

Ein Algorithmus zur systematischen Constraintzeugung
für die constraintbasierte Refaktorisierung

Masterarbeit

Eingereicht von Erland Müller

13. Mai 2011

Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen

Betreuer:
Prof. Dr. Friedrich Steimann
Dipl.-Inform. Andreas Thies

Zusammenfassung

Refaktorisierungswerkzeuge dienen dazu, Programme in bessere Ordnung zu bringen, ohne deren Bedeutung zu ändern. Nützliche Werkzeuge arbeiten korrekt und zügig. Bisher erfüllen sie oft nur eine dieser beiden Anforderungen. Diese Arbeit zeigt exemplarisch, dass constraintbasierte Refaktorisierung, die nur tatsächlich benötigte Constraints berechnet und löst, weitgehend fehlerfrei *und* schnell sein kann und dabei geringe Speicherplatzanforderungen stellt.

Werkzeuge, die auf Grundlage von Typconstraints arbeiten, sind verbreitet im Einsatz. Constraintbasierte Werkzeuge, die auch Zugreifbarkeitsregeln von Programmiersprachen berücksichtigen, können Programme zwar nahezu korrekt refaktorisieren, brauchen dafür bisher jedoch unpraktikabel viel Zeit und Speicher. Steimann und Thies [36] schlagen vor, anstatt ein vollständiges Constraintnetzwerk zu erzeugen und zu lösen, nur genau die Constraints zu berechnen, die eine Refaktorisierung korrekt spezifizieren. Diese Arbeit greift diesen Ansatz auf, arbeitet ihn an einem Beispiel aus und prüft ihn auf praktische Eignung.

Auf Grundlage der Vorarbeit [36] werden Constraintregeln für eine PULL-UP-FIELD-Refaktorisierung von Java-Programmen ausgewählt und verfeinert. Eine graphentheoretische Interpretation der Constraintmenge führt die Ermittlung genau der nötigen Constraints auf Tiefendurchlauf des Hypergraphen zurück. Ein universeller Constraintlöser berechnet Lösungen der gewonnenen Constraintmenge. Der Algorithmus zur selektiven Constraintterzeugung wird als Eclipse-Plugin auf Basis der *Java-Development-Tools* implementiert. Seine Richtigkeit kann nur empirisch untermauert werden, da der Algorithmus eine komplexe Programm-analyse umfasst. Zu diesem Zweck wird eine Eclipse-Refaktorisierung implementiert, die eine Lösung der Constraintmenge in Programmcode zurückschreibt. Das prototypische Werkzeug wird an konstruierten Testfällen und an quelloffenen Projekten, die über eine gute Testabdeckung verfügen, getestet. Entdeckte Fehler sind auf Grenzen der Constraintregeln und Mängel der provisorischen Implementierung der Code-Transformation zurückführbar. Laufzeitmessungen und Abschätzungen des Speicherplatzbedarfs ergeben, dass das Verfahren für den praktischen Einsatz geeignet ist. Dieses Ergebnis unterstützt die Vermutung, selektive Constraintterzeugung sei auch für schwierigere Refaktorisierungen praktikabel.

Erklärung

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

Hamburg, 13. Mai 2011

Erland Müller

Inhaltsverzeichnis

Zusammenfassung	2
1 Einleitung	5
1.1 Refaktorisierung als Einschränkungsproblem	5
1.2 Untersuchungsfragen und Aufgabenstellung	6
1.3 Beitrag der Arbeit	7
1.4 Aufbau der Arbeit	8
2 Vorarbeiten	8
2.1 Refaktorisierung im Kontext	9
2.2 Refaktorisierungswerkzeuge	9
2.3 Constraintbasierte Refaktorisierung	10
3 Grundlagen	13
3.1 Constraintgraph	13
3.2 Komponenten	15
3.3 Algorithmus und Datenstrukturen	18
3.4 Komplexität	19
3.5 Redundante Constraints	20
3.6 Lösen der Constraintmenge	20
3.7 Testen des Algorithmus	22
4 Implementierung	24
4.1 Anforderungen	24
4.2 Lösungsansatz	25
4.3 Erweiterung des Eclipse-Java-Modells	27
4.4 Berechnen der Constraintmenge	29
4.5 Adapter zur Constraintlösung	31
4.6 Ändern des Quellcodes	33
4.7 Testen der Refaktorisierung	35
5 Ergebnisse	37
5.1 Übersicht	37
5.2 Constraint- und Lösungsmengen	38
5.3 Testergebnisse	39
5.4 Rechen- und Implementierungsaufwand	41
6 Diskussion	43
6.1 Constraint- und Lösungsmengen	43
6.2 Testergebnisse	44
6.3 Eignung für die Praxis	46
6.4 Probleme der Implementierung	48
6.5 Constraintlösung parallel zur Constraint erzeugung	50
6.6 Offene Fragen	52
6.7 Schlussfolgerungen	54

