen:

- 1) JBCAnalyser -> Fehlschlag
- 2) jbcanalyser.JBCAnalyser -> Fehlschlag
- 3) upb.jbcanalyser.JBCAnalyser -> Fehlschlag
- 4) de.upb.jbcanalyser.JBCAnalyser -> Packageangabe korrekt

Bei diesem Vorgehen kann ein weiteres Problem auftreten. Dieses tritt aber nur selten auf, falls zwei unterschiedliche Dateien den selben Namen und das selbe Package (oder beide keine Packageangabe) haben, jedoch die Verzeichnisstruktur verschieden ist z.B.:

Datei 1:

C:\NewVersion\de\upb\jbcanalyser\JBCAnalyser.class mit Package de.upb.jbcanalyser

Datei 2:

C:\OldVersion\de\upb\jbcanalyser\JBCAnalyser.class mit Package de.upb.jbcanalyser

Sollen beide Dateien gleichzeitig analysiert werden, kann der ClassLoader beim Erstellen der Objekte nicht unterscheiden, welche der beiden er verwenden soll. Daher kann es zu Fehlern in dem erstellten Klassendiagramm kommen. Der ClassLoader würde beim Laden der ersten Datei korrekt vorgehen und das richtige Objekt erstellen. Bei der zweiten jedoch würde das Problem entstehen, dass er die ersten wieder lädt, da von dieser der Pfad als erstes im ClassPath steht und er die Einträge nicht unterscheiden kann. Da der JBCAnalyser dieses Problem nicht lösen kann, sollte dies vor der Analyse von Dateien berücksichtigt werden.

Als Zwischenspeicher für alle erstellten Class-Objekte dient ein Vektor, der anschließend an den JBCAnalyserBackBone zurückgegeben wird, um die weiteren Analysen durchführen zu können.

## 4.2 Erstellen des Klassendiagramms

Die Informationen, die durch die Java-Reflection-API erhalten wurden, dienen zum Erstellen eines eigenen Modells aus dem Meta-Modell von FUJABA. Hierfür steht eine Anzahl von Klassen innerhalb von FUJABA zur Verfügung. Die für diese Analyse benötigten Dateien befinden sich im Package „de.upb.fujaba.uml“ und beginnen mit „UML“ vor dem eigentlichen Namen. Das so erstellte logische Modell kann in FUJABA grafisch als Klassendiagramm dargestellt werden.

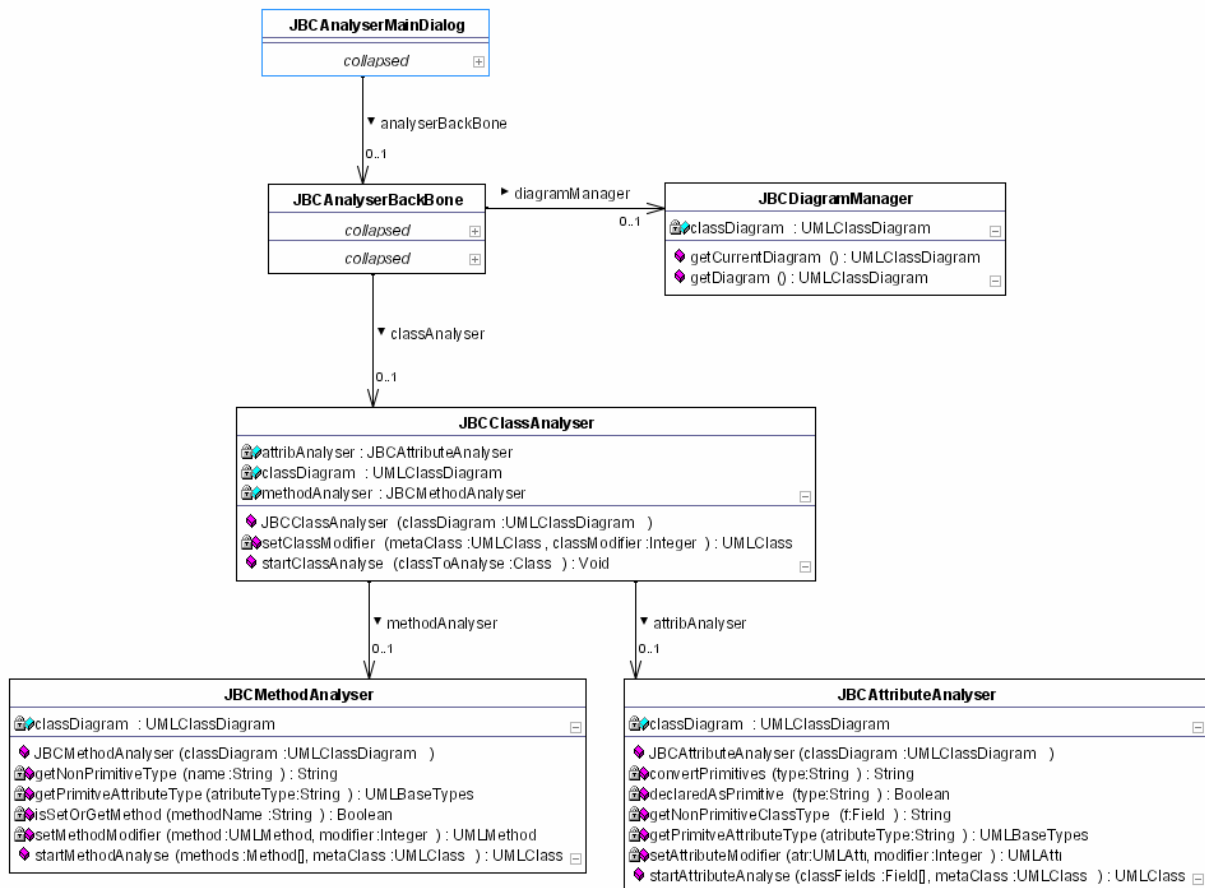


Abbildung 11) Für die Analyse von Class-Objekten zuständige Dateien

Wie schon zuvor leitet die Klasse JBCAnalyserBackBone alle nötigen Schritte der weiteren Analyse ein. Bevor die eigentliche Untersuchung der Class-Objekte beginnt, benötigt diese erst eine Referenz auf das aktuelle Klassendiagramm innerhalb von FUJABA. Hierfür dient die Klasse JBCDiagramManager, die je nach den gesetzten Einstellungen innerhalb des Einstellungsmenü für Plug-ins das aktuelle oder ein neu erstelltes Diagramm zurückliefert. Die Klassen innerhalb der Abbildung 11 sind für die Analyse der Class-Objekte (mittels der Java-Reflection-API) zuständig und für das Erstellen des Klassendiagramms in FUJABA.

### 4.2.1 Analyse der Class-Objekte

Da nun eine Referenz auf das zu verwendende Klassendiagramm besteht und von den gewählten kompilierten Dateien jeweils ein Objekt vom Typ „Class“ besteht, kann mit der eigentlichen Untersuchung begonnen werden. FUJABA stellt für den Aufbau des Meta-Modells eigene Klassen zur Verfügung, in die die gewonnenen Daten überführt werden müssen, um sie als Klassendiagramm anzuzeigen. Dieser Aufbau erfolgt nach dem Baukastensystem. Innerhalb der Klasse JBCClassAnalyser werden unter der Verwendung der Java-Reflection-API folgende Werte ermittelt:

- Typ des Objektes
- ein eventuell gesetzter Packagepfad
- Oberklasse
- implementierte Interfaces
- ob es sich hierbei evtl. um ein Interface handelt
- Sichtbarkeit (Abstrakt oder Public)
- deklarierten Attribute als ein Array vom Typ „Field“
- Methodenköpfe als Array vom Typ „Method“

Der JBCClassAnalyser erstellt für jedes untersuchte Class-Objekt ein eigenes Objekt vom Typ „UMLClass“. Diesen können entsprechende Werte hinzugefügt werden, die auch anschließend im Klassendiagramm wieder zu finden sind:

- Typ
- Packagepfad
- Sichtbarkeit des Objektes

Zusätzlich wird ein Objekt vom Typ „UMLStereoType“ hinzugefügt, das bei der Darstellung für eine Unterscheidung sorgt, ob es sich um ein Interface oder ein eigenständiges Objekt handelt. Oberklassen und implementierte Interfaces werden ebenfalls in das Diagramm mit eingefügt, aber nicht weiter analysiert, falls sie nicht zu der getroffenen Auswahl an Klassendateien gehören. Zwischen ihnen und der untersuchten Klasse wird eine Verbindung vom Typ „UMLGeneralization“ erstellt, um die Beziehung auch grafisch wiederzugeben. Die Suche nach allen Oberklassen kann, falls erwünscht und in den Einstellungen des JBCAnalyser der entsprechende Checkbutton gesetzt wurde, erweitert werden. Hierbei versucht der JBCClassAnalyser alle Oberklassen bis zum Typ „Object“ herauszufinden.

Ein Problem entsteht bei der Analyse von Klassen, die eine Referenz zu einer anderen besitzen und diese nicht erreichbar ist. Dieses Problem tritt auf, falls kein entsprechender Eintrag im ClassPath des UPBClassLoaders vorhanden ist. Der JBCAnalyser gibt eine entsprechende Fehlermeldung aus mit der Information, welches Objekt nicht erreichbar war. Der Benutzer hat nun die Möglichkeit, innerhalb der Einstellungen des JBCAnalyser den Pfad zu der fehlenden Klassendatei anzugeben. Bei der wiederholten Analyse wird dieser vorher dem ClassPath hinzugefügt und kann somit ohne Fehler beendet werden.

Für die Analyse der Attribute und Methoden existieren eigene Klassen, die vom JBCClassAnalyser gestartet werden.

## 4.2.2 Analyse der Attribute

Für die Analyse der Attribute ist die Klasse „JBCAttributeAnalyser“ zuständig. Diese erhält bei der Initialisierung eine Referenz auf das aktuelle Klassendiagramm und beim Aufruf der Methode „startAttributeAnalyse“ die UMLClass der untersuchten Klassendatei, sowie ein Array mit allen ausgelesenen Attributen vom Typ „Field“. Diese werden untersucht und eine Unterscheidung getroffen. Handelt es sich bei dem Field-Objekt um ein Array, so wird ein Objekt vom Typ „UMLArray“ erzeugt. Anderenfalls ein Objekt vom Typ „UMLAttr“. Diesen werden folgende ermittelte Werte hinzugefügt:

- Name des Attributes
- Typ des Attributes
- Sichtbarkeit

Da sich das Meta-Modell wie ein Baukastensystem verhält, werden die Objekte der übergebenen UMLClass (von der untersuchten Datei) hinzugefügt. Der Benutzer kann in den Einstellungen des JBCAnalyser festlegen, welche Attribute als primitiv gelten. Alle hier nicht benannten sind somit nicht primitive Attribute und werden entsprechend anders behandelt. Dieses hat Auswirkungen auf das Diagramm und die Assoziationserkennung. Attribute, die nicht als primitiv deklariert sind, werden der zugehörigen UMLClass hinzugefügt, jedoch für den Benutzer in dem Diagramm nicht sichtbar. Diese Beziehung zwischen den Klassen soll später durch eine Assoziation gekennzeichnet werden. Daher wird für sie eine eigene UMLClass mit dem Typ des Attributes erzeugt und in das Diagramm eingefügt.

