Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Masterarbeit

Automatisierte Refactorings zur Optimierung nicht-funktionaler Eigenschaften in Software Produktlinien

Verfasser:

Son Hoang Luong

19. September 2011

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl. -Inform. Martin Kuhlemann,
Dipl. -Inform. Norbert Siegmund

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Luong, Hoang Son:
Automatisierte Refactorings zur Optimierung nicht-funktionaler Eigenschaften in Software Produktlinien
Mastermarbeit, Otto-von-Guericke-Universität Magdeburg, 2011.

Abstract

One of the most important concerns of software engineering is to develop software products in a shorter time at lower costs and tailorable for special demands of customers. Software product lines can satisfy these requirements as they are implemented by using reusable software assets to create a variety of related products. The core idea of software product lines is using features to describe different issues of a software domain, which encompass not only the functionalities but also non-functional properties of products. For example: performance, resources consumption, energy saving and maintainability are important non-functional properties that are pursued in most software products. Refactoring is a technique that can be used to optimize qualitative aspects of variants of a software product line in which the internal structure of source code is modified while the external behaviors of the program remain intact.

A refactoring process consists of determining if an application should be refactored, identifying where and which refactoring(s) should be applied and performing the selected refactorings. Most current-day development environments support only the third step and leave the two remaining steps for users, which are non-trivial, labored and error-prone. We try to tackle these problems and concentrate in this thesis on enhancement of the maintainability of products in a software product line. We use a set of software metrics that can predict maintainability of software and analyze the impact for each metric when the corresponding refactorings are applied on a sample application. We found out that refactorings can actually improve the values of these metrics. However, in some cases, only the value of a target class became better while the average value of the program is deteriorated. Using the knowledge collected we implement a prototypical plug-in in Eclipse that uses heuristic strategies based on measurement results of the metrics to identify locations in source code that should be refactored and performs the appropriate refactoring.

We then evaluated the prototype by applying it on variants of two software product lines. We found out that refactoring chances can be discovered by our heuristic strategies although sometimes one may need knowledge of the source code to have the best solution. We observed that the optimization of one metric can lead to degradation of other metrics as these metrics are in some ways related to each other. The possibility of combining different refactorings that enhance different metrics is also investigated. It turned out that applying successive refactorings can actually improve the values of the corresponding metrics. Nevertheless, it may also worsen the values of other metrics. The results of the evaluation also yield some problems for future works such as determining how the sequence of the successive refactorings can affect the end result, which may improve the implemented tool.

Danksagung

Bedanken möchte ich mich bei Prof. Dr. rer. nat. habil. Gunter Saake, Dipl. Inform. Martin Kuhlemann und Dipl. Inform. Norbert Siegmund, die die Betreuung der Masterarbeit übernommen und mir durch ihre zahlreichen Hinweise während der Bearbeitung des Masterthemas geholfen haben. Mein weiterer Dank gilt meiner Familie, die mich während des gesamten Studiums in jeder Hinsicht unterstützt haben.

Inhaltsverzeichnis

Inhaltsverzeichnis **v**

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **xi**

Verzeichnis der Abkürzungen **xv**

1 Einleitung **1**

1.1 Problemstellung 2

1.2 Ziel und Inhalt 3

2 Grundlagen **5**

2.1 Objekt-orientierte Programmierung 5

2.2 Definition für Software-Qualität 6

2.2.1 Wartbarkeit 8

2.3 Software Produktlinien 9

2.4 Feature-orientierte Programmierung 10

2.5 Refactoring 11

2.5.1 Automatisiertes Refactoring 14

2.5.2 Refactorings und Software-Qualität 15

2.6 Refactoring Feature Module 16

3 Software-Metriken **19**

3.1 Messung der Software-Qualität mit Software-Metriken 19

3.2 Traditionelle Software-Metriken für Wartbarkeit 21

3.2.1 Lines Of Code 21

3.2.2 McCabes Komplexitätsmetrik 21
3.2.3 Halsteads Software Science-Metriken 23
3.3 Wartbarkeit-Metriken für OOP 25
3.3.1 Kohäsion 26
3.3.2 Kopplung 31
3.4 Metriken-Suite für die Wartbarkeit 32
3.4.1 Maintainability Index 33
3.4.2 Die Metriken-Suite von Li und Henry 35
3.4.3 Ein Metriken-Modell von Heitlager et al. 36
3.5 Zusammenfassung 37

4 Verwandte Arbeiten 39
4.1 Optimierung qualitativer Eigenschaften von SPLs 39
4.2 Identifizierung von Schwachstellen im Quellcode 40
4.3 Refactorings zur Unterstützung von NFPs in SPLs 42

5 Realisierung der Metriken und Refactorings 45
5.1 Beispiel-Anwendung: die Graph-Produktlinie 45
5.2 Realisierung der Software-Metriken 49
5.2.1 Implementierung der LOC-Metrik 49
5.2.2 Implementierung der CBO-Metrik 50
5.2.3 Implementierung der RFC-Metrik 51
5.2.4 Implementierung der CC-Metrik 52
5.2.5 Implementierung der DBC-Metrik 53
5.3 Umsetzung der unterstützenden Refactorings 54
5.3.1 Implementierung des EXTRACT METHOD-Refactorings 55
5.3.2 Implementierung des EXTRACT CLASS-Refactorings 56
5.3.3 Implementierung des DECOMPOSE CONDITIONAL-Refactorings 58
5.3.4 Implementierung des REPLACE METHOD WITH METHOD OBJECT-Refactoring 58
5.3.5 Implementierung des MOVE METHODS AND FIELDS-Refactoring 59
5.4 Anwendung der Refactorings auf Testbeispiel 60
5.4.1 Refactorings für die RFC-Metrik 60

5.4.2 Refactorings für die DBC-Metrik 63
5.4.3 Refactoring für die CBO-Metrik 65
5.4.4 Refactorings für die CC-Metrik 67
5.4.5 Refactorings für die LOC-Metrik 69
5.5 Zusammenfassung 70

6 Automatisierte Anwendung der Refactorings 71
6.1 Implementierungskonzept des Plug-ins 71
6.2 Automatisierte Identifizierung von Refactoring-Parametern 74
6.2.1 Identifizierung der Refactoring-Parameter für die RFC-Metrik . . 74
6.2.2 Identifizierung der Refactoring-Parameter für die DBC-Metrik . . 78
6.2.3 Identifizierung der Refactoring-Parameter für die CBO-Metrik . . 79
6.2.4 Identifizierung der Refactoring-Parameter für die CC- und LOC-Metrik 81
6.3 Evaluierung des Plug-ins 82
6.3.1 Evaluierung mit der TankWar-SPL 83
6.3.2 Evaluierung mit der VioletUMLEditor-SPL 90
6.4 Zusammenfassung 95

7 Zusammenfassung und Ausblick 97

Literaturverzeichnis 101

Abbildungsverzeichnis

2.1 Architektur des RFCComposer auf Basis von [KBA09] 17

3.1 Beispielmethode für zyklomatische Komplexität von 2. 22
3.2 Venn Diagramm-Darstellung der Klasse X 27
3.3 LCOM-Anomalie 27
3.4 LCOM Klasse Z 27

5.1 Feature-Modell der Graph-Produktlinie 48
5.2 Die Vererbungshierarchie zwischen den Klassen *WorkSpace, NumberWorkSpace, CycleWorkSpace, FinishTimeWorkSpace* und *WorkSpaceTranspose* 48

6.1 Das erweiterte Kontextmenü der PackageExplorer-View mit dem NFP Measure-Untermenü 72
6.2 Der Baum der CC-Metrik einer Instanz der Graph-Produktlinie 73
6.3 Grundelemente der Graph-Klasse 73
6.4 Die RFC-View 75
6.5 Vorschläge des REPLACE METHOD WITH METHOD OBJECT-Refactorings für die RFC-Metrik 75
6.6 Liste der potenziellen Zielklasse des MOVE METHODS AND FIELDS-Refactorings für die RFC-Metrik der Graph-Klasse 76
6.7 Liste der potenziellen Methoden des MOVE METHODS AND FIELDS-Refactorings für die RFC-Metrik der Graph-Klasse 76
6.8 Liste der Paare von Methoden, die gemeinsame externe Methoden aufrufen 77
6.9 Liste der potenziellen Methoden des MOVE METHODS AND FIELDS-Refactorings für die RFC-Metrik der Graph-Klasse 77

6.10 Potenzielle Zielklasse des MOVE METHODS AND FIELDS-Refactorings für die DBC-Metrik 78
6.11 Verwandte Methoden der Kandidat-Felder für das MOVE METHODS AND FIELDS-Refactoring der *Graph*-Klasse 79
6.12 Kandidat-Felder und -Methoden des EXTRACT CLASS-Refactorings für die DBC-Metrik der *Graph*-Klasse 80
6.13 Potenzielle Zielklassen des MOVE METHODS AND FIELDS-Refactorings für CBO-Metrik der *Graph*-Klasse 80
6.14 Verwandte Methoden des MOVE METHODS AND FIELDS-Refactorings für CBO-Metrik 81
6.15 Die CC Editor-View des EXTRACT METHOD-Refactorings für die CC-Metrik 82
6.16 gewählte Features der Variante der TankWar-SPL 84

Tabellenverzeichnis

3.1 Die Menge der betrachteten Eigenschaften einer Methode bzw. eines Attributes 30
3.2 Kohäsion zwischen Entitäten der Klasse X und Y 30

5.1 Relationen zwischen den Features der Graph-Produktlinie 47
5.2 Ergebnisse der RFC-Metrik vor und nach dem MOVE METHODS AND FIELDS-Refactoring 62
5.3 Ergebnisse der RFC-Metrik vor und nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring 63
5.4 Ergebnisse der RFC-Metrik vor und nach dem EXTRACT CLASS-Refactoring 64
5.5 Ergebnisse der DBC-Metrik vor und nach dem MOVE METHODS AND FIELDS-Refactoring 64
5.6 Ergebnisse der DBC-Metrik vor und nach dem EXTRACT CLASS-Refactoring 65
5.7 Ergebnisse der CBO-Metrik vor und nach dem MOVE METHODS AND FIELDS-Refactoring 66
5.8 Ergebnisse der CBO-Metrik vor und nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring 67
5.9 Ergebnisse der CBO-Metrik vor und nach dem EXTRACT CLASS-Refactoring 67
5.10 Ergebnisse der CC-Metrik vor und nach dem EXTRACT METHOD-Refactoring 68
5.11 Ergebnisse der LOC-Metrik vor und nach dem EXTRACT METHOD-Refactoring 69
5.12 Unterstützung der Refactoring für die Metriken 70

6.1 Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für RFC-Metrik 84
6.2 Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für RFC- und CBO-Metrik 85
6.3 Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für DBC-Metrik 86
6.4 Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für DBC- und RFC-Metrik 86
6.5 Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für CBO-Metrik 87
6.6 Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für CBO- und RFC-Metrik 88
6.7 Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für CC-Metrik 88
6.8 Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für CC-Metrik 89
6.9 Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für LOC-Metrik 89
6.10 Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für LOC- und CC-Metrik 90
6.11 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für RFC-Metrik 91
6.12 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach den sukzessiven Refactorings für RFC- und CBO-Metrik 92
6.13 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für DBC-Metrik 92
6.14 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach den sukzessiven Refactorings für DBC- und CBO-Metrik 93
6.15 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für CC-Metrik 94
6.16 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach den sukzessiven Refactorings für CC- und RFC-Metrik 94
6.17 Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für LOC-Metrik 95

6.18 Ergebnisse der Evaluierung des Refactoring Plug-ins 95

Verzeichnis der Abkürzungen

AST	Abstract Syntax Tree
CBO	Coupling Between Object Classes
CC	Cyclomatic Complexcity
CSP	Constraint Satisfaction Problem
DBC	Distance Based Cohesion
DIT	Depth of Inheritance Tree
FOP	Feature-orientierte Programmierung
IDE	Integrated Development Environment
LCOM	Lack of Cohesion in Methods
LMP	Logic Meta Programming
LOC	Lines Of Code
MI	Maintainability Index
NOC	Number of Children
OOP	Objekt-orientierten Programmierung
RFC	Response For a Class
RFM	Refactoring Feature Module
SPL	Software-Produktlinie
WMC	Weighted Methods Per Class

Hervorhebungen im Text werden *kursiv* dargestellt.

Kapitel 1

Einleitung

Heutzutage befindet sich Software in fast allen modernen hightech-Produkten. Sie spielt eine wichtige Rolle bei dem Erfolg eines Produkts [vdLSR07]. Deswegen ist in jeder Firma eine Frage von entscheidender Bedeutung: wie kann man Software schnell, effektiv mit geringem Kosten und Wartungsaufwand entwickeln? Die Antwort dieser Frage liegt in der Wiederverwendung von Software-Komponenten [Opd92]. Wiederverwendbare Software und Software-Komponenten bringen bemerkenswerte Produktivitätserhöhung ein, reduzieren die Produkteinführungszeit, die Entwicklungskosten und verbessern die Wartbarkeit und Qualität einer Software [vdLSR07],[Nor02].

Software-Produktlinie (SPL) ist ein Lösungskonzept um wiederverwendbare Software-Artefakte zu implementieren und in großem Maßstab zu benutzen. SPLs können Kosten sowie die Einführungszeit der Softwareentwicklung bis zu 90% einsparen [vdLSR07]. Infolgedessen wurde das Konzept der SPL als ein bedeutender Paradigmenwechsel im Softwareentwicklungsprozess betrachtet [vdLSR07] und bereits von vielen Unternehmen als Entwicklungsmodell übernommen.

Eine SPL ist eine Sammlung von Software-Produkten, die eine gemeinsame, verwaltete Menge von Merkmalen teilen, die die Anforderungen eines bestimmten Markt-Segments oder eines Anwendungsgebiets genügen [Nor02]. Das Anwendungsgebiet einer SPL wird als Domäne bezeichnet [vdLSR07]. Bausteine einer SPL sind Features, die sich auf die von der Domäne definierten Belange beziehen. Features erweitern das bestehende Programm um Funktionalitäten und Entwurfsentscheidungen und stellen neue potenzielle Konfigurationsmöglichkeiten zur Verfügung [ALMK08]. Aus einer Menge von Features können zahlreiche unterschiedliche Programmvarianten durch Feature-Auswahl generiert werden.

Feature-orientierte Programmierung (FOP) ist ein Programmierparadigma, welches auf dem Konzept von Features basiert. Das Ziel von FOP ist der Aufbau von großen, maßgeschneiderten Software-Systemen [AK09]. Features werden in Feature Modulen realisiert. Die Beziehungen und Abhängigkeiten zwischen Features werden durch ein Feature-Modell beschrieben [Bat04]. Das Generieren der unterschiedlichen Anwendungen wird durch Auswahl der entsprechenden Feature-Module erfolgt. Dieser Vorgang wird als Feature-Komposition bezeichnet [BSR04] und stellt als Endergebnis ein betriebsfähiges Programm zur Verfügung. Die Menge der ableitbaren Programme von der Menge der Features bilden eine SPL. Die Gültigkeit und Anwendbarkeit eines Programms im Einsatz hängt aber nicht nur von dessen Funktionalitäten sondern auch von dessen

nicht-funktionalen Eigenschaften ab (z.B. dessen unterstützte Betriebssysteme). Nicht-funktionale Eigenschaften diktieren nicht was eine Software machen kann, sondern wie und in welchen Umständen die Software ihre vorgesehene Funktionen ausführen kann [SSSP07]. Das Konzept von Features ist für die Beschreibung von sowohl funktionalen als auch von nicht-funktionalen Anforderungen geeignet [SKPA10]. Durch Bereitstellung von alternativen Implementierungen für ein Feature, die für die geforderten Eigenschaften optimiert sind, werden nicht-funktionale Anforderungen erfüllt und neue Variationspunkte eingeführt[SRK+08].

Software-Refactoring oder nur Refactoring bietet die Möglichkeit, Struktur bestehender Programme zu ändern und zugleich ihren externen Verhalten beizubehalten [Fow99]. Das Zusammenarbeit zwischen FOP und Refactoring, welches als Refactoring Feature Modulen (RFM) bezeichnet wird [KBA09], ist in aktueller Forschung einer der Ansätze um generierte Programme einer SPL an spezifische nicht-funktionale Anforderungen anzupassen.

1.1 Problemstellung

In vielen Fällen reicht die Entwicklung der SPL in Bezug auf funktionale Eigenschaften nicht aus, sondern es müssen zusätzlich nicht-funktionale Anforderungen berücksichtigt werden. Ein Datenbankmanagementsystem auf einem mobilen Geräte wie Smartphone oder PDA muss an die Einschränkungen der Systemressourcen angepasst werden. Software auf akku- oder batteriebetriebenen Systeme müssen energiesparsam sein. Software für Echtzeitsanwendungen sollten Performance-optimiert werden.

Quellcode-Qualität ist auch eine wichtige nicht-funktionalen Eigenschaften von Software. Insbesondere müssen sich Software-Hersteller immer intensiver um Softwarewartbarkeit kümmern. Bereits seit den Neunzigerjahren trägt der Wartungsaufwand von Software bis zu 60% des gesamten Produktionskosten bei[CALO94]. Die Optimierung von qualitativen Eigenschaften bezüglich der Wartbarkeit von Software ist deshalb in dieser Arbeit das Hauptthema. Einer der Lösungsansätze ist die Implementierung verschiedener Alternativen für ein Feature, dabei optimiert jede Alternative spezifische nicht-funktionale Eigenschaften [SKPA10]. Das System wächst aber demnach schnell und wird schwer zu warten und zu weiterentwickeln [SKPA10]. Ein anderer Lösungsansatz ist die Verwendung von Refactorings, welche Quellcode umwandeln können ohne dabei die semantische Integrität des Programms zu verletzen.

Im Allgemein wird ein Refactoring-Prozess in folgenden Schritten durchgeführt [TM03]:

1. Feststellen ob das Programm refaktorisiert werden sollte.

2. Identifizierung der Refactoring-Möglichkeiten, d.h. Ermittlung der Schwachstellen im Quellcode, die schlechtes Design oder schlechte Programmierung aufweisen und Auswahl der entsprechenden Refactorings.

3. (automatisierte) Anwendung der gewählten Refactorings: zuerst muss geprüft werden, ob das Refactoring die Semantik des Programms bewahren kann. Danach wird die tatsächliche Transformation durchgeführt.

Obwohl viele bekannte integrierte Entwicklungsumgebungen (engl. integrated development environment - IDE) bereits Refactorings in sich eingebettet haben, beschränkt sich die Unterstützung nur auf den dritten Schritt des oben geschilderten Prozesses. Dafür bieten die IDEs dem Anwender eine Liste von Refactorings an und basierend auf einer Auswahl führen sie das Refactoring automatisch durch. Man muss selbst die Entscheidung treffen, ob das Programm refaktorisiert werden soll und danach muss sowohl die Entdeckung von Refactoring-Kandidaten als auch die Auswahl der entsprechenden Refactorings manuell erfolgen. Diese Aufgaben sind nicht trivial und fehleranfällig, weil:

- gängige IDEs keine genügende Übersicht über verschiedene nicht-funktionale Aspekte eines Programms zur Verfügung stellen, mit der man die Aspekte messen, vergleichen und klassifizieren kann. Meistens werden die nicht-funktionalen Eigenschaften nur durch Erfahrungen der Entwickler nachvollzogen (z.B. lange Methoden sind oft kompliziert oder Klassen mit großer Vererbungshierarchie sind schwer zu ändern etc.)
- die Ermittlung von Refactoring-Möglichkeiten wenig durch die IDEs unterstützt wird. Man kann nur bestimmte einfache Anfragen an Quellcodes des Programms stellen z.B. welche Methoden benutzen das eine Feld der Klasse. Komplexere Anfragen wie z.B. Finde alle Methoden und Felder, auf die eine Methode zugreift, sind nur durch manuelle Untersuchung des Quellcodes zu beantworten.
- die Entscheidung hinsichtlich eines vernünftigen Refactorings schwer zu treffen ist. Selbst wenn man feststellen kann, dass eine Stelle im Quellcode refaktorisiert werden muss, muss man zwischen mehreren anwendbaren Refactorings wählen. Außerdem sind nicht alle nützlichen Refactorings schon in den IDEs vorhanden, daher kann man sie nur manuell durchführen.

Alle diese Probleme hindern den Einsatz von Refactorings zum Optimieren der nicht-funktionalen Eigenschaften einer SPL.

1.2 Ziel und Inhalt

Das Ziel der Arbeit besteht darin, Lösungen für die Probleme bei Anwendung von Refactorings zur Optimierung der nicht-funktionalen Eigenschaften zu finden, die die Wartbarkeit von Produkten einer SPL bestimmen. Es wird versucht, folgende Fragen zu beantworten:

- Wie kann man unterschiedliche Aspekte der Wartbarkeit einer Software quantifizieren?
- Wie kann man anhand von Metriken Stellen im Quellcode erkennen, an denen Refactorings eine Verbesserung der jeweiligen Metrik erzielen können?
- Wie werden Refactorings für bestimmte Eigenschaften ausgewählt?

Anhand der gewonnenen Erkenntnisse soll ein Prototyp zur Verbesserung von nicht-funktionalen Eigenschaften in Bezug auf die Wartbarkeit einer Software implementiert werden, welcher auf Refactorings in Quellcode-Ebene basiert.

Die Diskussion bezüglich der Eigenschaften der unterstützenden Refactorings beruht auf einem Beispiel einer Graph-Produktlinie. Dabei werden gewählte nicht-funktionale Eigenschaften bemessen und dann werden entsprechende Refactorings empfohlen und angewendet. Die Ergebnisse des Beispiels werden zusammengefasst und daraus Ableitungen für die automatisierte Umsetzung von Refactorings gezogen. Schließlich wird der Prototyp an Anwendungsbeispielen zweier SPLs evaluiert.

Die Arbeit ist folgendermaßen strukturiert. Kapitel 2 führt in die grundlegenden Begriffe und Konzepte ein. Daran schließt sich die Einführung von Software-Metriken im Kapitel 3 an, die der Beschreibung verschiedener Aspekte der Wartbarkeit im Quellcode dienen. Verwandte Arbeiten und Lösungskonzepte werden in Kapitel 4 dargestellt und mit der vorliegenden Arbeit verglichen. Kapitel 5 ist das Hauptkapitel der Arbeit. In diesem Kapitel wird vorab angezeigt, wie durch Quellcode-Metriken Probleme der Wartbarkeit erfasst werden. Anschließend werden Probleme bei Anwendung der optimierenden Refactorings in dem gewählten Anwendungsbeispiel erörtert und zusammengefasst. Schließlich werden aus der Zusammenfassung Schlussfolgerungen für die automatisierte Umsetzung von Refactorings gezogen. Kapitel 6 erklärt das Implementierungskonzept des Prototyps. Ergebnisse der Anwendung des Prototyps werden präsentiert und evaluiert. Zum Schluss fasst Kapitel 7 die Arbeit zusammen und stellt den Ausblick auf weitere Forschungsfragen vor.

Kapitel 2

Grundlagen

Dieses Kapitel führt in die theoretischen Grundlagen der Arbeit ein. Beginnend werden die Prinzipien der Objekt-orientierten Programmierung (OOP) beschrieben. Dabei wird der Aspekt der Wiederverwendung gezielt erklärt. Der darauf folgende Abschnitt erläutert die allgemeinen Begriffe von Software-Qualität und Software-Wartbarkeit. Daran schließen sich die Erläuterungen zu SPL und FOP an. Die Beschreibung des Begriffs Refactoring erfolgt im nächsten Abschnitt und wird in Zusammenhang mit Automatisierung und Software-Qualität gebracht. Im letzten Abschnitt werden die Grundlagen von RFM eingeführt.

2.1 Objekt-orientierte Programmierung

Der hohe Aufwand des Softwareentwicklungsprozesses motiviert die Wiederverwendung und Entwicklung von bestehenden Software-Komponenten [Opd92]. Wiederverwendung umfasst den Prozess, indem Kenntnisse über das eine Software-System für die Entwicklung und Wartung des anderen Systems benutzt werden [Opd92]. Die seit langem vorliegende Herausforderung und auch das Ziel von Softwareentwicklung ist es, Programme von wiederverwendbaren Standardbestandteilen entwerfen und entwickeln zu können [WOZ91].

Die Einführung des Objekt-orientierten Paradigmas ist ein signifikanter Fortschritt in dem Softwareentwicklungsgebiet. Wie dessen Vorgänger, das prozedurale Paradigma, nutzt Objekt-orientierte Paradigma auch funktionale Dekomposition aus, um Funktionen zu spezifizieren, die ein Problem lösen werden [KM90] . Darüber hinaus zielt es mehr auf die Beschreibung von Daten [KM90]. Die Konzepte von OOP wie Objekt, Klasse, Datenabstraktion, Vererbung und Polymorphismus unterstützen explizit die Wiederverwendung von Software-Komponenten [Opd92]. Diese Konzepte werden wie folgt beschrieben:

- Objekte: modellieren Entitäten in einer Anwendungsdomäne [KM90]. Ein Objekt besteht aus Operationen, die das Verhalten des Objekts charakterisieren, und einem gemeinsamen Zustand. Der Zustand eines Objekts lässt sich von seinen Variablen beschreiben, die als Instanzvariablen bezeichnet werden[Weg90].

- Klassen: dienen als Vorlagen für Objekte, woraus Objekte erstellt werden können

[Weg90]. Durch Klassendefinitionen erfolgt die Kapselung von Zustand und Verhalten. Dabei können Implementierungsdetails verborgen werden [Opd92]. Die Veröffentlichung des Zustands und der Operationen, die den Zustand modifizieren, stellt eine Schnittstelle zur Verfügung.

Die Wiederverwendung durch Klassen ergibt sich dadurch, dass Objekte einer Klasse sich die in der Klassendefinition deklarierten Instanzvariablen und Operationen teilen.

- Vererbung: ermöglicht die Wiederverwendung der Verhalten und unter Umständen der Instanzvariablen der einen Klasse in der Definition neuer Klassen [Weg90]. Erbende Klassen werden als Unterklassen oder Subklassen bezeichnet [Weg90]. Eine Klasse, die von anderen Klassen geerbt wird, ist Oberklasse von diesen Unterklassen. Dadurch entsteht die Klassenhierarchie des Programms.

  Vererbung ist ein vielversprechendes Konzept um das Ziel zu realisieren, Software-Systeme von wiederverwendbaren Bestandteilen konstruieren zu können [KM90]. Durch Vererbung können verschiedene Formen von Beziehungen zwischen Verhalten beschrieben werden: Klassifizierung, Spezialisierung und Generalisierung [Weg90]. Änderung eines Verhalten an der Oberklasse muss nicht in jeder Unterklasse realisiert werden, sondern wird automatisch in Unterklassen reflektiert. Erweiterungen in erbenden Klassen verursachen keine Änderungen in geerbter Klasse, dadurch bleibt die vorliegende Code-Basis unberührt.

- Polymorphismus: im Allgemein bedeutet Polymorphismus die Fähigkeit, mehrere Formen zu haben [KM90]. Monomorphe Programmiersprachen verstehen nur Variablen und Werte, die zu einem einzigen Typ gehören [CW85]. Im Gegensatz dazu dürfen Werte und Variablen in polymorphen Programmiersprachen mehr als einen Typ haben. Polymorphe Typen sind Typen, deren Operationen auf Werte mehrerer Typen anwendbar sind [CW85]. Die gültige, anwendbare Menge der Typen ist zur Laufzeit mittels der Klassen-Hierarchie zu ermitteln [KM90]. Demnach ist das Konzept der Vererbung und die Idee des Polymorphismus eng miteinander verbunden. Sollte Klasse Y Klasse X erben, ist Y eine X und deshalb ist Y für X vertretbar, wo X erwartet wird.

2.2 Definition für Software-Qualität

Einer der entscheidenden Faktoren, der ein erfolgreiches Softwareprojekt von anderen misslungenen Projekten unterscheidet, besteht darin, dass das gelungene Projekt ein gutes Software-Qualitätsmanagement hatte während die anderen nicht [Kan02]. Jedoch wird der Begriff Software-Qualität in Software-Qualitätsengineering und -management oft mehrdeutig benutzt [Kan02]. Dies führt darauf zurück, dass Qualität keine einzelne Idee sondern eine Kombination mehrerer Faktoren ist, ein multidimensionales Konzept ist [Kan02, Mey97]. Der zweite Grund liegt darin, dass es verschiedene Abstraktionsebenen für das Qualitätskonzept gibt und man auf unterschiedlicher Ebenen mit einander spricht [Kan02]. Die allgemeine Benutzung des Begriffs Qualität im Alltag kann auch zu unterschiedlichen Sichten von Benutzerseite und Entwicklerseite führen [Kan02]. Diese

zwei Sichten werden als externe Faktoren und interne Faktoren der Software-Qualität bezeichnet [Mey97].

Die externen Faktoren der Software-Qualität sind kundenspezifische Anforderungen wie z.B. Geschwindigkeit oder einfache Nutzung, die von Benutzer erkannt werden [Mey97]. Oft betrachten Benutzer Software-Qualität als eine unfassbare Charakteristik: sie kann besprochen, nachempfunden, bewertet aber nicht gewogen oder bemessen werden [Kan02]. Im Gegensatz dazu sind interne Faktoren wie z.B: Modularisierbarkeit, Wartbarkeit oder Wiederverwendbarkeit nur von Entwicklern wahrzunehmen da sie mit Software-Artefakten wie Architektur-Design, Modell-Design und Quellcode verbunden sind [Mey97]. Letztendlich sind nur die externen Qualitäten für die Benutzerzufriedenheit des Produkts verantwortlich aber der Schlüssel zum Erreichen dieser Qualitäten liegt in der Sicherung der internen Qualitäten: Entwickler müssen Techniken anwenden um diese internen Qualitäten zu gewährleisten [Mey97].

In 1991 wurde ein internationaler Konsensus über die Terminologie für Qualitätscharakteristik zur Evaluierung der Software-Produkte unter dem Namen: ISO/IEC IS 9126:1991 eingeführt [HKV07]. Anschließend wurde diese Standardisierung von 2001 bis 2004 erweitert und enthält bis jetzt eine internationale Standardisierung (engl. international standard) [ISO01] und drei technischen Berichten (engl. technical report)

- IS 9126-1: das Qualitätsmodel
- TR 9126-2: externe Metriken
- TR 9126-3: interne Metriken
- TR 9126-4: Laufzeitsqualität-Metriken

Das Qualitätsmodel von [ISO01] teilt das Konzept der internen und externen Faktoren der Software-Qualität in 6 Kategorien auf, die weiter in 27 Subkategorien aufgeteilt werden. Die 6 Qualitätscharakteristiken einer Software sind:

- Funktionalität: erfüllt die Software die erforderlichen Funktionen?
- Zuverlässigkeit: inwiefern ist die Software zuverlässig?
- Nutzbarkeit: ist die Software leicht zu bedienen?
- Effizienz : wie verbraucht die Software Ressourcen?
- Wartbarkeit: ist die Software leicht zu ändern?
- Portabilität: ist es leicht, die Software in eine andere Umgebung zu übernehmen?

Zur Erhöhung der allgemeinen Benutzerzufriedenheit der Software müssen diese Qualitätsattributen sowohl in der Plan- und Designphase als auch in der Programmierphase beachtet werden. Jedoch ist es auch zu bemerken, dass eine optimale Umsetzung aller diesen Attributen zur gleichen Zeit nicht möglich ist da sie zum Teil sich gegenseitig ausschließen[Kan02]. Beispielsweise je höher die Funktionalität einer Software ist, desto komplizierter ist die Software zu warten. Für unterschiedliche Benutzer und unterschiedliche Software-Typen sollen unterschiedliche Gewichtungsfaktoren für die Software-Qualitätsattribute definiert werden [Kan02]: z.B. komplexe Software soll sich mehr auf

Wartbarkeit konzentrieren während eine Software für Kunden mit einfachen Operationen sich der Nutzbarkeit widmen soll. Als Nächstes werden die für die Arbeit relevanten Qualitätsattribute genauer betrachtet.

2.2.1 Wartbarkeit

Um den Begriff Wartbarkeit zu verstehen muss zunächst der Begriff Software-Wartung erklärt werden. Software-Wartung wurde von IEEE in dem Artikel IEEE Standard for Software Maintenance [IEE19] wie folgt definiert:

> Software-Wartung ist die Modifikation eines Software-Produkts nach der Auslieferung um Fehler zu beheben, Performance oder andere Attribute zu verbessern oder um das Produkt an geänderte Umgebung anzupassen.

Im Prinzip kann jede Aktion, die das Software-Produkt nach der Auslieferung modifiziert, als Software-Wartung betrachtet werden [MVL03a]. Weiterhin wird Software-Wartung in [IEE19] in 3 Aktivitätstypen klassifiziert:

- Adaptive Wartung: ist die Modifikation eines Software-Produkts nach der Auslieferung um ein Programm in einer geänderten oder ändernden Umgebung betriebsfähig zu halten.
- Korrigierende Wartung: ist die Modifikation eins Software-Produkts nach der Auslieferung um gefundene Fehler zu beheben.
- Vervollkommnende Wartung: ist die Modifikation eines Software-Produkts nach der Auslieferung um Performance oder Wartbarkeit zu verbessern.

Software-Wartung ist einer der relevanten Interessen der Software-Industrie geworden da die Kosten für Wartung bis zu 60% der gesamten Kosten betragen können[CALO94]. Demzufolge ist Wartbarkeit eines der entscheidende Qualitätsattribute von Software. In [ISO01] wird die Wartbarkeit einer Software wie folgt definiert:

> Wartbarkeit ist die Leichtigkeit, wie ein Software-System oder eine Software-Komponente modifiziert werden kann, um Fehler zu korrigieren, Performance oder andere Attribute zu verbessern oder an eine geänderte Umgebung anzupassen.

Diese Definition behaltet die drei oben genannten Wartungsaktivitäten (adaptive, korrigierende und vervollkommnende Wartung). Die Charakteristik der Wartbarkeit lässt sich durch 4 Subkategorien beschreiben [ISO01]:

- Analysierbarkeit: benötigter Aufwand, um Mangel oder Ursachen von Fehlern zu diagnostizieren, oder Teile zu identifizieren, die zu ändern sind.
- Modifizierbarkeit: benötigter Aufwand um Programm zu verbessern, Fehler zu beheben oder um an Umgebungsänderungen anzupassen.

- Stabilität: Wahrscheinlichkeit, dass unerwartete Wirkungen von Änderungen auftreten.
- Testbarkeit: benötigter Aufwand zum Validieren der Software nach Änderungen.

Mit Hilfe dieser Standardisierung und Aufteilung der Charakteristiken lassen sich Qualitätsattribute der Systemebene auf Codeebene abbilden [HKV07]. Somit kann die Wartbarkeit einer Software quantifiziert und bemessen werden.

2.3 Software Produktlinien

Die Idee einer Produktlinie dient dem Ziel, maßgeschneiderte Produkten zu vernünftigen Preisen liefern zu können [PBvdL05]. Durch Software-Produktlinien kann das Ziel erreicht werden, indem Software-Hersteller die Gemeinsamkeiten ihrer Software-Produkte ausnutzen [Nor02]. Eine SPL ist eine Sammlung von Software-Produkten, die eine gemeinsame, verwaltete Menge von Merkmalen teilen, die die Anforderungen eines bestimmten Markt-Segments oder allgemein einer Anwendungsdomäne genügen [Nor02]. Die gemeinsamen Merkmalen werden durch wiederverwendbare Software-Artefakten entwickelt [Nor02]. Diese umfassen alle Typen der Softwareentwicklungsartefakte z.B.: Anforderungen, Architekturmodelle, Design-Modelle, Software-Komponenten, Testpläne und Test-Design [PBvdL05]. Software-Produkte werden dadurch erstellt, indem die entsprechenden Merkmale ausgewählt werden. Maßgeschneiderte Anforderungen können durch vorausgeplante Variationsmechanismen (z.B: durch Vererbung) geschafft werden[Nor02].

Die wirtschaftlichen Gründe sind nicht die einzige Motivation der Entwicklung von SPLs. Durch intensive Wiederverwendung der Software-Artefakte in vielen Produkten werden Fehler schneller entdeckt und behoben, dadurch erhöht sich die Qualität der Produkte [PBvdL05]. Die Produkteinführungszeit ist auch ein wichtiger Faktor und für eine SPL ist sie in der Tat anfangs höher, da gemeinsame Artefakten konstruiert werden müssen. Allerdings auf lange Sicht sinkt die Einführungszeit für neue Produkte erheblich wegen der massenweisen Wiederverwendung von Software-Komponenten [PBvdL05]. Im Bezug zu den Aspekten der Wartbarkeit und Erweiterbarkeit von Software ist SPL auch eine vernünftige Lösung. Der Wartungsaufwand wird reduziert, indem Änderungen an einem Artefakt in alle Produkte propagiert werden, die dieses Artefakt benutzen [PBvdL05]. Die Einführung neuer Artefakte in die Plattform stellt Möglichkeiten zur Verfügung, abgeleitete Produkte der SPL erweitern zu können.

Die Entwicklung einer SPL geschieht mit zwei Prozessen und wird als Software Produktlinien Engineering bezeichnet [PBvdL05]:

- Domain Engineering ist verantwortlich für die Definition der wiederverwendbaren Plattform und demzufolge für die Definition der Gemeinsamkeit und der Variabilität der SPL [PBvdL05].
- Application Engineering ist verantwortlich für die Generierung von Produkten basierend auf der Plattform, die durch Domain Engineering realisiert wurde. Dabei werden Variationspunkte benutzt und es wird sichergestellt, dass sich die Zusammenstellung an die Spezifikation der Applikation hält [PBvdL05].

2.4 Feature-orientierte Programmierung

Features (dt. Merkmale) repräsentieren die Charakteristika von Software-Anwendungen und dienen dazu, Programmvarianten einer Anwendungsdomäne zu spezifizieren und zu unterscheiden [Gri00]. In der Regel entspricht ein Feature einer bestimmten Anforderung der Domäne und kann deswegen als Kommunikationsmittel zwischen Kunde und Softwareentwickler dienen [Gri00]. Unter den Anforderungen zählen sowohl Funktionalitäten als auch Eigenschaften der Software (z.B.: Performance oder Quellcode-Qualität etc.). Entsprechend den Anforderungen erweitern Features das bestehende Programm um Funktionalitäten und Entwurfsentscheidungen und stellen dabei neue potenzielle Konfigurationsmöglichkeiten zur Verfügung [AK09]. Aus einer Menge von Features können zahlreiche unterschiedliche Programmvarianten durch Feature-Auswahl generiert werden.

FOP ist eine Erweiterung der OOP [Pre97], welche auf dem Konzept von Features basiert. Die grundlegende Idee von FOP ist die Zerlegung eines Software-Systems hinsichtlich der Features, die es unterstützt [AK09]. Features dienen der Strukturierung des Designs und des Codes und werden als wichtigste Wiederverwendungseinheiten benutzt [AK09]. Die aus den Features abgeleitete Menge von Software-Anwendungen bilden eine SPL.

Features werden oft durch mehrere Klassen implementiert und Klassen implementieren auch oft mehr als ein Feature. Das Verstreuen der Implementierung eines Features über mehrere Module (Klassen) wird als das Querschneidende Belange - Problem (engl. crosscutting concerns problem) bezeichnet [Lad03]. Dies verletzt das Prinzip der Modularisierung, welches fordert, dass semantisch zusammenhängende Einheiten in einem Modul implementiert werden sollten [Par72], und führt somit zur Einschränkung der Verständlichkeit und Wartbarkeit der Software.

Prehofer forderte in [Pre97] eine explizite Realisierung von Merkmalen auf Programmiersprachenebene. Der gemeinsame Basis-Code wird vom ergänzenden Code getrennt, welcher die unterschiedlichen Features realisiert. Der ergänzende Code wird als First-Class-Entität der Programmiersprache betrachtet und in einem Modul gekapselt, welches in FOP als Feature Module bezeichnet wird [BSR04]. Ein Feature Modul besteht aus mehreren Klassen und Refinements (dt. Verfeinerungen) [AK09]. Ein Refinement beschreibt Änderungen an Basis-Code, die der Implementierung eines Features dienen, ohne dabei den Basis-Code zu ändern [AK09]. Änderungen können die Modifikation vorhandener Methoden oder das Hinzufügen neuer Attribute oder Methoden sein [BSR04]. Dafür führen einige FOP-Spracherweiterungen wie FeatureC++ [ALRS05] und AHEAD Tool Suite [Bat04] neue Schlüsselwörter für Refinement ein z.B.: *refines* für Klassenverfeinerung und *Super* für Methodenverfeinerung. Der Ansatz von der AHEAD Tool Suite steht im Mittelpunkt der Betrachtung dieser Arbeit. Um unterschiedliche Features zu implementieren, können eine Basis-Klasse mehrmals verfeinert werden (in unterschiedlichen .jak Datein), diese Verfeinerungen bilden ein Verfeinerungskette (engl. refinement chain), wobei jedes Kettenglied als Vererbungsklasse seines Vorgängers betrachtet werden kann. Bei Ableitung einer Variante muss diese Verfeinerungskette in eine einzelne Datei zusammengepackt werden. Die AHEAD Tool Suite ermöglicht zwei Varianten der Komprimierung mit den Tools *jampack* und *mixin*. Während das *jampack*-Tool die Kette in eine einzelne Klasse oder ein einzelnes Interface unterdrückt, hält *mixin* die Vererbungshierarchie in der erzeugten Datei auf. Dabei wird nur die am niedrigsten liegende

Klasse als „public“ und konkret deklariert, die anderen Klassen werden als „abstract“ implementiert. Die von *mixin* generierten Version hat den Vorteil, dass sie leichter zu debuggen ist, weil die Änderungen da automatisch an die entsprechende Verfeinerung propagiert werden. Neben der Implementierung der Features in .jak Dateien wird auch ein Feature-Modell definiert, was die Features und die Beschränkungen für die Kombination der Features beschreibt. Durch eine gültige Kombination der Features wird eine Konfigurationsdatei erstellt, die die Reihenfolge der Verfeinerungen bestimmt. Schließlich kann man mittels des Tools *composer* die Varianten erzeugen.

