

Fernuniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

**Abschlussarbeit im Studiengang
Master of Computer Science**

Systematisches Testen von Constraintregeln

Marius Kreis

Matrikelnummer: 7614535

Betreuer:
Andreas Thies

15. Mai 2011

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Wörtlich oder inhaltlich übernommene Literaturquellen wurden besonders gekennzeichnet.

Marius Kreis
Stuttgart, Mai 2011

Inhaltsverzeichnis

1	Einleitung	7
2	Theoretische Grundlagen	9
2.1	Was ist ein Constraint?	9
2.2	Constraint-basiertes Refactoring	11
2.3	Lösung von Constraint-Satisfaction- und Optimierungsproblemen	11
2.3.1	Systematische Suche	12
2.3.1.1	Generate and Test	12
2.3.1.2	Backtracking	12
2.3.1.3	Backjumping	13
2.3.1.4	Backchecking und Backmarking	14
2.3.2	Konsistenzverfahren	15
2.3.3	Constraint Optimierung	16
2.3.3.1	Branch and Bound	16
2.4	Refacola	17
2.4.1	Darstellung eines Programms mittels Constraints	19
2.4.1.1	Programmelemente	19
2.4.1.2	Verwendung von Domains	20
2.4.1.3	Herleitung von Constraints aus dem Programmtext	20
2.4.1.4	Beispiele für Queries	21
2.4.1.5	Beispiele Constraint Rules	21
2.4.1.6	Welche Constraints gibt es?	22
2.4.1.7	Welche Constraints werden erzeugt?	23
2.4.2	Finden der Lösung	24
2.4.3	Zurückschreiben der Constraints in den Programmtext	24
2.4.4	Definition von Refactorings	25
2.4.5	Ablauf von Refactorings	26
2.5	Programmiersprachenentwicklung mit Xtext und Xpand	27
2.5.1	Die Entwicklung der Refacola Sprachdefinition mit Xtext	28
2.5.2	Die Erzeugung der Refacola Implementierung mit Xpand	30
3	Praktische Umsetzung	34
3.1	Planung	34

Inhaltsverzeichnis

3.1.1	Anforderungen	34
3.2	Implementierung	35
3.2.1	Implementierung mittels eines Eclipse Plugins	35
3.2.2	Einbindung in Eclipse	37
3.2.2.1	Das Eclipse Launch Framework	37
3.2.2.2	Die Ansicht der Testergebnisse (Test Result View)	38
3.2.2.3	Die Ansicht der Details zu einem Test (Test Detail View)	38
3.2.2.4	Die Ansicht der Programmänderungen (Refactoring View)	40
3.2.3	Architektur	40
3.2.3.1	Hierarchie der Packages	40
3.2.4	Die Ausführung im Detail	41
3.2.4.1	Konfiguration und Start des Tests	41
3.2.4.2	Verwendung der Refacola Java Implementierung	41
3.2.4.3	Durchführung der Tests	42
4	Durchführung der Tests	44
4.1	Planung	44
4.1.1	Aufgetretene Probleme und ihre Lösung	45
4.1.1.1	OutOfMemoryError: Java heap space	45
4.1.1.2	OutOfMemoryError: PermGen space	47
4.1.1.3	IllegalAccessError: tried to access class X from class Y	47
4.2	Durchführung und gefundene Fehler	47
4.2.1	Gefundene Fehler	49
4.2.1.1	ArrayIndexOutOfBoundsException	49
4.2.1.2	Fehler während der Faktengenerierung	49
4.2.1.3	ClassCastException	49
4.2.1.4	AssertionFailedException	50
4.2.1.5	Keine Lösung für das Constraint System	50
4.2.1.6	Mehrere Klassen mit Sichtbarkeit „public“ in einer Datei	50
4.2.1.7	Sichtbarkeit von importierten Klassen	50
4.2.2	Variationen mit anderen Properties	50
5	Fazit und Ausblick	52
A	Stacktraces der gefundenen Fehler	54
B	Inhalt der beigelegten CD	57
C	Installation des Constraint Testers	58
D	Benutzungsanleitung für den Constraint Tester	59

Abbildungsverzeichnis

2.1	Vorgehensweise beim Backtracking	13
2.2	Vorgehensweise beim Backjumping	15
2.3	Bestandteile der Refacola Architektur, in Anlehnung an (Pil11)	18
2.4	Constraintregel	20
2.5	Constraintregel für die Bindung von Variablen	21
3.1	Übersicht der Bestandteile des Constraint Testers	36
3.2	Klassenhierarchie der Launch Configuration (Auszug an Methoden)	37
3.3	Klassenhierarchie der Benutzungsoberfläche (Auszug an Methoden)	38
3.4	Bildschirmfoto des LaunchConfigurationDialog	39
3.5	Darstellung des Testergebnisses	43
4.1	Aufteilung des Heap Speichers (nach (SM03))	45
4.2	CPU und Heap Auslastung mit manuellen GC Parametern	47
D.1	Anordnung der Views	59
D.2	Konfiguration des Tests	60

Abkürzungsverzeichnis

AST Abstract Syntax Tree

COP Constraint Optimization Problem

DSL Domain Specific Language

EMF Eclipse Modeling Framework

GPL General Purpose Language

IDE Integrated Developing Environment

JVM Java Virtual Machine

UI User Interface

1. Einleitung

Mit zunehmendem Umfang und steigender Komplexität des Quelltexts von Programmen werden Hilfsmittel benötigt, die den Umgang (z. B. Wartung, Fehlersuche, Umstrukturierung) mit diesen Programmen bestmöglich vereinfachen oder ganz übernehmen. Dies sind hauptsächlich die Entwicklungsumgebungen (Integrated Developing Environment, IDE), die z. B. während des Tippens durch automatische Vervollständigung (sog. Code Completion) die Arbeit des Entwicklers sehr vereinfachen. Auch bieten Sie Unterstützung, wenn nachträglich Änderungen am Programm vorgenommen werden müssen, z. B. um die Struktur zu verbessern, indem Programmteile in eigene Methoden ausgelagert werden. Hierbei darf sich jedoch das Verhalten, also die Semantik des Programms, keinesfalls verändern und es muss syntaktisch korrekt bleiben. Man spricht bei dieser Tätigkeit von Refaktorisierungen (engl. Refactoring).

Die Syntax moderner Programmiersprachen wie Java bietet eine Vielzahl von Möglichkeiten, die bereits die Entwicklung von Werkzeugen für einfache Refaktorisierungen sehr aufwendig machen, da es immer Sonderfälle gibt, die berücksichtigt werden müssen. Dies zeigt sich z. B. schon bei einer einfachen Refaktorisierung wie dem Verschieben einer Klasse in ein anderes Paket, für die Sonderfälle existieren, die selbst in aktuellen Werkzeugen wie Eclipse, NetBeans oder IDEA IntelliJ nicht korrekt durchgeführt werden (siehe Abschnitt 2.1 in (ST09)). Eine Fehlerquelle ist die Art und Weise, wie die Refaktorisierungen programmiert wurden. Bei einer Formulierung aller einzelnen, für die Durchführung notwendigen Schritte, sind Sonderfälle besonders problematisch. Für jeden Sonderfall müssen zusätzliche Schritte, Abfragen und Verzweigungen eingefügt werden, wodurch die ganze Prozedur so unübersichtlich wird, dass sich leicht Fehler einschleichen. Zudem ist der Code schwer wartbar und erweiterbar.

Eine Lösung bietet hier die deklarative Beschreibung in Form von sog. Constraintregeln (siehe Abschnitt 2.1). Die Regeln werden unabhängig von einander formuliert, jeder zusätzliche Sonderfall kann unabhängig von den Hauptregeln formuliert werden, wovon die Lesbarkeit profitiert.

Refaktorisierungen mit Hilfe von Constraintregeln (Abschnitt 2.2) sind aktueller Forschungsgegenstand des Lehrgebiets Programmiersysteme der Universität Hagen. Die Herausforderung liegt darin, die oben skizzierten Problemfelder durch ein formale, sprachunabhängige Beschreibung der Umformungsregeln zu adressieren, um sie auf unterschiedliche Programmiersprachen anwenden zu können. Zusätzlich muss sichergestellt sein, dass das Programm, wie oben erwähnt, semantisch und syntaktisch korrekt bleibt.

1. Einleitung

Denn nur wenn die Regeln vollständig und korrekt sind, ist jede Lösung, die alle Bedingungen erfüllt, eine semantisch und syntaktisch korrekte Refaktorisierung des ursprünglichen Programms.

In mehreren wissenschaftlichen Artikeln (u.a. (ST09), (TKB03)), werden bereits Constraints definiert und Regeln aufgestellt, wie man aus einem Programm ein System mit Constraints ableiten kann.

In einem aktuellen Projekt an der Universität Hagen wird „Refacola“ entwickelt, eine Sprache, um diese Constraintregeln und Refaktorisierungen unabhängig von einer bestimmten Programmiersprache zu formulieren. Dazu gehört auch noch ein umfassendes Rahmenwerk, das in Abschnitt 2.4 erklärt wird.

Die Aufgabenstellung dieser Arbeit ist es, ein Werkzeug zu erstellen, mit dem die Korrektheit der Constraintregeln getestet werden kann (siehe Abschnitt 3.1.1). Dabei werden unter Einhaltung der Constraintregeln zufällige Refaktorisierungen vorgenommen. Das so veränderte Programm wird anschließend verschiedenen Tests unterzogen, um sicherzustellen, dass sich das Programm semantisch nicht verändert hat und syntaktisch korrekt ist.

2. Theoretische Grundlagen

2.1. Was ist ein Constraint?

Der Definition in (Bar05) zufolge ist ein Constraint eine logische Beziehung zwischen zwei Variablen, wobei jeder Variable Werte aus einem gegebenen Bereich (engl. Domain) zugewiesen werden können. Das bedeutet, dass ein Constraint die möglichen Werte einschränkt, die eine Variable annehmen kann. In (Apt03) wird ein Constraint wie folgt definiert: Ein Constraint auf einer Reihe von Variablen ist eine Beziehung unter ihren Wertebereichen (Domains). Er kann als eine Voraussetzung gesehen werden, die ausdrückt, welche Kombinationen von Werten der Variablen zulässig sind. Dabei ist ein Constraint deklarativ, d. h. er beschreibt nur, welche Bedingung erfüllt sein muss, aber nicht, wie dies berechnet wird.

Als Constraint-logisches Problem wird eine ungeordnete Sammlung von Constraints bezeichnet, wobei jedes eine Teilmenge der verfügbaren Variablen einschränkt und gleichermaßen Gültigkeit hat.

Als erste Implementierung von Constraints in der Informatik wird die interaktive Grafikanwendung Sketchpad von I. Sutherland aus dem Jahr 1963 gesehen, in der Beziehungen zwischen Grafikobjekten als Constraints ausgedrückt werden konnten, die über numerische Verfahren gelöst wurden (MS98). In den siebziger Jahren sind weitere experimentelle Sprachen entstanden, die das Konzept der Constraints übernahmen und das Lösen der Constraints zur Grundlage hatten. Auf dem Gebiet der Künstlichen Intelligenz wurden diese sog. Constraint Satisfaction, also die Suche nach Belegungen für die Variablen, mit denen alle Constraints erfüllt sind, weiter erforscht. Des Weiteren wurden Konsistenzverfahren entwickelt, bei denen die Domains der Constraints bereits vor der Suche eingeschränkt werden, wodurch die Suche beschleunigt wird. Für die Suche wurden Methoden entwickelt, so z.B. das sog. Backtracking, das seine Ursprünge bereits im 19. Jahrhundert hat.

In den Achtzigern wurden die Constraint-Programmierung entwickelt. Die wichtigsten davon basierten auf dem Paradigma der Logischen Programmierung, woraus sich die Constraint-Logische-Programmierung entwickelte, eine Erweiterung der Logischen Programmierung durch den Aspekt der Constraints.

Durch diese Fortschritte entstanden in den Neunzigern neue Anwendungsgebiete für die Constraint-Programmierung, insbesondere auf dem Gebiet des Operations Research

2. Theoretische Grundlagen

und der Numerischen Analyse. Besonders die Entwicklung neuer Constrainttypen verhalf der Constraint-Programmierung zu diesem Fortschritt (HW09).

Constraints werden oft in Bereichen verwendet, in denen Fragestellungen, die sich aus realen Bedingungen ergeben, mit Hilfe des Computers gelöst werden sollen. In diesem Fall vereinfacht die Verwendung von Constraints durch ihre deklarative Beschreibung die Formulierung der Regeln. Aus diesem Grund können Regeln meist ohne aufwendige Umformulierung übernommen werden.

Ein einfaches Beispiel für Constraints, die im täglichen Leben vorkommen, sind folgende Sätze:

- „Wenn ich Urlaub mache, dann will ich eine Stadtrundfahrt machen.“
- „Ich kann nur von Mai bis Juni Urlaub nehmen.“
- „Für die Stadtrundfahrt gibt es nur noch im Zeitraum vom 26.6. bis 31.7. freie Plätze.“
- „Stadtrundfahrten finden nur Sonntags statt.“

Umgeformt, so dass sie auch ein Computer versteht, lauten sie wie folgt:

- $\text{Stadtrundfahrtstag} = \text{Urlaubstag}$
- $1.5.2011 \leq \text{Urlaubstag} \leq 31.6.2011$
- $26.6.2011 \leq \text{Stadtrundfahrtstag} \leq 31.7.2011$
- $\text{Wochentag}(\text{Stadtrundfahrtstag}) = \text{'Sonntag'}$

Die Reihenfolge der Anwendung von Constraints spielt keine Rolle, jedoch teilen sich Constraints oft Variablen, deren Belegung für die Lösung herausgefunden werden muss, wie in diesem Beispiel. Hier fällt die Lösung leicht, da alle Informationen bekannt sind um auf genau eine Lösung zu kommen: Die Stadtrundfahrt muss am 26.6.2011 stattfinden. Es ist jedoch auch möglich, dass Constraints nur Teilinformationen enthalten, z. B. „ $X > 5$ “. Hier kann X die Werte 6, 7, 8 etc. annehmen, wodurch der Wertebereich für X eingeschränkt wird. Constraints können in beide Richtungen umgeformt werden, z. B. wenn $X = Y + 2$ gilt, dann gilt auch $Y = X - 2$. Wie oben im letzten Constraint gezeigt, ist es auch möglich, dass Constraints Variablen unterschiedlicher Wertebereiche (sog. Domains, siehe 2.4.1.2) miteinander mischen.

2.2. Constraint-basiertes Refactoring

Die Idee, Constraints zu verwenden, um sicherzustellen, dass sich das Verhalten trotz Veränderungen am Programmen nicht ändert, wird bereits in (Opd92) beschrieben. Ein Vorteil von Constraints ist, dass sie einen beschreibenden Charakter haben, weshalb sie leicht aus Sprachdefinitionen abgeleitet werden können. Insbesondere die Eiffel Sprachdefinition (ECM06) enthält mehrere Regeln, die sich leicht auf Constraints abbilden lassen. Dies ist auch einer der Gründe, warum Eiffel (neben Java) als Zielsprache für die Implementierung von Refacola herangezogen wurde (SKP11).

Erste Formulierungen von Constraints für ein Typsystem (Type Constraints) finden sich in (PS94). Um Probleme bei der Typüberprüfung (Type checking) und -herleitung (Type inference) zu lösen, wurden in diesem Buch formal Beziehung zwischen Typen und Subtypen von Programmelementen definiert. Diese Definitionen werden in vielen wissenschaftlichen Artikeln zum Thema Refaktorisierung mit Type Constraints aufgegriffen.

Ein Beispiel hierfür ist der Artikel (TKB03), in dem Frank Tip eine Lösung vorstellt, wie Refactorings für die Generalisierung umgesetzt werden können. Die Idee dabei ist, aus einem existieren Programm die oben genannten Type Constraints abzuleiten.

Für das auf diese Weise erstellte Constraint System stellt das ursprüngliche Programm nur eine von mehreren Lösungen dar, es sollte jedoch auch möglich sein, andere Lösungen zu finden, die dann einer refaktorierten Version des ursprünglichen Programms entsprechen.

Bisherige Entwicklungen auf dem Gebiet der constraintbasierten Refaktorisierung betrachten meist nur einen Teilaspekt (Typconstraints oder Sichtbarkeitsconstraints), wodurch die Art der möglichen Refaktorisierungen eingeschränkt ist. Außerdem wird größtenteils nur eine Programmiersprache, meist Java, betrachtet. Vor diesem Hintergrund wurde am Lehrgebiet Programmiersysteme der Universität Hagen das in Abschnitt 2.4 vorgestellte Projekt Refacola gestartet, dessen Ziel es ist, eine Framework für die sprachunabhängige Formulierung von Refactorings jedweder Art zu entwickeln und konkret für die Sprachen Eiffel und Java umzusetzen.

2.3. Lösung von Constraint-Satisfaction- und Optimierungsproblemen

Bei der Lösung von Constraint-Satisfaction-Problemen geht es darum, eine Belegung für alle Constraint Variablen zu finden, so dass alle Constraints erfüllt sind, wobei jede Lösung gleich gut ist. Bei Constraint Optimierungsprobleme hingegen gibt es unterschiedlich gute Lösungen, wobei eine von der konkreten Aufgabenstellung abhängige Methode jede gefundene Lösung nach bestimmten Kriterien bewertet, so dass unter allen

Lösungen die beste ermittelt werden kann. Im folgenden werden zunächst verschiedene Vorgehensweisen vorgestellt, mit denen Constraint-Satisfaction-Probleme gelöst werden können, anschließend wird ein Verfahren für die Lösung von Constraint-Optimierungsproblemen beschrieben.