## 4.2.3 Analyse der Methoden

Nachdem alle Attribute einer Klasse untersucht wurden, wird von dem JBCClassAnalyser durch die Klasse JBCMethodAnalyser die Methodenuntersuchung gestartet. Bei der Initialisierung erhält diese eine Referenz auf das aktuelle Klassendiagramm, sowie beim Aufruf der Methode „startMethodAnalyse“ die Methoden als Array vom Typ „Method“ und die zugehörige UMLClass. Für alle Methoden wird eine eigene Instanz vom Typ „UMLMethod“ gebildet und hinzugefügt, mit dem kompletten Namen der Methode, der Sichtbarkeit zu anderen Klassen und den Typ des Rückgabewertes. Parameter werden dieser als Objekt vom Typ „UMLParam“ oder gegebenenfalls als „UMLArray“ hinzugefügt. Methoden liefern für die spätere Assoziationserkennung einen Hinweis auf die Speicherung von Objekten in Containerklassen. Daher wird von jedem Parameter ein eigenes Objekt vom Typ „UMLClass“ erstellt und ins Klassendiagramm eingefügt, jedoch zunächst ohne Sichtbarkeit für den Benutzer.

In den Einstellungen des JBCAnalyser können unter dem Eintrag SET/GET-Methods Methoden definiert werden, die zum Speichern oder Auslesen von Objekten in und aus Containerklassen dienen. Alle Methoden einer Klasse werden mit den hier gesetzten Werten mittels Patternmatching verglichen. Bei zutreffenden Namen wird die Methode innerhalb des Klassendiagramms als für den Benutzer nicht sichtbar gesetzt.

Nachdem die Methodenanalyse für jede Klasse abgeschlossen ist, wurde ein komplettes Klassendiagramm erstellt. Um weitere Aussagen über die Verbindungen zueinander zu treffen, wird im Anschluss die Assoziationsanalyse gestartet.

### 4.3 Muster für die Assoziationserkennung

Für die Analyse soll das schon vorhandene Plug-in „InferenceEngine“ genutzt werden, das ausschließlich auf dem Klassendiagramm und somit auf dem darunter liegenden erstellten Meta-Modell von FUJABA arbeitet. Die Erkennung von den Assoziationen basiert hierbei auf vorher definierten Mustern. Diese können mittels des Plug-in „PreferenceEngine“ erstellt werden. Für eine standardisierte Suche existiert für den JBCAnalyser ein Standardmusterkatalog, der einen Großteil aller Assoziationen abdeckt. Eine Auflistung der einzelnen Regeln befindet sich im Anhang. Die InferenceEngine erweitert das Diagramm beim Auffinden eines Musters durch Objekte vom Typ „UMLAnnotation“, die im späteren Verlauf durch Assoziationen ersetzt werden sollen. Jedoch müssen spezielle Konventionen bei den Regeln eingehalten werden, um ein späteres Auslesen der nötigen Werte zu gewährleisten, da dieses durch den selben Namen der Verbindungen (annotatedElement) geschieht, die im Muster vergeben wurden.

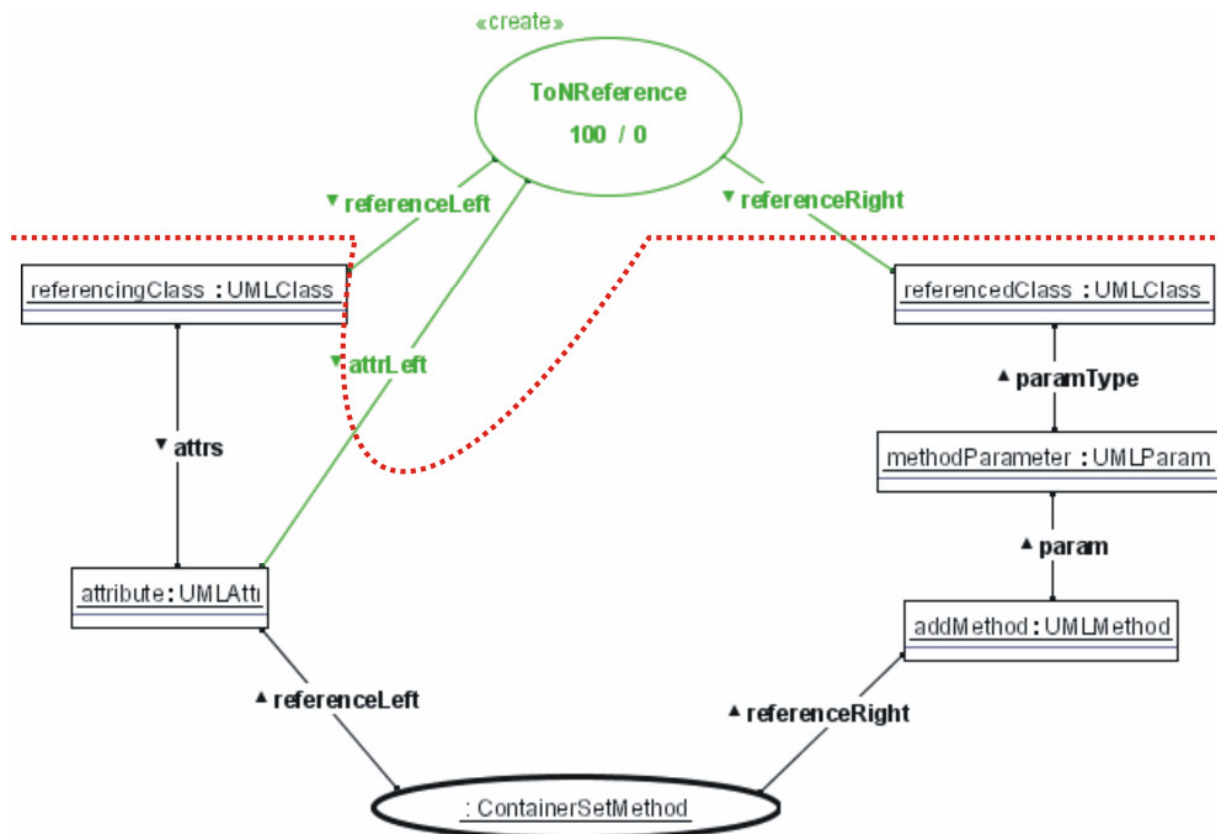


Abbildung 12) Beispiel einer Musterregel

Wie schon im Konzept erwähnt wurde, müssen für spezielle Elemente der Muster Konventionen eingehalten werden, um die Flexibilität zu gewährleisten. In Abbildung 12 besteht diese für den Namen der Regel (Oval mit <<create>>) und dessen Verbindungen zu den ASG-Elementen (referenceLeft, referenceRight, attrLeft). Diese dienen bei der Assoziationserstellung, um eine Referenz auf die Objekte innerhalb des Klassendiagramms zu erhalten, deren Daten benötigt werden. Die Elemente oberhalb der gestrichelten Linie bilden später die UMLAnnotation, die bei Übereinstimmung dieses Musters in das Klassendiagramm eingefügt wird. Alle anderen unterhalb der Linie bilden die Struktur, mit der die Struktur beschrieben wird, wie eine Assoziation in dem logischen Modell des Klassendiagramms

auszusehen hat. Der Trigger ist hier gleichzeitig ein anderes Muster (ContainerSetMethod im Oval mit dicker Umrandung), das in diesem zusätzlich verwendet wird. Das Muster ContainerSetMethod (Muster im Anhang dargestellt) beschreibt die Struktur von Containerklassen und deren Zugriffsmethoden. Dieses und das hier beschriebene Muster müssen übereinstimmen, damit die ToNReference erkannt und eine entsprechende Annotation eingefügt wird.

### 4.3.1 Konventionen für den Standardmusterkatalog

Für die beiden UMLClass-Objekte, zwischen denen bei Übereinstimmung des Musters eine Assoziation eingezeichnet werden soll, müssen Verbindungen (annotatedElement) mit den folgenden Namen verwendet werden:

- referenceLeft
- refernceRight

Für jede dieser Klassen kann zusätzlich noch ein weiteres Element annotiert werden, vom Typ UMLAttr, UMLMethod oder UMLParam. Anhand dieser Elemente werden die Namen für die Assoziation ermittelt. Hierfür stehen für die jeweilige Seite und Objekt folgende Namen für die Verbindungen zur Verfügung:

- attrLeft
- methodLeft
- paraLeft
- attrRight
- methodRight
- paraRight

Der Standardmusterkatalog enthält einige Regeln, die nicht alleine durch ihre Struktur beschrieben werden konnten. Es besteht aber die Möglichkeit, dass zusätzlich zum Vergleich des Musters durch Methodenaufrufe die Regel erweitert wird. Hierfür stehen innerhalb der Klasse JBCPatternMethods zwei statische Methoden zur Verfügung, die zusätzlich zum Muster weitere Vergleiche durchführen. Der Rückgabewert muss dabei immer vom Typ „boolean“ sein.

- isContainerClass(UMLClass)  
Diese Funktion vergleicht den Namen der übergebenen UMLClass mit den in den Einstellungen unter „Container Classes“ gesetzten Werten. Mit dieser Methode werden Klassen identifiziert, die vom Benutzer als Container gesetzt wurden und zum Speichern von Objekten anderen Typs dienen können und somit einen Hinweis auf Assoziationen mit höheren Kardinalität geben.
- isSetMethod(UMLMethod, UMLAttr)  
Diese Funktion vergleicht den Methodennamen mit denen in den Einstellungen unter SET/GET-Methods gesetzten Werten mittels Patternmatching. Hierbei werden alle als SET-Method gesetzten Einträge verwendet, der Platzhalten mit dem Namen des übergebenen Objektes UMLAttr ersetzt und mit den Methodennamen der UMLMethod verglichen. Diese Funktion dient zum Auffinden von Methoden, die einer Containerklasse Objekte hinzufügen.

Bei der Erstellung der Muster wurde weitestgehend auf eine Wiederverwendung von bereits erstellten Regeln als Teil einer anderen verzichtet, damit der Benutzer die Möglichkeit hat Änderung möglichst einfach vorzunehmen. Die Kardinalitäten und die Art der Assoziationen werden für die Standardmuster durch einen Namensvergleich der Annotationen innerhalb der Klasse JBCAssociationAnalyser ermittelt. Für benutzerdefinierte Muster gelten weitere Konventionen.

### 4.3.2 Konventionen für benutzerdefinierte Muster

Um das Plug-in flexibel zu gestalten, besteht die Option für den Benutzer eigene Regeln zu erstellen, falls der beigefügte Katalog des JBCAnalyser nicht ausreicht. Diese unterliegen einer zusätzlichen Konvention für die zu vergebenen Musternamen, da ansonsten die Art der Assoziation und die Kardinalitäten nicht ermittelt werden können:

„Name“\_“Art“\_“Kardinalität für links“\_“Kardinalität für rechts“

Name:

Dieser ist für die Analyse nicht von Bedeutung und kann vom Benutzer frei gewählt werden (für eigene Zwecke).

Art:

Dieser Wert dient zum Erstellen der Assoziationsart und darf nur aus den Strings „association“ oder „reference“ bestehen.

Kardinalität für links:

Legt den Wert für die Kardinalität der linken Seite der Assoziation fest.

Kardinalität für rechts:

Legt den Wert für die Kardinalität der rechten Seite der Assoziation fest.

