Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Christian Hubrich

Praktische Erprobung von Reverse Engineering Werkzeugen für Java vom Bytecode bis zum UML-Modell

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Hubrich

Praktische Erprobung von Reverse Engineering Werkzeugen für Java vom Bytecode bis zum UML-Modell

Abschlussarbeit eingereicht im Rahmen Bachelorthesis

im Studiengang Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Frau Prof. Dr. Steffens
Zweitgutachter : Herr Prof. Dr. Sarstedt

Abgegeben am 23.05.2016

Christian Hubrich

Thema der Arbeit/Ausarbeitung
Praktische Erprobung von Reverse Engineering Werkzeugen für Java vom Bytecode bis zum UML-Modell

Stichworte
Bytecode, Quellcode, Sourcecode, UML, Java, Reverse Engineering, Werkzeuge, Entwicklungsumgebung, Plug-In

Kurzzusammenfassung
In dieser Arbeit soll die praktische Eignung des Reverse Engineering von Software untersucht werden. Dabei wurde der Focus sowohl auf manuelle Methoden, sowie auf werkzeuggestütztes Reverse Engineering gelegt.

Christian Hubrich

Title of the paper
Practical testing of reverse engineering tools for Java

Keywords
Bytecode, Sourcecode, UML, Java, Reverse Engineering, Tool, IDE, Plugin

Abstract
This paper (thesis) was designed to check practical potentials of software investigation (understanding, destruction) with respect to reverse engineering. The trials concentrated as well on manual procedures as on reverse engineering supported by tools.

Inhaltsverzeichnis

1 Einleitung

In der Software Entwicklung ändern sich die Anforderungen ständig. Es gibt verschiedene Wege Software zu entwickeln. Traditionell wird ein Modell erstellt, bevor die Software letztendlich implementiert wird. Dieser Prozess kann manuell - aber auch durch Werkzeuge unterstützt - ablaufen. Die Weiterentwicklung der verfügbaren Hilfsmittel und Technologien wird ständig vorangetrieben. Bestimmte Technologien, Sprachen und Werkzeuge haben sich bereits erfolgreich in der Software-Entwicklung bewährt. Hierzu zählen die Modellierungssprache Unified Modeling Language (UML) und die Programmiersprache Java. Moderne Software wird in Projekten mit Hilfe von Entwicklungsumgebungen umgesetzt. Diese bieten ihrerseits eine Fülle von Erweiterungen an, die die Software Entwicklung automatisieren.
Im Gegensatz dazu befasst sich das Reverse Engineering mit dem fertigen Softwareprodukt. Dabei werden folgende Problemstellungen untersucht: Lässt sich der Weg der traditionellen Software-Entwicklung umkehren? Kann bereits fertige Software wieder verwendet werden, um sie zu verbessern? Welche Faktoren sind dabei wichtig? Oftmals müssen bestehende Programme erweitert, neu aufgesetzt oder (neu) dokumentiert werden. Oder es gilt Versäumnisse, wie z.B. eine mangelhafte Dokumentation, Abweichungen vom ursprünglichen Modell und der umgesetzten Implementation oder aber ein fehlendes Modell nachzuholen. Wie ist in solchen Fällen vorzugehen?

1.1 Zielsetzung

In dieser Arbeit wird zunächst die Stellung des Reverse Engineering in Bezug zur Software-Entwicklung abgeklärt. .Anschließend werden zwei Anwendungsbereiche vorgestellt.
Der erste Teil untersucht die Möglichkeit, vorhandenen Bytecode wieder vollständig in Quellcode zurück zu übersetzen. Der zweite Abschnitt befasst sich mit der Evaluierung von Werkzeugen, die die automatische Generierung von Quellcode in Modelle unterstützen.

1.2 Struktur der Arbeit

Diese Arbeit ist in sieben Kapitel gegliedert, die folgendermaßen aufgebaut sind:

Kapitel 1 gibt eine Einleitung und führt in die Thematik ein. Danach (in Kapitel 2) folgen begriffliche Einordnungen, die zur Vermittlung theoretischer Grundkenntnisse dienen sollen.

Behandelt werden hier Themengebiete wie Reverse Engineering, Technologien und Werkzeugunterstützung. Es schließt sich der Analyseteil in zwei Abschnitten an. Kapitel 3 beschäftigt sich mit der Analyse des Reverse Engineering von Bytecode in Quellcode. Der nächste Abschnitt (Kapitel 4) umfasst die werkzeuggestützten Modellgenerierung von Quellcode in UML-Modelle und stellt zunächst die Anforderungen, die in der Fachliteratur an die Werkzeuge gestellt werden, zusammen. In Kapitel 5 wird dann ein Fallbeispiel eingeführt, welches die Grundlage für die Analyse der zu untersuchenden Werkzeuge in Kapitel 6 bildet. Die Arbeit endet mit einem Fazit und gibt einen Ausblick auf weitere mögliche Fragestellungen (Kapitel 7).

2 Begriffliche Einordnung

Das folgende Kapitel bietet einen Überblick über die wichtigsten Begriffe zum Thema “Reverse Engineering“ und legt dar, welche Einsatzarten und Werkzeuge unterschieden werden können und welche Motivation hinter den einzelnen Arten steckt. Des Weiteren werden Begriffe, die für diese Arbeit im Zusammenhang mit dem Reverse Engineering in Bezug auf Softwareentwicklung wichtig sind erklärt.

2.1 Reverse Engineering von Software

Reverse Engineering hat seinen Ursprung in der Analyse von Hardware. Es dient dazu das eigene Produkt gegenüber dem der Konkurrenz zu verbessern. M.G. Rekoff beschreibt den Prozess als eine Entwicklung die von Personen geleitet wird die nicht die Vorlage selbst entwickelt haben. (Vergl. REKO, S. 244ff).

Auf die Analyse von Software übertragen verwendet man dieses Konzept, um die Komponenten des Systems und ihrer Beziehungen zueinander zu identifizieren und um eine Repräsentation des Systems in einer anderen Form oder auf einer höheren Abstraktionsebene zu erzeugen. Hier liegt der entscheidende Unterschied zum Reverse Engineering von Hardware, welches normalerweise einen Klone des zu untersuchenden Produktes als Ziel hat.

Für die im Anschluss folgenden Definitionen im Zusammenhang von Reverse Engineering von Software sind vorab noch weitere Begriffe für das Gesamtverständnis von Bedeutung. Es handelt sich um Software Wartung, Software Life Cycles, Subjekt System und Abstraktion.

Software Wartung:
Die ANSI Definition von Software Wartung besagt sinngemäß: „Software Wartung ist die Modifikation eines Software Produktes nach dessen Auslieferung um Fehler zu korrigieren, die Performance zu verbessern oder Attribute hinzuzufügen.“ (Vergl. ANSI)

Software Life Cycles, Subjekt System und Abstraktion:
Um auf hinreichende Weise das Forward Engineering und das Reverse Engineering zu beschreiben, müssen zuerst drei voneinander abhängige Konzepte erklärt werden:
- die Existenz von einem Life Cycle Modell,
- die Gegenwart eines Subjekt Systems,
- die Identifikation von Abstraktionsebenen.

Wir gehen davon aus, dass jedem Software Entwicklungsprozess ein Life Cycle Modell zugrunde liegt (Wasserfall, etc.). Wir erwarten von diesem Modell, dass es iterativ für die verschiedenen Schritte des Modells ist, evtl. auch rekursiv. Daraus leiten (definieren) wir Forward und Backward Aktivitäten ab.

Beim Subjekt System handelt es sich entweder um ein einzelnes Code Fragment oder um ein komplexes System. In jedem Fall ist das Subjekt System im Forward Engineering das Ergebnis des Entwicklungsprozesses. Beim Reverse Engineering ist das Subjekt System im Allgemeinen der Ausgangspunkt des Entwicklungsprozesses.

Beim Life Cycle Modell bedeuten die frühen Schritte eher eine allgemeine Darstellung des zu entwickelnden Systems. In späteren Phasen gehen sie mehr ins Detail der Implementation. Während des Life Cycle Modells geht die Entwicklung mit wachsender Detailtiefe einher mit der Sicht auf die Abstraktionsebenen. Es ist allerdings wichtig, bei Abstraktionsebenen zu unterscheiden, ob diese innerhalb einer Phase des Entwicklungsprozesses oder über den gesamten Entwicklungsprozess betrachtet werden.

Der Zusammenhang zwischen einer Wartung (in der Reverse Engineering seine Anwendung finden kann) und dem Life Cycle Modell ergibt sich aus dem Zeitpunkt, zu dem Änderungen an der technischen Konfiguration stattfinden. Erfolgen diese Änderungen vor der Freigabe eines Softwareprodukts, so bezeichnet man sie als Entwicklung. Werden nach Auslieferung einer Anwendung Anpassungen vorgenommen, so versteht man dies als Wartung.

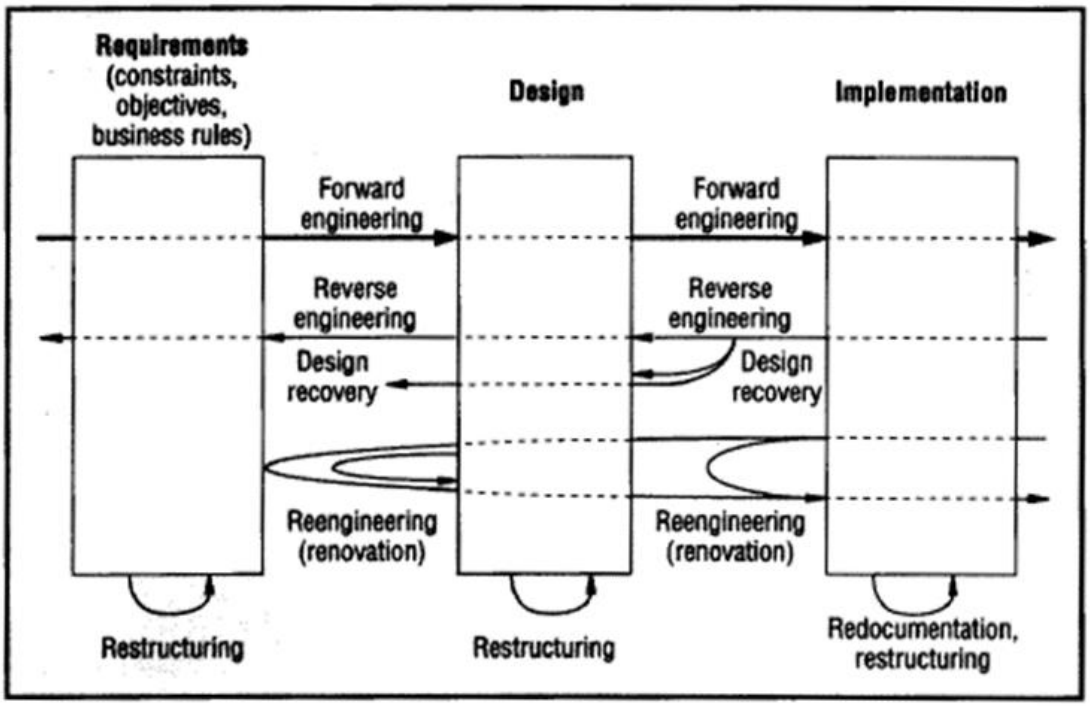

Figure 1. Relationship between terms. Reverse engineering and related processes are transformations between or within abstraction levels, represented here in terms of life-cycle phases.

Abbildung 1 (Quelle: CHIK, S.14)

Forward Engineering

Forward Engineering steht für den traditionellen Prozess der Softwareentwicklung. Dabei wird von einer abstrakten implementationsunabhängigen Ebene ausgegangen, und auf dieser basierend wird eine konkrete Implementation realisiert. Forward Engineering beginnt bei der Anforderungsdefinition eines Systems. Diese wird in einem Entwurf umgesetzt, daraus resultiert schließlich die Implementation.

Reverse Engineering

Reverse Engineering umschreibt einen Analyseprozess. Dieser soll die Komponenten des Systems und ihrer Beziehungen zueinander identifizieren und eine Repräsentation des Systems in einer anderen Form oder auf einer höheren Abstraktionsebene erzeugen. Reverse Engineering dient lediglich zur gründlichen Untersuchung des Systems. Es werden keinerlei Veränderungen vorgenommen. Reverse Engineering kann von jeder Abstraktionsebene oder jeder Phase des Life Cycle Models gestartet werden. Es gibt weitere Teilbereiche des Reverse Engineering, die zwei Bereiche auf die am häufigsten zurückgegriffen wird sind die Re-Documentation und das Design Recovery.

Re-Documentation
Re-Documentation erzeugt oder überarbeitet eine semantisch äquivalente Repräsentation auf der gleichen Abstraktionsebene. Re-Documentation findet ausschließlich innerhalb der Implementationsebene statt. Zur äquivalenten Repräsentation zählen u.a. Veränderungen der Gestalt am Quelltext (Einrückungen, Umbrüche, etc.), Diagrammerzeugung (Datenfluss, Kontrollfluss, etc.) sowie die Erzeugung von Querverweislisten. Das Ziel ist die vereinfachte Visualisierung von Beziehungen zwischen den Programmkomponenten.

Design Recovery
Design Recovery bezeichnet einen Teilbereich des Reverse Engineering. Hierbei wird bei der Untersuchung eines Systems ein breites Spektrum an Informationen (externes Wissen) in den Abstraktionsvorgang einbezogen als es üblicherweise in Quelltexten und konventionellen Dokumenten in der Softwareentwicklung der Fall ist. Externes Wissen bedeutet in diesem Zusammenhang Wissen aus einem Anwendungsbereich, Deduktion, Fuzzy Reasoning, externe Dokumentation, persönliche Erfahrung sowie sonstige zur Verfügung stehende Informationen. Ziel ist das Zusammenfassen aller verfügbaren Informationen über ein System auf einer höheren Abstraktionsebene.

Restructuring
Restructuring dient zur Transformation von einer Darstellungsform in eine andere - auf der gleichen Abstraktionsebene -, wobei die Funktionalität (das Verhalten nach außen, funktional und semantisch) unverändert bleibt. Anders als bei der Re-Documentaion, die zu einem besseren Verständnis beiträgt, dient das Restructuring zu einer qualitativen Verbesserung der Dokumente. Ein Beispiel hierfür ist die Umstrukturierung von sogenannten Spaghetti-Quelltext in eine strukturierte Form (ohne GOTOs).

Reengineering
Unter Reengineering versteht man die Neuentwicklung oder Modifikation eines Systems unter vorheriger Untersuchung eines Altsystems mittels Reverse Engineerings.
(Vergl. CHIK S. 13-17)

2.1.1 Werkzeuge des Reverse Engineering

Der Reverse Engineering Prozess kann durch eine Vielzahl an Werkzeugen unterstützt werden. Sie existieren für unterschiedliche Zwecke und können in vier Klassen aufgeteilt werden: Es handelt sich dabei um Editor- und Browsing-Werkzeuge, Visualisierungs-Werkzeuge, Metrik-Werkzeuge und Quelltexttransformations-Werkzeuge.

Editor- und Browsing-Werkzeuge werden zur Betrachtung von Quelltexten benutzt. In ihrem Funktionsumfang finden sich vordefinierte oder auch benutzerdefinierte Formatierungen. So können bestimmte Konstrukte farblich hervorgehoben, eingerückt oder in einem

bestimmten Font dargestellt werden. Teilweise sind auch Suchmechanismen implementiert, um definierte oder angewandte Vorkommen aufzufinden.

Visualisierungs-Werkzeuge erlauben dem Anwender, Detailinformationen des Quelltextes auszublenden und grafische Repräsentationen auf einem abstrakteren Niveau zu erzeugen. Hierzu gehören Werkzeuge die Kontroll- und/ oder Datenflussdiagramme erzeugen.

Metrik-Werkzeuge erzeugen Messwerte für existierende Anwendungen. Typischerweise werden sie dazu benutzt um z.B. die Größe eines Programms (Zeilen und Werte) festzustellen; daraus kann man einen Rückschluss auf die Komplexität des Programms ziehen. In der Praxis sind Metrik-Werkzeuge bereits oftmals in Browsing- und Visualisierungswerkzeuge integriert.

Die letzte Klasse bilden die Quelltexttransformations-Werkzeuge. Im Unterschied zu den vorherigen Werkzeugarten verändern diese Quelltexte. Sie werden für eine Überführung in eine strukturierte Form, in der als Kontrollstrukturen nur noch die Sequenz sowie die bedingte Verzweigung und die Schleife auftreten, benutzt. Des Weiteren können damit Anpassungen an Namenskonventionen und Programmierrichtlinien durchgeführt werden.
Auch Werkzeuge, die Quelltexte von einer Programmiersprache in eine andere transformieren, zählen zu dieser Klasse (vergl. CREM S.10f).