Die Implementierung einer SPL mittels der Jak-Sprache kann durch das Beispiel einer Chat-SPL erläutert werden. Dieses Chat-Programm dient dem Austausch von internen Instanznachrichten für Firmen. Da gemeinsame bzw. individuelle Anforderungen von unterschiedlichen Firmen vorliegen, sollte eine SPL implementiert werden. Features der Chat-SPL werden durch den Domain Engineering-Prozess identifiziert: beispielsweise muss die SPL in der Lage sein, Programme für unterschiedliche Betriebssystem-Plattforme (Windows, Unix etc.) generieren zu können, da unterschiedliche Betriebssysteme in den Firmen vorhanden sind. Ein anderer Variationspunkt ist das Feature Log, welches die vertauschten Nachrichten in Dateien protokolliert. Dafür muss eine neue Klasse *Log* wie im Listing 2.1 definiert werden, die das Schreiben von Nachrichten in eine Datei realisiert.

Listing 2.1: Log-Klasse

```
1  public class Log {
2      public static final String defaultLogFile = "chat_log.txt";
3
4      public static void write(String s) throws IOException {
5        write(defaultLogFile, s);
6      }
7
8      public static void write(String f, String s) throws IOException {
9          TimeZone tz = TimeZone.getTimeZone("EST");
10         Date now = new Date();
11         DateFormat df = new SimpleDateFormat ("yyyy.mm.dd␣hh:mm:ss␣");
12         df.setTimeZone(tz);
13         String currentTime = df.format(now);
14
15         FileWriter aWriter = new FileWriter(f, true);
16         aWriter.write(currentTime + "␣" + s + "\n");
17         aWriter.flush();
18         aWriter.close();
19     }
20 }
```

Beteiligt sind auch die Klassen *Client* und *Connection*, die das Loggen sowohl auf Benutzerseite und als auch auf Serverseite erledigen. Demnach müssen die beiden Klassen verfeinert werden (vgl. Listing 2.2 und Listing 2.3).

2.5 Refactoring

Softwareentwicklung ist ein iterativer Prozess, da neue oder veränderte Belange von Kunden während oder nach Implementierung der Software immer noch auftreten können. Änderungen, die kurzfristige Ziele realisieren oder ohne vollständiges Verständnis für das Design der Software durchgeführt wurden, können dazu führen, dass die Integrität des Systems allmählich nachlässt und die Programmstruktur nicht mehr dem Original-

design entspricht [Fow99]. Dies führt weiterhin dazu, dass neue kommende Änderungen nicht problemlos realisiert werden können oder Software-Komponenten nicht leicht wiederverwendbar sind [Opd92].

Listing 2.2: verfeinerte Client-Klasse

```
1  layer Log;
2
3  public refines class Client{
4
5      protected void handleIncomingMessage(Object msg) {
6    Super(Object).handleIncomingMessage(msg);
7    try{
8        Log.write(((TextMessage) msg).getContent());
9    }
10   catch (IOException e){}
11     }
12 }
```

Listing 2.3: verfeinerte Connection-Klasse

```
1  layer Log;
2  import java.io.IOException;
3  public refines class Connection {
4
5    protected void handleIncomingMessage(String name, Object msg) {
6      Super(String,Object).handleIncomingMessage(name,msg);
7      try{
8          Log.write(((TextMessage) msg).getContent());
9      }
10     catch (IOException e){}
11   }
12 }
```

Man muss an dieser Stelle das Software-System zuerst neu strukturieren (engl. Restructuring) [Opd92]. Restructuring ist laut Definition von [CCI90] die Umwandlung von einer Darstellungsform in eine andere Darstellungsform der gleichen Abstraktionsebene. Dabei muss das externe Verhalten des Systems beibehalten werden. Dieses von außen erkennbare Verhalten werden auch als Semantik des Programms bezeichnet [CCI90].

Refactoring ist ein Begriff, der ursprünglich von William Opdyke in seiner Dissertation eingeführt wurde [Opd92]. Im Prinzip ist Refactoring die objekt-orientierte Variante der Restructuring [MT04]. Refactoring wird in [Fow99] als ein Prozess definiert, der ein (objekt-orientiertes) Software-System auf solche Weise ändert, dass das beobachtbare Verhalten erhalten bleibt und dennoch die interne Struktur verbessert wird. Auf den ersten Blick lässt sich Refactoring als eine Art von Quellcode-Bereinigung begreifen aber das Ziel des Refactorings geht weiter. Refactorings sollen den Quellcode verständlicher und veränderungsbereit machen [Fow99].

Die Anwendung von Refactorings wird durch das folgende Beispiel demonstriert werden: in der Chat-Anwendung werden die Nachrichten wie im Listing 2.4 modelliert. Die *TextMessageEncoded* und *TextMessageColored* abstrahieren verschlüsselte Nachrichten bzw. Nachrichten mit Farben. Die Konstruktoren von diesen Unterklassen haben gemeinsame Felder, die im Konstruktor von *TextMessage* gesetzt werden sollen.

Listing 2.4: TextMessage-Klasse und ihre Unterklassen

```
1  public class TextMessage implements Serializable {
2    protected int sender;
3    protected int empfaenger;
4    protected String content;
5    public TextMessage(){}
6  }
7  public class TextMessageEncoded extends TextMessage{
8    private int encodingMethod;
9    TextMessageEncoded(int sender, int empfaenger, String content, int encodingMethod){
10     this.sender = sender;
11     this.empfaenger = empfaenger;
12     this.content = content;
13     this.encodingMethod = encodingMethod;
14   }
15 }
16 public class TextMessageColored extends TextMessage{
17   private Color color;
18   TextMessageColored(int sender, int empfaenger, String content, Color color){
19     this.sender = sender;
20     this.empfaenger = empfaenger;
21     this.content = content;
22     this.color = color;
23   }
24 }
```

Das PULLUP CONSTRUCTOR BODY-Refactoring von der Refactorings-Liste von Fowler [Fow99] ist demnach an dieser Stelle anwendbar. Die Motivation dieses Refactoring besagt, dass gemeinsame Verhalten in Konstruktoren der Unterklasse im Konstruktor der Oberklasse erfolgen sollen und er von den in Unterklassen aufgerufen werden soll. Nach der Ausführung von dem PULLUP CONSTRUCTOR BODY-Refactoring wird ein neuer Konstruktor in die TextMessage-Klasse eingeführt und der gemeinsame Code dahin verschoben. Ein Aufruf auf diesen Konstruktor ersetzt den gemeinsamen Code in Konstruktoren der Unterklassen. Das Ergebnis des Refactorings soll das Listing 2.5 veranschaulichen.

Listing 2.5: TextMessage-Klasse und ihre Unterklassen nach dem PULLUP CONSTRUCTOR BODY-Refactoring

```
1  public class TextMessage implements Serializable {
2    protected int sender;
3    protected int empfaenger;
4    protected String content;
5    public TextMessage(){
6    }
7    public TextMessage(int sender, int empfaenger, String content){
8      this.sender = sender;
9      this.empfaenger = empfaenger;
10     this.content = content;
11   }
12 }
13 public class TextMessageEncoded extends TextMessage{
14   private int encodingMethod;
15   TextMessageEncoded(int sender, int empfaenger, String content, int encodingMethod){
16     super(sender,empfaenger,content);
17     this.encodingMethod = encodingMethod;
18   }
19
20 }
21 public class TextMessageColored extends TextMessage{
22   private Color color;
23   TextMessageColored(int sender, int empfaenger, String content, Color color){
24     super(sender,empfaenger,content);
25     this.color = color;
26   }
27 }
```

2.5.1 Automatisiertes Refactoring

Refactorings sind hilfreiche Mittel zur Verbesserung der Verständlichkeit und Wartbarkeit von Software-Anwendungen. Jedoch sind Refactorings auch gefährlich in dem Sinne, dass sie bereits korrekten Code ändern und möglicherweise dabei Fehler einführen können[Fow99]. Darüber hinaus können Refactorings sehr zeitaufwändig und erschöpfend sein, besonders bei Ad-hoc-Anwendung der Refactorings kann es zu weiter tieferen Refactorings führen und man kann dabei schnell den Überblick verlieren. Die Langwierigkeit und Fehleranfälligkeit des manuellen Refactoring-Prozesses fordern die Entwicklung von Tools, die den Refactoring-Prozess automatisieren [Rob99].

Diese Tools müssen das beobachtbare Verhalten der Software bewahren. Das Problem liegt aber darin, dass es keine deutliche Definition für Verhalten gibt bzw. die Definition nicht ausreichend ist, um diese in der Praxis prüfen zu können [MT04]. Die originelle Definition von Opdyke besagt,dass ein Refactoring Verhalten-Beibehalten heißt, wenn das Programm vor und nach diesem Refactoring die gleichen Ausgaben für die gleiche Eingabe produziert [Opd92]. Im Zusammenhang mit der Definition hat Opdyke auch die Benutzung von Refactoring-Vorbedingungen vorgeschlagen, um die Verhalten-Beibehalten-Eigenschaft eines Refactorings zu gewährleisten. Vorbedingungen (engl. preconditions) prüfen, ob der zu modifizierende Code gültig für die erfolgreiche Anwendung des Refactorings ist z.B.: für das PULLUP CONSTRUCTOR BODY-Refactoring muss es geprüft werden, ob die von Konstruktoren der Unterklassen nach Oberklasse verschobenen Felder auch tatsächlich in Oberklasse vorhanden sind. In vielen Anwendungsdomänen reicht die Erhaltung von Eingabe-Ausgabe-Verhalten jedoch nicht aus[MT04]. Beispielsweise muss eine Anwendung für eingebettete Systeme auf Speicherverbrauch achten oder eine Anwendung für Echtzeitsysteme die Ausführungszeit optimieren. Deswegen müssen auch andere Aspekte des Verhaltens eines Programm berücksichtigt werden, die Roberts in seiner Dissertation als Refactoring-Nachbedingungen (engl. postconditions) kennzeichnete [Rob99]. Die Nachbedingungen beschreiben wie der Code nach dem Refactoring sich ändert und funktioniert [Rob99]. Beispielsweise für das PULLUP CONSTRUCTOR BODY-Refactoring muss der gemeinsame Code in Konstruktoren der Unterklasse durch den Aufruf des Konstruktors der Oberklasse ersetzt werden. Nachbedingungen können zu einigen Zwecken benutzt werden: um den Aufwand der Analyse von kommenden Refactorings zu reduzieren, um Vorbedingungen kombinierter Refactorings abzuleiten und um Abhängigkeiten zwischen Refactorings zu berechnen [Rob99]. Die Techniken von Vor- und Nachbedingungen können Rücksetzen, Benutzer-definierte Kombination von Refactorings und Multi-Benutzer Refactoring bei Refactoring Tools unterstützen [Rob99].

Weiterhin schlagt Roberts in [Fow99] einige Kriterien vor, die ein Refactoring Tool erfüllen muss, um in der Praxis einsetzbar zu sein. Zum einen gibt es technische Kriterien, die im Folgenden beschrieben werden.

- Programm-Datenbank: Die erste Aufgabe eines Refactoring Tools ist die Bereitstellung der Möglichkeit, unterschiedliche Programmentitäten wie Variablen oder Methoden über das ganze Programm suchen zu können. Man kann somit beispielsweise alle Methoden suchen, die auf eine bestimmte Variable zugreifen. Um diese durchsuchbare Datenbank zu erstellen, muss das Programm semantisch analysiert werden und jedes Codezeichen der entsprechenden Wortklasse zugeordnet werden.

- Parsebaum: Ein Parsebaum ist eine Datenstruktur, die die interne Struktur ei-

ner Klasse darstellt. Die meisten Refactorings manipulieren den Code unter der Methodenebene und benutzen dabei den Parsebaum um diese Codefragmente zu identifizieren und modifizieren.

- Korrektheit: Eine absolute Bewahrung des Verhaltens des Programms von Refactoring ist nicht in allen Fällen möglich. Ein Beispiel dafür ist, wenn ein Programm eine String-Variable definiert und das Java Reflection API benutzt um die Methode auszuführen, deren Namen die String-Variable enthält und man ändert den Name der Methode durch ein RENAME METHODE Refactoring (dt. Methode umbenennen). Diese Änderung wird nicht in dem Inhalt der String-Variable reflektiert und es führt demnach dazu, dass zur Laufzeit eine NoSuchMethodException (dt. keine solche Methode-Ausnahme) entstehen wird. Der Grund dafür ist, dass der Parsebaum nicht solche Semantiken erfassen kann und demzufolge muss der Entwickler selbst diese Fälle erkennen und beim Refactoring rücksichtsvoll übernehmen.

Zum anderen werden folgende praktische Kriterien genannt, die für die Benutzung der Refactorings von Anwendern relevant sind. Diese Kriterien bilden auch den Leitfaden für die Entwicklung des Refactoring-Tools dieser Arbeit.

- Geschwindigkeit: Das Verhältnis zwischen den Kosten (Ausführungszeit) und der Korrektheit muss immer überlegt werden da die benötigte Analyse und Transformation zur Durchführung eines Refactorings sehr zeitaufwändig sein kann, wenn das Refactoring kompliziert ist. In diesen Fällen soll man die Verantwortung für die Eingabe der Informationen und die Prüfung der Korrektheit dieser Informationen in die Hand des Entwicklers geben. Es ermöglicht somit die Analyse schneller durchzuführen. Obwohl dieser Ansatz unzuverlässig ist, weil der Entwickler Fehler machen kann, unterstützt er den Entwickler besser beim Anwenden des Refactoring Tools da in meisten Fällen der Entwickler die Semantik des Programms schon kennt.

- Rücksetzen: Mit automatisierten Refactorings besteht die Möglichkeit, Software nach dem „Try and Error“-Prinzip zu entwickeln. Man kann Änderungen an einem bestehenden Design ausprobieren und wenn das Ergebnis eines Refactoring ungenügend ist, kann man durch ein inverses Refactoring das ursprüngliche Design wiederbekommen. Dieses inverse Refactoring, was das originale Refactoring zurücksetzt, ist auch ein Refactoring und bewahrt demnach das Verhalten des Programms.

- Integration in IDE: Eine IDE spielt eine wichtige Rolle in den meisten Softwareentwicklungsprojekten. Eine IDE stellt zur Verfügung Editor, Compiler, Linker, Debugger und alle benötigten Werkzeuge, die der Entwicklung der Software dienen. Demnach sollten Refactorings in die IDE integriert werden und sind somit dem Entwickler zu jeder Zeit verfügbar.

2.5.2 Refactorings und Software-Qualität

Refactorings haben Einflüsse auf Quellcode-Qualität und nicht-funktionale Eigenschaften einer Software [MT04, SKPA10]. Diese Einflüsse können positiv für eine Eigenschaft,

aber zugleich negativ für andere Eigenschaften sein. Beispielsweise kann ein EXTRACT METHOD-Refactoring die Verständlichkeit des Quellcodes verbessern und damit die Wartbarkeit erhöhen indem es ein komplexes Codefragment einer Methode in eine neue Methode extrahiert, deren Name den Zweck des Codefragments erläutert. Die Performance bekommt aber dabei einen negativen Einfluss da die Originalmethode die neue Methode aufrufen muss. Refactorings können nach ihren Einflüssen auf Software-Qualität klassifiziert werden. So besteht die Möglichkeit, die Software-Qualität durch Anwendung passender Refactorings an richtigen Stellen im Quellcode zu verbessern [MT04]. In [SKPA10] werden die Refactorings von Fowlers Refactoring-Liste [Fow99] analysiert und nach ihren positiven bzw. negativen Einflüssen auf nicht-funktionale Eigenschaften aufgeteilt. Dennoch ist die Aufteilung nicht formell bewiesen und auch nicht alle Refactorings wurden evaluiert.

Um die Einwirkung eines Refactorings auf nicht-funktionale Eigenschaften bemessen oder bewerten zu können werden zumeist Software-Metriken eingesetzt [MT04, SKPA10]. Das Problem liegt darin, dass sich nicht alle Software-Eigenschaften leicht bemessen lassen [SKPA10]. Für andere Eigenschaften wie Wartbarkeit gibt es im Gegensatz dazu keine Einigung unter den Metriken. Siegmund et al. klassifizierten nicht-funktionale Eigenschaften in drei Kategorien, die dabei helfen, geeignete Messungstechniken für eine Eigenschaft zu selektieren [SRK+08].

2.6 Refactoring Feature Module

Feature Module können neue Klassen im bestehenden Programm hinzufügen, vorhandene Methoden verfeinern und neue Attribute und neue Methoden in die Klassen einführen. Dabei können neue Programme aus den Features generiert werden und es ist möglich, dass diese Programme als Bibliotheken von anderen Programmen benutzt werden [KBA09]. Es kann aber vorkommen, dass die externen Programme eine bestimmte Schnittstelle z.B.: Klassen- oder Methodenname erwarten, die jedoch nicht durch Komposition der Feature Module generiert wurden. Dieser Versatz wird als Inkompatibilität bezeichnet und hindert die Wiederverwendung von Features [KBA09]. Bei der Entwicklung einer SPL spielen nicht nur funktionale Belange, sondern auch die nicht-funktionalen Belange eine wichtige Rolle. Anforderungen bezüglich z.B.: der Performance oder der Wartbarkeit eines Programms kommen in der Praxis oft vor. Einer der Lösungsansätze ist die Implementierung verschiedener Alternativen für ein Feature, dabei optimiert jede Alternative spezifische nicht-funktionale Eigenschaften [SKPA10]. Das System wächst aber demnach schnell und es wird schwer zu warten bzw. weiterzuentwickeln [SKPA10]. Es ist deshalb in beiden Fällen notwendig, die Struktur der entsprechenden Software-Komponenten modifizieren zu können ohne dabei die Semantik des Programms zu verlieren.

Refactorings bieten diese Möglichkeit. In [KBA09] schlagen Kuhlemann et al. vor, Refactorings in Feature Module einzubinden. Dies wird als RFM bezeichnet [KBA09]. Zuerst wird die Software-Anwendung nach funktionalen Anforderungen durch Auswahl der entsprechenden Features erstellt. Dann werden Refactorings in RFM angewendet um die Inkompatibilität zu überwinden oder die nicht-funktionalen Anforderungen zu unterstützen. Dabei wird jedes Refactoring in eine Refactoring-Einheit gepackt, die als ein Element der Feature Modules angesehen werden können. Ähnlich wie bei Komposition

einer Feature-Sequenz (vgl. Abschnitt 2.4) kann man auch ein RFM in einer Sequenz definieren, die den Quellcode in dieser Reihenfolge generiert bzw. transformiert. Zur Realisierung des Konzepts haben die Autoren ein Framework implementiert, welches die Entwicklung der Refactorings und die Komposition der RFMs ermöglicht. Das Framework wird als *RFMComposer* bezeichnet [KBA09].

Die Idee des Frameworks ist, dass jedes Refactoring ein Interface hat, welches für jeden von dem Refactoring benötigten Parameter eine Getter-Methode zur Verfügung stellt. Eine Refactoring-Einheit wird in Form einer Klasse gekapselt, die das entsprechende Interface implementiert. Dafür muss sie alle Getter-Methoden des Interfaces mit den Werten der entsprechenden Parameter überschreiben. Ein Refactoring wird angewendet wenn das entsprechende RFM komponiert wird. D.h die Transformation wird auf den Code durchgeführt, der durch die Komposition der vor diesem RFC in der Sequenz stehenden Module erstellt wird. In dem Framework wird diese Aufgabe durch eine Composer-Komponente behandelt. Wenn der Composer auf ein RFC stoßt wird die Refactoring-Einheit analysiert und das entsprechende Refactoring über ein Plug-In-System geladen. Das *RFMComposer*-Framework wurde als eine Erweiterung für die *Jak*-Sprache entwickelt und benutzt das *jampack*-Tool aus der AHEAD Tool Suite für die Komposition der Feature Modules. Einen Überblick über die Architektur des *RFMComposer* liefert die Abbildung 2.1.

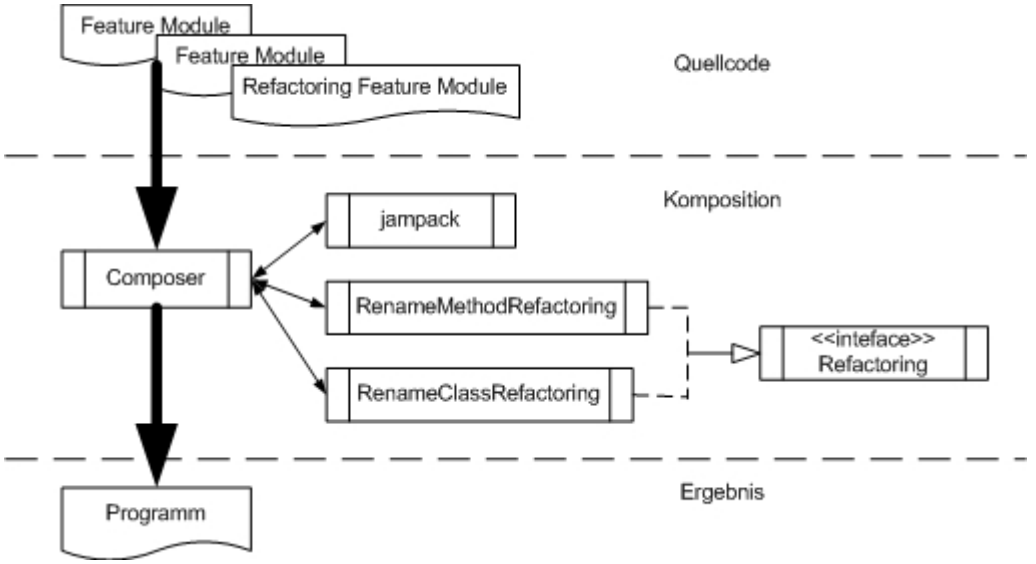

Abbildung 2.1: Architektur des RFCComposer auf Basis von [KBA09]

Zur Erläuterung der Vorgehensweise von RFMs soll ein Beispiel von der Chat-SPL dienen. Die generierte Chat-Anwendung aus der SPL wird in ein größeres Kommunikationssystem einer Firma integrieren. Dieses System wurde davor entwickelt und stellt einen Erweiterungspunkt für eine Chat-Anwendung zur Verfügung, jedoch erwartet das System unterschiedliche Schnittstellen als die von der SPL bereitgestellt werden z.B. Bezeichnungen für Klassen und Methoden. An dieser Stelle sollen RFMs definiert werden, welche Anpassungen an Quellcode der generierten Anwendung durchführen und demnach diese Anwendung integrierbar machen. Beispielsweise kan man mit einem RENAME CLASS-Refactoring (dt. Klasse umbenennen) den Namen einer Klasse umbenennen. Dafür müssen als Parameter die alte Klasse (inklusive das enthaltende Paket) und den

neuen Namen eingegeben werden. Die Listings 2.6 und 2.7 veranschaulichen die Vorgehensweise bei Umbenennung der *Server*-Klasse in eine *CommSystemChatServer*-Klasse.

Listing 2.6: bereitgestelltes Interface von dem Refactoring

```
1 public String getInterfaceOfRefactoringDefs() {
2     return "public␣interface␣RenameClassRefactoring␣{\n" +
3          "␣␣String␣getOldClass();\n" +
4          "␣␣String␣getNewClassName();\n" +
5          "}";
6 }
```

Listing 2.7: Implementierung des Interfaces in der Refactoring-Einheit

```
1 public class RenameClassRefactoring1 implements RenameClassRefactoring {
2   public String getOldClass(){
3     return "server.Server";
4   }
5   public String getNewClassName(){
6     return "CommSystemChatServer";
7   }
8 }
```

Kapitel 3

Software-Metriken

Das letzte Kapitel hat den Begriff der Software-Qualität im Zusammenhang mit den nicht-funktionalen Eigenschaften von Software insbesondere der Wartbarkeit erklärt. Dieses Kapitel stellt die wichtigen Arbeiten vor, die Software-Qualität mittels Metriken erfassen und kontrollierbar machen. Dabei werden die Metriken für die Wartbarkeit einer Software genauer betrachtet. Als Erstes werden die Eigenschaften von allgemeinen Software-Metriken veranschaulicht. Der folgende Abschnitt geht auf traditionelle Software-Metriken ein, die zur Messung der Wartbarkeit von Software im prozeduralen Programmierparadigma benutzt wurden. Im nächsten Abschnitt werden die für OOP relevanten Metriken eingeführt. Anschließend folgt die Erläuterung einiger wichtigen Metriken-Suiten für Software-Wartbarkeit. Dabei werden die Beschränkungen jeder Metriken-Suite diskutiert und die Schlussfolgerungen für die Anwendbarkeit der Metriken in dieser Arbeit gezogen. Eine Zusammenfassung bildet den Abschluss dieses Kapitels.

3.1 Messung der Software-Qualität mit Software-Metriken

Software-Qualität verstehen und kontrollieren zu können hilft bei Verbesserung des Softwareentwicklungsprozesses. Dafür muss man in der Lage sein, unterschiedliche Aspekte der Software-Qualität quantifizieren und messen zu können [DeM89]. Software-Metriken dienen diesem Zweck. Im Allgemein wird der Begriff *Messen* wie folgt definiert [FP97]:

> Messen beschreibt den Prozess, wobei Zahlen oder Symbole den Attributen von Entitäten der realen Welt zugeordnet werden, sodass diese Attribute von klar definierten Regeln beschrieben werden.

Eine Entität der Software-Messung kann sowohl ein Objekt wie eine Software-Spezifikation als auch ein Ereignis wie die Testphase eines Softwareprojekts sein [Fen94]. Ein Attribut bezeichnet ein Merkmal oder eine Eigenschaft einer Entität, dem oben Beispiel entsprechend kann es die Funktionalität (einer Spezifikation) oder die Dauer (der Testphase) sein [Fen94]. Bei Zuordnung der Zahlen oder Symbole müssen alle empirischen Beobachtungen über die Attribute und Entitäten aufbewahrt werden [Fen94]. Dafür wird in der Regel ein Modell für die Entitäten definiert, welches einen bestimmen

Gesichtspunkt spezifiziert [Fen94] (z.B. die Komplexität oder die Größe eines Moduls). Ein gutes Modell ist für Software-Messen besonders wichtig um Ambiguität zu vermeiden [Fen94], z.B. um Länge eines Programms anhand Anzahl der Codezeilen (engl. lines of code (LOC)) zu bewerten muss ein Programm-Modell definiert werden, um einzelne Codezeile identifizieren zu können.

Im Prinzip gibt es zwei Typen der Messung: direkte und indirekte [FP97]. Die Messung eines Attributs ist direkt wenn sie nicht von der Messung anderer Attributen abhängig ist während eine indirekte Messung auf die Ergebnisse anderer Messungen basiert [FP97]. In vielen Fällen ist eine Sammlung mehrerer Messungen (auch als Metriken-Suite genannt wird) wünschenswert, um den Überblick über den gesamten Entwicklungs- oder Wartungsprozess zu haben. Eine Metriken-Suite sollte sagen können, ob ein Software-Produkt gut oder schlecht ist, wobei jede Metrik einen Aspekt der Qualität erfasst [FP97]. Allerdings ist es oft umstritten, welche Metriken welchen Aspekten entsprechen und wie diese Metriken kombiniert werden können. Dies führt darauf zurück, dass viele Konzepte der Software-Qualität wie z.B. Wartbarkeit, Komplexität, Kohäsion, Kopplung etc. nicht klar definiert werden können und es vielen Metriken an theoretischen Grundlagen fehlt [BMB96, CK94]. Einige Arbeit von [Wey88], [BMB96] oder [CK94] haben versucht grundlegende Eigenschaften für Software-Metriken festzustellen und somit kann die Gültigkeit neuer Software-Metriken geprüft werden. Dennoch sind die vorgeschlagenen Eigenschaften selbst wegen ihrer Inkonsistenz und Unvollständigkeit noch zu diskutieren [BMB96] und daher ist noch keine Standardisierung für Software-Metriken vorhanden.

Es sei darauf hingewiesen, dass der Begriff Software-Metrik sich nicht nur auf Software-Qualität beschränken muss. Im Allgemein lassen sich Software-Metriken in drei folgenden Kategorien klassifizieren [Kan02]:

- Produkt-Metriken: beschreiben verschiedene Eigenschaften eines Software-Produktes wie Performance, Komplexität, Funktionalitäten oder Software-Qualitäten etc.

- Prozess-Metriken: beziehen sich auf Prozesse der Entwicklung einer Software wie Testen, Debuggen oder Fehlerbehebung etc.

- Projekt-Metriken: charakterisieren unterschiedliche Aspekte eines Software-Projektes wie Anzahl der beteiligten Entwickler, Produktivität oder Kosten etc.

Da die Optimierung von Qualitäten eines Software-Produktes das Ziel dieser Arbeit ist, stehen Software-Qualität-Metriken im Mittelpunkt der Betrachtung der Arbeit. Demnach werden im letzten Teil der Arbeit die Begriffe Software-Metriken und Software-Qualität-Metriken als Synonyme benutzt. Es sei darauf hingewiesen, dass entsprechend unterschiedlichen Abstraktionsebenen unterschiedliche Software-Metriken definiert werden. Diese Arbeit konzentriert sich auf Software-Metriken auf Quellcode-Ebene, die Eigenschaften der Wartbarkeit einer Software beschreiben. Im Folgenden werden diese Metriken genauer eingegangen.

3.2 Traditionelle Software-Metriken für Wartbarkeit

Seit den siebziger Jahren wurden Software-Metriken für das prozedurale Paradigma entwickelt [McC76, Hal77]. Jedoch beschrieben diese Metriken nicht direkt die Wartbarkeit sondern spezifische Aspekte wie die Komplexität, die Verständlichkeit oder den Datenfluss eines Programms etc. Unter den Metriken zählen beispielsweise die zyklomatische Komplexität von McCabe (1976) oder die Software Science-Metriken von Halstead (1977). Basierend darauf haben einige Arbeiten versucht, die Wartbarkeit eines (prozeduralen) Programms durch Software-Metriken vorherzusagen [WH88, Rom87].

3.2.1 Lines Of Code

Lines of Code (dt. Anzahl der Codezeilen) ist eine der gebräuchlichsten Metriken [FP97]. Es liegt daran, dass LOC in vielen Editoren verfügbar ist [MVL03b] und LOC dem Entwickler eine Intuition über die Größe der Software gibt. Im Prinzip verlangt ein großes System mehr Wartungsaufwand: das System wird schwerer zu analysieren und zu verstehen [Fow99, HKV07]. Jedoch sind einige Codezeilen von anderen Codezeilen unterschiedlich. Beispielsweise werden Leerzeilen oft benutzt um das Layout des Codes klarer und den Code lesbarer zu machen, die sollten aber nicht zu der Komplexität des Codes beitragen. Gleichermaßen könnte eine Methode mit Kommentaren, die die Funktion der Methode erklären, leichter zu verstehen als die ohne Kommentare, obwohl der LOC-Wert der Version mit Kommentaren offenbar größer als die ohne Kommentare. Es gibt daher verschiedene Strategien zum Zählen der Codezeilen, jede mit eigenem Zweck. An dieser Stelle muss man ein klares Modell definieren, welches die zu zählenden Codezeilen eindeutig identifizieren [FP97]. Laut Fenton und Pfleeger wird die Definition der Codezeilen von Hewlett-Packard am breitesten akzeptiert: eine Codezeile ist irgendwelche Anweisung im Programm, die keine Leerzeile und kein Kommentar ist [FP97]. Dies wird als NCLOC (engl. Non-Commented Lines Of Code) bezeichnet.

3.2.2 McCabes Komplexitätsmetrik

Die von McCabe im 1976 eingeführte zyklomatische Komplexität (eng. cyclomatic complexcity - CC) ist eine statische Metrik, die auf Graphentheorie basiert. Diese Metrik, abgekürzt mit v(G), gibt die Anzahl der logischen Pfade eines Programms an, indem sie die Anzahl der linearen unabhängigen Zyklen des Kontrollgraphen kalkuliert [McC76]. Jeder Knoten des Graphen entspricht einer Anweisung und jede Kante entspricht dem Übergang zwischen den Knoten [McC76]. Die zyklomatische Komplexität lässt sich dann wie folgt berechnen:

$$v(G) = e - n + 2p$$

wobei:

- e: die Anzahl der Kanten ist.
- n: die Anzahl der Knoten ist.

- p: die Anzahl der zusammenhängenden Komponenten des Graphen ist.

Das folgende Beispiel im Listing 3.1 dient der Erläuterung der Berechnung des v(G)-Werts einer Methode:

Listing 3.1: min-Methode

```
1 public int min(int a, int b) {
2     int min = a;
3     if (b < min){
4       min = b;
5     }
6     return min;
7 }
```

Der Flussgraph der Methode hat 4 Knoten, 4 Kanten und eine zusammenhängende Komponente (der Graph selbst), deshalb hat die Methode eine zyklomatische Komplexität von 2.

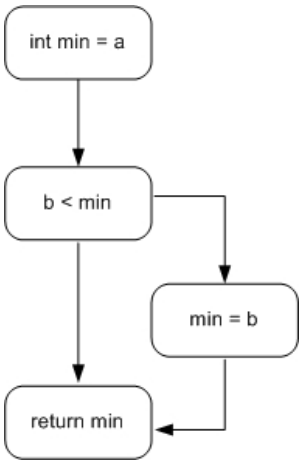

Abbildung 3.1: Beispielmethode für zyklomatische Komplexität von 2.

McCabe hat auch in [McC76] gezeigt, dass die Berechnung der v(G) vereinfacht werden kann, indem man die Anzahl der konditionellen Zweige des Flussdiagramms plus eins zählt. In dem obigen Beispiel enthält die Methode eine Verzweigung durch eine *if*-Anweisung, demnach ist der Wert der v(G) 2. Eine hohe v(G) bedeutet, dass das entsprechende Modul viele Pfade beinhaltet und deshalb schwerer zu verstehen und zu testen da die Testfälle für eine Methode mindestens so viel sind wie der Wert der v(G). McCabe schlug vor, dass die v(G) einer Methode einen Wert bis 10 betragen sollte. Grady berichtet in [Gra94] das Ergebnis eines in Hewlett-Packard durchgeführten Experiment. In diesem Experiment wurden 830.000 Fortran Codezeilen hinsichtlich der Beziehung zwischen Änderungen und den v(G)-Werten analysiert. Es stellt sich heraus, dass die Anzahl der Änderungen proportional zu der Anzahl der logischen Pfade eines Moduls ist und ein Grenzwert für die v(G) eines Moduls bis 14 akzeptabel ist.

Ursprünglich kam die Definition der zyklomatischen Komplexität aus der Zeit des prozeduralen Paradigmas aber diese Metrik sollte von Programmiersprachen unabhängig sein, da nur die konditionellen Zweige des Kontrollflusses gezählt werden müssen. In einem White Paper von Cullmann und Lambertz wird eine Strategie zur Berechnung der zyklomatischen Zahl für die C++ Programmiersprache eingeführt [CL07]. Folgende

Sprachkonstruktionen werden festgestellt, die zu dem Wert der v(G) eines C++ Moduls beitragen: *if, for, while, case, catch,* &&, ||, ?, *#if, #ifdef, #ifndef, #elif.* Aufgrund ihrer Unabhängigkeit von Programmiersprachen findet McCabes zyklomatische Komplexität ihren Einsatz auch in Metriken-Suiten für OOP [SK03]. Eine der bekanntesten ist die Metriken-Suite von Chidamber und Kemerer, die in dem Abschnitt 3.3 detailliert beschrieben werden.

3.2.3 Halsteads Software Science-Metriken

Die Metriken von Maurice Halstead waren die Ergebnisse eines der frühesten Versuche, die Größe und Komplexität von Software über die LOC hinaus zu erfassen [FP97]. Halsteads Metriken wurden teilweise anhand linguistischer und psychologischer Disziplinen entwickelt. Die Bausteine sind zwei sich gegenseitig ausschließenden Entitäten, woraus jedes Programm besteht: die Operatoren und die Operanden.

- n1: die Anzahl der verschiedenen Operatoren.
- n2: die Anzahl der verschiedenen Operanden.
- N1: die Gesamtanzahl der Operatoren.
- N2: die Gesamtanzahl der Operanden.

Beispielsweise hat die folgende Anweisung 3 verschiedene Operanden (y, x, 1) und 3 verschiedene Operatoren (=, *, +) ((vgl. Listing 3.2)). In diesem Beispiel ist die Gesamtanzahl der Operatoren 3 während die Gesamtanzahl der Operanden 4 ist da x zweimal vorkommt :

Listing 3.2: Halstead Metrik-Beispiel

```
1 y = x*x + 1;
```

Aus diesen Bausteinen lassen sich die folgenden Metriken ableiten:

- Programmlänge (eng. Program length): N = N1 + N2.
- Vokabulargröße (eng. Vocabulary size): n = n1 + n2.
- Volumens des Programm (eng. Program volume): V = N * log2(n).
- Schwierigkeitsgrad (eng. Difficulty level): D = (n1/2)*(N2/n2).
- Programmniveau (eng. Program level): L = 1/D.
- Implementierungsaufwand (eng. Effort to implement): E = V*D.
- Implementierungszeit (eng. Implementation time): T = E/18.
- Anzahl der ausgelieferten Bugs (eng. Number of delivered bugs): B = $(E^{2/3})/3000$.

Die Metriken von Halstead bekamen großes Interesse von Wissenschaftlern. Einige Arbeiten zeigen, dass die theoretischen Voraussagen der Implementierungszeit und der Anzahl der ausgelieferten Bugs mit einer erstaunlichen Quote korrekt [FL78] sind. Fitzsimmon und Love finden eine hohe Korrelation zwischen der Implementierungsaufwand-Metrik (E) und verschiedenen wichtigen Faktoren eines Softwareprojekts z.B. der Programmierungszeit, der Verständlichkeit des Programms, der Anzahl der Fehler während der Programmierung [FL78]. Sie fügen weiter hinzu, dass die Metriken von Halstead eher für große Softwareentwicklungsprojekte als für einzelne Programme geeignet sind. O'Neal zeigt in einer Studie die Übereinstimmung zwischen der Volumen-Metrik (V) und der intuitiven Vorstellung der Entwickler von Komplexität [O'N93]. Die Volumen-Metrik korreliert sehr gut mit der Anzahl der Änderungen an 44 Programmen über einem Zeitraum von einem Jahr [O'N93].

Im Gegensatz zu der zyklomatischen Komplexität-Metrik von McCabe lassen sich die Halsteads Metriken nicht problemlos in das objektorientierte Paradigma integrieren. Eines der umstrittensten Probleme bezieht sich auf die Aufzählung der Operatoren und Operanden. Genauer gesagt, den Metriken fehlt es an einer präzisen Definition von Operatoren und Operanden, die Halstead nie tatsächlich bereitgestellt hat [Sal82]. Er glaubt, dass gewisse Symbole intuitiv deutlich der normalen Definition von Operatoren konform sind. Er betrachtet diese Operatoren als Verben der Programmierung, die den Zustandswechsel der Nomina, der Operanden, hervorruft. Die Operanden sind dann Entitäten, die Daten enthalten. Eine Definition von Operanden ist seiner Meinung nach intuitiv klar und braucht daher keine Erklärungen [Sal82]. Es sei allerdings darauf hingewiesen, dass ursprünglich die Metriken von Halstead nur für die niedrigen Programmiersprachen wie z.B. Assembler gedacht sind und bei diesen Sprachen kann man tatsächlich leicht die Operanden und Operatoren eines Programms unterscheiden. Einige Arbeiten haben später versucht, eine praktische Definition für höhere Programmiersprachen einzubringen [Sal82, MF82, LKO04], allerdings bleibt das Problem immer noch bestehen.

Zusätzlich dazu werden Halsteads Metriken oft wegen des Mangels an sowohl theoretischer als auch empirischer Basis kritisiert [MVL03a]. Manche Metriken von Halstead basieren auf linguistische und psychologische Grundlagen aber es scheint, dass sich seine Beobachtungen für die Verständlichkeit vom Quellcode und von normalen Texten zu widersprechen [MF82]. Beispielsweise kann es aus dem zweiten Teil der Formel für den Schwierigkeitsgrad (D) geschlussfolgert, dass ein Modul mit großer Anzahl von wenigen verschiedenen Operanden (Worten) schwerer zu verstehen als eins mit kleiner Anzahl von zahlreichen Operanden ist. Dies stimmt nicht mit der normalen Betrachtung der Fassbarkeit von gewöhnlichen Texten wie Büchern oder Artikeln überein, da in diesen Fällen oft davon ausgegangen wird, dass Texte mit mehrfachen Auftreten weniger Worten verständlicher sind. Die restlichen Metriken werden auch stark kritisiert. Fenton und Pfleeger halten sie für intuitiv unplausibel [FP97] während Hamer und Frewin anhand ihres Experiments zeigen, dass Halsteads Metriken keine versprechende Ergebnisse liefern [HF82]. Obwohl Halsteads Metriken tatsächlich später in eine der bekanntesten Metriken-Suite integriert wurde, die später im Abschnitt 3.4.1 eingeführt wird, werden sie aufgrund der oben genannten Probleme nicht in dieser Arbeit zum Einsatz gebracht.

3.3 Wartbarkeit-Metriken für OOP

Die Einführung der OOP, eines der wichtigsten Paradigmenwechsel in dem Softwareentwicklungsgebiet [CK94], stellt neue Anforderungen an die Messung objektorientierter Struktur. Der Leitgedanke der OOP ist die Modellierung der realen Welt bezüglich ihrer Objekte, dabei werden zusammenhängende Daten und Operationen, die die Daten modifizieren, in eine einzelne Einheit, bekannt als Klasse, zusammengepackt. Diese Einstellung differenziert sich stark zu dem traditionellen prozeduralen Ansatz, der sich auf die funktionale Dekomposition konzentriert und Daten und Funktionen als separate Komponenten betrachtet. Angesichts der fundamentalen inhärenten Unterschiede zwischen den beiden Programmierparadigmen ist es nicht überraschend zu finden, dass sich die traditionellen Metriken nicht nach solchen Begriffen wie z.B. Klassen, Vererbung, Kapselung oder Nachrichtenaustausch richten [CK91]. Da die Metriken für das prozedurale Paradigma nicht die grundlegenden Konzepte der OOP erfassen können, ist es erforderlich, neue Metriken zu entwickeln, die zum Messen der besonderen Aspekte der OOP gedacht werden.