Die innerhalb des Namens gesetzten „\_“ dienen zum Trennen des Strings in die einzelnen Werte und muss eingehalten werden. Der Teil „Name“ darf optional weggelassen werden, da er für die Analyse keine Relevanz hat. Für den Aufbau des Musters gelten dieselben Konventionen, die unter Kapitel 3.1.1 definiert worden sind.

Der Standardmusterkatalog ist innerhalb des Plug-in JBCAnalyser in kompilierter Form beigefügt und als FUJABA Projekt. Somit kann der Benutzer jederzeit diesen verändern oder erweitern, um ihn an seine Analyse anzupassen.

## 4.4 Erkannte Muster durch Assoziationen ersetzen

Der JBCAnalyseBackBone initialisiert hierfür die Klasse JBCAssociationAnalyser mit einer Referenz auf das aktuelle Klassendiagramm und startet die Analyse mittels der darin enthaltenen Methode „startAssociationAnalyse“. Zunächst wird die InferenceEngine mit dem, in den Einstellungen, angegebenen Musterkataloges initialisiert. Das Plug-in untersucht, wie

in Kapitel 2.3 beschrieben, das Diagramm nach entsprechenden Mustern und führt bei Übereinstimmungen entsprechende Annotationen ein.

Während dessen wartet der JBCAnalyser auf den Abschluss der InferenceEngine und untersucht anschließend alle Elemente im Klassendiagramm nach Objekten vom Typ „UMLAnnotation“. Diese besitzen denselben Namen, der für das entsprechende Muster vergeben wurde.

Für den Aufbau der Assoziationen stehen innerhalb von FUJABA die Klasse UMLAssoc zur Verfügung. In dieser kann ein Name angegeben werden, der innerhalb des Klassendiagramms angezeigt wird. Die UMLAssoc besitzt zwei Seiten denen jeweils ein Objekt vom Typ UMLRole zugewiesen wird. Diesen wiederum werden die Klassen zugewiesen, zwischen denen die Assoziation besteht. Zusätzlich kann in ihnen für die jeweilige Seite die Kardinalität und ein zusätzlicher Name angegeben werden. Gerade bei Assoziationen ist dieses sehr Nützlich, um den Namen beider Elemente aus den Klassen anzuzeigen, zwischen denen die bidirektionale Verbindung besteht (Attribute, Methoden oder Parameter). Bei Referenzen besteht der Unterschied, dass die Verbindung zwischen zwei Objekten nur einseitig ist und wird deshalb speziell durch einen Pfeil visualisiert. Dieses kann innerhalb der UMLRole durch das angeben eines Wertes (Adornment) der entsprechenden Seite gesetzt werden. Die Entscheidung, ob es sich um eine Referenz oder Assoziation handelt, geschieht durch den Annotationsnamen. Die Erstellung dieser Verbindungen geschieht in drei Schritten, wobei jedes Mal das komplette Klassendiagramm durchlaufen wird.

- Im ersten Schritt werden alle UMLAnnotationen untersucht, ob diese Klassen referenzieren, die innerhalb des Diagramms als „nicht sichtbar“ gesetzt wurden (Methodenparameter), um diese für den Benutzer sichtbar zu machen, da eine Assoziation erkannt wurde.
- Im zweiten Schritt werden nach UMLAnnotationen gesucht, die Referenzen markieren, da diese aus Gründen einer Optimierung vor den Assoziationen bearbeitet werden müssen.
- Im dritten Schritt wird nach den Assoziationen gesucht und eine Optimierung durchgeführt, um mehrfache Verbindungen zwischen zwei Klassen zu vermeiden (Problem wird im Folgenden noch erläutert).

Untersuchte und bearbeitete UMLAnnotationen werden anschließend aus dem Diagramm gelöscht, da sie für den Benutzer keine relevanten Informationen enthalten. Die zugehörigen GFRNAnnotations bleiben jedoch erhalten, damit bei der mehrfachen Ausführung des JBCAnalyser diese schon gefundenen Muster durch die InferenceEngine nicht nochmals erkannt werden.

Anhand der Namen der UMLAnnotation wird erkannt, welche Kardinalität an den beiden Seiten vergeben wird und ob es sich um eine Assoziation oder Referenz handelt. Durch die Verbindung zu der zugehörigen GFRNAnnotation kann ermittelt werden, welche Elemente innerhalb einer Klasse mit dem Muster übereinstimmen (Attribute, Methoden oder Parameter). Diese werden verwendet, um die Rollennamen zu ermitteln und der Assoziation hinzuzufügen. Erstellte Assoziationen werden allerdings noch nicht in das Diagramm eingefügt, sondern innerhalb der Klasse JBCAssociationVault zwischengespeichert, um eine Optimierung zu ermöglichen. Dabei wird getrennt zwischen Assoziationen und Referenzen. Die Optimierung ist notwendig, da eine Assoziation auch durch zwei Referenzen dargestellt werden kann.

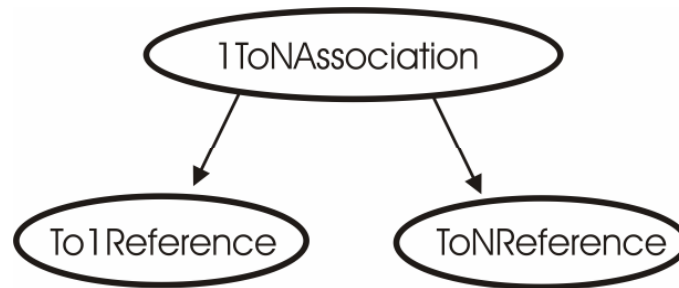


Abbildung 13) Assoziationen definiert durch zwei Referenzen

Durch die InferenceEngine werden dementsprechend zwischen zwei Klassen die drei Annotationen eingefügt. Abbildung 13 zeigt eine solche doppelte Definierung von einer Assoziation und den entsprechenden zwei Referenzen anhand von Mustern aus dem Standardkatalog. Diese redundanten Informationen sollen vermieden werden. Daher müssen zunächst alle Referenzen untersucht und innerhalb des JBCAssociationVaults gespeichert werden. Beim anschließenden Einfügen von Assoziationen wird eine Optimierung gestartet. Diese untersucht alle Referenzen, ob diese zwischen denselben Klassen besteht. Bei zutreffenden wird noch der vergebene Rollenname mit den beiden der Assoziation verglichen und bei Übereinstimmung wird die Referenz gelöscht. Durch dieses Vorgehen bleibt das Diagramm übersichtlich und vermeidet die redundanten Informationen.

Eine zusätzliche Optimierung tritt bei dem Einfügen von Referenzen oder Assoziationen ein, die durch die Erkennung von Containerklassen und deren Zugriffsmethoden entstehen. Häufig ist diese Containerklasse ein nicht primitives Attribut und wird als eigene Klasse dem Diagramm hinzugefügt. Die Mustererkennung findet eine Referenz zwischen der deklarierenden Klasse und der Containerklasse. Da der Schwerpunkt auf der Erkennung des inneren Objekttyps liegt, ist diese Referenz nicht relevant. Daher wird beim Einfügen von Assoziationen oder Referenzen mit höherer Kardinalität zwischen zwei Klassen überprüft, ob diese ebenfalls Verbindungen zu Containerklassen besitzen. Diese Referenz kann gelöscht werden, falls der Rollenname zur Containerklasse identisch mit einem aus der eingefügten Assoziation ist.

Für den Standardmusterkatalog werden alle Informationen über die Kardinalität und den Typ der Assoziation (Assoziation oder Referenz) anhand der Namen innerhalb der UMLAnnotation ermittelt. Diese werden innerhalb der Klasse JBCAssociationAnalyser gesetzt. Da aber eine Möglichkeit besteht, dass ein Benutzer eigene Muster verwenden will, müssen die Informationen über die Kardinalität und den Typ der Assoziation durch einen anderen Weg ermittelt werden. Durch die im Kapitel 3.1.2 festgelegten Konventionen für die Namensvergebung der benutzerdefinierten Muster wird dieses gewährleistet. Der vergebene Name wird mittels des Trenners „\_“ in seine jeweiligen Werte aufgeteilt.

„Name“\_“Art“\_“Kardinalität für links“\_“Kardinalität für rechts“

1. „Name“: Dieser Wert spielt für die Analyse keine Rolle und muss nur optional gesetzt werden.
2. „Art“: Legt fest, ob es sich um eine Assoziation oder Referenz handelt. Durch einen Stringvergleich wird der gesetzte Wert ermittelt.
3. „Kardinalität für links“: Dieser Wert wird direkt übernommen und in die innerhalb der UMLRole der rechten Seite der Assoziation gesetzt.
4. „Kardinalität für rechts“: Dieser Wert wird direkt übernommen und in die innerhalb der UMLRole der linken Seite der Assoziation gesetzt.

Die weiterführende Analyse der benutzerdefinierten Muster entspricht der des Standardkataloges.

Nachdem alle UMLAnnotationen untersucht und gelöscht wurden, werden die Assoziationen und Referenzen in das Klassendiagramm eingefügt. Um zu gewährleisten, dass die aktuelle Anzeige auf dem Bildschirm alle eingefügten Elemente beinhaltet, wird die Darstellung von FUJABA nochmals aktualisiert. Der JBCAnalyser hat nun ein komplettes Klassendiagramm erstellt und möglichst alle Assoziationen und Referenzen untereinander ermittelt. Daher beendet er sich selbstständig.



## Kapitel 5

### 5 Beispiel einer Java Byte Code Analyse

Anhand eines Beispiels soll der Umgang mit dem Plug-in genauer erklärt werden. Dieses soll anhand von Klassendateien geschehen, die ein Ausschnitt einer Universität darstellen. Das Beispiel veranschaulicht fast alle Analysen, die der JBCAnalyser bereithält. Erkennung von einfachen Referenzen durch deklarierte Attribute, implementierte Interfaces, Objekttypen innerhalb von Containerklassen.

Vor der eigentlichen Analyse müssen die Einstellungen für den JBCAnalyser getroffen werden, um nach entsprechenden Methoden zu suchen und die Darstellung abzustimmen. Eine genauere Beschreibung dazu befindet sich im Kapitel 5.3.

#### 5.1 Auswahl der zu analysierenden Dateien

Innerhalb von FUJABA wird der JBCAnalyser in dem Menüeintrag Import/Export unter „Import Java Byte Code“ gestartet und es erscheint der Hauptdialog. Dieser ist für die Auswahl der Dateien zuständig und gibt eine Hilfestellung bei eventuell auftretenden Fehlermeldungen. Mittels des Menüeintrages „Open Files“ oder dem zugehörigen Iconbutton kann die Auswahl durchgeführt werden, wobei nur Klassendateien, Jar-Archive oder Verzeichnisse erlaubt sind. Die Auswahl wird dann als Baum auf der linken Seite des Dialoges angezeigt und kann mittels der Checkbuttons verfeinert werden. Die einzelnen Äste lassen sich hierbei auf- bzw. zuklappen. Die rechte Seite des Dialoges zeigt einige Informationen über den kompletten Pfad und die Anzahl der Dateien. Durch den Button „Create“ startet die eigentliche Analyse. Unter dem Menüeintrag „Help“ ist eine Hilfestellung zu finden, die eventuell auftretende Fehler kurz erläutert und Informationen zu deren Lösung bereithält. Auch die Funktionen der einzelnen Buttons sind hier genauer beschrieben.