2.1.2 Bedeutung für diese Arbeit

Bezieht man die Definitionen aus Kapitel 2.1 auf Quellcode und UML, bedeutet Reverse Engineering nichts anderes, als die Generierung von UML-Modellen aus Quellcode. Demgegenüber bezeichnet der Begriff Forward Engineering die Generierung von Quellcode aus UML-Modellen. Diese Vorgänge können manuell durch den Entwickler durchgeführt, oder auch mit Hilfe von speziellen Werkzeugen (s. Kapitel 2.1.1) teilweise oder vollständig automatisiert werden.
In dieser Arbeit werden in einem konstruierten Anwendungsfall (Kapitel 5f) verschiedene Programme analysiert, die zu der Klasse der Visualisierungs-Werkzeuge zählen. Sie können Java Quellcode in UML Modelle transformieren. Die in Kapitel 3 durchgeführte Rückübersetzung von Bytecode in Quellcode findet mit Hilfe eines Dekompilers statt, der zur Klasse der Quelltexttransformations-Werkzeuge zählt. Die Analyse dieser Transformation wird jedoch ohne weitere Werkzeughilfe durchgeführt.

2.2 UML

Die Kurzform UML steht für Unified Modeling Language. Es handelt sich um eine vereinheitliche Modellierungssprache, die zur Konstruktion, Dokumentation, Spezifikation und Visualisierung komplexer Systeme dient (vergl. WEB1). Sie bleibt dabei unabhängig von deren Fach- und Realisierungsgebiet. Die UML bietet Notationselemente sowohl für statische als auch dynamische Analyse-, Design- und Architekturmodelle. Besonders gut ist sie für objektorientierte Vorgehensweisen geeignet (vergl. RUPP S.4f).

2.2.1 Einsatzarten von UML

UML kann als Modellierungssprache über den gesamten Softwareentwicklungsprozess hinweg eingesetzt werden. Dies hat den Vorteil, dass in allen Stufen der Entwicklung die gleichen Sprachkonzepte und Notationen verwendet werden. Durch die UML wird kein eigenes Vorgehensmodell definiert und sie legt sich auch nicht auf ein bestimmtes Vorgehensmodell fest. UML ist unabhängig von Entwicklungswerkzeugen und Programmiersprachen. UML bietet somit eine Eignung für Anwendungsbereiche, mit unterschiedlichsten Anforderungen in Bezug auf Komplexität, Datenvolumen, Berechnungsintensität, Echtzeit oder Verteilung. Aufgrund dieser Eigenschaften reichen die Einsatzmöglichkeiten von der Analysephase bis hin zur Implementierungsbeschreibungen einer Softwareentwicklung. UML kann sowohl bei Echtzeitsystemen wie auch für verteilte Systeme angewendet werden (vergl. HITZ S.7).

Die in dieser Arbeit verwendete Version 2.0 wurde von der Object Management Group im April 2005 offiziell verabschiedet. Die Object Management Group ist ein internationales Konsortium, welches sich um die Entwicklung von Standards kümmert (vergl. HITZ S.3).
Neben einigen Änderungen und Erweiterungen, werden in UML 2.0 im Vergleich zur Version 1.x, auch einige neue Diagramme beschrieben.

2.2.2 UML 2.0

Die UML 2.0 zeichnet sich durch 14 verschiedene Diagrammtypen aus, von denen sieben für die Modellierung der Statik des Systems gedacht sind. Es gibt sieben Verhaltensdiagramme, vier dieser Diagramme werden den Interaktionsdiagrammen zu geordnet. Alle Diagramme sind in Tabelle 1 aufgelistet (vergl. RUPP S.7).

Strukturdiagramme	**Verhaltensdiagramme**	
		Interaktionsdiagramme
Klassendiagramm	Use-Case-Diagramm	Sequenzdiagramm
Paketdiagramm	Aktivitätsdiagramm	Kommunikationsdiagramm
Objektdiagramm	Zustandsautomat	Timingdiagramm
Kompositionsstrukturdiagramm		Interaktionsübersichtsdiagramm
Komponentendiagramm		
Verteilungsdiagramm		
Profildiagramm		

Tabelle 1: vergl. RUPP S.7 Tabelle 1.1

2.2.3 Relevante UML 2.0 Diagrammarten

Für diese Arbeit sind zwei Diagrammtypen von besonderer Bedeutung. Dabei handelt es sich um Klassendiagramme (Strukturdiagramm) und Sequenzdiagramme (Verhaltensdiagramm). Im weiteren Verlauf werden verschiedene Tools zur Erstellung von UML-Diagrammen aus vorhandenem Java Quellcode untersucht und bewertet. Diese Programme beschränken sich zum Teil auf wenige Diagrammtypen. Damit ein möglichst genauer Vergleich der Werkzeuge stattfinden kann, konzentriert sich diese Arbeit auf diese zwei Diagrammtypen. Sie werden hier kurz vorgestellt.

Klassendiagramm

Klassendiagramme bieten die Möglichkeit, die Struktur des zu entwerfenden oder abzubildenden Systems darzustellen. Es beschreibt die statischen Eigenschaften einer Klasse und deren Beziehungen zueinander. Das Klassendiagramm bildet den Kern der Modellierungssprache UML und nimmt somit einen sehr wichtigen Faktor im Softwareentwicklungsprozess ein (vergl. HITZ S. 52f). Es gibt eine mögliche Antwort auf die Frage „Wie sind die Daten und das Verhalten meines Systems im Detail strukturiert?“(vergl. RUPP S. 107).

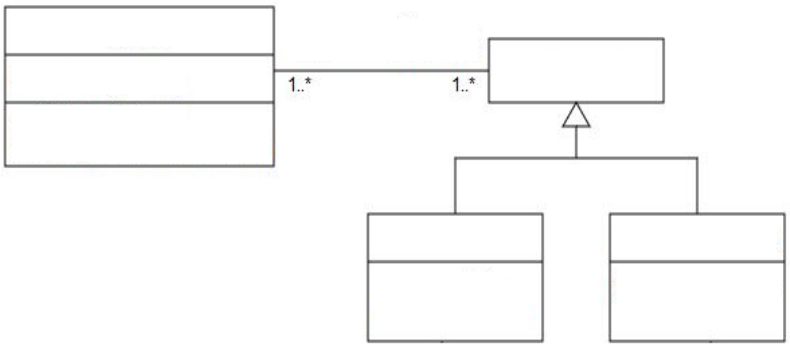

Abbildung 2: Klassendiagramm als Skizze

Sequenzdiagramm

Sequenzdiagramme dienen zur Modellierung des Informationsaustausches zwischen beliebigen Kommunikationspartnern innerhalb eines Systems. So können feste Reihenfolgen, zeitliche und logische Ablaufbedingungen, Schleifen und Nebenläufigkeiten dargestellt werden (vergl. HITZ S. 251f). Es beantwortet die Frage »Wie läuft die Kommunikation in meinem System ab«(vergl. RUPP S.401).

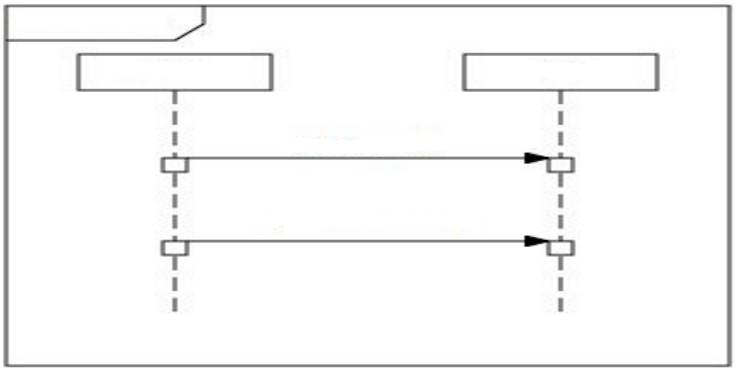

Abbildung 3: Sequenzdiagramm als Skizze

2.3 Java

Java ist eine objektorientierte Programmiersprache. Sie ist ein Bestandteil der Java-Technologie und besteht grundsätzlich aus dem Java-Entwicklungswerkzeug (JDK) zum Erstellen von Java-Programmen sowie der Java-Laufzeitumgebung (JRE) zu deren Ausführung. Die Laufzeitumgebung selbst umfasst die virtuelle Maschine (JVM) und die mitgelieferten Bibliotheken.
Die Eigenschaft der Objektorientierung von Java ermöglicht es moderne und wiederverwertbare Softwarekomponenten zu programmieren. Diese liegen zunächst als sogenannter Quellcode (Sourcecode) vor. Der Quellcode wird von einem Compiler in einen maschinenverständlichen Code übersetzt, den sogenannten Java-Bytecode. Anschließend wird der Bytecode von einer virtuellen Maschine interpretiert. Dieses Merkmal unterscheidet Java von anderen Programmiersprachen, da der Code nicht direkt durch Hardware (spezieller Mikroprozessor), sondern durch entsprechende Software auf der Zielplattform ausgeführt wird.
Durch die Virtualisierung ist der Javacode plattformunabhängig (Windows und Linux). Die Installation der Java Laufzeitumgebung ermöglicht es den Code bzw. das in Java geschriebene Programm auf jeder Rechnerarchitektur laufen zu lassen.
Um die Ausführungsgeschwindigkeit zu erhöhen, werden Konzepte wie die Just-in-time-Kompilierung und die Hotspot-Optimierung verwendet. In Bezug auf den eigentlichen Ausführungsvorgang kann die JVM den Bytecode also interpretieren, ihn bei Bedarf jedoch auch kompilieren und optimieren (vergl. ULLE Kap. 1.2ff).

Für diese Arbeit wurden sämtliche Beispiele in Java programmiert. Der oben beschriebene Vorteil der Just-in-time-Kompilierung erleichtert die Arbeit beim Dekompilieren von Java Bytecode sehr. Dieser Vorgang wird in Kapitel 3 weiter ausgeführt. Für diese Arbeit habe hat der Verfasser die Version JRE 1.7 verwendet, da die zu untersuchenden Tools (s. Kapitel 5 und folgende) mit dieser Version fehlerfrei liefen.

2.4 Design Pattern

Design Pattern (Entwurfsmuster) sind bewährte Lösungswege für wiederkehrende Designprobleme in der Softwareentwicklung. Durch sie werden Entwurfsentscheidungen beschrieben. Der Einsatz von Design Pattern führt zu einem Entwurf der an Flexibilität, Wiederverwendbarkeit, Erweiterbarkeit gewinnt, darüber hinaus ist er einfacher zu verwenden und änderungsstabil.

Das erste Buch über Entwurfsmuster wurde von der sogenannten Gang of Four veröffentlicht, dieses umfasst 23 Design Patterns für eine Vielzahl von Problemen (vergl. BRÜG, S. 719ff).

Die Autoren beschreiben 4 maßgebliche Elemente die jedes Pattern umfasst. Diese sind der Pattern-Name, der zur Referenzierung des jeweiligen Design Problems und seiner Lösung dient. Die Problemstellung welche die Situation beschreibt für die das Pattern anzuwenden ist. Die Lösung, die jedoch keine konkrete Implementierung sein soll, sondern vielmehr eine Art Schablone die für viele verschiedene Situationen anwendbar ist. Das letzte Element sind die Konsequenzen. Diese beschreiben eine Übersicht der Auswirkungen und Kompromisse, die die Anwendung des Patterns mit sich bringt (vergl. GAMM, S.28).

Die Abbildung zeigt ein Entwurfsmuster, das sogenannte Strategy Pattern.

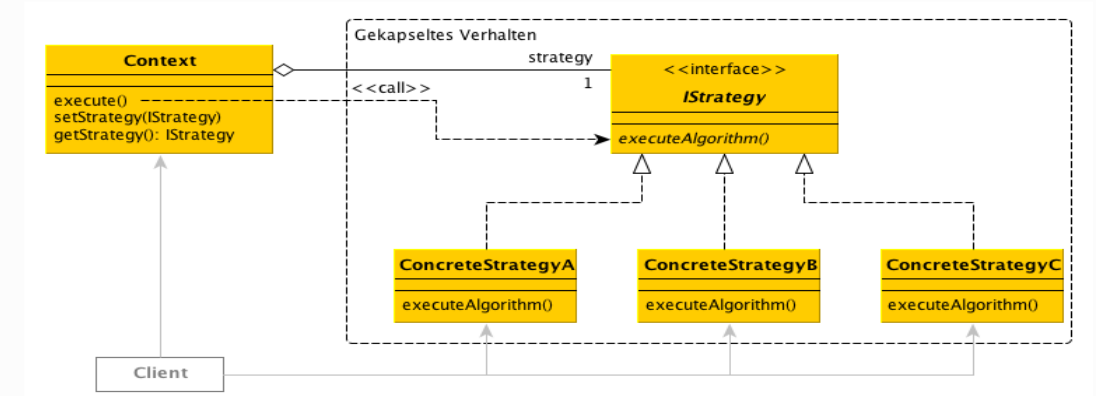

Abbildung 4: vergl. WEB2

In dieser Arbeit wird an späterer Stelle ein Fallbeispiel analysiert, dabei wird das oben angesprochene Design Pattern in Form des Strategy Pattern angewandt.

2.5 Eclipse

Eclipse ist die Bezeichnung eines Projektes. Dieses dient der Schaffung von Open-Source-Tools und –Frameworks. Es zeichnet sich durch seine Unabhängigkeit von bestimmten Sprachen oder Technologien aus. Vor der Version 3.0 wurde Eclipse als integrierte Entwicklungsumgebung (IDE) für die Programmiersprache Java genutzt. Inzwischen wird es wegen seiner Erweiterbarkeit auch für viele andere Entwicklungsaufgaben eingesetzt. Eclipse lässt sich durch eine Vielzahl sowohl quelloffener als auch kommerzieller Erweiterungen ergänzen (vergl. KÜNN, S.15).

Für diese Arbeit wurde die Version Luna 4.4 verwendet.

2.5.1 Eclipse Plug-In

Plug-Ins erweitern Software um zusätzliche Funktionen. Alle Plug-Ins werden in Java programmiert und bleiben damit plattformunabhängig.
Es gibt verschiedene Arten wie sich Plug-Ins in Eclipse integrieren lassen. Am einfachsten lassen sich diese Erweiterungen über das in Eclipse integrierte Menu für die Installation neuer Software erledigen. Sollte das nicht möglich sein, kann ein Plug-In auch manuell installiert werden.
Da es inzwischen eine sehr große Anzahl an Plug-Ins für Eclipse gibt ist eine genaue Beschreibung bzw. Kategorisierung der Erweiterung notwendig. Der Eclipse Marketplace erleichtert es dem Anwender durch zahlreiche Hilfsfunktionen, das passende Plug-In zu finden (vergl. KÜNN, S.181ff).
In der vorliegenden Arbeit werden verschiedene Plug-Ins für Reverse Engineering untersucht und bewertet. Diese können bereits vorhandenen Quellcode in verschiedene UML-Modelle mittels Reverse Engineering transformieren.

3 Das Dekompilieren von Java Bytecode

Der folgende Abschnitt bildet den Einstieg in den praktischen Teil der Arbeit. Er behandelt das Reverse Engineering von Java Bytecode in Java Quellcode. Dieser Code bildet die Grundlage für die anschließende Rückübersetzung in ein UML Modell. Zunächst werden die Grundlagen des Reverse Engineering mittels eines Dekompilers erklärt, anschließend wird ein konkreter Fall gezeigt.

Da in der Praxis nicht immer Quellcode zur Verfügung steht, sondern meist nur Bytecode muss letzterer zuerst zurückübersetzt werden. Zum besseren Verständnis sollen hier die Grundlagen erklärt und einige Beispiele gegeben werden. Abschließend wird eine Klasse aus einem vom Verfasser umgesetzten Pattern - ebenfalls per Reverse Engineering – zurück übersetzt. Es ist anzumerken, dass die Rückgewinnung eines UML-Modells nicht zu den Haupteinsatzgebieten des Reverse Engineering von Java Bytecode zählt. Dieses Kapitel soll zeigen, dass dies aber grundsätzlich möglich ist und im gezeigten Rahmen auch sinnvoll sein kann.