Anhang

A Constraintregeln	56
A.1 Notation	56
A.2 Zugreifen	59
A.3 Erben	60
A.4 Verbergen	62
A.5 Abschatten	63
A.6 Verschiedenes	65
A.7 Ergänztes	66
A.8 Ausgelassenes	68
A.9 Grenzen	68
B Tabellen	71
B.1 Verwendete Formelzeichen	71
B.2 Abkürzungen	72
B.3 Bezeichnung der Constraintregeln	72
B.4 Gliederung der Implementierung	72
B.5 Anzahlen von Lösungen der Constraintmengen	73
C Bedienung und Installation	73
C.1 Demo-Refaktorisierung	73
C.2 Quellcode der Implementierung	74
Literatur	75

1 Einleitung

Die Struktur einer Software, wie die anderer Gebrauchstexte auch, verkommt durch häufige Korrekturen und Ergänzungen. Sie muss regelmäßig in Ordnung gebracht werden. Weitere Änderungen sind leichter und verursachen weniger neue Fehler, wenn im Text alles an seinem Platz ist. *In Ordnung bringen* heißt bei Programmtexten *refaktorisieren*, wenn die Bedeutung des Programms dabei erhalten bleibt. Einschränkungen, auch *Constraints* genannt, schützen beim Refaktorisieren davor, die Bedeutung versehentlich zu ändern. Beim Aufräumen helfen Refaktorisierungswerkzeuge, die alle nötigen Constraints kennen und beachten. Werkzeuge sind aber nur eine Hilfe, wenn sie keine Fehler verursachen und schnell genug arbeiten. In dieser Arbeit geht es darum, ein Verfahren zur Berechnung der nötigen Constraints zu realisieren und zu prüfen, ob es diese Anforderungen erfüllt.

1.1 Refaktorisierung als Einschränkungsproblem

Refaktorisieren ist gängige Praxis der Programmierung, beim Entwickeln neuer Software ebenso wie zur Vorbereitung von Reparaturen oder Erweiterungen. Dabei kommt es darauf an, das beobachtbare Verhalten des Systems nicht zu verändern. Das ist eine schwierige und oft langwierige Aufgabe, weil dabei komplexe Bedingungen und Wechselwirkungen zu beachten sind. Im Grunde ist die Tätigkeit gut automatisierbar. Entwicklungsumgebungen sind entsprechend mit Refaktorisierungswerkzeugen ausgestattet. Allerdings sind diese Werkzeuge bisher weit weniger verlässlich als Compiler.

Das Ändern der Zugriffsmodifikatoren von Deklarationen scheint auf den ersten Blick eine einfache Refaktorisierung zu sein. Bouillon et al. [4] zeigen jedoch, dass Auswirkungen dieser Refaktorisierung sich im Programm weit ausbreiten und unbemerkt eine Änderung der Bedeutung des Programms verursachen können. Das Problem lässt sich mithilfe von Constraints lösen.

Constraints sind mathematisch ausgedrückte Einschränkungen. Sie beschreiben die Beziehungen der Elemente eines Programms, die den Regeln der Programmiersprache gehorchen müssen. Ein Beispiel: In Java erbt eine Subklasse ein Feld, das eine Superklasse in einem anderen Package deklariert, nur dann, wenn das Feld wenigstens mit dem Zugriffsmodifikator *protected* deklariert ist. Angenommen, es gibt in der Subklasse eine Referenz auf das Feld, dann verlangt ein Constraint, dass der Zugriffsmodifikator der Felddeklaration größer oder gleich *protected* sein und bleiben muss, solange Sub- und Superklasse verschiedenen Packages angehören.¹

Constraints sind an sich modular; jedes Constraint beschreibt unabhängig von anderen eine Bedingung des Programms, die mit anderen additiv zusammenwirkt. Sie formulieren zusammen genommen eine Refaktorisierung als Einschränkungsproblem. Damit sind die deklarativen Verfahren der Constraintprogrammierung zur Lösung von Refaktorisierungsaufgaben verwendbar.