2.3.1. Systematische Suche

2.3.1.1. Generate and Test

Bei dieser Methode werden die Constraintvariablen mittels eines sog. Generators mit Werten aus ihren Domains belegt. Für jede Belegung wird anschließend geprüft, ob damit alle Constraints erfüllt sind. Ist dies nicht der Fall, erzeugt der Generator die nächste Belegung. Dadurch, dass der Generator alle möglichen Werte aus den Domains benutzt, sucht diese Methode den ganzen Raum aller möglichen Zuweisungen ab. Ist eine Lösung gefunden, wird die Suche beendet. Erst wenn alle Zuweisungen ausprobiert wurden und keine Lösung gefunden wurde, kann man sicher sein, dass kein Lösung existiert. Die maximale Anzahl aller zu untersuchenden Zuweisungen entspricht damit dem kartesischen Produkt aller Domains. Die Generate-and-Test-Methode ist sehr ineffizient, vor allem da jede Belegung ungeachtet von offensichtlichen Konflikten geprüft wird. Besser wäre es, wenn beispielsweise der Generator bei der Erzeugung der Belegung berücksichtigen würde, warum der vorangegangene Test fehlgeschlagen ist. Außerdem könnten Generator und Test zusammengeführt werden, indem Constraints frühestmöglich getestet werden, nämlich sobald für die benötigten Variablen Werte zugewiesen wurden.

2.3.1.2. Backtracking

Beim Backtracking werden, wie bereits als Verbesserung des Generate-and-Test vorgeschlagen, die Generierungs- und Testphase kombiniert. Die Lösungsstrategie besteht aus einer schrittweisen Belegung der Constraintvariablen, wobei in jedem Schritt bereits eine Prüfung der Constraints erfolgt. Wird ein Constraint verletzt, wird versucht, die letzte Zuweisung rückgängig zu machen und stattdessen einen anderen Wert aus der jeweiligen Domain auszuwählen. Wurden bereits alle Werte erfolglos getestet (man befindet sich also in einer Sackgasse), wird die vorhergehende Zuweisung rückgängig gemacht. Diese Vorgehensweise nennt man chronologisches Backtracking, da die jüngsten Zuweisungen zuerst rückgängig gemacht werden. Durch das frühere Testen wird der Suchbereich frühzeitig um ungültige Bereiche verkleinert, die beim Generate-and-Test-Verfahren getestet worden wären. Aufgrund dieser Tatsache zeigt sich, dass Komplexität beim Backtracking im schlechtesten Fall immer noch so gut ist wie Generate-and-Test.

Abbildung 2.1 stellt die Vorgehensweise grafisch dar: Zu Beginn wird für die erste Variable der erste Wert A aus der Domain A, B, C ausgewählt und geprüft. Anschließend für die zweite Variable der erste Wert 1 aus der Domain $1, 2, 3$ ausgewählt und ebenfalls

2. Theoretische Grundlagen

geprüft. Für die dritte Variable wird der Wert A aus der Domain A, B, C ausgewählt, jedoch schlägt die Prüfung fehl. Nun wird der nächste Wert B ausprobiert, ebenfalls erfolglos. Nachdem der dritte und letzte Wert C aus der Domain ebenfalls zu keine Lösung führt, wird die Belegung der zweiten Variable zu 2 geändert.

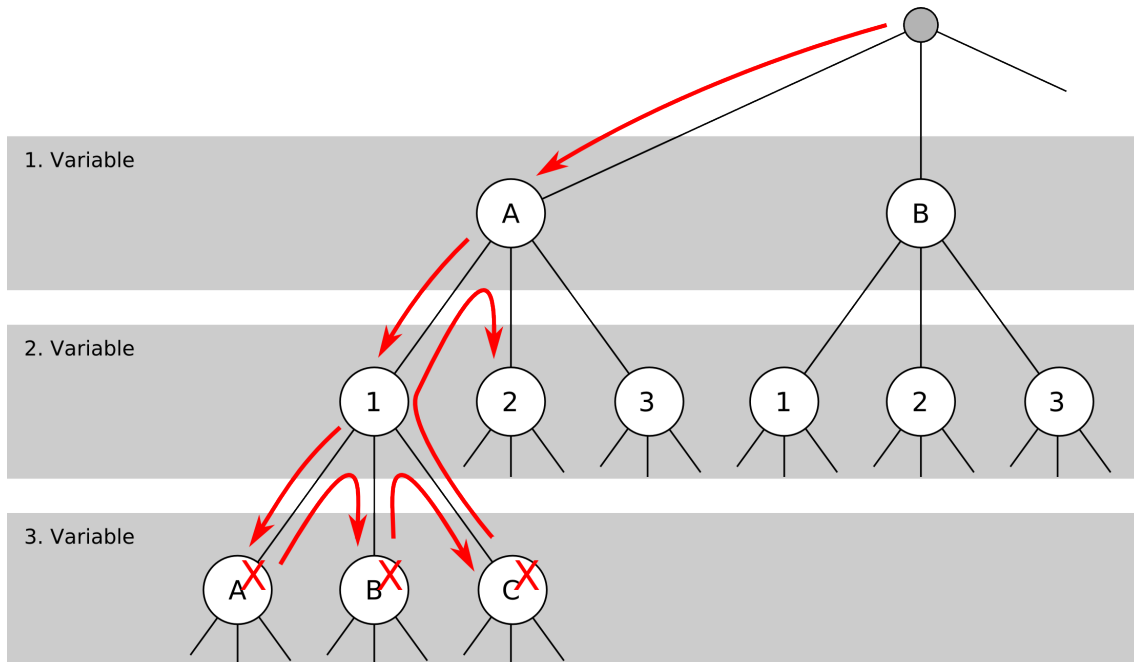


Abbildung 2.1.: Vorgehensweise beim Backtracking

2.3.1.3. Backjumping

Ein Problem des Backtracking ist, dass die Suche nach einer neuen Belegung mehrfach aufgrund der selben Ursache, z. B. einer Variable, die schon viel früher belegt wurde, fehlschlagen kann. Das Backtracking beschränkt sich jedoch zuerst darauf, die letzten Belegungen rückgängig zu machen, wodurch die Ursache bestehen bleibt.

Beim Backjumping wird, wie beim Backtracking auch, jeweils eine neue Belegung für eine Constraintvariable gesucht, die zusammen mit den bereits getroffenen Belegungen keine Constraintregeln verletzt. Gibt es jedoch einen Konflikt, dann wird beim Backjumping die Ursache gesucht, wobei die verletzte Constraintregel als Hinweis dient.

Der bekannteste Backjumping Algorithmus von John Gaschnig arbeitet zunächst wie beim Backtracking. Erst wenn er in eine Sackgasse gelangt, also alle Belegungen für die letzte noch zu belegende Variable erfolglos geprüft wurden, führt er eine Ursachenforschung durch, und zwar für jede der fehlgeschlagenen Belegungen: Es wird eine Liste der Variablen erstellt, die in die verletzten Constraintregeln einfließen. Dann wird für jedes Constraint die Variable ausgewählt, die zuletzt belegt wurde. Schließlich wird unter allen ausgewählten Variablen erneut diejenige ausgewählt, die zuerst belegt wurde.

2. Theoretische Grundlagen

Tabelle 2.1.: Statustabelle Variablenbelegungen beim Backjumping

Ebene	verletzte Constraintregeln			
	C_1	C_2	C_3	C_4
1		X		
2	X		X	$X_{L_{min}(2)}$
3	X	$X_{L_{min}(1)}$		
4			X_L	
5	X_L			
6				
7	X	X	X	X
Belegung	A		B	

Für alle (fehlgeschlagenen) Belegungen werden die verletzten Constraints untersucht. Gesucht ist die Variable, die in diesen Constraintregeln vorkommt und möglichst spät (vor möglichst wenigen Durchläufen) zugewiesen wurde.

Tabelle 2.1 stellt ein Beispiel dar (in Anlehnung an (Bar05), S.10ff), in dem bereits die Constraintvariablen $X_1 \dots X_6$ belegt wurden und nun ein Wert für die siebte gesucht wird. Möglichen Belegungen für X_7 sind die Werte A und B, wobei in beiden Fällen Constraintregeln verletzt werden, für A sind dies C_1 und C_2 , und für B sind es C_3 und C_4 . Die Variablen, die in den jeweiligen Constraintregeln vorkommen, sind in der Spalte unter ihnen eingetragen. Je höher die Ebene, desto später wurden sie belegt. Die Variablen mit der höchsten Ebene wurden als letztes belegt und sind damit der Konfliktvariable am nächsten, sie sind in der Tabelle mit dem Index L markiert. Sie stellen die Variablen dar, deren Änderung den aktuellen Konflikt lösen könnte, und die deshalb neu belegt werden müssen. Unter diesen Variablen wurden diejenigen auf der niedrigsten Ebene mit dem Index L_{min} gekennzeichnet. Sie kommen in die nähere Auswahl, weil sie am frühesten belegt wurden und damit die Neubelegung der anderen mit L markierten Variablen ohnehin notwendig ist. Somit bleiben nur noch zwei Möglichkeiten, jeweils eine pro Belegung A oder B übrig, markiert mit (1) und (2). Von diesen beiden wird nun diejenige ausgewählt, die als letztes belegt wurde - einfach um sich zusätzliche Neubelegungen zu sparen. Das Ergebnis ist nun, dass der Algorithmus zurück zur Belegung der Variable $X_{L_{min}(1)}$ springt und alle späteren Belegungen erneut auswählt.

2.3.1.4. Backchecking und Backmarking

Ein Nachteil des Backtracking und Backjumping ist, dass die gleichen Prüfungen von Constraintregeln mehrfach durchgeführt werden und die Variablen, die mit einander in Konflikt stehen, nicht gespeichert werden. Beim Backchecking werden Konflikte, die zwischen den Zuweisungen verschiedener Constraintvariablen entdeckt wurden, zwischen-

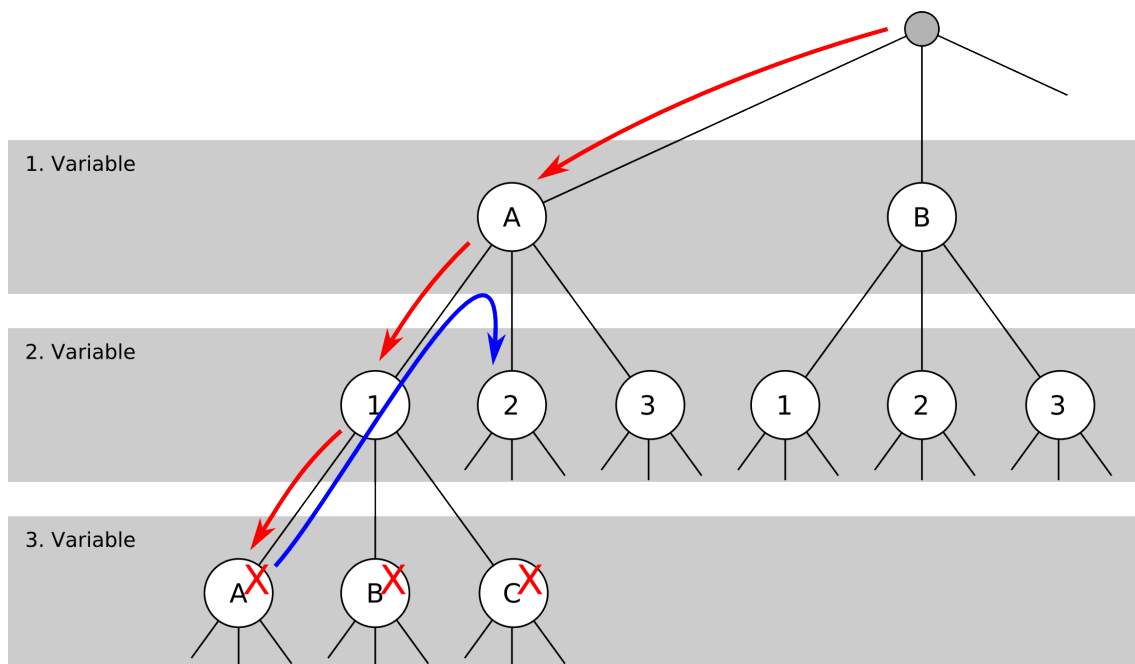


Abbildung 2.2.: Vorgehensweise beim Backjumping

gespeichert. Eine erneute Zuweisung erfolgt dann nicht mehr, so lange eine der beiden Werte zugewiesen bleibt. Eine weitere Verbesserung stellt der Backmarking Algorithmus dar, der zusätzlich die erfolgreichen Zuweisungen zwischenspeichert.

2.3.2. Konsistenzverfahren

Bei den beschriebenen Lösungsverfahren wurden die Wertebereiche der Variablen uneingeschränkt verwendet. Die Idee der Konsistenzverfahren ist, bereits vor der Lösungssuche die Domänen soweit wie möglich einzuschränken, um ungültige Belegungen frühzeitig auszuschließen. Bei leeren Wertebereichen kann das Constraintproblem bereits frühzeitig für unlösbar erklärt werden. Das einfachste Konsistenzverfahren betrifft Constraints, die nur eine Variable enthalten (sogenannte unäre Constraints). Indem man anhand all dieser Constraints die Wertebereich der betreffenden Variablen beschränkt, sind die Constraints immer erfüllt und können gestrichen werden. Den Zustand, den man durch die Streichung aller unären Constraints erreicht, nennt man Knotenkonsistenz (engl. node consistency).

Nächste Konsistenzebene betrifft sog. binäre Constraints mit zwei Variablen, wobei man von sog. Kantenkonsistenz (engl. arc consistency) spricht. Die die Betrachtung geht dabei immer von einer der beiden Variablen aus, so dass überprüft wird, dass es für jeden Wert aus der Domäne auch Belegungen für die zweite Variable gibt. Ist dies nicht der Fall, kann der Werts aus der Domäne der ersten Variable entfernt werden.

Anschließend wird in umgekehrter Richtung, also mit der zweiten Variable als Ausgangswert, geprüft. Diese Prüfung der Kantenkonsistenz muss möglicherweise mehrfach durchgeführt werden, da mehrere Constraints die gleichen Variablen enthalten und somit jede Änderungen Auswirkungen auf andere Constraints hat, wodurch sich der Wertebereich weiter einschränkt. Die dritte Konsistenzebene wird Pfadkonsistenz (engl. path consistency) genannt. Sie betrifft alle Constraints, deren Variablen v_1, v_2, \dots, v_m als Pfad von v_1 bis v_m betrachtet werden. Für jedes Teilstück v_i, v_{i+1} werden nun seinerseits die Konsistenzbedingung (Pfad- oder Kantenkonsistenz) geprüft, damit der Gesamtpfad als pfadkonsistent gilt. In anderen Worten muss es für Teilgruppen für benachbarte Constraintvariablen gültige Belegungen aus den Wertebereichen geben. Ungültige Belegungen werden gestrichen.

2.3.3. Constraint Optimierung

Die dargestellten Verfahren dienen dazu, für mehrere Constraints eine oder alle Lösungen zu suchen, wobei unter den Lösungen keine weitere Unterscheidung getroffen wird, sie sind alle gleich gut. In vielen Fällen, so auch bei der Ermittlung der Refaktorisierung, gibt es sowohl gute als auch schlechte Lösungen und der Constraint Solver soll die beste Lösung ermitteln. Um dies zu erreichen, bedarf es zusätzlich zu den Constraints noch einer Methode, die die Qualität einer Lösung berechnen kann. Mit Hilfe dieser Methode bewertet der Constraint Solver die gefundenen Lösungen und kann so die schlechten Lösungen verwerfen.

2.3.3.1. Branch and Bound

Bei dem Branch and Bound Verfahren handelt es sich um das gängigste Verfahren, um Constraint Optimierungsprobleme zu lösen (Bar05). Grundlage ist eine heuristische Methode, die auch für eine Teilbelegung der Constraintvariablen eine Schätzung liefern kann, welchen Wert die Bewertungsfunktion (für eine vollständige Belegung) berechnet.

Mit der heuristischen Methode kann der Suchraum schnell verkleinert werden, indem Suchpfade frühzeitig verlassen werden. Und zwar nicht nur, wenn absehbar ist, dass der aktuelle Pfad keine Lösung enthält, sondern auch, wenn absehbar ist, dass er keine optimale Lösung beinhaltet.

Je genauer diese Schätzungen an der echten Bewertungsfunktion liegen, desto effizienter arbeitet die Branch and Bound Methode. In keinen Fall darf die Schätzung aber verhindern, dass eine optimale Lösung nicht gefunden wird. Aus diesem Grund darf die Abschätzung der heuristischen Methode nie schlechter als die der echten Bewertungsfunktion sein. Je genauer die Heuristische Methode ist, desto größer wird auch der Rechenaufwand sein, so dass letztlich ein Kompromiss zwischen Qualität der Abschätzung und Dauer der Berechnung getroffen werden muss.

Die Vorgehensweise des sog. „Depth-First“ Algorithmus entspricht dem des chronologischen Backtrackings: Unter Annahme, dass eine Lösung gesucht wird um den Wert der Bewertungsfunktion zu minimieren (je kleiner, desto besser), wird eine Variable für eine Schranke (engl. Bound) mit dem größtmöglichen Wert „Unendlich“ initialisiert. Diese Schranke stellt den Wert für die bislang beste gefundene Lösung dar. Anschließend wird jede Teillösung mit der heuristischen Methode bewertet. Ist das Resultat kleiner als die Schranke, wird mit dieser Teillösung weiter gearbeitet, d.h. es wird eine Belegung für die nächste Variable gesucht. Andernfalls wird die Belegung der letzten Variable wieder rückgängig gemacht (Backtracking) und der entsprechende Teilbaum komplett verworfen. Ist die Belegung komplett, d.h. für alle Variablen wurden gültige Werte gefunden, dann wird die echte Bewertungsfunktion verwendet, um die Qualität der gefundenen Lösung zu berechnen. Liegt dieser Wert unter der Schranke, wird der Wert der Schranke nach unten korrigiert und die Lösung als beste bekannte Lösung gespeichert. Nachdem alle Lösungen gefunden und bewertet wurden, ist die beste Lösung bekannt.

2.4. Refacola

Bei Refacola handelt es sich um eine in (SKP11) vorgestellte, am Lehrgebiet Programmiersysteme der Fernuniversität Hagen entwickelte Sprache, in der constraintbasierte Refaktorisierungen spezifiziert werden können. Dazu gehören aber auch Werkzeuge wie ein Compiler, die diese Spezifikation in eine konkrete Implementierung umwandeln, die die Refaktorisierung durchführt. Refacola ist nicht auf eine einzelne Programmiersprache beschränkt, zeitgleich mit der Entwicklung von Refacola wurden auch Beispielimplementierungen für Eiffel und Java erstellt.