Typische Anwendungsgebiete sind:
• „Quick and dirty“-Fehlerbehebung von Klassendateien. Es besteht keine Notwendigkeit die dekompilierten Ergebnisse neu zu kompilieren.
• Das Analysieren von verschleiertem Code.
• Das Erstellen von verschleiertem Code.
• Das Erstellen von Compiler-Codegeneratorn für Sprachen die auf der JVM laufen (Scala, Clojure, Groovy, etc.) (vergl. WEB3).

Da der Java-Compiler keine Optimierungsarbeiten übernimmt, lässt sich der Java Bytecode grundsätzlich gut dekompilieren. Die JVM erledigt dies zur Laufzeit. Das hat zur Folge, dass der Bytecode von Klassendateien i.d.R. sehr gut lesbar ist.

3.1 JVM Speichermodell

Man betrachtet drei voneinander isolierte Bereiche:

• Local Variable Array(LVA):
Es wird als Speicher für eingehende Funktionsargumente und lokale Variablen eingesetzt. Anweisungen wie „iload_0“ (vergl. WEB4) laden Werte aus dem LVA. Anweisungen wie „istore“ speichern Werte im LVA. Jeder Slot hat eine Größe von 32-Bit. Daher belegen Werte vom Datentyp Long und Double zwei Slots.

• Stack:
Der Stack wird zum Abspeichern von Berechnungen und Methoden Parametern beim Aufrufen anderer Funktionen verwendet.

• Heap:
Wird als Speicher für Objekte und Arrays verwendet.

3.2 Einfache Wertrückgabe

Zu den einfachsten Java Methoden zählt jene, die lediglich einen Wert zurückgibt.
Ein Beispiel in Quellcode:

```
public class ret0 {
        public static int main(String[] args) {
                return 0;
        }
}
```

Die Klasse als Bytecode:
Der Bytecode ist für einen Softwareentwickler schlecht bzw. unlesbar, es lassen sich zwar gewisse Informationen ableiten, jedoch wird der überwiegende Teil kryptisch dargestellt.

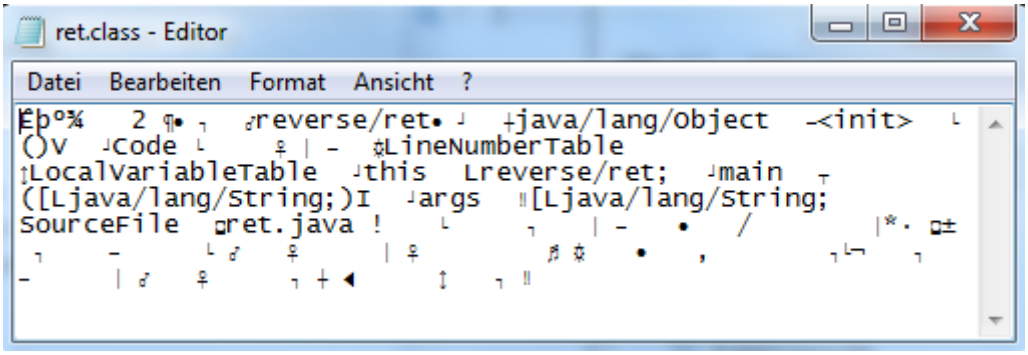

Abbildung 5: Editoranzeige der Java Klasse ret.class

Java Standard Dekompiler:
Wie oben beschrieben ist die Ausgabe des Standard Dekompilers sehr gut lesbar. Der Befehl zur Ausführung des Dekompilers in diesem Fall lautet: javap -c -verbose ret.class.

Wir erhalten somit folgendes:

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: iconst_0
       1: ireturn
```

Abbildung 6: Ausgabe des Standard Dekompilers der Java Klasse ret.class

Java-Entwickler entschieden, dass die „0“ eine der meist genutzten Konstanten in der Programmierung ist, daher gibt es eine separate Anweisung „iconst_0 „ .Diese legt die „0“ auf den Stack. Es gibt noch weitere „iconst <n>“ Anweisungen (n=1,…,5), die die jeweils den Wert 1,…,5 auf den Stack legen. Außerdem gibt es eine Anweisung „inconst_m1“ die den Wert -1 auf den Stack legt.

Der Stack wird in der JVM für die Daten verwendet, die in Funktionen aufgerufen werden und für Rückgabewerte. So legte „iconst_0“ die „0“ in den Stack und „ireturn“ liefert den Integer-Wert (i im Namen bedeutet Integer) von der obersten Position des Stacks (TOS).

Folgende Tabelle gibt eine Übersicht über die verschiedenen Datentypen in Java und die Ausgabe des Dekompilers wenn diese mit einer Methode zurückgegeben werden:

Methode	Datentyp	Ausgabe	Bemerkung

public static int main (String[]args) {
return 1234;
}
	Integer	stack=1, locals=1, arg_size=1
		0: sipush 1234
		3: ireturn
			sipush (short integer) legt den Wert 1234 auf den Stack. Short impliziert einen 16-Bit Wert, der auf den Stack gelegt wird. Die Zahl 1234 passt in der Tat in einen 16-Bit Wert.

public static int main (String[]args) {
return 12345678;
}
	Integer	stack=1, locals=1, arg_size=1
		0: ldc #2 //12345678
		2: ireturn
			Es ist nicht möglich einen 32-Bit Wert in einen JVM Befehlsoperationscode zu codieren. Daher wird ein 32-Bit Wert wie 12345678 in den sogenannten “constant pool” gespeichert. Dieser ist so etwas wie ein Nachschlagewerk für die meist genutzten Konstanten (einschl. Strings, Objekten, etc).

public static boolean main (String[]args) {
return true;
}
	Boolean	stack=1, locals=1, arg_size=1
		0: iconst_1
		1: ireturn
			Dieser JVM Bytecode unterscheidet sich nicht von dem, der die Konstante 1 auf den Stack legt. 32-Bit data slots im Stack werden hier ebenfalls für Boolean Werte benutzt. Die Typ Information ist in der „class file“ gespeichert und wird zur Laufzeit geprüft.

public static short main (String[]args) { return 1234; }	Short	stack=1, locals=1, arg_size=1 0: sipush 1234 3: ireturn	Dieser Fall ist genauso wie das erste Beispiel dieser Tabelle. Short wird intern als Integer behandelt.

public static char main (String[]args) {
return ‘A’;
}
	Char	stack=1, locals=1, arg_size=1
		0: bipush 65
		2: ireturn
			bipush bedeutet “push byte”. Char ist in Java ein 16-bit UTF-16 Character und ist äquivalent mit Short, aber der ASCII code von Character “A” ist 65. Daher ist es möglich die Anweisung für „push byte“ zu verwenden um den Wert auf den Stack zu legen.

public static byte main (String[]args){
return 123;
}
	Byte	stack=1, locals=1, arg_size=1
		0: bipush 123
		2: ireturn
			Siehe vorheriges Beispiel.

public static long main (String[]args) { return 1234567890123456789L; }	Long	stack=2, locals=1, arg_size=1 0: ldc _w #2 //123456789… 2: lreturn	Ein Long 64-Bit Wert wird ebenfalls im „constant pool“ gespeichert, ldc2_w läd ihn und lreturn (long return) liefert ihn zurück.

public static double main (String[]args) {
return 123.456d;
}
	Double	stack=2, locals=1, arg_size=1
		0: ldc _w #2 //123.456d
		2: dreturn
			Die ldc2_w Anweisung wird ebenfalls für Double Werte genutzt. (Diese belegen ebenfalls 64 Bits). dreturn steht für “return double”.

public static float main (String[]args) {
return 123.456f;
}
	Float	stack=1, locals=1, arg_size=1
		0: ldc _w #2 //123.456f
		2: freturn
			Die ldc_w Anweisung wird hier genauso benutzt wie für 32-Bit Integer Werte die vom „constant pool“ geladen werden. freturn steht für “return float”.

public static void main (String[]args) { return; }	Nothing	stack=1, locals=1, arg_size=1 0: return	Die aktuelle Methode wird beendet bzw. verlassen, wobei kein Rückgabewert an den Methoden-Aufrufer zurückgegeben wird

Tabelle 2: Beispiele für Methoden mit Wertrückgaben

Mit diesem Wissen ist es sehr einfach, den Wert des Rückgabetypens abzuleiten.

3.3 Einfache Berechnungen (+, -, /, *)

Die Klasse „calc_half.java“ in Quellcode:

```
public class calc_half {
        static int half(int a) {
                return a / 2;
        }
}
```

Die Ausgabe des Dekompiliers:

```
public static int half(int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: iload_0
       1: iconst_2
       2: idiv
       3: ireturn
```

Abbildung 7: Ausgabe des Dekompilers der Java Klasse calc_half.java

Die erste Anweisung “iload_0” legt das erste Argument der Methode auf den Stack. Die zweite Anweisung „iconst_2“ legt die Konstante „2“ auf den Stack. Nach diesen beiden Anweisungen sieht der Stack wie folgt aus:

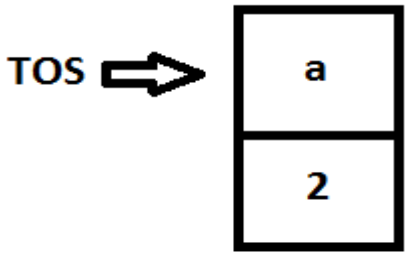

Abbildung 8: Aktueller Stack Status nach den Anweisungen „iload_0“ und „iconst_2“

Die Anweisung “idiv” nimmt die beiden obersten Werte des Stacks und dividiert diese miteinander. Das Ergebnis wird an die oberste Stelle des Stacks gelegt:

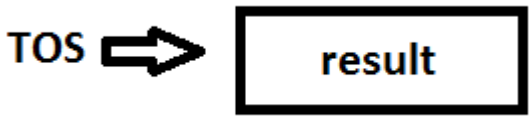

Abbildung 9: Aktueller Stack Status nach der Anweisung „idiv“

Die Anweisung „ireturn“ nimmt das Ergebnis und liefert es zurück.

Die folgende Tabelle gibt einen Überblick über weitere Rechenbeispiele:

Methode	Datentyp	Ausgabe	Bemerkung

public static double half_double(double a){
return a/2.0;
}
	Double	stack=4, locals=2, args_size=1
		0: dload_0
		1: ldc2_w #2 // double 2.0d
		4: ddiv
		5: dreturn
			Es passiert hier genau das gleiche wie im einführenden Rechen Beispiel, nur das hier der Datentyp double ist. Erkennbar durch das prefix „d“ welches vor den jeweiligen Anweisungen steht.

public static long lsum(long a, long b){
return a+b;
}
	Long	stack=4, locals=4, args_size=2
		0: lload_0
		1: lload_2
		2: ladd
		3: lreturn
			The second lload instruction takes second argument from 2nd slot. That’s because 64-bit *long* value occupies exactly two 32-bit slots

public static int mult_add(int a, int b, int c) {
return a*b+c;
}
	Integer	stack=2, locals=3, args_size=3
		0: iload_0
		1: iload_1
		2: imul
		3: iload_2
		4: iadd
		5: ireturn
			Zuerst wird die Multiplikation durchgeführt. Das Produkt liegt an der obersten Stelle des Stacks. Nachdem die Anweisung „iload_2“ ausgeführt wurde, liegt „c“ an oberster Stelle. Jetzt wird mit „iadd“ die Addition ausgeführt und zum Schluss per „ireturn“ zurück geliefert.

Tabelle 3: Beispiele für Methoden mit Berechnungen

3.4 Klassen und Methodenaufrufe

Klasse- /Methodenaufruf	Ausgabe	Bemerkung

public class test {
 	
public int a;
 	
public test(){
a=0;
}
 	
public void set_a (int input)
{
a=input;
}
 	
Public int get_a () {
return a;
}
}
	*/ Class */
	public re_class.test();
	flags: ACC_PUBLIC
	Code:
	stack=1, locals=1, args_size=1
	0: aload_0
	1: invokespecial #1
	4: aload_0
	5: iconst_0
	5: putfield # 2
	9: return
	
	*/ Setter a */
	public static void set_a(int);
	flags: ACC_PUBLIC, ACC_STATIC
	Code:
	stack=1, locals=1, args_size=1
	0: iload_0
	1: putfield #2
	4: return
	
	*/ Getter a */
	public static int get_a();
	flags: ACC_PUBLIC, ACC_STATIC
	Code:
	stack=1, locals=0, args_size=0
	0: getfield #2
	3: ireturn
		*/ Class */
		Zuerst wird die locale Variable auf den Stack gelegt. Dann wird mit der Anweisung “invokespecial” der Konstruktor aufgerufen. Der Verweis auf den „constant pool“ mit #1 zeigt uns was genau passiert. Der Ausdruck „Method java/lang/Object. “<init>”:()V” sagt aus, dass zuerst der Konstruktor der Superklasse (hier die Klassse Object) aufgerufen wird. Anschliessend wird die Konstante 0 auf den Stack gelegt (iconst_0) und mit der Variablen a zugewiesen (putfield). Danach wird der Konstruktor verlassen (return).
		
		/* Setter */
		Zuerst wird die lokale Variable Input auf den Stack geladen (iload_0), dann wird der Wert über „putfield“ gesetzt und im Anschluss wird die Methode verlassen (return).
		
		/* Getter */
		Der Wert der Instanzvariablen des Objektes wird ausgelesen (getfield) und dann zurückgegeben (ireturn).

Tabelle 4: Beispiel für eine Klasse mit Methoden

Es gibt natürlich für jedes Sprachkonstrukt von Java eine geeignete Anweisung, die beim Dekompilieren ausgeführt wird. Diese umfassen z.B. Arrays, Schleifen, Switch-Anweisungen, Exceptions, usw. Es würde jedoch an dieser Stelle den Rahmen überschreiten, alle Befehlsvarianten aufzuführen. Der Verfasser verweist deshalb auf das Buch „Reverse Engineering for Beginners“ (vergl. YURI, S. 620ff).

3.5 Beispielklasse aus dem Pattern

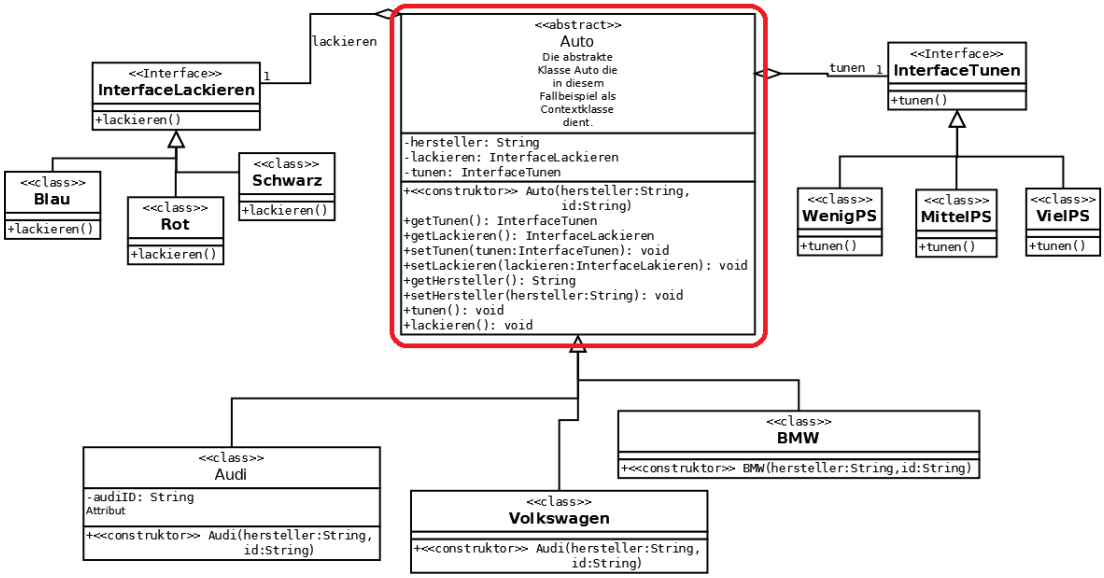

Abbildung 10: Die Autoklasse im Gesamtmodell als UML Klassendiagramm

Die obigen Begriffe sollen nun an einem konkreten Fall anwendet werden. Als Grundlage diene hierbei die Klasse „Auto.class“. Diese ist Teil eines vom Verfasser entworfenen Beispiels, welches im weiteren Verlauf dieser Arbeit als Untersuchungsgegenstand für die Quellcode Rückübersetzung in ein UML-Modell gilt. Unter der Annahme, dass uns nur der Bytecode der „Auto.class“-Datei zur Verfügung steht, wird der Verfasser in diesem Kapitel die Klasse in den Quellcode mittels Reverse Engineering zurück übersetzen. Die Ausgaben des Java Standard Dekompilier, angewendet auf die „Auto.class“-Datei, werden dann Schritt für Schritt analysiert und so der Quellcode wieder hergestellt.

Wir starten mit dem Aufruf des Dekompilers mittels der Eingabe „javap –c –verbose Auto.class“ und erhalten die folgenden Ausgaben:

```
C:\re>javap -c -verbose Auto.class
Classfile /C:/re/Auto.class
  Last modified 20.08.2015; size 1453 bytes
  MD5 checksum 6e1fdd244355ed958e25eb5db7638fa3
  Compiled from "Auto.java"
abstract class patternAuto.Auto
  SourceFile: "Auto.java"
  minor version: 0
  major version: 50
  flags: ACC_SUPER, ACC_ABSTRACT
Constant pool:
   #1 = Class              #2             //  patternAuto/Auto
   #2 = Utf8               patternAuto/Auto
   #3 = Class              #4             //  java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               tunen
   #6 = Utf8               LpatternAuto/InterfaceTunen;
   #7 = Utf8               lackieren
   #8 = Utf8               LpatternAuto/InterfaceLakieren;
   #9 = Utf8               hersteller
  #10 = Utf8               Ljava/lang/String;
  #11 = Utf8               <init>
  #12 = Utf8               (Ljava/lang/String;)V
  #13 = Utf8               Code
  #14 = Methodref          #3.#15         //  java/lang/Object."<init>":()V
  #15 = NameAndType        #11:#16        //  "<init>":()V
  #16 = Utf8               ()V
  #17 = Class              #18            //  patternAuto/WenigPS
  #18 = Utf8               patternAuto/WenigPS
  #19 = Methodref          #17.#15        //  patternAuto/WenigPS."<init>":()V
  #20 = Fieldref           #1.#21         //  patternAuto/Auto.tunen:LpatternAuto/InterfaceTunen;
  #21 = NameAndType        #5:#6          //  tunen:LpatternAuto/InterfaceTunen;
  #22 = Class              #23            //  patternAuto/Blau
  #23 = Utf8               patternAuto/Blau
  #24 = Methodref          #22.#15        //  patternAuto/Blau."<init>":()V
  #25 = Fieldref           #1.#26         //  patternAuto/Auto.lackieren:LpatternAuto/InterfaceLakieren;
```

Abbildung 11: Ausgabe des Java Dekompilers nach Aufruf der Klasse „Auto.class“ Teil 1

```
#26 = NameAndType        #7:#8          //  lackieren:LpatternAuto/InterfaceLakieren;
#27 = Fieldref           #1.#28         //  patternAuto/Auto.hersteller:Ljava/lang/String;
#28 = NameAndType        #9:#10         //  hersteller:Ljava/lang/String;
#29 = Utf8               LineNumberTable
#30 = Utf8               LocalVariableTable
#31 = Utf8               this
#32 = Utf8               LpatternAuto/Auto;
#33 = Utf8               getTunen
#34 = Utf8               ()LpatternAuto/InterfaceTunen;
#35 = Utf8               setTunen
#36 = Utf8               (LpatternAuto/InterfaceTunen;)V
#37 = Utf8               getLackieren
#38 = Utf8               ()LpatternAuto/InterfaceLakieren;
#39 = Utf8               setLackieren
#40 = Utf8               (LpatternAuto/InterfaceLakieren;)V
#41 = Utf8               getHersteller
#42 = Utf8               ()Ljava/lang/String;
#43 = Utf8               setHersteller
#44 = InterfaceMethodref #45.#47        //  patternAuto/InterfaceTunen.tunen:()V
#45 = Class              #46            //  patternAuto/InterfaceTunen
#46 = Utf8               patternAuto/InterfaceTunen
#47 = NameAndType        #5:#16         //  tunen:()V
#48 = InterfaceMethodref #49.#51        //  patternAuto/InterfaceLakieren.lackieren:()V
#49 = Class              #50            //  patternAuto/InterfaceLakieren
#50 = Utf8               patternAuto/InterfaceLakieren
#51 = NameAndType        #7:#16         //  lackieren:()V
#52 = Utf8               SourceFile
#53 = Utf8               Auto.java
```

Abbildung 12: Ausgabe des Java Dekompilers nach Aufruf der Klasse „Auto.class“ Teil 2

Die Abbildungen 11 und 12 geben uns die ersten Informationen über die dekompilierte Klasse. Der Name der Klasse ist „Auto“ und der Accessmodifier ist „abstract“. Die nachfolgenden Ausgaben, die mit einem Rautezeichen „#“ (#1 - #53) gekennzeichnet sind, beinhalten alle Informationen über die im „constant pool“ gespeicherten Werte. In der nachfolgenden Analyse der Methoden der Klasse wird dies noch deutlicher. Die nachfolgenden Befehlssätze der JVM hat der Autor zur besseren Übersicht in einer separaten Tabelle im Anhang zusammengefasst. Hier wird der jeweilige Befehlssatz erklärt, um die Ausgaben in den Quellcode zurück zu übersetzen.

```
public patternAuto.Auto(java.lang.String);
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=2, args_size=2
       0: aload_0
       1: invokespecial #14                 // Method java/lang/Object."<init>":()V
       4: aload_0
       5: new           #17                 // class patternAuto/WenigPS
       8: dup
       9: invokespecial #19                 // Method patternAuto/WenigPS."<init>":()V
      12: putfield      #20                 // Field tunen:LpatternAuto/InterfaceTunen;
      15: aload_0
      16: new           #22                 // class patternAuto/Blau
      19: dup
      20: invokespecial #24                 // Method patternAuto/Blau."<init>":()V
      23: putfield      #25                 // Field lackieren:LpatternAuto/InterfaceLakieren;
      26: aload_0
      27: aload_1
      28: putfield      #27                 // Field hersteller:Ljava/lang/String;
      31: return
    LineNumberTable:
      line 19: 0
      line 10: 4
      line 15: 15
      line 21: 26
      line 22: 31
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      32     0  this   LpatternAuto/Auto;
          0      32     1 hersteller   Ljava/lang/String;
```

Abbildung 13: Ausgabe des Java Dekompilers nach Aufruf der Klasse „Auto.class“ Teil 3

Die Befehlssätze in Abbildung 13 stehen für drei verschiedene Bereiche. Zuerst wird der Auto-Konstruktor Aufruf über „invokespecial“ gestartet und der Konstruktor der Superklasse wird aufgerufen (hier Klasse „Objekt“). Die Stackeinträge 4-12 stehen für die Initialisierung der ersten Instanzvariablen, dieser wird das Objekt „WenigPS“ zugewiesen. Dieselbe Prozedur wird für die Stackeinträge 15-23 wiederholt, diesmal für die zweite Instanzvariable das Objekt „Blau“ zugewiesen. In den Kommentaren des „constant pool“ stehen alle relevanten Informationen wie Name und Typ. Der Befehl new steht für eine neue Instanz eines Objekts. Die letzte Anweisung ist jeweils „putfield“, in dem das neue Objekt der jeweiligen Instanzvariablen zugewiesen wird. Als letzter Schritt findet die Wertzuweisung im Auto-Konstruktor statt (26-31), hier wird „this“ der Wert „hersteller“ zugewiesen, bevor der Konstrutor mit „return“ verlassen wird.

Folgender Quellcode kann aus diesen Informationen abgeleitet werden:

```
abstract class Auto {
        private InterfaceTunen tunen = new WenigPS();
        private InterfaceLakieren lackieren = new Blau();
        private String hersteller;
        public Auto(String hersteller) {  this.hersteller = hersteller;   }
}
```

```
public patternAuto.InterfaceTunen getTunen();
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: getfield      #20                 // Field tunen:LpatternAuto/InterfaceTunen;
       4: areturn
    LineNumberTable:
      line 25: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   LpatternAuto/Auto;
```

Abbildung 14: Ausgabe des Java Dekompilers für die Methode getTunen()

Die Methode hat den Namen „getTunen()“ und hat keinen Parameter; sie hat den Accessmodifier public. Der Rückgabe Typ ist InterfaceTunen, anhand des vom Autor gewählten Namens kann man hier auf ein Interface schließen; dies ist aber nicht die Regel. Man kann lediglich unterscheiden, ob es sich um einen primitiven Datentyp oder einen Referenzdatentyp handelt. Die Angabe „1: getfield“ mit dem Verweis auf den“ Constant pool“ (siehe # 20) gibt uns den Namen für den Wert und den Typ der zurückgegeben wird. Hier ist dies „tunen“ vom Typ InterfaceTunen. Die Anweisung „areturn“ gibt eine Objektreferenz der aktuellen Methode zurück. Somit haben wir bereits alle Angaben der Methode zurückübersetzt und erhalten folgenden Quellcode:

```
public InterfaceTunen getTunen() {
                return tunen;
        }
```

```
public void setTunen(patternAuto.InterfaceTunen);
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=2
       0: aload_0
       1: aload_1
       2: putfield      #20                 // Field tunen:LpatternAuto/InterfaceTunen;
       5: return
    LineNumberTable:
      line 29: 0
      line 30: 5
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   LpatternAuto/Auto;
          0       6     1 tunen   LpatternAuto/InterfaceTunen;
```

Abbildung 15: Ausgabe des Java Dekompilers für die Methode setTunen()

Die Methode hat den Namen „setTunen()“ und hat einen Parameter vom Typ InterfaceTunen, sie hat den Accessmodifier public. Der Rückgabe Typ ist „void“. Die Angeben „aload_0“ und „aload_1“ speichern die Werte der Lokalenvariablen auf den Stack, hier „this“ und „tunen“. Die Angabe „2: pufield“ mit dem Verweis auf den

Constantpool gibt uns den Namen für den Wert und den Typ der gesetzt wird, hier ist dies „tunen“ vom Typ InterfaceTunen. Die Angabe „5: return“ mag auf den ersten Blick ein wenig verwirrend sein, da man bei einem Setter keinen Rückgabewert erwartet. Dies ist hier auch der Fall, da return nichts zurückgibt. Somit haben wir wieder alle Angaben der Methode zurückübersetzt und erhalten folgenden Quellcode:

```
public void setTunen(InterfaceTunen tunen) {
                this.tunen = tunen;
        }
```

Da es sich bei den Methoden „getLackieren“, „setLackieren“, „getHersteller“ und „setHersteller“ ebenfalls um Getter und Setter Methoden handelt, werden diese nicht erklärt. Das Prinzip ist identisch und daher trivial. Lediglich die letzten beiden Methoden sind keine Getter bzw. Setter, daher werden diese wieder ausführlicher erklärt.

```
public void tunen();
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: getfield      #20                 // Field tunen:LpatternAuto/InterfaceTunen;
       4: invokeinterface #44,  1           // InterfaceMethod patternAuto/InterfaceTunen.tunen:()V
       9: return
    LineNumberTable:
      line 49: 0
      line 50: 9
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  this   LpatternAuto/Auto;
```

Abbildung 16: Ausgabe des Java Dekompilers für die Methode tunen()

Die meisten Elemente sind bereits bekannt, als neue Anweisung kommt „invokeinterface“ hinzu. Diese Anweisung ruft eine Interfacemethode auf (Methode, die in einem Interface deklariert ist). Der Hinweis auf den „constant pool“ #44 liefert den Namen und den Typ der aufgerufenen Methode. Hier das Interface „InterfaceTunen“ und die Methode „tunen“.
Der Quellcode lautet:

```
public void tunen() {
                tunen.tunen();
        }
```

```
public void lackieren();
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: getfield      #25                 // Field lackieren:LpatternAuto/InterfaceLakieren;
       4: invokeinterface #48,  1           // InterfaceMethod patternAuto/InterfaceLakieren.lackieren:()V
       9: return
    LineNumberTable:
      line 53: 0
      line 54: 9
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  this   LpatternAuto/Auto;
```

Abbildung 17: Ausgabe des Java Dekompilers für die Methode lackieren()

Die Methode kann analog zum vorherigen Beispiel zurückübersetzt werden.