Wie gewinnt man die Constraints? Die Regeln der Programmiersprache lassen sich in Form allgemeingültiger Constraintregeln ausdrücken. Constraintregeln geben an, welches Constraint für welche Konstellation in einem Programm erfüllt sein muss. Diese Constraintregeln, ergänzt um das Ziel einer konkreten Refaktorisierung, werden auf das zu refaktorisierende Programm angewendet. Das heißt, das Programm wird analysiert und beim Antreffen einer Situation, die eine Constraintregel beschreibt, wird das entsprechende Constraint erzeugt. Die erhaltene Menge von Constraints beschreibt die Vor- und Nachbedingungen der Refaktorisierungsaufgabe. Findet man eine Lösung, die die Constraintmenge erfüllt, kann die Lösung in eine Änderung des Programms übersetzt werden. Diese Programmänderung erhält die Bedeutung des refaktorierten Programms, vorausgesetzt die Constraintregeln sind vollständig und die Programmanalyse ist korrekt.

¹Wie Abschnitt A.3 zeigen wird, ist das Constraint tatsächlich etwas verwickelter.

Typbezogene Refaktorisierungen funktionieren auf der Basis von Constraints sehr verlässlich. Sie werden beispielsweise in der Entwicklungsumgebung Eclipse praktisch eingesetzt. Steimann und Thies [36] ist es gelungen, constraintbasierte Verfahren auf Refaktorisierungen zu übertragen, die auch Zugriffsmodifikatoren berücksichtigen. Eine neuere Arbeit [35] kombiniert Typ- und Zugreifbarkeitsconstraints.

1.2 Untersuchungsfragen und Aufgabenstellung

Constraintbasierte Refaktorisierung von Zugriffsmodifikatoren ist bisher sehr aufwendig, weil das gesamte Programm analysiert wird und sehr viele Constraints erzeugt werden. Das ist jedoch gar nicht nötig, denn eine geplante Refaktorisierung ist nur von vergleichsweise wenigen Constraints beschränkt. In dieser Arbeit geht es darum, gerade die wirksamen Constraints zu finden und damit die Refaktorisierung deutlich zu beschleunigen. Dieser Abschnitt beschreibt, wie man dahin kommt.

Das Problem

Constraintbasierte Refaktorisierung ist geeignet, korrekte Refaktorisierungsoperationen zu berechnen, die Programmteile verlagern oder Typbeziehungen berühren und dabei Zugriffsmodifikatoren ändern [36]. Dazu werden vorläufig *alle* Constraintregeln auf *alle* möglichen Stellen des zu refaktorisierenden Programms angewendet. Das erzeugt eine sehr große Anzahl von Constraints. Der Aufwand für die Programmanalyse ist deshalb sehr hoch, und das resultierende Constraintproblem ist ebenfalls nur mit hohem Rechenaufwand zu lösen.

Für interaktive Refaktorisierung wäre ein Werkzeug, das die Constraints auf diese Weise erschöpfend erzeugt und löst, kaum brauchbar, weil eine Programmiererin nicht so lange auf ein Ergebnis warten wollte, sondern das Programm stattdessen eher manuell refaktorisierete [25].

Fragen und Ziele

Steimann und Thies [36] stellen fest, vor einer Refaktorisierung seien alle auf einem Programm erzeugbaren Constraints erfüllt. Eine geplante Refaktorisierung kann nur wenige dieser Constraints verletzen und ist nur durch wenige beschränkt. Die Autoren schlagen deshalb vor, nur die nötigen Programmteile zu analysieren und nur die Constraints zu erzeugen, die eine geplante Refaktorisierung überhaupt verletzen kann.