Für jede Programmiersprache müssen in Refacola zwei Teile definiert werden, die Sprachdefinition und die Constraintregeln. Die Sprachdefinition gibt vor, welche Programmelemente (engl. Program Elements) es in der Sprache gibt, welche Eigenschaften (engl. Properties) sie haben, welcher Wertebereich (engl. Domains) für die Properties gültig ist und welche Fakten abgeleitet werden können (siehe 2.4.1.1). Die Fakten spiegeln die Beziehungen zwischen den Programmelementen wider und werden in einer Datenbank gespeichert, so dass Abfragen (sog. Queries) einfach und effizient erfolgen können. Zweiter Teil ist die Definition der Constraintregeln (engl. Constraint Rules, siehe 2.4.1.5), anhand derer ein Constraintsystem erzeugt wird, welches das Programm repräsentiert. Die Constraintregeln werden mit Hilfe o. g. Queries formuliert, die auf die Faktendatenbank zugreifen. Unter der Annahme, dass beide Teile vollständig und korrekt definiert sind, können beliebige Programme in ein Constraintsystem umgewandelt werden. Dabei stellt das Ursprungsprogramm selbst nur eine Lösung des Systems dar, mit Hilfe eines Constraint Solvers können weitere Lösungen gefunden werden, die syntaktisch und semantisch korrekten Refaktorisierungen entsprechen.

2. Theoretische Grundlagen

Da das Ziel jedoch nur eine ganz bestimmte Refaktorisierung ist, muss die gewünschte Änderung im erstellten Constraintsystem vorgenommen werden, indem entweder Constraints entfernt, hinzugefügt oder geändert werden oder Domänen von Variablen eingeschränkt werden. Anschließend wird mit dem Constraint Solver eine Lösung des Systems ermittelt.

Die in Abbildung 2.3 dargestellte Architektur zeigt das Zusammenspiel der verschiedenen Komponenten, die für die Entwicklung und Umsetzung einer Refaktorisierung mit Refacola beteiligt sind.

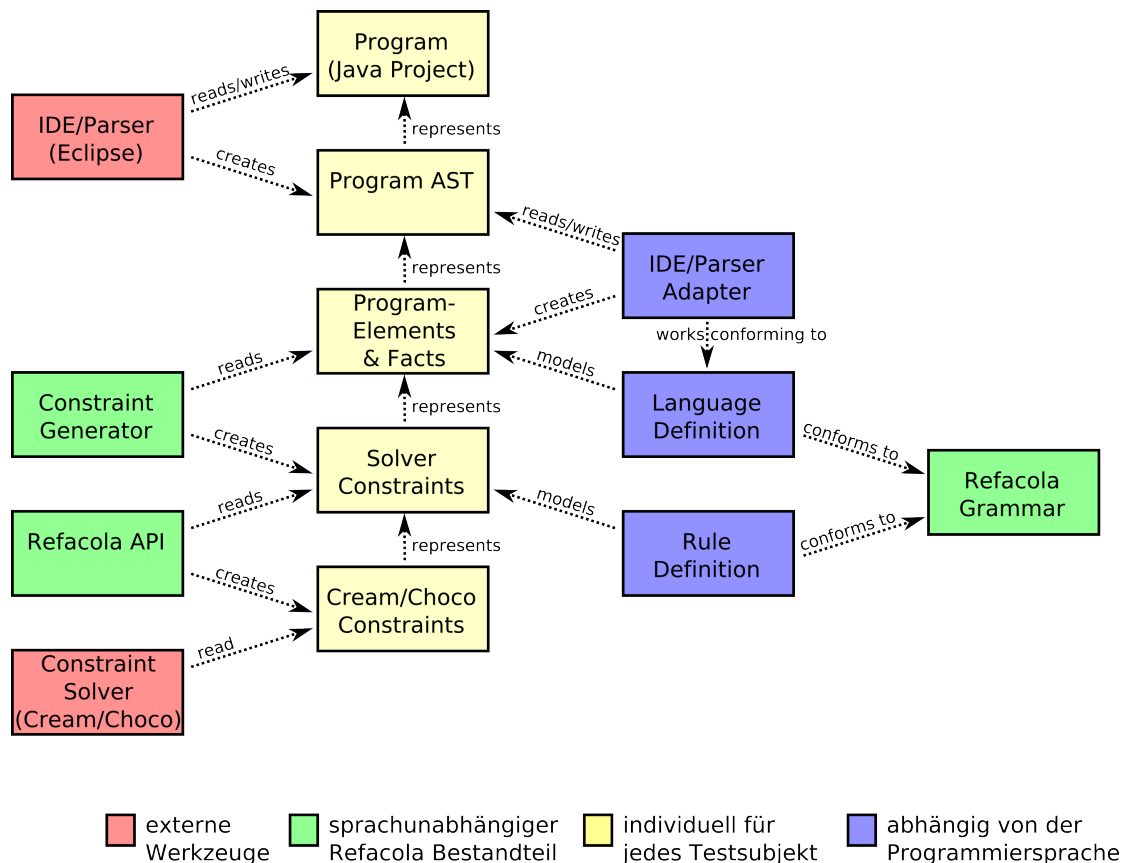


Abbildung 2.3.: Bestandteile der Refacola Architektur, in Anlehnung an (Pil11)

Um den Aufwand für große Programme in Grenzen zu halten und möglichst einfache (mit wenigen Änderungen durchzuführende) Refaktorisierungen zu erhalten, sind gewisse Optimierungen notwendig. So werden beispielsweise nur die Constraints erzeugt, die letztlich auch benötigt werden (siehe den Algorithmus in 2.1). Für die Suche nach einer Lösung wird eine Bewertungsfunktion eingesetzt, die dem Constraint Solver eine Rückmeldung über die Qualität einer gefundenen Lösung gibt.

Die Implementierung von Refacola erfolgt in Java unter Zuhilfenahme der Werkzeuge Xtext und Xpand, eine genauere Beschreibung hierzu findet sich in Abschnitt 2.5. Als

Beispiel für die Sprachunabhängigkeit hat das Team um Refacola auch Implementierungen von Refaktorisierungen für Eiffel und Java erstellt. Im Rahmen dieser Arbeit wird ausschließlich die Implementierung für Java betrachtet.

2.4.1. Darstellung eines Programms mittels Constraints

2.4.1.1. Programmelemente

Ein Programm wird durch zwei Dinge repräsentiert: Zum einen eine Menge von Constraintvariablen und zum anderen eine Menge von Constraints, die diese Variablen in Zusammenhang stellen. Durch die Anwendung von Constraintregeln (engl. Constraint Rules) wird das Programm in diese Form überführt.

Grundsätzlicher Baustein eines jeden Programms ist das sog. Program Element. Je nach Programmiersprache können diese unterschiedliche Ausprägungen haben, bei objektorientierten Sprachen gibt es üblicherweise folgende Arten (Kinds):

- Klassen (Class)
- Felder (Field)
- Methoden (Method)
- Referenzen (Reference)
- lokale Elemente (Locals), lokale Variablen und Parameter

Jedes dieser Programmelemente hat eine Reihe von Eigenschaften (Properties):

- Bezeichner (Identifier)
- Sichtbarkeit (Access modifier oder Accessibility)
- deklarierter Typ (declared Type)
- Klasse, in denen es definiert werden wurde (Owner)
- usw.

Jede Eigenschaft wird durch eine Constraintvariable repräsentiert. Die Domäne dieser Variablen definiert, welche gültigen Belegungen möglich sind. Für die Sichtbarkeit besteht die Domäne beispielsweise aus den Werten *public*, *protected*, *private* und *default*.

2.4.1.2. Verwendung von Domains

Wie bereits beschrieben, handelt es es sich bei einer Domain um den Wertebereich einer Constraintvariable. Dies können Zahlen sein, die z. B. im Fall der Constraintvariablen „accessibility“ anstelle der Schlüsselwörter *public*, *protected*, *private* oder *default* stehen. Für den Identifier ist es jedoch die Menge aller gültigen Bezeichner, die in der jeweiligen Programmiersprache erlaubt sind. Aus diesen Unterschieden ergibt sich, dass die jeweiligen Werte in den Domains zwar typkompatibel sind, aber unterschiedliche Bedeutungen haben, weshalb sie nicht miteinander gemischt werden sollten. In Refacola wird durch eine Typprüfung verhindert, dass beispielsweise ein Bezeichner aus dem Bereich der gültigen Access Modifier ausgewählt wird.

Manche Domains ergeben sich erst aus dem zugrunde liegendem Programm. Das Property „Location“ definiert die Klasse, in der das jeweilige Program Element definiert wurde. Das bedeutet, dass der Wert des Properties selbst ein Program Element ist, wodurch die Domain vom jeweiligen Programm abhängig wird (sog. Program-dependent domain). Da die verfügbaren Constraint Solver verständlicherweise nicht mit Domains arbeiten können, die für Arten von Programmelemente definiert sind, muss man sich hier mit einer Abbildung helfen: Und zwar wird eine injektive Abbildung für jede Art von Program Elements auf spezielle Domains vorgenommen, so dass jedes Programmelemente einem bestimmten Wert der Domain entspricht.

Für manche Domains ist es wichtig, dass die Werte eine besondere Ordnung haben. Dies trifft beispielsweise auf Domains der Art „Klasse“ zu, deren Vererbungshierarchie eine nicht-totale Ordnung darstellen, die dem Constraintlöser in Form einer Ordnungsmatrix bereitgestellt wird.

2.4.1.3. Herleitung von Constraints aus dem Programmtext

Die Umsetzung des Programms in Constraints basiert auf einem eigenen Regelwerk, bestehend aus den sog. Constraintregeln (engl. constraint rules, siehe 2.4.1.4). Jede dieser Regeln besteht aus zwei Teilen, einer Vorbedingung und einem Aktionsteil. Die allgemeine Notation ist in 2.4 dargestellt. Bei der Vorbedingung handelt es sich um einen logische Ausdruck (sog. Program Query), mit dem alle Program Elements ausgewählt werden, auf die dann der Aktionsteil angewandt wird. Die Anwendung des Aktionsteils besteht darin, dass die Constraints, die dort definiert werden, alle passenden Programmelemente erzeugt werden.

$$\frac{\text{Vorbedingung}}{\text{Aktionsteil}}$$

Abbildung 2.4.: Constraintregel

Im Program Query können Variablen eingesetzt werden, die bei einer erfolgreichen

2. Theoretische Grundlagen

Suche in der Faktendatenbank mit den passenden Elementen belegt werden, ähnlich zu der Unifikation von Klauseln bei logischen Programmiersprachen. Die gleichen Variablen können im Aktionsteil benutzt werden, um die erzeugten Constraints abhängig von den gefundenen Elementen zu formulieren. Ein Beispiel für eine solche Regel ist in 2.5 dargestellt, in der die Variablen r und d über das Query 'binds' mit konkreten Programmelementen belegt werden. Der Aktionsteil der Regel definiert dann die Constraints, die diesem Fall gelten. Die Constraints verwenden Attribute (Properties) der Programmelemente, in diesem Fall ι für den 'identifier' (Namen) und τ für den Typ. Im Klartext bedeutet die in 2.5 dargestellte Regel, dass, wenn eine Referenz auf eine Definition zeigt, der Name und der Typ übereinstimmen müssen.

$$\frac{\text{binds}(r,d)}{r.\iota = d.\iota \quad r.\tau = d.\tau}$$

Abbildung 2.5.: Constraintregel für die Bindung von Variablen

Sind die Constraintregeln für eine Programmiersprache vollständig und korrekt definiert und alle Domains nach diesen Regeln für ein gegebenes Programm eingeschränkt, so ist das Ergebnis ein Constraintsystem, dessen Lösungen für sich jede eine gültige Refaktorisierung des ursprünglichen Programms darstellen.

2.4.1.4. Beispiele für Queries

In den Artikeln (ST09) und (SKP11) werden verschiedene Queries erklärt, die verwendet werden, um Constraintregeln aufzustellen. Die wichtigsten davon sind die folgenden:

assignment(R_1, R_2) Die Referenz R_2 wird zugewiesen an Referenz R_1

binds(R, E) Die Referenz R ist an das deklarierte Element E gebunden

receiver(R_1, R_2) Der Zugriff auf ein bestimmtes Feld oder eine Methode R_1 wird von Referenz R_2 'empfangen' (engl. received). Dieses Query ist insbesondere für Fälle interessant, in denen aufgrund der Vererbungshierarchie und Sichtbarkeit einzelner Methoden/Fehler der jeweilige Empfänger (Sub- oder Superklasse) eines Aufrufs davon abhängt, wo die Referenz definiert wurde (siehe Kapitel 2.1 in (ST09)).

2.4.1.5. Beispiele Constraint Rules

Im folgenden werden zwei Constraintregeln aus der Definition der Refacola Implementierung für Java dargestellt, dabei handelt es sich nur um einen kleinen Teil des umfangreichen Regelwerks.

Constraintregel „Name Based Access“ Wird eine Referenz an deklariertes Element gebunden, müssen beide Bezeichner gleich sein.

```
for all
  r : CommonJava.NamedReference
  E : CommonJava.NamedEntity
do
  if
    CommonJava.binds(r, E)
  then
    r.identifizier = E.identifizier
end
```

Constraintregel „Explicit Type Access“ Verweist eine Typreferenz auf einen Typ aus einem anderen Package, muss dieser Typ die Sichtbarkeit „public“ besitzen.

```
for all
  r : CommonJava.TypeReference
  T : CommonJava.TopLevelType
do
  if
    CommonJava.binds(r, T)
  then
    r.hostPackage != T.hostPackage > T.accessibility = #public
end
```

2.4.1.6. Welche Constraints gibt es?

Bei den Constraints, die anhand der Regeln erzeugt werden und die verschiedenen Fakten in Beziehung setzen, werden die folgenden Typen unterschieden:

Composed Constraint sind aus anderen Constraints aufgebaut, welche durch logische Operatoren verknüpft sind. Dabei können die logischen Operatoren *And* (und), *Or* (oder), *Not* (nicht), *If (...) Then (...) Else (...)* (wenn (...) dann (...) sonst (...)) und *Implies* (impliziert) verwendet werden.

Comparison Constraint sind Constraints, die Vergleiche (engl. Comparison) ausdrücken, wobei die Vergleichsoperatoren *Eq* (gleich, engl. equal), *Neq* (ungleich, engl. not equal), *GE* (größer oder gleich, engl. greater or equal), *GT* (größer als, engl. greater than), *LE* (kleiner oder gleich, engl. less or equal), *LT* (kleiner als, engl. lower than) verwendet werden können.

Const Constraint drücken nur Logische Konstanten *True* (Wahr) oder *False* (Falsch) aus.

Relational Constraint drückt die Beziehung zwischen zwei Werten über ein Query aus (siehe 2.4.1.4).

2.4.1.7. Welche Constraints werden erzeugt?

Ziel des Refactoring Frameworks ist es natürlich, die Aufgabe möglichst effizient zu lösen. Das bedeutet, dass je nach gewünschtem Refactoring nur bestimmte Arten von Constraints erzeugt werden und dann auch nur für solche Programmelemente, die von dem Refactoring betroffen sind. Da ein Refactoring in Form von veränderten Properties definiert wird, geht es darum herauszufinden, an welcher Stelle die veränderten Properties vorkommen.

Listing 2.1 zeigt den Algorithmus aus (SKP11), der, ausgehend von den Properties, die über die Refaktorisierung geändert werden müssen (Ziel der Refaktorisierung) und geändert werden dürfen, alle Constraints ermittelt, die Einfluss auf das Ergebnis haben. Dies sind alle Constraints, die die erwähnten Properties direkt oder indirekt beschränken und somit beeinflussen können. Alle anderen Constraints können ignoriert werden.

Des Weiteren hat der Algorithmus die Funktion, die Constraintvariablen mit Werten zu belegen. Die Werte sind entweder die neuen Werte, die das gewünschte Refactoring darstellen, oder die bisherigen Werte, wodurch bestimmte Veränderungen verhindert werden. Oder es ist kein Wert, dann wird dieser durch den Constraint Solver bestimmt.

Der dargestellte Algorithmus arbeitet wie folgt: Er iteriert über eine Liste von Properties, die geändert werden sollen. Anschließend wird geprüft, welche Constraint Rules dieses Property p einschränken. Von jeder Constraint Rule, auf die das zutrifft, werden die Constraints (aus dem Aktionsteil) weiter untersucht, die das Property p einschränken. Nun werden alle in diesem Constraint vorkommenden Properties untersucht. Dabei gibt es drei Fälle: Ist es ein Property, das geändert werden soll, wird der neue Wert zugewiesen. Ist es ein Property, das sich nicht ändern darf, wird der Standardwert zugewiesen. Alle anderen Properties dürfen sich ändern, in diesem Fall wird das entsprechende Property in die Liste der zu ändernden Properties eingefügt, so dass in einem späteren Durchlauf alle weiteren Abhängigkeiten untersucht werden.