Falls eine falsche Auswahl getroffen wurde, kann die Anzeige mittels des Buttons „Discard“ gelöscht werden.

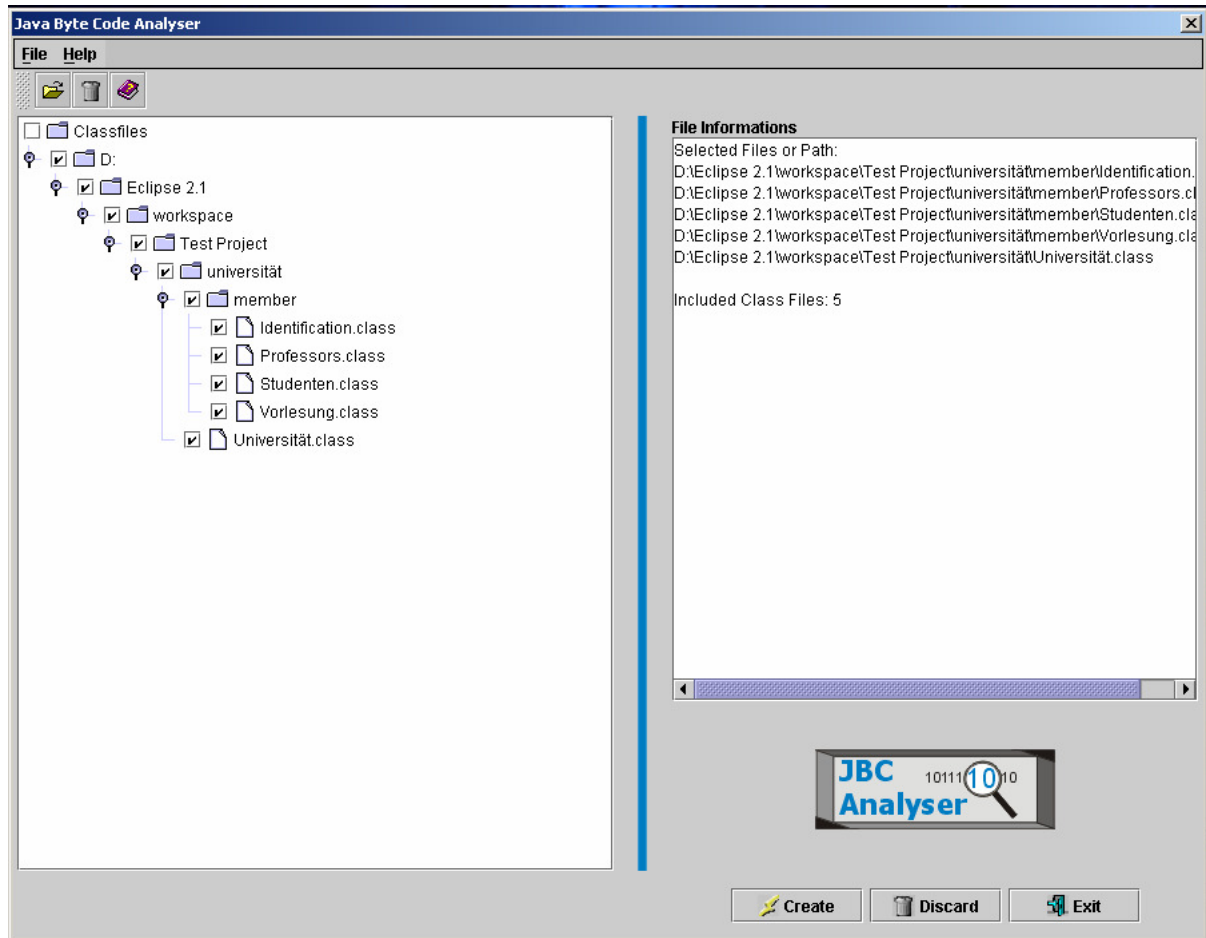


Abbildung 14) Beispiel des Auswahldialoges des JBCAnalyser

Abbildung 14 zeigt den Hauptdialog des JBCAnalyser. In dem linken Feld mit der Baumdarstellung kann die getroffene Auswahl verfeinert werden. Nur ausgewählte Dateien (mittels des Checkbuttons) werden in die Analyse mit einbezogen. Das rechte Textfeld zeigt zusätzliche Informationen über die Pfade und die Anzahl der Dateien. Der Create-Button startet die eigentliche Analyse.

## 5.2 Ergebnis der Analyse

Nachdem der Button „Create“ betätigt wurde, analysiert der JBCAnalyser die Daten und startet das Plug-in InferenceEngine mit dem Standardmuskatalog. Dieses gibt einen Statusbildschirm aus, der die gefundenen Regeln anzeigt und vom Benutzer per Hand geschlossen werden muss. Aus dieser Analyse wird das folgende Diagramm erzeugt.

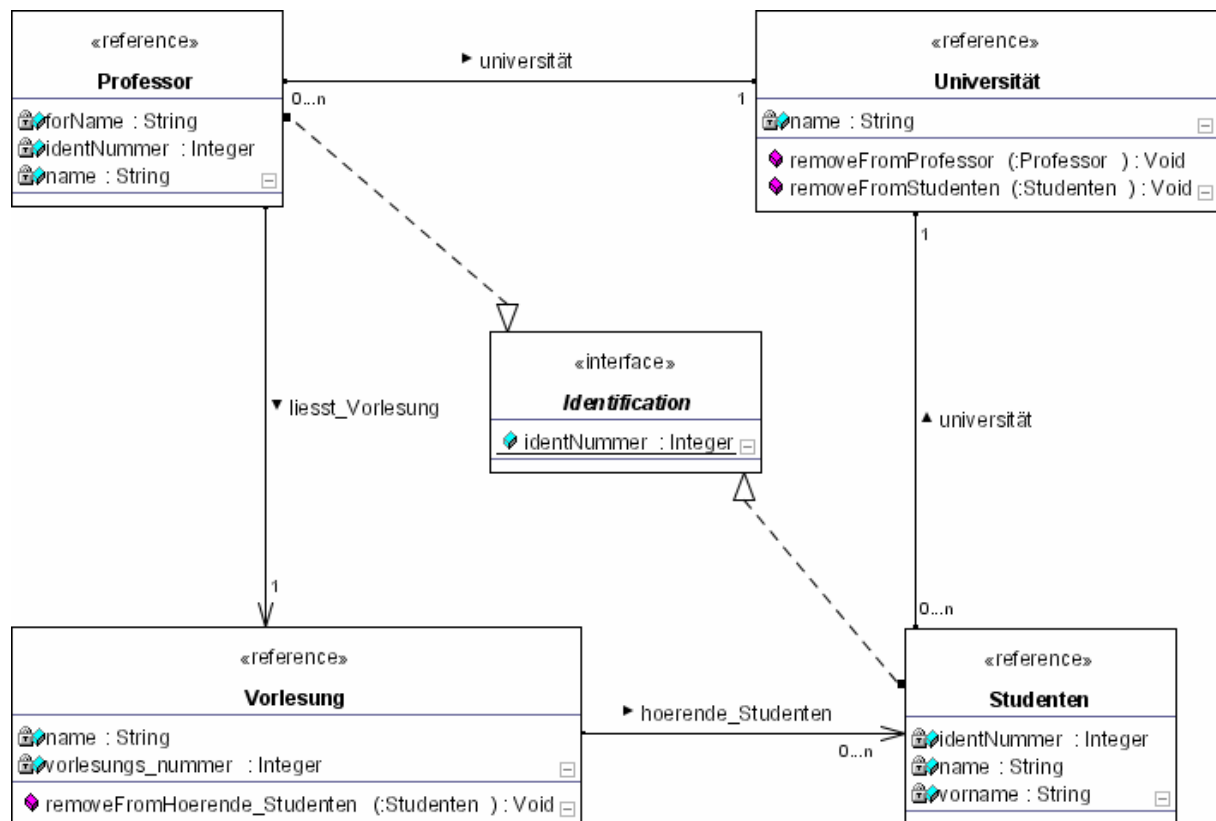


Abbildung 15) Ergebnisklassendiagramm des Beispiels

Die Abbildung 15 zeigt das komplette Klassendiagramm. Alle Assoziationen und Referenzen sind eingezeichnet, wobei doppelte oder zu Containerklassen durch die Optimierung (Kapitel 4.6) weggelassen wurden. Implementierte Interfaces sind durch entsprechende Verbindungen gesondert gekennzeichnet. Die Kardinalitäten sind vorhanden und signalisieren die Häufigkeit der Verbindung. Der Name für die Assoziationen ergibt sich häufig aus dem auslösenden Element (z.B. To1Reference bei Attributen → Assoziation erhält Attributnamen). Aus diesem Klassendiagramm sind nun alle wichtigen Elemente ablesbar und die Zusammenhänge veranschaulicht. Unwichtige Attribute oder Zugriffsmethoden wurden ausgeblendet, da diese Abhängigkeiten schon durch die Referenzen oder Assoziationen gekennzeichnet wurden. Durch entsprechende Einstellungen für den JBCAnalyser lässt sich dieses Ausblenden deaktivieren.

## 5.3 Einstellungsmöglichkeiten des JBCAnalyser

### 1. Registerkarte „Primitive Classes“

Wie der Name schon ausdrückt, werden hier die primitiven Klassen deklariert. Eine vordefinierte Menge wurde bereits in den JBCAnalyserPreferences festgelegt und kann mittels der „Default“ Option abgerufen werden. Zusätzlich können eigene hinzugefügt, oder Einträge entfernt werden. Bei der Analyse von Dateien werden gefundene Attributnamen mit diesen Einträgen verglichen. Fällt dieser positiv aus, wird dieses Attribut mit seinem Namen und Typ sichtbar in dem Klassendiagramm dargestellt.

Nicht primitive werden dieser Klasse hinzugefügt und sind normalerweise (optional, siehe Registerkarte „Preferences“) nicht sichtbar. Stattdessen wird von ihnen eine eigene UMLClass in das Diagramm eingefügt (mit dem Namen des Attributtyps), um in der späteren Analyse die Assoziationen zu erstellen.

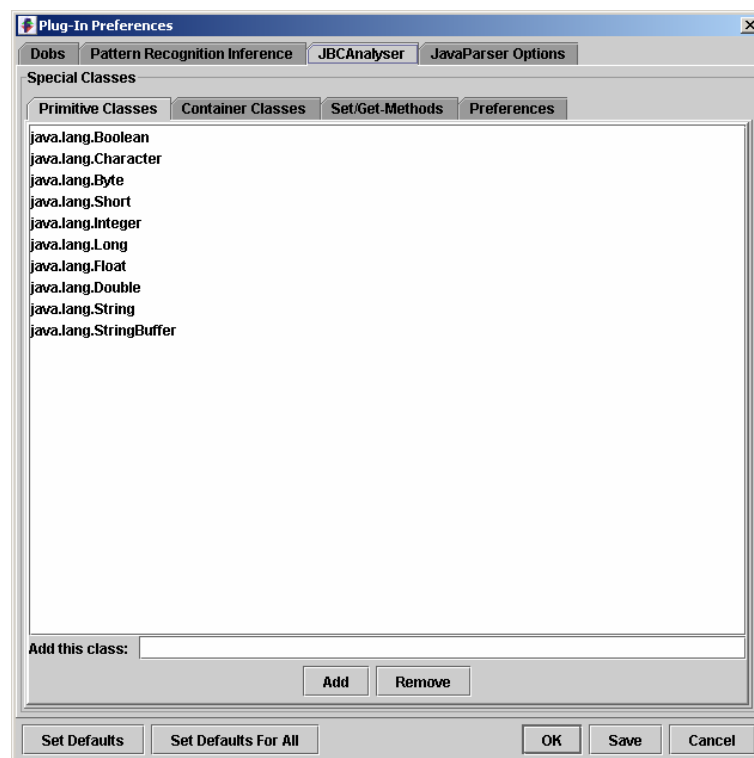


Abbildung 16) Einstellungsmöglichkeiten zum Deklarieren der primitiven Klassen

Der in Abbildung 16 gezeigte Bildschirmausschnitt zeigt den für die Definition von primitiven Klassen zuständigen Karteireiter. Durch die Angabe weiterer Klassen in dem Textfeld „Add this Class“ kann die Liste entsprechend erweitert werden. Durch Markierung von Einträgen und anschließendem Betätigen des Remove-Buttons, können diese wiederum entfernt werden.