Die zu klärende Frage ist, welche Constraints das sind und wie gerade diese Constraints zu finden sind. Die Untersuchungsfrage lautet mit anderen Worten: Welche Constraintregeln müssen auf welche Programmstellen angewendet werden, um selektiv die Constraints zu erzeugen, die eine geplante Refaktorisierungsoperation auf einem gegebenen Programm beschränken? Die Frage muss in einer programmatisch ausführbaren Form beantwortet werden, um die Antwort in einem Refaktorisierungswerkzeug zu nutzen. Das heißt, es ist eine Rechenvorschrift gesucht, die Constraints selektiv erzeugt. Diesen Algorithmus anzugeben und zu testen, ist das primäre Ziel dieser Arbeit.

Ein Verfahren, das nur die notwendigen Constraints erzeugt, ist der erschöpfenden Constraintinterzeugung nur dann überlegen, wenn es die Berechnung der Refaktorisierungsoperation deutlich beschleunigt und geringere Speicherplatzanforderungen stellt. Ist die selektive Constraintinterzeugung selbst so aufwendig, dass der Vorteil, kleinere Constraintmengen lösen zu können, wieder zunichte gemacht wird, ist nichts gewonnen. Damit ist die zweite Frage aufgeworfen: Ist der Rechenaufwand des Verfahrens klein genug, um es praktisch einzusetzen? Eine Antwort auf die Frage nach praktischer Eignung beantwortet gleichzeitig die Frage nach der Relevanz eines Verfahrens zur selektiven Constraintinterzeugung. Den Algorithmus auf Eignung für die Praxis zu prüfen, ist ein sekundäres Ziel der Arbeit.

Aufgabe

Die Untersuchung wird am Beispiel einer PULL-UP-FIELD-Refaktorisierung für die Programmiersprache Java in der Version 1.5 durchgeführt. Diese Refaktorisierung zieht ein statisches oder nicht-statisches Feld von einem Sub- in einen Supertypen hoch. Bei Bedarf werden weitere Felder hochgezogen und Zugriffsmodifikatoren angepasst. Die Refaktorisierung ist nicht so trivial, dass von der Untersuchung keine Aussagekraft für andere Refaktorisierungen zu erwarten ist. PULL-UP-FIELD ist aber auch nicht so komplex, dass die Aufgabe im Rahmen einer Abschlussarbeit nicht zu lösen wäre.

Zusammen mit den Zielen ist damit die Aufgabenstellung der Arbeit bestimmt: Es ist ein Algorithmus zur selektiven Constraintterzeugung abzuleiten und für die PULL-UP-FIELD-Refaktorisierung zu implementieren. Die Implementierung soll glaubhaft machen, dass der Algorithmus richtige Ergebnisse liefert. Sie soll außerdem zeigen, welchen Rechenaufwand der Algorithmus erfordert. Um die Aufgabe zu lösen, müssen zunächst die Constraintregeln ausgewählt und verfeinert werden. Eine präzise Formulierung der Grenzen des Verfahrens soll es erleichtern, die Ergebnisse der Arbeit einzuordnen.

Abgrenzung

Es geht in dieser Arbeit jedoch nicht darum, ein fertiges Refaktorisierungswerkzeug kommerzieller Qualität zu erstellen. So dient die Implementierung der Änderung des Quellcodes des zu refaktorisierenden Programms lediglich der Prüfung des Algorithmus zur Constraintterzeugung. Auch können folgende Fragen im Rahmen der Arbeit *nicht* gelöst werden: (1) Können Lösungssuche oder Einschränkungsförderung im Zuge der Ermittlung der Constraints die Lösung des Constraintproblems bzw. das Feststellen der Nicht-Erfüllbarkeit zusätzlich beschleunigen? (2) Wie werden statische Feldreferenzen bei der Refaktorisierung automatisch kanonisch qualifiziert?

Steimanns Reformulierung vorausschauender Constraintregeln [34] und die Constraintsprache *Refacola* [35] überholen die frühere Arbeit [36] und damit auch Teile dieser Abschlussarbeit, die auf der früheren Arbeit basiert. Steimann et al. [35] stellen auch bereits einen ähnlichen Algorithmus zur selektiven Constraintterzeugung auf Grundlage der neueren Formulierung von Constraintregeln vor.