```
public void lackieren() {
                lackieren.lackieren();
        }
```

Somit ist die Klasse „Auto“ vollständig in Quellcode zurückübersetzt.

3.6 Fazit und Ausblick

In den vorstehenden Abschnitten konnte gezeigt werden, dass Reverse Engineering des Java Bytecodes mit Hilfe des Java Dekompilers grundsätzlich möglich ist. Allerdings scheint der manuelle Aufwand sehr groß. Daher macht das Verfahren nur für bestimmte Codeabschnitte Sinn. Möchte man jedoch gewisse Techniken wiederverwenden, deren Dokumentation bzw. Quellcode fehlt, so kann sich diese Art des Reverse Engineering als sehr nützlich erweisen. Oft reicht es bereits, wenn man nur Teile des Bytecodes (z. B. Methodennamen) rückübersetzt und dann mit Hilfe des Internets bereits implementierten frei zugänglichen Quellcode sucht. Die Fundstücke kann man ohne Einschränkungen anpassen und wiederverwenden. Diese Art des Reverse Engineering findet auf einer sehr niedrigen Abstraktionsebene statt, je nach Tiefe der Ebene kommen verschiedene Techniken und/oder Werkzeuge zum Einsatz.

4 Anforderungen an die Reverse Engineering Werkzeuge

Das Kapitel behandelt notwendige Anforderungen, die an die Reverse Engineering Werkzeuge zu stellen sind. Anschließend wird ein Kriterienkatalog abgeleitet, der als Maßstab für die Auswahl und Bewertung zu analysierender Werkzeuge dienen soll.

4.1 Allgemeine Anforderungen

Die in der Literatur verfügbaren Informationen zum Reverse Engineering beschränken sich hier auf Quellen zu solchen Werkzeugen, die Teil der Analyse eines Fallbeispiels sein werden, d.h. die aus Quellcode UML-Modelle generieren können. Darüber hinaus legt der Verfasser eigene Anforderungen an die Werkzeuge fest, damit diese definierten Standards entsprechen können und somit vergleichbar sind.

Brügge schreibt, dass Reverse Engineering eine UML-Klasse für jede Klassendeklaration erstellen muss und für jedes Feld/Attribut und jede Methode ein UML-Attribut und eine UML-Operation ins Modell einzufügen ist (vergl. BRÜG, S.424).

Fowler erwartet vom Reverse Engineering, dass es Abhängigkeiten erkennen muss, weil sich diese einfach aus dem Code ermitteln lassen und im Forward Engineering Prozess meist nicht vollständig berücksichtigt werden. Des Weiteren sollten sich detaillierte Diagramme zu den wesentlichen Teilen eines Systems generieren lassen können.
Er findet es besonders praktisch ein Sequenzdiagramm zu generieren, in dem mehrere Objekte in einer komplexen Methode zusammenarbeiten (vergl. FOWL, S. 51f).

Larman fordert das Reverse Engineering von Paketdiagrammen, um die im Forward Engineering Prozess erstellten Diagramme aktuell zu halten.
Ebenfalls hält er das Reverse Engineering von Interaktionsdiagrammen zur Unterstützung agiler Vorgehensmodelle für unverzichtbar.
Daneben ist ihm die Integration in eine populäre Entwicklungsumgebung (Eclipse, Netbeans, Visual Paradigm) wichtig (vergl. LARM, S.232 u. S. 409).

Vinita, A. Jain & D. K. Tayal erstellten ein Regelwerk für Reverse Engineering-Software. Sie fordern darin die Erkennung von Sichtbarkeit, Multiplizität und Parametern von Klassen, sowie die Identifikation von Gruppen (Namespaces für Paketdiagramme), Kommentare, Objekte (Klasseninstanzen für Objektstrukturdiagramme), Realisierungen von Interfaces, Assoziationen, Aggregationen und Kompositionen (vergl. VINI).

Die vom Autor dieser Arbeit selbst gestellten Anforderungen lassen sich wie folgt beschreiben:
Das Werkzeug muss mindestens ein Klassendiagramm und optional ein Sequenzdiagramm die dem UML 2.0 Standard entsprechen erzeugen können, sich in die IDE Eclipse integrieren lassen und Java Quellcode in UML Modelle (Klassen- und Sequenzdiagramm) wandeln können.

4.2 Kriterienkatalog

Aus den im vorherigen Kapitel zusammengestellten Anforderungen konnte jetzt ein Kriterienkatalog abgeleitet werden. Aufgrund der relativ allgemeinen Vorgaben in der Literatur, war es notwendig, den Katalog zu spezialisieren. Das bedeutet vor allem eine Anpassung an die Java typischen Sprachkonstrukte.
Die einzelnen Kriterien wurden gemäß ihrer Bedeutung für den Prozess der grafischen Entwicklung des UML-Modells gewichtet. Die vom Verfasser in Kapitel 4.1 selbst gestellten Anforderungen sind daher nicht Teil des Kriterienkatalogs, sie bilden vielmehr die Grundvoraussetzung für die Auswahl der Tools.

Zwei Kriterienkataloge (1x Strukturdiagramm und 1x Verhaltensdiagramm), je einer für Klassendiagramme und einer für Sequenzdiagramme.

Kriterium	**Gewichtung**
Klassen (vollständig)	3
Abstrakte Klassen	2
Superklassen & Subklassen	3
Innere Klassen	2
Anonyme Klassen	1
Interfaces	2
Annotationen	1
Sichtbarkeit	3
Konstruktor & überladener Konstruktor	3
Javadoc (Klassen-Kommentar)	1
Attribute (vollständig)	3
Datentyp	3

Kriterium	Gewichtung
Sichtbarkeit	3
Javadoc (Attribut-Kommentar)	1
Methoden (vollständig)	3
Signatur	3
Sichtbarkeit	3
Überschriebene & überladene Methoden	1
Interface-Methoden	1
Klassenmethoden	2
Javadoc (Methoden-Kommentar)	1
Generics	3
Exceptions	2
Assoziationen	3
Multiplizität	3

Tabelle 5: Kriterienkatalog Klassendiagramm

Kriterium	Gewichtung
Kopfbereich (Klasse)	2
Kommunikationspartner (Objekte)	3
Operationsaufrufe (Methode) ggf. mit Antwort	3
Anzeige der zeitlichen Ordnung der Ereignisse	1

Tabelle 6: Kriterienkatalog Sequenzdiagramm

5 Fallbeispiel

Das nachstehende Kapitel behandelt ein eigenes Fallbeispiel, für das ein Design Pattern, das sogenannten Strategy Pattern, gewählt wurde. Zunächst werden das Pattern allgemein und die Umsetzung vorgestellt. Anschließend werden die Werkzeuge auf der Grundlage dieser Pattern-Umsetzung analysiert und bewertet. Ein Teilausschnitt des Fallbeispiels fand bereits in Kapitel 3 Verwendung. Es musste jedoch für die anschließende Analyse um diverse Java Sprach Konstrukte erweitert werden, um die Werkzeuge möglichst vollständig prüfen zu können.

5.1 Strategy Pattern

Das Strategy Pattern gehört zu den sogenannten Gang of Four-Entwurfsmustern. Es ist ein Verhaltensmuster und definiert eine Familie austauschbarer Algorithmen (vergl. GAMM).

Eine ausgelagerte Strategieklasse (Strategie = gekapselter Algorithmus) kapselt das Verhalten (Funktionalität, Algorithmus) eines Objektes (dem Context).
Zur Ausführung des ausgelagerten Verhaltens delegiert der Context den Aufruf an sein referenziertes Strategieobjekt. Es erfolgt keine konkrete Implementierung im Context, er arbeitet stattdessen mit einer Schnittstelle. Dadurch wird der Context von der Implementierung unabhängig und das Verhalten kann jederzeit geändert bzw. ein neues hinzugefügt werden.
Der Context muss lediglich die jeweils neue Strategie des Strategyinterfaces korrekt implementieren (vergl. WEB2).

Anwendungsbeispiele lassen sich in der Java Bibliothek wiederfinden. Hier sind die Delegierung des Layouts von AWT-Komponenten an entsprechende LayoutManager (BorderLayout, FlowLayout, etc.) zu nennen.

5.2 Umsetzung

Als Beispiel für die Umsetzung Strategy Pattern dienten Fahrzeuge in einer fiktiven Werkstatt. Das Lackieren und das Tunen eines Fahrzeuges stellen hier das Verhalten dar. Dazu wurde je eine eigene Interfaceklasse definiert. Der konkrete Algorithmus wurde in sechs Klassen (jeweils drei für „Tunen“ und drei für „Lackieren“) ausgelagert, die die Interfaceklassen

implementieren. Den Context bildet die Autoklasse, diese hält je eine Referenz auf ihr Strategieobjekt, in diesem Fall ein Objekt des Interface „InterfaceLackieren“ und des Interface „InterfaceTunen“. Ein neues Verhalten kann so jederzeit einfach über ein neues Interface hinzugefügt oder ein bestehendes verändert werden ohne Einfluss auf die Implementierung der Contextklasse (Auto.java) zu nehmen. Die Autoklasse ist abstrakt, die konkreten Autoobjekte werden wiederum in jeweils drei Subklassen von der Autoklasse erzeugt. Diese erben von der Autoklasse (Superklasse).

Die Abbildung 18 zeigt die vollständige Umsetzung, abgebildet als Klassendiagramm in UML.

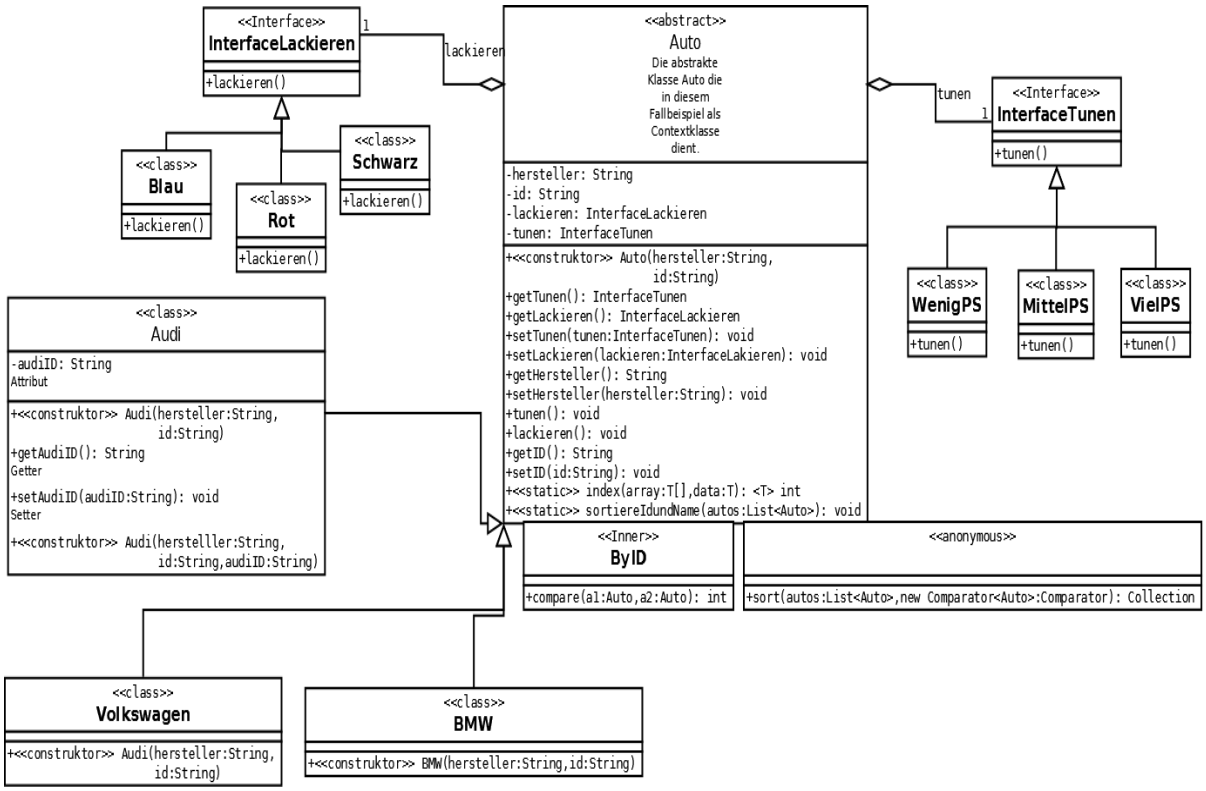

Abbildung 18: UML Klassendiagramm Fallbeispiel Strategy Pattern als Mustervorlage

Der vollständige Quellcode ist im Anhang bzw. auf der Begleit-CD zu finden.

6 Bewertung der Tools

Im folgenden Kapitel werden die ausgewählten Werkzeuge vorgestellt und anhand des entsprechenden Kriterienkatalogs bewertet (s. Abschnitt 4.2, Tabelle 5 & Tabelle 6). Für die Auswahl hatten die Werkzeuge die vom Verfasser festgelegten Grundvoraussetzungen zu erfüllen. Das jeweilige Werkzeug musste sich in die Entwicklungsumgebung Eclipse integrieren lassen, UML 2.0 Diagramme (Klassendiagramm und Sequenzdiagramm) erstellen können und die Programmiersprache Java unterstützen. Alle folgenden Werkzeuge haben diese Grundvoraussetzungen erfüllt. Grundlage für die Recherche der Werkzeuge lieferte der Eclipse Marketplace, mit einer großen Auswahl an Werkzeugen für unterschiedliche Einsatzzwecke.

6.1 Soyatec eUML2

Soyatec eUML2 ist ein Eclipse Plug-In (WEB3). Es gibt zwei verschiedene Versionen. Eine „Free Version“ und eine „Studio Version“. Die „Free Version“ wurde nicht weiterentwickelt und unterstützt auch nicht die Generierung von Sequenzdiagrammen. Die „Studio Version“ ist ebenfalls kostenlos ausführbar, daher erfolgte die Analyse mit der „Studio Version“.

Die Untersuchung wurde mit Eclipse Version 4.4 (Luna) und Soyatec eUML2 Studio Version durchgeführt. Das Werkzeug kann über den in Eclipse integrierten Software Installationsprozess der Entwicklungsumgebung hinzugefügt werden. Das Erstellen von UML Klassen- und Sequenzdiagrammen lässt sich über die Menüoberfläche von Eclipse problemlos ausführen. Das Tool unterstützt Roundtrip-Engineering, d.h. Änderungen am Quellcode werden direkt in die erstellten Diagramme übernommen. Die Synchronisation erfolgt auch bei Änderungen am Diagramm in den Quellcode. Vor Erstellung des Diagramms kann über diverse Einstellungen bestimmt werden was alles angezeigt werden soll. So legt der Anwender fest, ob er für die Anzeige des Diagramms „Association“, „Inheritance“ und/oder „Dependency“ auswählen möchte. Der Detailgrad kann also frei vom User bestimmt werden. Eine weitere Einstellung vermag man bei der Auswahl der anzuzeigenden Klassen treffen: Es können alle Klassen oder nur bestimmte Klassen eines Packages ausgewählt werden. Die Erstellung des kompletten Diagramms erfolgt anschließend automatisch per Knopfdruck.

6.1.1 Klassendiagramm

Das Tool hat erfolgreich ein Klassendiagramm (s. Abbildung 18) erstellt. Im Folgenden werden die einzelnen Punkte des Kriterienkataloges, wie in den Tabellen 7, 8, usw. dargestellt, auf Erfüllung untersucht.

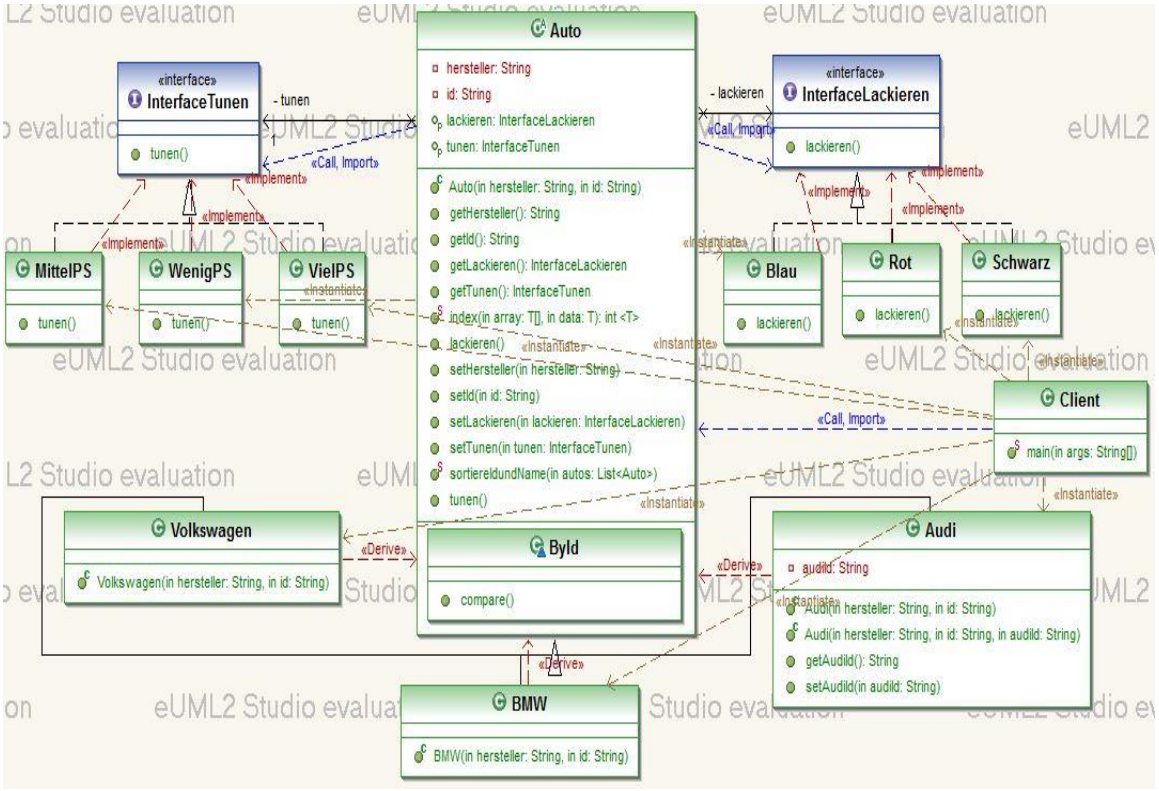

Abbildung 19: Soyatec eUML2 Klassendiagramm

Klassen

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Klassen	Ja	Alle Klassen des Fallbeispiels mit korrekt wiedergegebenen Klassennamen im Diagramm enthalten.
Abstrakte Klassen	Ja	Gekennzeichnet durch ein „A“ Symbol
Super-& Subklassen	Ja	Grafisch verbunden durch entsprechendes Pfeilsymbol.
Innere Klassen	Ja	Grafische Darstellung innerhalb der umgebenen Klasse (blaues Dreieck), korrekte Benennung der inneren Methoden.
Anonyme Klasse	Nein	Die im Fallbeispiel umgesetzte Implementierung des Comparators als anonyme Klasse wird nicht als solche Klasse erkannt, Kennzeichnung erfolgt lediglich als Methode.
Interfaces	Ja	Implementierte Interfaces durch „I“-Symbol gekennzeichnet.
Annotationen	Nein	Keine grafische Umsetzung.
Sichtbarkeit	Ja	Grafische Kennzeichnung: grüner Punkt, wenn „public“, rotes Viereck, wenn „private“.
Konstruktor	Teilweise	Definierter Konstruktor mit „C“-Symbol markiert. (Keine Unterscheidung „überladener“/ „nicht überladener“ Konstruktor)