Die bekannteste Arbeit in dem Gebiet der OOP-Metriken gehört zu der Metriken-Suite von Chidamber und Kemerer [Män03], die auch als CK-Metriken bezeichnet wird. Die Grundlagen der CK-Metriken beziehen sich auf ein Set der ontologischen Prinzipien, die von Bunge vorgeschlagen und später auf objektorientierte Systeme angewendet werden [FP97]. Bei dieser Ontologie wird die Welt auf diese Weise betrachtet, dass sie aus substanziellen Individuen besteht, welche eine endliche Menge von Eigenschaften besitzen. Ein substanzielles Individuum und seine Eigenschaften stellen ein Objekt dar und eine Menge von Objekten, die sich gemeinsame Eigenschaften teilen, bildet eine Klasse. Weitere Attribute der OOP wie Kohäsion, Kopplung, Gültigkeitsbereich einer Eigenschaft oder Komplexität eines Objekts können sich anhand der Ontologie von Bunge beschreiben lassen. Darauf werden folgende Metriken für die Metriken-Suite definiert: *Weighted Methods Per Class* (WMC, dt. gewichtete Methoden per Klasse), *Depth of Inheritance Tree* (DIT, dt. Tiefe des Vererbungsbaums), *Number of Children* (NOC, dt. Anzahl der Kinder), *Coupling Between Object Classes* (CBO, dt. Kopplung zwischen Klassen), *Response For a Class* (RFC, dt. Beantwortungsset einer Klasse) und *Lack of Cohesion in Methods* (LCOM, dt. Mangel an Kohäsion zwischen Methoden).

Ursprünglich befassen die CK-Metriken nicht direkt mit der Prognose über die Wartbarkeit sondern sie beziehen sich eher auf den Komplexitätsaspekt von Software. Erst später werden sie von Li und Henry zusammen mit einigen Size-Metriken benutzt um die Fähigkeit zu diesem Zweck zu testen [LH93]. Es stellt sich heraus, dass es eine solide Beziehung zwischen der Wartbarkeit eines Programms und den Metriken gibt. Insbesondere werden die Kohäsion- und die Kopplung-Metriken von vielen Wissenschaftlern und Software-Experten als der wichtigste quantifizierbare Indikator betrachtet [CK94, HM95, FP97, DJ03, DBDV04, DC04]. Eine Software mit hoher Kohäsion und loser Kopplung ist viel versprechend, dass sie auch leicht zu ändern und zu warten. Deswegen werden die Begriffe von Kohäsion und Kopplung und deren entsprechende Metriken in dem nächsten Abschnitt genauer erklärt.

3.3.1 Kohäsion

Der Begriff Kohäsion wird erst in die strukturierte Programmierungstechnik von Stevens et al. eingeführt und beschreibt, wie eng Elemente innerhalb eines Moduls verbunden sind [SMC74]. In der Welt der OOP ersetzen die Klassen die Module mit Methoden und Attributen als ihre Elemente. In diesem Kontext stellt dann die Kohäsion einer Klasse die Qualität der von der Klasse erfassten Abstraktion dar. Eine gute Abstraktion weist typischerweise eine hohe Kohäsion auf [HM95]. Die erste Kohäsion-Metrik für OOP wird von Chidamber und Kemerer definiert und bildet eine inverse Messung für Kohäsion. Diese Metrik wird als Lack of Cohesion in Methods (LCOM) bezeichnet und wie folgt berechnet:

Für eine Klasse C_1 mit n Methoden M_1, M_2, ..., M_n. Sei I_j = die Menge der Instanzvariablen, die von der Methode M_j benutzt werden (gelesen oder geschrieben). Es gibt dann n solche Mengen I_1, ..., I_n. Sei $P = \{(I_i, I_j) | I_i \cap I_j = \oslash\}$ und $Q = \{(I_i, I_j) | I_i \cap I_j \neq \oslash\}$. Seien alle Mengen $\{I_1\}, ..., \{I_n\}$ gleich $\oslash$ dann P = $\oslash$.

$$LCOM = \begin{cases} |P| - |Q|, & wenn |P| > |Q| \\ 0, & sonst \end{cases} .$$

Darüber hinaus beschreiben Chidamber und Kemerer die durch ihre LCOM-Metrik festgestellten Eigenschaften wie folgt:

- Hohe Kohäsion zwischen Methoden einer Klasse fördern die Kapselung.
- Niedrige Kohäsion impliziert eine hohe Komplexität und folglich die Wahrscheinlichkeit von Fehlern. Eine Klasse mit niedriger Kohäsion sollte dann in zwei oder mehrere Klassen unterteilt werden.
- Problematische Stellen in dem Design der Klassen kann identifiziert werden, indem die Unterschiede zwischen den Methoden anhand der Metrik bemessen werden.

Zur Erläuterung der oben genannten Begriffe dient das Beispiel im Pseudocode-Format im Listing 3.3. Die Klasse X hat 3 Methoden M1, M2, M3 und 5 Attributen A1, A2, A3, A4, A5.

Listing 3.3: LCOM-Beispiel

```
1 public class X{
2   String A1, A2, A3, A4, A5;
3   void M1(){ benutzt A1, A2, A3; ...}
4   void M2(){ benutzt A2, A3, A4; ...}
5   void M3(){ benutzt A5; ...}
6 }
```

Wie die Abbildung 3.2 veranschaulicht, gibt es zwei Paare von Methoden, die keine gemeinsamen Instanzvariablen haben ({m1,m3} und {m2,m3}) und ein Paar von Methoden, die sich gemeinsamen Instanzvariablen (A2, A3) teilen ({m1,m2}). Demzufolge ist die LCOM-Metrik von Klasse X gleich 1. LCOM ist eine inverse Metrik, das heißt ein hoher Wert der LCOM-Metrik impliziert eine niedrige Kohäsion und umgekehrt. Obwohl die grundlegende Idee hinter der Definition von LCOM vernünftig erscheint, kann die resultierende Metrik in vielen Fällen den Verstand über die Eigenschaft nicht reflektieren [HM95]. Gegeben seien zwei Klassen X und Y, die in Abbildung 3.3 definiert sind:

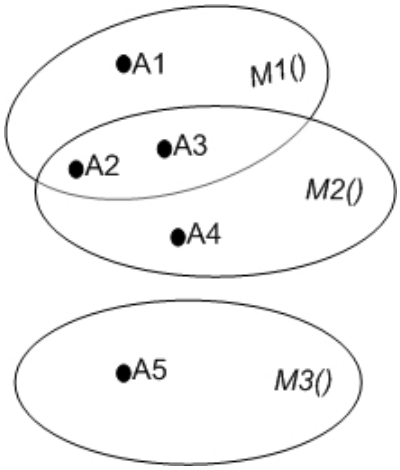

Abbildung 3.2: Venn Diagramm-Darstellung der Klasse X

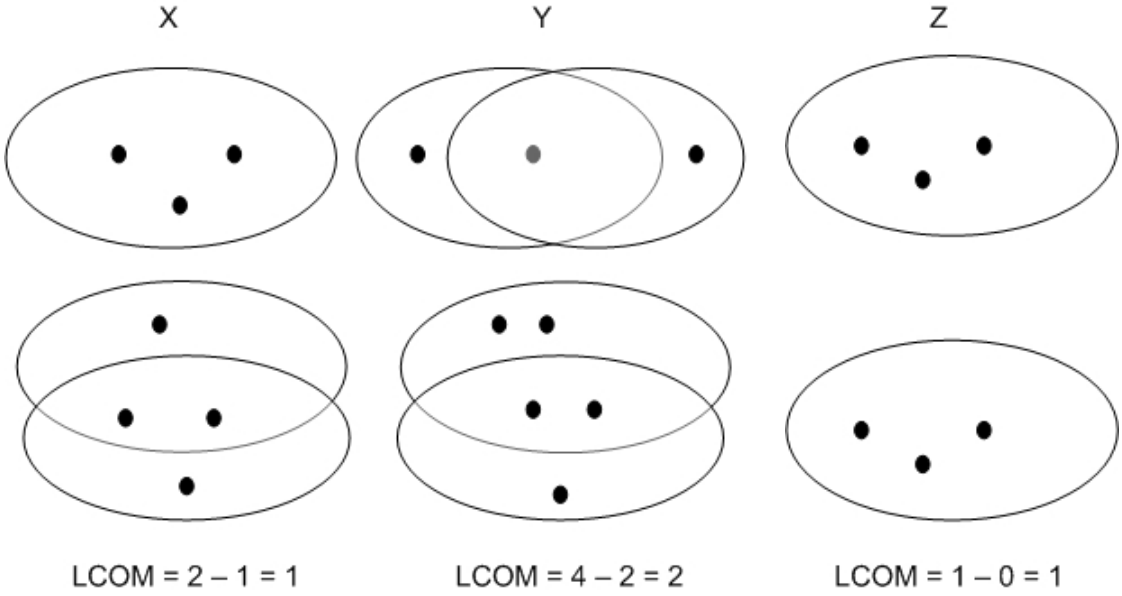

Abbildung 3.3: LCOM-Anomalie

Abbildung 3.4: LCOM Klasse Z

Intuitiv betrachtet sind die zwei Klassen gleich inkohärent: ohne die semantischen Informationen zu kennen, sollte davon ausgegangen, dass beide Klassen aufgeteilt werden müssen obwohl die Klasse X einen niedrigen Wert für die LCOM-Metrik hat. Man kann argumentieren, dass das Hinzufügen der neuen Methode in Klasse Y den erhöhten Wert verursacht hat. Jedoch ist diese Schlussfolgerung inkonsistent in dem Falle der Klasse Z 3.4, die einen gleichen LCOM-Wert wie Klasse X hat aber ohne eine überlappende Methode. Demzufolge ist die Definition für Kohäsion-Metrik von Chidamber und Kemerer das Ziel vieler Kritiken. Simon et al. fasst die Nachteile der LCOM-Metrik in [SLL99] zusammen:

- LCOM zieht nur die Benutzung der Attribute in Betracht und liefert somit nur eine spezifische Perspektive nämlich die Kohäsion durch die Zugriffe auf gemeinsame Attribute. Dies kann in vielen Fällen zu irrationalen Schlussfolgerungen führen, dass zwei Methoden, die sich keine gemeinsamen Attribute teilen, überhaupt nicht

zusammenhängend sind.

- Das Wertespektrum der Metrik wird von 0 bis 1 beschränkt.
- Die Aufsummierung der überlappenden Methoden bzw. der disjunkten Methoden hindert daran, tiefere Interpretationen zu machen. Ein schlechter Wert der LCOM erklärt nicht, welche Methoden diesen Wert verursachen, die dann in separate Klassen verschoben werden müssen um die Kohäsion verbessern zu können.

Viele andere Autoren haben auch ihre eigenen Interpretationen des Begriffs Kohäsion eingeführt. Eine der frühesten verbesserten Versionen von LCOM ist die von Li und Henry [LH93]. Sie wird wie folgt informell definiert:

> LCOM = Anzahl der disjunkten Sets der lokalen Methoden; keine zwei Sets überschneiden sich miteinander; jede zwei Methoden eines gemeinsamen Sets teilen sich mindestens ein Attribut; der Wert variiert von 0 bis N, wobei N eine positive Ganzzahl ist.

Intuitiv kann man feststellen, dass ein Wert k dieser Metrik bedeutet, dass die Klasse möglicherweise in k kleinere, kohäsivere Klassen unterteilt werden sollte. Wendet man diese Definition auf die oben genannten Beispiele an, erhält man bei allen einen Wert von 2, welcher dem intuitiven Begreifen von Kohäsion auch entspricht (alle diese 3 Beispielklassen sollen in zwei kleinere Klassen unterteilt werden). Allerdings enthält diese Definition von Li und Henry immer noch einige semantische Schwachstellen [HM95]: Eines der Prinzipien des objektorientierten Designs verlangt, dass Zugriffe auf private Attribute durch spezifizierte lesende bzw. schreibende Methoden beschränkt werden sollen. Der Definition von Li und Henry nach kann eine kohäsive Klasse trotzdem einen hohen LCOM-Wert haben (welcher impliziert dass die Klasse inkohäsiv ist) da alle „realen“ Methoden die Klassenattribute nicht mehr direkt zugreifen und daher sich keine gemeinsamen Attribute teilen. Darüber hinaus kommt es in der Praxis oft vor, dass eine Methode auf überhaupt keine Attribute zugreift (weder direkt noch durch „Get and Set“-Methoden) und durch andere Basismethoden der Klasse aufgebaut wird. Die Definition von Li und Henry kann solche Beziehungen nicht erfassen und klassifiziert somit solche Methoden als strukturell unverbunden mit dem Rest der Klasse.

Simon et al. führen in [SLL99] eine Metrik für Kohäsion ein, die Distanz-basierte Kohäsion-Metrik (engl. distance based cohesion- DBC), die die oben geschilderten Probleme der Metriken von Chidamber und Kemerer bzw. Li und Henry überwinden kann. Die Definition von Simon et al. basiert auf die Ontologie von Bunge über die Ähnlichkeit zwischen Entitäten. Der Ähnlichkeitsgrad zweier Entitäten lässt sich durch ihre gemeinsamen Eigenschaften beschreiben. Alle Entitäten sind somit ähnlich in gewissen Mengen von Eigenschaften. Formell definiert kann der Ähnlichkeitsgrad zwischen zwei Entitäten x und y in Bezug auf eine Menge von Eigenschaften B wie folgt sein:

$$s(x,y) := \frac{|(p(x) \cap p(y)) \cap B|}{|(p(x) \cup p(y)) \cap B|}$$

indem p(x) := $\{P_i \mid$ x besitzt die Eigenschaft $P_i\}$.

Folgende Schlussfolgerungen können durch die Definition gezogen werden:

- Die Ähnlichkeit ist von dem Gesichtspunkt abhängig: aus zwei unterschiedlichen Gesichtspunkten kann der Ähnlichkeitsgrad für zwei gewisse Entitäten von ähnlich zu different variieren.
- Ohne die Menge von betrachteten Eigenschaften zu nennen liefert die Berechnung der Ähnlichkeit keine sinnvolle Ergebnisse.
- Der Begriff der Ähnlichkeit ist nur zwischen den Paaren der Entitäten gültig. Es gibt keine Ähnlichkeit für eine einzelne Entität.
- Die Menge der betrachteten Eigenschaften B muss für jedes Paar der Entitäten kompatibel sein, das heißt die Entitäten müssen mindestens eine Eigenschaft besitzen. Ohne diese Voraussetzung wird die Menge B als Ähnlichkeitsinkompatibel (engl. similarity incompatible) bezeichnet und der Begriff Ähnlichkeit existiert nicht hinsichtlich solcher Eigenschaftsmengen.

Aus der Definition des Ähnlichkeitsgrads kann man die Distanz zwischen zwei Entitäten bezüglich einer Menge der betrachteten Eigenschaften B entsprechend wie folgt definieren:

$$dist(x, y) := 1 - \frac{|(p(x) \cap p(y))|}{|(p(x) \cup p(y))|}$$

indem p(x) := $\{P_i \in B \mid$ x besitzt die Eigenschaft $P_i\}$.

Dem Konzept der Distanz nach bedeutet, dass Entitäten mit niedrigen Distanzen zu einander kohäsiv sind während Entitäten mit höheren Distanzen weniger kohäsiv sind. Bevor man die Metrik zu einem bestimmten Zweck einsetzen kann muss Folgendes festgelegt werden [SSL01]:

- welche Typen der Entitäten werden bemessen? In OOP gibt es folgende Entitätsklassen, die auch als Abstraktionsebenen bezeichnet werden:

Ebene Nr.	Ebenenname	betrachtete Entitäten.
1	Variablenebene	Attribute des Systems.
2	Methodenebene	Methoden des Systems.
3	Klassenebene	abstrakte Datentypen, die die entsprechenden Attribute und Methoden kapseln.
4	Systemebene	das ganze System

- welche Eigenschaften sollen in Betracht gezogen werden? z.B. die Eigenschaften können die Menge der Oberklassen und Unterklassen sein, die eine Klasse hat oder in meisten Fällen die Benutzung von anderen Entitäten.

Um die Wartbarkeit von Software zu gewährleisten wird gefordert, zusammenhängende Entitäten in einem gemeinsamen Modul zu kapseln. Deshalb sollte die Kohäsion-Metrik auf die Benutzungsbeziehung basieren [SSL01]. Weiterhin muss die Metrik auch für die Ursachenanalyse im Falle der schlechten Kohäsion-Werte geeignet sein, demzufolge werden die Variablen- und Methodenebene in der Metrik berücksichtigt werden. Die Menge der zu betrachtenden Eigenschaften B für eine Methode bzw. ein Attribut gibt die folgende Tabelle an:

B_m für eine Methode	B_a für ein Attribut
{die Methode selbst} ∪ {alle von dieser Methode direkt aufgerufenen Methoden} ∪ {alle von dieser Methode direkt benutzten Attribute}	{das Attribut selbst} ∪ {alle Methoden, die das Attribut zugreifen}

Tabelle 3.1: Die Menge der betrachteten Eigenschaften einer Methode bzw. eines Attributes

Das Beispiel im Listing 3.4 dient der Erläuterung von Berechnung der Kohäsion zwischen Entitäten unterschiedlicher Klassen. Angenommen seien zwei Klassen X und Y, die folgendermaßen im Pseudocode-Format definiert sind:

Listing 3.4: Definitionen von Klassen X und Y

```
1  public class X{
2    static String X1, X2;
3    void MX1(){ benutzt X1, MX2; ...}
4    void MX2(){ benutzt X2; ...}
5  }
6  public class Y{
7
8    static String Y1, Y2;
9    void MY1(){ benutzt X1, MX2; ...}
10   void MY2(){ benutzt Y1, Y2; ...}
11 }
```

Die resultierende Ergebnisse der Kohäsion von diesen Klassen gibt die folgende Tabellen an:

	MX1()	MX2()	X1	X2	MY1()	MY2()	Y1	Y2
MX1()	0							
MX2()	0.75	0						
X1()	0.5	1	0					
X2()	0.75	0	1	0				
MY1()	0.5	0.75	0.5	0.75	0			
MY2()	1	1	1	1	1	0		
Y1()	1	1	1	1	1	0.33	0	
Y2()	1	1	1	1	1	0.33	0.67	0

Tabelle 3.2: Kohäsion zwischen Entitäten der Klasse X und Y

Aus den Kohäsion-Werten fällt man die Methode MY1() auf da sie die einzige Methode ist, die schlechte Kohäsion-Werte zu Entitäten der enthaltenen Klasse hat (alle sind 1). Zugleich teilt sie Gemeinsamkeiten mit Attributen und Methoden der Fremdklasse X: Zu den 4 Entitäten der Klasse X (MX1, MX2, X1, X2) hat MY1 gute Kohäsion-Werte (durchschnittlich 0.625). An dieser Stelle sollte MY1() in Klasse X extrahiert werden und somit können sich die Kohäsion beider Klassen verbessern.

3.3.2 Kopplung

Chidamber und Kemerer gelten als die Erster, die die Definition von Kopplung zwischen Objekten geliefert haben [HM95]. Diese Definition spezifiziert die allgemeine Definition von Wand und Weber [WW90]: die Kopplung zwischen zwei Entitäten existiert wenn zumindest eine Entität auf die andere wirkt. Ein Entität X hat eine Wirkung auf eine andere Entität Y wenn X die Historie von Y beeinflusst, wobei die Historie von Y die zeitlich geordneten Zustände von Y ist. Die Interpretation von Chidamber und Kemerer kann wie folgt formell beschrieben werden [CK91]:

Seien X = $\langle\ x,\ p(x)\ \rangle$ und Y = $\langle\ y,\ p(y)\ \rangle$ zwei Objekte.

p(x) = $\{\ S_X\ \} \cup \{\ I_X\ \}$

p(y) = $\{\ S_Y\ \} \cup \{\ I_Y\ \}$

wobei $\{\ S_i\ \}$ die Menge der Methoden und $\{\ I_i\ \}$ die Menge der Attribute des Objekts i ist.

Der Definition von Wand und Weber nach bildet jede Aktion von $\{\ S_X\ \}$ auf $\{\ S_Y\ \}$ oder $\{\ I_Y\ \}$ eine Kopplung und umgekehrt. Zwei Objekte sind dann gekoppelt wenn eine Methode des einen die Methoden oder die Attribute des anderen benutzt. Als Metriken für Kopplung führen Chidamber und Kemerer die CBO-Metrik ein, die wie folgt definiert ist:

> CBO-Metrik für eine Klasse ist die Anzahl der anderen Klassen, mit denen die Klasse gekoppelt ist.

Die CBO-Metrik beschreibt nämlich die Abhängigkeiten zwischen Objekten unterschiedlicher Klassen. Dadurch können einige Eigenschaften für Kopplung festgestellt werden:

- exzessive Kopplung zwischen Objekten außerhalb der Vererbungshierarchie kann die Wiederverwendung hindern da sie das Modularisierungsprinzip vom objektorientierten Design verletzt. Je mehr unabhängig ein Objekt ist, desto leichter kann es in anderen Anwendungen wiederverwendet werden.
- Kopplung ist nicht assoziativ, das heißt sei A mit B gekoppelt und sei B mit C gekoppelt, es impliziert nicht dass A mit C auch gekoppelt sein kann.
- Eine niedrige Kopplung sollte gehalten werden um bessere Modularisierung und Kapselung zu unterstützen. Je mehr Kopplungen existieren desto mehrere Arbeiten müssen erledigt werden wenn eine Änderung durchgeführt wird. Dies beeinträchtigt die Wartbarkeit einer Software.
- Hohe Kopplung fordert zugleich komplexere und härte Teste.

Zusätzlich dazu definieren Chidamber und Kemerer noch die RFC-Metrik, die sich auch auf den Begriff Kopplung bezieht [CK91]:

RFC = $|\ RS\ |$ wobei RS das Beantwortungsset der Klasse ist.

RS = $\{\ M_i\ \} \cup_{alle\ n} \{\ R_i\ \}$

wobei $\{\ M_i\ \}$ = alle Methoden in der Klasse

und $\{\ R_i\ \}$ = die Menge aller Methoden, die von $\{\ M_i\ \}$ aufgerufen werden.

RFC erfasst dann die Diensten, die die Klassen den anderen Klassen bietet und auch die die Klasse von anderen verlangt. Dadurch lässt sich die Kommunikation zwischen den Klassen beschreiben, die auch zu dem Konzept der Kopplung gehört. Die Eigenschaften der RFC-Metrik wird in [CK91] analysiert:

- Wenn auf eine Nachricht viele Methoden aufgerufen werden müssen, können das Testen und Debuggen schwierig sein da die Fehler schwerer festzustellen sind.
- Die Komplexität eines Objekts ist proportional zu der Anzahl der Methoden, die von diesem Objekt durchgeführt werden können.
- Je größer die Anzahl der externen Methoden ist desto mehr wird es verlangt, um das Programm zu verstehen.

Die Definitionen der CBO- und RFC-Metrik von Chidamber und Kemerer werden von einigen Autoren kritisiert. Einige Hauptpunkte befinden sich in der Arbeit von Hitz und Montazeri [HM95]:

- Die CBO-Metrik berücksichtigt die Stärke der Kopplung nicht. Obwohl die Kopplung eine Beziehung zwischen zwei Objekten definiert wird, wird sie als Metrik auf der Anzahl der gekoppelten Klassen aggregiert, welches impliziert, dass alle Basiskopplungen gleich stark sind. Dies gilt im Allgemein natürlich nicht: es gibt unterschiedliche Typen von Kopplung mit unterschiedlichen Stärken [LHR88] z.B eine Kopplung durch direkten Zugriff auf Attribute einer Klasse wird schlechter betrachtet als die durch „Get and Set"-Methoden.
- Die Kopplung durch Vererbung wird nicht betrachtet: Objekte einer Klassen können auf die Attribute oder die Methoden der Oberklasse zugreifen, eine Kopplung wird auch dadurch gebildet. Diese Art von Kopplung verletzt die Kapselung einer Klasse und wird somit als eine der schlechtesten Kopplungsarte angesehen [Sny86]

In der Literatur werden zahlreiche andere Interpretationen für den Begriff der Kopplung von anderen Autoren eingeführt [LH93, HM95]. Jedoch werden die CBO- und RFC-Metrik von Chidamber und Kemerer am breitesten akzeptiert und weithin als Metriken für die Kopplung implementiert [MP05]. Viele empirische Studien haben nachgewiesen, dass die CBO- und RFC-Metrik in der Tat ein guter Vorhersager für die Wartbarkeit, die Fehleranfälligkeit und die Wiederverwendbarkeit sind [BBM96, BWIL99, SK03]. Deswegen werden auch diese Metriken zur Ermittlung von Wartbarkeit in dieser Arbeit benutzt.

3.4 Metriken-Suite für die Wartbarkeit

Viele Attribute eines Software-Systems können nicht direkt quantifiziert werden, da die Konzepte hinter diesen Attributen oft kompliziert sind [FP97]. Stattdessen muss man bestimmte Eigenschaften direkt bemessen und aus den erhaltenen Ergebnissen können die Schlussfolgerungen über die Qualität des entsprechenden Attributs abgeleitet werden [WH88, FP97, DC04]. Die Wartbarkeit einer Software ist eines solcher Attribute. Deswegen werden in der Literatur viele Metriken-Suiten entwickelt, die zum Prognostizieren der Wartbarkeit dienen. Jede Metrik einer Metriken-Suite erfasst dann einen Aspekt der Wartbarkeit. Im Folgenden werden einige bedeutende Metriken-Suiten eingeführt.

3.4.1 Maintainability Index

Eine der frühesten und auch bekanntesten Arbeiten unter den Metriken-Suiten gilt der Maintainability Index (MI, dt. Index der Wartbarkeit) von Oman und Hagemeister [OH92]. Aus dem Zustand des Quellcodes liefert MI eine objektive Ermittlung über die Wartbarkeit des Systems. Diese Metriken-Suite besteht aus einigen traditionellen Metriken, die in einen einzigen Wert zusammengesetzt werden. Originell werden folgende Metriken von Oman und Hagemeister benutzt: die Halsteads Volumen-Metrik (HV), die zyklomatische Komplexität, die Anzahl der Codezeilen und als optional den Prozentanteil der Kommentare eines Moduls. Entsprechend werden zwei Formeln gebildet, eine mit Berücksichtigung der Kommentare und eine ohne:

$$MI = 171 - 5.2ln(aveV) - 0.23aveV(g') - 16.2ln(aveLOC)$$

wobei

- aveV das durchschnittliche Halsteads Volumen per Module ist,
- aveV(g') die durchschnittliche zyklomatische Komplexität per Module ist,
- und aveLOC die durchschnittliche Anzahl der Codezeilen per Module ist,

und

$$MI = 171 - 5.2ln(aveV) - 0.23aveV(g') - 16.2ln(aveLOC) + 50.0sin(\sqrt{2.46 * perCM})$$

wobei

- aveV das durchschnittliche Halsteads Volumen per Module ist,
- aveV(g') die durchschnittliche erweiterte zyklomatische Komplexität per Module ist,
- aveLOC die durchschnittliche Anzahl der Codezeilen per Module ist,
- und perCM der durchschnittliche Prozentanteil der Kommentare per Module ist.

Das Grundprinzip der Auswahl der Metriken dient der Konstruktion einer robusten, leicht zu bedienende Metrik. Die Basismetriken sollten dann folgende Eigenschaften erfassen:

- die Dichte der Operatoren und Operanden (wie viele Variablen und wie werden sie benutzt)
- die logische Komplexität (wie viele Ausführungspfade gibt es in dem Code)
- die Größe (wie viel Code)
- die menschliche Einsicht (Kommentare im Code)

Die obigen polynomiellen Formeln ergeben sich durch die Tests über zahlreiche Software-Systeme von Hewlett-Packard von ungefähr 50 Regressionsmodellen und werden durch subjektive Evaluierung von Experten kalibriert [CALO94]. Jedes Regressionsmodell besteht aus verschiedenen Software-Metriken. Das Ziel ist ein einfaches und trotzdem generisches Modell, welches auf unterschiedliche Software-Systeme angewendet werden kann. Je größer der Wert von MI ist, desto leichter ist das System zu warten. Viele empirische Studien haben den MI für die Ermittlung der Wartbarkeit von Software getestet. Coleman et al. haben die Wartbarkeit von 11 Software-Systemen analysiert. Laut ihrem Bericht entspricht der MI der Intuition von Experten über die Wartbarkeit der Komponenten der Systeme [CALO94]. Darüber hinaus kann MI in vielen Fällen zusätzliche Informationen liefern, die die Entscheidungen der Entwickler für das Design unterstützen. Heitlager et al. benutzen den MI neben anderen Software-Metriken zur Untersuchung der Wartbarkeit von Software in ihrer Beratungsarbeit [HKV07]. Einige kommerzielle Software-Qualität-Tools wie JHawk[1] oder Testwell CMT[2] implementieren MI als Teil eines Frameworks zur Ermittlung der Wartbarkeit.

Obwohl der MI viel versprechend erscheint, bekommt er jedoch auch viele Kritiken [Wel01, HKV07]. Im Folgenden werden die Beschränkungen des MI detailliert beschrieben [HKV07].

- Die Ursachenanalyse (engl. root-cause analysis): Um in der Praxis akzeptiert werden zu können muss eine numerische Metrik im Prinzip dem Anwender die Fähigkeit geben, feststellen zu können, welche Änderungen an dem System zu einer Änderung an dem Wert der Metrik führen. Diese Eigenschaft besitzt der MI nicht. Weil MI eine zusammengesetzte Metrik ist, ist es oft schwer zu determinieren, welche Faktoren einen bestimmten Wert verursacht haben. Eigentlich basieren die Formeln für MI vollkommen auf statischen Korrelationen deswegen könnte es sein, dass überhaupt keine kausale Relation zwischen den Basismetriken und dem abgeleiteten Wert des MI. Diese Ambiguität führt dazu, dass wenn der Wert von MI niedrig ist, ist es nicht sofort klar, was zu unternehmen ist, um den Wert zu verbessern.

- Die durchschnittliche Komplexität: Der Durchschnitt der zyklomatischen Komplexität aller Module dient als eine Basismetrik der Konstruktion des MI. Die Gültigkeit dieses Werts ist fraglich insbesondere für Software-Systeme, die auf OOP basieren. Der Grund ist, dass objektorientierte Systeme viele get- bzw. set-Methoden benutzen um die Kapselung der Klassen zu realisieren. Die durchschnittliche Komplexität wird tendenziell niedrig da alle get- bzw. set-Methoden die Komplexität von 1 haben. Jedoch kommt es in der Praxis oft vor, dass Probleme des Warten oft mit den wenigen Ausreißern passieren, die eine besonders hohe Komplexität haben.

- Die Berechenbarkeit: Die Halsteads Volumen-Metrik ist schwer zu definieren und zu berechnen. Es gibt keinen Konsensus über die Definition von Operanden und Operatoren für populäre Programmiersprachen wie C++ oder Java. Demzufolge ist die Anwendung der Halsteads Metriken in der Praxis nur beschränkt. Selbst wenn

[1] http://www.virtualmachinery.com/jhawkprod.htm
[2] http://www.verifysoft.com/

es solche Definition gibt, ist die Berechnung der Halstead-Metriken immer noch relativ kompliziert. Die Anzahl der Operatoren und Operanden kann nur festgestellt werden, indem man alle Klassen vollständig und korrekt in Tokens übersetzt. In manchen Fällen ist die Tokenisierung lediglich nicht ausreichend sondern zusätzliche syntaktische und semantische Analysen sind erforderlich.

- Kommentare: Wenn es um Software-Wartung geht, werden Kommentare oft als ein zweischneidiges Schwert betrachtet [Wel01]. Intuitiv ist gut dokumentierter Quellcode leichter zu warten als der nicht-dokumentierte. Allerdings können Kommentare, die nicht mit der Entwicklung des zugehörigen Codes halten, die Wartbarkeit verschlechtern. Darüber hinaus kommt es oft in der Praxis vor, dass Kommentare eigentlich nur auskommentierter Code sind und selbst wenn sie wirkliche Kommentare von Entwickler sind, beziehen sie sich oft auf vorherige Versionen des Codes. Wahrscheinlich wenn die Autoren des MI die Kommentar-Metrik als optional setzen, wollten sie die Verantwortung in die Hand des Anwenders geben da die Entscheidung ob die Kommentare eines Systems gut oder sinnlos sind kann nur von Menschen getroffen werden.

- Die Verständlichkeit: Es gibt keine logische Argumentationen für die Konstruktion der Formeln des MI. Die Formeln wurden nur von den gesammelten Daten abgeleitet und daher sind sie unverständlich und schwer zu erklären. Dies ist kein guter Punkt wenn die Wartbarkeit einer Software zwischen Stakeholdern diskutiert werden.

Neben den obigen Punkten kann man leicht feststellen, dass die Metriken der Metriken-Suite die besonderen Aspekte der OOP wie Vererbung, Klassen etc. nicht erfassen. Um in OOP eingesetzt werden zu können, müssen die Metriken an OOP-Eigenschaften angepasst werden [Wel01]. Zusammenfassend wird festgestellt, dass wegen der fehlenden Eigenschaften, die oben genannt werden und tatsächlich zur Entwicklung des Prototyps erforderlich sind, wird die MI-Metriken-Suite nicht als Mittel zur Ermittlung der Wartbarkeit in dieser Arbeit benutzt.

3.4.2 Die Metriken-Suite von Li und Henry

Um die Beziehungen zwischen der Wartbarkeit eines Systems und den Metriken herauszufinden benutzen Li und Henry [LH93] die Metriken-Suite von Chidamber und Kemerer. 5 von 6 dieser Metriken wurden übernommen: DIT, NOC, RFC, LCOM und WMC. Sie definieren zwei zusätzliche Metriken für das Messen von Kopplung: Message Passing Coupling (MPC, dt. Kopplung durch Nachrichtenaustausch) und Data Abstraction Coupling (DAC, dt. Kopplung durch Datenabstraktion). Darüber hinaus benutzen sie noch zwei Size-Metriken weil sie glauben, dass Size-Metriken auch ein wichtiger Indikator für die Wartbarkeit sind:

SIZE1 = Anzahl der Semikolons in einer Klasse (LOC)

SIZE2 = Anzahl der Attribute + Anzahl der lokalen Methoden.

Anhand der Metriken wurden zwei kommerzielle Systeme untersucht, die mit der objektorientierten Programmiersprache Classic-ADA implementiert wurden. Der Wartungsaufwand wird durch die Anzahl der geänderten Codezeilen in einer Klasse bemessen. Folgende wichtige Schlussfolgerungen werden aus dem Ergebnis gezogen:

- Die gewählten Software-Metriken können den Wartungsaufwand gut voraussagen (87% und 85% der Wartungsaktivitäten in zwei entsprechenden Systemen können durch die Metriken bestimmt werden).
- Die Size-Metriken sind auch ein guter Indikator für den Wartungsaufwand.
- Die objektorientierten Metriken (DIT, NOC, MPC, RFC, LCOM, DAC, WMC) können manche Wartungsaufwände identifizieren, was allein durch die einfachen Size-Metriken unmöglich ist.

3.4.3 Ein Metriken-Modell von Heitlager et al.

Heitlager et al. entwickeln in [HKV07] ein Metriken-Modell für die OOP basierend auf dem ISO 9126 Qualitätsmodell für Software. Zusätzlich dazu haben sie auch die folgenden Voraussetzungen für ein Metriken-Modell festgestellt, welches die Wartbarkeit von Software-Systemen charakterisiert:

- Die Metriken sollen möglichst von Programmiertechnik und Programmiersprachen unabhängig sein, da heutige Software-Systeme oft mehrsprachig sind, d.h. sie benutzen oft mehr als eine Programmiersprache für die Implementierung.
- Jede Metrik muss klar definiert werden damit sie auch leicht berechnet werden kann.
- Die Metriken sollen einfach zu verstehen und zu erklären sein und am besten als ein Kommunikationsmittel zwischen unterschiedlichen Anwendern benutzt werden können.
- Das Modell soll in der Lage sein, die Ursachen deutlich analysieren zu können. Eine kausale Beziehung zwischen den Eigenschaften auf Codeebene und der Qualität auf Systemebene muss definiert werden. Dies ermöglicht die Verbesserung der System-Qualität durch entsprechende Änderungen am Quellcode.

Anhand des ISO 9126 Qualitätsmodells [ISO01] wird jede System-Charakteristik einer entsprechenden Quellcode-Eigenschaft zugeordnet. Die Wartbarkeit einer Software lässt sich durch 4 Kategorien beschreiben: die Analysierbarkeit, die Modifizierbarkeit, die Stabilität und die Testbarkeit. Entsprechend werden einige Quellcode-Eigenschaften identifiziert, die die Charakteristiken der Wartbarkeit beeinflussen können. Das Volumen eines Moduls trägt bei der Analysierbarkeit des Moduls bei, da ein größes Modul oft schwieriger zu verstehen ist. Für diese Eigenschaft wird LOC als die Metrik gewählt. Die durchschnittliche Komplexität per Module gibt an, wie kompliziert ein Modul zu modifizieren und zu testen ist. Die zyklomatische Komplexität-Metrik dient dem Messen dieser Eigenschaft. Codeklone sollte die Analysierbarkeit und Modifizierbarkeit beeinträchtigen da intuitiv machen Codeklone das System größer als es sein soll und unter dem geklonten Code kann eine Änderung zu Änderungen an vielen Stellen führen, wo sich der Code repliziert. Eine einfache und trotzdem, von den Autoren bestätigt, ausreichende Strategie zum Entdecken von Codeklonen wird benutzt. Die Größe der Code-Einheit (Code Unit) ist auch ein wichtiger Faktor für die Analysierbarkeit und Testbarkeit. Eine Code-Einheit ist das kleinste Codefragment, das individuell ausführbar und testbar ist.

In Programmiersprachen wie Java oder C++ ist sie eine Methode. Die Größe einer Code-Einheit wird durch ihre LOC dargestellt. Schließlich werden Unit Tests betrachtet. Unit Tests sind kleine Programme, die entwickelt werden um Code-Einheit automatisch testen zu können. Demzufolge ist die Testbarkeit direkt mit Unit Tests verbunden. Unter Verwendung von diesem Metriken-Modell auf 3 moderne, objektorientierte Software-Systeme können die Autoren signifikante Unterschiede in der Wartbarkeit zwischen den Systemen feststellen.

3.5 Zusammenfassung

In diesem Kapitel wurde auf verschiedene Ansätze eingegangen, die das Konzept der Wartbarkeit zu quantifizieren versuchen. Die bekanntesten und auch in der Praxis meist eingesetzten Metriken werden für sowohl das prozedurale Programmierparadigma als auch für OOP eingeführt. Es wurde festgestellt, dass die Wartbarkeit einer Software tatsächlich durch Metriken erfasst werden kann. Dafür ist es erforderlich, ein Metriken-Modell (oder eine Metriken-Suite) zu definieren, weil die Wartbarkeit ein kompliziertes Konzept ist und eine einzige Metrik nicht zur genauen Beschreibung dieses Konzepts dienen kann. Die in dieser Arbeit benutzten Software-Metriken müssen für die Implementierung klar definiert werden können und den wichtigsten Aspekten der Wartbarkeit in OOP entsprechen. Um die Software-Qualität verbessern zu können, muss man in der Lage sein die Ursachen der schlechten Ergebnisse anhand der Metriken deutlich analysieren zu können.

Im Abschnitt 3.2 und 3.3 wurde gezeigt, dass einige Metriken bzw. Metriken-Suiten wie der MI nicht die oben genannten Voraussetzungen erfüllen können, obwohl ihre Fähigkeit zur Ermittlung der Wartbarkeit von Software auch vielversprechend ist. Die CBO-, RFC- und DBC-Metrik werden in der Arbeit gewählt, weil sie die Essenz der OOP beschreiben können. Der Ermittlung der Komplexität, eines der wichtigen Aspekte der Wartbarkeit, dient die CC-Metrik. Wie im Abschnitt 3.2.2 gezeigt, wurde die Anwendbarkeit dieser Metrik durch zahlreiche Studien bewiesen und obwohl sie ursprünglich eine Metrik für die prozedurale Programmierung war, lässt sie sich leicht an das objektorientierte Paradigma anpassen. Die LOC-Metrik bildet das letzte Element des Metriken-Modells dieser Arbeit. Sie ist eine einfache Metrik trotzdem in vielen Fällen kann diese Metrik die Intuition von Entwicklern über die Analysierbarkeit und Komplexität einer Software darstellen.

Kapitel 4

Verwandte Arbeiten

In diesem Kapitel werden wissenschaftliche Veröffentlichungen in verschiedenen Forschungsbereichen vorgestellt, die sich direkt oder teilweise mit den Problemen der Optimierung von nicht-funktionalen Eigenschaften in SPLs beschäftigen. Das Kapitel ist wie folgt strukturiert: Zuerst werden unterschiedliche Ansätze in Betracht gezogen, die allgemeine qualitative Anforderungen von SPLs berücksichtigen. Anschließend werden Arbeiten betrachtet, die sich mit der Identifizierung von Schwachstellen im Quellcode widmen, woran Maßnahmen zur Verbesserung der Quellcode-Qualität eingesetzt werden können. Schließlich werden Arbeiten vorgestellt, die sich direkt mit der Umsetzung von Refactorings zur Unterstützung der nicht-funktionalen Eigenschaften befassen. Die Ergebnisse der präsentierten Arbeiten werden im Zusammenhang mit den Zielen dieser Masterarbeit betrachtet.

4.1 Optimierung qualitativer Eigenschaften von SPLs

Benavides et al. forderten in [BMAC05, BSTC07], dass sich die Produkte einer SPL nicht nur in funktionalen sondern auch in nicht-funktionalen Features unterschieden werden können. Solche nicht-funktionalen Features werden in ihren Begriffen als extra-funktionale oder qualitative Features gekennzeichnet und sollen in dem Feature Modell bzw. bei Generierung der Varianten der SPL berücksichtigt werden. Dafür erweiterten sie das bekannte Feature Modell von Czarnecki [EC00] wie folgt: Ein Feature kann jetzt mehrere Attribute haben, die die messbaren Eigenschaften des Features darstellen. Entsprechend jedem Attribut wird eine Attribut-Domäne definiert, die die gültigen Werte eines Attributs beschränkt. Neben den Relationen zwischen den Features (obligatorisch, optional oder alternative) werden Bedingungen von einem Attribut bzw. Relationen zwischen mehreren Attributen eines Features definiert. Diese Bedingungen bzw. Relationen bilden die extra-funktionalen Features der SPL. Anhand dieser Erweiterung wird ein Algorithmus entwickelt, der ein Feature Modell in ein Bedingungserfüllungsproblem (engl. Constraint Satisfaction Problem - CSP) transformiert. Dabei werden die Features als Variablen und die Relationen als Bedingungen des CSPs betrachtet. Die gültigen resultierenden Produkte der SPL sind die Lösungen des entsprechenden CPSs.