1.3 Beitrag der Arbeit

Die vorliegende Arbeit setzt Steimanns und Thies' [36] Vorschlag erfolgreich um, selektiv gerade die wenigen Constraints zu erzeugen und zu lösen, die eine geplante Refaktorisierung beschränken. Ein entsprechender Algorithmus wird abgeleitet, exemplarisch implementiert und getestet. Folgende Absätze erläutern den Beitrag der Arbeit.

Die Arbeit verfeinert und ergänzt die in der Vorarbeit [36] entwickelten Constraintregeln für eine PULL-UP-FIELD-Refaktorisierung von Java-Programmen. Die Constraintregeln behandeln nun auch Mehrfachvererbung, geschachtelte Classifier und das mit Schachtelung verbundene Problem des Abschattens. Der Anhang begründet die Regeln und ihre Grenzen mit Bezug auf die Java-Sprachspezifikation.

Die Ableitung des Algorithmus geht von folgender Beobachtung aus: Das Constraintnetzwerk repräsentiert einen Graphen, der in Komponenten zerfällt. Änderungen an Programm-elementen wirken nur über Constraints, die mit diesen Programmelementen eine Komponente bilden. Die benötigten Constraints sind daher gerade alle Constraints der Komponente, die das von der Refaktorisierung zu ändernde Programmelement enthält. Der implementierte Algorithmus ermittelt die benötigte Komponente des Constraintgraphen per Tiefendurchlauf. Der Graph entsteht erst beim Durchlauf durch die Analyse des zu refaktorisierenden Programms.

Die Implementierung des Algorithmus wird automatisch auf reale Programme angewendet und das Ergebnis einer großen Anzahl von Refaktorisierungsoperationen geprüft. Die refaktorierten Programme bleiben fehlerfrei kompilierbar und sie bestehen ihre Testfälle – soweit die Constraintregeln eine fehlerfreie Refaktorisierung garantieren. Das Ergebnis untermauert die Ausgangshypothese, eine Lösung für den kleinen, benötigten Teil der Constraints spezifiziere korrekte Refaktorisierungen. Weiterhin unterstützt das Ergebnis die Arbeitshypothese, der Algorithmus und seine Implementierung seien richtig.

Die Ergebnisse der Testläufe zeigen, dass die Anzahl benötigter Constraints meist vergleichsweise klein ist. Die Rechenzeit für das Erzeugen und Lösen des Constraintnetzwerks ist bis auf Ausnahmen deutlich geringer als die Rechenzeit für die Quellcode-Änderung und das inkrementelle Kompilieren der refaktorierten Programme. Der Algorithmus ist für einen Einsatz in der Praxis schnell genug und sein Speicherplatzbedarf ist gering. Ein Eclipse-Plugin, das die implementierte Refaktorisierung interaktiv ausführt, demonstriert ihre praktische Anwendbarkeit.

Das Ergebnis der Arbeit eröffnet den Weg, gleich beim Durchlauf zur Erzeugung der Constraints eine Lösung des Constraintnetzwerks zu suchen und damit die Anzahl benötigter Constraints weiter zu reduzieren.

1.4 Aufbau der Arbeit

Der Hauptteil der Arbeit ist folgendermaßen organisiert: Er beginnt mit einer Übersicht des Forschungsstands zur werkzeuggestützten Refaktorisierung (Kapitel 2). Die Darstellung anderer Arbeiten legt den Schwerpunkt auf die Ergebnisse der Vorarbeiten über constraintbasierte Refaktorisierung.

Das Kapitel 3 (Grundlagen) führt den Algorithmus zur Constraintgenerierung auf einen Tiefendurchlauf des Constraintgraphen zurück, schätzt seine Komplexität ab und zeigt, wie eine Lösung des erhaltenen Constraintnetzwerks ermittelt wird. Möglichkeiten, die Korrektheit des Algorithmus zu prüfen, werden hier eruiert.

Die Ausführung des praktischen Teils der Arbeit behandelt Kapitel 4. Es beschreibt die Implementierung des Algorithmus, die Ermittlung von Lösungen der Constraintmenge und die Änderung des Quellcodes. Zur Implementierung gehören auch die Einbindung des Algorithmus in das Rahmenwerk der Entwicklungsumgebung, ein Demowerkzeug und der Adapter zum Betrieb der Refaktorisierung mit einem *Refactoring-Tool-Tester*.