## 2. Registerkarte „Container Classes“

Die hierbei gesetzten Einträge definieren Container Klassen, die zum Speichern anderer Klassentypen dienen können. Auch hierbei existiert eine vordefinierte Menge von diesen Klassen, die auch durch den Benutzer veränderbar ist. Während der Analyse werden die Attribute einer Klasse mit den hier gesetzten Einträgen verglichen. Sind beide identisch, muss durch weitere Untersuchungen nach den Zugriffsmethoden gesucht werden, die in diesem Container anderer Objekte speichert.

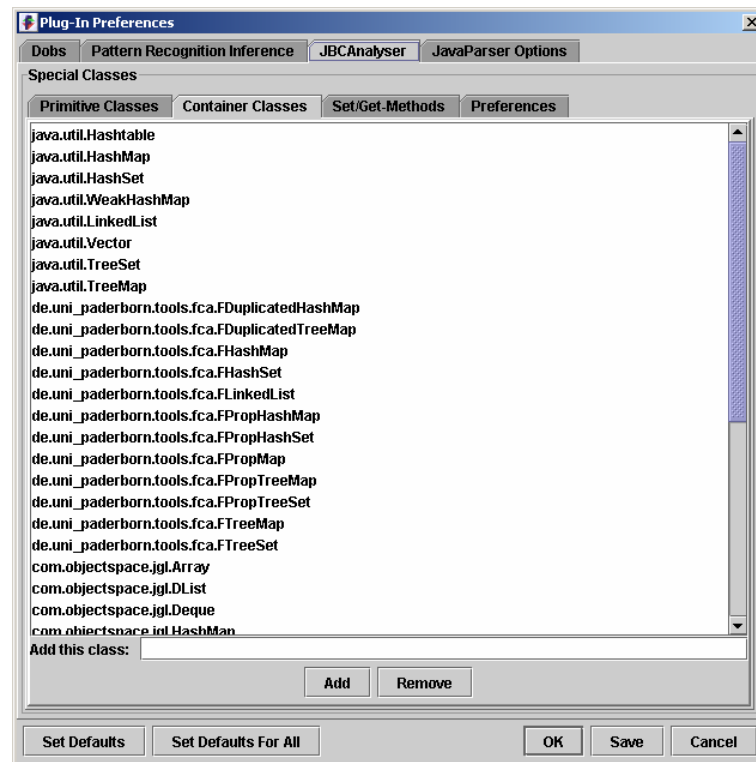


Abbildung 17) Einstellungsmöglichkeiten zum Deklarieren der Container Klassen

Der in Abbildung 17 gezeigte Bildschirmausschnitt zeigt den für die Definition von Container Klassen zuständigen Karteireiter. Durch die Angabe weiterer Klassen in dem Textfeld „Add this Class“ kann die Liste entsprechend erweitert werden. Durch Markierung von Einträgen und anschließendem Betätigen des Remove-Buttons, können diese wiederum entfernt werden.

### 3. Registerkarte „Set/Get-Methods“

Wie schon bei den „Container Classes“ erwähnt wurde, wird nach speziellen Methoden gesucht, die zum Speichern von Objekten innerhalb der Container dienen. Es existiert eine Menge vordefinierter Methodennamen, die der Benutzer an seine Suche anpassen kann. Diese besitzen einen Platzhalter in der Form „ $\$x\$\$$ “, der in der späteren Analyse durch den Attributnamen einer Klasse ersetzt wird.

Der Benutzer muss bei dem Hinzufügen von Methodennamen anhand von Checkbuttons festlegen, ob es sich um eine SET- oder GET-Methode handelt. Alle Methoden, die mit einem der hier gesetzten Einträge übereinstimmen, werden innerhalb des Diagramms normalerweise nicht angezeigt (optional, siehe Registerkarte „Preferences“). Für die Assoziationen sind allerdings nur die SET-Methoden von Relevanz.

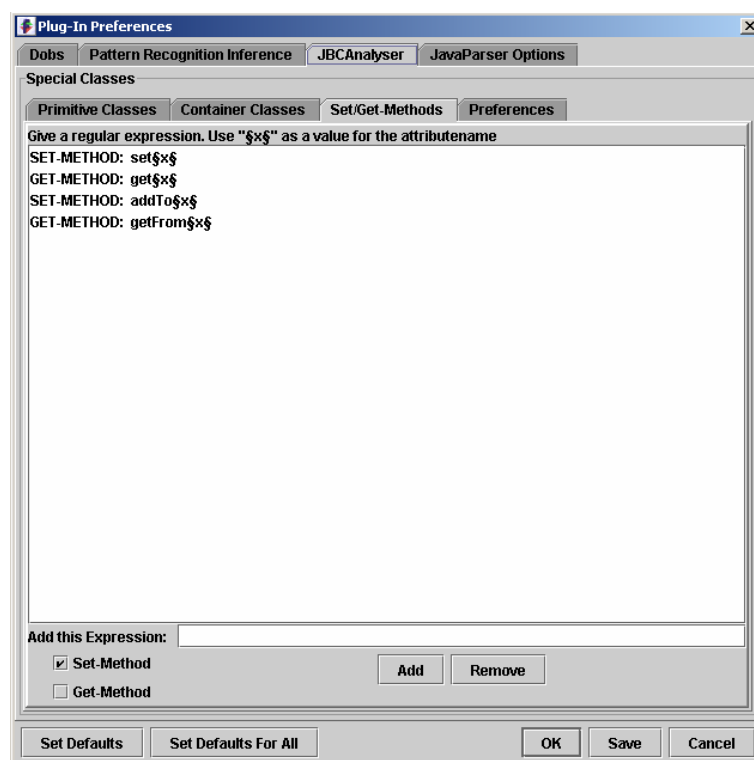


Abbildung 18) Einstellungsmöglichkeiten zum Deklarieren von SET/GET-Methoden

Abbildung 18 zeigt den für die Definition der Zugriffsmethoden zuständigen Karteireiter. In dem Feld „Add this Expression“ kann die Liste entsprechend erweitert werden. Links unten befinden sich die zusätzlichen Checkbuttons, die die Art der Methode festlegen. Durch die Markierung von Einträgen und anschließenden betätigen des Remove-Buttons können diese entfernt werden.

#### 4. Registerkarte „Preferences“

Die ersten vier Checkbuttons dienen zum Einstellen der Darstellung von Klassendiagrammen.

Show all Attributes:

Nach dem Auswählen dieses Checkbuttons werden alle Attribute einer Klasse, einschließlich der nichtprimitiven angezeigt.

Show all Methods:

Nach dem Auswählen dieses Checkbuttons werden alle Methoden, einschließlich der als SET- und GET-Method deklarierten angezeigt. Diese Option hat keine Auswirkungen auf die Analyse der Assoziationen.

Show all Superclasses:

Nach dem Auswählen dieses Checkbuttons wird der JBCAnalyser versuchen, alle Oberklassen herauszufinden und anzuzeigen.

Open always a new Diagram:

Nach dem Auswählen dieses Checkbuttons, wird bei jedem Ausführen des JBCAnalyser ein neues Klassendiagramm innerhalb von FUJABA erzeugt, um eine bessere Übersicht zu gewährleisten.

Den Classpath erweitern:

Es besteht die Möglichkeit, dass während einer Analyse eine Fehlermeldung erscheint, dass bestimmte Attribute oder Methoden nicht erreicht werden konnten. Dieses kann passieren, wenn eine Klasse auf eine andere verweist, die nicht im Classpath des UPBClassLoaders vorhanden ist. Um diesen Fehler zu umgehen, kann hier eine Bibliothek, Verzeichnis oder Klasse von dem nicht erreichbaren Objekt angegeben werden. Vor einer Analyse wird der Pfad dem Classpath hinzugefügt und kann somit bei einer erneuten Analyse vom UPBClassLoader erreicht werden.

Den Muster Katalog festlegen:

Im untersten Bereich besteht die Möglichkeit einen Musterkatalog zu bestimmen. Dieser ist notwendig für die Assoziationsanalyse. Ohne diese Angabe erscheint beim Starten des JBCAnalyser ein Hinweis, dass kein Musterkatalog angegeben wurde und somit auch keine Assoziationserkennung durchgeführt wird. Ein Klassendiagramm kann dennoch erstellt werden.

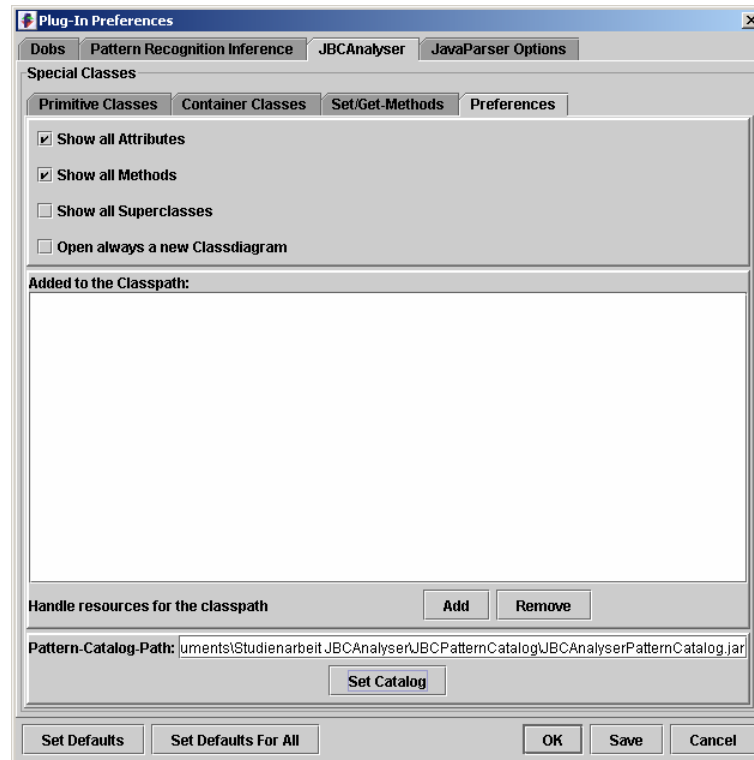


Abbildung 19) Einstellungsmöglichkeiten für die Anzeige, Classpath und Musterkatalog

Abbildung 19 zeigt den Karteireiter, der zusätzliche Einstellungen erlaubt. Die Checkbuttons können die grafische Wiedergabe des Klassendiagramms verändern. In dem mittleren Feld können Klassendateien zum Classpath des ClassLoaders hinzugefügt werden. Der Add-Button öffnet dabei einen FileChoser. Durch die Markierung von Einträgen und anschließenden Betätigen des Remove-Buttons können diese wieder entfernt werden. In dem unteren Feld kann der zu verwendende Musterkatalog definiert werden.