Javadoc (Klassen-Kommentare)	Teilweise	Im Diagramm keine direkte Darstellung, abzurufen im Eigenschaftendialog der Klasse (vergl. Abb. 19).

Tabelle 7: Kriterienkatalog zum Soyatec eUML2-Klassendiagramm

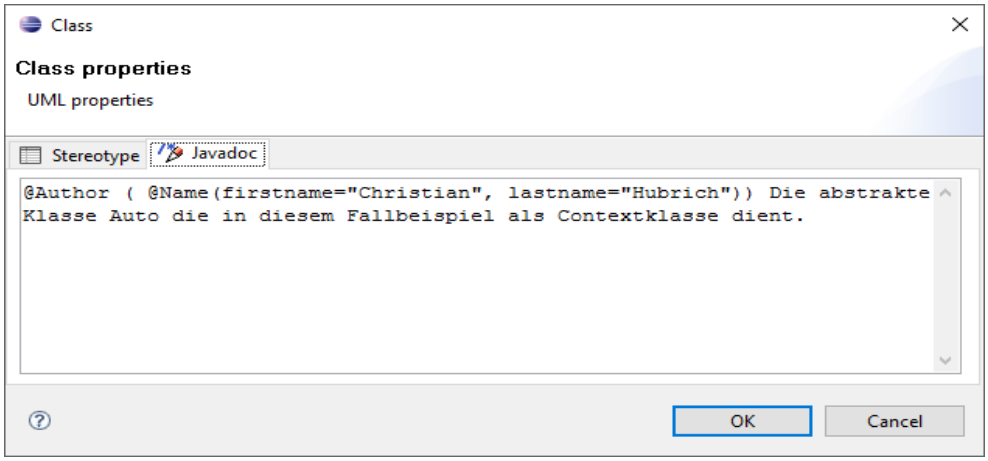

Abbildung 20: Javadoc-Kommentar im Eigenschaftendialog

Attribute

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 8) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Attribute	Ja	Korrekte Benennung
Datentyp	Ja	Korrekte Benennung, ggf. vorliegende Initialisierungswerte nicht grafisch dargestellt, Anzeige über Eigenschaftendialog möglich (vergl. Abb. 20).
Sichtbarkeit	Ja	Anzeige analog zur Klassensichtbarkeit: grüner Kreis für „public“, rotes Viereck für „private“.
Javadoc (Attribut-Kommentare)	Teilweise	Anzeige über Eigenschaftendialog

Tabelle 8: Attribut-Katalog zum Soyatec eUML2-Fallbeispiel

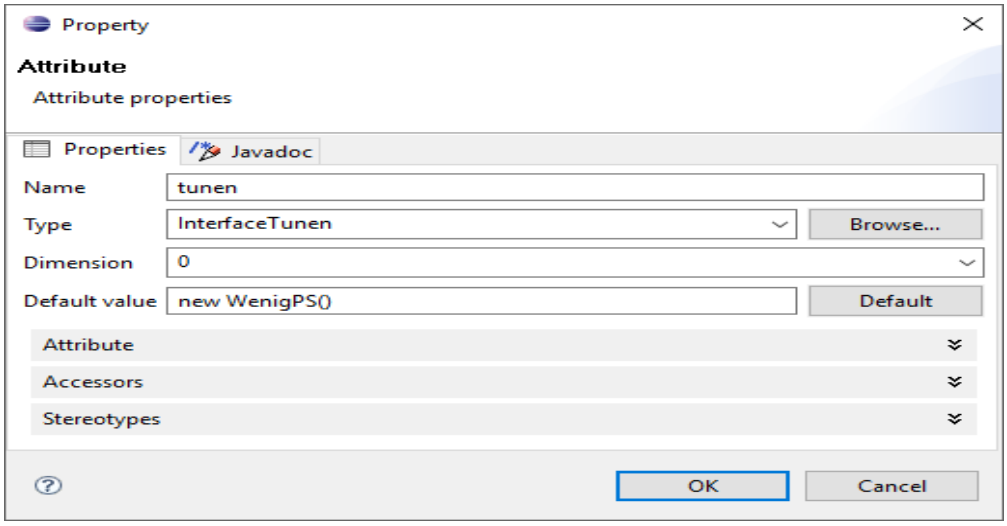

Abbildung 21: Soyatec Eigenschaftsdialog für Attribute

Methoden

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 9) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Methoden	Ja	Vollständig.
Signatur	Ja	Anzeige in der Grafik von: „Name“, „Rückgabewert“ und „Parameter“.
Sichtbarkeit	Ja	Entsprechend „Klassen“ bzw. „Attributen“ (Tabelle 7 bzw. 8).
Überschriebene bzw. überladene Methoden	Nein	Nicht gekennzeichnet.
Implementierte Interface Methoden	Nein	Nicht gekennzeichnet.
Klassen-Methoden	Ja	Mit „S“-Symbol gekennzeichnet
Javadoc (Methoden-Kommentare)	Teilweise	(Analog zu Klassen- bzw. Attributkommentaren, s.o.)

Tabelle 9: Methoden-Katalog zum Soyatec eUML2-Fallbeispiel

Generics, Exceptions, Assoziationen und Multiplizität

Die Ergebnisse dieser Kriterien werden im nachstehenden Katalog (Tabelle 10) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Generics	Ja	Generische Eigenschaften erkannt und angegeben.
Exceptions	Teilweise	Erkennung nur zum Teil: Wenn bereits in der Signatur der Methode angegeben, wird sie in den Eigenschaften unter dem Reiter „Exception“ aufgeführt. Wenn in der Semantik der Methode integriert ist, wird sie nicht erkannt.
Assoziationen	Ja	Vollständig.
Multiplizität		Vollständige, korrekte Angabe.

Tabelle 10: Katalog zu Generics, Exceptions, Assziationen und Multiplizität zum Soyatec eUML2-Fallbeispiel

6.1.2 Sequenzdiagramm

Es wurde erfolgreich ein Sequenzdiagramm erstellt. Die Bewertung erfolgt analog zur Bewertung des Klassendiagramms.

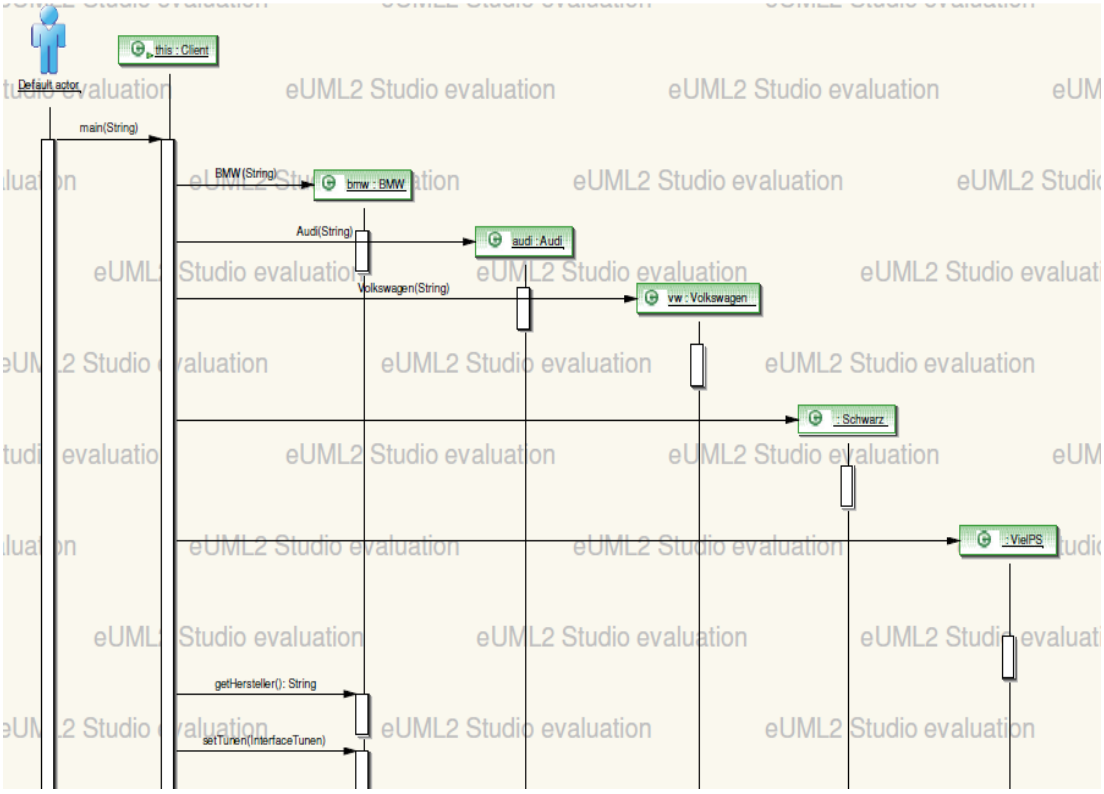

Abbildung 22: Ausschnitt aus dem Soyatec eUML2 Sequenzdiagramm

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Kopfbereich (Klasse)	Nein	Klassenname für generierte Methode (hier: „main()“) wird nicht angezeigt.
Kommunikationspartner (Objekte)	Ja	Alle aufgeführt und richtig benannt.
Operationsaufrufe (Methode), ggf. mit Antwort	Ja	Alle Methodenaufrufe korrekt.
Anzeige der der zeitlichen Ordnung der Ereignisse	Ja	Anzeige der „Sequenz Numbers“ (= zeitlicher Ablauf der Ereignisse)in den Eigenschaften

Tabelle 11: Sequenzdiagramm-Katalog zum Soyatec eUML2 - Fallbeispiel

6.2 ObjectAid UML

Der ObjectAid UML Explorer ist ein Eclipse Plug-In (WEB5). Es gibt zwei verschiedene Versionen. Eine „Free Version“ und eine kommerzielle Version. Die „Free Version“ unterstützt leider nicht die Generierung von Sequenzdiagrammen. Die Analyse erfolgt mit der „Free Version“.

Die Analyse erfolgt wieder mit der Eclipse Version 4.4 Luna. Die Installation über den in Eclipse integrierten Installationsprozess funktioniert ohne Probleme. Die Erstellung des Klassendiagramms geschieht über das Menü. Roundtrip-Engineering (Änderungen am Quellcode) werden unterstützt. Der Anwender hat vor Erstellung des Diagramms die Möglichkeit verschiedene Einstellungen für die angezeigte Detailtiefe festzulegen. Die Auswahl ist in drei Bereiche unterteilt. Diese sind „Classifiers“, „Relationships“ und „Attributes“. Des Weiteren kann der Anwender das Format (PNG, JPEG, GIF) für ein Image auswählen. Abbildung 22 zeigt den Auswahldialog.

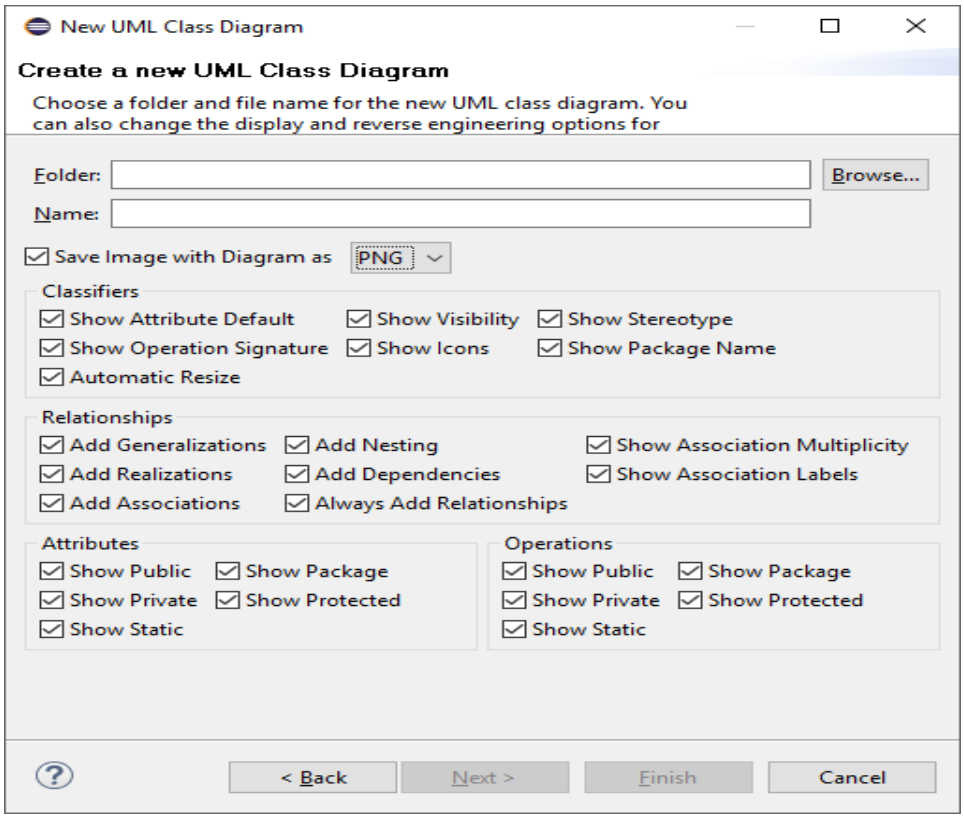

Abbildung 23: Dialogfenster für die Erstellung eines ObjectAid Klassendiagramms

Anschließend wird ein leeres Fenster erstellt. Der Anwender kann die gewünschten Klassen per Drag & Drop in das Fenster ziehen. Das Diagramm baut sich daraufhin auf.

6.2.1 Klassendiagramm

Ein Klassendiagramm wurde erfolgreich erstellt. Die Analyse des Werkzeugs gemäß des Kriterienkatalogs erfolgt analog zur vorherigen Untersuchung.

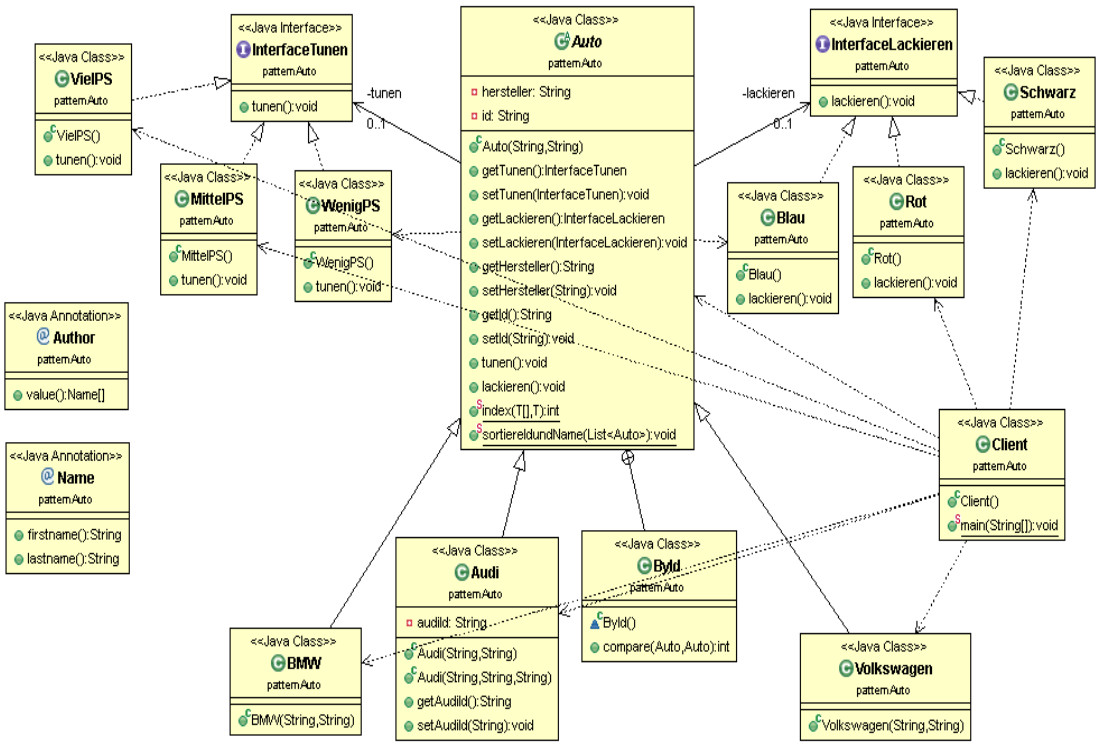

Abbildung 24: ObjectAid UML Explorer Klassendiagram

Klassen

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 12) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Klassen	Ja	Alle Klassen des Fallbeispiels mit korrekt wiedergegebenen. Klassennamen im Diagramm enthalten. Vergabe entsprechender Symbol („C“=Klasse) und Bezeichnung (<< Java Class>>)
Abstrakte Klassen	Ja	Erweitert um ein „A“ Symbol
Super-& Subklassen	Ja	Grafisch verbunden durch entsprechendes Pfeilsymbol.
Innere Klassen	Ja	Grafische Darstellung wie „normale“ Klassen, aber Erweiterung des Konstruktors um blaues Dreieck, Methode wird erkannt und benannt.
Anonyme Klasse	Nein	Im Fallbeispiel umgesetzte Implementierung des Comparators als anonyme Klasse nicht als solche Klasse erkannt, Kennzeichnung lediglich als Methode.
Interfaces	Ja	Implementierte Interfaces durch „I“-Symbol gekennzeichnet. Bezeichnung << Java Interface>>.
Annotationen	Ja	Annotationen durch „@“-Symbol gekennzeichnet. Bezeichnung <<Java Annotation>>.
Sichtbarkeit	Ja	Grafische Kennzeichnung: grüner Punkt, wenn „public“, rotes Viereck, wenn „private“.
Konstruktor	Ja	Definierter Konstruktor mit „C“-Symbol markiert. Unterscheidung „überladener“/ „nicht überladener“ Konstruktor anhand der Parameter möglich.
Javadoc (Klassen-Kommentare)	Nein	Nicht angezeigt und nicht im Diagramm abrufbar.

Tabelle 12: Kriterienkatalog zum ObjectAid UML-Klassendiagramm

Attribute

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 13) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Attribute	Teilweise	Unvollständig, Attribute vom Datentyp Interface werden nicht erkannt.
Datentyp	Ja	Erkannte Attribute werden mit korrektem Datentyp angegeben, ggf. vorliegende Initialisierungswerte nicht grafisch dargestellt und können nicht abgerufen werden.
Sichtbarkeit	Ja	Anzeige analog zur Klassensichtbarkeit: grüner Kreis für „public“, rotes Viereck für „private“.
Javadoc- (Attribut-Kommentare)	Nein	Nicht angezeigt und nicht im Diagramm abrufbar.

Tabelle 13: Attribut-Katalog zum ObjectAid UML-Fallbeispiel

Methoden

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 14) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Methoden	Ja	Vollständig.
Signatur	Ja	Anzeige in der Grafik von: „Name“, „Rückgabewert“ und „Parameter“.
Sichtbarkeit	Ja	Entsprechend „Klassen“ bzw. „Attributen“ (Tabelle 12 bzw. 13).
Überschriebene bzw. überladene Methoden	Nein	Nicht gekennzeichnet.
Implementierte Interface Methoden	Nein	Nicht gekennzeichnet.
Klassen-Methoden	Ja	Mit „S“-Symbol gekennzeichnet
Javadoc- (Methoden-Kommentare	Nein	Nicht angezeigt und nicht im Diagramm abrufbar.

Tabelle 14: Methoden-Katalog zum ObjectAid UML-Fallbeispiel

Generics, Exceptions, Assoziationen und Multiplizität

Die Ergebnisse dieser Kriterien werden im nachstehenden Katalog (Tabelle 10) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Generics	Ja	Generische Eigenschaften erkannt und angegeben.
Exceptions	Nein	Keine Erkennung und Darstellung im Diagramm
Assoziationen	Ja	Vollständig.
Multiplizität	Ja	Vollständige, korrekte Angabe.

Tabelle 15: Katalog zu Generics, Exceptions, Assziationen und Multiplizität zum ObjectAid UML-Fallbeispiel

6.2.2 Sequenzdiagramm

Ein Sequenzdiagramm ist im Funktionsumfang der Freeversion nicht enthalten, daher wird es für dieses Tool nicht bewertet.

6.3 ModelGoon UML4Java

ModelGoon UML4Java ist ein Eclipse Plug-In (WEB6). Es ist als „Free Version“ erhältlich. Die aktuelle Version des Plug-Ins ist 4.4.1.

Die Untersuchung wurde mit Eclipse Version 4.4 (Luna) und ModelGoon UML4Java Version 4.4.1 durchgeführt. Das Werkzeug kann über den in Eclipse integrierten Software Installationsprozess der Entwicklungsumgebung hinzugefügt werden. Das Erstellen von UML Klassen- und Sequenzdiagrammen lässt sich über die Menüoberfläche von Eclipse problemlos ausführen. Das Tool unterstützt kein Roundtrip-Engineering. Das Tool bietet keine weiteren Einstellmöglichkeiten vor der Erstellung des Diagramms.
Es wird ein leeres Fenster erstellt. Der Anwender kann die gewünschten Klassen per Drag & Drop in das Fenster ziehen. Das Diagramm baut sich daraufhin auf. Anschließend kann der Anwender durch Rechtsklick auf die jeweilige Klasse den Detailgrad der Anzeige für Attribute und Methoden bestimmten (s. Abbildung 25).

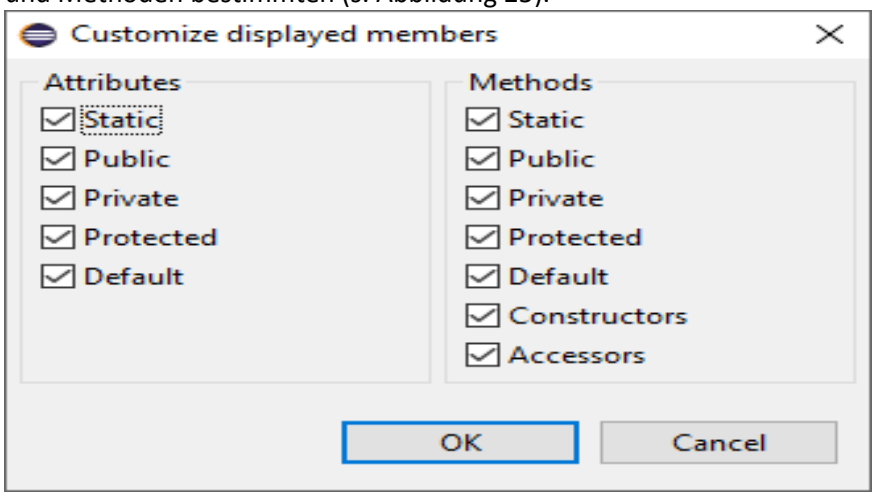

Abbildung 25: Dialog für Filterelemente des Tools ModelGoon UML4Java

6.3.1 Klassendiagramm

Ein Klassendiagramm wurde erfolgreich erstellt. Die Analyse des Werkzeugs gemäß des Kriterienkatalogs erfolgt analog zur vorherigen Untersuchung.

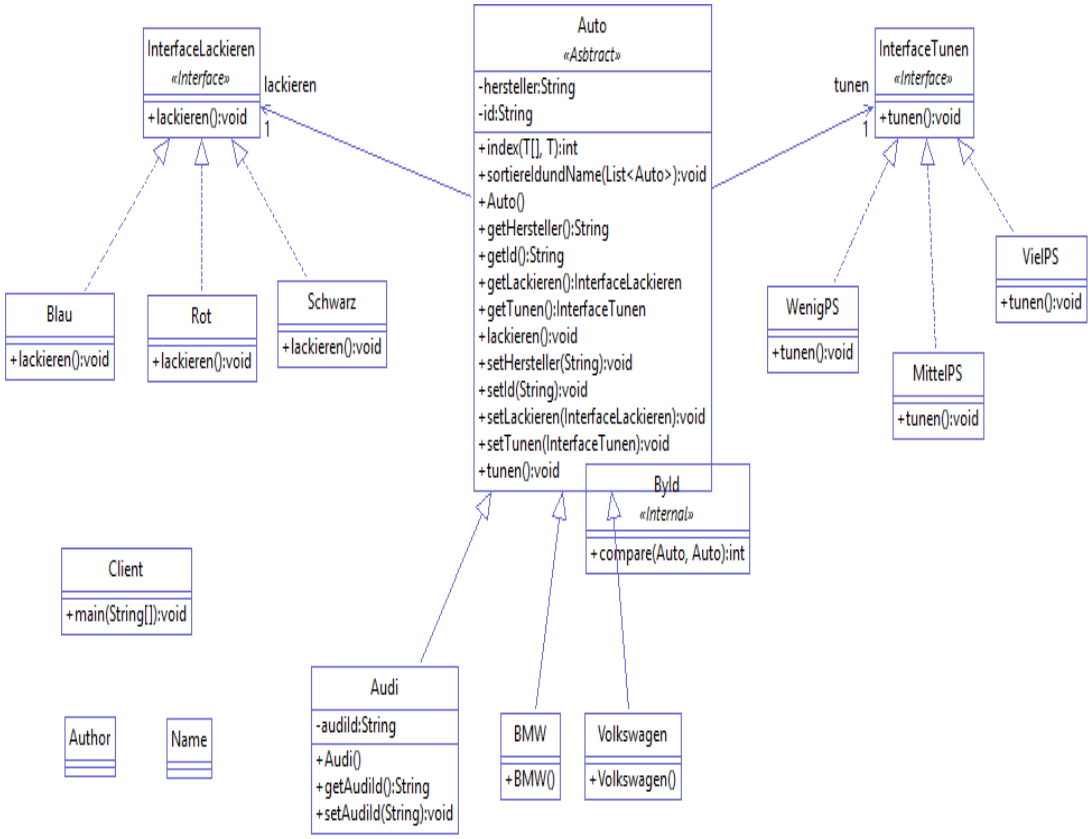

Abbildung 26: ModelGoonUML4Java Klassendiagramm

Klassen

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 16) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Klassen	Ja	Alle Klassen des Fallbeispiels mit korrekt wiedergegebenen. Klassennamen im Diagramm enthalten.
Abstrakte Klassen	Ja	Erweitert um eine Bezeichnung «Abstract».
Super-& Subklassen	Ja	Grafisch verbunden durch entsprechendes Pfeilsymbol.
Innere Klassen	Ja	Erweitert um eine Bezeichnung «Internal».
Anonyme Klasse	Nein	Im Fallbeispiel umgesetzte Implementierung des Comparators als anonyme Klasse nicht als solche Klasse erkannt, Kennzeichnung lediglich als Methode.
Interfaces	Ja	Erweitert um eine Bezeichnung «Interface».
Annotationen	Teilweise	Annotationen werden als Klasse erkannt, jedoch keine zusätzliche Kennzeichnung als Annotation.
Sichtbarkeit	Nein	Keine Kennzeichnung der Sichtbarkeit bei Klassen
Konstruktor	Teilweise	Definierter Konstruktor wird erkannt. Überladener Konstruktor wird nicht erkannt.
Javadoc (Klassen-Kommentare)	Nein	Nicht angezeigt und nicht im Diagramm abrufbar.

Tabelle 16: Kriterienkatalog zum ModelGoon UML4Java-Klassendiagramm

Attribute

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 17) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Attribute	Teilweise	Unvollständig, Attribute vom Datentyp Interface werden nicht erkannt.
Datentyp	Ja	Erkannte Attribute werden mit korrektem Datentyp angegeben, ggf. vorliegende Initialisierungswerte nicht grafisch dargestellt und können nicht abgerufen werden.
Sichtbarkeit	Ja	Grafische Kennzeichnung: „+“, wenn „public“, „-“, wenn „private“.
Javadoc (Attribut-Kommentare)	Nein	Nicht angezeigt und nicht im Diagramm abrufbar.

Tabelle 17: Attribut-Kriterienkatalog zum ModelGoon UML4Java-Klassendiagramm

Methoden

Die Ergebnisse werden im nachstehenden Katalog (Tabelle 18) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Methoden	Ja	Vollständig.
Signatur	Ja	Anzeige in der Grafik von: „Name“, „Rückgabewert“ und „Parameter“.
Sichtbarkeit	Ja	Entsprechend „Attributen“ (Tabelle 17).
Überschriebene bzw. überladene Methoden	Nein	Nicht gekennzeichnet.
Implementierte Interface Methoden	Nein	Nicht gekennzeichnet.
Klassen-Methoden	Nein	Nicht gekennzeichnet.
Javadoc (Methoden-Kommentare)	Nein	Nicht angezeigt und nicht im Diagramm abrufbar.

Tabelle 18: Methoden-Kriterienkatalog zum ModelGoon UML4Java-Klassendiagramm

Generics, Exceptions, Assoziationen und Multiplizität

Die Ergebnisse dieser Kriterien werden im nachstehenden Katalog (Tabelle 19) dargestellt.