Kapitel 5 präsentiert Ergebnisse. Das sind zuerst die Anzahlen von Constraints je Refaktorisierungsoperation, die Anzahlen möglicher Lösungen der ermittelten Constraintmengen und die Ergebnisse der Laufzeitmessungen. Das Kapitel beschreibt außerdem die Testergebnisse und erklärt Tests, die fehlschlagen.

Abschließend diskutiert und bewertet das Kapitel 6 die Ergebnisse, benennt Probleme und offene Fragen, schlägt Verbesserungen des Algorithmus vor und zieht Schlussfolgerungen.

Constraintbasiertes Refaktorisieren setzt korrekte Constraintregeln voraus. Ihre Auswahl und Ergänzung gehören jedoch nicht zur Aufgabe, einen Algorithmus abzuleiten, zu implementieren und zu testen. Die relevanten Constraintregeln, ihre Notation und Grenzen sind deshalb im Anhang A zusammengestellt und begründet.

2 Vorarbeiten

Werkzeuggestützte Refaktorisierung ist seit gut 20 Jahren Gegenstand wissenschaftlicher Forschung. Dieser Abschnitt ordnet die vorliegende Arbeit in den Zusammenhang ein und referiert vor allem die Vorarbeiten über constraintbasierte Refaktorisierung unter Berücksichtigung der Zugreifbarkeit von Deklarationen.

2.1 Refaktorisierung im Kontext

Mens und Tourwé [22] präsentieren eine Übersicht der Forschung zur Refaktorisierung von Software. Sie verstehen unter *Refaktorisierung* das Restrukturieren *objektorientierter* Systeme mit dem Ziel, ihre Qualität zu verbessern, ohne dabei ihr äußeres Verhalten zu ändern. Fowler betont die Qualitätsmerkmale *Verständlichkeit* und *Änderbarkeit* mit der Anmerkung, alle könnten Quellcode schreiben, den Computer verstehen; Code guter Programmiererinnen sei hingegen für Menschen verständlich [9, S. 15].

Der mehrdeutige Begriff *Refaktorisierung* bezeichnet den Vorgang und sein Resultat, aber auch das Muster des Vorgangs sowie das Werkzeug, das den Vorgang automatisiert. Die eigentliche Tätigkeit kann im Deutschen als *refaktorisieren* abgegrenzt werden [31, S. 153 f.].

Die Tätigkeit lässt sich in Teilaufgaben gliedern: (1) Entscheiden, welche Refaktorisierung an welcher Stelle eines Programms angewendet werden soll, (2) den Erhalt des Verhaltens sicherstellen, (3) die Refaktorisierung ausführen, (4) die Auswirkung auf die Qualität beurteilen und (5) die Konsistenz von Programm und seiner Dokumentation erhalten [22]. Diese Arbeit beschäftigt sich mit Aspekten der Teilaufgaben (2) und (3).

Bei den Begriffsdefinitionen bleibt vage, was unter „beobachtbarem Verhalten² der Software“ [9, S. 53] zu verstehen sei. Das reflektiert die von Dijkstra [7, S. 24] beklagte grundsätzliche Schwierigkeit, Programme zu vergleichen. Die Anforderung, eine gegebene Eingabe produziere unverändert die gleiche Ausgabe, ist in manchen Kontexten zu kurz gegriffen. Eingebettete, zeit- oder sicherheitskritische Software hat spezifischere Anforderungen zu erfüllen [22].

Fowlers Standardwerk [9] katalogisiert systematisch kleinere Refaktorisierungen. Es versteht jede Refaktorisierung mit einem eindeutigen Namen, gibt (implizit) ihre Vorbedingungen an und beschreibt die Schritte ihrer Ausführung. Die Vorbedingungen bleiben jedoch informell und unvollständig. Sie können die Nachbedingung unveränderter Bedeutung nicht garantieren [31, S. 154 f., S. 158 ff.]. Fowler setzt deshalb vollständige, automatisch ausführbare Tests des refaktorierten Programms voraus [9, S. 7 f.] – allerdings entwerfen manche Refaktorisierungen diese Tests [22].