Weil CSP ein NP-vollständiges Problem ist und die Anzahl der Features einer SPL

bis zu hunderte oder tausende betragen könnten, könnte die Suche nach der exakten Lösung des CSPs in einer inakzeptablen Zeit dauern. Deshalb präsentiert White et al. in [WDS09] einen Ansatz, der mittels Filtered Cartesian Flattening eine gute Lösung in Linearzeit approximieren kann. Beide Ansätze von Benavides und White optimieren die Produkte der SPL durch Auswahl der entsprechenden optimalen Features oder der optimalen Implementierung der Features. Der Ansatz dieser Arbeit ermöglicht weitere Optimierung durch RFMs nachdem eine Variante der SPL generiert wurde.

Andere Ansätze basieren auf die Technik der modellgetriebenen Softwareentwicklung (engl. model-driven software development - MDSD) [Mat06, KPS08, RPB09], wobei anstatt Klassen Modelle als Kernkomponenten der Abstraktion angesehen werden. Ein Modell stellt ein konkretes Bild eines Systems dar und besitzt eine Architektur, die die Komponenten und deren Zusammenspiel innerhalb des Softwaresystems beschreibt. Der Schlüssel der modellgetriebene Entwicklung liegt in der Transformation der Modelle. Eine Transformation ist ein Prozess, der ein Modell in andere Modelle des gleichen Systems umwandelt. Für eine Transformation wird eine Menge von Regeln definiert, wobei jede einzelne Regel einen bestimmten Teil des Quell-Modells in das entsprechende Element des Ziel-Modells abbildet. Da ein Modell im Zusammenhang mit seiner Architektur liegt, führt eine Transformation des Modells auch zur Änderung der entsprechenden Architektur und folglich der qualitativen Attribute des Modells [RPB09]. Transformationen, die auf qualitative Anforderungen zurückzuführen sind, werden von Matinlassi als „quality-driven software architecture model transformation“ (dt. qualitätsgetriebene Transformation der Softwarearchitektur-Modelle) gekennzeichnet. Kim et al. entwickelten ein Framework [KPS08], welches Architekturen für unterschiedliche Aspekte der Software-Qualität zur Verfügung stellt. Durch Anwendung des entsprechenden Architektur-Stils können die qualitativen Attribute eines Softwaresystems verbessert bzw. degradiert werden z.B. das Model View Controller (MVC, dt. Modell-Präsentation-Steuerung) erhöht die Benutzbarkeit aber lässt die Performance oder die Wiederverwendbarkeit des Systems nach. Rossel et al. implementiert Modell-Transformationen für die inkrementelle Konstruktion der Softwarearchitektur entsprechend den gewählten funktionalen und qualitativen Anforderungen [RPB09]. Die vorgestellten Arbeiten fokussieren sich auf Optimierung der nicht-funktionalen Eigenschaften auf Modelle-Ebene während diese Arbeit sich der Transformationen auf Quellcode-Ebene widmet .

4.2 Identifizierung von Schwachstellen im Quellcode

Schwachstellen sind Stellen im Quellcode, wobei schlechtes Design oder schlechte Programmierung aufweist und der entsprechende Code überarbeitet werden sollte (z.B. durch Refactorings). Diese werden von Kent Beck als „Bad smells“ (dt. übelriechender Code) bezeichnet. In der Arbeit von Tourwe und Mens [TM03] wird versucht, „Bad smells“ im Quellcode mit Hilfe von Logic Meta Programming (LMP, dt. Logik Meta-Programmierung) zu identifizieren. LMP stützt sich auf eine enge Symbiose zwischen einer objektorientierten Basis-Sprache und einer deklarativen logischen Meta-Sprache. Unter Anwendung von LMP kann man nicht nur alle Entitäten des objektorientierten Programms (i.e. Variablen, Methoden, Klassen etc.) direkt zugreifen, sondern auch komplizierte Relationen durch logische Anfragen herausfinden. In ihrer Arbeit implementierten die Autoren SOUL, eine Variante der logischen Sprache Prolog als Meta-Sprache

für die objektorientierten Sprache Smalltalk. Als Beispiel bauen sie dann die logischen Regeln auf um zwei „Bad smells“: unbenutzte Parameter (eng. obsolete Parameters) und unpassende Interfaces (eng. inappropriate interfaces) zu identifizieren. Zur Untersuchung von Schwachstellen im Quellcode werden in dieser Masterarbeit die AST-Struktur und Software-Metriken benutzt. Während Software-Metriken den Anwendern Räume für die Interpretationen gibt, gibt die Technik von Tourwe und Mens strikter Aussagen an, d.h. entweder der Code ist ein „Bad smell“ oder nicht.

In [DW02] zeigten Dudziak und Wloka einen Ansatz zur Ermittlung von „Bad smells“ im objektorientierten Code mittels AST-Struktur und statischer Programmanalyse. Dabei folgen sie einer heuristischen Strategie: für jedes „Bad smell“ werden einige Metriken definiert, die durch Analyse des Programms festgestellt werden. Darauf werden Regeln definiert, die entscheiden, ob der Code refaktorisiert werden sollte oder nicht. Die Autoren konnten mit ihrem Tool 12 aus 22 von Fowler und zwei selbst definierten „Bad smells“ mit einer guten Genauigkeit identifizieren. Dieser Ansatz ist dem von dieser Arbeit ähnlich dennoch konzentriert sich diese Arbeit nicht auf alle „Bad smells“ sondern gezielt auf die, die die Wartbarkeit eines Programms vermindern. Dementsprechend werden in dieser Arbeit keine ad-hoc-Metriken sondern nur Metriken in der Literatur benutzt, ihre Anwendbarkeit zur Ermittlung der Wartbarkeit getestet und bestätigt wurde.

In [vM02] charakterisierten die Autoren „Bad smells“ anhand ihrer Aspekte. Sie klassifizierten diese Aspekte in zwei Kategorien: primitive Aspekte (engl. primitive smell aspects), die man direkt im Code erkennen kann, und abgeleitet Aspekte (engl. derived smell aspects), die aus primitiven Aspekten gezogen werden können. Mittels eines AST-ähnlichen Analyser-Tools werden zuerst die primitiven Aspekte der „Bad smells“ gesammelt und dann werden abgeleitete Aspekte berechnet. Als Prototyp entwickelten die Autoren ein Tool zur Ermittlung von „Bad smells“, die durch Missbrauch von *instanceof* und Typumwandlung (engl. type cast) identifiziert werden können. Die Ergebnisse der Analyse stellt das Tool graphisch dar indem es die Programmentitäten (i.e. Pakete, Klassen, Methoden etc) als Knoten und ihre Beziehungen (i.e. Vererbung, Aufrufe etc.) als Kanten eines Graphen modelliert. Entitäten mit „Bad smells“ können dann durch Farben oder durch Layout des Graphen hervorhoben werden. Eine konkrete Vorgehensweise gaben die Autoren aber nicht an, wie man abgeleitete Aspekte aus den primitiven folgern und diese Aspekte zu den „Bad smells“ zuordnen kann. In der vorliegenden Masterarbeit werden Eigenschaften von „Bad smells“ durch Software-Metriken quantifiziert. Durch Interpretierung der numerischen Ergebnisse können Refactoring-Möglichkeiten festgestellt werden.

Die Arbeit von Simon et al. benutzte auch Software-Metriken als Grundlagen für die Ermittlung von Refactoring-Möglichkeiten [SSL01]. Der Grundstein dafür ist die DBC-Metrik (siehe Abschnitt 3.3.1), deren Messung-Ergebnisse in einem 3D-Modell visualisiert werden. Basierend auf die Distanz-Relationen der Entitäten in der Visualisierung können Kandidaten für die Anwendung der 4 Refactorings *MOVE METHOD/ATTRIBUTE* und *EXTRACT/INLINE CLASS* herausgefunden werden. Um verschiedene Aspekte der Wartbarkeit einer Software erfassen zu können werden in dieser Masterarbeit mehrere Metriken eingesetzt. Die Identifizierung von Schwachstellen im Quellcode erfolgt durch Interpretierung der numerischen Ergebnisse dieser Metriken.

4.3 Umsetzung von Refactorings zur Unterstützung von NFPs in SPLs

In [SKPA10] präsentierten Siegmund et al. einen Ansatz zur Optimierung der NFPs von SPLs mittels RFMs, worauf diese Arbeit auch basiert. Die Autoren haben als Prototyp ein Tool entwickelt, welches den Prozess zur Umsetzung des INLINE METHOD-Refactorings automatisiert. Dieses Refactoring sollte die Performance des Programms verbessern können. Nach der Anwendung dieses Tools stellten sich die Ergebnisse heraus, dass die Performance des Programms gestiegen sind aber zugleich die Größe des Programms im Arbeitsspeicher auch gewachsen ist. Diese Masterarbeit konzentriert sich auf andere Aspekte der NFPs von SPLs, nämlich die, die die Wartbarkeit der Produkte einer SPL bestimmen. Dazu werden sowohl unterschiedliche Software-Metriken implementiert als auch verschiedene Refactorings und deren entsprechende Umsetzungsprozesse automatisiert.

Wie oben erwähnt, schlugen Simon et al. vor, dass 4 Refactorings *MOVE METHOD/ATTRIBUTE* und *EXTRACT/INLINE CLASS* die Kohäsion-Eigenschaft eines Programms verbessern können [SSL01]. Allerdings führten sie in ihrer Arbeit keine wirkliche Test-Szenarien zur Validierung dieser Hypothese durch. Du Bois et al. zeigten in [DBDV04], dass Refactorings zwei qualitative Attribute, Kohäsion und Kopplung unterstützen können. Dazu haben sie 5 Refactorings EXTRACT METHOD, EXTRACT CLASS, MOVE METHOD, REPLACE METHOD WITH METHOD OBJECT und REPLACE DATA VALUE WITH OBJECT aus der Refactoring-Liste von Fowler [Fow99] ausgewählt, die vermutlich unterschiedliche Einflüsse auf die Eigenschaften haben. Für jedes Refactoring wird eine heuristische Strategie vorgestellt, wobei die Anwendung des entsprechenden Refactorings zu positiven Änderungen der Software-Qualität führen kann. Beispielsweise kann man mit dem EXTRACT CLASS-Refactoring die Kohäsion verbessern und die Kopplung verringern indem man eine Gruppe von Methoden und Feldern einer Klasse in eine neue Klasse extrahieren, die zusammenhängend sind und wenige Beziehungen zu anderen Entitäten der Quellklasse haben. Zur Validierung ihrer Hypothesen wenden sie die genannten Refactorings auf das Softwaresystem von Apache Tomcat an. Dabei entwickelten sie ein Tool zur Ermittlung von Refactoring-Möglichkeiten, welches die Benutzung-Relationen zwischen Methoden und Feldern der Klassen analysiert und die entsprechenden Metriken für die Kohäsion und Kopplung berechnet. Die Ergebnisse stellten sich heraus, dass die Anwendung der MOVE METHOD, REPLACE METHOD WITH METHOD OBJECT, REPLACE DATA VALUE WITH OBJECT und EXTRACT CLASS Refactorings auf die gefundenen Kandidaten tatsächlich die Kohäsion und Kopplung des Programms verbessert hat. Lediglich konnte das EXTRACT METHOD-Refactoring keine positive Änderungen erbringen. Diese Masterarbeit zählt auch die Kohäsion und Kopplung zu den bedeutenden Kennzeichen der Wartbarkeit eines Programms. Zusätzlich dazu werden weitere Metriken auch berücksichtigt, die die anderen Aspekte der Wartbarkeit beschreiben können (z.B. Komplexität oder Codegröße etc.). Außerdem gaben die Autoren in ihrer Arbeit nicht an, ob sie die Refactorings automatisiert oder manuell angewendet haben. Die Automatisierung der Erstellung und Durchführung der Refactorings durch RFMs ist einer der wichtigsten Teile dieser Masterarbeit.

In [RBJ97] wurde ein Refactoring Tool für die objektorientierte Sprache Smalltalk

entwickelt, welches verschiedene Refactorings in Bezug auf Variablen, Methoden und Klassen automatisiert hat. Eine Unterstützung zur Ermittlung von Refactoring-Möglichkeiten bietet dieses Tool aber nicht. Das Tool von Dudziak und Wloka [DW02] befasste sich mit diesem Problem und automatisierte sowohl den Prozess zur Entdeckung von „Bad smells“ als auch die Auswahl und Durchführung der entsprechenden Refactorings. Zur Vereinfachung der Benutzung wurde das Tool in die Netbeans-IDE integriert. Dieser Ansatz ist dem von dieser Masterarbeit ähnlich, dennoch widmet sich diese Arbeit einer spezifischen Aufgabe, qualitative Eigenschaften eines Programms in Bezug auf die Wartbarkeit zu unterstützen. Darüber hinaus realisiert diese Arbeit den Refactoring-Prozess durch RFMs, die in die Feature Module einer SPL leicht integriert werden können.

Kapitel 5

Realisierung der Wartungsmetrik unterstützenden Refactorings

In dem vorherigen Kapitel 3 wurden Software-Metriken eingeführt, die die Wartbarkeit einer Software vorhersagen können und daraus wurden 5 Metriken zur Realisierung in der Arbeit gewählt. Sie sind: LOC, zyklomatische Komplexität, RFC, CBO und Distanzbasierte Kohäsion. In diesem Kapitel wird die Anwendung von Refactorings, die diese Wartungsmetriken unterstützen, mittels einer Beispiel-Anwendung beschrieben. Die Refactorings werden aus dem Buch von Fowler [Fow99] genommen und in den entsprechenden Abschnitten genauer dargestellt. Das Anwendungsgebiet der Metriken und der Refactorings wird auf die Programmiersprache Java eingeschränkt. Das Kapitel fängt mit der Beschreibung der Beispiel-Anwendung an. Als Nächstes wird die Realisierung der Software-Metriken betrachtet. Anschließend werden die Anwendungen der unterschiedlichen Refactorings auf die Beispiel-Anwendung präsentiert und besprochen. Der letzte Abschnitt fasst die Ergebnisse zusammen und zieht die Schlussfolgerungen für die Automatisierung der Anwendung der Refactorings.

5.1 Beispiel-Anwendung: die Graph-Produktlinie

Die Graph-Produktlinie ist eine Beispiel-Produktlinie aus der AHEAD Tool Suite [1], sie implementiert unterschiedliche Graphenalgorithmen. Dieses Beispiel wurde genommen weil die Graphentheorie eine klassische Domäne ist, deren Algorithmen vielen Informatikern allgemein bekannt sind. Generell besteht ein Graph aus einer (endlichen) Menge von Knoten, die eventuell durch Kanten miteinander verbunden sind. Eine Kante kann gerichtet sein und ein Gewicht haben. Dies und weitere grundlegende Merkmale der Graphen werden durch folgende Features dargestellt:

- gerichteter Graph: anstatt Kanten besteht ein gerichteter Graph aus Pfeilen, wobei ein Pfeil von einem Quellknoten zu einem Zielknoten zeigt und verdeutlicht, dass sich die Kante nur in dieser Richtung durchlaufen lässt.
- ungerichteter Graph: durch die verbindende Kante kann man in beiden Richtungen zwischen den zwei Knoten gehen.

[1] http://www.cs.utexas.edu/users/dsb/GPL/graph.htm

- gewichteter Graph: die Kanten eines Graphen können mit Werten versehen werden, die spezifizieren, wie teuer der Durchlauf durch eine Kante sein kann. Zum Beispiel verbinden zwei Kanten die Knoten A und C und die Knoten B und C mit den entsprechenden Gewichten 1 und 2. D.h ein Durchlauf von B bis C ist teurer als der Durchlauf von A bis C. Bei der Implementierung von gewichteten Kanten in diesem Beispiel wird davon ausgegangen, dass alle Werte nichtnegative ganze Zähle sind.

- ungewichteter Graph: ein Durchlauf von A bis B durch die entsprechende Kante kostet nichts.

- Tiefensuche (engl. Depth First Search - DFS): bezeichnet ein Verfahren zum Durchsuchen bzw. Durchlaufen der Knoten eines Graphen ausgehend von einem Startknoten.

- Breitensuche (engl. Breadth First Search - BFS): ist auch ein Verfahren zum Durchsuchen bzw. Durchlaufen der Knoten eines Graphen von einem Startknoten.

- Nummerierung der Knoten (engl. Vertex Numbering): die Nummerierung der Knoten stellt die Reihenfolge der durch einen Durchlauf-Algorithmus z.B. BFS oder DFS etc. besuchten Knoten dar.

- zusammenhängende Komponenten (engl. Connected components): Dieser Begriff bezieht sich auf die ungerichteten Graphen. Ein ungerichteter Graph heißt zusammenhängend wenn es zu je zwei beliebigen Knoten v und w aus der Menge der Knoten einen Pfad gibt, wobei v der Startknoten und w der Zielknoten ist. Ist ein Teilgraph des Graphen, dessen Knotenmenge eine Teilmenge der Knotenmenge des originellen Graphen ist, ein zusammenhängender Graph, wird er als eine Zusammenhangskomponente des Graphen bezeichnet.

- stark zusammenhängende Komponenten (engl. Strongly connected components): Dieser Begriff ist dem Begriff von zusammenhängenden Komponenten ähnlich, bezieht sich aber auf die gerichteten Graphen.

- Zyklen in Graphen finden (engl. Cycle Checking): Ein Zyklus existiert in einem (gerichteten oder ungerichteten) Graphen wenn es einen Pfad gibt wobei er aus verschiedenen Kanten besteht und der Startknoten und der Zielknote der gleiche Knoten ist.

- Minimaler Spannbaum (engl. Minimum Spanning Tree): Ein Spannbaum ist ein Teilgraph eines ungerichteten Graphen, der ein Baum ist und alle Knoten des Graphen enthält. Der Begriff des minimalen Spannbaum bezieht sich auf die gewichteten Graphen. Ein Spannbaum heißt minimal wenn es keine andere Spannbäume gibt, die ein geringeres Gewicht als das von dem Spannbaum haben.

- Einzelquelle - kürzester Pfad (engl. Single-Source Shortest Path): Der Algorithmus findet einen Pfad mit minimaler Länge von einem Startknoten zu anderen übrigen Knoten des Graphen. Dabei werden die Gewichte der Kanten berücksichtigt und der kürzeste Pfad muss nicht unbedingt der Pfad mit minimaler Anzahl von Knoten sein sondern der Pfad mit dem minimalen Gewicht.

Die von der Graph-Produktlinie abgeleiteten Produkte können mittels der Kombination unterschiedlicher Features variiert werden z.B. der Graph kann gerichtet oder ungerichtet, mit Kantengewicht oder ohne Kantengewicht sein. Die Variation kann auch einen oder mehreren Graphenalgorithmen mit dem entsprechenden Suche-Algorithmus implementieren. In einer typischen SPL sind nicht alle Kombinationen der Features gültig z.B. ein Graph kann nicht die beiden zusammenhängende Komponenten- und start zusammenhängende Komponenten- Algorithmen implementieren da es verlangt, dass der Graph zugleich gerichtet und ungerichtet sein muss. Die Tabelle 5.1 fasst alle Beschränkungen für die Kombination der Features der Graph-Produktlinie zusammen:

Tabelle 5.1: Relationen zwischen den Features der Graph-Produktlinie

Algorithmus	**Suche-Algorithmus**	**Graphtypen**	**Kantentypen**
Nummerierung der Knoten	BFS, DFS	gerichtet, ungerichtet	gewichtet, ungewichtet
zusammenhängende Komponenten	BFS, DFS	ungerichtet	gewichtet, ungewichtet
stark zusammenhängende Komponenten	DFS	gerichtet	gewichtet, ungewichtet
Zyklen in Graphen finden	DFS	gerichtet, ungerichtet	gewichtet, ungewichtet
minimaler Spannbaum	Nichts	ungerichtet	gewichtet
Einzelquelle - kürzester Pfad	Nichts	gerichtet	gewichtet

Die Graph-Produktlinie wurde mittels der Feature-orientierten Programmiersprache Jak, einer Erweiterung der objektorientierten Programmiersprache Java, implementiert [Bat03]. Die Verwaltung der Relationen zwischen den Features bzw. die Generierung unterschiedlicher Varianten der Produktlinie wird durch die AHEAD Tool Suite erledigt. Die Abbildung 5.1 stellt das entsprechende Feature-Modell der Graph-Produktlinie dar.

Zusätzlich zu den Features von Graphen werden zwei Features Prog und Benchmark implementiert. Das Feature Prog stellt die ausführbare Instanz des Programms dar, die die Präsentation eines Graphen aus einer Datei liest und alle mögliche Algorithmen darauf anwendet. Das Feature Benchmark protokolliert die Ausführungszeit bei Anwendung der Algorithmen. Um die Messung der gewählten Metriken und den Effekt der unterstützenden Refactorings zu testen wird eine Variante der Graph-Produktlinie erstellt mit den folgenden Features: gerichtet, gewichtet, DFS, Einzelquelle-kürzester Pfad, Zyklen finden, stark zusammenhängende Komponenten, Nummerierung der Knoten, Benchmark und Prog. Dabei wird das jampack-Tool verwendet weil kein Debuggen erforderlich ist und der von jampack generierten Quellcode wegen weniger Vererbung einfacher ist. 11 Klassen werden generiert und in ein gemainsames Paket, namens *src-NumberStrongConnectCycleShortestPath*, zusammengepackt. Die Klassen *Vertex, Neighbor* und *Edge* modellieren die Knoten und die gerichteten und gewichteten Kanten eines Graphen.

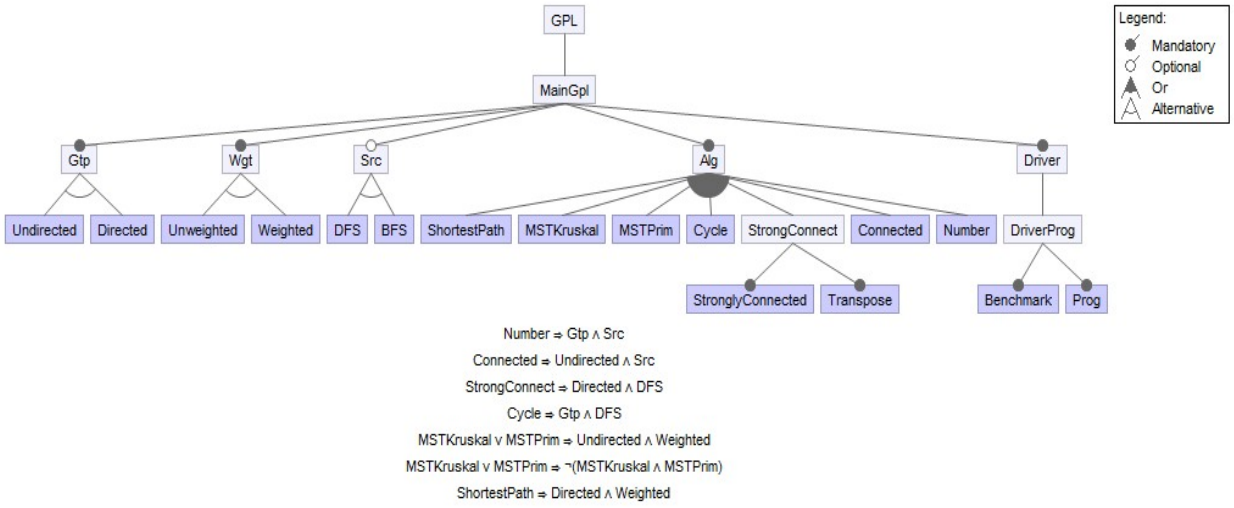

Abbildung 5.1: Feature-Modell der Graph-Produktlinie

Da viele Algorithmen auf dem Durchlauf des Graphen basierend dient die Klasse *WorkSpace* als eine Vorlage und deklariert alle benötigten Aktionen, die während des Durchlaufs durchgeführt werden sollen. Eine Klasse, die einen bestimmten auf dem Durchlauf basierenden Algorithmus realisiert, erweitert die WorkSpace-Klasse und implementiert die entsprechenden Aktionen z.B.: die Klasse *CycleWorkSpace* implementiert den Algorithmus zum Finden der Zyklen in dem Graphen, dazu muss er während des Durchlaufs die Knoten färben, diese Aktion wird dann in der Klasse *CycleWorkSpace* realisiert. Auf die gleiche Weise wird der Algorithmus zur Nummerierung der Knoten in der Klasse *NumberWorkSpace* und der Algorithmus zum Berechnen der stark zusammenhängenden Komponenten entsprechend in der Klasse *FinishTimeWorkSpace* und *WorkSpaceTranspose* implementiert (vgl. Abbildung 5.2).

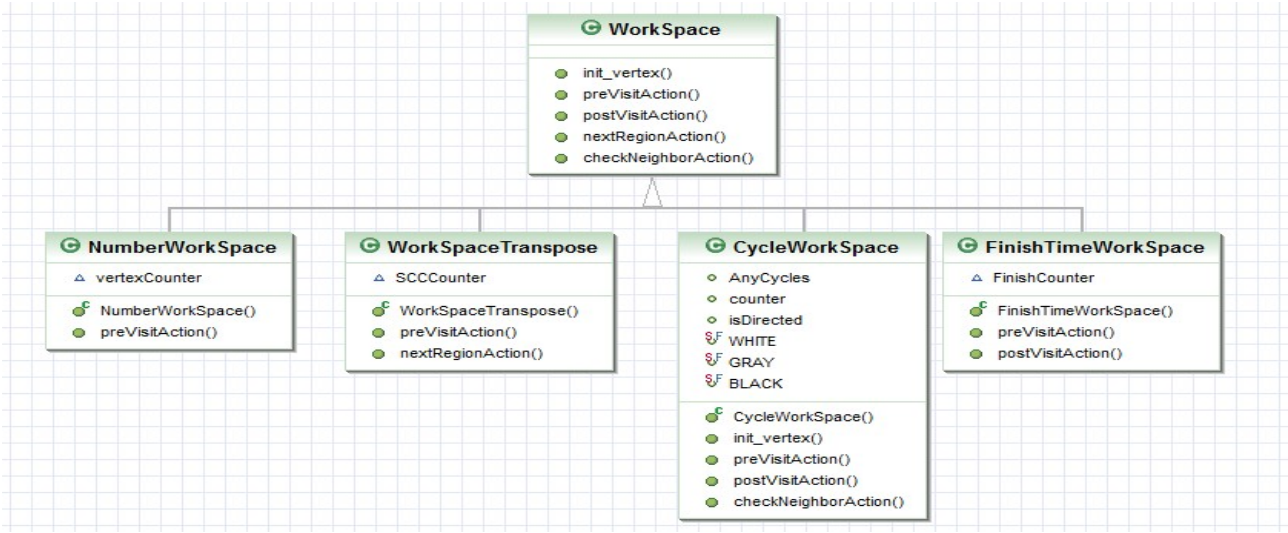

Abbildung 5.2: Die Vererbungshierarchie zwischen den Klassen *WorkSpace, NumberWorkSpace, CycleWorkSpace, FinishTimeWorkSpace* und *WorkSpaceTranspose*

Schließlich bildet die Graph-Klasse die Abstraktion für die Graphen. Zusammen mit anderen Klassen implementiert sie die gewählten Algorithmen und das Benchmark-Feature. Die Main-Klasse stellt das Feature Prog dar, sie realisiert eine ausführbare Instanz des Programms indem sie eine Eingabedatei liest, die einen Graphen darstellt, und die Algorithmen darauf anwendet.

5.2 Realisierung der Software-Metriken

Dieser Abschnitt beschreibt die Implementierung der im vorherigen Kapitel gewählten Software-Metriken, die die Wartbarkeit einer Software diagnostiziert. Anschließend werden die Ergebnisse der Anwendung dieser Metriken auf die Beispiel-Anwendung präsentiert. Dabei wird genauer eingegangen, wie man anhand der Messungsergebnisse potenzielle Stellen im Quellcode identifizieren, wo man die Refactorings anwenden kann.

5.2.1 Implementierung der LOC-Metrik

Die Implementierung der LOC-Metrik hält sich an die Definition von Hewlett-Packard ein, die am breitesten akzeptiert ist [FP97]: eine Codezeile ist irgendwelche Anweisung im Programm, die keine Leerzeile und kein Kommentar ist. Das LOC-Messung-Programm nimmt als Eingabe den Pfad des Quellcode-Ordners des Programms. Für jede Klasse wird die LOC-Metrik für alle lokalen Methoden und die Klasse selbst berechnet. Eine lange Methode bzw. eine lange Klasse wird oft als Kandidat für die Refactorings betrachtet. Wie man aber eine Methode oder eine Klasse als lang definieren kann hängt von dem Kontext der Methode bzw. der Klasse und von der persönlichen Einstellung jedes Software-Entwicklers ab. In [CL07] argumentieren die Autoren, dass die Länge einer Methode zwischen 4 und 40 und die Länge einer Klasse zwischen 4 und 400 liegen sollte. Andere Autoren schlagen vor, dass die ganze Methode in dem Fenster der Entwicklungsumgebung (IDE) passen sollte [2]. Das folgende Listing 5.1 stellt die Ergebnisse der LOC-Messung für die Beispiel-Anwendung dar:

Listing 5.1: LOC-Metrik der Beispiel-Anwendung

```
1  Max. LOC Value: 301 from Graph
2  Average LOC Value: 54.09090909090909
3  Number of classes: 11
4
5  Graph: 301
6  - Graph.ShortestPath(): 105
7  Vertex: 85
8  Main: 64
9  CycleWorkSpace: 40
10 Edge: 31
11 WorkSpaceTranspose: 16
12 WorkSpace: 15
13 Neighbor: 14
14 FinishTimeWorkSpace: 13
15 NumberWorkSpace: 10
16 GlobalVarsWrapper: 6
```

Die Graph-Klasse ist die längste Klasse des Programms mit 301 Codezeilen und die Durchschnittslänge liegt bei 54.09 Codezeilen. Diese Werte liegen noch im empfohlenen

[2] http://codebetter.com/jeremymiller/2005/04/26/long-methods-and-classes-are-evil/

Bereich aber wenn man die Länge der Methoden betrachtet, existiert in der Graph-Klasse eine Methode *ShortestPath* und in der Main-Klasse eine Methode *main* mit der Länge von 105 Codezeilen bzw. 59 Codezeilen. Diese Methoden sind demnach potenzielle Kandidaten für Refactorings.

5.2.2 Implementierung der CBO-Metrik

Die CBO-Metrik einer Klasse wird als die Anzahl der mit dieser Klasse gekoppelten Klassen betrachtet. Es sei an dieser Stelle wieder darauf hingewiesen, dass eine Klasse X mit einer Klasse Y gekoppelt heißt, wenn die lokalen Methoden von X die Felder oder die Methoden von Y benutzen. Die Definition der Metrik betrachtet nicht die Anzahl der Zugriffe von X auf Y sondern nur ob X auf Y zugreift. Je größer der Wert der Metrik ist, desto dichter ist die Klasse mit anderen Klassen gekoppelt und demnach schwieriger ist die Klasse zu warten. Chidamber und Kemerer, die Autoren der Metrik, setzten keinen empfohlenen Grenzwert für die Metrik und es sollte auch keinen geben, da die Werte von Programm zu Programm sehr variierend können. Die Klassen mit höchsten Werten werden daher als Kandidaten für Refactorings betrachtet. Die Berechnung der Werte basiert auf der Abstract Syntax Tree (AST)-Struktur der Klassen (dt. Abstrakter Syntaxbaum), die den Quellcode durch einen Syntaxbaum darstellt. In dieser Arbeit wird das Tool JastAdd[3] für die Analyse der AST-Struktur benutzt. Das CBO-Messung-Programm nimmt den Pfad des Quellcode-Ordners des Programms und anhand der AST-Struktur wird eine Programm-Datenbank erstellt, die verschiedene Anfragen über die Beziehungen der Programmentitäten (i.e. Variablen, Methoden, Klassen) ermöglicht. Man kann somit beispielsweise alle Methoden suchen, die von einer bestimmten Methoden zugegriffen werden. Für die CBO-Metrik wird zuerst für jede Methode herausgefunden, welche externen Klassen sie zugreift. Dabei werden die Zugriffe auf vererbten Felder oder Methoden auch berücksichtigt (Kopplung durch Vererbung). Es ist auch zu beachten, dass eine Klasse innere Klassen und eine Methode lokale und anonyme Klassen enthalten könnte. In diesen Fällen werden alle Zugriffe auf Felder oder Methoden solcher Klassen als lokal betrachtet. Die Union der Zugriff-Menge aller Methoden ergibt die Zugriff-Menge der entsprechenden Klasse, das ist auch der CBO-Wert der Klasse. Die Ergebnisse der Messung der CBO-Metrik für die Beispiel-Anwendung stellt das Listing 5.2 dar:

Listing 5.2: CBO-Metrik der Beispiel-Anwendung

```
1  Max. CBO Value: 8 from Graph
2  Average CBO Value: 2.0
3  Number of classes: 11
4
5  Graph: 8
6  Main: 2
7  FinishTimeWorkSpace: 2
8  WorkSpaceTranspose: 2
9  Vertex: 2
10 NumberWorkSpace: 2
11 Edge: 2
12 CycleWorkSpace: 2
13 WorkSpace: 0
14 Neighbor: 0
15 GlobalVarsWrapper: 0
```

[3] http://jastadd.org/web/

Die Graph-Klasse hat den höchsten CBO-Wert: Zur Implementierung ihrer Funktionalitäten benutzt diese Klasse die Felder und Methoden von 8 anderen Klassen (von insgesamt 11 Klassen). Die zweitrangige Klasse (Main) greift nur auf zwei andere Klassen zu, so ist es an dieser Stelle vernünftig zu entscheiden, dass die Graph-Klasse ein möglicher Kandidat für die Refactorings ist.

5.2.3 Implementierung der RFC-Metrik

Die RFC-Metrik einer Klasse ist die Summe der Anzahl der lokalen Methoden und der Anzahl der externen Methoden, die von den lokalen Methoden aufgerufen werden. Für eine externe Methode zählt nur ob sie von den lokalen Methoden aufgerufen wird. Von wie vielen Methoden oder wie oft sie aufgerufen wird, stehen nicht in der Betrachtung. Ein hoher Wert der Metrik impliziert, dass die Klasse schwierig zu warten ist dennoch sollte wie bei der CBO-Metrik die Interpretierung der Werte von Programm zu Programm variieren. Ausreißer werden oft als Kandidaten für die Refactorings angesehen. Dennoch sollte man das Verhältnis zwischen der Anzahl der lokalen Methoden und der Anzahl der externen Methoden auch in Betrachtung ziehen, da eine Klasse mit wenigen externen Methoden lose Kopplungen zu anderen Klassen darstellt, was auch den Prinzipien des objektorientierten Designs entspricht. Die Berechnung der RFC-Metrik basiert auch auf der AST-Struktur und der Programm-Datenbank. Beginnend wird für jede lokale Methode analysiert, wie viele unterschiedliche externe Methoden sie zugegriffen hat. Zugriffe auf lokal definierte Methoden werden nicht berücksichtigt. Dabei ist auch zu beachten, dass die Zugriffe auf Methoden der inneren Klassen oder der in einer Methode lokal definierten Klassen oder der anonymen Klassen nicht als extern klassifiziert werden. Die Union der Zugriff-Menge aller lokalen Methoden und der Menge der lokalen Methoden selbst konstituiert den RFC-Wert für die Klasse. Das Listing 5.3 präsentiert die Ergebnisse der Messung für die Beispiel-Anwendung:

Listing 5.3: RFC-Metrik der Beispiel-Anwendung

```
1  Max. RFC Value: 46 from Graph
2  Average RFC Value: 10.454545454545455
3  Number of classes: 11
4
5  Graph: 46 (local methods: 29, externe methods: 17)
6  Main: 17 (local methods: 2, externe methods: 15)
7  Vertex: 17 (local methods: 13, externe methods: 4)
8  FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1)
9  WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1)
10 NumberWorkSpace: 3 (local methods: 2, externe methods: 1)
11 Edge: 9 (local methods: 8, externe methods: 1)
12 CycleWorkSpace: 6 (local methods: 5, externe methods: 1)
13 WorkSpace: 6 (local methods: 6, externe methods: 0)
14 Neighbor: 2 (local methods: 2, externe methods: 0)
15 GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)
```

Mögliche Kandidaten zu refactorisieren sind hier die Graph-Klasse und die Main-Klasse mit dem entsprechenden CBO-Wert 46 bzw. 17. Man kann beobachten, dass die Vertex-Klasse auch einen CBO-Wert von 17 wie die Main-Klasse hat, dennoch besitzt sie nur 4 externen Methoden während die Main-Klasse 15 externen Methoden aufgerufen hat. Demzufolge ist es sinnvoller, die Main-Klasse als Kandidat zu nennen und die Vertex-Klasse nicht.

5.2.4 Implementierung der CC-Metrik

Ursprünglich wurde die CC-Metrik auf Methoden-Ebene definiert, sie gibt die Anzahl der logischen Pfade einer Methode an. Zur Berechnung der Metrik zeigt McCabe in [McC76], dass man einfach die Anzahl der konditionellen Zweige des Flussdiagramms plus eins zählen muss. Für unterschiedliche Programmiersprachen gibt es entsprechend unterschiedliche konditionelle Anweisungen, deshalb ist es auch erforderlich eine konkrete Zählungsstrategie für die Zielsprache zu definieren. In [CL07] wird eine Strategie für die C++ Sprache definiert, wobei folgende Sprachkonstruktionen als Beiträger zu dem CC-Wert identifiziert werden: *if, for, while, case, catch,* &&, ||, ?, *#if*, *#ifdef*, *#ifndef*, *#elif*. Basierend darauf wird in dieser Arbeit auch eine Zählungsstrategie für die Java Sprache spezifiziert:

- für jede Methode wird jedes Auftreten folgender Konstruktionen gezählt: *if, else, else if, for, while, do .. while, case, default, catch,* &&, || und die Kombination : und ?.

- da eine Methode lokal definierte oder anonyme Klassen beinhalten könnte wird die Kontribution aller Methoden solcher Klassen auch mitberücksichtigt. Sie wird zu dem gesamten CC-Wert der Wurzel-Methode akkumuliert.

- auf der Klassen-Ebene wird der CC-Wert der Klasse als der maximale Wert aller ihren Methoden festgelegt.

Empfohlen wurde von McCabe ein CC-Wert bis 10 für jede Methode. Sollte eine Methode den CC-Wert über 10 haben, sollte man Maßnahmen zur Vereinfachung der Methode unternehmen. In einer späteren Arbeit schlug Grady einen neuen Grenzwert vor, welcher bis 14 beträgt [Gra94]. Diese Arbeit übernimmt den Grenzwert von Grady zum Herausfinden der potenziellen Refactoring-Kandidaten. Die CC-Werte der Klassen in der Beispiel-Anwendung werden durch das folgende Listing 5.4 zusammengefasst:

Listing 5.4: CC-Metrik der Beispiel-Anwendung

```
1  Max. Cyclomatic Complexity Value: 19 from Graph
2  Average Cyclomatic Complexity Value: 4.545454545454546
3  Number of classes: 11
4
5  Graph: 19
6  - Graph.ShortestPath(): 19
7  Main: 10
8  CycleWorkSpace: 8
9  Vertex: 3
10 FinishTimeWorkSpace: 2
11 WorkSpaceTranspose: 2
12 NumberWorkSpace: 2
13 WorkSpace: 1
14 Neighbor: 1
15 Edge: 1
16 GlobalVarsWrapper: 1
```

Lediglich hat die Graph-Klasse einen CC-Wert über 14, welcher von der ShortestPath-Methode beigetragen wird. Da keine andere Methode den Grenzwert übertritt ist die ShortestPath-Methode der einzige Kandidat zu refactorisieren.

5.2.5 Implementierung der DBC-Metrik

Die DBC-Metrik wird von Simon et. al. [SSL01] vorgeschlagen und basiert auf der Benutzen-Beziehung von den Feldern bzw. den Methoden einer Klasse. Für ein Feld bildet die Menge der Methoden, die dieses Feld zugreifen und das Feld selbst die Benutzen-Menge dieses Felds. Für eine Methode werden die Menge der Methoden und der Felder, die von dieser Methode zugegriffen werden, zusammen mit der betrachteten Methode als die Benutzen-Menge der Methode berechnet. Ein hoher Wert der Distanz bedeutet, dass die Entitäten wenig mit einander kohäsiv sind während ein niedriger Wert eine gute Kohäsion impliziert. Die Interpretierung der Werte kann aber von Programm zu Programm sehr unterschiedlich deshalb sollte man auch keinen konstanten Grenzwert festlegen. Ursprünglich wird nur die Distanz zwischen den Programmentitäten definiert, um die Refactoring-Kandidaten identifizieren zu können werden in der Arbeit folgende Definitionen weiter eingeführt:

- Distanz einer Entität zu einer Klassen:

$$dist(x, C) = \begin{cases} \frac{\sum_{i=1}^{n} C_i}{n}, & \text{wenn } n > 0 \\ 0, & sonst \end{cases} .$$

  wobei:

  - C_i: die Distanz von x zu der Entität i der Klasse C ist,
  - n: die Anzahl der Entitäten in der Klasse C.

  Zu beachten ist, wenn man die Distanz einer Entität zu ihrer enthaltenen Klasse berechnet, zieht man nur die (n-1) anderen Entitäten der Klasse in Betrachtung.

- Kohäsion einer Klasse:

$$co(C) = \begin{cases} \frac{\sum_{i=1}^{n} x_i}{n}, & \text{wenn } n > 0 \\ 0, & sonst \end{cases} .$$

  - x_i: die Distanz der Entität x_i zu der Klasse C ist, x_i ist eine Entität von C.
  - n: die Anzahl der Entitäten in der Klasse C.

Ähnlich wie bei der Distanz zwischen zwei Entitäten bedeutet ein niedriger Wert bei der Distanz einer Entität zu einer Klasse, dass sie stark mit den Entitäten dieser Klasse verbunden ist. Die Kohäsion einer Klasse beschreibt wie die Entitäten der Klassen miteinander zugehörig sind. Das bedeutet ein hoher Wert impliziert eine lose Verbindung und umgekehrt. Demnach können die Klassen mit höchsten Kohäsion-Werten als potenzielle Kandidaten für die Refactorings betrachtet werden. Anhand der AST-Struktur und der Programm-Datenbank können die Benutzen-Mengen der Entitäten schnell abgefragt werden. Da die inneren, die lokalen und die anonymen Klassen auch als Entitäten einer Klasse gelten, gehören alle Felder und Methoden solcher Klassen zu der Entitäten-Menge der Wurzel-Klasse. Darüber hinaus sollte auch ein Anwendungsbereich für die Berechnungen der Distanzen zwischen den Entitäten mitberücksichtigt werden da solche Berechnungen für alle Entitäten des ganzen Programms sehr aufwändig sein könnten.