Algorithm GenerateConstraints

Input:

M , the set of properties whose values must change

N , the set of properties whose values may change

v_0 , a function mapping properties to their initial values

v_f , a function mapping the members of M to their new values

R , a set of constraint rules

Output:

C , a set of constraints

Steps:

2. Theoretische Grundlagen

```
1. let  $P = M, Q = \emptyset, C = \emptyset$ 
2. repeat
3.   move one property  $p$  from  $P$  to  $Q$ 
4.   for each constraint rule  $r$  in  $R$ 
5.     if  $r$  can constrain  $p$ 
6.       for each constraint  $c$  generated by  $r$ , constraining  $p$ 
7.         for each property  $p'$  occurring in  $c$ 
8.           if  $p' \in M$ 
9.             replace every occurrence of  $p'$  in  $c$  with  $vf(p')$ 
10.          else if  $p' \notin N$ 
11.            replace every occurrence of  $p'$  in  $c$  with  $v_0(p')$ 
12.          else if  $p' \notin Q$ 
13.            add  $p'$  to  $P$ 
14.          add  $c$  to  $C$ 
15. until  $P = \emptyset$ 
```

Listing 2.1: Ermittlung der notwendigen Constraints (aus (SKP11))

2.4.2. Finden der Lösung

Es genügt nicht, eine beliebige Lösung der Constraints zu ermitteln, die Lösung soll auch dem Anspruch genügen, dass es sich um die kleinstmögliche Veränderung am Programm handelt, um das gewünschte Ergebnis zu erhalten. Es handelt sich somit um ein sog. Constraint Optimization Problem (COP). Um dies zu erreichen, wird der Constraint Solver eine Bewertungsfunktion verwenden, um die gefundenen Lösungen miteinander zu vergleichen. Die Bewertungsfunktion wird in Form sog. „penalties“ individuell zusammen mit jedem Refactoring definiert und basiert auf Strafpunkten, die für jede zusätzliche (unnötige) Veränderung vergeben werden. Ziel ist es, eine Refaktorisierung zu finden, die möglichst direkt die gewünschte Änderung umsetzt und demzufolge einen niedrigen Wert für „penalties“ erreicht. Zum Ermitteln der besten Lösung greifen die verwendeten Solver auf das „Branch and Bound“-Verfahren zurück, dargestellt in Abschnitt 2.3.3.1.

2.4.3. Zurückschreiben der Constraints in den Programmtext

Nachdem die Refaktorisierung angewandt wurde, d.h. für das Constraintsystem mit den geänderten Properties wurde von einem Constraint Solver eine optimale Lösung ermittelt, müssen die Änderungen zurück in den Quelltext geschrieben werden. Die genaue Methode ist abhängig von der konkreten Implementierung, deshalb wird im folgenden die Vorgehensweise exemplarisch für Java beschrieben: Die veränderten Properties sind jeweils einem Programmelement zugehörig. Dieses Programmelement wiederum repräsentiert einen Knoten im Abstrakten Syntax Baum (Abstract Syntax Tree, AST)

und der AST kann direkt auf den Quellcode übertragen werden. Aufgabe der Implementierung ist es nun, die Änderungen im AST durchzuführen, der letzte Schritt wird von der Eclipse Java Umgebung übernommen.

2.4.4. Definition von Refactorings

Vorraussetzung für die Definition eines Refactorings in Refacola ist, dass eine Sprachspezifikation für die Zielsprache existiert und alle Constraint Rules definiert wurden, um die notwendigen Constraints zu erzeugen. Sind diese Voraussetzungen gegeben, kann definiert werden, welche Properties geändert werden müssen. Dabei wird zwischen zwei Arten unterschieden: Properties, deren Änderung erzwungen wird (engl. forced), und Properties, deren Änderung erlaubt wird (engl. allowed). Alle anderen Properties sind vor Änderungen geschützt. In 2.2 ist eine Definition eines Refactorings zu sehen, das ein Programmelement umbenennt. Zunächst werden mit dem Befehl `import` andere Dateien eingebunden, welche die Sprachspezifikation und Constraintregeln enthalten. Anschließend wird mit den Befehlen `refactoring` ein Name für das vorliegende Refactoring definiert, mit `language` die Zielsprache, in diesem Fall Java, festgelegt, und mit `uses` die zu verwendenden Constraintregeln (in gruppierter Form) angegeben.

In den Zeilen 9f. werden die erzwungenen Änderungen definiert, in diesem Fall der neue Name (repräsentiert durch das Property `identifier`, siehe 2.4.1.1), und in den Zeilen 11f. die erlaubten Änderungen, konkret die Namen anderer Programmelemente. Andere Properties (die z.B. die Sichtbarkeit betreffen) sind nicht genannt und bleiben daher von dem Refactoring in jedem Fall unverändert.

Die Zeilen 13ff. enthalten die Definition, wie der Constraint Solver gefundene Lösungen bewerten soll.

```
1 import "CommonJava.language.refacola"
2 import "Accessibility.ruleset.refacola"
3 import "Location.ruleset.refacola"
4
5 refactoring RefactoringName
6 language CommonJava
7 uses Accessibility, Location
8
9 forced
10  identifier of CommonJava.NamedEntity
11 allowed
12  identifier of CommonJava.NamedEntity
13 penalties
14  changedIdentifier "identifier should not be changed"
15    for all
16      d: CommonJava.NamedEntity
17    penalize
```

```
18      d.identifier != d.identifier  initialwith 1
```

Listing 2.2: Definition eines „Rename“ Refactorings in Refacola

2.4.5. Ablauf von Refactorings

Im folgenden wird der Ablauf der in Listing 2.2 dargestellten Refaktorisierung, dem Umbenennen eines Elements, dargestellt:

Erzwungene Änderung: Ein „identifier“ Property (der zu ändernde Name eines Programmelements)

Mögliche Änderungen: Alle anderen „identifier“ Properties (von Referenzen), sonstige Properties wie z.B. „access modifier“ oder „location“ sind nicht definiert und werden demzufolge auch nicht verändert.

Nun kommt der Algorithmus aus Listing 2.1 zum Einsatz, der alle von dieser Änderung betroffenen Constraints filtert.

Das Refactoring wird dann in folgenden Schritten durchgeführt:

1. Die Sprachdefinition liegt vor, in der die Konstrukte einer Sprache (Regeln aus der Sprachspezifikation) definiert sind, so dass diese in anderen Schritten entsprechend berücksichtigt werden.
2. Der Quelltext wird in eine Faktendatenbank eingelesen. Dies kann beispielsweise über einen AST als Zwischenschritt erfolgen, der im Fall der Eclipse Entwicklungsumgebung für den importieren Quelltext automatisch zur Verfügung steht.
3. Auf die Faktendatenbank werden die Queries angewandt, woraus Constraints erzeugt werden (siehe 2.4.1.5).
4. Die gewünschte Refaktorisierung wird gesteuert, indem ein bestimmtes „forced“ Property (der „identifier“ des Programmelements, das umbenannt werden soll) auf einen neuen Wert (den neuen Namen) gesetzt wird (Zeile 9).
5. Allen anderen „identifier“ Properties, den sog. „allowed“ Properties, wird der gesamte Wertebereich ihrer Domain (inklusive ihres initialen Werts aus dem AST) als gültige Wertebelegung zugewiesen (Zeile 11).

6. Auf Grundlage der Constraints wird mittels eines Constraint Solvers (siehe 2.3) eine Lösung gesucht, in der jedem Property ein gültiger Wert zugewiesen wird. Jede Lösung entspricht einer gültigen Refaktorisierung, die nach bestimmten Regeln bewertet wird: Wie in Zeile 13 definiert wird jeder Fall, in dem ein „identifier“ Property geändert wurde, mit einem Strafpunkt bewertet. Gesucht ist die Refaktorisierung mit der besten Bewertung (demzufolge mit den wenigsten Änderungen).
7. Aus der besten Lösung wird die Belegung der Properties zurück in den AST übertragen, wodurch sie auch im Quelltext geändert werden.

2.5. Programmiersprachenentwicklung mit Xtext und Xpand

Xtext und Xpand sind Werkzeuge, mit denen man auf schnelle und unkomplizierte Weise eine eigene Programmiersprache zu entwickeln kann. Angefangen bei einfachen Sprachen für spezielle Anwendungen (sog. domänenspezifische Sprachen, engl. Domain Specific Language, DSL) bis zu vollwertige Programmiersprachen bietet sich ein breites Einsatzspektrum.

Als domänenspezifische Sprache bezeichnet man Sprachen, die nur auf ein bestimmtes Anwendungsgebiet fokussiert sind und deren Grammatik, Schlüsselwörter und Formulierung dadurch eine möglichst intuitive einfache Formulierung für dieses Problemfeld erlauben. Den Gegensatz dazu stellen Allzwecksprachen (engl. General Purpose Language, GPL) wie C++ oder Java dar, die zwar für alle Probleme eingesetzt werden können, jedoch nicht immer die einfachste und beste Lösung darstellen, in Bezug auf Wartbarkeit und Programumfang.

Die Anwendungsgebiete sind vielfältig, angefangen bei eingebetteten Systemen bis hin zu Unternehmensanwendungen werden Xtext-basierte Sprachen verwendet um über Codegeneratoren, wie z.B. Xpand, Code in Java, C++, Objective C, C#, Python o.a. zu erzeugen, der auf verschiedensten Plattformen kompiliert werden kann.

Die Vorgehensweise bei der Entwicklung mit Xtext und Xpand ist, dass in der Xtext-eigenen Sprache die verschiedenen Aspekte der Zielsprache beschrieben werden. Hieraus generiert Xtend dann Parser, AST, semantisches Modell, sowie ggf. Code Generator oder Interpreter für die Java Laufzeitumgebung. Außerdem werden Plugins für Eclipse erzeugt, die einen eigenen Projekttyp und Editor mit Code Highlighting und Auto Completion bereitstellen. Mit Xpand wird anschließend die Codeerzeugung definiert, also welche Ausgabe aus dem eingelesenen Programm erzeugt werden soll, und geschieht mittels sog. Templates, also Vorlagen, die als Grundlage dienen und mit Informationen aus dem Modell gefüllt werden. Das Template kann z. B. ein Gerüst aus Java Code

sein, in das die konkreten Felder und Methoden automatisiert eingefügt werden, so dass der erzeugte Code anschließend zu einem funktionierenden Programm kompiliert werden kann.

```
grammar de.feueps.refacola.dsl.Refacola with org.eclipse.xtext.common.
    Terminals

generate refacola "http://de.feueps.refacola.dsl.refacola"

RefacolaDefinition:
    (imports+=Import) *
    (languageDef=LanguageDefinition |
    ruleset=RuleSet |
    refactoring=Refactoring);

Import:
    "import" importURI=STRING;
LanguageDefinition:
    "language" name=ID
    "kinds" (kinds+=KindDecl) *
    "properties" (propertyDecls+=PropertyDecl) *
    "domains" (domainDecls+=DomainDecl) *
    ("macros" (macros+=MacroDecl) +) ?
    "queries" (queryDecls+=QueryDecl) *
    ("penalties" (penaltyDecl+=PenaltyDecl) +) ?
;

RuleSet:
    "ruleset" name=ID
    "language" language=[LanguageDefinition]
    "rules" (rules+=Rule) *
    ("macros" (macros+=MacroDecl) +) ?;

(...)
```

Listing 2.3: Auszug aus der Refacola Grammatik

2.5.1. Die Entwicklung der Refacola Sprachdefinition mit Xtext

Für Refacola wurde Xtext verwendet um die Sprache umzusetzen, in der die `.refacola` Dateien formuliert werden, welche die Definitionen der Zielsprache, Constraintregeln und Refaktorisierungen beinhalten. Grundlage hierfür ist die in Listing 2.3 dargestellte Spezifikation der Grammatik `refacola.xtext`. Die erste Zeile definiert den Namen der Sprachdefinition mit einer existierenden Xtext Vorlage als Ausgangspunkt. In dieser Vorlage sind grundlegende Dinge wie Zeilenumbrüche und Kommentare bereits geregelt, und zwar auf die Art und Weise, wie sie bei Java Verwendung findet. Wäre dies

2. Theoretische Grundlagen

nicht gewünscht, könnte man es an dieser Stelle selbst definieren. Anschließend wird der Name für die Sprache festgelegt und danach folgt die Syntax. Dort ist festgelegt, dass eine Refacola Definition zunächst mit beliebig vielen Import-Statements anfängt. Es folgt entweder eine Sprachdefinition (Language Definition), eine Constraintregeldefinition (Ruleset) oder die Definition eines Refactorings. Die nächsten Zeilen spezifizieren dann im Detail, wie Import, Sprachdefinition und Refactoring auszusehen haben. Auf Grundlage dieser Grammatik werden ein Parser und ein Modell generiert. Mit dem Parser können dann Refacola-Dateien eingelesen werden, die entsprechend der Grammatik formuliert sind, und daraus ein AST erzeugt werden. Ein gekürztes Beispiel für die Sprache Java ist in Listing 2.4 dargestellt. Aus solch einer Spezifikation wird das Modell als sog. EMF (Eclipse Modeling Framework) Modell erzeugt, eine sehr häufig verwendete Technologie, die eine einfache Weiterverwendung mit anderen Werkzeugen ermöglicht. Für die Elemente des EMF Modells werden eigene Javaklassen erzeugt, so dass die Handhabung sehr einfach und intuitiv ist.

```
language CommonJava
kinds

    abstract Reference <: REFERENCE
    abstract NamedReference <: Reference { identifier }
    abstract TypeReference <: NamedReference { hostPackage }
    abstract ReferenceInType <: Reference { owner, towner, hostPackage }

    abstract Entity <: ENTITY
    abstract DeclaredEntity <: Entity { hostPackage, towner }
    abstract NamedEntity <: Entity { identifier }
    abstract AccessibleEntity <: Entity { accessibility }

(...)

properties
    identifier "\\iota": Identifier
    accessibility "\\alpha": AccessModifier
    owner "\\chi": Type ""
    towner "\\lambda": LocationType ""
    hostPackage "\\pi": LocationPackage "location of a reference, i.e. a
        Class"

(...)

domains
    AccessModifier = {private, package, protected, public}
    LocationType = [ TopLevelType ] "location of a reference, i.e. its
        containing class"
    LocationPackage = [ Package ] "location of a reference, i.e. its
        containing class"
```

```
Type = [ Type ]  
  
queries  
  
  binds(reference: Reference, entity: Entity) "reference binds to  
    entity"  
  
(...)
```

Listing 2.4: Auszug aus der Java-Sprachdefinition für Refacola

2.5.2. Die Erzeugung der Refacola Implementierung mit Xpand

Nachdem `.refacola` Dateien eingelesen und aus ihnen EMF Modelle erstellt werden können, in denen die Regeln für Zielsprache und Constraint-erzeugung spezifiziert sind, muss aus diesen Regeln Code erzeugt werden, der die Anweisungen ausführt. Diese Code-erzeugung erfolgt mit dem Werkzeug Xpand, das den Constraint-Generator und eine Repräsentation der Zielsprache mit vordefinierten Templates erzeugt. Die Templates für die Zielsprachen Eiffel und Java wurden von dem Refacola Team als Anwendungsbeispiel implementiert. Die Templates, die bereits für Refacola erstellt wurden, sind theoretisch bereits ausreichend, um alle Klassen zu generieren, die benötigt werden, um anhand einer gegebenen Faktendatenbank Refaktorisierungen durchzuführen. Manuell implementiert werden müssen die Faktendatenbank sowie das Einlesen von Programmen in die Datenbank und das Schreiben der Datenbank zurück als Programme. Dies wurde bereits als Anwendungsbeispiel für Java und Eiffel bereits getan.

Listing 2.5 stellt einen Auszug aus dem Template dar, mit dem eine Javaklasse erzeugt wird, die ein Property repräsentiert. Das Template enthält den Code für die Java Klasse, der jedoch mit Steuerbefehlen erweitert ist. Diese Steuerbefehle sind mit umgekehrten französischen Anführungszeichen (sog. Guillemets, « und ») markiert. Bevor der Code in eine Klasse geschrieben wird, werden diese Befehle ausgeführt, wodurch z. B. Schleifen und Bedingungen realisiert sind, die sich dann in der Ausgabe widerspiegeln. Ein Auszug aus dem Code, der mit diesem Template generiert wurde, ist in Listing 2.6 dargestellt.