## Kapitel 6

### 6 Zusammenfassung und Ausblick

Die Funktionalität von FUJABA im Bereich des Reverse Engineering war bisher nur auf die Analyse von Dateien im Quellcode beschränkt. Das Plug-in JBCAnalyser erweitert nun diese Funktionalität auf die Analyse von kompilierten Dateien. Das Ergebnis wird auf abstrakte Weise als UML-Klassendiagramm dargestellt und Beziehungen der einzelnen Komponenten untereinander ermittelt.

Um diese Analyse zu realisieren, untersucht der JBCAnalyser Klassen unter Verwendung der Java-Reflection-API. Mittels der ermittelten Informationen wird ein Modell aus dem abstrakten Syntaxgraph von FUJABA erstellt. Dieses kann anschließend grafisch als Klassendiagramm dem Benutzer dargestellt werden.

Für die Ermittlung der Assoziationen zwischen den einzelnen Klassen wird eine schon vorhandene Mustererkennung genutzt. Mit dem Plug-in PatternSpecification kann ein Musterkatalog erstellt werden, der die Strukturen von Assoziation definiert. Das Plug-in InferenceEngine kann mit diesem Katalog nach entsprechenden Vorkommen in dem erstellten Klassendiagramm suchen und fügt bei Übereinstimmungen zusätzlich Objekte zur Markierung ein. Diese kann der JBCAnalyser anschließend nutzen, um Assoziationen zu erstellen, die innerhalb des Klassendiagramms dem Benutzer visualisiert werden. Eine Besonderheit liegt hier bei der Erkennung und Untersuchung von Containerklassen, die zur Speicherung von Objekten anderen Typs dienen. Deren Zugriffsmethoden mit Parametern dienen zur Erstellung von Assoziationen mit höherer Kardinalität.

Das mit dem JBCAnalyser erstellte Klassendiagramm liefert dem Benutzer einen Einblick in den Aufbau einer Software und mittels der Assoziationserkennung werden die Beziehungen der Komponenten zueinander verdeutlicht. Es hilft beim „Verstehen einer Software“ und spart somit Zeit und Geld beim Reverse Engineering eines Produktes. Durch die leichte Veränderbarkeit der Muster sowie der Containerklassenanalyse ist die Assoziationsuche flexibel gehalten. Sie kann an die eigenen Ansprüche oder Konventionen angepasst werden. Somit können auch zukünftige Konventionen für Zugriffsmethoden auf Containerklassen berücksichtigt werden.

Die Klassendiagramme können weiter verwendet werden, um die Qualität einer Software zu beurteilen. Dabei wird auf Basis des Diagramms nach Design-Pattern gesucht. Diese beschreiben Programmstrukturen, deren Verwendung sich als sehr nützlich und effizient herausgestellt hat. Dementsprechend existieren auch Anti-Patterns, bei deren Auffinden von schlechter Qualität ausgegangen werden kann. Die Suche kann mittels dem schon für die Assoziationsanalyse genutzten Plug-ins InferenceEngine durchgeführt werden unter Verwendung eines entsprechenden Musterkataloges. Durch das Auffinden solcher Design-Patterns und deren Anzahl kann ein Rückschluss auf die Qualität der Software getroffen werden. Dieses kann für Entwickler wichtig sein, falls zum Beispiel eine fremde Bibliothek in das eigene Projekt mit eingebunden werden soll, um einen Qualitätsstandard zu sichern. Des Weiteren bewahrt diese Bewertung einer Software vor Fehleinkäufen. Die Arbeit des fremden Entwicklerteams kann somit überprüft werden und bewahrt vor unnötigen Ausgaben für ein minderwertiges Produkt.

Eine weitere Einsatzmöglichkeit des JBCAnalyser besteht bei der Analyse einer fremden Bibliothek, die in das eigene Projekt mit eingebunden werden soll. Mittels der Informationen aus dem erstellten Klassendiagramm kann die Dokumentation vervollständigt werden. Es bietet Aufschluss über die Schnittstellen und Methoden der in der Bibliothek enthaltenen

Klassen. Bei der Projektentwicklung mit FUJABA können die Klassendiagramme direkt in die eigenen mit eingebunden werden. Die Quellcodegenerierung kann diese berücksichtigen und spart somit Entwicklungszeit.

Der JBCAnalyser kann auch bei Designentscheidungen einer Softwareentwicklung hilfreich sein. Das Plug-in gewährt einen Einblick in ähnliche schon veröffentlichte Software von anderen Entwicklungsteams. Die Ergebnisse können übernommen werden, oder zeigen evtl. schlechte Implementierungen, die in dem eigenen Projekt vermieden werden sollten.

Der JBCAnalyser kann somit in den Bereichen des Forward- und Reverse Engineering eingesetzt werden. Die erstellten Klassendiagramme helfen beim „Verstehen einer Software“ und sparen somit Zeit und Geld. Die Ergebnisse des JBCAnalyser können direkt in eine eigene Entwicklung mit einfließen. Die Einsatzmöglichkeiten von FUJABA im Bereich des Reverse Engineering werden gesteigert, da nun Quell- oder kompilierter Code analysiert werden kann.

## Literaturverzeichnis

- [ChCr90] E.J. Chikofsky, J.H. Cross: Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, 1990
- [FNT98] T.Fischer / J.Niere / L.Torunski: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling, Diplomarbeit, Universität Paderborn, 1998
- [Fra92] A. Frazer. Reverse Engineering - hype, hope or here? In P.A.V. Hall, Hrsg., Software Reuse and Reverse Engineering in Practice, Jgg. 12 of UNICOM Applied Information Technology, Seiten 209-243. Chapman & Hall, 1992
- [Krü02] G. Krüger: Handbuch der Java-Programmierung, Addison-Wesley Verlag 2002, 3. Auflage
- [LoWen] L. Wendehals: 10 Steps to Build a Fujaba Plug-in, University Paderborn, [http://wwwcs.upb.de/cs/fujaba/documents/developer/howtos/plugins/10\\_Steps\\_To\\_Build\\_a\\_Fujaba\\_Plugin.pdf](http://wwwcs.upb.de/cs/fujaba/documents/developer/howtos/plugins/10_Steps_To_Build_a_Fujaba_Plugin.pdf)
- [NSWW02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh, 'Towards Pattern-Based Design Recovery', ACM Press, May 2002
- [ZSG79] M.V. Zelkowitz, A.C. Shaw, J.D. Gannon. Principles of Software Engineering and Design. Prentice-Hall, Inc., Englewood Cliffs, 1979
- [Oe01] Bernd Oestereich: Objektorientierte Softwareentwicklung, Oldenbourg Verlag 2001, 5. Auflage
- [Wen01] L. Wendehals: 'Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets', Master's thesis, University of Paderborn, Germany, October 2001

## Anhang

### Standardmusterkatalog des JBCAnalyser:

#### Muster für Referenzen:

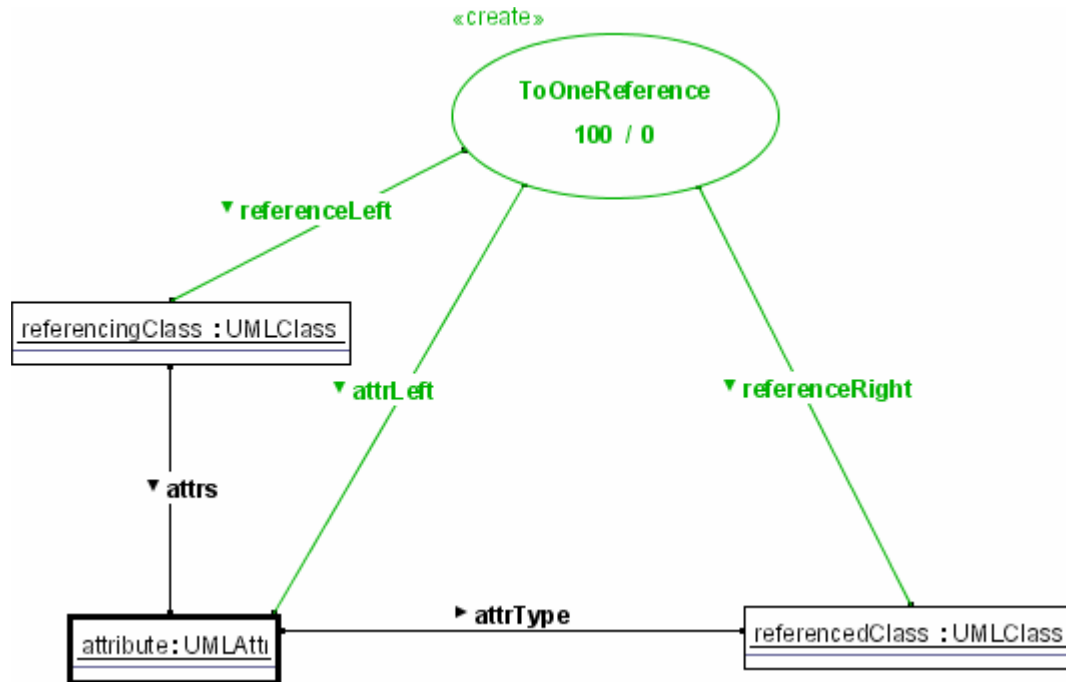


Abbildung 20) Muster für Referenz mit Kardinalität "1"