Refaktorisieren hat sich in der Programmierpraxis etabliert und ist fester Bestandteil verbreiteter agiler Vorgehensweisen [22]. Dennoch wird oft von Hand refaktoriert, obwohl es eine langwierige und fehleranfällige Tätigkeit ist [23]. Ein Grund dafür mag die mangelnde Qualität gängiger Refaktorisierungswerkzeuge sein [33].

2.2 Refaktorisierungswerkzeuge

Populäre Entwicklungsumgebungen wie Eclipse, *Visual Studio* oder *Squeak* implementieren semi-automatische Refaktorisierungen [23] – sie heißen *semi-automatisch*, weil die Programmiererin vorgibt, welche Refaktorisierung wo angewendet wird. Entscheidend ist, dass die Refaktorisierung die Korrektheit und die Bedeutung des Programms erhält [22]. Eine Reihe von Arbeiten belegt jedoch, dass die Werkzeuge diesem Anspruch nicht gerecht werden [5, 15, 18, 26, 36].

Mens und Tourwé [22] kategorisieren verschiedene Ansätze, die zu bedeutungserhaltenden Refaktorisierungen führen, und resümieren die Ergebnisse. Folgende dieser Kategorien sind im Zusammenhang der vorliegenden Arbeit wesentlich:

1. Eine Refaktorisierung kann formal mit Invarianten, Vor- und Nachbedingungen spezifiziert werden. Ein Werkzeug prüft typischerweise die Vorbedingungen und führt eine Refaktorisierung nur aus, wenn sie erfüllt sind. Die Ausführung berücksichtigt die Invarianten und erfüllt im Ergebnis die Nachbedingungen, die das Refaktorisierungsziel

²Programme verhalten sich nicht. Der Begriff hat sich in der objektorientierten Programmierung jedoch eingebürgert [31, S. 41]. Es ist üblich, vom *Verhalten* (*behaviour*) eines Systems zu sprechen. Auf Programme bezogen sind die Begriffe *Bedeutung* oder *Semantik* angemessener. Selbstverständlich ist *Semantik* nicht synonym zu *Verhalten*.

und den Erhalt der Bedeutung des Programms zusichern. Mens und Tourwé zählen constraintbasierte Refaktorisierung, die der folgende Abschnitt behandelt, zu dieser Kategorie.

2. Programme lassen sich als Graphen auffassen. Daher liegt es nahe, die Theorie der Graphen-Transformation für Refaktorisierungen zu nutzen. Verschiedene Arbeiten zeigen, dass sich auf diese Weise Nachbedingungen formal in Vorbedingungen überführen lassen, die den Erhalt der Bedeutung zusichern können. Refaktorisierungen lassen sich durch eine Verkettung einfacher Graphen-Transformationen realisieren, deren Korrektheit jeweils für sich bewiesen werden kann.
3. *Programm-Verfeinerung (refinement)* nennen die Autoren Ansätze, die ein zu refaktorisierendes Programm schrittweise in eine formal handhabbare Sprache übersetzen, dort die Restrukturierung beweisbar korrekt durchführen und das Ergebnis anschließend in die Ausgangssprache zurück transformieren.

Der dritten Kategorie lassen sich auch die aktuellen Arbeiten von Schäfer et al. [26, 27] zuordnen. Sie spezifizieren und implementieren verschiedene Refaktorisierungen von Java-Programmen auf Basis des *JastAddJ*-Compilers. Die Refaktorisierungen werden aus sogenannten Mikro-Refaktorisierungen zusammengesetzt, die auf einer Erweiterung von Java definiert sind. In der erweiterten Sprache sind Deklarationen fest an ihre Referenzen gebunden (*locked*) und eine Restrukturierung erhält verifizierbare Bindungen, Daten- und Kontrollfluss. Scheitert eine Rücktransformation in gewöhnliches Java (*unlocking*), die bei Bedarf Qualifizierungen einfügt, erweist sich die Refaktorisierung im Nachhinein als undurchführbar und wird nicht ausgeführt.