«REM»

```
Copyright (c) 2010 FernUniversitaet in Hagen  
All rights reserved. This program and the accompanying materials  
are made available under the terms of the Eclipse Public License v1.0  
which accompanies this distribution, and is available at  
http://www.eclipse.org/legal/eplv10.html
```

Contributors:

2. Theoretische Grundlagen

```
Jens von Pilgrim initial implementation
<<ENDREM>>
<<IMPORT de::feu::ps::refacola::dsl::refacola>>

<<EXTENSION templates::Extensions>>

<<REM>>
*****

Class templates for PropertyDecl
*****
<<ENDREM>>
<<DEFINE class FOR PropertyDecl>>
<<LET basePackagePath() + "/props/" + this.className(this) + ".java" AS
currentCtx>>
<<FILE currentCtx >>
<<EXPAND Common::javaFileHeader FOR this >>
package <<basePackage()>>.props;

<<EXPAND _Import::imports(className(currentCtx)) FOR importedTypes (
currentCtx) ONFILECLOSE>>

<<importApiClass(currentCtx, "language.AbstractProperty") >>
/**
 * Property "<<name>>"<<IF doc.length>0>: <<doc>><<ENDIF>>
 *
 * <p>Original refacola definition:
 * <pre>
 * <<this.prettyPrintJavaDoc()>>
 * </pre>
 * </p>
 *
 * <p>Template: src/templates/javaAPI/PropertyDecl_java.xpt</p>
 * <p>Model: <<EXPAND Common::modelFile FOR this>> </p>
 * @generated
 */
public class <<this.className(currentCtx) >>
extends AbstractProperty<<this.domain.domainElementTypeName (
currentCtx)>> {

/**
 * {@inheritDoc}
 * @return <<IF {"ID_DOMAIN_IDENTIFIERS", "ID_DOMAIN_BOOLEAN"}.contains
(this.domain.idConst())>>
<<importApiClass(currentCtx, "language.LanguagePackage") >>
LanguagePackage.<<this.domain.idConst()>>
<<ELSE >>
```

2. Theoretische Grundlagen

```
        <<getLanguageDefinition().packageName(currentCtx) >>.<<this.
            domain.idConst()>>
    <<ENDIF>>
    * @see de.feup.ps.refacola.api.language.AbstractProperty#getDomainID
      ()
    * @generated
    */
    @Override
    protected LanguageElementID getDomainID() {
        return
            <<IF {"ID_DOMAIN_IDENTIFIER", "ID_DOMAIN_BOOLEAN"}.contains(this.
                domain.idConst())>>
                <<importApiClass(currentCtx, "language.LanguagePackage") >>
                LanguagePackage.<<this.domain.idConst()>>
            <<ELSE >>
                <<getLanguageDefinition().packageName(currentCtx) >>.<<this.
                    domain.idConst()>>
            <<ENDIF>>;
    }
}

(...)
```

Listing 2.5: Auszug aus dem Xpand Template für Properties

```
(...)
package de.feup.ps.refacola.language.commonjava.props;

import de.feup.ps.refacola.api.language.AbstractProperty;
(...)

/**
 * Property "accessibility"
 *
 * <p>Original refacola definition:
 * <pre>
 * accessibility "\\alpha": AccessModifier
 * </pre>
 * </p>
 *
 * <p>Template: src/templates/javaAPI/PropertyDecl_java.xpt</p>
 * <p>Model: CommonJava.language.refacola </p>
 * @generated
 */
public class RefacolaAccessibility
    extends
        AbstractProperty<RefacolaAccessModifierLiteral> {

    /**
```

2. Theoretische Grundlagen

```
* {@inheritDoc}
* @return      CommonJavaPackage.ID_DOMAIN_ACCESSMODIFIER
* @see de.feu.ps.refacola.api.language.AbstractProperty#getDomainID
*   ()
* @generated
*/
@Override
protected LanguageElementID getDomainID() {
    return CommonJavaPackage.ID_DOMAIN_ACCESSMODIFIER;
}

(...)
}
```

Listing 2.6: Mit dem Template erzeugter Java Code

3. Praktische Umsetzung

3.1. Planung

3.1.1. Anforderungen

Die Implementierung soll eine Möglichkeit bieten, um die Mechanismen von Refacola für Java auf eine automatisierte Art und Weise anhand einer bestimmten Auswahl von Testprogrammen zu testen. Die Vorgehensweise soll dabei wie folgt sein:

1. Einlesen des Java Code
2. Erstellung der Faktendatenbank
3. Erzeugung der Constraints anhand der Constraint Rules
4. Einschränkung der Domains (definiert den Umfang der Tests)
5. Ermittlung einer Lösung für alle Constraints
6. Auswahl einer zufälligen Lösung
7. Änderung des des Java Code entsprechend der Lösung (Refaktorisierung)
8. Prüfung auf syntaktische Fehler (Compilerfehler)
9. Prüfung auf semantische Fehler (mit Hilfe von JUnit Tests)
10. Wiederholung ab Schritt 6 mit einer anderen Lösung

Fragen, die dabei beantwortet werden müssen, sind:

Welche und wie viele Refaktorisierungen werden durchgeführt? Nur einzelne Refaktorisierungen zu testen, ergibt für eine frühe Testphase Sinn, in der grundlegende Fehler noch nicht ausgeschlossen werden können. Auf diese Weise können Fehler einfacher isoliert und ihre Ursache schneller lokalisiert werden. Treten soweit keine Fehler mehr auf, kann die Anzahl der aufeinander folgenden Änderungen erhöht werden. Dadurch entstehen möglicherweise unvorhergesehene Wechselwirkungen, die zu einem Fehlverhalten

3. Praktische Umsetzung

führen, und die im Einzeltest nicht erkennbar waren. Um diese verschiedenen Einsatzbereiche zu unterstützen, könnte die Anzahl der Änderungen als Parameter vorgesehen werden, der vom Benutzer festgelegt werden kann. Von dieser Funktion wurde jedoch wieder Abstand genommen, da einerseits nach jeder Refaktorisierung die Faktengenerierung, Erzeugung der Constraints und Berechnung der Lösungen durchgeführt werden müsste, wodurch der Rechenaufwand enorm steigt. Andererseits wären Fehler, die erst nach einer großen Anzahl von Refaktorisierungen auftreten, für die Fehlersuche sehr schwer zu reproduzieren.

Wie werden die Refaktorisierungen ausgewählt? Es ist davon auszugehen, dass der Constraint Solver eine große Anzahl an Lösungen ermittelt, so dass nicht jede einzelne Lösung getestet werden kann. Aus diesem Grund soll eine repräsentative Auswahl getroffen werden, in dem eine Teilmenge der Lösungen nach dem Zufallsprinzip ausgewählt wird. Die Anzahl der auszuwählenden Lösungen muss in Abhängigkeit von den gefundenen Lösungen frei konfigurierbar sein.

Wie wird verifiziert, dass das Programm nach der Refaktorisierung noch syntaktisch und semantisch korrekt ist? Die syntaktische Korrektheit wird durch eine fehlerfreie Kompilierung geprüft. Für die semantische Korrektheit sind zusätzliche Tests, z.B. in Form von JUnit Testsuiten notwendig, wobei eine hohe Testabdeckung dieser Testsuiten wichtig ist. Für jedes Testprojekt werden JUnit-Testsuites auswählbar sein, die im Anschluss an eine Änderung ausgeführt werden.

Welche Testergebnisse werden auf welche Art präsentiert? Da ein Teil der Testparameter durch Zufallswerte bestimmt werden, ist es notwendig, dass zumindest im Nachhinein eine Auswertung erzeugt wird, anhand derer die Testabdeckung bestimmt werden kann. Dieser Bericht sollte die Programmelemente und Propertytypen auflisten, die verändert wurden. Des Weiteren ist es für die Fehlersuche wichtig, dass aufgezeichnet wird, welche Fehler bei den Prüfungen aufgetreten sind, so dass eine Fehlersuche vereinfacht wird.

3.2. Implementierung

3.2.1. Implementierung mittels eines Eclipse Plugins

Die Umsetzung des Constraint Testers erfolgt auf Basis von Plugins für die Entwicklungsumgebung Eclipse. Geeigneter Ausgangspunkt für eine intuitive Bedienung stellt das sog. Launch Framework dar, das in der Standarddistribution zur Java Entwicklung mit Eclipse mitgeliefert wird. Auf diese Weise können die Constraintregel Tests ähnlich

3. Praktische Umsetzung

gestartet werden wie JUnit Tests. Das Launch Framework bietet eine einheitliche Benutzungsoberfläche die leicht erweitert werden kann. Funktionen, die auf diese Weise verwendet werden können ist beispielsweise die Verwaltung von Testkonfigurationen.

Die Testergebnisse werden in sog. Views (Ansichten) dargestellt, die beim Starten der Tests automatisch eingeblendet werden. Dort kann der Benutzer einerseits die Statistiken zu den Tests (Anzahl der generierten Fakten, Anzahl der Tests, Anzahl der fehlgeschlagenen Tests, usw.) ablesen, andererseits auch einen einzelnen Test detailliert betrachten. Diese detaillierte Betrachtung bietet einerseits die Auflistung der Änderungen, die am Originalprogramm vorgenommen wurden, und andererseits Liste der Fehler, die während der syntaktischen und semantischen Prüfung aufgetreten sind. Die nächsten Abschnitte erläutern die Entwicklung der Plugins für Eclipse.

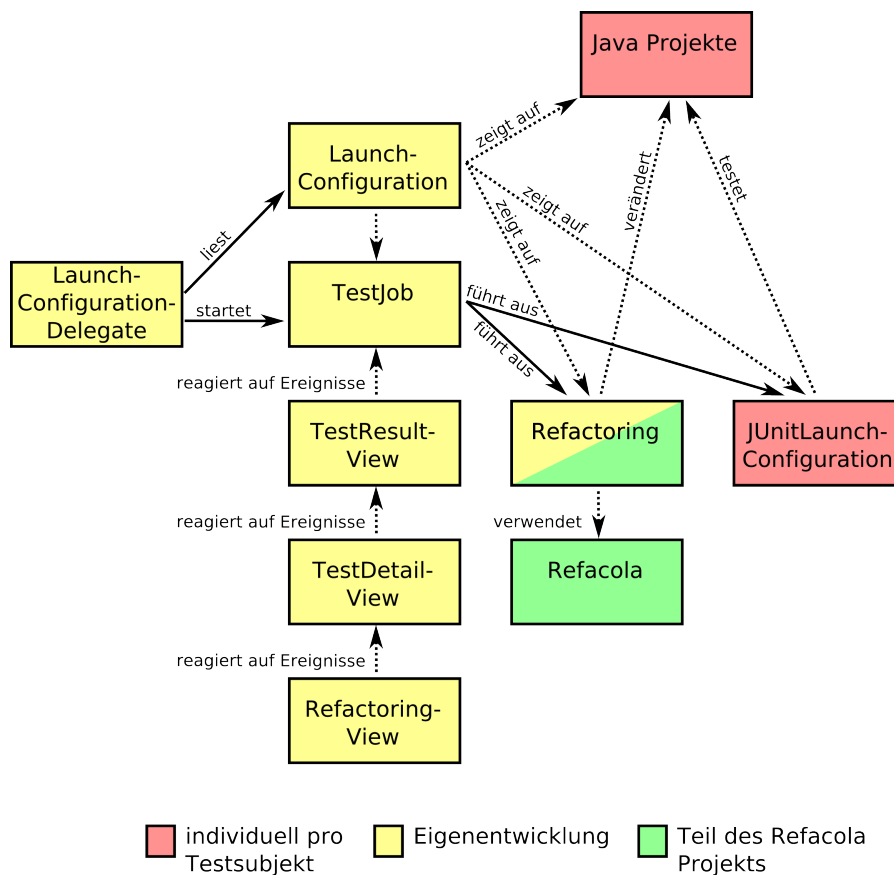


Abbildung 3.1.: Übersicht der Bestandteile des Constraint Testers

3.2.2. Einbindung in Eclipse

3.2.2.1. Das Eclipse Launch Framework

Das Launch Framework wurde mit Version 2.0 von Eclipse eingeführt, um für das Starten von Anwendungen aus der Entwicklungsumgebung heraus eine einheitliche, leicht erweiterbare Infrastruktur zu bieten.

Es besteht aus zwei Teilen, einerseits die Kernfunktionalität und andererseits die Benutzungsschnittstelle (engl. User Interface, UI).

Die wichtigsten Bestandteile seitens der Kernfunktion sind die verschiedenen Typen an Konfigurationen, sog. `LaunchConfigurationTypes`, die das generelle Verhalten für eine Startkonfiguration definieren, und die konkrete Konfiguration, sog. `LaunchConfiguration`, die individuelle, vom Benutzer festgelegte Einstellungen enthält. Für eine eigene Erweiterung ist ein neuer `LaunchConfigurationType` in Form einer Klasse zu entwickeln, die das Interface `ILaunchConfigurationDelegate` implementiert. Bei der `LaunchConfiguration` ist keine eigene Erweiterung notwendig, da es sich bereits um eine generische Implementierung handelt, die Werte vom Typ `String` speichert. Wenn sie als Parameter übergeben wird, dann implementiert sie die Interfaces `ILaunchConfiguration` (nur Lesen) oder `ILaunchConfigurationWorkingCopy` (Lesen und Schreiben).

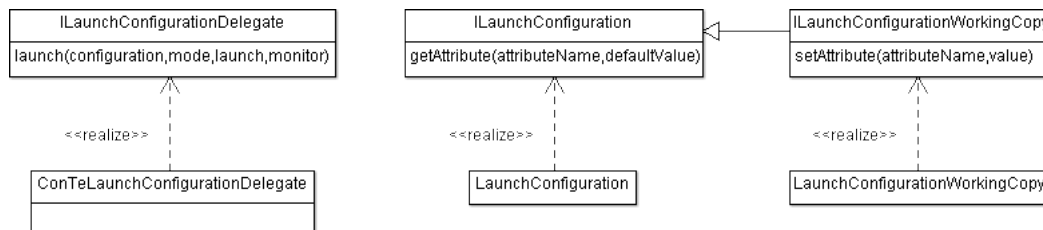


Abbildung 3.2.: Klassenhierarchie der Launch Configuration (Auszug an Methoden)

Die Benutzungsoberfläche des Launch Frameworks besteht im wesentlichen aus einem Dialog, dem `LaunchConfigurationDialog`, in dem neue Konfigurationen unterschiedlicher Typen angelegt, bearbeitet und gelöscht werden können. Wie in Abbildung 3.4 gezeigt, werden auf der linken Seite des Dialogs in einer Baumstruktur die verschiedenen Typen aufgelistet, die angehängten Knoten stellen einzelne Konfigurationen des jeweiligen Typs dar. Die Einstellungen für die gerade ausgewählte Konfiguration werden auf der rechten Seite angezeigt. Je nach Typ variieren die Karteireiter (Tabs) in der oberen Hälfte.

Für jeden Typ der `LaunchConfiguration` gibt es Gruppen von Tabs (engl. `Tabgroups`). Diese Gruppen müssen das Interface `ILaunchConfigurationTabGroup` implementieren und die einzelnen Tabs das Interface `ILaunchConfigurationTab`.

3. Praktische Umsetzung

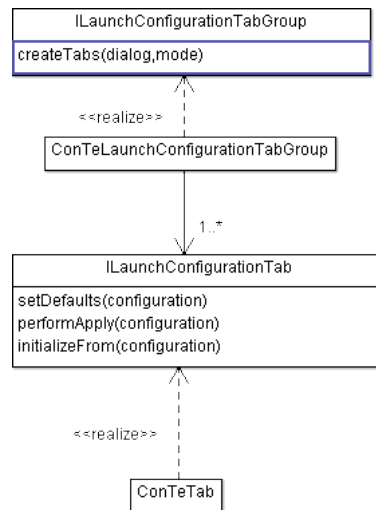


Abbildung 3.3.: Klassenhierarchie der Benutzeroberfläche (Auszug an Methoden)

3.2.2.2. Die Ansicht der Testergebnisse (Test Result View)

Bei der Test Result View handelt es sich um eine Tabelle, die als Übersicht die Werte „Project Name“ (Name des getesteten Projekts), „Duration“ (Zeitdauer um die notwendigen Programm Elemente, Fakten und Constraints zu erzeugen), „Elements“, „Facts“, „Constraints“ (Anzahl der erzeugten Programm Element, Fakten und Constraints), „Solutions“ (Anzahl der berechneten Lösungen, je nach Einstellung begrenzt), „Tested Solutions“ (Anzahl der getesteten Lösungen) und „Failed Tests“ (Anzahl der fehlgeschlagenen Tests) dargestellt sind.

Die Tabelle aktualisiert sich selbsttätig, sobald die Tests für ein Projekt abgeschlossen sind. Wird eine Zeile ausgewählt, können die Details in der „Test Detail View“ betrachtet werden.

3.2.2.3. Die Ansicht der Details zu einem Test (Test Detail View)

In der Test Detail View werden alle Tests aufgelistet, die für ein Projekt durchgeführt wurden. Je nachdem ob die Tests erfolgreich waren oder nicht, werden sie mit einem grünen Haken oder roten Kreuz gekennzeichnet.

Die Darstellung ist eine Baumstruktur, um Details zu einem einzelnen Test zu erfahren, kann dieser Knoten aufgeklappt werden. Die darunter aufgehängenen Knoten sind zum einen die Änderung (Change) im Eclipse-eigenen Format, wie sie vom Refactoring Werkzeug ausgeführt werden. Danach folgt die Sammlung an Änderungen (sog. Change-Set), wie es von Refacola berechnet wurde, anschließend der Status der Refaktorisierung mit etwaiger Fehlermeldung, dann die Fehlermeldungen (Marker), die mit der refaktorierten Lösung aufgetreten sind und zuletzt das Ergebnis der ausgeführten JUnit Tests (sofern welche ausgewählt wurden). Die ersten beiden Knoten, die Änderungen,

3. Praktische Umsetzung

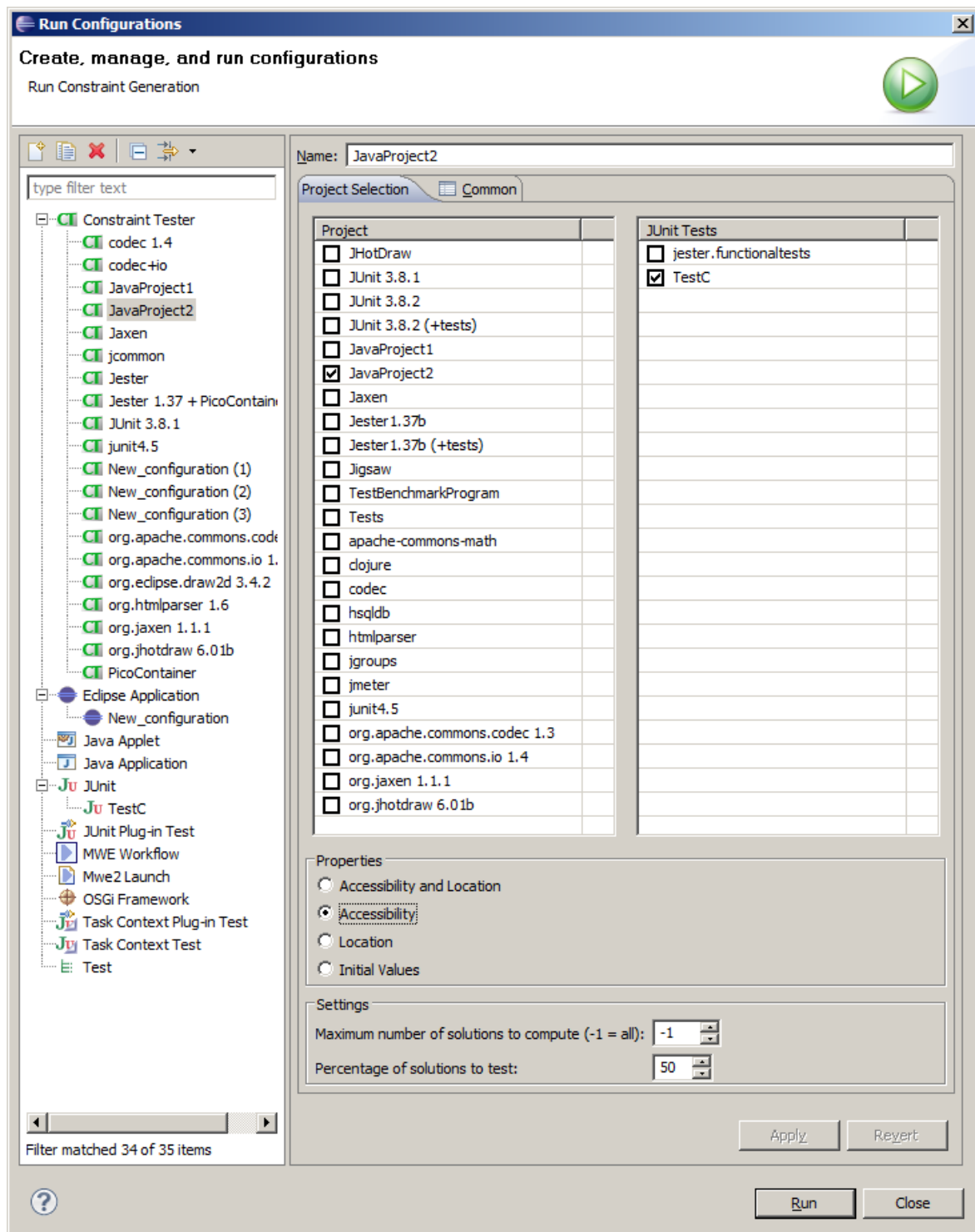


Abbildung 3.4.: Bildschirmfoto des LaunchConfigurationDialog

3. Praktische Umsetzung

enthalten die Einzelschritte als Unterknoten. Im Falle der Änderung im Eclipse-eigenen Format können die Änderungen konkret im Quelltext betrachtet werden, wenn man auf den jeweiligen Knoten klickt. Die Darstellung erfolgt dann in der „Refactoring View“.