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Generics	Ja	Generische Eigenschaften erkannt und angegeben.
Exceptions	Nein	Keine Erkennung und Darstellung im Diagramm
Assoziationen	Ja	Vollständig.
Multiplizität	Ja	Vollständige, korrekte Angabe.

Tabelle 19: Kriterienkatalog zum ModelGoon UML4Java-Klassendiagramm

6.3.2 Sequenzdiagramm

Es wurde erfolgreich ein Sequenzdiagramm erstellt. Die Bewertung erfolgt analog zur Bewertung des Klassendiagramms.

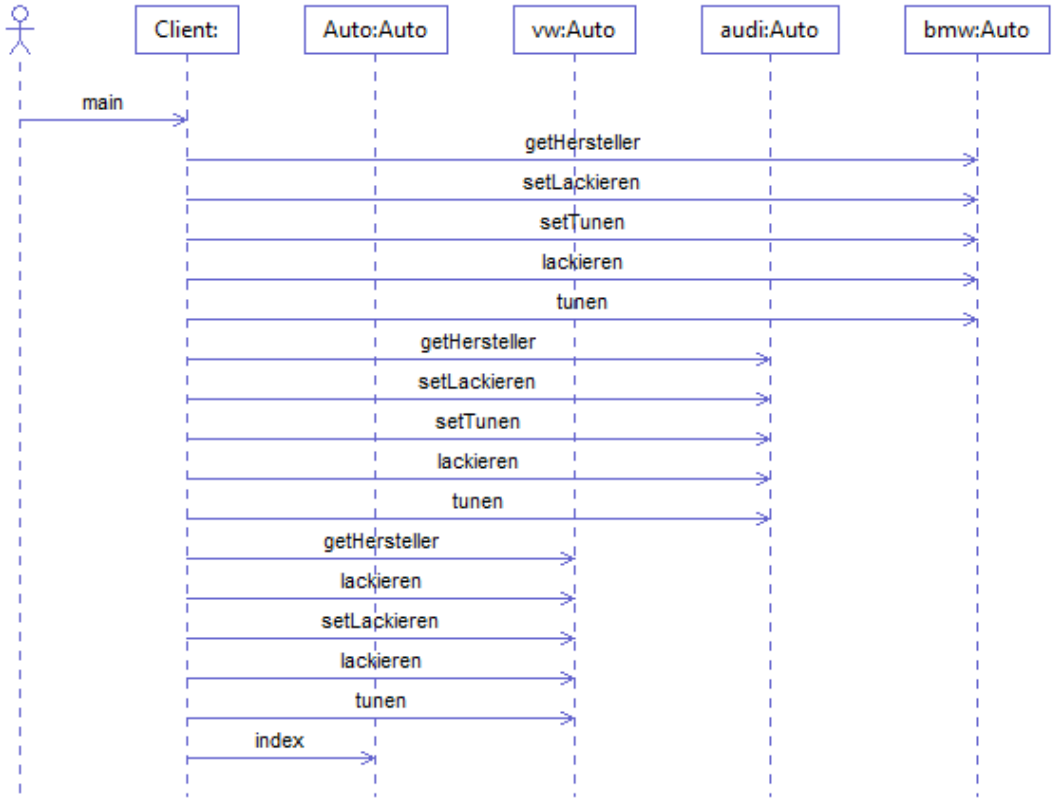

Abbildung 27: ModelGoon UML4Java Sequenzdiagramm

Kriterium	*Erfüllung (Erkennung)*	*Kommentar*
Kopfbereich (Klasse)	Nein	Klassenname für generierte Methode (hier: „main()“) wird nicht angezeigt.
Kommunikationspartner (Objekte)	Ja	Alle aufgeführt und richtig benannt.
Operationsaufrufe (Methode), ggf. mit Antwort	Ja	Alle Methodenaufrufe korrekt.
Anzeige der der zeitlichen Ordnung der Ereignisse	Nein	Keine Option für zeitliche Ordnung

Tabelle 20: Sequenzdiagramm-Katalog zum ModelGoon UML4Java-Fallbeispiel

6.4 Überblick der Analyse-Ergebnisse

Dieses Kapitel zeigt eine Zusammenfassung der vorherigen Analysen.

Kriterium	Gewichtung	Soyatec eUML2	ObjectAid UML	ModelGoon UML4Java
Klassen (vollständig)	3	3	3	3
Abstrakte Klassen	2	2	2	2
Superklassen & Subklassen	3	3	3	3
Innere Klassen	2	2	2	2
Anonyme Klassen	1	0	0	0
Interfaces	2	2	2	2
Annotationen	1	0	1	0,5
Sichtbarkeit	3	3	3	0
Konstruktor (inc. überladen)	3	1,5	3	1,5
Javadoc (Klassen)	1	0,5	0	0
Attribute (vollständig)	3	3	1,5	1,5
Datentyp	3	3	3	3
Sichtbarkeit	3	3	3	3
Javadoc (Attribut)	1	0,5	0	0
Methoden (vollständig)	3	3	3	3
Signatur	3	3	3	3
Sichtbarkeit	3	3	3	3
Überschriebene/Überladene Methoden	1	0	0	0
Klassenmethoden	2	2	2	0
Interface-Methoden	1	0	0	0
Javadoc (Methoden)	1	0,5	0	0
Generics	3	3	3	3
Exceptions	2	1	0	0
Assoziationen	3	3	3	3
Multiplizität	3	3	3	3
Gesamt	**56**	**48**	**46,5**	**39,5**

Tabelle 21: Ergebnisse Klassendiagramme

Kriterium	Gewichtung	Soyatec eUML2	ObjectAid UML	ModelGoon UML4Java
Kopfbereich (Klasse)	2	0	-	0
Kommunikationspartner (Objekte)	3	3	-	3
Operationsaufrufe (Methode) ggf. mit Antwort	3	3	-	3
Anzeige der zeitlichen Ordnung der Ereignisse	1	1	-	0
Gesamt	**9**	**7**	**-**	**6**

Tabelle 22: Ergebnisse Sequenzdiagramme

Die Auswertung der Ergebnisse für die Erstellung von Klassendiagrammen hat Folgendes geliefert:

Das Tool von Soyatec eUML2 erzielte mit 48 von 56 möglichen Punkten, knapp vor ObejctAid UML (46,5), das beste Ergebnis. Etwas abgeschlagen auf dem dritten Rang landete ModelGoon UML4Java mit 39,5 Punkten. Es bleibt festzuhalten, dass alle drei Tools zufrieden stellende Ergebnisse brachten, jedoch in der Detailtiefe voneinander abweichen. Die genauen Unterschiede sind in Tabelle 20 einzusehen.

Für die Sequenzdiagrammerstellung konnten nur zwei der drei ausgewählten Tools ein Ergebnis liefern. In der Freeversion des Tools ObjectAid UML ist die Sequenzdiagrammerstellung nicht enthalten. Die beiden übrigen Werkzeuge erstellten erfolgreich ein Diagramm. Als einzigen merklichen Unterschied bot das Tool von ModelGoon UML4Java keine Möglichkeit die zeitliche Ordnung der Ereignisse anzuzeigen. Deshalb erzielt Soyatec eUML2 auch hier ein leicht besseres Ergebnis.

Im Endeffekt stellte Soyatec eUML2 das Tool mit den besten Eigenschaften.

7 Fazit und Ausblick

Nachfolgend werden die Erkenntnisse dieser Arbeit zusammengefasst und weitere mögliche Folgeuntersuchungen diskutiert.

Die eingangs gestellte Fragestellung lautete:

Lässt sich ein Softwareprodukt, das mittels Forward Engineering erstellt wurde, per Reverse Engineering wieder zu seinem Ausgangsmodell zurück übersetzten?

Diese Arbeit konnte in zwei Schritten darauf eine Antwort geben:

Die erste Untersuchung umfasste das manuelle Reverse Engineering von Bytecode in Quellcode mit Hilfe eines Dekompilers. In der zweiten Phase wurden Werkzeuge eingebunden, die aus dem zurück gewonnen Quellcode ein UML Model erstellen sollten. Beide Schritte ergaben positive Antworten im Sinne der obigen Problemstellung. Allerdings ist zu hinterfragen, ob sich der Einsatz der jeweiligen Reverse-Engineering-Methode sinnvoll gestaltet, wenn man den Umfang der Aufgabe berücksichtigt. Betrachtet man das in dieser Arbeit künstlich erstellte Fallbeispiel, ist es natürlich nicht mit einem komplexen Softwareprodukt zu vergleichen. Beide Schritte zeigen hier aber, dass in einem ausgesuchten kleineren Kontext Reverse Engineering von Software sehr gut funktioniert. Die manuelle Rückübersetzung von Java Bytecode findet in der Praxis beispielsweise für bestimmte Code-Fragmente statt, dort wird speziell nach bestimmten Methoden gesucht. Es ist gar nicht nötig den gesamten Code zurück zu gewinnen, oft lassen sich mittels des rückübersetzen Methodennamens entsprechende Beispiele im Internet in OpenSource Projekten finden. Diese können beliebig auf das eigene Projekt angepasst und verwendet werden. Die erstellten Modelle erfüllen die Anforderungen nahezu vollständig. Lediglich sehr spezifische Sprachkonstrukte, der objektorientierten Programmiersprache Java wurden nicht erfüllt. (Als Beispiel sei hier die Anonyme Klasse genannt, die kein Tool erkannt hat). Diese Fälle sind aber für einen Modellüberblick zu vernachlässigen und fallen somit nicht ins Gewicht.

Weitere Arbeiten könnten sich noch mit folgenden Schwerpunkten befassen:

- Muster- und Regelerkennung
- Generierung von weiteren Diagrammtypen (Paketdiagramme, Zustandsdiagramme, Aktivitätsdiagramm, usw.)
- Kompatibilität mit anderen UML-Werkzeugen

- Andere Programmiersprachen (C#, C++, usw.)
- Werkzeuge die automatisch Bytecode in Quellcode dekompilieren

Literaturverzeichnis

ANSI
ANSI/IEEE Std 729-1983
https://standards.ieee.org/findstds/standard/729-1983.html, Abruf: 15.01.2016

BRÜG
Bernd Brügge, Allen H. Dutoit, Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java, Prentice Hall, München, 2004; ISBN 3-8273-7082-5

CHIK
Chikofsky, E. J. und J. H. Cross II: Reverse engineering and design recovery: A taxonomy. IEEE Software, 7:13–17, 1990, ISSN 0740-7459.

CREM
Katia Cremer, Graphbasierte Werkzeuge zum Reverse Engineering und Reengineering, Springer Fachmedien, Wiesbaden 2000, ISBN 978-3-8244-0497-1

FOWL
Fowler, M.: UML konzentriert: eine kompakte Einführung in die Standard Objektmodellierungssprache, Addison- Wesley, München, 3. Aufl., 2004., ISBN 3-8273-2126-3

GAMM
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software, mitp Verlag, 1. Auflage, 2015, ISBN 978-3-8266-9700-5

HITZ
M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger; UML@work - Objektorientierte Modellierung mit UML 2, 3. Auflage, dpunkt, Heidelberg, 2005 ISBN 3-89864-261-5

KÜNN
Thomas Künneth, Einstieg in Eclipse, 5., aktualisierte Auflage, rheinwerk-verlag, 2014; ISBN 978-3-8362-2958-6

LARM
Larman, C.; UML und Patterns angewendet- objektorientierte Softwareentwicklung. München: mitp-Verlag, 2005, ISBN 9783826614538.

RUPP
Chris Rupp; Stefan Queins & die Sophisten: UML 2 glasklar. 4.Auflage. München : Carl Hanser Verlag, 2012.- ISBN 978-3-446-43057-0

ULLE
Christian Ullenboom: Java ist auch eine Insel, Galileo Computing, 11. Auflage 2014, ISBN 3-8362-2873-4

VINI
Vinita, A. Jain und D. K. Tayal: On reverse engineering an objectoriented code into UML class diagrams incorporating extensible mechanisms. SIGSOFT Softw. Eng. Notes, 33:9:1–9:9, 2008, ISSN 0163-5948.

WEB1
http://www.omg.org/spec/UML/2.0/Infrastructure/PDF, Abruf: 20.02.2016

WEB2
http://www.philipphauer.de/study/se/design-pattern/strategy.php, Abruf: 02.02.2016

WEB3
https://en.wikipedia.org/wiki/List_of_JVM_languages, Abruf: 05.01.2016

WEB4
http://www.javaseiten.de/jvminstructionset.html, Abruf: 27.12.2015

WEB5
http://www.soyatec.com/euml2, Abruf: 07.03.2016

WEB6
http://www.objectaid.com, Abruf: 04.04.2016

WEB7
http://www.modelgoon.org, Abruf: 21.04.2016

YURI
Dennis Yurichev: Reverse Engineering for Beginners, Free book http://beginners.re, Abruf: 10.12.2015

I. Abkürzungsverzeichnis

ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
GOTO	Sprungbefehl
IDE	integrated development environment (Entwicklungsumgebung)
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LVA	Local Variable Array
PNG, JPEG, GIF	Bilddateiformate
TOS	Top of the Stack
UML	Unified Modeling Language
UTF	Unicode Transformation Format

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

A. Anhang

Auto.java

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * @Author ( @Name(firstname="Christian", lastname="Hubrich")) Die abstrakte Klasse Auto die in
diesem Fallbeispiel als Contextklasse dient.
 */
public abstract class Auto {

	/**
	*  Instanzvariablen vom Typ des Interfaces. Default-Einstellungen
	*/
	/**
	 * @uml.property name="tunen"
	 * @uml.associationEnd multiplicity="(1 1)"
	 */
	private InterfaceTunen tunen = new WenigPS();
	/**
	 * @uml.property name="lackieren"
	 * @uml.associationEnd multiplicity="(1 1)"
	 */
	private InterfaceLackieren lackieren = new Blau();

	/**
	 * Attribute
	 */
	private String hersteller;
	private String id;

	/**
	 * Konstruktor der Autoklasse
	 * @param String, String
	 */
	public Auto(String hersteller, String id) {
```

```
		this.hersteller = hersteller;
		this.id = id;
	}

	/**
	 * Getter und Setter
	 */
	public InterfaceTunen getTunen() {
		return tunen;
	}

	public void setTunen(InterfaceTunen tunen) {
		if(tunen==null){
			throw new IllegalArgumentException("Null ist ein ungültiger Wert.");
		}
		this.tunen = tunen;
	}

	public InterfaceLackieren getLackieren() {
		return lackieren;
	}

	public void setLackieren(InterfaceLackieren lackieren) {
		if(lackieren==null){
			throw new IllegalArgumentException("Null ist ein ungültiger Wert.");
		}
		this.lackieren = lackieren;
	}

	public String getHersteller() {
		return hersteller;
	}

	public void setHersteller(String hersteller) {
		if(hersteller==null){
			throw new IllegalArgumentException("Null ist ein ungültiger Wert.");
		}
		this.hersteller = hersteller;
	}

	public String getId() {
		return id;
	}

	public void setId(String id) {
		if(id==null){
			throw new IllegalArgumentException("Null ist ein ungültiger Wert.");
```

```
            }
            this.id = id;
}

/**
 * Interface Methoden der Verhaltens-Interfaces:
 * InterfaceLackieren und InterfaceTunen
 */

public void tunen() {
            tunen.tunen();
}

public void lackieren() {
            lackieren.lackieren();
}

/**
* Generische Klassenmethode, gibt den Index eines übergebenen * Elements in einem
übergebenen Array wieder, falls das       * Element nicht im Array ist -1
*/
public static <T> int index(T[] array, T data) {
            for (int i = 0; i < array.length; i++) {
                        if (data.equals(array[i])) {
                                    return i;
                        }
            }
            return -1;
}

/**
  * Innere Klasse zum Vergleichen von Auto Objekten mittels    * Comparator
 */
class ById implements Comparator<Auto> {

            @Override
            public int compare(Auto a1, Auto a2) {
                        return a1.getId().compareTo(a2.getId());
            }

}

/**
  * Klassenmethode mit anonymer Klasse zum Sortieren mittels     * Comparator
 */

public static void sortiereIdundName(List<Auto> autos) {
```

```
                Collections.sort(autos, new Comparator<Auto>() {
                        @Override
                        public int compare(Auto a1, Auto a2) {
                                int byId = a1.getId().compareTo(a2.getId());
                                        return byId != 0 ? byId :
        a1.hersteller.compareTo(a2.hersteller);
                        }
                });
        }
}
```