Es würde sinnvoller wenn man die Berechnung auf Paket-Ebene einschränken kann denn die Interaktionen normalerweise in den Klassen des gleichen Pakets erfolgen. Wenn es nicht der Fall ist kann man wieder die Berechnung auf Programm-Ebene umschalten. Eine Zusammenfassung der Kohäsion-Werte der Klassen der Beispiel-Anwendung stellt das Listing 5.5 dar:

Listing 5.5: Kohäsion-Metrik der Beispiel-Anwendung

```
1  Worst Cohesion Value: 1.0 from WorkSpace
2  Average Cohesion Value: 0.8556367417922123
3  Number of classes: 11
4
5  WorkSpace: 1.0
6  Main: 1.0
7  GlobalVarsWrapper: 1.0
8  Graph: 0.96268
9  Vertex: 0.95217
10 Edge: 0.90074
11 CycleWorkSpace: 0.87595
12 FinishTimeWorkSpace: 0.70000
13 WorkSpaceTranspose: 0.69167
14 Neighbor: 0.68434
15 NumberWorkSpace: 0.64445
```

Zum großen Teil haben die Klassen schlechte Kohäsion-Werte (über dem Durchschnitt), darunter sind 3 Klassen (*WorkSpace, GlobalVarsWrapper* und *Main*) mit dem Wert 1, was bedeutet, dass die Entitäten dieser Klassen gar nicht miteinander zugehörig sind und im ersten Blick sollten diese Klassen refaktorisiert werden. Dennoch wenn man genauer betrachtet, sind die WorkSpace- und GlobalVarsWrapper-Klasse zwei Interfaces, die keine wirkliche Methoden implementieren. Die Main-Klasse realisiert die Anwendung der Algorithmen auf die Testdaten, sie implementiert nur eine einzige Methode (*main*), die die anderen Methoden von anderen Klassen aufruft. Nur die übrigen Klassen (*Graph, Vertex, Edge, CycleWorkSpace*) implementieren tatsächliche Funktionalitäten und daher als Kandidaten für die Refactorings gesehen werden sollten. Man kann an dieser Stelle die Nützlichkeit der Metrik fragen da die obigen Schlussfolgerungen nur durch Kenntnisse über das Programm und nicht direkt aus den Ergebnissen der Messung der Metrik gezogen werden können. Der Grund dafür ist, dass es verschiedene Kohäsion-Typen gibt [HM95], die diese einzelne Distanz-basierte Metrik nicht reflektieren kann.

5.3 Umsetzung der unterstützenden Refactorings

Im letzten Abschnitt wird die Implementierung der gewählten Software-Metriken dargestellt. Entsprechend diesen Metriken sollen in diesem Abschnitt die Refactorings vorgeschlagen werden, die die Metriken verbessern können. Dazu werden für jedes Refactoring die Bedingungen genauer eingegangen, die dafür sorgen, dass die globale semantische Integrität des Programms vor bzw. nach dem Refactoring erhalten bleibt. Alle vorgeschlagenen Refactorings werden aus dem Buch von Fowler [Fow99] genommen. Dabei ist zu beachten dass die ursprüngliche Motivation eines Refactorings in [Fow99] nicht unbedingt zur Unterstützung der gewählten Metriken gedacht wurde.

5.3.1 Implementierung des EXTRACT METHOD-Refactorings

Wenn ein Codefragment in einer Methode aus zusammenhängenden Anweisungen besteht kann man dieses Fragment in eine neue Methode extrahieren. Der Zweck dieses Refactorings ist die Vereinfachung einer langen oder komplizierten Methode. Fowler ist der Meinung, dass kommentiertes Codefragment in eine Methode extrahiert werden sollte, deren Name die Kommentare ersetzt und die Logik des Codefragments erläutert. Demnach muss der Name der neuen Methode sorgfältig ausgewählt werden. Im Folgenden wird die Vorgehensweise bei Durchführung eines EXTRACT METHOD-Refactorings beschrieben:

- Erstellung einer neuen Methode mit dem Namen, der den Zweck des Codefragments darstellt.
- Kopieren des zu extrahierenden Codefragments in die neu erstellte Methode.
- Herausfinden der Variablen in dem Codefragment, die in der Quellmethode lokal definiert sind. Sie werden in der Zielmethode entweder als lokale Variablen oder als Parameter der Methode genommen werden.
- Wenn eine lokale Variable nur in dem Codefragment zugegriffen wird kann diese Variable auch als eine lokale Variable in der neuen Methode deklariert werden.
- Herausfinden, ob irgendeine lokale Variable der Quellmethode in dem extrahierten Codefragment geändert wird. Wenn es eine solche Variable gibt, sollte man überlegen, ob die neue Methode als eine Anfrage betrachtet werden kann (d.h. mit einer gleichen Eingabe produziert die Methode immer die gleichen Resultate). Wenn es der Fall ist, ordnet man der Variable den entsprechenden Wert zu. Wenn nicht oder wenn es mehrere solche Variablen gibt, muss das Refactoring abgebrochen werden.
- Übergeben der von dem Codefragment gelesenen lokalen Variablen als Parameter für die Zielmethode.
- Kompilieren und Testen für Kompilierungsfehler.
- Ersetzen des Codefragments in der Quellmethode durch einen Aufruf an die neue Methode und Testen.

Diese Vorgehensweise von Fowler wurde für die manuelle Durchführung des Refactorings gedacht. Wenn man die Durchführung automatisieren will müssen mehrere Bedingungen in Betrachtung gezogen werden. Zuerst müssen die Anweisungen in dem zu extrahierenden Codefragment kontinuierlich sein, d.h. sie müssen in einem gleich Block und nacheinander folgend sein, wobei ein Block das ganze Code-Stück ist, was in einem geschweifte Klammern-Paar {..} steht. Weil der Wert einer Variable durch unterschiedliche Weise (z.B. durch direkte Zuordnung, durch einstellige Operationen oder durch Anwendung einer Methode auf die Variable etc.) geändert werden kann, ist das Herausfinden, ob irgendeine lokale Variable der Quellmethode in dem extrahierten Codefragment geändert wird, sehr schwierig und aufwändig. Stattdessen wird folgende Strategie adoptiert: die neue Methode gibt ein Objekt-Array zurück, welches alle Parameter und lokalen

Variablen enthält, die in dem Teil nach dem extrahierten Fragment der Quellmethode zugegriffen werden. Darüber hinaus kann das Codefragment Return-Statements beinhalten, deren entsprechenden Werte auch durch das Objekt-Array zurückgegeben werden müssen. Auf der Seite der Quellmethode wird es beachtet, dass den zurückgegebenen Werten die richtigen Variablen und Typen zugeordnet werden. Es sei darauf hingewiesen, dass wenn das Codefragment Return-Statements enthält kann durch unterschiedliche konditionelle Anweisungen der Aufruf an die neue Methode die Ausführung der Quellmethode beenden oder weiterleiten. Um die Logik in solchen Fällen zu gewährleisten muss eine globale Variable definiert werden, deren Wert zur Laufzeit durch die Ausführung der neuen Methode bestimmt wird und dementsprechend kann die Ausführung der Quellmethode gesteuert werden. Weiterhin müssen die lokalen Variablen, die in dem extrahierten Codefragment deklariert und später in dem Teil nach dem Codefragment in der Quellcode weiter benutzt werden, als neue lokalen Variablen in der Quellmethode deklariert werden, da durch Extrahieren solche Variablen nicht mehr zugreifbar sind. Das folgende Listing 5.6 stellt das Interface des EXTRACT METHOD-Refactoring dar. Dabei ist zu beachten, dass die getHostingBlockHierarchy-Methode die Hierarchie des enthaltenen Blocks in der Schachtelung der Blöcke beschreibt.

Listing 5.6: Interfaces des EXTRACT METHOD-Refactorings

```
1 public String getInterfaceOfRefactoringDefs() {
2     return "public␣interface␣ExtractMethodRefactoringExt␣{\n" +
3     "␣String␣getOldMethod();\n" +
4     "␣String␣getHostingBlockHierarchy();\n"+
5     "␣String␣getFirstStatement();\n"+
6     "␣String␣getLastStatement();\n"+
7     "␣String␣getNewMethodName();\n" +
8     "}";
9   }
```

Man kann argumentieren dass die Automatisierung dieses Refactorings möglicherweise zusätzliche Komplexität in das Programm einführen kann (durch die Rückgabe als Objekt-Array, die globale Variable zur Verwaltung der Ausführung der Return-Statements). Trotzdem kann das EXTRACT METHOD-Refactoring dadurch in mehreren Situation angewendet werden. Dabei werden durch Automatisierung auch Fehler oder Verletzungen der Integrität des Programms möglichst vermieden. Eine Entscheidung zwischen manueller und automatisierter Anwendung des Refactorings sollte durch Überlegung des konkreten Kontextes (Komplexität des extrahierten Codefragment, vorhandene Return-Statements) betroffen werden.

5.3.2 Implementierung des EXTRACT CLASS-Refactorings

Das EXTRACT CLASS-Refactoring wird auf eine Klasse angewendet, wenn sie die Arbeit macht, die von zwei Klassen zu erledigen wäre. Dafür muss man die Entscheidung treffen, welche Entitäten (Felder und Methoden) der Quellklasse in die neue Klasse verschoben werden sollten. Vernünftigerweise sollten Entitäten, die zusammenhängend und stark miteinander verbunden sind, als Kandidaten angesehen werden. Der Refactoring-Prozess wird wie folgt beschrieben:

- Festlegung der zu extrahierenden Entitäten der Quellklasse.

- Erstellung einer neuen Klasse (in dem gleichen Paket der Quellklasse), deren Name die Aufgabe der Klasse erläutert.

- Erstellung einer Verbindung von der Quellklasse zu der Zielklasse. Somit kann die Quellklasse die extrahierten Entitäten weiterhin zugreifen. Eventuell könnte auch eine Verbindung von der Zielklasse zu der Quellklasse notwendig sein da die extrahierten Entitäten auch verbliebene Entitäten zugreifen könnten.

- Verschieben der Felder in die neue Klasse.

- Kompilieren und Testen.

- Verschieben der Methoden in die neue Klasse. Ein Test sollte durchgeführt nach jeder Aktion.

- Festlegung der Weise der Veröffentlichung der neuen Klasse, d.h. wie ihre Entitäten der neuen Klasse von anderen Klassen zugegriffen werden können.

Basierend auf dieser Vorgehensweise kann das Refactoring folgendermaßen automatisiert werden. Als Erstes muss es sichergestellt werden, dass die zu extrahierenden Methoden nicht in Unterklassen überschrieben werden und die neue Klasse eindeutig in dem enthaltenen Paket ist. Anschließend wird ein Feld vom Typ neuer Klasse in der Quellklasse deklariert. Es ermöglicht den Zugriff auf die extrahierten Entitäten in der Quellklasse und an Stellen, wo vorher die extrahierten Entitäten durch Objekte vom Typ alter Klasse zugegriffen wurden. Solche Zugriffe erfolgen jetzt durch das Delegationsobjekt. Jeder extrahierten Methode wird ein Parameter vom Typ alter Klasse hinzugefügt um Zugriff auf Felder und Methoden der alten Klasse in neuer Klasse zu gewährleisten. Dazu müssen diese Felder und Methoden der Quellklasse als „public“ deklariert werden. Wenn eine extrahierte Methode in der Quellklasse direkt auf eine Entität zugreift (wie etwa x++;), die auch zu extrahieren ist, muss dieser Zugriff in der neuen Klasse erhalten bleiben. Erfolgt der Zugriff aber indirekt (wie etwa a.x++; wobei a ein Objekt vom Typ alter Klasse ist), muss er nun durch das Delegationsobjekt erfolgen (wie etwa a.b.x++; wobei b das Delegationsobjekt vom Typ neuer Klasse ist). Zum Vereinfachen der Implementierung werden alle extrahierten Felder und Methoden in der neuen Klasse als „public“ deklariert. Das für eine Refactoring-Einheit bereitgestellte Interface des EXTRACT CLASS-Refactorings wird wie im Listing 5.7 definiert:

Listing 5.7: Interfaces des EXTRACT CLASS-Refactorings

```
1 public String getInterfaceOfRefactoringDefs() {
2     return "import␣java.util.Set;\n" +
3         "public␣interface␣ExtractClassRefactoring{\n" +
4         "␣Set<String>␣getMembersToExtract();\n"+
5         "␣String␣getClassToExtractTo();\n"+
6         "␣String␣getNewSourceClassField();\n"+
7         "␣String␣getTargetClassField();\n"+
8         "}";
9 }
```

5.3.3 Implementierung des DECOMPOSE CONDITIONAL-Refactorings

Das DECOMPOSE CONDITIONAL-Refactoring dient dazu, komplizierte konditionelle Anweisungen einer If-Anweisung zu vereinfachen. Dafür wird eine komplexe Kondition, die aus verschiedenen Logik-Operationen besteht, durch eine Methode ersetzt. Der Name der Methode sollte dann der Erläuterung des Zwecks der Kondition dienen. Das DECOMPOSE CONDITIONAL-Refactoring ist dem EXTRACT METHOD-Refactoring ähnlich in dem Sinne, dass es auch einen bestimmten Teil einer Methode in eine neue Methode extrahiert, genauer gesagt, die Kondition einer If-Anweisung. Refactoring-Einheiten, die dieses Refactoring anwenden möchten, sollten folgendes Interface (vgl. Listing 5.8) implementieren:

Listing 5.8: Interfaces des DECOMPOSE CONDITIONAL-Refactorings

```
1 public String getInterfaceOfRefactoringDefs() {
2     return "public␣interface␣DecomposeConditionalRefactoring␣{\n" +
3     "␣String␣getHostingMethod();\n" +
4     "␣String␣getHostingBlockHierarchy();\n"+
5     "␣String␣getIfStatement();\n"+
6     "␣String␣getNewMethodName();\n" +
7     "}";
8 }
```

Die Implementierung des Refactorings kann folgendermaßen erfolgen. Zuerst muss die enthaltene Methode und das enthaltene Block der If-Anweisung eindeutig identifizieren kann. Die If-Anweisung wird durch ihre Position in dem Block lokalisiert. Anschließend wird die Kondition analysiert und werden alle in dieser Kondition zugegriffenen Variablen als Parameter für die neue Methode übergeben. Dann wird die neue Methode in der Klasse mit der entsprechenden Parameter-Liste und einem booleschen Rückgabetyp deklariert. Sie enthält eine einzige Return-Anweisung, die den Wert der Kondition zurück gibt. Schließlich wird die Kondition in der If-Anweisung durch einen Aufruf an die neue Methode ersetzt.

5.3.4 Implementierung des REPLACE METHOD WITH METHOD OBJECT-Refactoring

In der Beschreibung des EXTRACT METHOD-Refactorings kann man erkennen, dass wenn die Methode und das zu extrahierende Codefragment zu viele lokalen Variablen enthalten, hindert es die Anwendung des Refactorings oder kann es durch Durchführung des EXTRACT METHOD-Refactorings unbenötigte zusätzliche Komplexität in die Klasse einführen. Diese Situation motiviert die Benutzung eines REPLACE METHOD WITH METHOD OBJECT-Refactorings, welches die komplexe Methode in eine neue Klasse verschiebt. Der Unterschied zu einem EXTRACT CLASS-Refactoring ist, dass bei dem REPLACE METHOD WITH METHOD OBJECT-Refactoring werden alle lokalen Variablen der Methode als globale Variablen (Felder) in der neuen Klasse deklariert. Somit kann man später EXTRACT METHOD-Refactoring auf die Methode der neuen Klasse bequem anwenden da es keine lokalen Variablen mehr gibt, die berücksichtigt werden müssen. Der Refactoring-Prozess wird in folgenden Schritten beschrieben

- Erstellung der neuen Klasse, deren Name dem Namen der Methode entspricht.

- Deklarieren eines Felds für das Objekt der Quellklasse.
- Deklarieren eines Felds für jede Variable und jedes Parameters der extrahierten Methode in der Zielklasse.
- Erstellung eines Konstruktors für die neue Klasse, der das Quellobjekt und die Parameter der zu extrahierenden Methode als Parameter nimmt.
- Erstellung einer Methode in der Zielklasse, namens „compute“
- Kopieren des Methodenrumpfs der Originalmethode in die compute-Methode. Behandlung der Zugriffe auf andere Methoden und Felder der alten Klasse.
- Ersetzen des Methodenrumpfs der Originalmethode durch ein Statement, welcher ein neues Objekt der neuen Klasse erstellt und die compute-Methode aufruft.

Bei Automatisierung des Refactorings muss folgende Aspekte beachtet werden: Zuerst muss man prüfen, dass die neue Klasse mit dem entsprechenden Namen eindeutig in dem enthaltenen Paket ist. Für die lokalen Variablen in der Methode, müssen alle Deklarationen in der neuen compute-Methode entfernt werden. Dabei müssen auch die Initialisierungen dieser Variablen berücksichtigt werden. Darüber hinaus müssen Felder und Methoden der Quellklasse, die von der Originalmethode direkt zugegriffen werden, als „public“ deklariert werden. Das folgende Listing 5.9 stellt das Interface des Refactorings dar:

Listing 5.9: Interfaces des REPLACE METHOD WITH METHOD OBJECT-Refactorings

```
1 public String getInterfaceOfRefactoringDefs() {
2     return "public␣interface␣ReplaceMethodWithMethodObjectRefactoring␣{\n" +
3         "␣String␣getOldMethod();\n" +
4         "␣String␣getNewClassName();\n" +
5         "}";
6 }
```

5.3.5 Implementierung des MOVE METHODS AND FIELDS-Refactoring

Wenn man erkennt, dass ein Feld oder eine Methode mehr von den Methoden einer anderen Klasse als von lokalen Methoden der enthaltenen Klasse benutzt wird, sollte er überlegen, ob dieses Feld oder diese Methode tatsächlich zu der anderen Klasse gehört und dahin angesiedelt werden sollte. Das MOVE METHODS AND FIELDS-Refactoring in dieser Arbeit ist eine Kombination des MOVE METHOD- und MOVE FIELD-Refactoring, die in [Fow99] definiert wurden. Diese Kombination basiert auf der Beobachtung, dass Felder und Methoden oft zusammenhängend sind. Wird ein Feld oder eine Methode verschoben, kann das Refactoring eventuell leichter sein wenn die zugehörigen Felder und Methoden auch zusammengepackt werden. Das MOVE METHODS AND FIELDS-Refactoring ähnelt sich das von dem EXTRACT CLASS-Refactoring: in beiden Fällen werden Felder und Methoden der Quellklasse in eine andere Klasse extrahiert. Der Unterschied ist, dass beim MOVE METHODS AND FIELDS-Refactoring werden die Felder und Methoden in eine vorhandene Klasse verschoben. Dafür müssen folgende zusätzliche Sachen berücksichtigt werden

- Als Erstes muss man sicher stellen dass die Zielklasse tatsächlich existiert
- Es muss geprüft werden, ob Felder mit den gleichen Namen oder Methoden mit den gleichen Signaturen wie die zu extrahierenden Entitäten schon in der Zielklasse vorkommen. Wenn es der Fall ist, muss das Refactoring abgebrochen werden.
- wie beim EXTRACT CLASS-Refactoring wird ein Feld vom Typ der Zielklasse in der Quellklasse definiert, welches die Zugriffe auf die extrahierten Entitäten ermöglicht. Jeder verschobenen Methode wird ein Parameter vom Typ alter Klasse hinzugefügt um Zugriff auf Felder und Methoden der alten Klasse in der Zielklasse zu gewährleisten. Dazu müssen diese Felder und Methoden der Quellklasse als „public“ deklariert werden.
- Kommen in einer Originalmethode Zugriffe auf Entitäten der Zielklasse vor, die durch ein Delegationsobjekt der Zielklasse erfolgen, müssen diese Zugriffe in direkte Form umgewandelt.

Das Interface des Refactorings wird wie im Listing 5.10 beschrieben:

Listing 5.10: Interfaces des MOVE METHODS AND FIELDS-Refactorings

```
1 public String getInterfaceOfRefactoringDefs() {
2     return "import␣java.util.Set;\n" +
3         "public␣interface␣MoveMethodsAndFieldsRefactoring{\n" +
4         "␣Set<String>␣getMembersToMove();\n"+
5         "␣String␣getClassToMoveTo();\n"+
6         "␣String␣getNewSourceClassField();\n"+
7         "␣String␣getTargetClassField();\n"+
8         "}";
9 }
```

5.4 Anwendung der implementierten Refactorings auf die Beispiel-Anwendung

Dieser Abschnitt stellt die Ergebnisse der Anwendung auf die Beispiel-Anwendung von den Refactorings in Bezug auf ihre entsprechende Metriken dar. Abgesehen von der Durchführung der Refactorings, die schon in dem letzten Abschnitt automatisiert wurde, wird die Erstellung der RFMs manuell gemacht werden. Danach werden die Ergebnisse sowohl für die betroffenen Klassen als auch für das gesamte Programm zusammen betrachtet. Es sei darauf hingewiesen, dass ein Refactoring nicht unbedingt den Metriken-Wert des gesamten Programms verbessern muss. In vielen Fällen reicht die Unterstützung für einige Klassen, die außergewöhnliche Metriken-Werte (im Vergleich zu den Werten der anderen Klassen) haben. In der Praxis werden die meisten Probleme von solchen wenigen Klassen verursacht. Dementsprechend kann es vernünftiger, gezielt auf diese Klassen zu unterstützen als auf das ganze Programm konzentrieren zu müssen.

5.4.1 Refactorings für die RFC-Metrik

Als Kandidaten wurden die *Graph-* und *Main*-Klassen mit den entsprechenden RFC-Werten 46 bzw. 17 identifiziert. Zur Darstellung deren Einflüsse auf die Metrik, reicht

es aus, die Refactorings lediglich auf die *Graph*-Klasse zu verwenden. Im Prinzip sollte eine Methode in die Klasse verschoben werden, bei der sie mehr Methoden benutzt als die von der enthaltenen Klasse. Dieser Aufgabe dient das MOVE METHODS AND FIELDS-Refactoring. Im Folgenden werden die Effekte beim Verschieben einer Methode analysiert.

- Für die Quellklasse: die Anzahl der lokalen Methoden wird um 1 reduziert. Die Originalmethode ist nun eine externe Methode geworden, sie trägt daher zu der Zugriff-Menge der Quellklasse bei. Da die Zugriff-Menge der Originalmethode nicht mehr bei Berechnung der Zugriff-Menge der Quellklasse mitberücksichtigt wird, kann diese Zugriff-Menge eventuell reduziert werden. Es hängt aber davon ab, ob die Zugriff-Menge der Originalmethode gemeinsame Elemente mit den Zugriff-Mengen der anderen Methoden von der Quellklasse teilt. Am schlechtesten Fall, wenn die Zugriff-Menge der extrahierten Methode kein eigenes Element hat, bleibt die Zugriff-Menge von der Quellklasse gleich und auch ihrer RFC-Wert.

- Für die Zielklasse: die Anzahl der lokalen Methoden wird um 1 inkrementiert. Greift eine Methode der Zielklasse vor dem Refactoring auf die Originalmethode zu, wird dieser Zugriff nach dem Refactoring nicht mehr gezählt, da die Originalmethode nun lokal in der Zielklasse ist. Dadurch wird die Zugriff-Menge von der Zielklasse um 1 reduziert. Die kann aber trotzdem erhöht werden da die Zugriff-Menge der extrahierten Methode nun zur Berechnung der Zugriff-Menge von der Zielklasse auch betrachtet wird. Am besten Fall teilt die Zugriff-Menge der Methode alle Elemente mit anderen Zugriff-Mengen der anderen Methoden von er Zielklasse. So bleibt die Zugriff-Menge von der Zielklasse gleich und ihrer RFC-Wert.

- Für andere Klassen: deren Zugriff-Mengen und RFC-Werte sind gleich wie vor dem Refactoring.

Aus den obigen Beobachtungen kann man erwarten, dass beim Verschieben mehrerer Methoden der RFC-Wert der Quellklasse reduziert und der Wert der Zielklasse nicht erhöht wird. Besitzt die Union der Zugriff-Menge dieser Methoden viele eigenen Elemente und teilt sie zugleich viele Elemente mit der Zugriff-Menge der Zielklasse kann man dementsprechend den maximalen Effekt bekommen. Im Folgenden wird es versucht, den RFC-Wert der *Graph*-Klasse mittels eines MOVE METHODS AND FIELDS-Refactorings zu verbessern. Dazu wird ein RFM definiert, welches die *GraphSearch*- und *addEdge*-Methode der *Graph*-Klasse in die *Vertex*-Klasse verschiebt. Diese Methoden werden genommen weil sie auf 3 Methoden der *Vertex*-Klassen und keine lokalen Methoden der *Graph*-Klassen zugreifen.

Die Einflüsse des Refactorings werden in Tabelle 5.2 zusammengefasst. Der RFC-Wert der *Graph*-Klasse und der Durchschnittswert des Programms wurden verbessert: obwohl die *Graph*-Klasse immer noch den höchsten RFC-Wert hat, besitzt sie anstatt 17 jetzt nur 14 externen Methoden, weniger als die von der zweitrangigen Main-Klasse (15). Der RFC-Wert der Zielklasse *Vertex* wurde um 4 erhöht (von 17 auf 21) dabei wurden aber nur zwei zusätzliche externe Methoden eingeführt. Das Refactoring kann weiter verbessert werden, wenn man mehrere Methoden finden kann, die mit den extrahierten Methoden zusammenhängend sind.

Tabelle 5.2: Ergebnisse der RFC-Metrik vor und nach dem MOVE METHODS AND FIELDS-Refactoring

Originale Werte der RFC-Metrik	Die RFC-Metrik nach dem MOVE METHODS AND FIELDS-Refactoring
Max. RFC Value: 46 from Graph Average RFC Value: 10.45454 Number of classes: 11	Max. RFC Value: 41 from Graph Average RFC Value: 10.36363 Number of classes: 11
Graph: 46 (local methods: 29, externe methods: 17) Main: 17 (local methods: 2, externe methods: 15) Vertex: 17 (local methods: 13, externe methods: 4) FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1) WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1) NumberWorkSpace: 3 (local methods: 2, externe methods: 1) Edge: 9 (local methods: 8, externe methods: 1) CycleWorkSpace: 6 (local methods: 5, externe methods: 1) WorkSpace: 6 (local methods: 6, externe methods: 0) Neighbor: 2 (local methods: 2, externe methods: 0) GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)	Graph: 41 (local methods: 27, externe methods: 14) Main: 17 (local methods: 2, externe methods: 15) Vertex: 21 (local methods: 15, externe methods: 6) FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1) WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1) NumberWorkSpace: 3 (local methods: 2, externe methods: 1) Edge: 9 (local methods: 8, externe methods: 1) CycleWorkSpace: 6 (local methods: 5, externe methods: 1) WorkSpace: 6 (local methods: 6, externe methods: 0) Neighbor: 2 (local methods: 2, externe methods: 0) GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)

In einigen Situationen eignen sich die Methoden nicht für ein Verschieben in eine vorhandene Klasse. Vielmehr sollte für sie eine eigene Klasse erstellt werden, da sie eine bestimmte Abstraktion darstellen. Zu diesem Zweck kann man ein EXTRACT CLASS-Refactoring oder ein REPLACE METHOD WITH METHOD OBJECT-Refactoring verwenden. Die beiden Refactorings unterscheiden sich von dem MOVE METHODS AND FIELDS-Refactoring hauptsächlich darin, dass sie eine neue Klasse erstellen und die Methoden dahin verschieben. Basierend auf den beobachten Effekten des MOVE METHODS AND FIELDS-Refactorings werden die Einflüsse des EXTRACT CLASS- bzw. des REPLACE METHOD WITH METHOD OBJECT-Refactorings im Folgenden diskutiert.

- Das REPLACE METHOD WITH METHOD OBJECT-Refactoring extrahiert eine einzige Methode der Quellklasse in die neue Klasse. Um den RFC-Wert der Quellklasse verbessern zu können muss die Zugriff-Menge der zu extrahierenden Methode möglichst viele eigenen Elemente besitzen. Da die neue Klasse auch ihren eigenen RFC-Wert hat, kann eventuell der Durchschnittswert des Programms erhöht werden.

- Das EXTRACT CLASS-Refactoring hat eventuell den gleichen Effekt wie das MOVE METHODS AND FIELDS-Refactoring auf der Seite der Quellklasse. Die extrahierten Methoden müssen daher zusammenhängend, d.h sie greifen auf viele gleiche Methoden zu und möglichst wenig mit der Quellklasse verbunden sein. Der durchschnittliche RFC-Wert des Programms kann aber steigen weil die neue Klasse zusätzlich dazu beitragen wird.

Das folgende Beispiel veranschaulicht die Einflüsse der Anwendung des REPLACE METHOD WITH METHOD OBJECT-Refactorings. Dabei wird die *GraphSearch*-Methode von der *Graph*-Klasse extrahiert und die neue *GraphSearch*-Klasse erstellt. Diese Methode wird gewählt weil ihre Zugriff-Menge 3 eigene Elemente hat. Die Tabelle 5.3 stellt die Ergebnisse des Refactorings dar.

Tabelle 5.3: Ergebnisse der RFC-Metrik vor und nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring

Originale Werte der RFC-Metrik	Die RFC-Metrik nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring
Max. RFC Value: 46 from Graph Average RFC Value: 10.45454 Number of classes: 11	Max. RFC Value: 45 from Graph Average RFC Value: 9.91666 Number of classes: 12
Graph: 46 (local methods: 29, externe methods: 17) Main: 17 (local methods: 2, externe methods: 15) Vertex: 17 (local methods: 13, externe methods: 4) FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1) WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1) NumberWorkSpace: 3 (local methods: 2, externe methods: 1) Edge: 9 (local methods: 8, externe methods: 1) CycleWorkSpace: 6 (local methods: 5, externe methods: 1) WorkSpace: 6 (local methods: 6, externe methods: 0) Neighbor: 2 (local methods: 2, externe methods: 0) GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)	Graph: 45 (local methods: 29, externe methods: 16) Main: 17 (local methods: 2, externe methods: 15) Vertex: 17 (local methods: 13, externe methods: 4) GraphSearch: 5 (local methods: 2, externe methods: 3) FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1) WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1) NumberWorkSpace: 3 (local methods: 2, externe methods: 1) Edge: 9 (local methods: 8, externe methods: 1) CycleWorkSpace: 6 (local methods: 5, externe methods: 1) WorkSpace: 6 (local methods: 6, externe methods: 0) Neighbor: 2 (local methods: 2, externe methods: 0) GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)

Obwohl die gezielte *Graph*-Klasse wenig verbessert (deren RFC-Wert wird nur von 46 auf 45 reduziert) und die neue Klasse mit einem RFC-Wert von 5 eingeführt wird, bekommt man einen besseren Durchschnittswert als bei Anwendung des MOVE METHODS AND FIELDS-Refactorings. Dies führt darauf zurück, dass die Anzahl der Klassen in diesem Beispiel relativ wenig ist (insgesamt 11 Klassen) und die Einführung einer neuen Klasse schon einen Unterschied in dem Durchschnittswert machen kann. Des Weiteren wird die Anwendung des EXTRACT CLASS-Refactorings betrachtet. Dazu werden 3 Methoden *ShortestPath, ComputeTranspose* und *addAnEdge* in eine neue Klasse, namens *GraphUtility*, extrahiert. Diese Methoden haben die größte gemeinsame Zugriff-Menge (5 Aufrufe auf externe Methoden) und somit kann das Extrahieren den RFC-Wert der Quellklasse Graph verbessern. Die Ergebnisse stellen sich heraus, dass Verbesserungen zu der Metrik tatsächlich durch Anwendung des Refactorings eingebracht wurden. Der RFC-Wert der Graph-Klasse wurde von 46 auf 37 reduziert, davon nur noch 13 externe Methoden. Obwohl die neu eingefügte Klasse *GraphUtility* einen RFC-Wert von 16 hat, wurde der durchschnittliche Wert des Programms immer noch verbessert. Die Tabelle 5.4 fasst die Ergebnisse zusammen.

5.4.2 Refactorings für die DBC-Metrik

Durch die Messung der DBC-Metrik stellt sich heraus, dass Klassen der Beispiel-Anwendung zum großen Teil eine schlechte Kohäsion haben. Als Refactoring-Kandidaten wurden im Abschnitt 5.2.5 folgende Klassen identifiziert: *Graph, Vertex, Edge, CycleWorkSpace*. Ihre DBC-Werte sind über dem Durchschnittswert und sie implementieren die meisten Funktionalitäten des Programms. Theoretisch kann man annehmen, dass wenn zusammenhängende Entitäten in die entsprechende Klasse zusammengepackt werden, können die Kohäsion-Werte verbessert werden. In Bezug auf die DBC-Metrik heißen zwei Entitäten zusammenhängend, wenn ihre Benutzen-Mengen viele gemeinsame Elemente teilen. Findet man dass die Refactoring-Kandidaten in eine vorhandene Klasse verschoben werden sollen, kann man das MOVE METHODS AND FIELDS-Refactoring

Tabelle 5.4: Ergebnisse der RFC-Metrik vor und nach dem EXTRACT CLASS-Refactoring

Originale Werte der RFC-Metrik	Die RFC-Metrik nach dem EXTRACT CLASS-Refactoring
Max. RFC Value: 46 from Graph Average RFC Value: 10.45454 Number of classes: 11	Max. RFC Value: 37 from Graph Average RFC Value: 10.16666 Number of classes: 12
Graph: 46 (local methods: 29, externe methods: 17) Main: 17 (local methods: 2, externe methods: 15) Vertex: 17 (local methods: 13, externe methods: 4) FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1) WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1) NumberWorkSpace: 3 (local methods: 2, externe methods: 1) Edge: 9 (local methods: 8, externe methods: 1) CycleWorkSpace: 6 (local methods: 5, externe methods: 1) WorkSpace: 6 (local methods: 6, externe methods: 0) Neighbor: 2 (local methods: 2, externe methods: 0) GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)	Graph: 37 (local methods: 24, externe methods: 13) Main: 17 (local methods: 2, externe methods: 15) GraphUtility: 16 (local methods: 6, externe methods: 10) Vertex: 17 (local methods: 13, externe methods: 4) FinishTimeWorkSpace: 4 (local methods: 3, externe methods: 1) WorkSpaceTranspose: 4 (local methods: 3, externe methods: 1) NumberWorkSpace: 3 (local methods: 2, externe methods: 1) Edge: 9 (local methods: 8, externe methods: 1) CycleWorkSpace: 6 (local methods: 5, externe methods: 1) WorkSpace: 6 (local methods: 6, externe methods: 0) Neighbor: 2 (local methods: 2, externe methods: 0) GlobalVarsWrapper: 1 (local methods: 1, externe methods: 0)

benutzen. Auf der Seite der Quellklasse hängt der resultierende Wert der Metrik davon ab, ob die verbliebenen Entitäten auch stark miteinander verbunden sind. Wenn es der Fall ist, kann die Kohäsion nach dem Refactoring tatsächlich verbessert werden. Auf der Seite der Zielklasse kann man erwarten, dass die extrahierten Entitäten die vorhandenen Entitäten der Klasse noch mehr kohäsiv machen. Zur Darstellung der Auswirkung dieses Refactorings wird es auf die Beispiel-Anwendung angewendet. Dabei werden die 3 Felder *last, current, accum* und 4 Methoden *stopProfile, startProfile, endProfile* und *resumeProfile* der *Graph*-Klasse in die *Main*-Klasse extrahiert. Diese Entitäten implementieren das *Benchmark*-Feature und obwohl sie in der *Graph*-Klasse ansiedeln, werden sie nur von der *Main*-Klasse benutzt.

Tabelle 5.5: Ergebnisse der DBC-Metrik vor und nach dem MOVE METHODS AND FIELDS-Refactoring

Originale Werte der DBC-Metrik	Die DBC-Metrik nach dem MOVE METHODS AND FIELDS-Refactoring
Worst Cohesion Value: 1.0 from WorkSpace Average Cohesion Value: 0.85563 Number of classes: 11	Worst Cohesion Value: 1.0 from WorkSpace Average Cohesion Value: 0.83353 Number of classes: 11
WorkSpace: 1.0 Main: 1.0 GlobalVarsWrapper: 1.0 Graph: 0.96268 Vertex: 0.95217 Edge: 0.90074 CycleWorkSpace: 0.87595 FinishTimeWorkSpace: 0.70000 WorkSpaceTranspose: 0.69167 Neighbor: 0.68434 NumberWorkSpace: 0.64445	WorkSpace: 1.0 Main: 0.75385 GlobalVarsWrapper: 1.0 Graph: 0.96573 Vertex: 0.95217 Edge: 0.90074 CycleWorkSpace: 0.87595 FinishTimeWorkSpace: 0.70000 WorkSpaceTranspose: 0.69167 Neighbor: 0.68434 NumberWorkSpace: 0.64445

Die Tabelle 5.5 beschreibt die zusammengefassten Ergebnisse der Anwendung des Refactorings. Der DBC-Wert der *Main*-Klasse wurde dadurch deutlich verbessert (von 1.0 auf 0.75). Der Durchschnittswert des Programms hat auch eine positive Veränderung (von 0.85 auf 0.83). Nur die *Graph*-Klasse hat sich dabei verschlechtert, ihrer Wert wurde

von 0.962 auf 0.965 erhöht. Es führt darauf zurück, dass die verbliebenen Entitäten dieser Klasse nicht wirklich mit einander verbunden sind.

Es sei wieder darauf hingewiesen, dass die Refactoring-Kandidaten nicht unbedingt in eine vorhandene Klasse verschoben werden müssen. Findet man, dass sie eine eigene Klasse haben sollen, kann man auf das EXTRACT CLASS-Refactoring zugreifen. Ähnlich wie bei dem MOVE METHODS AND FIELDS-Refactoring wird es auf der Seite der Quellklasse erwartet, dass die verbliebenen Entitäten kohäsiv sind und somit der DBC-Wert verbessert werden kann. Da die extrahierten Entitäten zusammenhängend sind wird eventuell die neue Klasse eine gute Kohäsion haben. Als Beispiel wird ein RFM definiert, welches das Feld *inFile* und die *readNumber*-, *stopBenchmark*- und *runBenchmark*-Methode der *Graph*-Klasse in eine neue Klasse extrahiert. Diese Entitäten implementieren auch das *Benchmark*-Feature und daher wird die neue Klasse als *Benchmark* genannt. Durch Anwendung dieses Refactoring bekommt die *Graph*-Klasse tatsächlich eine positive Änderung in ihrem DBC-Wert (von 0.962 auf 0.960). Die neu eingeführte Klasse hat eine gute Kohäsion von 0.75 und dadurch wurde auch der Durchschnittswert des Programms verbessert. Die Tabelle 5.6 fasst die Ergebnisse nach dem Refactoring zusammen.

Tabelle 5.6: Ergebnisse der DBC-Metrik vor und nach dem EXTRACT CLASS-Refactoring

Originale Werte der DBC-Metrik	Die DBC-Metrik nach dem EXTRACT CLASS-Refactoring
Worst Cohesion Value: 1.0 from WorkSpace Average Cohesion Value: 0.85563 Number of classes: 11	Worst Cohesion Value: 1.0 from WorkSpace Average Cohesion Value: 0.84662 Number of classes: 12
WorkSpace: 1.0 Main: 1.0 GlobalVarsWrapper: 1.0 Graph: 0.96268 Vertex: 0.95217 Edge: 0.90074 CycleWorkSpace: 0.87595 FinishTimeWorkSpace: 0.70000 WorkSpaceTranspose: 0.69167 Neighbor: 0.68434 NumberWorkSpace: 0.64445	WorkSpace: 1.0 Main: 1.0 GlobalVarsWrapper: 1.0 Graph: 0.96015 Vertex: 0.95217 Edge: 0.90074 CycleWorkSpace: 0.87595 Benchmark: 0.75000 FinishTimeWorkSpace: 0.70000 WorkSpaceTranspose: 0.69167 Neighbor: 0.68434 NumberWorkSpace: 0.64445

5.4.3 Refactoring für die CBO-Metrik

Die *Graph*-Klasse wurde durch die Messung der Metrik als der einzige Kandidat für die Refactorings festgestellt. Sie ist mit 8 anderen Klassen gekoppelt während das Programm nur aus 11 Klassen besteht und die anderen Klassen maximal auf zwei Klassen zugreifen (vgl. Abschnitt 5.2.2). Dies verletzt die Design-Prinzipien der OOP, die verlangen, dass eine Klasse wenig von externen Klassen abhängig ist. Demnach zur Unterstützung der Metrik sollten die Methoden, die eine gemeinsame Menge von Klassen benutzen, in die Klasse hineingesteckt werden, wo auch diese Menge von Klassen zugegriffen werden. Passen die Methoden zu der Logik einer vorhandenen Klasse, kann man mittels eines MOVE METHODS AND FIELDS-Refactorings die Transformation erledigen. Dadurch könnte der CBO-Wert der Quellklasse reduziert und der von der Zielklasse erhalten bleiben.

Ein konkretes MOVE METHODS AND FIELDS-Refactorings wird an dieser Stelle auf die Beispiel-Anwendung angewendet um seine Auswirkung auf die CBO-Metrik zu testen. Das MOVE METHODS AND FIELDS-Refactoring extrahiert 3 Methoden *ShortestPath, addEdge* und *ComputeTranspose* in die *Edge*-Klasse. Diese Methoden benutzen zwei Klassen *Neighbor* und *Vertex*, die ebenfalls von der *Edge*-Klasse zugegriffen wird. Die Ergebnisse der Metrik nach dem Refactoring werden in Tabelle 5.7 zusammengefasst.