3.2.2.4. Die Ansicht der Programmänderungen (Refactoring View)

Die Refactoring View stellt einzelne Änderungen dar, die vorgenommen wurden, um eine berechnete Lösung zu testen. Die Darstellung erfolgt auf die gleiche Art, wie auch die Vorschau für Refaktorisierungen erzeugt wird: In einer zweigeteilten Ansicht wird links der ursprüngliche Programmtext und rechts der veränderte Programmtest angezeigt, wobei die Änderungen farblich hervorgehoben sind.

3.2.3. Architektur

3.2.3.1. Hierarchie der Packages

Der entwickelte Constraint Tester besteht aus den folgenden Packages:

de.feu.ps.conte Dieses Package enthält nur die sog. Aktivatorklasse, die für das Starten und Stoppen des Plugins zuständig ist. In diesem Fall handelt es sich um eine generierte Klasse, die das Plugin als Singleton (siehe (GHJV94)) implementiert.

de.feu.ps.conte.actions In diesem Package befindet sich die Aktion, die im Kontextmenü von Javaprojekten angezeigt wird. Es handelt sich dabei um das direkte Starten des Constraint Tester mit dem selektierten Projekt, wobei entweder eine existierende Startkonfiguration verwendet wird oder, wenn keine gefunden werden konnte, eine neue Konfiguration erzeugt wird.

de.feu.ps.conte.junit Der sog. Listener (siehe (GHJV94)), der beim JUnit Framework registriert wird, um über das Resultat von JUnit Testdurchläufen informiert zu werden, wurde in dieses Package gelegt.

de.feu.ps.conte.launch Klassen, die zwar mit der Ausführung der Tests zu tun haben aber unabhängig von der Benutzungsoberfläche (UI) sind, befinden sich in diesem Package.

de.feu.ps.conte.model Zur Darstellung der ermittelten Testergebnisse wurden eigene Datenstrukturen (Modelle) entwickelt, die in diesem Package liegen.

de.feu.ps.conte.refactorings Für die Refaktorisierungen gibt es unterschiedliche Konfigurationen, die sich in der Auswahl der betroffenen Properties (Accessibility, Location) unterscheiden. Diese Konfigurationen befinden sich in diesem Package.

de.feu.ps.conte.ui.launch Dieses Package enthält die Klassen, die für die Testkonfiguration und -ausführung in Zusammenhang mit der UI zuständig sind, beispielsweise für die Startkonfigurationen.

de.feu.ps.conte.ui.view In diesem Package befinden sich die zur Darstellung der Testergebnisse verwendeten Fenster (Views).

3.2.4. Die Ausführung im Detail

3.2.4.1. Konfiguration und Start des Tests

Wie bereits beschrieben findet die Konfiguration des Tests in Form einer sog. `LaunchConfigurationType` statt, in der alle Einstellungen in Form von `Strings` enthalten sind. Wird der Test gestartet, wird die Konfiguration an den `ConTeLaunchConfigurationDelegate` übergeben. Diese Klasse ist dafür zuständig, einen neue Aufgabe (sog. Job) in Form der Klasse `TestJob` anzulegen und die Konfigurationsparameter (nach entsprechender Umwandlung) zu übergeben. Anschließend wird der Job für die baldmöglichste Ausführung eingeplant. Die Klasse `TestJob` implementiert die Logik, nach der die folgenden Schritte ausgeführt werden, dabei wird über die Liste von Javaprojekten iteriert und für jedes Projekt werden mit der Refacola Java Implementierung mehrere Refaktorisierungen berechnet, wie im nächsten Abschnitt beschrieben wird.

3.2.4.2. Verwendung der Refacola Java Implementierung

Der Constraint Tester verwendet die Refacola Implementierung auf ganz ähnliche Weise, wie sie auch ein Refaktorisierungswerkzeug verwenden würde. Unterschiedlich ist jedoch einerseits, dass keine konkrete Refaktorisierung gefordert wird und andererseits, dass alle Constraints erzeugt werden, nicht nur die für eine einzelne Refaktorisierung benötigten. Nichtsdestotrotz werden Subklassen des Typs `AbstractRefactoring` verwendet, um die Veränderungen vorzunehmen. Der Ablauf ist dabei wie folgt: Dem sog. `ProgramInfoProvider` wird zunächst das aktuelle Projekt übergeben, anhand dessen Programmelemente und Fakten erzeugt werden. Anschließend werden die Domains für die jeweiligen Properties festgelegt. Dabei gibt es drei Möglichkeiten: Entweder wird der ganze Bereich der Domain zugelassen, nur der initiale Wert (aus dem Quelltext) wird zugelassen (damit wird keine Änderung vorgenommen) oder nur ein ganz bestimmter Wert (der die gewünschte Änderung der Refaktorisierung darstellt). Da für die Tests keine bestimmte Refaktorisierung gewünscht ist, werden die Domains größtenteils komplett zugelassen, Einschränkungen erfolgen nur zugunsten der Geschwindigkeit (durch große Domains steigt auch die Anzahl der möglichen Lösungen und damit die Berechnungsdauer). Im nächsten Schritt werden all diese Parameter (Programmelemente, Fakten, Domains) zusammen mit einem Regelsatz für die Programmiersprache Java dem

3. Praktische Umsetzung

`ConstraintGenerator` übergeben, der daraus `Constraints` erzeugt. Anhand dieser `Constraints` berechnet nun der `Constraint Solver` alle Lösungen. Die notwendigen Änderungen, die notwendig sind, um eine Lösung auf den aktuellen Quelltext zu übertragen, werden als sog. `Changes` in einem `ChangeSet` zusammengefasst, somit entspricht die Liste von `ChangeSets` im Ergebnis je einer Lösung. Nachdem also alle Lösungen bekannt sind, erfolgt die Testphase: Je nach Voreinstellungen werden einzelne Lösungen (in Form von `ChangeSets`) ausgewählt. Für jedes `ChangeSet` werden aus den Änderungen im Refacolaspezifischen Format die Änderungen am AST des Ursprungsprojekts im Eclipse Format generiert. Diese Änderungen können nun direkt angewandt werden und das Javaprojekt somit der Lösung entsprechend geändert werden werden.

3.2.4.3. Durchführung der Tests

Die Änderungen, die im vorhergehenden Schritt erzeugt wurden, können direkt ausgeführt werden, wodurch das Javaprojekt verändert wird. Ausgelöst dadurch wird das Projekt von Eclipse erneut kompiliert (in der Standardeinstellung), wodurch auch eine syntaktische Prüfung erfolgt. Fehler, die hierbei auftreten, werden für das Testergebnis gesichert. Anschließend werden die JUnit Tests gestartet, die in der `LaunchConfiguration` ausgewählt wurden. Der `Constraint Tester` registriert sich beim JUnit Framework als sog. `Listener`, wodurch er über die Testergebnisse informiert wird. Diese Ergebnisse werden ebenfalls zwischengespeichert. Nachdem alle Tests abgeschlossen sind, wird das Ergebnis in der `Test Result View` dargestellt. Dann werden die vorgenommenen Änderungen wieder rückgängig gemacht, so dass sich das Javaprojekt im Ursprungszustand befindet. Anschließend kann die nächste Lösung getestet werden. Die gesammelten Ergebnisse und Fehlermeldungen werden in der `Test Detail View` angezeigt, wenn der Benutzer auf eine Zeile mit Testergebnissen klickt.

3. Praktische Umsetzung

The screenshot displays the Eclipse IDE interface during a refactoring process. The main editor shows the original source code for class C, which has been refactored into classes A and B. The 'Refactoring View' window provides a side-by-side comparison of the original and refactored source code. The 'Test Detail View' window shows the results of the refactoring, including a 'Pull Up Field' operation and a 'ChangeSet' for accessibility changes. The 'Test Result View' window at the bottom shows the test results for the refactored code, indicating that all tests passed successfully.

```
package y;

public class C {

    public Object j = 0;
    public Object i = j;

    public Object getI() {
        return i;
    }
}
```

```
Original Source
package y;

public class A {
}

class B extends A {

    public Object j = 0;
    public Object i = j;

    Object getI() {
        return i;
    }
}

Refactored Source
package y;

class A {
}

public class B extends A {

    public Object j = 0;
    public Object i = j;

    Object getI() {
        return i;
    }
}
```

Test Results:

- OK
- NOK
- Pull Up Field <org.eclipse.jdt.core.refactoring.CompilationUnitChange@15d3352/> <org.eclipse.jdt.core.refactoring.CompilationUnitChange: /JavaProject2/src/y/A.java>
- Change: /JavaProject2/src/y/C.java
- ChangeSet: TopLevelClass[y_A_A].y_A_A.accessibility -> package; TopLevelClass[y_A_B].y_A_B.accessibility -> public; TopLevelClass[y_C_C].y_C_C.accessibility -> package
- TopLevelClass[y_A_A].y_A_A.accessibility -> package
- TopLevelClass[y_C_C].y_C_C.accessibility -> package
- TopLevelClass[y_A_B].y_A_B.accessibility -> public
- Refactoring Status: <OK>
- Markers
- The public type B must be defined in its own file
- JUnit Test Sessions
- TestC
- TestSuite: test.TestC : Completed - OK (2)
- TestCase: test.TestC.testGetI : Completed - OK
- TestCase: test.TestC.testGetI2 : Completed - OK
- TestC
- NOK

Project Name	Duration	Elements	Facts	Constraints	Solutions	Tested Solutions	Failed Tests
JavaProject2	5574 ms	47	161	150	8	4	3

Abbildung 3.5.: Darstellung des Testergebnisses

4. Durchführung der Tests

4.1. Planung

Ziel der Tests ist es, Fehler in den Constraint Regeln aufzudecken. Dadurch werden aber zwangsläufig auch die Refacola Implementierung sowie die Sprachdefinition für Java und ihre Implementierung getestet. Bei der Entscheidung, wie die Testfälle zu gestaltet sind, stehen folgende Variationsmöglichkeiten zur Verfügung:

- Umfang des zu Quelltexts, der für die Refaktorisierung herangezogen wird. Je umfangreicher das Programm ist, desto größer wird der Berechnungsaufwand. Gleichzeitig steigt aber auch die Komplexität und damit die Wahrscheinlichkeit, Fehler zu finden.
- Auswahl der Properties, die für die Refaktorisierung geändert werden dürfen. Die aktuelle Refacola Implementierung für Java beschränkt sich noch auf die Properties Accessibility und Location. Daraus ergeben sich drei Kombinationen: Beide gleichzeitig, nur Accessibility oder nur Location.
- Test der semantischen Korrektheit: Anzahl und Testabdeckung der JUnit Tests sind vom jeweiligen Projekt abhängig.

Das ganze Spektrum der oben genannten Variationen abzudecken war jedoch im Rahmen dieser Arbeit nicht möglich, da aufgrund der in 4.2.1 dargestellten fehlgeschlagenen Tests Einschränkungen vorgenommen werden mussten. Diese Einschränkungen sind wie folgt:

- Bei einem Großteil der Testprojekte treten entweder bereits während der Generierung der Fakten oder später beim Erzeugen der Constraints und der Lösungssuche noch unterschiedliche Fehler auf, die auf die Refacola Implementierung zurückzuführen sind.
- Es können nicht beide Properties Accessibility und Location gleichzeitig betrachtet werden, da dies zum einen mit dem verfügbaren Arbeitsspeicher des Testcomputers nicht durchführbar ist. In den folgenden Tests wird deshalb ausschließlich das Property Accessibility als unbelegte Constraintvariable verwendet.

Die Testdurchführung erfolgt in zwei Schritten: Zunächst wird ein Test mit jedem der verfügbaren Projekte durchgeführt um ihren Umfang abzuschätzen und zu prüfen, ob dabei Probleme auftreten, die eine weitere Verwendung verhindern. Anschließend werden mit einer kleinen Auswahl der Projekte, mit denen ohne Fehler getestet werden konnten, zusätzliche Variationen getestet.

4.1.1. Aufgetretene Probleme und ihre Lösung

4.1.1.1. OutOfMemoryError: Java heap space

Ein weitere Begrenzung ist der Umfang der zu testenden Projekte. Je umfangreicher das Constraint System, desto größer ist auch der Speicherbedarf, der beim Lösen benötigt wird. Wie sich herausgestellt hat, werden von dem Constraint Solver eine sehr große Anzahl kurzlebiger Objektinstanzen angelegt, wodurch die Größe des Heaps sehr schnell ansteigt und dann an ihre Grenzen stößt. Der in der Oracle Java per Default genutzte Strategie für den Garbage Collector konnte den Heap leider nicht schnell genug aufräumen, so dass hier schnell eine OutOfMemoryException auftrat.

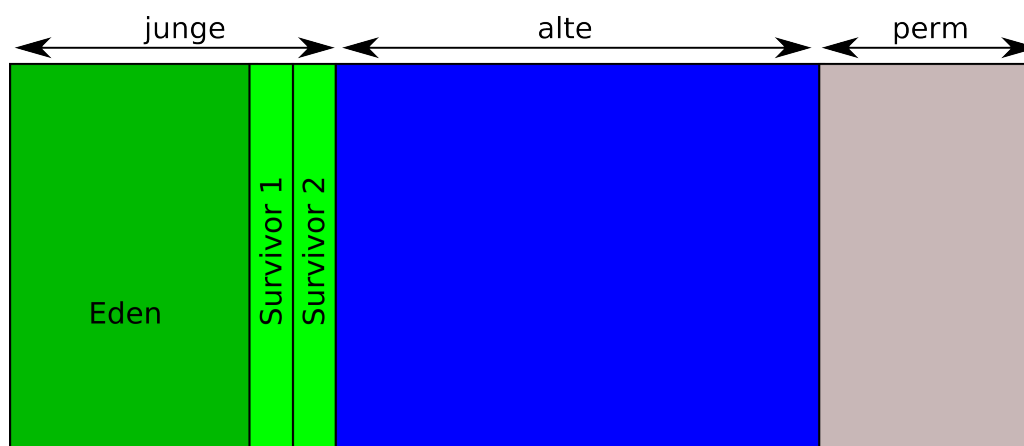


Abbildung 4.1.: Aufteilung des Heap Speichers (nach (SM03))

Für dieses Problem gibt es verschiedene Lösungen: Einerseits muss der Heap vergrößert werden, da er in der Standardeinstellung lediglich 64 MB groß ist. Hier wurde mit dem Parameter „-Xmx768m“ ein Wert von 768 MB gesetzt. Andererseits ist der Heap in verschiedene Abschnitte aufgeteilt, in denen die Objektinstanzen je nach Alter verteilt werden (siehe Abbildung 4.1). Dies liegt darin begründet, dass typischerweise hauptsächlich junge Objekte wieder „sterben“, d.h. nicht mehr referenziert werden und damit entfernt werden können (man spricht auch von „toten Objekten“). Um die Garbage Collection möglichst effizient zu gestalten, werden also häufiger die jungen Objekte in den Bereichen „Eden“ und „Survivor“ überprüft (dies wird als „Minor Collection“), und seltener alle Objekte („Major Collection“).

4. Durchführung der Tests

Zunächst werden alle neuen Objekte in dem Bereich „Eden“ angelegt. Je einer der Bereiche „Survivor 1“ und „Survivor 2“ ist abwechselnd leer. Ist der „Eden“ voll, prüft eine „Minor Collection“ die Bereiche „Eden“ und den nicht-leeren Bereich „Survivor 1“ oder „Survivor 2“ auf tote Objekte. Alle lebenden Objekte werden dann in den noch leeren Bereich „Survivor 1“ oder „Survivor 2“ kopiert. Nun wird der Speicherplatz des anderen „Survivor“ Bereich freigegeben und damit als leer betrachtet, wodurch sich beide Bereiche abwechseln. Hat ein Objekt eine gewisse Zeitspanne des Hin- und Herkopierens zwischen den beiden „Survivor“ Bereichen überstanden, wird es letztlich in den Bereich der alten Objekte kopiert, in dem seltener eine Garbage Collection stattfindet.

Die Dimensionierung des Bereichs „Eden“ ist ein Kompromiss zwischen der Häufigkeit, wie oft eine „Minor Collection“ notwendig ist, dem Speicher, der dadurch belegt wird und der Zeit, die für die Collection beansprucht wird.

Ist der Heap sehr groß und damit auch der Bereich „Eden“, dauert es eine Weile, bis eine Garbage Collection stattfindet. Durch die große Anzahl der angesammelten Objekte ist dann auch der Aufwand sehr groß, den gesamten Bereich zu überprüfen. Ist der Bereich „Eden“ und die davon abhängigen „Survivor“ Bereiche sehr klein, kann es passieren, dass der Platz in einem der „Survivor“ Bereiche nicht ausreicht, um alle lebenden Objekte aus dem anderen „Survivor“ Bereich und dem „Eden“ Bereich zu kopieren, wodurch eine „Major Collection“ ausgelöst wird. Dies bedeutet einen viel größeren Aufwand, den es zu vermeiden gilt.