Audi.java

```
/*
 * @author C. Hubrich
 * Die Klasse Audi erbet von der Kontextklasse Auto.
 */

public class Audi extends Auto {

        /**
         * Attribut
         */
        private String audiId;

        /**
         * Konstruktor geerbt von Auto
         * @param hersteller
         * @param id
         */

        public Audi(String hersteller, String id) {
                super(hersteller, id);
        }

        /**
         * Überladener Konstruktor
         * @param hersteller, id, audiId
         */

        public Audi(String hersteller, String id, String audiId) {
                super(hersteller, id);
                this.audiId = audiId;
        }

        /**
         * Getter und Setter
```

```
	 */

	public String getAudiId() {
		return audiId;
	}

	public void setAudiId(String audiId) {
		if(audiId==null){
			throw new IllegalArgumentException("Null ist ein ungültiger Wert.");
		}
		this.audiId = audiId;
	}
}
```

BMW.java

```
package patternAuto;

/*
 * @author C. Hubrich
 * Die Klasse BMW erbet von der Kontextklasse Auto.
 */

public class BMW extends Auto {

	/**
	 * Konstruktor geerbt von Auto
	 * @param hersteller, id
	 */

	public BMW(String hersteller, String id) {
		super(hersteller, id);
	}
}
```

Volkswagen.java

```
package patternAuto;

/**
 * @author C. Hubrich
 * Die Klasse Volkswagen erbet von der Kontextklasse Auto.
 */

public class Volkswagen extends Auto {
```

```
	/**
	 * Konstruktor geerbt von Auto
	 * @param hersteller, id
	 */

	public Volkswagen(String hersteller, String id) {
		super(hersteller, id);
	}
}
```

InterfaceLackieren.java

```
/*
 * @author C. Hubrich
 * Das Interface "InterfaceLackieren" kapselt das Verhalten der
 * konkreten Strategie lackieren.
 */

public interface InterfaceLackieren {

	/**
	 * Methodenhülle, konkrete Implementierung erfolgt in den
	 * erbenden Klassen.
	 *

	public void lackieren();
}
```

Blau.java

```
/**
 * @author C. Hubrich
 * Die Klasse Blau implementiert das konkrete Verhalten.
 */

public class Blau implements InterfaceLackieren {

	/**
	 * Geerbte Interfacemethode die mit einem konkreten Verhalten
	 * überschrieben ist.
	 */
	@Override
	public void lackieren() {
		System.out.println("Das Auto ist jetzt blau");
	}
```

```
}
```

Rot.java

```
/**
 * @author C. Hubrich
 * Die Klasse Rot implementiert das konkrete Verhalten.
 */

public class Rot implements InterfaceLackieren {

	/**
	 * Geerbte Interfacemethode die mit einem konkreten Verhalten
	 * überschrieben ist.
	 */
	@Override
	public void lackieren() {
		System.out.println("Das Auto ist jetzt rot");
	}
}
```

Schwarz.java

```
/**
 * @author C. Hubrich
 * Die Klasse Schwarz implementiert das konkrete Verhalten.
 */

public class Schwarz implements InterfaceLackieren {

	/**
	 * Geerbte Interfacemethode die mit einem konkreten Verhalten
	 * überschrieben ist.
	 */
	@Override
	public void lackieren() {
		System.out.println("Das Auto ist jetzt schwarz");
	}
}
```

InterfaceTunen.java

```
/**
 * @author C. Hubrich
 * Das Interface "InterfaceTunen" kapselt das Verhalten der
 * konkreten Strategie tunen.
```

```
 */

public interface InterfaceTunen {

	/**
	 * Methodenhülle, konkrete Implementierung erfolgt in den
	 * erbenden Klassen.
	 */

	public void tunen();
}
```

WenigPS.java

```
/**
 * @author C. Hubrich
 * Die Klasse WenigPS implementiert das konkrete Verhalten.
 */

public class WenigPS implements InterfaceTunen {

	/**
	 * Geerbte Interfacemethode die mit einem konkreten Verhalten
	 * überschrieben ist.
	 */
	@Override
	public void tunen() {
		System.out.println("Das Auto hat jetzt 50 PS");
	}
}
```

MittelPS.java

```
/**
 * @author C. Hubrich
 * Die Klasse MittelPS implementiert das konkrete Verhalten.
 */

public class MittelPS implements InterfaceTunen {

	/**
	 * Geerbte Interfacemethode die mit einem konkreten Verhalten
	 * überschrieben ist.
	 */
	@Override
	public void tunen() {
```

```
                System.out.println("Das Auto hat jetzt 100 PS");
        }
}
```

VielPS.java

```
/**
 * @author C. Hubrich
 * Die Klasse VielPS implementiert das konkrete Verhalten.
 */

public class VielPS implements InterfaceTunen {

        /**
         * Geerbte Interfacemethode die mit einem konkreten Verhalten
         * überschrieben ist.
         */
        @Override
        public void tunen() {
                System.out.println("Das Auto hat jetzt 200 PS");
        }
}
```

Author.java

```
/**
 * @author C. Hubrich
 * Annontationsklasse definiert eine eigen @Author Annotation
 */
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})

public @interface Author {
        Name[] value();
}
```

Name.java

```
/**
 * @author C. Hubrich
 * Annotationsklasse, definiert den Namen für die @Author
```

```
 * Annotation.
 */
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface Name {
	String firstname();

	String lastname();
}
```

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____