Tabelle 5.7: Ergebnisse der CBO-Metrik vor und nach dem MOVE METHODS AND FIELDS-Refactoring

Originale Werte der CBO-Metrik	Die CBO-Metrik nach dem MOVE METHODS AND FIELDS-Refactoring
Max. CBO Value: 8 from Graph Average CBO Value: 2.0 Number of classes: 11	Max. CBO Value: 7 from Graph Average CBO Value: 2.0 Number of classes: 11
Graph: 8 Main: 2 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 Vertex: 2 NumberWorkSpace: 2 Edge: 2 CycleWorkSpace: 2 WorkSpace: 0 Neighbor: 0 GlobalVarsWrapper: 0	Graph: 7 Main: 2 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 Vertex: 2 NumberWorkSpace: 2 Edge: 3 CycleWorkSpace: 2 WorkSpace: 0 Neighbor: 0 GlobalVarsWrapper: 0

Während der Durchschnittswert des Programms gleich bleibt wurde der CBO-Wert der *Graph*-Klasse durch das Refactoring verbessert. Das Verschieben der Methoden führte eine zusätzliche externe Klasse in die *Edge*-Klasse ein, demnach ist ihrer Wert um 1 gestiegen. Wie bei der RFC- oder DBC-Metrik kann man auch die Methoden in eine neue Klasse extrahieren, wenn sie nicht zu einer schon existierenden Klasse passen und die Quellklasse gezielt verbessert werden sollte. Dazu kann man das REPLACE METHOD WITH METHOD OBJECT- oder das EXTRACT CLASS-Refactoring zum Einsatz bringen. Um positive Änderungen zu bekommen sollte man Methoden auswählen, die viele eigene Elemente in der Zugriff-Menge haben. Im Folgenden werden zwei RFMs erstellt um die Einflüsse der genannten Refactorings zu experimentieren. Das REPLACE METHOD WITH METHOD OBJECT-Refactoring nimmt aus der *Graph*-Klasse die *StrongComponents*-Methode aus und konstruiert damit eine neue Klasse, namens *StrongComponents*. Die resultierenden Ergebnisse (vgl. Tabelle 5.8) zeigen, dass das Refactoring den CBO-Wert der *Graph*-Klasse verbessern konnte aber durch die Einführung der neuen Klasse der durchschnittliche Wert des Programms verschlechtert wurde.

Die 3 Methoden *ShortestPath, addEdge* und *ComputeTranspose* werden ebenfalls als Kandidaten für das EXTRACT CLASS-Refactoring herausgefunden da sie die maximale gemeinsame Zugriff-Menge teilen. Nach dem Refactoring wird festgestellt (vgl. Tabelle 5.9), dass keine positive Änderung durch das Refactoring eingebracht wurde. Es liegt daran, dass neben der gemeinsamen Zugriff-Menge haben die Methoden auch einige eigenen Zugriffe, die auch zu dem CBO-Wert der Zielklasse beitragen. Darüber hinaus kann man annehmen, dass diese Methoden keine eigene Abstraktion darstellen und daher eher zu einer vorhandenen Klasse gehören sollen.

Tabelle 5.8: Ergebnisse der CBO-Metrik vor und nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring

Originale Werte der CBO-Metrik	Die CBO-Metrik nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring
Max. CBO Value: 8 from Graph Average CBO Value: 2.0 Number of classes: 11	Max. CBO Value: 7 from Graph Average CBO Value: 2.08333 Number of classes: 12
Graph: 8 Main: 2 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 Vertex: 2 NumberWorkSpace: 2 Edge: 2 CycleWorkSpace: 2 WorkSpace: 0 Neighbor: 0 GlobalVarsWrapper: 0	Graph: 7 StrongComponents: 4 Main: 2 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 Vertex: 2 NumberWorkSpace: 2 Edge: 2 CycleWorkSpace: 2 WorkSpace: 0 Neighbor: 0 GlobalVarsWrapper: 0

Tabelle 5.9: Ergebnisse der CBO-Metrik vor und nach dem EXTRACT CLASS-Refactoring

Originale Werte der CBO-Metrik	Die CBO-Metrik nach dem EXTRACT CLASS-Refactoring
Max. CBO Value: 8 from Graph Average CBO Value: 2.0 Number of classes: 11	Max. CBO Value: 8 from Graph Average CBO Value: 2.16666 Number of classes: 12
Graph: 8 Main: 2 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 Vertex: 2 NumberWorkSpace: 2 Edge: 2 CycleWorkSpace: 2 WorkSpace: 0 Neighbor: 0 GlobalVarsWrapper: 0	Graph: 8 GraphUtility: 4 Main: 2 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 Vertex: 2 NumberWorkSpace: 2 Edge: 2 CycleWorkSpace: 2 WorkSpace: 0 Neighbor: 0 GlobalVarsWrapper: 0

5.4.4 Refactorings für die CC-Metrik

Als Refactoring-Kandidaten werden Methoden betrachtet, die einen CC-Wert über 14 haben. In der Beispiel-Anwendung wurde durch die Messung der Metrik die *Shortest-Path*-Methode als einer Kandidat identifiziert. Zur Vereinfachung einer Methode kann man ein komplexes Codefragment oder eine komplexe konditionelle Anweisung nehmen und es bzw. sie in eine neue Methode umwandeln, deren Name den Zweck des Teils oder der Kondition beschreiben. Dazu dienen das EXTRACT METHOD- und das DECOMPOSE CONDITIONAL-Refactoring. Zu beachten ist, dass das Codefragment oder die Kondition möglichst viele konditionelle Sprachkonstruktionen (vgl. Abschnitt 5.2.4) enthalten sollte um bessere Ergebnisse bekommen zu können. Als Erstes wird die Auswirkung des EXTRACT METHOD-Refactorings betrachtet indem man ein RFM definiert, welches das folgende Codefragment aus der ShortestPath-Methode extrahiert (vgl. Listing 5.11).

Listing 5.11: Das zu extrahierende Codefragement der ShortestPath-Methode

```
1  while ( Queue.size()!=0 ){
2    ...
3      for( k=0; k < Uneighbors.size(); k++ ){
4          ...
5          if ( v.dweight > ( u.dweight +  wuv ) ){
6              ...
7              if ( indexNeighbor>=0 ){
8                  ...
9                  int position = Collections.binarySearch( Queue,v,new Comparator() {
10                     public int compare( Object o1, Object o2 ){
11                         ...
12                         if ( v1.dweight < v2.dweight )return -1;
13                         if ( v1.dweight == v2.dweight )return 0;
14                         return 1;
15                     }
16                 });
17                 ...
18                 if ( position < 0 ) {Queue.add( - ( position+1 ),v );}
19                 else {Queue.add( position,v );}
20             }
21         }
22     }
23 }
```

In dem obigen Listing werden nur die konditionellen Konstruktionen angezeigt. Das ganze Codefragment ist in einer *while*-Anweisung enthalten, die im dem ersten Block der ersten Stufe der Block-Hierarchie liegt. Die resultierenden Ergebnisse zeigen, dass der CC-Wert von sowohl der *Graph*-Klasse als auch von dem gesamten Programm deutlich verbessert wird. Für die *Graph*-Klasse wurde ihrer Wert von 19 auf 11 reduziert, welcher in dem empfohlenen Bereich für alle Methoden ist. Die Tabelle 5.10 fasst die Ergebnisse nochmal zusammen.

Tabelle 5.10: Ergebnisse der CC-Metrik vor und nach dem EXTRACT METHOD-Refactoring

Originale Werte der CC-Metrik	Die CC-Metrik nach dem EXTRACT METHOD-Refactoring
Max. Cyclomatic Complexity Value: 19 from Graph Average Cyclomatic Complexity Value: 4.54545 Number of classes: 11	Max. Cyclomatic Complexity Value: 11 from Graph Average Cyclomatic Complexity Value: 3.81818 Number of classes: 11
Graph: 19 - Graph.ShortestPath(): 19 Main: 10 CycleWorkSpace: 8 Vertex: 3 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 NumberWorkSpace: 2 WorkSpace: 1 Neighbor: 1 Edge: 1 GlobalVarsWrapper: 1	Graph: 11 - Graph.ShortestPath(): 11 Main: 10 CycleWorkSpace: 8 Vertex: 3 FinishTimeWorkSpace: 2 WorkSpaceTranspose: 2 NumberWorkSpace: 2 WorkSpace: 1 Neighbor: 1 Edge: 1 GlobalVarsWrapper: 1

Für den Einsatz des DECOMPOSE CONDITIONAL-Refactorings muss eine *if*-Anweisung herausgefunden, die möglichst viele Logik-Operationen beinhaltet. Das ist nicht der Fall mit der *ShortestPath*-Methode. Trotzdem zur Darstellung der Einflüsse dieses Refactorings wird die *checkNeighborAction*-Methode aus der *CycleWorkSpace*-Klasse als ein Beispiel genommen. Dabei wird die Kondition der folgenden *if*-Anweisung (vgl. Listing 5.12) in eine neue Methode extrahiert.

Listing 5.12: Die zu extrahierende *if*-Anweisung der *checkNeighborAction*-Methode

```
1 if ( ( vsource.VertexColor == GRAY ) && ( vtarget.VertexColor == GRAY )
2     && vsource.VertexCycle != vtarget.VertexCycle+1 ){
3     AnyCycles = true;
4 }
```

Die Anwendung des Refactorings hat tatsächlich positive Wirkungen auf den Wert der betroffenen *checkNeighborAction*-Klasse und des ganzen Programms. Durch die zwei Experimente mit dem EXTRACT CLASS- und DECOMPOSE CONDITIONAL-Refactoring kann man beobachten, dass die Komplexität der Originalmethode durch Refactorings auf zwei Methoden aufgeteilt wird. Da der maximale CC-Wert auch der Wert der Klasse ist, sollte die Aufteilung möglichst gleichwertig sein. Somit erhält man die beste Auswirkung von den Refactorings.

5.4.5 Refactorings für die LOC-Metrik

Die Refactoring-Kandidaten der LOC-Metrik sind Methoden und Klassen, deren LOC-Werte über 40 bzw. über 400 sind. Es kann aber passieren, dass eine Klasse einen LOC-Wert unter 400 hat und Methoden mit den LOC-Werten über 40 enthält und umgekehrt. Dementsprechend können die Refactorings EXTRACT METHOD, EXTRACT CLASS und REPLACE METHOD WITH METHOD OBJECT eingesetzt werden. Anhand der Messung-Ergebnisse der Metrik wurde festgestellt, dass die *ShortestPath*-Methode der *Graph*-Klasse, die eine Länge von 105 Codezeilen hat, refaktorisiert werden muss. An dieser Stelle kann man ein EXTRACT METHOD-Refactoring einsetzen um die *ShortestPath*-Methode zu zerkleinern. Dafür werden die Anweisungen von Position 30 bis 38 in eine neue Methode, namens *calculateShortestPath*, extrahiert. Die resultierende Ergebnisse zeigen (vgl. Tabelle 5.11), dass obwohl die *ShortestPath*-Methode nach dem Refactoring kleiner ist (deren Länge jetzt nur noch 80 beträgt), ist die gesamte Klasse wegen des Einfügens der neuen Methode größer geworden.

Tabelle 5.11: Ergebnisse der LOC-Metrik vor und nach dem EXTRACT METHOD-Refactoring

Originale Werte der LOC-Metrik	Die LOC-Metrik nach dem EXTRACT METHOD-Refactoring
Max. LOC Value: 301 from Graph Average LOC Value: 54.09090 Number of classes: 11	Max. LOC Value: 309 from Graph Average LOC Value: 54.81818 Number of classes: 11
Graph: 301 - Graph.ShortestPath: 105 Vertex: 85 Main: 64 CycleWorkSpace: 40 Edge: 31 WorkSpaceTranspose: 16 WorkSpace: 15 Neighbor: 14 FinishTimeWorkSpace: 13 NumberWorkSpace: 10 GlobalVarsWrapper: 6	Graph: 301 - Graph.ShortestPath(): 80 - Graph.calculateShortestPath(): 33 Vertex: 85 Main: 64 CycleWorkSpace: 40 Edge: 31 WorkSpaceTranspose: 16 WorkSpace: 15 Neighbor: 14 FinishTimeWorkSpace: 13 NumberWorkSpace: 10 GlobalVarsWrapper: 6

5.5 Zusammenfassung

Dieses Kapitel stellt eines der wichtigen Teile der Arbeit vor, wobei die gewählten Software-Metriken implementiert, die Refactorings selektiert und automatisiert und die Eigenschaften bei der Anwendung dieser Refactorings zur Unterstützung der Metriken betrachtet werden. Daraus lassen sich folgende Schlussfolgerungen ableiten:

1. Die gewählten Metriken können die Eigenschaften bezüglich der Wartbarkeit eines Programms beschreiben. Dennoch in einigen Fällen reichen sie allein nicht aus und verlangt es von dem Anwender, Kenntnisse über das System zu haben.

2. Das Automatisieren eines Refactorings kann nicht immer nach der manuellen Weise nachgemacht werden da die Semantik eines Programms nicht vollkommen durch die AST-Struktur dargestellt werden kann. Durch einige Änderungen und Beschränkungen kann das Refactoring realisierbar sein, was aber auch dazu führen kann, dass das Refactoring bei Durchführung zusätzliche Komplexität in das Programm einführt.

3. Für die RFC, CBO und DBC-Metrik kann ein Refactoring sowohl positive als auch negative Auswirkungen haben. Das Vorhersagen über die Auswirkung eines Refactorings ist wegen der unterschiedlichen Beziehungen zwischen den Klassen schwer zu realisieren.

4. Für eine bestimmte Metrik können zugleich mehrere Refactorings sie unterstützen. Da ihre Einflüsse schwer vorherzusagen sind, wird empfohlen, alle Möglichkeiten auszuprobieren.

5. Ein Refactoring kann die Metrik einer bestimmten Klasse verbessern und dennoch den Durchschnittswert negativ beeinflussen. Daher sollte das Ziel eines Refactorings vor der Durchführung klar definiert werden.

Die folgende Tabelle fasst die Beziehung zwischen den Refactorings und den Metriken nochmal zusammen.

Tabelle 5.12: Unterstützung der Refactoring für die Metriken

Metrik	ECR	EMR	DCR	RMMOR	MMFR
LOC	X	X		X	
CC		X	X		
RFC	X			X	X
DBC	X				X
CBO	X			X	X

Bis dahin wurden die wirklichen Objekte der Refactorings, die Methoden und Felder, nur anhand der Erkenntnisse über die Beispiel-Anwendung manuell ausgewählt. Ebenfalls wurden die RFMs auch per Hand erstellt, was mühsam und fehleranfällig ist. Aus den gewonnenen Kenntnissen wird in dem nächsten Kapitel einen Prototyp entwickelt, der bei Auswahl der Refactoring-Kandidaten unterstützt und die Durchführung der RFMs automatisiert.

Kapitel 6

Automatisierte Anwendung der Wartungsmetrik unterstützenden Refactorings

Im Kapitel 5 wurden die gewählten Refactorings auf eine Beispiel-Anwendung der Graph-Produktlinie (siehe Abschnitt 5.1) angewendet und deren Auswirkungen auf die wartungsbezogenen Software-Metriken betrachtet. Dazu wurden die Messung der Metriken und die Durchführung der Refactorings bereits implementiert. Dennoch mussten sowohl die Identifizierung der Refactoring-Parameter als auch die Erstellung der entsprechenden RFMs manuell getätigt werden. Einer der Hauptpunkte der vorliegenden Arbeit besteht darin, diesen schwierigen, fehleranfälligen Prozess zu unterstützen und zu automatisieren. Deshalb wird in diesem Kapitel ein Tool vorgestellt, welches diese Aufgabe erfüllt. Dieses Tool wird als ein Plug-in der Eclipse-IDE [1] implementiert. Das Kapitel ist wie folgt gegliedert: Zunächst stellt der Abschnitt 6.1 das allgemeine Implementierungskonzept des Plug-ins dar. Der darauf folgende Abschnitt 6.2 geht genauer in die Strategie zur automatisierten Identifizierung von Refactoring-Parametern für jede der gewählten Metriken ein. Im Abschnitt 6.3 wird das Tool evaluiert, indem es zur Verbesserung der Metriken unterschiedlicher Varianten zweier SPLs zum Einsatz gebracht wird. Der letzte Abschnitt 6.4 schließt das Kapitel mit einer Zusammenfassung der Ergebnisse der Evaluierung.

6.1 Implementierungskonzept des Plug-ins

Im Abschnitt 2.5.1 wurden die technischen und praktischen Kriterien für ein Refactoring-Tool vorgestellt, die das Tool erfüllen muss um in der Praxis einsetzbar zu sein. Die technischen Kriterien hält die Implementierung der gewählten Refactorings im Kapitel 5 ein wobei die AST-Struktur zur Analyse der Programmstruktur benutzt und die Vor- bzw. Nachbedingung zur Gewährleistung der Korrektheit des Programms nach dem Refactoring geprüft wird. Eines der praktischen Kriterien besagt, dass das Tool in eine IDE integriert werden soll, da sich die Entwicklung heutiger Software-Projekte meistens dort befindet. Die Eclipse-IDE wurde gewählt weil sie eine der populärsten IDEs ist [MKF06]

[1] http://www.eclipse.org/

und ihre Systemarchitektur die Entwicklung von eigenen Plug-ins ermöglicht. Es wird an dieser Stelle kurz über die grundlegenden Begriffe von Eclipse informiert, die bekannt sein müssen, um mit Eclipse arbeiten zu können.

- *Workspace*: bezieht sich auf ein Verzeichnis, welches Softwareprojekte und deren entsprechende Dateien bzw. Ressourcen beinhaltet. Man kann mehrere Workspaces definieren aber zu einem Zeitpunkt darf nur ein Workspace benutzt werden.
- *Editor*: stellt die Funktionen zur Verfügung damit man die Dateien öffnen, bearbeiten, speichern und schließen kann.
- *View*: liefert spezifische Informationen über ein Element, das gerade betrachtet wird, oder einen Vorgang, der gerade passiert. Unter den Elementen zählt eine Datei, ein Ordner oder ein ganzes Projekt.
- *Perspektive*: bezeichnet die Anordnung und den Umfang der anzuzeigenden Views und Editors in der Oberfläche der Entwicklungsumgebung. Für unterschiedliche Zwecke (Debuggen, Entwickeln oder Profilieren etc.) werden unterschiedliche Perspektiven bereits vorgefertigt jedoch darf nur eine Perspektive zu einem Zeitpunkt aktiv sein. Prinzipiell ist eine Perspektive eine Sammlung von so genannten Views (dt. Sichten) und Editoren.

Zur Optimierung der nicht-funktionalen Eigenschaften von Produkten einer SPL verfolgt das zu implementierende Plug-in folgende Vorgehensweise. Zuerst muss das Produkt als ein Projekt in dem *Workspace* der IDE eingetragen werden wobei der Quellcode in dem Standardordner *src* liegt. Anschließend wird eine vom Anwender spezifizierte Metrik vom Tool bemessen. Dazu wird das Kontextmenü der *PackageExplorer*-View wie in Abbildung 6.1 mit einem *NFP Measure*-Untermenü erweitert. Um Fehler zu vermeiden (wie z.B. Messen eines Elements, was nicht zum Quellcode zugehörig ist) wird das *NFP Measure*-Untermenü nur aktiviert wenn das Projekt-Element oder ein Quellcode-Ordner selektiert wurde.

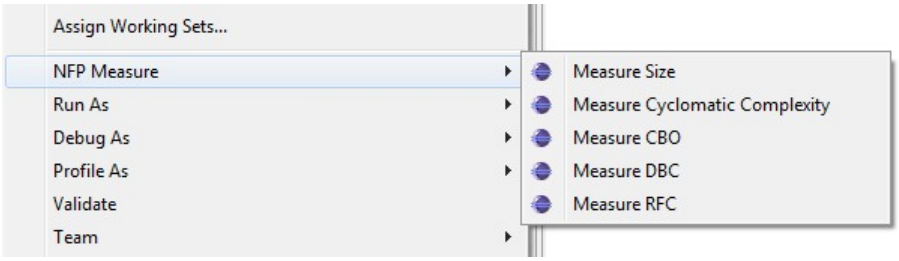

Abbildung 6.1: Das erweiterte Kontextmenü der PackageExplorer-View mit dem NFP Measure-Untermenü

Die Ergebnisse werden in einer entsprechenden View in Form eines Baums dargestellt. Die Wurzel des Baums ist das Programm selbst, deren direkte Kinder die in dem Quellcode-Ordner vorkommenden Klassen sind. Wiederum hat jede Klasse mehrere Kinderknoten, die die Grundelemente der Metrik bilden. Beispielsweise sind für die DBC-Metrik die Felder und Methoden einer Klasse die Grundelemente während für die RFC-Metrik nur die Methoden betrachtet werden. Es sei darauf hingewiesen, dass eine

Methode anonyme bzw. lokale Klassen enthalten kann, die dann als Kinderknoten dieser Methode angesehen werden. Als Beispiel stellen die Abbildung 6.2 und 6.3 den Baum für die CC-Metrik der Graph-Klasse der Graph-SPL graphisch dar.

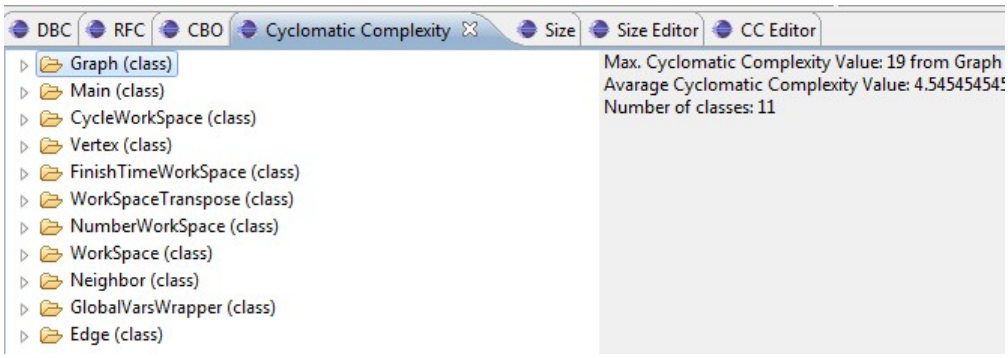

Abbildung 6.2: Der Baum der CC-Metrik einer Instanz der Graph-Produktlinie

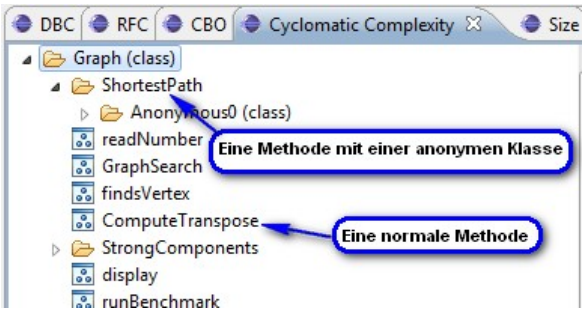

Abbildung 6.3: Grundelemente der Graph-Klasse

Da die implementierten Refactorings Methoden bzw. Felder als Parameter der Transformation nehmen, werden als Nächstes potenzielle Refactoring-Objekte der Kandidat-Klassen durch entsprechende heuristische Strategien ermittelt. Diese Strategien werden in den entsprechenden Abschnitten genauer eingegangen. Danach wird ein RFM generiert, welches dem Interface des Refactorings konform ist und die Transformationen beschreibt. Während dieses Prozesses wird der Anwender zusätzliche Informationen gefragt werden, die für die Ausführung des Refactorings benötigt sind (z.B. Name der neuen Klasse bei REPLACE METHOD WITH METHOD OBJECT-Refactoring oder Name der neuen Methode bei EXTRACT METHOD-Refactoring etc.). Die Eingabe solcher Informationen ist notwendig da sie nicht einfach durch die AST-Struktur ermittelt werden können. Nach Durchführung des Refactorings schließt sich eine Optimierungsphase für eine gewählte Metrik. Das weitere praktische Kriterium verlangt, dass ein Refactoring-Tool die Möglichkeit zur Verfügung stellen soll, ein Refactoring rückgängig machen zu können. Somit kann man für eine Metrik mehrere Refactorings ausprobieren und das Refactoring mit dem besten Ergebnis herausfinden. Dies wird in dem Plug-in erfüllt, indem die Transformationen nur auf eine Kopie des Quellcodes angewendet werden. Jedes Refactoring erstellt einen eigenen Refactoring-Ordner und kopiert die Quellcode-Dateien des Projekts in einen Unterordner. Die Ergebnisse des Refactorings werden in einem anderen Unterordner gelagert, namens *newsrc*. Der originale Quellcode bleibt somit unberührt während der neue Quellcode zu jeder Zeit einfach aus dem Projekt gelöscht

werden kann. Weiterhin stellt das Plug-in mit diesem Mechanismus die Möglichkeit zur Verfügung, schrittweise Refactorings auszuführen da der *newsrc*-Ordner nun als neuer Quellcode-Ordner des Projekts ergo neuer Ausgangspunkt für eine neue Optimierungsphase (welches das Messen, Finden der Refactoring-Kandidaten und Ausführen des Refactorings umfasst) betrachtet werden kann. Dieser Ansatz hat den Nachteil, dass die Größe des Projekts nach vielen sukzessiven Anwendungen der Refactorings schnell zunehmen kann. An dieser Stelle muss man entscheiden, ob der generierte Code bis zu einem gewissen Refactoring-Schritt in den ursprünglichen Quellcode-Ordner, den *src*-Ordner, übernommen werden kann. Wenn es möglich ist kann man dann alle unbenötigte Refactoring-Ordner löschen.

6.2 Strategien zur automatisierten Identifizierung von Refactoring-Parametern

Anhand der Messergebnisse der gewählten Metriken kann lediglich direkt festgestellt werden, welche Klassen (im Falle der RFC-, CBO-, DBC- und LOC-Metrik) bzw. welche Methoden (in Falle der CC- und LOC-Metrik) refactorisiert werden sollen. Die Refactorings, die diese Metriken unterstützen, verlangen konkretere Informationen. Genauer gesagt müssen die EXTRACT CLASS-, REPLACE METHOD WITH METHOD OBJECT- und MOVE METHODS AND FIELDS-Refactorings wissen, mit welchen Methoden und Feldern sie operieren sollen. Analog dazu muss für eine Kandidat-Methode der CC- bzw. LOC-Metrik bestimmt werden, welche Anweisungen ein EXTRACT METHOD- bzw. DECOMPOSE CONDITIONAL-Refactoring extrahieren sollen. Um die Identifizierung der Refactoring-Parameter automatisieren zu können wurden die Metriken auf solche Weise implementiert, dass sie die Methoden und Felder als Bausteine der Metriken einnehmen. Jedoch die Interpretierung der Ergebnisse wurde bis dahin (im Abschnitt 5.4) nur manuell gemacht, was ermüdend und fehleranfällig ist. Dieser Prozess soll in diesem Abschnitt durch heuristische Strategien unterstützt werden.

6.2.1 Identifizierung der Refactoring-Parameter für die RFC-Metrik

Die Messergebnisse der RFC-Metrik werden in einer separaten View, namens *RFC*, wie in Abbildung 6.4 dargestellt. Die linke Seite der RFC-View zeigt die hierarchische Struktur der Klassen und ihrer entsprechenden Methoden, die zur Berechnung der RFC-Metrik nötig sind. Die Klassenknoten sind mit ihren entsprechenden RFC-Werten versehen. Obwohl die Metrik die Summe der Anzahl von lokalen und externen Methoden ist (vgl. Abschnitt 5.2.3), sollte das Verhältnis zwischen den lokalen und externen Methoden einer Klasse berücksichtigt werden, da eine Klasse mit wenigen externen Methoden lose Kopplungen zu anderen Klassen darstellt und es so auch von den Prinzipien des objektorientierten Designs empfohlen wird. Deshalb werden die Messergebnisse der Klassen in der RFC-View auch nach Anzahl der externen Methoden sortiert. Konkrete Information über das Verhältnis zwischen lokalen und externen Methoden einer Klasse kann über Doppelklick auf dem Klassenknoten dargestellt werden. Auf der rechten Seite werden die Ergebnisse der Metrik zusammengefasst. Da die RFC-Metrik durch das EXTRACT

CLASS-, das REPLACE METHOD WITH METHOD OBJECT- oder das MOVE METHODS AND FIELDS-Refactoring verbessert werden kann, werden drei Strategien zur Entdeckung der Kandidaten vorgestellt, die über das Kontextmenü der View zugreifbar sind. Es sei darauf hingewiesen, dass Konstruktoren und Methoden, die eine Methode der Oberklasse überschreiben oder von Methoden der Unterklassen überschrieben werden, nicht als Objekte der oben genannten Refactorings angesehen werden können. Deshalb werden solche Konstruktoren und Methoden explizit in der View markiert (mit der Bezeichnung „constructor“ bzw. mit dem „*“- Zeichen).

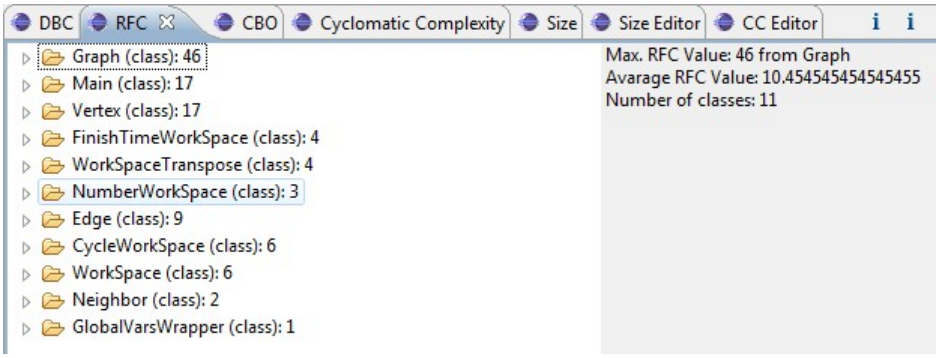

Abbildung 6.4: Die RFC-View

Für das REPLACE METHOD WITH METHOD OBJECT-Refactoring müssen die Kandidat-Methoden viele eigene Elemente in der Zugriff-Menge besitzen (siehe Abschnitt 5.4.1). Detaillierte Informationen über die Zugriff-Menge bzw. die eigenen Elemente einer Methode lassen sich durch Doppelklick auf den entsprechenden Knoten anzeigen. Aus der Methodenmenge der gewählten Klasse wird eine Liste erstellt, die nach Anzahl der eigenen Zugriff-Elemente der Methoden sortiert wird. Die Abbildung 6.5 stellt als Beispiel die Liste der Graph-Klasse dar. Nachdem eine potenzielle Methode ausgewählt wurde, wird der Name der neu zu erstellenden Klasse gefragt und dann die ausführbare Datei angelegt.

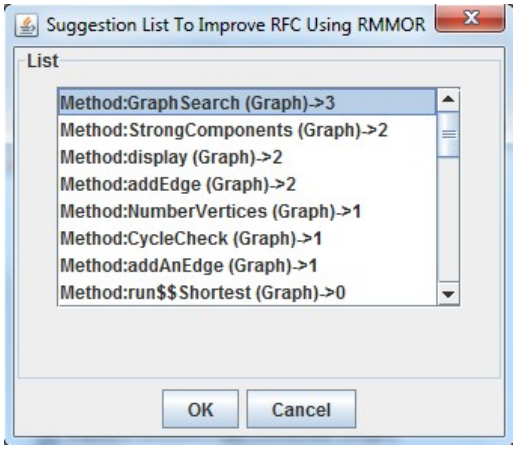

Abbildung 6.5: Vorschläge des REPLACE METHOD WITH METHOD OBJECT-Refactorings für die RFC-Metrik

Die Refactoring-Kandidaten für das MOVE METHODS AND FIELDS-Refactoring sind Methoden, die mehr Methoden von einer Fremdklasse benutzen als die von ih-

ren enthaltenden Klasse. Zur Entdeckung solcher Methoden wird zuerst eine Liste von potenziellen Zielklassen einer gewählten Quellklasse erfasst. Unter den potenziellen Klassen zählt man Klassen, deren lokale Methoden von Methoden der Quellklasse zugegriffen werden. Je mehr solche Zugriffe durch Verschieben der Methoden entfernt werden, desto besser wird die RFC-Metrik. Deshalb wird die Liste der Zielklassen auch nach Anzahl der aufgerufenen Methoden dieser Klassen sortiert. Die Abbildung 6.6 zeigt diese Liste der Graph-Klasse.

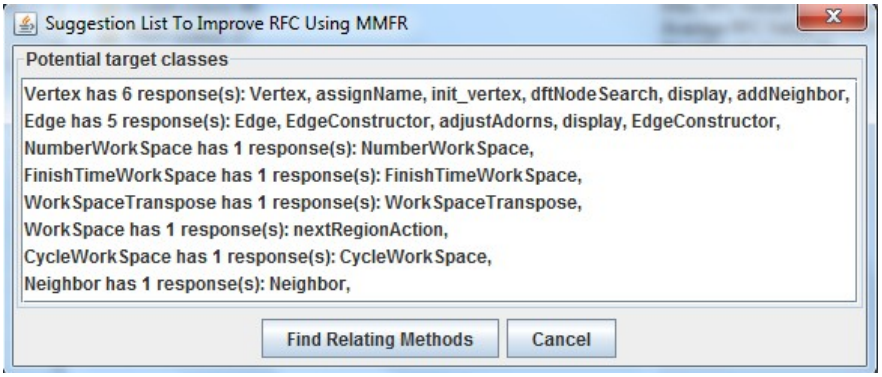

Abbildung 6.6: Liste der potenziellen Zielklasse des MOVE METHODS AND FIELDS-Refactorings für die RFC-Metrik der Graph-Klasse

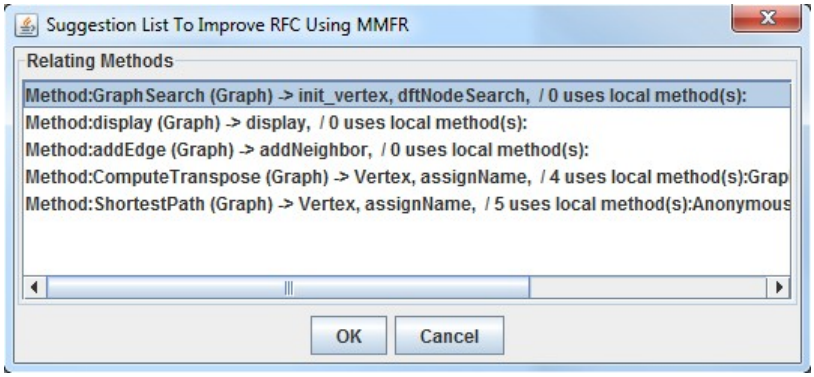

Abbildung 6.7: Liste der potenziellen Methoden des MOVE METHODS AND FIELDS-Refactorings für die RFC-Metrik der Graph-Klasse

Nachdem die Zielklasse gewählt wurde, wird wiederum eine Liste erstellt, die alle Methoden der Quellklasse auflistet, die mindestens eine Methode der Zielklasse aufrufen. Darüber hinaus wird auch informiert, welche lokale Methoden der Quellklasse von den aufgelisteten Methoden zugegriffen werden. Da die Menge der zugegriffenen lokalen Methoden kleiner als die Menge der externen Methoden sein sollte, werden in dem Beispiel der Graph-Klasse, wobei die *Vertex*-Klasse als Zielklasse selektiert wurde, nur die *GraphSeach*-, *display*- und *addEdge*-Methode gewählt, die keine lokale Methode benutzen (vgl. Abbildung 6.7). Es sei aber hingewiesen, dass man die Menge der lokalen Methoden berücksichtigen sollte, weil darin Methoden vorhanden sein könnten, die man auch als Refactoring-Parameter identifiziert. Wenn dies der Fall ist sollten diese Methoden nicht mehr in der Menge der lokalen Methoden betrachtet werden. In dem Beispiel der Graph-Klasse benutzen die Methoden *ComputeTranspose* und *ShortestPath* jeweils

zwei Methoden der Zielklasse, dennoch greifen sie auch auf 4 bzw. 5 lokalen Methoden zu. Deshalb können sie nicht als Refactoring-Parameter angesehen werden.

Im Vergleich zu dem MOVE METHODS AND FIELDS-Refactoring muss man sich nicht beim EXTRACT CLASS-Refactoring um externe Methoden einer bestimmten Klasse kümmern da die Zielklasse jetzt eine neue Klasse ist. Stattdessen sollte eine Gruppe von Methoden herausgefunden werden, die auf viele gleiche Methoden zugreifen und zugleich sich wenige Aufrufe auf gleiche Methoden mit dem Rest der Quellklasse teilen. Da es möglicherweise einige solche Gruppen geben und die Berechnung der optimalen Lösung schwer sein kann, wird an dieser Stell eine heuristische Strategie angewendet. Beginnend werden Paare von Methoden der Quellklasse in einer Liste angegeben, die gleiche externe Methoden aufrufen. Diese Liste wird nach Anzahl der ähnlichen Methodenaufrufe des jeweiligen Paars wie in Abbildung 6.8 sortiert. Wählt man ein Paar aus der Liste, werden erst die Methoden dieses Paars in eine Liste hinzugefügt, die alle Methoden registriert, die später als Parameter des EXTRACT CLASS-Refactorings eingegeben werden. Anhand dieser Liste werden weitere Methoden entdeckt, die sich Zugriffe auf gleiche externe Methoden mit den selektierten Methoden teilen. Fügt man eine verwandte Methode in die Liste der selektierten Methoden hinzu, kann sich eventuell die Menge der verwandten Methoden ändern da die neu eingefügte Methode andere verwandte Methoden haben könnte, die nicht in der aktuellen Menge vorhanden sind. Deshalb sollte nach jeder Änderung der selektierten Methoden die Menge der verwandten Methoden entsprechend aktualisiert werden. Dieser Prozess wird durch Abbildung 6.9 veranschaulicht. Die *addAnEdge* ist eine verwandte Methode mit den selektierten Methoden *ShortestPath* und *ComputeTranspose* weil sie auf dieselbe externe Methode mit der *ShortestPath*-Methode zugreift.

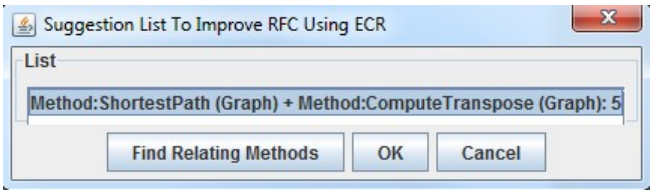

Abbildung 6.8: Liste der Paare von Methoden, die gemeinsame externe Methoden aufrufen

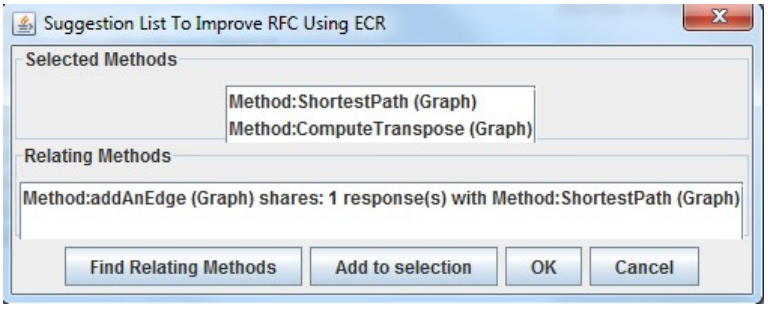

Abbildung 6.9: Liste der potenziellen Methoden des MOVE METHODS AND FIELDS-Refactorings für die RFC-Metrik der Graph-Klasse

6.2.2 Identifizierung der Refactoring-Parameter für die DBC-Metrik

Ähnlich wie bei der RFC-Metrik werden die Messergebnisse der DBC-Metrik in einer separaten View, namens *DBC*, angezeigt. Die Klassen werden nach ihren DBC-Werten absteigend geordnet und die mit den höchsten Werten werden als potenzielle Kandidaten für Refactorings betrachtet. Zu beachten ist, dass Interface-Klassen oder Klassen, die nichts implementieren, vorkommen können, die den negativsten Wert von 1 haben. Solche Klassen können aber nicht durch Refactorings verbessert werden und deshalb werden nicht in Betrachtung gezogen. Um die tatsächlichen Kandidaten (unter den Klassen mit höchsten DBC-Werten) herauszufinden, kann man einfach die Struktur der Klassen in der View durchgehen. Implementiert eine Klasse mit einem hohen DBC-Wert viele Methoden, sollte sie eventuell refactorisiert werden.

Wie im Abschnitt 5.4.2 festgestellt wurde, kann man die DBC-Metrik mit zwei Refactorings unterstützen: das EXTRACT CLASS- und das MOVE METHODS AND FIELDS-Refactoring. Da die Essenz zur Verbesserung der DBC-Metrik darin liegt, zusammenhängende Elemente (Felder und Methoden) in der passenden Klasse zu packen, werden bei dem MOVE METHODS AND FIELDS-Refactoring für eine gewählte Quellklasse alle Klassen identifiziert, die bessere Kohäsion zu bestimmten Elementen der Quellklasse haben. In [Fow99] wird empfohlen, dass das Verschieben der Elemente zuerst mit den Feldern stattfindet, da Felder einfache und eindeutige Beziehungen zu den Klassen haben sollten. Infolgedessen werden für eine potenzielle Zielklasse alle Felder der Quellklasse herausgefunden, die besser mit dieser Zielklasse als mit der Quellklasse verbunden sind. Erst wenn eine Zielklasse keine solche Beziehungen zu den Feldern der Quellklasse hat werden ihre Beziehungen zu den Methoden der Quellklasse betrachtet. Hat diese Zielklasse immer noch keine bessere Kohäsion zu den Methoden der Quellklasse wird sie aus der Liste der Zielklassen entfernt. Die vorgeschlagene Liste der Quellklasse *Graph* in Abbildung 6.10 verdeutlicht diese Strategie, wobei man feststellen kann, dass die Felder *last, current, accum, vertices* und *inFile* in die *Main*-Klasse verschoben werden sollten.