Wie zu Anfang erwähnt, war das konkrete Problem mit dem Standard Garbage Collector, dass der Constraint Solver sehr große Mengen an kurzlebigen Objekten erzeugt. Auf diese Weise sammeln sich viele tote Objekte an, so dass der verfügbare Heap schnell ausgelastet ist und ein `OutOfMemoryError` auftritt.

Erster Schritt zur Lösung war der Einsatz der Strategie „ParallelGC“, bei dem die Garbage Collection auf parallele Threads verteilt wird. Dies ist bei Computern mit mehreren Prozessoren sehr vorteilhaft. Außerdem soll diese Strategie im Falle vieler junger Objekte (SM03) zufolge besonders effizient sein. Die Aktivierung erfolgt in Form des Parameters „-XX:+UseParallelGC“ für die virtuelle Maschine (sog. Java Virtual Machine, JVM). Nichtsdestotrotz ist der Aufwand für den Garbage Collector sehr groß, weshalb auch die Zeit, die er beansprucht, sehr lang ist. Die JVM hat einen Sicherheitsmechanismus, der verhindern soll, dass der Garbage Collector zu häufig und zu lange läuft, da dies durch einen Fehler im Programm ausgelöst werden kann (z.B. in einer Endlosschleife). Im Falle des Constraint Solvers ist es jedoch ein zulässiger Zustand, so dass diese Mechanismus mit dem JVM Parameter „-XX:-UseGCOverheadLimit“ deaktiviert werden muss.

Zusätzlich wurde die Bereiche „Eden“ und „Survivor“ verkleinert, so dass der Garbage Collector häufiger eine „Minor Collection“ durchführt.

Im Ergebnis hat sich gezeigt, dass der Aufwand für die Garbage Collection steigt, dafür die Speicherauslastung nur halb so hoch ist, wie in Abbildung 4.2 dargestellt.

4. Durchführung der Tests

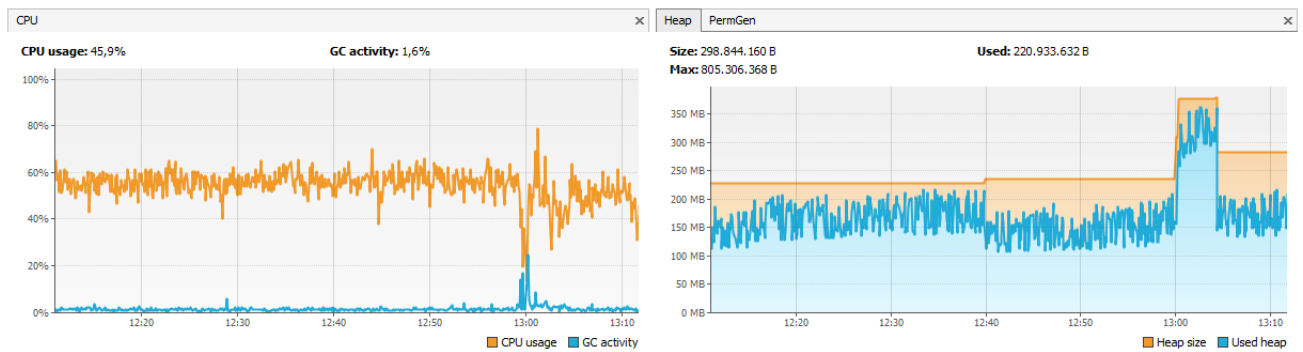


Abbildung 4.2.: CPU und Heap Auslastung mit manuellen GC Parametern

4.1.1.2. OutOfMemoryError: PermGen space

Eine andere Fehlermeldung, die während der Testdurchläufe in Bezug auf den Heap Speicher auftritt, lautet „OutOfMemoryError: PermGen space“. Bei dem sog. „PermGen Space“ handelt es sich um den in Abbildung 4.1 auf der rechten Seite dargestellten grauen Bereich. In ihm speichert die virtuelle Maschine ihre internen Daten, insbesondere alle geladenen Javaklassen. Werden sehr viele Klassen geladen, kann es geschehen, dass der maximal verfügbare Speicherbereich nicht ausreicht. Mit dem Parameter „-XX:MaxPermSize=256m“ wird die maximale Größe des Speicherbereichs auf die Größe von 256MB (der Standardwert ist 64MB) begrenzt.

4.1.1.3. IllegalAccessError: tried to access class X from class Y

Für die Durchführung der semantischen Tests wurden die JUnit Testfälle zunächst in einem getrennten Projekt gespeichert, so dass die Tests nicht den Umfang der Faktengenerierung unnötigerweise vergrößerten und auch nicht refaktorisiert wurden. Dies hatte jedoch zur Folge, dass die Zugriffe der JUnit Tests auf einzelne Klassen nicht als Constraints abgebildet wurde. Dadurch wurde bei manchen Refaktorisierungen die Sichtbarkeit von diesen Klassen eingeschränkt und bei der Ausführung der Tests trat die Exception „IllegalAccessError: tried to access class X from class Y“ auf. Um dieses Problem zu lösen, wurden die JUnit Tests in das zu testende Projekt integriert. Die Unterschiede bei der Anzahl der Programmelemente, Fakten, Constraints und Lösungen können in den ersten beiden Zeilen der Tabelle 4.1 abgelesen werden.

4.2. Durchführung und gefundene Fehler

Der Tabelle 4.1 ist zu entnehmen, welche Projekte getestet wurde und welche Daten dabei ermittelt wurden. Aufgetretene Fehler sind ebenso notiert und werden im Anschluss erläutert.

Tabelle 4.1.: Auswertung der Testprojekte

Projekt	P.Elemente	Fakten	Constraints	Lösungen	Dauer Fakten [s]	Dauer insg. [s]
Apache Commons Codec 1.3	2.189	5.027	9.594	147.456	44,9	67,6
Apache Commons Codec 1.3 + JUnit	11.532	27.661	58.269	144	8,6	2039,4
Apache Commons IO 1.4	5.876	13.058	Fehler ²		25,3	–
Apache Commons Math	49.417	139.372	244.598	0 ¹	39,9	57043,1
Apache Ivy	85.495	196.694	OOM ⁶		73,7	–
Clojure	AssertionFailedException bei der Faktengenerierung ⁵					
Eclipse Draw2d 3.4.2	Fehler bei der Faktengenerierung ⁴					
HTML Parser 1.6	21.961	51.152	Fehler ²		39,6	–
JavaProject2	47	161	150	8	0,2	0,4
Jaxen 1.1.1	12.244	31.626	Fehler ²		24,8	–
Jaxen	24.279	58.064	OOM ⁶		17,0	–
Jester1.37b	2.359	5.433	10.401	0 ¹	4,3	50,8
JHotdraw	36.619	91.512	OOM ⁶		28,4	–
JHotdraw 6.01b	30.983	71.904	137.943	0 ¹	13,1	11389,8
JMeter	53.509	133.902	OOM ⁶		180,5	–
JUnit 3.8.1	6.314	14.319	28.572	0 ¹	390,9	405,8
JUnit 3.8.2	Fehler bei der Faktengenerierung ⁴					
JUnit 4.5	ClassCastException ³					
PicoContainer	5.493	12.187	25.553	0 ¹	11,1	–
servngxml-1.1.2	99.874	282.016	OOM ⁶		749,9	–
XOM 1.2.5	51.990	112.391	OOM ⁶		35,8	–

¹ Der Constraint Solver findet keine Lösung, siehe 4.2.1.5² ArrayIndexOutOfBoundsException, siehe 4.2.1.1³ ClassCastException, siehe 4.2.1.3⁴ Errors during fact generation, siehe 4.2.1.2⁵ AssertionFailedException, siehe 4.2.1.4⁶ OutOfMemoryError: Die Testmaschine verfügte nicht über ausreichend Arbeitsspeicher, siehe auch 4.1.1.2

4. Durchführung der Tests

Tabelle 4.2.: Testergebnisse

Projekt	Constraints	Lösungen	getestet	Fehler
Apache Commons Codec 1.3	9.594	147.456	1.475	546 ¹
Apache Commons Codec 1.3 + JUnit	58.269	144	144	12 ²
JavaProject1	72	8	8	0
JavaProject2	150	8	8	1 ³

¹ Die JUnit Tests wurden bei den Constraints nicht berücksichtigt, siehe 4.1.1.3

² Importierte Klasse ist nicht mehr sichtbar (The type ... is not visible), siehe 4.2.1.7

³ In einer Datei wurde mehr als eine Klasse als *public* gekennzeichnet, siehe 4.2.1.6

4.2.1. Gefundene Fehler

4.2.1.1. `ArrayIndexOutOfBoundsException`

Abhängig vom Testprojekt tritt beim Lösen der Constraints ein Fehler des Typs „`ArrayIndexOutOfBoundsException`“ auf (siehe Listing A.1 im Anhang). Die Ursache für diesen Fehler ist derzeit noch unbekannt und eine Lösung war im zeitlichen Rahmen dieser Arbeit nicht möglich, weshalb die betroffenen Projekte für weitere Tests nicht verwendet werden konnten.

4.2.1.2. Fehler während der Faktengenerierung

Bei manchen Projekten trat der unten gezeigte Fehler während der Faktengenerierung auf, wobei die Fehlermeldung keinen Rückschluss auf die genaue Ursache zulässt. Die betroffenen Projekte wurden von den Tests ausgeschlossen.

```
java.lang.Error: Errors during fact generation
  at de.feu.ps.refacola.language.commonjava.factbase.CommonJavaFactbase
    .<init>(CommonJavaFactbase.java:86)
  at de.feu.ps.conte.launch.TestJob.run(TestJob.java:114)
  at org.eclipse.core.internal.jobs.Worker.run(Worker.java:54)
```

Listing 4.1: Stacktrace

4.2.1.3. `ClassCastException`

Bei drei Projekten konnten die Fakten erfolgreich generiert werden, jedoch trat die Exception aus Listing A.2 beim Lösung der Constraints auf. Dem Stacktrace ist zu entnehmen, dass der Fehler beim traversieren des AST eintritt, weil der Typ `RefacolaStaticMemberClassImpl` nicht zum Typen `RefacolaTopLevelType` gecastet werden kann.

4.2.1.4. **AssertionFailedException**

Der Fehler aus Listing A.3 trat ausschließlich bei dem Testprojekt „Clojure“ während der Faktengenerierung auf.

4.2.1.5. **Keine Lösung für das Constraint System**

Der Fehler, dass der Constraint Solver für manche Projekte keine Lösung findet, war mehrmals zu beobachten. Auch wenn alle Properties mit ihren Ursprungswerten aus dem Programmtext belegt wurden, konnte der Constraintsolver keine Lösung ermitteln. Dies stellt ebenfalls einen Fehler in der Refacola Implementierung dar, denn unter der Annahme, dass das Programm keine syntaktischen Fehler hat, muss zumindest die ursprüngliche Form immer als gültige Lösung gefunden werden.

4.2.1.6. **Mehrere Klassen mit Sichtbarkeit „public“ in einer Datei**

In der Java Spezifikation ist geregelt, dass jede Klasse mit der Sichtbarkeit „public“ in einer Datei gespeichert sein muss, die den Namen der Klasse trägt. Außerdem ist es erlaubt, dass mehrere Klassen, die nicht ineinander verschachtelt sind (sog. Top-Level-Klassen), in einer gemeinsame .java Datei enthalten sind. Durch die erste Regel ergibt sich jedoch die Einschränkung, dass nur eine dieser Klassen die Sichtbarkeit „public“ haben kann und alle anderen die Sichtbarkeit „package“ besitzen müssen (siehe (SM05), Abschnitt 7.6). Diese Regeln werden derzeit noch nicht als Constraints abgebildet, weshalb es vorkommen kann, dass mehr als einer Klasse die Sichtbarkeit „public“ zugewiesen wird, weshalb der Compiler einen syntaktischen Fehler bemängelt.

4.2.1.7. **Sichtbarkeit von importierten Klassen**

Im Testprojekt „Apache Commons Codec 1.4“ wurde die Sichtbarkeit der Klasse „BinaryEncoderAbstractTest“ im package „tests.apache.commons.codec“ von public auf package reduziert. Dies führte zu dem Fehler „The type tests.apache.commons.codec.BinaryEncoderAbstractTest is not visible“, der für die Klasse „AllTests“ im package „tests“ angezeigt wird. Die Ursache für diesen Fehler mag darin liegen, dass die Klasse selbst nicht verwendet sondern nur importiert wird, weshalb die Abhängigkeit nicht durch Constraints abgebildet wurde.

4.2.2. **Variationen mit anderen Properties**

Da mit dem Testprojekt „Apache Commons Codec 1.3“ (mit und ohne JUnit Tests) keine Probleme aufgetreten sind, war geplant, mit diesem Projekt weitere Tests durchzuführen, bei denen nur für das *Location* Property oder für *Location* und *Accessibility* gleichzeitig neue Werte gesucht werden. Jedoch ist dabei der in 4.2.1.1 dargestellte Fehler

4. Durchführung der Tests

aufgetreten, so dass diese Tests nicht durchgeführt werden konnten. Mit dem Testprojekt „JavaProjekt2“ trat dieser Fehler zwar nicht auf, jedoch wurde dadurch die Berechnung so aufwendig, dass beide Tests aus Zeitgründen nach über 24 Stunden abgebrochen werden mussten, um andere Tests durchführen zu können. Schon alleine für die in Listing 4.2 dargestellte Klasse dauerte die Berechnung der Lösungen für Belegungen der *Location* Properties ca. 24 Minuten.

```
package some.thing;

public class MyClass {

    Object i = 1;
    Object j = i;
}

class MyClass2 extends MyClass {

}
```

Listing 4.2: Einfache Java Klassen

5. Fazit und Ausblick

Wie in der Einleitung bereits erläutert, steigt mit dem Umfang von Softwareprojekten auch der Bedarf an Werkzeugen, die zuverlässig umfangreiche Refaktorisierungen durchführen können. Und je komplexer eine Refaktorisierung ist, desto wertvoller wird die Werkzeugunterstützung für den Entwickler, da er sie nicht manuell ausführen muss. Der herkömmliche Weg, solche Refaktorisierungen auf imperative Weise zu beschreiben, sind insbesondere bei vielen Sonderfällen fehlerträchtig. Eine Lösung bietet hier Refacola, bei dem es sich um ein sehr interessantes und innovatives Konzept handelt, um die Qualität von Refaktorisierungen zukünftig zu verbessern, indem es erlaubt, constraintbasierte Refaktorisierungen auf deklarative Weise zu definieren. Eine wichtige Voraussetzung für die anschließende korrekte Refaktorisierung ist jedoch, dass die Definition der Constraintregeln in Bezug auf die Sprachspezifikation vollständig und korrekt sind. Und die Korrektheit dieser Regeln auf automatisierte Weise zu Testen war die Grundidee des im Rahmen dieser Arbeit entwickelten Werkzeugs.

Die Entwicklung dieses Constraint Testers fand zeitlich parallel zu der Entwicklung von Refacola und der darauf basierenden Java Implementierung statt. Dies hatte einerseits den Vorteil, dass durch den Zugriff auf das Code Repository der Entwicklungsprozess schrittweise beobachtet und die Entwicklung der Architektur mitverfolgt werden konnte. Andererseits fehlte dadurch aber auch eine voll funktionale Implementierung und fest definierte Schnittstellen, auf denen der Constraint Tester hätte aufbauen können, denn beides entwickelte sich erst im Laufe der Zeit.

Dem jungen Alter der Refacola Implementierung ist geschuldet, dass bei den Testdurchläufen auch Fehler gefunden wurden, die nicht im ursprünglich angedachten Testgebiet, den Constraint Regeln, liegen. So wurden auch Fehler entdeckt, die noch in der Refacola Implementierung zu korrigieren sind.

Während der Entwicklungs- und Testphase haben sich verschiedene Möglichkeiten gezeigt, die sich als Verbesserung und Erweiterung für die Zukunft anbieten:

Gegenwärtig sind die zu testenden Properties fest im Constraint-Tester programmiert. Hier bietet sich dann, dass die betroffenen Teile in Zukunft automatisch mittels XPand Template aus der Refacola Definition generiert werden. Auf diese Weise wäre der Constraint-Tester immer in der Lage, die Belegung der Constraintvariablen für alle definierten Properties zu testen.

Die Belegung der Constraintvariable erfolgt derzeit in der Reihenfolge, wie sie von der Domain vorgegeben werden. Um für alle Werte sicherzustellen, dass sie mit der

5. Fazit und Ausblick

gleichen Wahrscheinlichkeit getestet werden, ist es deshalb notwendig, dass zunächst alle Lösungen berechnet und daraus dann zufällig Lösungen ausgesucht werden. Würden die Werte der Domain bereits zufällig sortiert werden, wäre die gleichmäßige Auswahl der Werte bereits sichergestellt und die Anzahl der zu berechnenden Lösungen könnte reduziert werden.

Aus zeitlichen Gründen musste der Test auf einen Constraint Solver („Choco 2“) beschränkt werden. Refacola bietet jedoch durch eine Abstrahierungsschicht die Möglichkeit, auch andere Constraint Solver wie z. B. „Cream“ zu verwenden.

Während der Testphase war der verwendete Testcomputer eine starke Einschränkung, da die Generierung der Fakten und Constraints sowie die Ermittlung der Lösungen sehr rechenintensiv ist. Hier wäre einerseits ein schnellerer Computer hilfreich, andererseits zeigte die Auslastung des Prozessors, dass nur einer der beiden Prozessorkerne aktiv war. Demzufolge liegt der Schluss nahe, dass der verwendete Constraint Solver nicht parallel arbeiten kann. Eine Verteilung der Arbeit auf parallele Prozesse (sog. Multithreading) würde gerade bei modernen Computern, die über vier und mehr Prozessorkerne verfügen, einen großen Geschwindigkeitsvorteil bringen. Dadurch, dass die Refacola Implementierung auf externe Constraint Solver zurückgreift, ist hier einerseits zwar nur eine geringe Einflussnahme möglich, aber andererseits fällt es so leichter, einen neuen Constraint Solver einzubinden, sollten sich dadurch Vorteile wie Multithreading ergeben. Eine andere Möglichkeiten, um die Durchführung der Tests zu beschleunigen, ist, den Test so zu gestalten, dass er von mehreren Rechnern gleichzeitig durchgeführt werden kann. Es existieren Open Source Projekte wie z. B. Apache Hadoop, die Werkzeuge entwickeln, mit denen die Durchführung von verteilten Berechnungen vereinfacht wird.