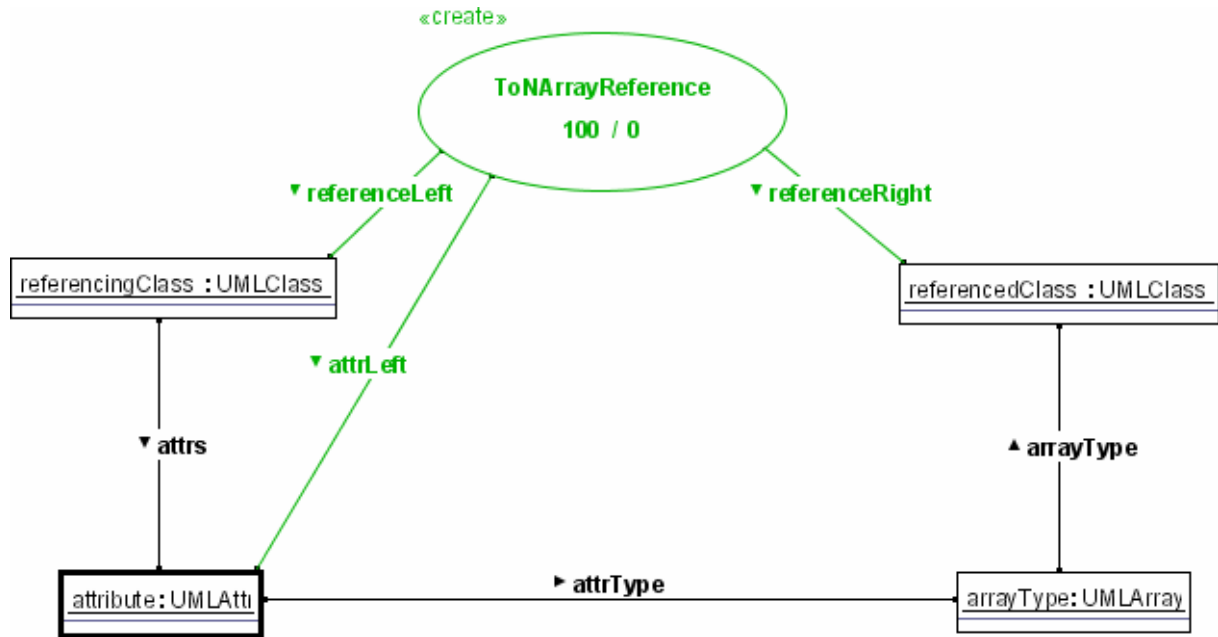


Abbildung 21) Muster für Referenz mit Kardinalität "n" durch Array

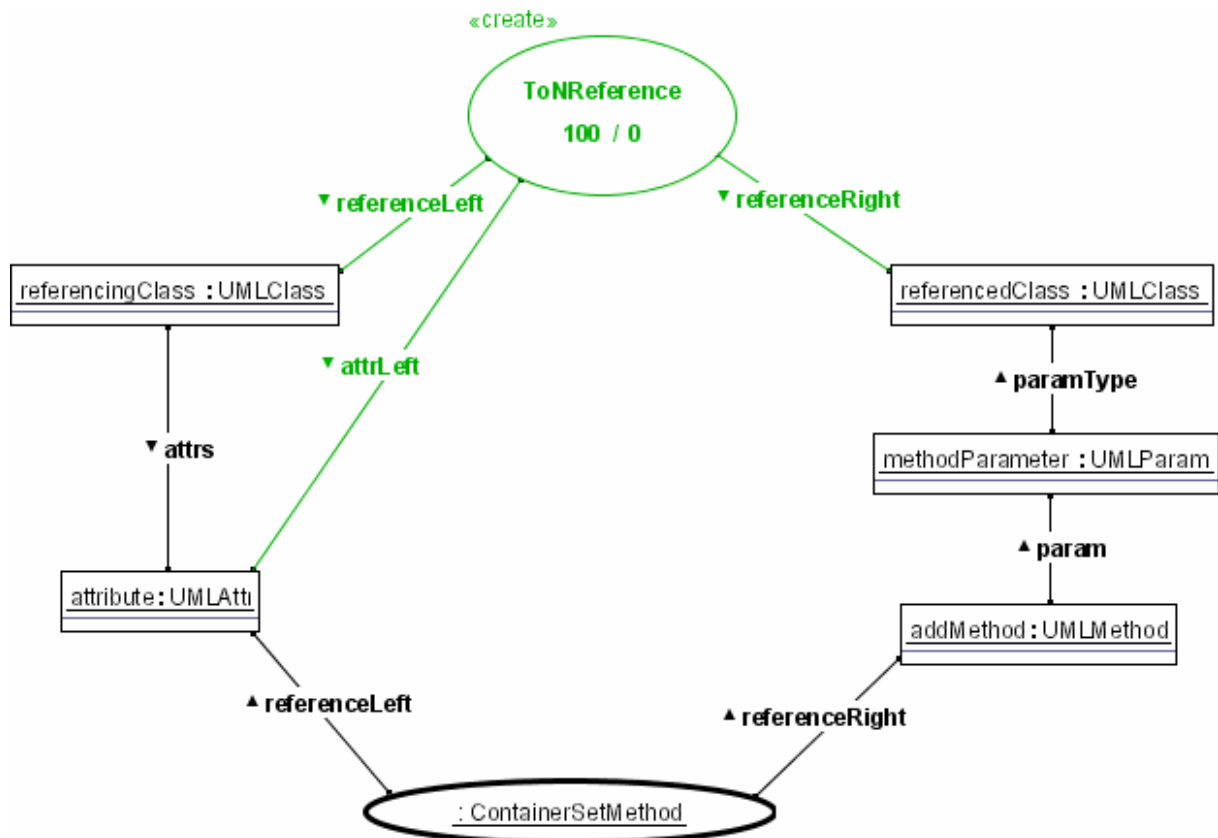
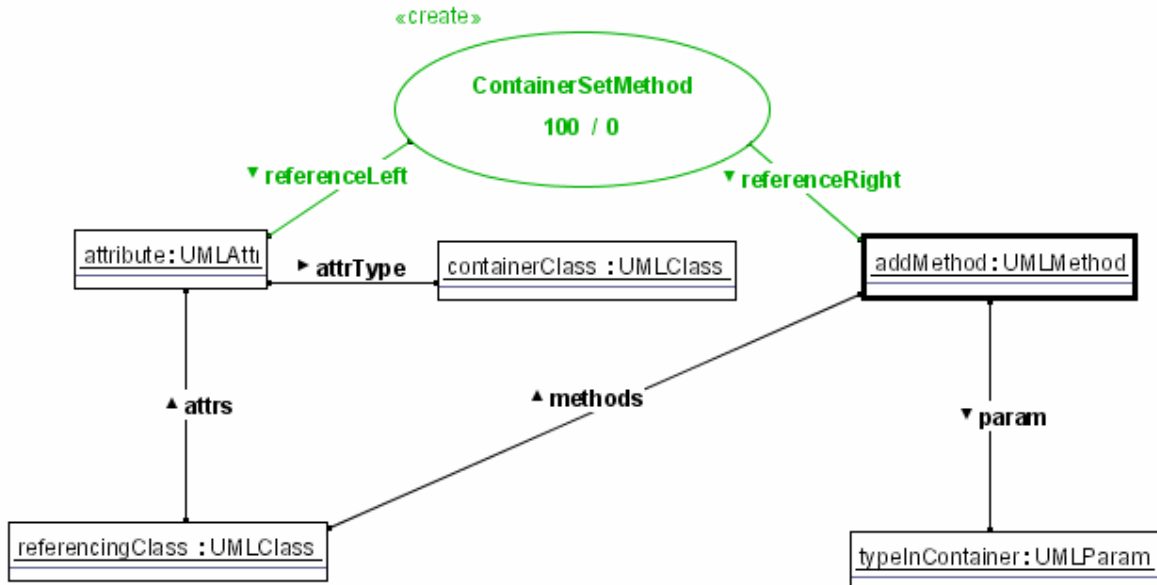


Abbildung 22) Muster für Referenz mit Kardinalität "n" durch Containerklasse

**Muster für die Erkennung der Set-Methoden zum Speichern innerhalb von Containerklassen:**



```

{ IsAddMethod : de.upb.jbcanalyser.data.JBCPatternMethods.isSetMethod(addMethod, attribute) }
{ IsContainerClass : de.upb.jbcanalyser.data.JBCPatternMethods.isContainerClass(containerClass) }
    
```

Abbildung 23) Muster für die Erkennung der Set-Methoden

**Muster für Assoziationen:**

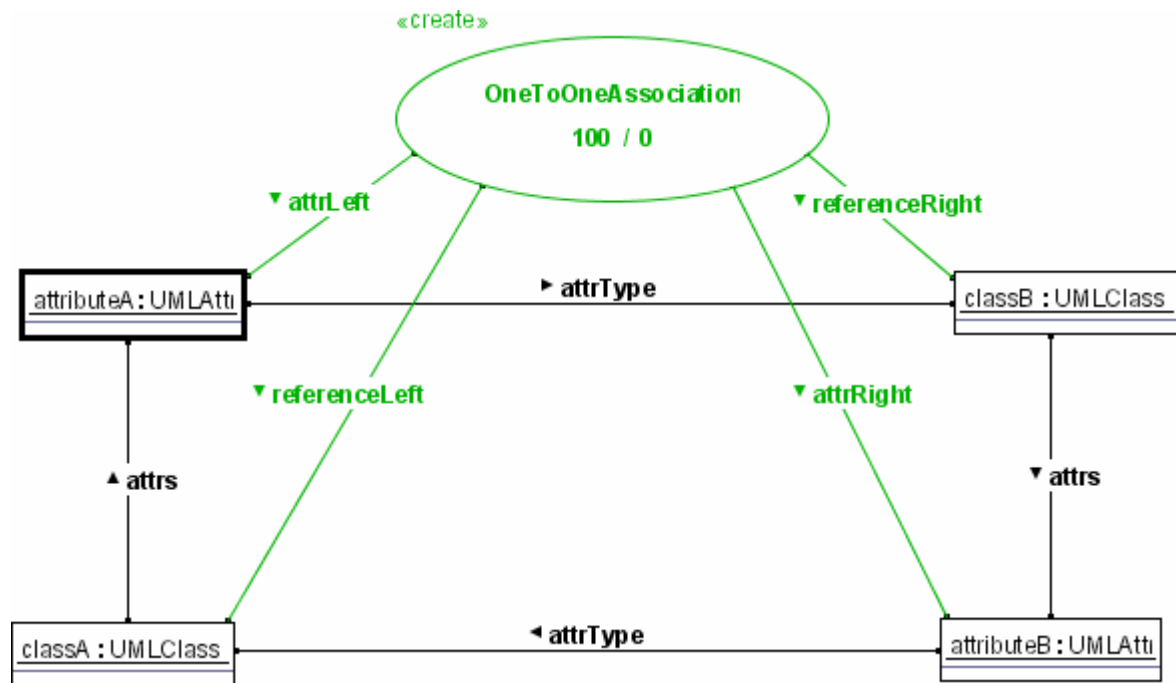


Abbildung 24) Muster für Assoziation mit Kardinalität "1" zu "1"