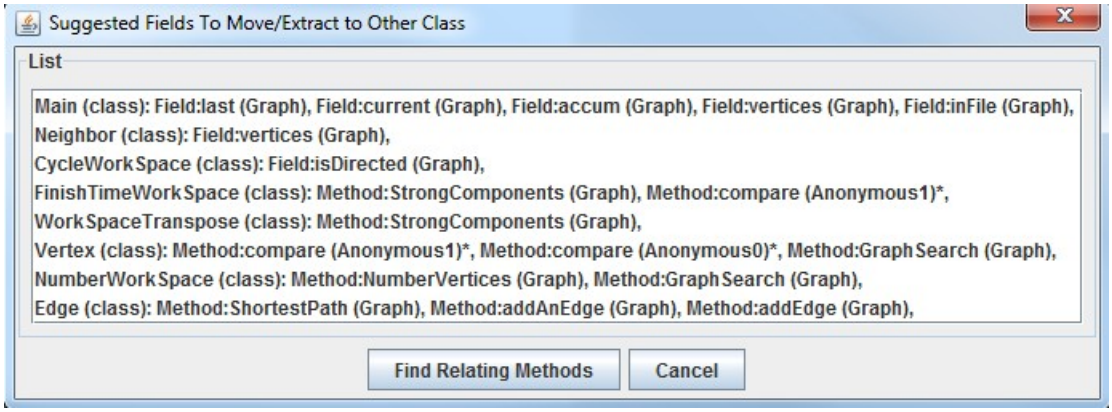

Abbildung 6.10: Potenzielle Zielklasse des MOVE METHODS AND FIELDS-Refactorings für die DBC-Metrik

Der nächste Schritt der Strategie besteht darin, verwandte Methoden der Kandidat-Felder bzw. -Methoden zu bestimmen. Eine Methode heißt verwandt zu den Kandidaten

wenn sie ein Kandidat-Feld oder eine Kandidat-Methode zugreift bzw. wenn sie von einer Kandidat-Methode zugegriffen wird. Die Liste der verwandten Methoden werden nach Anzahl der entsprechenden Kandidaten sortiert. Für die im obigen Schritt identifizierten Kandidat-Felder der *Graph*-Klasse wird diese Liste wie in Abbildung 6.11 vorgestellt. Anhand der vorgeschlagenen Methoden kann man leicht erkennen, dass zugleich 3 zusammenhängende Gruppen in Bezug zu der Zielklasse *Main* identifiziert werden können: eine mit den Feldern *last, current, accum*, eine mit dem Feld *vertices* und eine mit dem Feld *inFile*. Die erste Gruppe wird aufgrund ihrer starken Zugehörigkeit (fast jede verwandte Methode dieser Gruppe greifen auf alle Kandidat-Felder zu) als Parameter für das MOVE METHODS AND FIELDS-Refactoring übernommen.

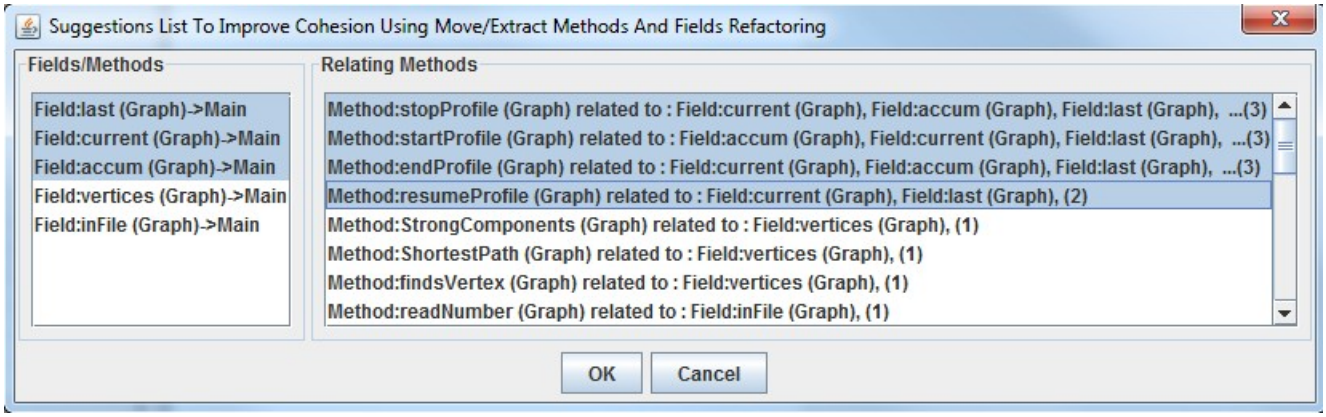

Abbildung 6.11: Verwandte Methoden der Kandidat-Felder für das MOVE METHODS AND FIELDS-Refactoring der *Graph*-Klasse

Die oben genannte Strategie zur Identifizierung von Refactoring-Parametern des MOVE METHODS AND FIELDS-Refactorings kann auch für das EXTRACT CLASS-Refactoring angewendet werden. Der Unterschied von den beiden Refactorings liegt darin, dass das EXTRACT CLASS-Refactoring eine neue Klasse erstellt und demnach kann man zugleich mehrere zusammenhängende Gruppen in Bezug zu mehreren Zielklassen in diese neue Klasse extrahieren. In der ersten Phase können unterschiedliche Felder und Methoden gewählt werden, die mehr zu Fremdklassen als zu der enthaltenen Klasse kohäsiv sind (vgl. Abbildung 6.12). Es sei darauf wieder hingewiesen, dass an dieser Stelle ein Kandidat-Feld mehr Priorität als eine Kandidat-Methode haben sollte. Nachdem man die Kandidat-Felder und -Methoden selektiert hat, erfolgt die Suche der verwandten Methoden wie beim MOVE METHODS AND FIELDS-Refactoring.

6.2.3 Identifizierung der Refactoring-Parameter für die CBO-Metrik

Ähnlich wie bei der RFC-Metrik kann die CBO-Metrik durch die 3 Refactorings EXTRACT CLASS, REPLACE METHOD WITH METHOD OBJECT und MOVE METHODS AND FIELDS verbessert werden. Die Entdeckung der Refactoring-Parameter für das REPLACE METHOD WITH METHOD OBJECT-Refactoring bzw. das EXTRACT CLASS-Refactoring hält die gleiche Strategie der RFC-Metrik ein da die Aufgabe dieses Prozesses liegt auch darin, eine Methode bzw. eine Gruppe von Methoden

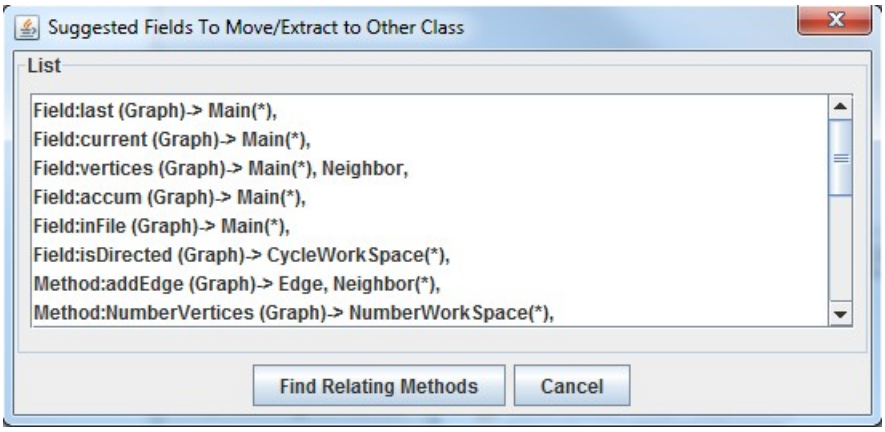

Abbildung 6.12: Kandidat-Felder und -Methoden des EXTRACT CLASS-Refactorings für die DBC-Metrik der *Graph*-Klasse

herauszufinden, die möglichst viele eigene Elemente in ihrer Zugriff-Menge besitzt. Für das MOVE METHODS AND FIELDS-Refactoring werden in der ersten Phase alle potenziellen Zielklassen festgestellt, die sich gemeinsame Kopplungen mit den Methoden der Quellklasse teilen. Solche Methoden werden als verwandte Methoden einer Zielklassen bezeichnet da sie die Zielklasse benutzen oder sie auf gleiche externe Klassen wie die Zielklasse zugreifen. Die Liste der Zielklassen wird nach der Menge ihrer verwandten Methoden aufsteigend geordnet, da je mehr verwandte Methoden eine Zielklasse hat desto höher ist die Wahrscheinlichkeit, dass die verschobenen Methoden weitere Kopplungen in die Zielklasse einführen können. Darüber hinaus sollte man auch darauf achten, dass durch Verschieben eine Kopplung zwischen der Zielklasse und der Quellklasse entstehen kann, die vorher nicht existiert, weil die verschobene Methode Felder oder andere Methoden der Quellklasse auch benutzen kann. Wenn diese Kopplung aber schon vor dem Verschieben vorhanden ist, muss man sich nicht mehr darum kümmern. Diese Information wird in der Vorschlagsliste wie in Abbildung 6.13 präsentiert.

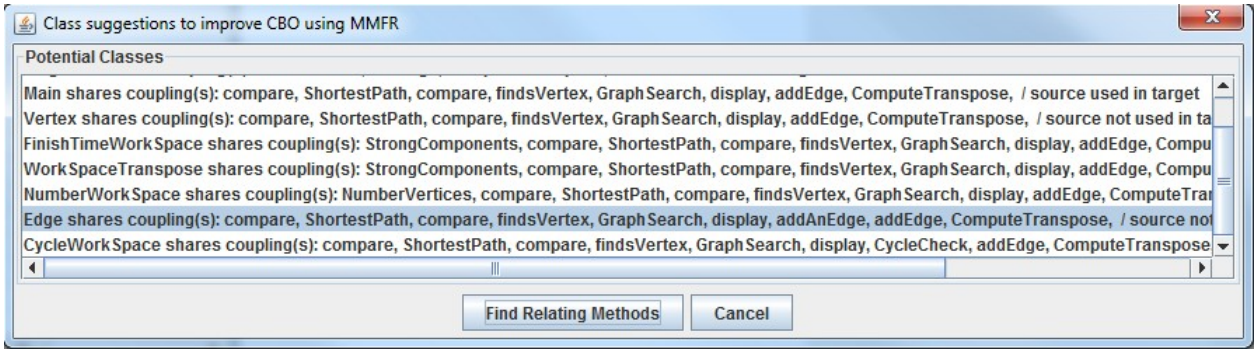

Abbildung 6.13: Potenzielle Zielklassen des MOVE METHODS AND FIELDS-Refactorings für CBO-Metrik der *Graph*-Klasse

Der nächste Schritt dient der Auswahl der verwandten Methoden, da nicht alle davon für das Verschieben in die Zielklasse geeignet sind. Folgende Aspekte muss man dabei berücksichtigen:

- Wenn eine Methode von anderen Methoden der Quellklasse benutzt wird, die nicht

mit in die Zielklasse extrahiert werden, entsteht somit eine Kopplung von der Quellklasse zu der Zielklasse

- Wenn die extrahierte Methode Felder der Quellklasse benutzt oder Methoden zugreift, die nicht mit in die Zielklasse extrahiert werden, entsteht auch eine Kopplung in der Gegenrichtung (von Zielklasse zu Quellklasse)
- Neben den gemeinsamen Kopplungen kann eine Methode auch eigene Kopplungen zu anderen Klassen beinhalten, die die Zielklasse nicht benutzen. Dies erhöht den Wert der CBO-Metrik der Zielklasse.

Wie in Abbildung 6.14 verdeutlicht wird, werden die gemeinsamen Kopplungen zwischen einer verwandten Methode und der Zielklasse mit Großbuchstaben dargestellt. Konkrete Informationen über die Benutzen-Beziehungen kann man über Doppelklick auf die entsprechende Methode bekommen. Für die *Edge*-Zielklasse sind aus der Menge der verwandten Methoden lediglich drei Methoden *ShortestPath, addEdge* und *ComputeTranspose* geeignet, da sie die gleichen Klassen wie die Zielklasse benutzen.

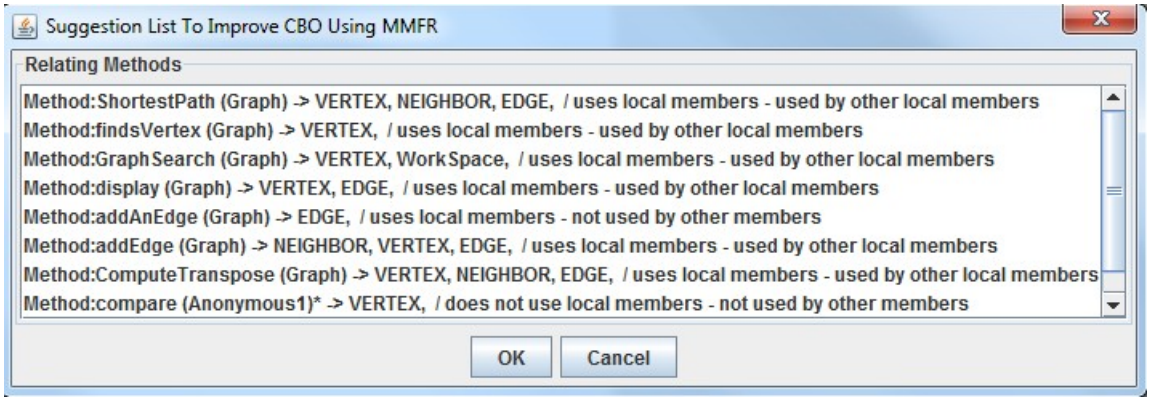

Abbildung 6.14: Verwandte Methoden des MOVE METHODS AND FIELDS-Refactorings für CBO-Metrik

6.2.4 Identifizierung der Refactoring-Parameter für die CC- und LOC-Metrik

Da die Vorgänge zur Entdeckung von Refactoring-Parametern dieser zwei Metriken in vielen Punkten ähnliche Merkmale aufweisen, werden diese Vorgänge hier gemeinsam vorgestellt. Beide Metriken können durch das EXTRACT METHOD-Refactoring unterstützt werden: für die LOC-Metrik kann eine lange Methode in zwei kleinere Methoden aufgeteilt werden und für die CC-Metrik wird die hohe Komplexität der Originalmethode durch zwei einfachere Methoden vermindert. Dazu muss man in der Lage sein, die zu extrahierenden Anweisungen der Originalmethode zu bestimmen. Ergänzend sei darauf hingewiesen, dass die Anweisungen kontinuierlich und in der gleichen Block-Ebene sein müssen (vgl. Abschnitt 5.3.1). Infolgedessen wird eine spezielle View entwickelt, die die Block-Struktur des Quellcodes der Kandidat-Methode in Form eines Baums darstellt. Wählt man eine Gruppe von aufeinander folgenden Anweisungen wird es automatisch ermittelt, in welcher Block-Hierarchie die Anweisungen siedeln und welche Position die

erste bzw. die letzte Anweisung der Gruppe in dem Block besitzt. Als Beispiel wird in Abbildung 6.15 ein Ausschnitt der View für die *ShortestPath*-Methode angezeigt. Man kann erkennen, dass die *while*-Anweisung in Position 26 des ersten Blocks ein guter Kandidat für das EXTRACT METHOD-Refactoring ist da sie fast die Hälfte der Komplexität der Methode beiträgt. In dem Bild ist auch die Struktur der unteren Blöcke dieser *while*-Anweisung zu sehen. Damit ist es auch möglich z.B. die *if*-Anweisung in Position 4 des Unterblocks *1.3.1* zu extrahieren.

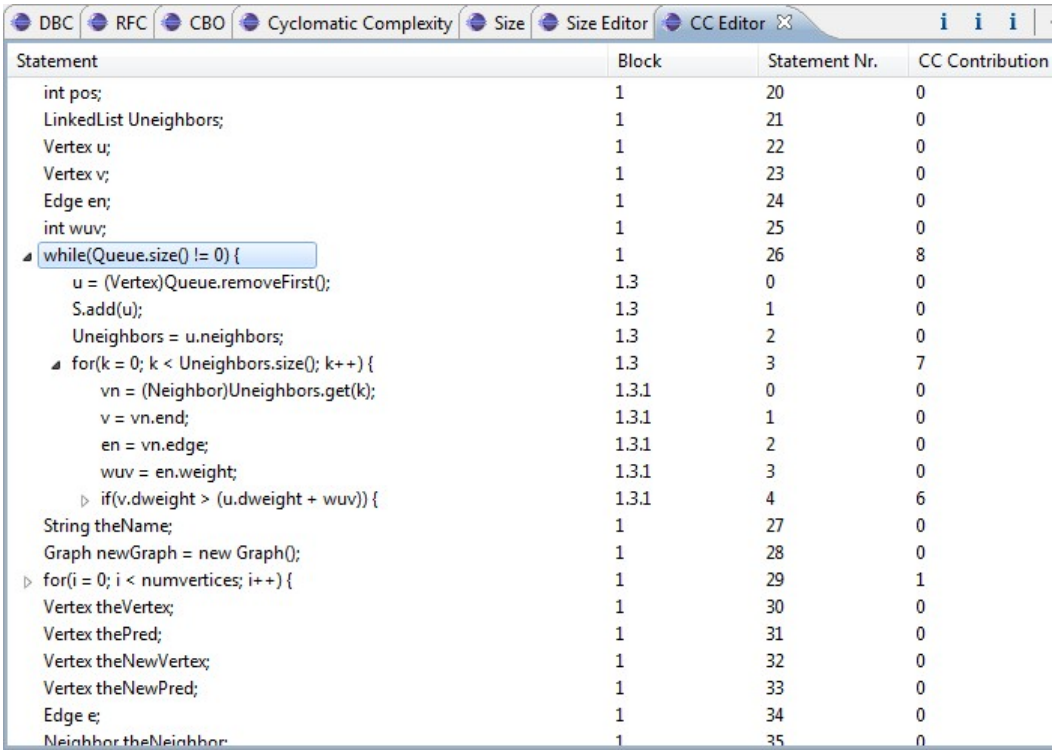

Abbildung 6.15: Die CC Editor-View des EXTRACT METHOD-Refactorings für die CC-Metrik

Das EXTRACT METHOD-Refactoring für die LOC-Metrik benutzt eine ähnliche View, namens *Size Editor*, wie bei der CC-Metrik. Anstatt der Beiträge zu Komplexität werden die Beiträge zu Anzahl der Codezeilen jeder Anweisung dargestellt und als Indikator zur Bestimmung der zu extrahierenden Anweisungen benutzt. Neben dem EXTRACT METHOD-Refactoring kann auch das DECOMPOSE CONDITIONAL-Refactoring die CC-Metrik verbessern indem sie die komplizierte Konditionen einer *if*-Anweisung in eine neue Methode packt. Solche *if*-Anweisungen können auch durch die *CC Editor*-View identifiziert werden. Für die LOC-Metrik kann das REPLACE METHOD WITH METHOD OBJECT-Refactoring zum Einsatz gebracht werden, um eine lange Klasse zu zerkleinern. Dazu wird eine Liste von Methoden der gewählten Quellklasse erstellt und nach den Werten der LOC-Metrik dieser Methoden sortiert.

6.3 Evaluierung des Plug-ins

Die Korrektheit und Anwendbarkeit des implementierten Plug-ins wird durch weitere Test-Anwendungen von unterschiedlichen SPLs evaluiert. Dabei werden die Einflüsse

der Anwendung eines bestimmten Refactorings nicht nur auf die gezielte Metrik sondern auch auf andere Metriken analysiert. Ein weiter zu betrachtender Punkt ist die Möglichkeit, unterschiedliche Refactorings für unterschiedliche Zielmetriken zu kombinieren. Dafür wird das folgende Testszenario eingesetzt: Beginnend wird eine bestimmte Metrik durch ein Refactoring verbessert. Da es mehrere Refactorings geben kann, die diese Metrik unterstützen können, wird nur das mit dem besten Ergebnis betrachtet. Die Änderungen anderer Metriken nach diesem Refactoring werden ermittelt und die Metrik, die am negativsten beeinflusst wurde, sollte wieder durch Refactorings verbessert werden. Danach werden die Zustände von sowohl den beiden unterstützten Zielmetriken als auch den anderen Metriken erfasst. Alle in der Evaluierung benutzten SPLs werden aus den Beispiel-SPLs von FeatureHouse [2] genommen.

6.3.1 Evaluierung mit der TankWar-SPL

Die TankWar-SPL implementiert ein Videospiel für sowohl normale Computer als auch Mobiltelefone. Mit 25 Features kann diese SPL als eine kleine SPL klassifiziert werden. Als Erstes wird die zu prüfende Variante mit den in Abbildung 6.16 gewählten Features generiert. Daraus werden 19 Klassen erstellt, die in ein gemeinsames Paket, namens *tankwarfull*, zusammengepackt werden. Ein neues Projekt wird für diese Variante mit dem entsprechenden Quellcode in der Eclipse-IDE angelegt, die mit dem Refactoring Plug-in vorgesehen wurde. Um die Effekte der Refactorings feststellen zu können, werden an dieser Stelle die originalen Werte aller 5 Metriken durch Listing 6.1 zusammengefasst.

Listing 6.1: Messergbnisse der Metriken für die TankWar-Variante

```
1  Max. RFC Value: 60 from Maler
2  Avarage RFC Value: 13.473684210526315
3  Number of classes: 19
4
5  Worst Cohesion Value: 1.0 from MIDlet
6  Avarage Cohesion Value: 0.8316004417335351
7  Number of classes: 19
8
9  Max. CBO Value: 9 from TankManager
10 Avarage CBO Value: 2.473684210526316
11 Number of classes: 19
12
13 Max. Cyclomatic Complexity Value: 104 from Maler
14 Avarage Cyclomatic Complexity Value: 15.578947368421053
15 Number of classes: 19
16
17 Max. LOC Value: 722 from Maler
18 Avarage LOC Value: 140.52631578947367
19 Number of classes: 19
```

Automatisierte Refactorings für die RFC-Metrik

Obwohl die *Maler*-Klasse den höchsten RFC-Wert hat, ist der Refactoring-Kandidat die *TankManager*-Klasse da sie mehr externe Methoden benutzt. Das Refactoring Plug-in kann Refactoring-Möglichkeiten für das MOVE METHODS AND FIELDS- und REPLACE METHOD WITH METHOD OBJECT-Refactoring herausfinden. Das bessere Ergebnis liefert das REPLACE METHOD WITH METHOD OBJECT-Refactoring,

[2] http://fosd.de/fh

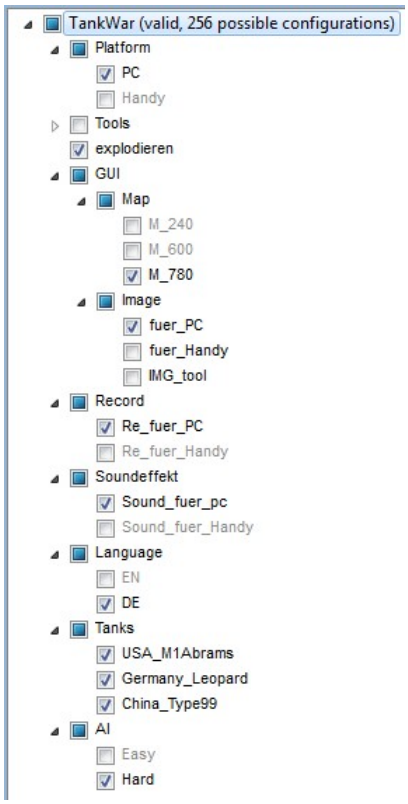

Abbildung 6.16: gewählte Features der Variante der TankWar-SPL

wobei die *malenkontrolle__wrappee__fuer_PC*-Methode mit 7 eigenen Zugriffen in ihrer Zugriff-Menge durch ein Methodenobjekt ersetzt wird. Dadurch wird der RFC-Wert der *TankManager*-Klasse von 50 auf 45 und der Durchschnittswert des Programms von 13.47368 auf 13.05 reduziert. In Bezug zu Änderungen anderer Metriken werden die Messergebnisse nach dem Refactoring in folgender Tabelle 6.1 zusammengefasst.

Tabelle 6.1: Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für RFC-Metrik

Originale Werte der Metriken	Die Metriken-Werte nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring für RFC-Metrik
Number of classes: 19	Number of classes: 20
Max. RFC Value: 60 from Maler - TankManager: 50 Avarage RFC Value: 13.47368	Max. RFC Value: 60 from Maler - TankManager: 45 Avarage RFC Value: 13.05
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.82676
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 10 from TankManager Avarage CBO Value: 2.55
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.3
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 722 from Maler Avarage LOC Value: 134.0

Negative Auswirkungen des Refactorings erhält die CBO-Metrik da die *TankManager*-Klasse zusätzliche Kopplung zu der neu erstellten Klasse haben muss. Im Gegensatz dazu werden durch Einführung der neuen Klasse die 3 restlichen Metriken (für sowohl die Quellklasse als auch den Durchschnittswert) verbessert. An dieser Stelle wird für die Analyse der Eigenschaften von sukzessiven Refactorings die CBO-Metrik als die nächst zu

verbessernde Metrik betrachtet. Zu dieser Aufgabe werden 6 Methoden der *TankManager*-Klasse durch ein MOVE METHODS AND FIELDS-Refactoring in die *Explodieren-Effekt*-Klasse verschoben, da *TankManager* auch die Klasse mit dem höchsten CBO-Wert nach dem ersten Refactoring ist. Es stellt sich heraus, dass obwohl der Durchschnittswert der CBO-Metrik nach dem zweiten Refactoring erhöht wird, kann der Wert der gezielte Klasse *TankManager* um 1 reduziert werden (vgl. Tabelle 6.2). Wichtig dazu ist, dass der nach dem ersten Refactoring verbesserte RFC-Wert der *TankManager*-Klasse nach Anwendung des zweiten Refactorings nicht verschlechtert wird. Vielmehr wird dieser Wert durch die Auslagerung der 6 Methoden weiter verbessert (von 45 auf 34).

Tabelle 6.2: Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für RFC- und CBO-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für RFC-Metrik	Die Metriken nach dem Refactoring für CBO-Metrik
Number of classes: 19	Number of classes: 20	Number of classes: 20
Max. RFC Value: 60 from Maler - TankManager: 50 Avarage RFC Value: 13.47368	Max. RFC Value: 60 from Maler - TankManager: 45 Avarage RFC Value: 13.05	Max. RFC Value: 60 from Maler - TankManager: 34 Avarage RFC Value: 13.4
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.82676	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.82836
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 10 from TankManager Avarage CBO Value: 2.55	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.6
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.3	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.1
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 722 from Maler Avarage LOC Value: 134.0	Max. LOC Value: 722 from Maler Avarage LOC Value: 134.2

Automatisierte Refactorings für die DBC-Metrik

Da die Klassen mit dem DBC-Wert von 1 *MIDlet* und *GameManager* eine leere Klasse bzw. ein Interface sind, wird die *TankManager*-Klasse wiederum als Quellklasse der Refactorings genommen. Obwohl die beiden EXTRACT CLASS- und MOVE METHODS AND FIELDS-Refactoring anwendbar sind, können sie den DBC-Wert der Quellklasse nicht deutlich verbessern. Darüber hinaus wird der durchschnittliche Wert der DBC-Metrik des gesamten Quellcodes in beiden Fällen etwas verschlechtert. Genauer betrachtet bekommt man durch das MOVE METHODS AND FIELDS-Refactoring einen besseren Durchschnittswert als bei dem EXTRACT CLASS-Refactoring. Deshalb wird der generierte Quellcode nach dem MOVE METHODS AND FIELDS-Refactoring, wobei die Felder *screenWidth, screenHeight, koernigkeit2* und die Methode *init__wrappee__fuer_PC* in die *Missile*-Klasse verschoben werden, als Ausgangspunkt für die Ermittlung der Änderungen anderer Metriken genommen. Dies stellt die Tabelle 6.3 genauer dar.

Während das verwendete Refactoring keine großen Einflüsse auf die CC- und Size-Metrik hat, mindert es die Qualität des Quellcodes hinsichtlich der CBO- und RFC-Metrik. Durch die Auslagerung der *init__wrappee__fuer_PC*-Methode in der Zielklasse entstehen neue Kopplungen zwischen einigen Klassen, die vorher nicht gekoppelt sind. Das Refactoring reduziert zwar den RFC-Wert der Quellklasse aber der Durchschnittswert wird erhöht da die Zielklasse *Missile* nun zugleich mehrere lokalen und externen Methoden besitzen. Basierend auf dieser Beobachtung sollte die RFC-Metrik als nächste

Tabelle 6.3: Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für DBC-Metrik

Originale Werte der Metriken	Die Metriken nach dem MOVE METHODS AND FIELDS-Refactoring für DBC-Metrik
Number of classes: 19	Number of classes: 19
Max. RFC Value: 60 from Maler Avarage RFC Value: 13.47368	Max. RFC Value: 60 from Maler Avarage RFC Value: 13.57894
Worst Cohesion Value: 1.0 from MIDlet - TankManager: 0.94768 - Missile: 0.88676 Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet - TankManager: 0.94122 - Missile: 0.92891 Avarage Cohesion Value: 0.83359
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.68421
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 722 from Maler Avarage LOC Value: 140.73684

Zielmetrik weiterbearbeitet werden. Es wird an dieser Stelle versucht, den RFC-Wert der betroffenen *Missile*-Klasse zu reduzieren. Dabei wird ein REPLACE METHOD WITH METHOD OBJECT-Refactoring auf die *komponenteMalen__wrappee__PC*-Methode angewendet. Dieses Refactoring dekrementiert den RFC-Wert der *Missile*-Klasse von 33 auf 30 und verbessert auch den durchschnittlichen Wert des ganzen Quellcodes. Hinsichtlich der anderen Metriken fasst die Tabelle 6.4 deren Messergebnisse zusammen. Man kann feststellen, dass nach dem zweiten Refactoring während der DBC-Wert der *TankManager*-Klasse wie nach dem ersten Refactoring erhalten bleibt, wird durch das Hinzufügen der neuen kohäsiven Klasse der DBC-Durchschnittswert des Programms verbessert.

Tabelle 6.4: Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für DBC- und RFC-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für DBC-Metrik	Die Metriken nach dem Refactoring für RFC-Metrik
Number of classes: 19	Number of classes: 19	Number of classes: 20
Max. RFC Value: 60 from Maler Avarage RFC Value: 13.47368	Max. RFC Value: 60 from Maler - Missile: 33 Avarage RFC Value: 13.57894	Max. RFC Value: 60 from Maler - Missle: 30 Avarage RFC Value: 13.5
Worst Cohesion Value: 1.0 from MIDlet - TankManager: 0.94768 - Missile: 0.88676 Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet - TankManager: 0.94122 - Missile: 0.92891 Avarage Cohesion Value: 0.83359	Worst Cohesion Value: 1.0 from MIDlet - TankManager: 0.94122 - Missile: 0.92980 Avarage Cohesion Value: 0.82925
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.68421	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.9
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.2
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 722 from Maler Avarage LOC Value: 140.73684	Max. LOC Value: 722 from Maler Avarage LOC Value: 134.3

Automatisierte Refactorings für die CBO-Metrik

Für die *TankManager*-Klasse, welche mit 9 anderen Klassen gekoppelt ist, kann das Plug-in Kandidaten für das EXTRACT CLASS- und MOVE METHODS AND FIELDS-Refactoring identifizieren. Beide Refactorings können den Wert der Quellklasse dekrementieren, erhöhen aber zugleich aufgrund der neu entstanden Kopplungen den

Durchschnittswert des ganzen Programms. Das Resultat des MOVE METHODS AND FIELDS-Refactorings, wobei eine Gruppe von 7 Methoden in die *Wall*-Klasse extrahiert wird, wird für andere Qualitätsaspekte des Programms gemessen (vgl. Tabelle 6.5).

Tabelle 6.5: Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für CBO-Metrik

Originale Werte der Metriken	Die Metriken nach dem MOVE METHODS AND FIELDS-Refactoring für CBO-Metrik
Number of classes: 19	Number of classes: 19
Max. RFC Value: 60 from Maler - TankManager: 50 - Wall: 10 Avarage RFC Value: 13.47368	Max. RFC Value: 60 from Maler - TankManager: 32 - Wall: 37 Avarage RFC Value: 13.94736
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83559
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 8 from TankManager Avarage CBO Value: 2.63157
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.31578
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 722 from Maler Avarage LOC Value: 140.73684

Allgemein betrachtet ändern sich die Metriken nach dem Refactoring nicht deutlich: es gibt kleine Verbesserung bei der CC-Metrik und kleine Verschlechterungen bei anderen Metriken (RFC, DBC und LOC). Bemerkenswert ist die Änderung an den betroffenen Klassen. Durch Verschieben der entsprechenden Methoden wird der RFC-Wert der *TankManager*-Klasse von 50 auf 32 reduziert aber zugleich der Wert der Zielklasse *Wall* von 10 auf 37 erhöht. Das gleiche Verhalten kann auch bei anderen Metriken beobachtet werden.

Als Nächstes wird für die sukzessive Anwendung der Refactorings die RFC-Metrik gezielt unterstützt werden. Für diese Metrik wird die *Wall*-Klasse als Kandidat-Klasse betrachtet. Ein REPLACE METHOD WITH METHOD OBJECT-Refactoring extrahiert die Methode *malenkontrolle__wrappee__fuer_PC*, die 8 eigenen Elementen in Zugriff-Menge besitzt, in eine neue Klasse und ersetzt die Originalmethode in der Quellklasse durch ein Methodenobjekt. Dadurch werden beide Werte der *Wall*-Klasse und des Programms in Bezug auf die RFC-Metrik verbessert. Weiterhin bleibt der CBO-Wert der *TankManager*-Klasse nach dem sukzessiven Refactoring unverändert (vgl. Tabelle 6.6).

Automatisierte Refactorings für die CC-Metrik

Es gibt zwar mehrere Methoden, die refactorisiert werden müssen weil sie einen CC-Wert über 14 haben. Zum Testen des Plug-ins kann aber darauf eingeschränkt, eine Methode mittels des EXTRACT METHOD-Refactorings zu vereinfachen. Dabei werden eine Reihenfolge der *if*-Anweisungen von Position 5 bis Position 27 auf der Block-Ebene 1.2.1 der Methode *drawImage* der Klasse *Maler* in eine neue Methode, namens *handleExplode*, extrahiert. Dieses Refactoring reduziert den CC-Wert der *drawImage*-Methode von 104 auf 81 (und somit auch den Wert der *Maler*-Klasse) und bringt die neue *handleExplode*-Methode mit einem CC-Wert von 24 hervor. Der durchschnittliche CC-Wert des ganzen Programm wird dadurch auch verbessert. Die Tabelle 6.7 präsentiert die Messergebnisse anderer Metriken nachdem die gewählten Anweisungen extrahiert wurden.

Die neue Methode führt zum Inkrement des RFC-Werts der Quellklasse (neue lokale Methode). Da die Transformation nur innerhalb der *Maler*-Klasse erfolgt, entsteht

Tabelle 6.6: Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für CBO- und RFC-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für CBO-Metrik	Die Metriken nach dem Refactoring für RFC-Metrik
Number of classes: 19	Number of classes: 19	Number of classes: 20
Max. RFC Value: 60 from Maler - TankManager: 50 - Wall: 10 Avarage RFC Value: 13.47368	Max. RFC Value: 60 from Maler - TankManager: 32 - Wall: 37 Avarage RFC Value: 13.94736	Max. RFC Value: 60 from Maler - TankManager: 32 - Wall: 31 Avarage RFC Value: 13.45
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83559	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83103
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 8 from TankManager Avarage CBO Value: 2.63157	Max. CBO Value: 8 from TankManager Avarage CBO Value: 2.7
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.31578	Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.1
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 722 from Maler Avarage LOC Value: 140.73684	Max. LOC Value: 722 from Maler Avarage LOC Value: 134.3

Tabelle 6.7: Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für CC-Metrik

Originale Werte der Metriken	Die Metriken nach dem EXTRACT METHOD-Refactoring für CC-Metrik
Number of classes: 19	Number of classes: 19
Max. RFC Value: 60 from Maler Avarage RFC Value: 13.47368	Max. RFC Value: 61 from Maler Avarage RFC Value: 13.52631
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83168
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368
Max. Cyclomatic Complexity Value: 104 from Maler - Maler.drawImage(): 104 Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 81 from Maler - Maler.drawImage(): 81 Avarage Cyclomatic Complexity Value: 14.36842
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 734 from Maler Avarage LOC Value: 141.15789

dadurch keine neue Kopplung zwischen den vorliegenden Klassen. So bleiben die CBO-Werte der Klassen gleich wie vor dem Refactoring. Betrachtet man die anderen Metriken (DBC und LOC) kann man folgende Änderungen erkennen: durch das Extrahieren verkleinert das verwendete Refactoring die *drawImage*-Methode aber erhöht die gesamte Anzahl der Codezeilen der Quellklasse. Die erstellte Methode ist auch mit anderen Methoden und Feldern der *Maler*-Klasse nicht sehr kohäsiv deshalb steigt der DBC-Wert dieser Klasse auch leicht auf. Trotzdem sind diese negativen Auswirkungen des Refactorings verhältnismäßig klein. Daher wird versucht, die CC-Werte weiter zu verbessern, die immer noch über dem Grenzwert von 14 liegen. Dafür wird weiterhin das EXTRACT METHOD-Refactoring als das sukzessive Refactoring auf die *drawImage*-Methode der Klasse *Maler* benutzt. Die Anweisungen von Position 20 bis 79 im Block 1.2.1 dieser Methode werden in eine Methode, namens *handleMovement*, extrahiert. Mit dem zweiten EXTRACT CLASS-Refactoring wird die Komplexität der *drawImage*-Methode nun auf 21 gesetzt und die durchschnittliche Komplexität des Programms weiter verbessert. Die Änderungen der anderen Metriken sind ähnlich wie beim ersten Refactoring und werden durch die Tabelle 6.8 dargestellt.

Tabelle 6.8: Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für CC-Metrik

Originale Werte der Metriken	Die Metriken nach dem ersten Refactoring für CC-Metrik	Die Metriken nach dem zweiten Refactoring für CC-Metrik
Number of classes: 19	Number of classes: 19	Number of classes: 19
Max. RFC Value: 60 from Maler Avarage RFC Value: 13.47368	Max. RFC Value: 61 from Maler Avarage RFC Value: 13.52631	Max. RFC Value: 62 from Maler Avarage RFC Value: 13.57894
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83168	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83175
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368
Max. Cyclomatic Complexity Value: 104 from Maler - Maler.drawImage(): 104 Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 81 from Maler - Maler.drawImage(): 81 Avarage Cyclomatic Complexity Value: 14.36842	Max. Cyclomatic Complexity Value: 61 from Maler - Maler.drawImage(): 21 - Maler.handleMovement(): 61 Avarage Cyclomatic Complexity Value: 13.31578
Max. LOC Value: 722 from Maler Avarage LOC Value: 140.52631	Max. LOC Value: 734 from Maler Avarage LOC Value: 141.15789	Max. LOC Value: 746 from Maler Avarage LOC Value: 141.78947

Automatisierte Refactorings für die LOC-Metrik

Die *Maler*-Klasse mit 722 Codezeilen wird durch ein REPLACE METHOD WITH METHOD OBJECT-Refactoring verkürzt. Dabei soll die *drawImage*-Methode (mit 317 Codezeilen) in eine neue Klasse, namens *ImageDrawer*, gelegt werden. Nach dem Refactoring enthält die Quellklasse nur noch 408 Codezeilen und die neue Klasse 333 Codezeilen. Der Durchschnittswert wird demnach auch verbessert. Für die restlichen Metriken werden folgende Ergebnisse in Tabelle 6.9 festgestellt.

Tabelle 6.9: Ergebnisse der Metriken der TankWar-Variante vor und nach dem Refactoring für LOC-Metrik

Originale Werte der Metriken	Die Metriken nach dem REPLACE METHOD WITH METHOD OBJECT-Refactoring für LOC-Metrik
Number of classes: 19	Number of classes: 20
Max. RFC Value: 60 from Maler Avarage RFC Value: 13.47368	Max. RFC Value: 62 from Maler Avarage RFC Value: 13.05
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.82675
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.45
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from ImageDrawer Avarage Cyclomatic Complexity Value: 15.45
Max. LOC Value: 722 from Maler - Maler: 722 Avarage LOC Value: 140.52631	Max. LOC Value: 518 from Tank - Maler: 408 Avarage LOC Value: 134.45

Die Einführung der neuen Klasse verursacht zusätzliche externe Methodenaufrufe bzw. Klassenzugriffe in der Quellklasse und erhöht somit ihren RFC-Wert und CBO-Wert. Da aber die *ImageDrawer*-Klasse einen guten RFC-Wert bzw. CBO-Wert hat werden die durchschnittlichen Werte reduziert. Die gleiche Änderung kann auch bei der DBC-Metrik beobachtet werden. Im Gegensatz dazu steigt der Durchschnittswert der CC-Metrik obwohl dieser Wert der *Maler*-Klasse dekrementiert wird. Der Grund dafür ist das, dass die extrahierte *drawImage*-Methode auch die Methode mit dem höchsten CC-Wert des ganzen Quellcodes ist. Demnach sollte für den nächsten Schritt die CC-Metrik der neu angelegten Klasse bzw. Methode unterstützt werden indem die kom-

plizierte Methode *compute* der *ImageDrawer*-Klasse vereinfacht wird. Dazu werden die Anweisungen von Position 5 bis 27 im Block 1.2.1 dieser Methode in eine neue Methode angelegt. Wie die Tabelle 6.10 darstellt, wird der CC-Wert der Kandidat-Methode (und dementsprechend der *ImageDrawer*-Klasse) von 104 auf 81 reduziert. Die anderen Metriken haben sich dadurch kaum geändert.

Tabelle 6.10: Ergebnisse der Metriken der TankWar-Variante vor und nach den sukzessiven Refactorings für LOC- und CC-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für LOC-Metrik	Die Metriken nach dem Refactoring für CC-Metrik
Number of classes: 19	Number of classes: 20	Number of classes: 20
Max. RFC Value: 60 from Maler Avarage RFC Value: 13.47368	Max. RFC Value: 62 from Maler Avarage RFC Value: 13.05	Max. RFC Value: 62 from Maler Avarage RFC Value: 13.1
Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.83160	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.82675	Worst Cohesion Value: 1.0 from MIDlet Avarage Cohesion Value: 0.82741
Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.47368	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.45	Max. CBO Value: 9 from TankManager Avarage CBO Value: 2.45
Max. Cyclomatic Complexity Value: 104 from Maler Avarage Cyclomatic Complexity Value: 15.57894	Max. Cyclomatic Complexity Value: 104 from ImageDrawer Avarage Cyclomatic Complexity Value: 15.45	Max. Cyclomatic Complexity Value: 81 from ImageDrawer Avarage Cyclomatic Complexity Value: 14.3
Max. LOC Value: 722 from Maler - Maler: 722 Avarage LOC Value: 140.52631	Max. LOC Value: 518 from Tank - Maler: 408 Avarage LOC Value: 134.45	Max. LOC Value: 518 from Tank - Maler: 408 Avarage LOC Value: 134.6

6.3.2 Evaluierung mit der VioletUMLEditor-SPL

Die VioletUMLEditor-SPL implementiert ein Programm zur Erstellung und Bearbeitung von unterschiedlichen UML-Diagrammen (u.a. Klassendiagram, Anwendungsfalldiagram etc.). Diese SPL hat eine Anzahl von 89 Features und kann somit als eine mittlere SPL bezeichnet werden. Die Testvariante wird mit allen Features generiert, wobei 67 Klassen erzeugt und in zwei Pakete eingepackt werden. Das Listing 6.2 stellt die Messergebnisse der 5 Metriken vor den Refactorings dar.