Bei einem Teil der verwendeten Testprojekte traten während der Faktengenerierung oder Lösungsberechnung Fehler auf, so dass diese Projekte leider nicht für weitergehende Tests verwendet werden konnten. Sind diese Fehler behoben, können auch diese Projekte wieder für weitere Tests herangezogen werden. Es ist davon auszugehen, dass diese Projekte noch ungetestete Sprachkonstrukte enthalten, wodurch sich die Testabdeckung verbessern würde. Die Implementierung des Constraint Testers stellt die notwendigen Funktionen bereit, so dass weiterführende Tests durchgeführt werden können.

A. Stacktraces der gefundenen Fehler

```
java.lang.Error: java.lang.ArrayIndexOutOfBoundsException: 585
at de.feu.ps.refacola.api.refactorings.AbstractRefactoring.
    runRefactoring(AbstractRefactoring.java:474)
at de.feu.ps.refacola.api.refactorings.AbstractRefactoring.execute(
    AbstractRefactoring.java:269)
at de.feu.ps.conte.launch.TestJob.run(TestJob.java:139)
at org.eclipse.core.internal.jobs.Worker.run(Worker.java:54)
Caused by: java.lang.ArrayIndexOutOfBoundsException: 585
at de.feu.ps.refacola.api.domains.DomainUtils.
    relationsToOrderMapWarshall(DomainUtils.java:164)
at de.feu.ps.refacola.api.domains.DomainUtils.createOrderMatrix(
    DomainUtils.java:80)
at de.feu.ps.refacola.api.solver.SolverData.getRelationMatrix(
    SolverData.java:239)
at de.feu.ps.refacola.api.constraints.RelationalConstraint.
    doCreateVariableConstraint(RelationalConstraint.java:322)
at de.feu.ps.refacola.api.constraints.Constraint.
    createSolverConstraints(Constraint.java:213)
at de.feu.ps.refacola.api.refactorings.AbstractRefactoring.
    createSolverConstraints(AbstractRefactoring.java:594)
at de.feu.ps.refacola.api.refactorings.AbstractRefactoring.
    runRefactoring(AbstractRefactoring.java:426)
... 3 more
```

Listing A.1: Stacktrace der ArrayIndexOutOfBoundsException

```
java.lang.ClassCastException: de.feu.ps.refacola.language.commonjava.
    kinds.impl.RefacolaStaticMemberClassImpl cannot be cast to de.feu.ps
    .refacola.language.commonjava.kinds.RefacolaTopLevelType
at de.feu.ps.refacola.language.commonjava.factbase.ImprovedASTVisitor.
    visitTypeOrEnumDeclaration(ImprovedASTVisitor.java:109)
at de.feu.ps.refacola.language.commonjava.factbase.ImprovedASTVisitor.
    preVisit(ImprovedASTVisitor.java:68)
at de.feu.ps.refacola.language.commonjava.factbase.ReferenceGenerator.
    preVisit(ReferenceGenerator.java:1)
at org.eclipse.jdt.core.dom.ASTVisitor.preVisit2(ASTVisitor.java:167)
at org.eclipse.jdt.core.dom.ASTNode.accept(ASTNode.java:2478)
at org.eclipse.jdt.core.dom.ASTNode.acceptChildren(ASTNode.java:2551)
at org.eclipse.jdt.core.dom.AnnotationTypeDeclaration.accept0(
    AnnotationTypeDeclaration.java:241)
```

A. Stacktraces der gefundenen Fehler

```
at org.eclipse.jdt.core.dom.ASTNode.accept (ASTNode.java:2480)
at org.eclipse.jdt.core.dom.ASTNode.acceptChildren (ASTNode.java:2551)
at org.eclipse.jdt.core.dom.CompilationUnit.accept0 (CompilationUnit.
  java:219)
at org.eclipse.jdt.core.dom.ASTNode.accept (ASTNode.java:2480)
at de.feu.ps.refacola.language.commonjava.factbase.CommonJavaFactbase.
  buildFacts (CommonJavaFactbase.java:259)
at de.feu.ps.refacola.language.commonjava.factbase.CommonJavaFactbase.<
  init> (CommonJavaFactbase.java:84)
at de.feu.ps.conte.launch.TestJob.run (TestJob.java:114)
at org.eclipse.core.internal.jobs.Worker.run (Worker.java:54)
```

Listing A.2: Stacktrace

```
org.eclipse.core.runtime.AssertionFailedException: assertion failed:
  Hit the root while searching for enclosing Type
at org.eclipse.core.runtime.Assert.isTrue (Assert.java:110)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.searchEnclosingType (
  ReferenceReceiverInspector.java:326)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.searchEnclosingType (
  ReferenceReceiverInspector.java:339)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.getReceiverTypeRecursion (
  ReferenceReceiverInspector.java:232)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.getReceiverTypeRecursion (
  ReferenceReceiverInspector.java:258)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.getReceiverTypeRecursion (
  ReferenceReceiverInspector.java:258)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.getReceiverType (
  ReferenceReceiverInspector.java:160)
at de.feu.ps.refacola.language.commonjava.factbase.
  ReferenceReceiverInspector.getReceiverType2 (
  ReferenceReceiverInspector.java:83)
at de.feu.ps.refacola.language.commonjava.factbase.ImprovedASTVisitor
  .isUnqualifiedFieldReference (ImprovedASTVisitor.java:387)
at de.feu.ps.refacola.language.commonjava.factbase.ReferenceGenerator
  .visit (ReferenceGenerator.java:233)
at org.eclipse.jdt.core.dom.SimpleName.accept0 (SimpleName.java:148)
at org.eclipse.jdt.core.dom.ASTNode.accept (ASTNode.java:2480)
at org.eclipse.jdt.core.dom.ASTNode.acceptChild (ASTNode.java:2528)
at org.eclipse.jdt.core.dom.SwitchCase.accept0 (SwitchCase.java:146)
at org.eclipse.jdt.core.dom.ASTNode.accept (ASTNode.java:2480)
(...)
```

A. Stacktraces der gefundenen Fehler

```
at org.eclipse.jdt.core.dom.CompilationUnit.accept0(CompilationUnit.  
    java:219)  
at org.eclipse.jdt.core.dom.ASTNode.accept(ASTNode.java:2480)  
at de.feu.ps.refacola.language.commonjava.factbase.CommonJavaFactbase  
    .buildFacts(CommonJavaFactbase.java:288)  
at de.feu.ps.refacola.language.commonjava.factbase.CommonJavaFactbase  
    .<init>(CommonJavaFactbase.java:91)  
at de.feu.ps.conte.launch.TestJob.run(TestJob.java:122)  
at org.eclipse.core.internal.jobs.Worker.run(Worker.java:54)
```

Listing A.3: Stacktrace

B. Inhalt der beigefügten CD

updatesite.zip enthält die kompilierte Version des Constraint Testers, der in Eclipse als Update Site installiert werden kann.

sourcecode.zip enthält den Quellcode des Constraint Testers sowie der notwendigen Projekte. Die Datei kann direkt in einen Eclipse Workspace importiert werden.

testprojekte.zip enthält die im Rahmen dieser Arbeit verwendeten Testprojekte, ebenfalls im Format für den Eclipse Workspace.

C. Installation des Constraint Testers

Der Constraint Testers wird wie folgt über eine sog. „Update Site“ in eine existierende Eclipse-Installation eingebunden:

1. Im Menü „Help“ die Option „Install New Software...“ auswählen
2. Auf den Knopf „Add...“ klicken
3. Auf den Knopf „Archive...“ klicken
4. Auf der CD die Datei „updatesite.zip“ auswählen
5. auf „OK“ klicken
6. In der Liste den Constraint Tester auswählen und auf „Next“ klicken
7. Nochmal auf „Next“ klicken, die Lizenzbestimmung akzeptieren und auf „Finish“ klicken
8. Die Frage, ob unsigned Software von unbekannter Herkunft installiert werden soll, mit „Ja“ beantworten
9. Eclipse neu starten

D. Benutzungsanleitung für den Constraint Tester

Nach der Installation stehen in Eclipse die drei neuen Views „Test Result View“, „Test Detail View“ und „Refactoring View“ zur Verfügung. Diese Views sollten zunächst geöffnet werden und entsprechend den eigenen Vorstellungen arrangiert werden. Eine mögliche Anordnung ist in Abbildung D.1 dargestellt. Da der Inhalt der Views von einander anhängig ist, empfiehlt es sich, sie so anzuordnen, dass alle drei Views gleichzeitig zu sehen sind.

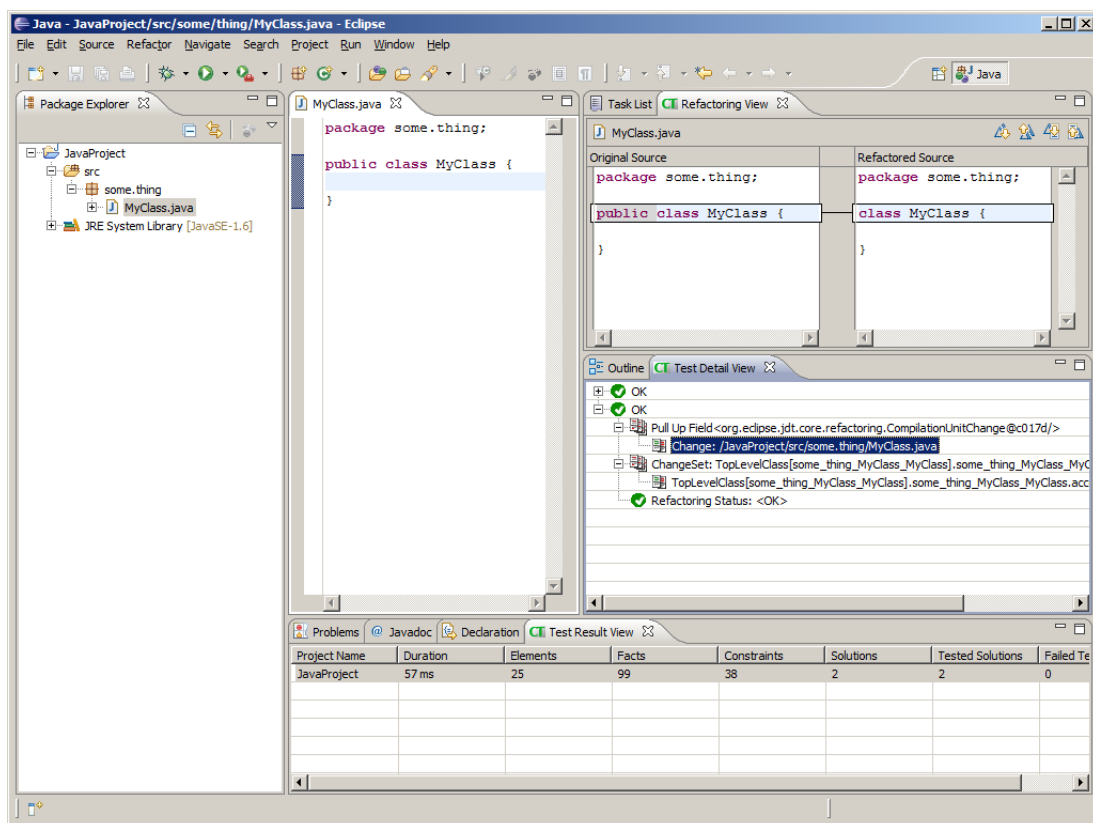


Abbildung D.1.: Anordnung der Views

Voraussetzung für die Ausführung des Constraint Testers ist, dass sich mindestens ein Java Projekt (ohne Kompilierungsfehler) im Workspace befindet. Nun wird zunächst

D. Benutzungsanleitung für den Constraint Tester

eine Launch Configuration angelegt, dies geschieht z. B. für das Menü „Run...“ und den Punkt „Run Configurations...“.

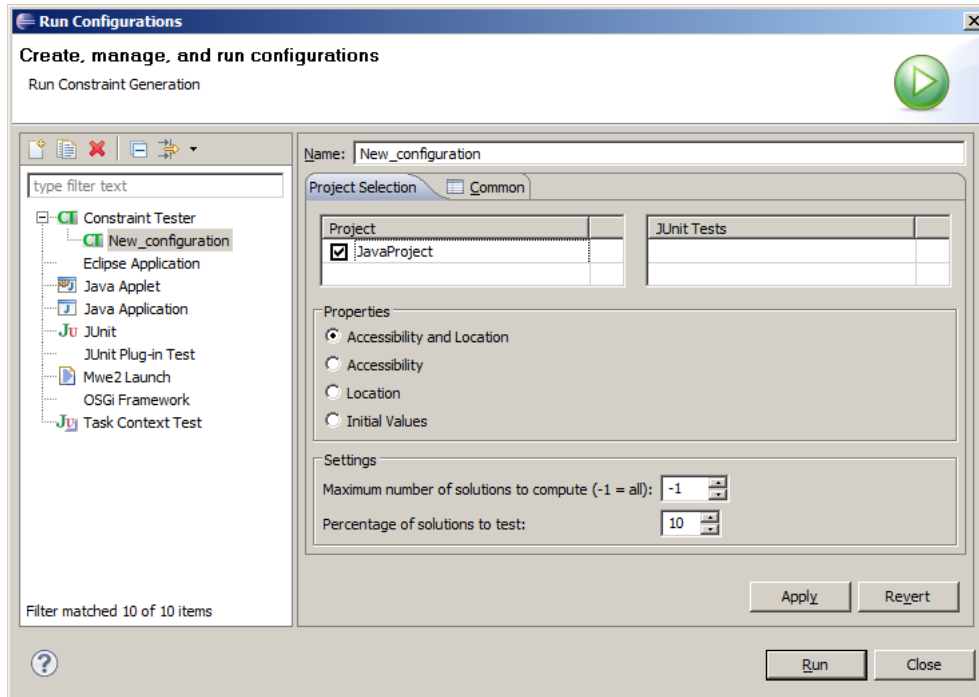


Abbildung D.2.: Konfiguration des Tests

Im nun erscheinenden Dialog kann über die Auswahl „Constraint Tester“ und den Knopf oben links eine neue Konfiguration angelegt werden. Nächster Schritt ist die Auswahl der zu testenden Projekte indem die jeweilige Checkbox selektiert wird. Auf der rechten Seite können die JUnit Tests ausgewählt werden, die nach jeder Refaktorisierung ausgeführt werden sollen. Die Liste enthält alle Launch Configurations für JUnit Tests und kann durch die Erstellen neuer Konfigurationen erweitert werden. Im unteren Teil des Dialogs können die Properties ausgewählt werden, die sich während der Refaktorisierung verändern dürfen. Außerdem kann die Höchstzahl der zu berechnenden Lösungen und der Prozentsatz der zu testenden Lösungen bestimmt werden.

Sind alle Einstellungen getroffen, werden sie über den Knopf „Apply“ gespeichert, über den Knopf „Run“ wird der Test gestartet.

Literaturverzeichnis

- [Apt03] APT, Krzysztof R.: *Principles of Constraint Programming*. Cambridge University Press, 2003
- [Bar05] BARTAK, Roman: *Constraint Propagation and Backtracking-based Search*. Charles Universität, Prag, 2005
- [BCE⁺10] BEHRENS, Heiko ; CLAY, Michael ; EFFTINGE, Sven u. a.: *Xtext User Guide*. URL http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf, 2010
- [CR09] CLAYBERG, Eric ; RUBEL, Dan: *eclipse Plug-ins*. Addison Wesley, 2009
- [ECM06] ECMA: *Standard ECMA-367 Eiffel: Analysis, Design and Programming Language*. URL <http://www.ecma-international.org/publications/standards/Ecma-367.htm>, 2006
- [GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Addison-Wesley Professional, 1994. – ISBN 9780201633610
- [HW09] HOFSTEDT, Petra ; WOLF, Armin: *Einführung in die Constraint-Programmierung*. Springer, 2009. – ISBN 3540231846
- [MS98] MARRIOTT, Kim ; STUCKEY, Peter J.: *Programming with Constraints: An Introduction*. MIT Press, 1998
- [Opd92] OPDYKE, William F.: *Refactoring Object-Oriented Frameworks*. 1992
- [Pil11] PILGRIM, Jens von: *Refacola. Konzepte, Frameworks, Umsetzung*. Clare Workshop, Hagen, 2011
- [PS94] PALSBERG, Jens ; SCHWARTZBACH, Michael I.: *Object-Oriented Type Systems*. John Wiley & Sons, 1994. – ISBN 9780471941286
- [SKP11] STEIMANN, Friedrich ; KOLLEE, Christian ; PILGRIM, Jens von: *A Refactoring Constraint Language and its Application*. 2011

- [SM03] SUN MICROSYSTEMS, Inc.: *Tuning Garbage Collection with the 5.0 Java Virtual Machine*. URL <http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html>, 2003
- [SM05] SUN MICROSYSTEMS, Inc.: *The Java Language Specification, Third Edition*. URL http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html, 2005
- [ST09] STEIMANN, Friedrich ; THIES, Andreas: *From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility*. 2009
- [Szu03] SZURSZEWski, Joe: *We Have Lift-off: The Launching Framework in Eclipse*. URL <http://www.eclipse.org/articles/Article-Launch-Framework/launch.html>, 2003
- [TKB03] TIP, Frank ; KIEZUN, Adam ; BÄUMER, Dirk: Refactoring for generalization using type constraints. In: *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM, 2003 (OOPSLA '03). – ISBN 1-58113-712-5, 13-26