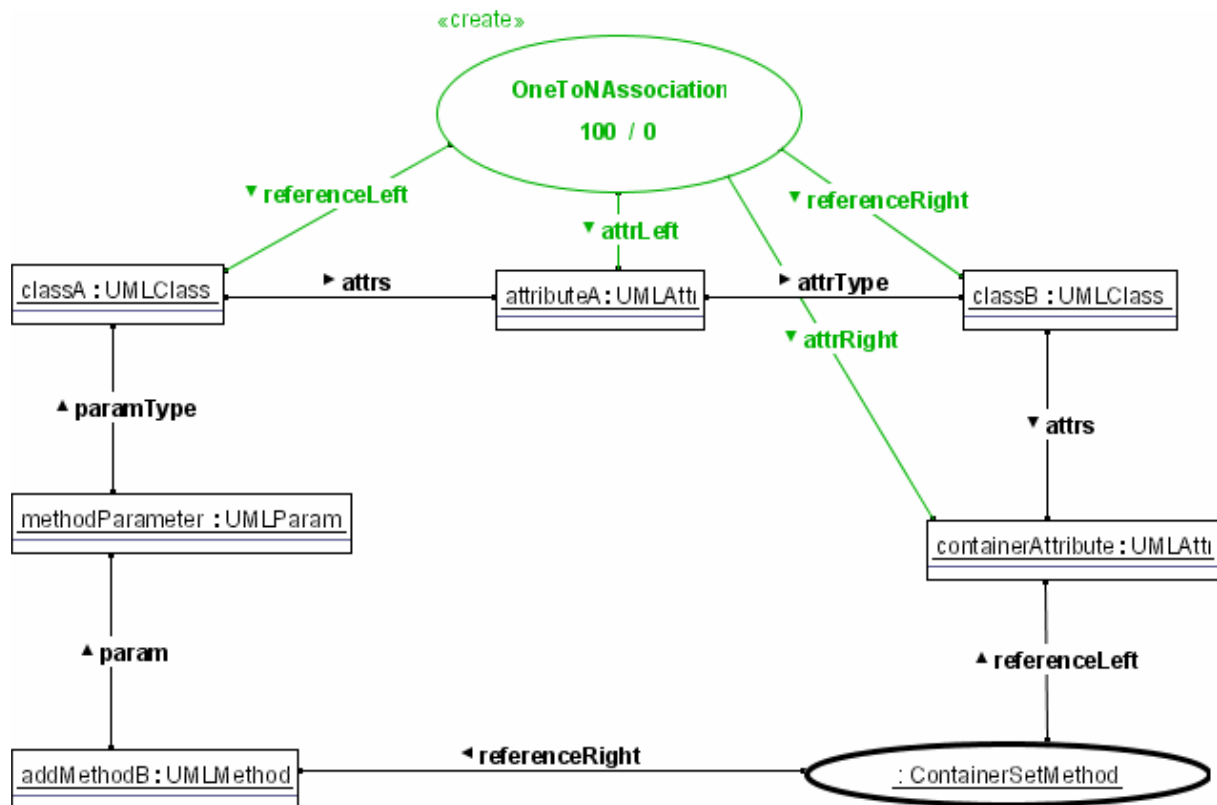


Abbildung 25) Muster für Assoziation mit Kardinalität "1" zu "n"

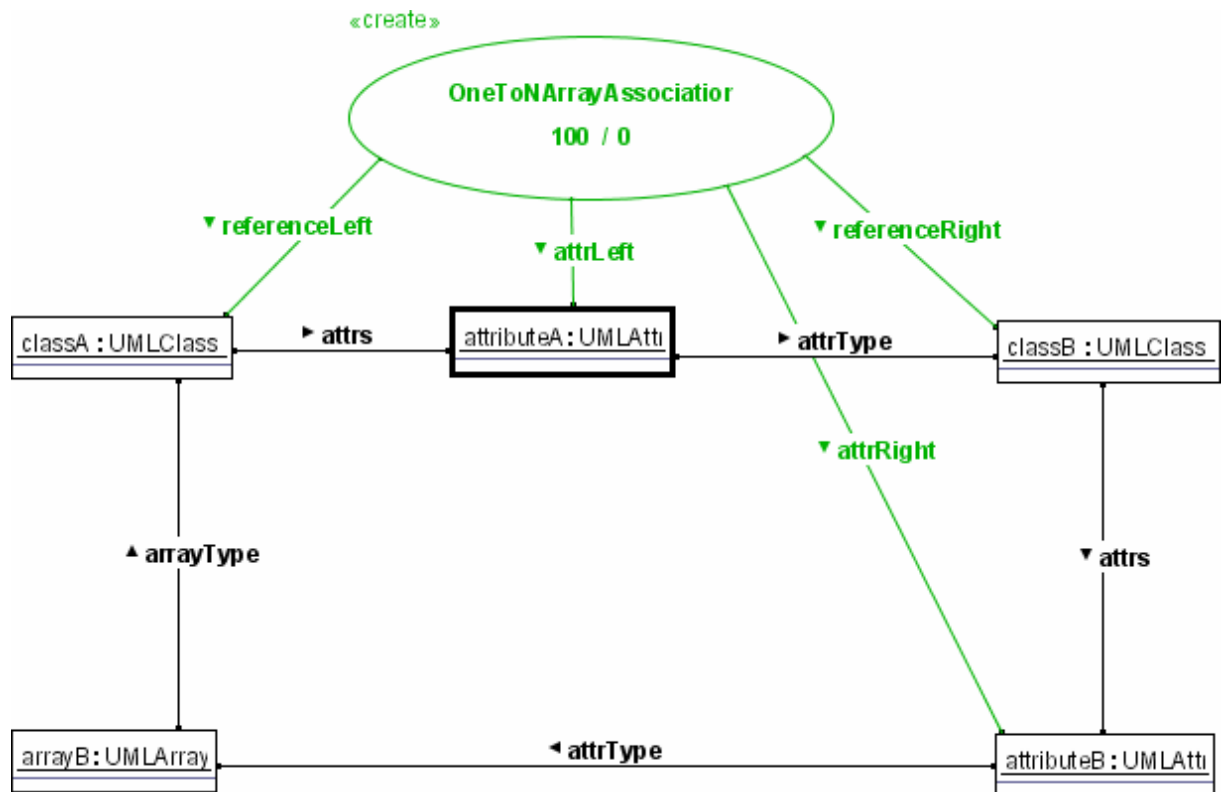


Abbildung 26) Muster für Assoziation mit Kardinalität "1" zu "n" durch Array

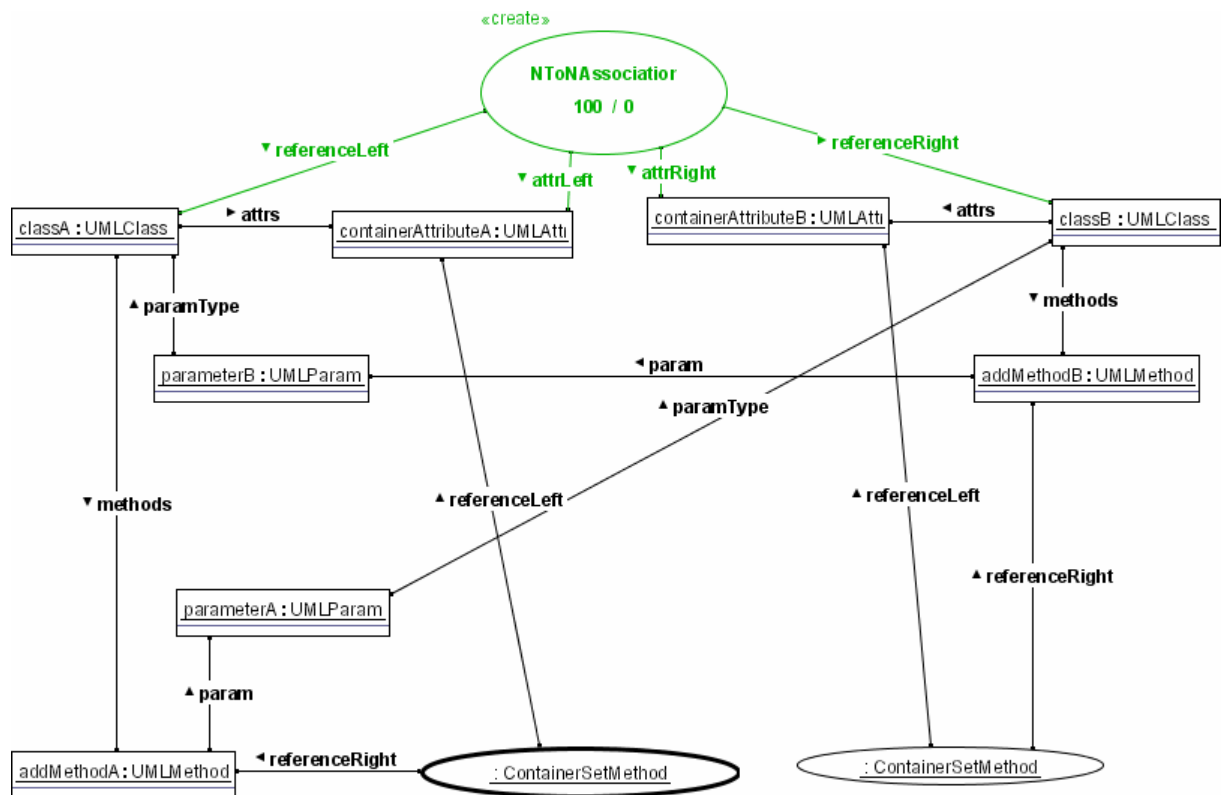


Abbildung 27) Muster für Assoziation mit Kardinalität "n" zu "n"

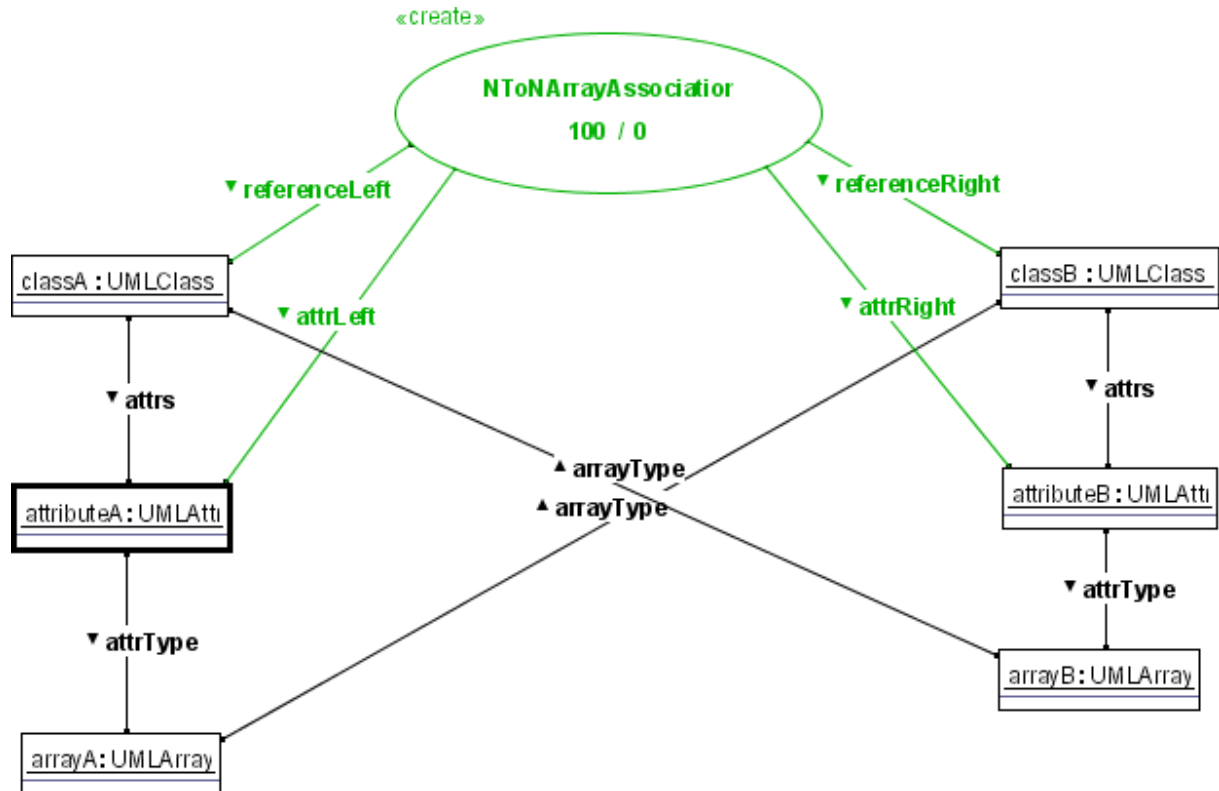


Abbildung 28) Muster für Assoziation mit Kardinalität "n" zu "n" durch Array