Listing 6.2: Messergbnisse der Metriken für die VioletUMLEditor-Variante

```
1  Max. RFC Value: 130 from EditorFrame
2  Avarage RFC Value: 14.656716417910447
3  Number of classes: 67
4
5  Worst Cohesion Value: 1.0 from ReturnEdge
6  Avarage Cohesion Value: 0.871678731060132
7  Number of classes: 67
8
9  Max. CBO Value: 12 from EditorFrame
10 Avarage CBO Value: 2.417910447761194
11 Number of classes: 67
12
13 Max. Cyclomatic Complexity Value: 48 from GraphPanel
14 Avarage Cyclomatic Complexity Value: 6.029850746268656
15 Number of classes: 67
16
17 Max. LOC Value: 676 from EditorFrame
18 Avarage LOC Value: 70.43283582089552
19 Number of classes: 67
```

Automatisierte Refactorings für die RFC-Metrik

Die *EditorFrame*-Klasse mit dem RFC-Wert von 130 sollte refactorisiert werden. Für das REPLACE METHOD WITH METHOD OBJECT-Refactoring wird kein guter Kandidat gefunden. Es gibt zwar den Konstruktor *EditorFrame* mit 10 eigenen Elementen in seiner Zugriff-Menge aber das REPLACE METHOD WITH METHOD OBJECT-Refactoring ist nicht für einen Konstruktor anwendbar. Das beste Ergebnis liefert das EXTRACT CLASS-Refactoring, wobei 9 Methoden in eine neue Klasse, namens *MenuUtility*, ausgelagert werden. Der RFC-Wert der Quellklasse wird auf 118 reduziert, der durchschnittliche Wert des Programms steigt zwar auf aber die Änderung ist sehr gering. Die Änderungen der anderen Metriken werden in Tabelle 6.11 zusammengefasst.

Tabelle 6.11: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für RFC-Metrik

Originale Werte der Metriken	Die Metriken nach dem EXTRACT CLASS-Refactoring für RFC-Metrik
Number of classes: 67	Number of classes: 68
Max. RFC Value: 130 from EditorFrame Avarage RFC Value: 14.65671	Max. RFC Value: 118 from EditorFrame Avarage RFC Value: 14.70588
Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87184
Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791	Max. CBO Value: 10 from EditorFrame - MenuUtility: 7 Avarage CBO Value: 2.45588
Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.07352
Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.43283	Max. LOC Value: 518 from EditorFrame Avarage LOC Value: 69.48529

Die LOC-Metrik lässt sich durch die Restrukturierung auch verbessern da die extrahierte Methode die Anzahl der Codezeilen der Quellklasse reduziert. Die DBC- und CC-Metrik haben sich kaum geändert. Obwohl der CBO-Wert der *EditorFrame*-Klasse verbessert wird (von 12 auf 10), wird durch das Hinzufügen der neuen Klasse (mit einem CBO-Wert von 7) der CBO-Durchschnittswert des Programms erhöht. Es sollte dann für die Analyser der sukzessive Anwendung von Refactorings versucht, die CBO-Metrik dieser Neuklasse zu unterstützen. Basierend auf die Vorschläge des Plug-ins werden 6 aus 9 Methoden der *MenuUtility*-Klasse mit einem MOVE METHODS AND FIELDS-Refactoring in die *GraphPanel*-Klasse umgesiedelt. Das verwendete Refactoring kann zwar den CBO-Wert der *MenuUtility*-Klasse von 7 auf 6 reduzieren, erhöht aber durch neue Kopplungen den Wert der Zielklasse und des ganzen Programms. Obwohl der verbesserte RFC-Wert der *EditorFrame*-Klasse nicht geändert wird, steigen der RFC-Wert der *GraphPanel*-Klasse und der durchschnittliche RFC-Wert des Programms auch (vgl. Tabelle 6.12).

Automatisierte Refactorings für die DBC-Metrik

Da die Klassen mit dem DBC-Wert 1.0 entweder Interfaces sind oder keine tatsächliche Funktionalitäten implementieren wird die *EditorFrame*-Klasse als Kandidat-Klasse eingenommen. Ein MOVE METHODS AND FIELDS-Refactoring verschiebt 12 Methoden der *EditorFrame*-Klasse in die *GraphFrame*-Klasse und kann zwar den DBC-Wert der Quellklasse unterstützen. Allerdings ist die Verbesserung gering und außerdem werden der DBC-Wert der Zielklasse und der Durchschnittswert dadurch verschlechtert (obwohl

Tabelle 6.12: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach den sukzessiven Refactorings für RFC- und CBO-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für RFC-Metrik	Die Metriken nach dem Refactoring für CBO-Metrik
Number of classes: 67	Number of classes: 68	Number of classes: 68
Max. RFC Value: 130 from EditorFrame Avarage RFC Value: 14.65671	Max. RFC Value: 118 from EditorFrame Avarage RFC Value: 14.70588	Max. RFC Value: 118 from EditorFrame Avarage RFC Value: 14.76470
Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87184	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87281
Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791	Max. CBO Value: 10 from EditorFrame - MenuUtility: 7 Avarage CBO Value: 2.45588	Max. CBO Value: 10 from EditorFrame - MenuUtility: 6 Avarage CBO Value: 2.47058
Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.07352	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 5.98529
Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.43283	Max. LOC Value: 518 from EditorFrame Avarage LOC Value: 69.48529	Max. LOC Value: 518 from EditorFrame Avarage LOC Value: 69.54411

auch sehr gering). Das Verschieben der Methoden verursacht den gleichen Effekt in der Quellklasse und der Zielklasse bei anderen Metriken. Die Ergebnisse dieser Metriken werden in folgender Tabelle 6.13 präsentiert.

Tabelle 6.13: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für DBC-Metrik

Originale Werte der Metriken	Die Metriken nach dem MOVE METHODS AND FIELDS-Refactoring für DBC-Metrik
Number of classes: 67	Number of classes: 67
Max. RFC Value: 130 from EditorFrame Avarage RFC Value: 14.65671	Max. RFC Value: 103 from EditorFrame Avarage RFC Value: 14.77611
Worst Cohesion Value: 1.0 from ReturnEdge - EditorFrame: 0.97911 - GraphFrame: 0.90517 Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge - EditorFrame: 0.97546 - GraphFrame: 0.96978 Avarage Cohesion Value: 0.87237
Max. CBO Value: 12 from EditorFrame - GraphFrame: 2 Avarage CBO Value: 2.41791	Max. CBO Value: 10 from EditorFrame - GraphFrame: 9 Avarage CBO Value: 2.50746
Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.07462
Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.43283	Max. LOC Value: 483 from EditorFrame Avarage LOC Value: 70.49253

Die CBO-Metrik bekommt die negativste Auswirkung indem der CBO-Wert der *GraphFrame*-Klasse von 2 auf 9 erhöht wird. Daher soll man im nächsten Schritt diese Metrik verbessern. Zu dieser Aufgabe werden 7 Methoden der *GraphFrame*-Klasse in die *UMLEditor*-Klasse umgesiedelt. Es stellt sich heraus, dass der CBO-Wert der Quellklasse verbessert werden kann. Dennoch inkrementiert das verwendete Refactoring auch den Wert der Zielklasse und den Durchschnittswert des ganzen Quellcodes. Der DBC-Wert der verbesserten Klasse *GraphFrame* bleibt gleich aber das zweite Refactoring verursacht Inkohärenz in den beiden betroffenen Klassen. Die Tabelle 6.14 stellt die Ergebnisse der sukzessiven Anwendung der zwei Refactorings dar.

Tabelle 6.14: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach den sukzessiven Refactorings für DBC- und CBO-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für DBC-Metrik	Die Metriken nach dem Refactoring für CBO-Metrik
Number of classes: 67	Number of classes: 67	Number of classes: 67
Max. RFC Value: 130 from EditorFrame Avarage RFC Value: 14.65671	Max. RFC Value: 103 from EditorFrame Avarage RFC Value: 14.77611	Max. RFC Value: 103 from EditorFrame Avarage RFC Value: 14.94029
Worst Cohesion Value: 1.0 from ReturnEdge - EditorFrame: 0.97911 - GraphFrame: 0.90517 Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge - EditorFrame: 0.97546 - GraphFrame: 0.96978 Avarage Cohesion Value: 0.87237	Worst Cohesion Value: 1.0 from ReturnEdge - EditorFrame: 0.97545 - GraphFrame: 0.96922 Avarage Cohesion Value: 0.87398
Max. CBO Value: 12 from EditorFrame - GraphFrame: 2 Avarage CBO Value: 2.41791	Max. CBO Value: 10 from EditorFrame - GraphFrame: 9 Avarage CBO Value: 2.50746	- GraphFrame: 8 Avarage CBO Value: 2.55223
Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.07462	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.11940
Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.43283	Max. LOC Value: 483 from EditorFrame Avarage LOC Value: 70.49253	Max. LOC Value: 483 from EditorFrame Avarage LOC Value: 70.50746

Automatisierte Refactorings für die CBO-Metrik

Für die Kandidat-Klasse *EditorFrame* kann das Plug-in keine guten Refactoring-Möglichkeiten herausfinden. Es liegt daran, dass die Methoden dieser Klasse wenige eigene Zugriffe in ihren Zugriff-Mengen besitzen. Darüber hinaus teilen sich viele Methoden Zugriffe auf gleiche Klassen mit Methoden von anonymen Klassen. Solche Methoden können nicht mit den zur Verfügung gestellten Refactorings behandelt werden. Es sei an dieser Stelle aber darauf hingewiesen, dass mit 67 Klassen könnten die Werte der CBO-Metrik dieses Programms (mit einem Durchschnitt von 2.41791 und dem höchsten Wert von 12) im Vergleich zu den CBO-Werten des TankWar-Beispiels schon akzeptierbar sein.

Automatisierte Refactorings für die CC-Metrik

Für diese Metrik wird der Konstruktor der *GraphPanel*-Klasse, dessen CC-Wert 48 ist, durch ein EXTRACT METHOD-Refactoring vereinfacht. Dabei werden die *if*-Anweisung in Position 6 des Blocks 1.1 und ihre untere Struktur in eine neue Methode angelegt. Das verwendete Refactoring dekrementiert die Komplexität des Konstruktors auf 19. Die erstellte Methode nimmt die Komplexität der *if*-Anweisung herein und hat nun einen CC-Wert 30. Dadurch wird auch der Durchschnittswert des ganzen Quellcodes verbessert. In Bezug auf die Änderungen anderer Metriken stellt die Tabelle 6.15 die Messergebnisse dieser Metriken nach dem Refactoring dar. Die neue Methode hat zwar den RFC- bzw. DBC-Wert der *GraphPanel*-Klasse und auch die Durchschnittswerte erhöht. Dennoch sind diese negativen Auswirkungen sehr gering. Die CBO-Werte bleiben gleich da die Umstrukturierung nur lokal in der Quellklasse erfolgt. Obwohl sich die Anzahl der Codezeilen der *GraphPanel* kaum geändert hat, wird der LOC-Wert des Konstruktors verbessert denn das Refactoring ihn verkleinert.

Für die Analyse der sukzessiven Anwendung von Refactorings kann die RFC-Metrik als nächste Zielmetrik gewählt werden. Es wird versucht, den RFC-Wert der *GraphPanel*-Klasse zu reduzieren. Dafür werden 3 Methoden durch ein MOVE METHODS AND

Tabelle 6.15: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für CC-Metrik

Originale Werte der Metriken	Die Metriken nach dem EXTRACT METHOD-Refactoring für CC-Metrik
Number of classes: 67	Number of classes: 67
Max. RFC Value: 130 from EditorFrame - GraphPanel: 53 Avarage RFC Value: 14.65671	Max. RFC Value: 130 from EditorFrame - GraphPanel: 54 Avarage RFC Value: 14.77611
Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87170
Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791	Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791
Max. Cyclomatic Complexity Value: 48 from GraphPanel - GraphPanel.GraphPanel(): 48 Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 43 from BentStyle - GraphPanel.GraphPanel(): 19 Avarage Cyclomatic Complexity Value: 5.76119
Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.43283	Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.47761

FIELDS-Refactoring in die *Graph*-Klasse verschoben. Nach dem Refactoring wird der RFC-Wert der *GraphPanel*-Klasse von 54 auf 47 reduziert. Dabei wird der nach dem ersten Refactoring verbesserter CC-Wert dieser Klasse nicht geändert. Für die restlichen Metriken sind keine große Änderungen festzustellen, ihre Messergebnisse werden durch die Tabelle 6.16 vorgestellt.

Tabelle 6.16: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach den sukzessiven Refactorings für CC- und RFC-Metrik

Originale Werte der Metriken	Die Metriken nach dem Refactoring für CC-Metrik	Die Metriken nach dem Refactoring für RFC-Metrik
Number of classes: 67	Number of classes: 67	Number of classes: 67
Max. RFC Value: 130 from EditorFrame - GraphPanel: 53 Avarage RFC Value: 14.65671	Max. RFC Value: 130 from EditorFrame - GraphPanel: 54 Avarage RFC Value: 14.77611	Max. RFC Value: 130 from EditorFrame - GraphPanel: 47 Avarage RFC Value: 14.70149
Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87170	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87193
Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791	Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791	Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.44776
Max. Cyclomatic Complexity Value: 48 from GraphPanel - GraphPanel.GraphPanel(): 48 Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 43 from BentStyle - GraphPanel.GraphPanel(): 19 Avarage Cyclomatic Complexity Value: 5.76119	Max. Cyclomatic Complexity Value: 43 from BentStyle - GraphPanel.GraphPanel(): 19 Avarage Cyclomatic Complexity Value: 5.85074
Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.43283	Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.47761	Max. LOC Value: 676 from EditorFrame Avarage LOC Value: 70.47761

Automatisierte Refactorings für die LOC-Metrik

Der Konstruktor der *EditorFrame*-Klasse mit 306 Codezeilen wird durch ein EXTRACT METHOD-Refactoring verkürzt. Durch Analyse der Struktur dieses Konstruktors in der *Size Editor*-View sollen die Anweisungen von Position 33 bis Position 42 des Blocks 1 in eine neue Methode extrahiert werden, da sie zur Gestaltung eines Menüs in der Benutzeroberfläche dienen. Das Refactoring kann zwar den LOC-Wert der Quellklasse und den durchschnittlichen Wert nicht verbessern aber die gezielte Methode wird auf 268 Codezeilen reduziert. Für die restlichen Metriken werden die Messergebnisse in

Tabelle 6.17 zusammengefasst. Die Erstellung der neuen Methode hat die RFC- und DBC-Metrik geändert aber die negativen Einflüsse können nicht berücksichtigt werden. Da dieses Refactoring dem bei der CC-Metrik verwendeten Refactoring ähnlich ist muss die Weiterbearbeitung der nächsten Zielmetrik an dieser Stelle nicht betrachtet werden.

Tabelle 6.17: Ergebnisse der Metriken der VioletUMLEditor-Variante vor und nach dem Refactoring für LOC-Metrik

Originale Werte der Metriken	Die Metriken nach dem EXTRACT METHOD-Refactoring für LOC-Metrik
Number of classes: 67	Number of classes: 67
Max. RFC Value: 130 from EditorFrame Avarage RFC Value: 14.65671	Max. RFC Value: 131 from EditorFrame Avarage RFC Value: 14.67164
Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87167	Worst Cohesion Value: 1.0 from ReturnEdge Avarage Cohesion Value: 0.87170
Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791	Max. CBO Value: 12 from EditorFrame Avarage CBO Value: 2.41791
Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 6.02985	Max. Cyclomatic Complexity Value: 48 from GraphPanel Avarage Cyclomatic Complexity Value: 5.97014
Max. LOC Value: 676 from EditorFrame - EditorFrame.EditorFrame(): 306 Avarage LOC Value: 70.43283	Max. LOC Value: 679 from EditorFrame EditorFrame.EditorFrame(): 268 Avarage LOC Value: 70.47761

6.4 Zusammenfassung

In diesem Kapitel wurde das Implementierungskonzept des prototypischen Refactoring Plug-ins vorgestellt. Mit diesem Tool kann der Prozess zur Optimierung unterschiedlicher nicht-funktionalen Eigenschaften in Bezug auf die Wartbarkeit des Produkts einer SPL automatisiert werden. Anschließend wurde das Tool mit Hilfe von generierten Programmen unterschiedlicher SPLs evaluiert. Die Ergebnisse der Evaluierung fasst die Tabelle 6.4 zusammen, wobei für jede Metrik die Anwendung der sukzessiven Refactorings dargestellt wird. Das „⋆“-Zeichen steht dafür, dass das verwendete Refactoring sowohl den Wert der Kandidat-Klasse als auch den durchschnittlichen Wert des ganzen Programms bzgl. der entsprechenden Metrik verbessern kann. Das „+“-Zeichen bedeutet, dass nur der Wert der gezielte Klasse unterstützt werden kann und der Durchschnittswert verschlechtert wird. Die Situation wenn für eine Metrik keine Verbesserung durch Refactorings erreicht werden kann, wird durch das „X“-Zeichen bezeichnet.

	Metriken				
Anwendungsbeispiel	RFC	DBC	CBO	CC	LOC
TankWar-Variante	⋆ | +	+ | ⋆	+ | ⋆	⋆ | ⋆	⋆ | ⋆
VioletUMLEditor-Variante	+ | +	+ | +	X	⋆ | +	⋆ | +

Tabelle 6.18: Ergebnisse der Evaluierung des Refactoring Plug-ins

Aus den Ergebnissen lassen sich folgende Eigenschaften der Anwendung des Refactoring Plug-ins feststellen:

1. Die Entdeckung der Refactoring-Möglichkeiten können durch die heuristischen Strategien unterstützt werden. Dennoch für eine optimale Lösung sind in einigen Fällen Kenntnisse über den Quellcode erforderlich.

2. Das Refactoring Plug-in kann in den meisten Fällen eine einzelne Metrik verbessern. Dennoch sind für unterschiedliche Metriken die Effekte der Refactorings auch unterschiedlich: einige Metriken (DBC, CBO) sind schwer zu optimieren während die anderen Metriken (RFC, CC und LOC) einfach und deutlich verbessert werden können. Die DBC- bzw. CBO-Metrik bekommt positive Änderungen meistens nur in besonderen Fällen wenn z.B. Methoden vorkommen, die keine Relationen zu anderen Feldern und Methoden der enthaltenden Klasse haben. Die Schwierigkeit der Optimierung der CBO-Metrik liegt an ihrer Definition, wobei allein die Benutzung einer externen Klasse zählt. Demnach ist es mit den implementierten Refactorings unmöglich, eine Kopplung zu einer Klasse zu entfernen, wenn sie von einer Methode einer anonymen Klasse zugegriffen wird. Für die DBC-Metrik könnte die Entdeckung der Refactoring-Parameter durch die heuristische Strategie des Plug-ins noch nicht optimal sein (insbesondere in großen Programmen).

3. Die Optimierung der einzelnen Metrik kann zur negativen Änderung anderer Metriken führen insbesondere zwischen den 3 Metriken RFC, DBC und CBO da sie auf die Benutzen-Beziehungen zwischen Elementen der Klassen basieren und ihre entsprechenden Refactorings diese Beziehungen ändern können.

4. Die Optimierung mehrerer Metriken durch sukzessive Anwendung von Refactorings ist möglich. Es wurde aber nicht festgestellt, ob die Reihenfolge der Refactorings die Endergebnisse bestimmen kann.

5. Eine Metrik kann nicht direkt durch ihre entsprechenden Refactorings unendlich verbessert werden. Die Grenzwerte einer Metrik, über den sich die Metrik nicht mehr optimieren lässt, variieren aber von Programm zu Programm und müssen von dem Entwickler festgestellt werden. Ob eine Kombination von Refactorings zur Verbesserung solcher Metriken führen könnte, kann durch die Evaluierung nicht bestätigt werden.

Trotz einiger oben genannter Nachteile haben die Ergebnisse der Evaluierung gezeigt, dass das implementierte Refactoring Plug-in zur Optimierung der wartungsbezogenen Metriken einsetzbar ist. Dadurch werden auch zukünftige Aufgaben abgeleitet (z.B. Ermittlung der Rolle der Reihenfolge von Refactorings für die Endergebnisse), die das vorliegende Plug-in weiter verbessern können.

Kapitel 7

Zusammenfassung und Ausblick

Die vorliegende Arbeit beschäftigt sich mit den Problemen der Anwendung von Refactorings zur Optimierung der nicht-funktionalen Eigenschaften der Produkte einer SPL. Unter diesen Eigenschaften konzentriert sich die Arbeit auf einige Eigenschaften, die sich mit der Wartbarkeit eines Programms befassen. Die Anwendung von Refactorings wird bis jetzt von den meistens heutigen IDEs nur beschränkt unterstützt indem die Durchführung eines gewählten Refactorings automatisiert wird. Bevor ein Refactoring angewendet werden kann, müssen aber zuerst Schwachstellen im Quellcode herausgefunden werden, die refaktorisiert werden sollten. Darüber hinaus muss man noch die richtigen Refactorings finden, die die Probleme der Schwachstellen beheben können. Diese zwei Aufgaben sind schwierig und fehleranfällig. Deshalb wurden für diese Arbeit folgende Ziele definiert. Zuerst wurden Software-Metriken festgestellt, die unterschiedliche Aspekte der Wartbarkeit beschreiben können. Anschließend wurden die Eigenschaften der Refactorings, die diese Metriken unterstützen können, anhand eines Anwendungsbeispiels erörtert und zusammengefasst. Basierend auf den gewonnenen Kenntnissen wurde ein Prototyp zur automatisierten Optimierung von nicht-funktionalen Eigenschaften in Bezug auf die Wartbarkeit einer Software implementiert. Diese Ziele wurden wie folgt realisiert.

Im Kapitel 3 sind unterschiedliche Software-Metriken vorgestellt, die sich auf verschiedene Aspekte der Wartbarkeit beziehen. Es gibt Metriken bzw. Metriken-Suiten, die zwar die Wartbarkeit einer Software gut diagnostizieren können aber schwer zu optimieren sind, da die Gründe für schlechte Messergebnisse schwer identifizierbar sind. Darüber hinaus ist festgestellt worden, dass zur Beschreibung der Wartbarkeit ein Metriken-Modell (oder eine Metriken-Suite) benutzt werden soll, weil die Wartbarkeit ein kompliziertes Konzept ist und eine einzige Metrik nicht zur genauen Charakterisierung dieses Konzepts dienen kann. Schließlich wurden 5 Metriken gewählt: die CC-Metrik erfasst die Komplexität, eine wichtige Eigenschaft der Wartbarkeit. Die RFC-, CBO- und DBC-Metrik beschreiben zwei wesentlichen Konzepte der OOP, Kohäsion und Kopplung, welche für die Wartbarkeit eine entscheidende Rolle spielen. Die LOC-Metrik bildet das letzte Element des Metriken-Modells. Sie ist eine einfache Metrik trotzdem in vielen Fällen kann diese Metrik die Intuition von Entwicklern über die Analysierbarkeit und Komplexität einer Software darstellen.

Das Kapitel 5 stellt Refactorings vor, die die Metriken verbessern können. Für die CBO- und RFC-Metrik sind 3 Refactorings: MOVE METHODS AND FIELDS, EX-

TRACT CLASS und REPLACE METHOD WITH METHOD OBJECT verantwortlich. Das EXTRACT CLASS- und MOVE METHODS AND FIELDS-Refactoring können die DBC-Metrik unterstützen. Die CC-Metrik kann durch ein EXTRACT METHOD- oder ein DECOMPOSE CONDITIONAL-Refactoring verbessert werden indem eine Reihenfolgen von komplizierten Anweisungen einer Methode bzw. eine komplizierte Kombination von Konditionen einer *if*-Anweisung in eine neue Methode umgewandelt wird, deren Name die Logik des extrahierten Teils erläutert. Für die LOC-Metrik können zur Optimierung das EXTRACT METHOD- oder das REPLACE METHOD WITH METHOD OBJECT-Refactoring eingesetzt werden. Alle Refactorings wurden aus dem Buch von Fowler [Fow99] entnommen, welches nur die manuelle Durchführung der Refactorings beschreibt. Die Automatisierung eines Refactorings kann aber nicht immer nach der manuellen Weise durchgeführt werden, da die Semantik eines Programms nicht durch Quellcode-Strukturen wie die AST-Struktur vollkommen dargestellt werden kann. Durch einige Änderungen und Beschränkungen kann das Refactoring realisierbar sein, was aber auch dazu führen kann, dass das Refactoring bei Durchführung zusätzliche Komplexität in das Programm einführt.

Zur Analyse der Eigenschaften der implementierten Refactorings wurden die Refactorings zur Verbesserung der Metriken einer Beispiel-Anwendung aus der Graph-SPL angewendet. Dazu wurde die Messung der Metriken auch implementiert. Die Messergebnisse wurden zunächst manuell interpretiert um Refactoring-Möglichkeiten aller für eine Metrik zur Verfügung gestellten Refactorings herauszufinden. Der von einem Refactoring generierte Quellcode wird wieder gemessen um die Einflüsse des entsprechenden Refactorings auf die betroffenen Klassen und das ganze Programm festzustellen. Es stellte sich heraus, dass für die RFC, CBO und DBC-Metrik ein Refactoring sowohl positive als auch negative Auswirkungen haben kann. Das Vorhersagen über die Auswirkung eines Refactorings ist wegen der unterschiedlichen Beziehungen zwischen den Klassen und den Elementen der Klassen (Methoden und Feldern) schwer zu realisieren. Darüber hinaus kann ein Refactoring die Metrik einer gezielten Klasse verbessern und dennoch den Durchschnittswert des ganzen Programms negativ beeinflussen. Daher sollte das Ziel eines Refactorings vor der Durchführung klar definiert werden. Für eine Metrik ist es auch empfohlen, verschiedene Möglichkeiten auszuprobieren, da es mehrere Refactorings geben kann, die die Metrik unterstützen können und ihre Einflüsse schwer vorherzusagen sind.

Die manuelle Interpretierung der Messergebnisse zur Entdeckung von Refactoring-Möglichkeiten ist nicht trivial, mühsam und fehleranfällig. Ein Refactoring Plug-in wurde zur Unterstützung dieses Prozesses in der Eclipse-IDE entwickelt. Dazu werden in diesem Plug-in die Messung von Metriken und die Durchführung der Refactorings eingebettet. Für jede Metrik und jedes unterstützende Refactoring wird eine heuristische Strategie zum Einsatz gebracht, die anhand der Messergebnisse Refactoring-Möglichkeiten für das Refactoring vorschlagen kann. Das Refactoring Tool wurde dann mit Hilfe von generierten Anwendungen verschiedener SPLs evaluiert. Dabei wurden nicht nur die Änderung an der gezielten Metrik sondern auch Änderungen an anderen Metriken des Programms beobachtet. Weiterhin wurde die Möglichkeit in Betrachtung gezogen, Refactorings zur Optimierung verschiedener Metriken sukzessiv anzuwenden. Es stellte sich heraus, dass die DBC- und CBO-Metrik schwer zu optimieren sind während die anderen Metriken (RFC, CC, LOC) einfach und deutlich verbessert werden können. Die Verbesserung der

einzelnen Metrik kann zur negativen Änderung anderer Metriken führen insbesondere zwischen den 3 Metriken RFC, DBC und CBO. Ergänzend sei darauf hingewiesen, dass es Grenzwerte für die Metriken gibt, über den die Metriken nicht mehr direkt durch ihre entsprechenden Refactorings verbessert werden können. Diese Grenzwerte sind aber von Programm zu Programm unterschiedlich und sollen vom Anwender bestimmt werden. Außerdem kann die Optimierung mehrerer Metriken anhand sukzessiver Anwendung von Refactorings durch das Refactoring Plug-in realisiert werden. Es wurde aber nicht festgestellt, ob die Reihenfolge der Refactorings die Endergebnisse beeinflussen kann.

Die Ergebnisse der Evaluierung hat gezeigt, dass das implementierte Refactoring Plug-in zur Optimierung der nicht-funktionalen Eigenschaften von Produkten einer SPLs in Bezug auf ihre Wartbarkeit benutzt werden kann. Zukünftige Arbeiten können sich mit den folgenden Aufgaben beschäftigen. Es sollte versucht werden, unterschiedliche Strategien zur Entdeckung von Refactoring-Möglichkeiten auszuprobieren (z.B. durch Visualisierung der Messergebnisse). Bezüglich der sukzessiven Anwendung von Refactorings kann man die Rolle der Reihenfolge der Refactorings ermitteln. Das implementierte Refactoring Plug-in kann für weitere Metriken erweitert werden, die sich nicht nur auf die Wartbarkeit beziehen müssen sondern andere Aspekte des Quellcodes wie z.B Code-Duplikation beschreiben können.

Literaturverzeichnis

[AK09] Apel, S.; Kästner, C.: An overview of feature-oriented software development. *Journal of Object Technology*, Band 8, Nr. 5, S. 49–84, 2009.

[ALMK08] Apel, S.; Lengauer, C.; Möller, B.; Kästner, C.: An algebra for features and feature composition. In Meseguer, J.; Rosu, G. (Hrsg.): *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, Lecture Notes in Computer Science, Band 5140, S. 36–50. Springer, 2008.

[ALRS05] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In Glück, R.; Lowry, M. R. (Hrsg.): *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, Lecture Notes in Computer Science, Band 3676, S. 125–140. Springer, 2005.

[Bat03] Batory, D.: A tutorial on feature oriented programming and product-lines. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, S. 753–754. IEEE Computer Society, Piscataway, NJ, Mai 3–10 2003.

[Bat04] Batory, D. S.: Feature-oriented programming and the AHEAD tool suite. In *ICSE*, S. 702–703. IEEE Computer Society, 2004.

[BBM96] Basili, V. R.; Briand, L. C.; Melo, W. L.: A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, Band 22, Nr. 10, S. 751–761, Oktober 1996.

[BMAC05] Benavides, D.; Martín-Arroyo, P. T.; Cortés, A. R.: Automated reasoning on feature models. In Pastor, O.; Cunha, J. F. e. (Hrsg.): *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, Lecture Notes in Computer Science, Band 3520, S. 491–503. Springer, 2005.

[BMB96] Briand, L.; Morasca, S.; Basili, V.: Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, Band 22, Nr. 1, S. 68–86, Januar 1996.

[BSR04] Batory, D. S.; Sarvela, J. N.; Rauschmayer, A.: Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, Band 30, Nr. 6, S. 355–371, Juni 2004.

[BSTC07] Benavides, D.; Segura, S.; Trinidad, P.; Cortés, A. R.: FAMA: Tooling a framework for the automated analysis of feature models. In Pohl, K.; Heymans, P.; Kang, K. C.; Metzger, A. (Hrsg.): *First International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2007, Limerick, Ireland, January 16-18, 2007. Proceedings*, Lero Technical Report, Band 2007-01, S. 129–134, 2007.

[BWIL99] Briand, L. C.; Wüst, J.; Ikonomovski, S. V.; Lounis, H.: Investigating quality factors in object-oriented designs: An industrial case study. In *ICSE*, S. 345–354, 1999.

[CALO94] Coleman, D.; Ash, D.; Lowther, B.; Oman, P.: Using metrics to evaluate software system maintainability. *Computer*, Band 27, S. 44–49, August 1994.

[CCI90] Chikofsky, E. J.; Cross II, J. H.: Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, Band 7, S. 13–17, January 1990.

[CK91] Chidamber, S. R.; Kemerer, C. F.: Towards a metrics suite for object oriented design. In *OOPSLA*, S. 197–211, 1991.

[CK94] Chidamber, S.; Kemerer, C.: A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, Band 20, Nr. 6, S. 476–493, 1994.

[CL07] Cullmann, X.-N.; Lambertz, K.: Komplexität und qualität von software, 2007.

[CW85] Cardelli, L.; Wegner, P.: On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, Band 17, S. 471–523, December 1985.

[DBDV04] Du Bois, B.; Demeyer, S.; Verelst, J.: Refactoring ïmproving coupling and cohesion of existing code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, S. 144–151. IEEE Computer Society, Washington, DC, USA, 2004.

[DC04] Dr. Ricky E. Sward, Dr. A.T. Chamillard; Cook, D. D. A.: Using software metrics and program slicing for refactoring. *CrossTalk - The Journal of Defense Software Engineering*, Juli 2004.

[DeM89] DeMarco, T.: *Software Projektmanagement : wie man Kosten, Zeitaufwand und Risiko kalkulierbar plant.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st. Auflage, 1989.

[DJ03] Dagpinar, M.; Jahnke, J. H.: Predicting maintainability with object-oriented metrics - an empirical comparison. In Deursen, A. v.; Stroulia, E.; Storey, M.-A. D. (Hrsg.): *10th Working Conference on Reverse Engineering (WCRE 2003), 13-16 November 2003, Victoria, Canada*, S. 155–164. IEEE Computer Society, 2003.

[DW02] Dudziak, T.; Wloka, J.: Tool-supported discovery and refactoring of structural weaknesses in code., 2002.

[EC00] Eisenecker, U. W.; Czarnecki, K.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Fen94] Fenton, N.: Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, Band 20, Nr. 3, S. 199–206, März 1994.

[FL78] Fitzsimmons, A.; Love, T.: A review and evaluation of software science. *ACM Computing Surveys*, Band 10, Nr. 1, S. 3–18, März 1978.

[Fow99] Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[FP97] Fenton, N.; Pfleeger, S. L.: *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997.

[Gra94] Grady, R. B.: Successfully applying software metrics. *IEEE Computer*, Band 27, Nr. 9, S. 18–25, September 1994.

[Gri00] Griss, M. L.: Implementing product-line features by composing aspects. In *1st Conf. Software Product Lines: Experience and Research Directions*, S. 271–288. Kluwer Academic Publishers, Boston, 2000.

[Hal77] Halstead, M. H.: *Elements of Software Science*. Elsevier, New York, 1977.

[HF82] Hamer, P. G.; Frewin, G. D.: M. H. halstead's software science - A critical examination. In *ICSE*, S. 197–207, 1982.

[HKV07] Heitlager, I.; Kuipers, T.; Visser, J.: A practical model for measuring maintainability. In Machado, R. J.; Abreu, F. B. e.; Cunha, P. R. d. (Hrsg.): *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*, S. 30–39. IEEE Computer Society, 2007.

[HM95] Hitz, M.; Montazeri, B.: Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC '95)*, Oktober 1995.

[IEE19] IEEE: *IEEE Standard for Software Maintenance*. Institute of Electrical and Electronics Engineers, 1219.

[ISO01] ISO/IEC9126-1: *Software engineering - product quality - part 1: Quality model*. ISO, Geneva, Switzerland, 2001.

[Kan02] Kan, S. H.: *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd. Auflage, 2002.

[KBA09] Kuhlemann, M.; Batory, D. S.; Apel, S.: Refactoring feature modules. In Edwards, S. H.; Kulczycki, G. (Hrsg.): *Formal Foundations of Reuse and Domain Engineering, 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings*, Lecture Notes in Computer Science, Band 5791, S. 106–115. Springer, 2009.

[KM90] Korson, T.; McGregor, J. D.: Understanding object-oriented: a unifying paradigm. *Commun. ACM*, Band 33, S. 40–60, September 1990.

[KPS08] Kim, J.; Park, S.; Sugumaran, V.: DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines. *The Journal of Systems and Software*, Band 81, Nr. 1, S. 37–55, Januar 2008.

[Lad03] Laddad, R.: *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[LH93] Li, W.; Henry, S.: Object oriented metrics that predict maintainability. *Journal of System Software*, Band 23, Nr. 2, S. 111–122, 1993.

[LHR88] Lieberherr, K.; Holland, I.; Riel, A.: Object-oriented programming: An objective sense of style. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, S. 323, November 1988. Published as Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices, volume 23, number 11.

[LKO04] Li, D. Y.; Kiricenko, V.; Ormandjieva, O.: Halstead's software science in today's object oriented world. In *Metrics News Journal (9) 2*, S. 33–40, 2004.

[Män03] Mäntylä, M.: Bad smells - a taxonomy and an empirical study, 2003.

[Mat06] Matinlassi, M.: *Quality-driven software architecture model transformation.* Dissertation, University of Oulu, 2006.

[McC76] McCabe, T. J.: A measure of complexity. *IEEE Transactions on Software Engineering*, Band 2, Nr. 4, S. 308–320, Dezember 1976.

[Mey97] Meyer, B.: *Object-oriented software construction (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[MF82] Misek-Falkoff, L. D.: A unification of halstead's software science counting rules for programs and english text, and a claim space approach to extensions. *SIGMETRICS Performance Evaluation Review*, Band 11, Nr. 2, S. 80–114, 1982.

[MKF06] Murphy; Kersten; Findlater: How are Java software developers using the Eclipse IDE? *IEEE Software*, Juli 2006.

[MP05] Mitchell, Á.; Power, J. F.: Using object-level run-time metrics to study coupling between objects. In Haddad, H.; Liebrock, L. M.; Omicini, A.; Wainwright, R. L. (Hrsg.): *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, S. 1456–1462. ACM, 2005.

[MT04] Mens, T.; Tourwé, T.: A survey of software refactoring. *IEEE Trans. Softw. Eng.*, Band 30, S. 126–139, February 2004.

[MVL03a] Mäntylä, M.; Vanhanen, J.; Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In *ICSM*, S. 381–384. IEEE Computer Society, 2003.

[MVL03b] Mäntylä, M.; Vanhanen, J.; Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In *ICSM*, S. 381–384. IEEE Computer Society, 2003.

[Nor02] Northrop, L. M.: Sei's software product line tenets. *IEEE Softw.*, Band 19, S. 32–40, July 2002.

[OH92] Oman, P.; Hagemeister, J.: Metrics for assessing a software system's maintainability. In *Proceedings of the International Conference on Software Maintenance 1992*, S. 337–344. IEEE Computer Society Press, November 1992.

[O'N93] O'Neal, M. B.: An empirical study of three common software complexity measures. In *SAC*, S. 203–207, 1993.

[Opd92] Opdyke, W. F.: *Refactoring object-oriented frameworks*. Dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[Par72] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM*, Band 15, S. 1053–1058, December 1972.

[PBvdL05] Pohl, K.; Böckle, G.; Linden, F. J. v. d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1. Auflage, 2005.

[Pre97] Prehofer, C.: Feature-oriented programming: A fresh look at objects. In *ECOOP*, S. 419–443, 1997.

[RBJ97] Roberts, D.; Brant, J.; Johnson, R. E.: A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, Band 3, Nr. 4, S. 253–263, 1997.

[Rob99] Roberts, D. B.: *Practical analysis for refactoring*. Dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999. AAI9944985.

[Rom87] Rombach, H. D.: A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, Band 13, Nr. 3, S. 344–354, März 1987.

[RPB09] Rossel, P. O.; Perovich, D.; Bastarrica, M. C.: Reuse of architectural knowledge in SPL development. In Edwards, S. H.; Kulczycki, G. (Hrsg.): *Formal Foundations of Reuse and Domain Engineering, 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings*, Lecture Notes in Computer Science, Band 5791, S. 191–200. Springer, 2009.

[Sal82] Salt, N. F.: Defining software science counting strategies. *SIGPLAN Notices*, Band 17, Nr. 3, S. 58–67, 1982.

[SK03] Subramanyam, R.; Krishnan, M. S.: Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Software Eng*, Band 29, Nr. 4, S. 297–310, 2003.

[SKPA10] Siegmund, N.; Kuhlemann, M.; Pukall, M.; Apel, S.: Optimizing non-functional properties of software product lines by means of refactorings. In Benavides, D.; Batory, D. S.; Grünbacher, P. (Hrsg.): *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, ICB-Research Report, Band 37, S. 115–122. Universität Duisburg-Essen, 2010.

[SLL99] Simon, F.; Löffler, S.; Lewerentz, C.: Distance based cohesion measuring. In *In Proceedings of the 2nd European Software Measurement Conference (FESMA)*, S. 69–83, 1999.

[SMC74] Stevens, W. P.; Myers, G. J.; Constantine, L. L.: Structured design. *IBM Systems Journal*, Band 2, S. 115–139, 1974.

[Sny86] Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In *ACM Conference on Object-Oriented Systems, Languages and Applications*, 1986.

[SRK+08] Siegmund, N.; Rosenmüller, M.; Kuhlemann, M.; Kästner, C.; Saake, G.: Measuring non-functional properties in software product line for product derivation. In *APSEC*, S. 187–194. IEEE, 2008.

[SSL01] Simon, F.; Steinbrückner, F.; Lewerentz, C.: Metrics based refactoring. In *CSMR*, S. 30–38, 2001.

[SSSP07] Sincero, J.; Spinczyk, O.; Schröder-Preikschat, W.: On the configuration of non-functional properties in software product lines. In *SPLC (2)*, S. 167–173. Kindai Kagaku Sha Co. Ltd, Tokyo, Japan, 2007.

[TM03] Tourwé, T.; Mens, T.: Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference on Software Maintenance and Reengineering (CSMR '03)*, S. 91–100, 2003.

[vdLSR07] Linden, F. v. d.; Schmid, K.; Rommes, E.: *Software product lines in action - the best industrial practice in product line engineering.* Springer, 2007.

[vM02] van Emden, E.; Moonen, L.: Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, S. 97–107. IEEE Computer Society Press, Oktober 2002.

[WDS09] White, J.; Dougherty, B.; Schmidt, D. C.: Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening. *The Journal of Systems and Software*, Band 82, Nr. 8, S. 1268–1284, August 2009.

[Weg90] Wegner, P.: Concepts and paradiconf/apsec/2008gms of object-oriented programming. *SIGPLAN OOPS Mess.*, Band 1, S. 7–87, August 1990.

[Wel01] Welker, K. D.: The software maintainability index revisited. *CrossTalk - The Journal of Defense Software Engineering*, August 2001.

[Wey88] Weyuker, E. J.: Evaluating software complexity measures. *IEEE Trans. Software Eng.*, Band 14, Nr. 9, S. 1357–1365, 1988.

[WH88] Wake, S. A.; Henry, S. M.: A model based on software quality factors which predicts maintainability. Technical Report Nr. TR-88-08, Virginia Polytechnic Inst. and State University, Mai 1988.

[WOZ91] Weide, B. W.; Ogden, W. F.; Zweben, S. H.: *Reusable software components*, S. 1–65. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[WW90] Wand, Y.; Weber, R.: An ontological model of an information system. *IEEE Transactions on Software Engineering*, Band 16, Nr. 11, S. 1282–1292, November 1990.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 19. September 2011

Son Hoang, Luong