Anders als alle oben genannten Arbeiten verfolgen Soares et al. [28] einen radikal pragmatischen Ansatz. Sie führen eine Refaktorisierung zunächst aus, so unvollkommen, wie sie eine Entwicklungsumgebung implementiert. Erst nachträglich prüfen sie, ob das Programm korrekt und seine Bedeutung erhalten blieb. Ist das nicht der Fall, wird die Refaktorisierung wieder rückgängig gemacht. Den Erhalt der Bedeutung des refaktorierten Programms prüfen die Autoren jeweils mithilfe einer automatisch generierten Test-Suite, die die Bedeutung des Programms vor der Refaktorisierung abbildet.

2.3 Constraintbasierte Refaktorisierung

Freuder erklärt, Constraintprogrammierung gehöre zu den Ansätzen der Informatik, die dem Ideal der Programmierung bisher am ehesten entsprechen: „Die Anwenderin stellt die Aufgabe, der Computer löst sie.“ [10] Dieser Abschnitt beginnt mit den grundlegenden Arbeiten über die Lösung von Refaktorisierungsproblemen mithilfe von Constraints und behandelt anschließend die unmittelbaren Vorarbeiten, die den Ansatz auf Zugreifbarkeitsconstraints erweitern.

2.3.1 Typconstraints

Tip [39] fasst die Arbeiten über constraintbasierte Refaktorisierung von Java-Programmen zusammen, die Änderungen an der Klassen- oder Typhierarchie vornehmen. Dazu gehören die EXTRACT-INTERFACE-, GENERALIZE-DECLARED-TYPE- und PULL-UP-MEMBER-Refaktorisierung sowie Refaktorisierungen, die parametrischen Polymorphismus (generische Typen) einführen oder die Migration von Klassen unterstützen. Die Implementierung der entsprechenden Werkzeuge in der Entwicklungsumgebung Eclipse beruht auf diesen Arbeiten.

Die Theorie der Typconstraints geht nach Tip [39] auf Arbeiten von Palsberg und Schwartzbach [24] über Typkorrektheit und Typinferenz objektorientierter Programme zurück. Typconstraints drücken Subtypbeziehungen der Typen von Ausdrücken, von Deklarationen und ihren Orten sowie von Typvariablen aus. Constraintregeln definieren, welches

Programmkonstrukt welches Constraint erfordert. Die Constraints werden durch Anwendung der Regeln auf den abstrakten Syntaxbaum des zu refaktorisierenden Programms gewonnen. Lösungen der erhaltenen Constraintmenge entsprechen Refaktorisierungen, die die Vorbedingungen erfüllen und zu typkorrekten Programmen führen.

Tip et al. [40] präsentieren einen Algorithmus zur Lösung der Constraintmenge durch Variablen-Eliminierung für die EXTRACT-INTERFACE-Refaktorisierung. Sie erbringen den Beweis, dass das Verfahren zu einem typkorrekten Programm führt, und formulieren die begründete Vermutung, seine Bedeutung bleibe unverändert. Sie zeigen weiter, wie die Constraints zur Prüfung der Vorbedingungen der PULL-UP-MEMBER-Refaktorisierung verwendet werden.

Steimann [29] und darauf aufbauend Kegel [19] verallgemeinern und automatisieren die EXTRACT-INTERFACE-Refaktorisierung zu INFER-TYPE. Diese Refaktorisierung berechnet den allgemeinsten möglichen Typen einer Variablendeklaration in Java-Programmen unter Berücksichtigung generischer Typen. Die Menge binärer (2-stelliger) Constraints interpretiert Kegel als gerichteten Graphen. Eine Lösung des Constraintproblems ermittelt sein Algorithmus in bis zu zwei Tiefendurchläufen.

Kegel und Steimann [20] nutzen den für die INFER-TYPE-Refaktorisierung entwickelten Algorithmus, um eine komplexe Refaktorisierung zu implementieren: Die REPLACE-INHERITANCE-WITH-DELEGATION-Refaktorisierung bereinigt die Schnittstelle eines Typen um nicht benötigte, geerbte Methoden, indem sie Vererbung durch *forwarding* ersetzt [31, S. 158 ff.].

2.3.2 Zugreifbarkeitsconstraints

Bouillon et al. [4] beschreiben die Relevanz und die Schwierigkeiten einer Änderung von Zugriffsmodifikatoren in Java-Programmen. Sie kann nicht nur Zugriffsfehler verursachen, sondern auch zu schwer absehbaren Änderungen von Bindungen führen und so unbemerkt die Bedeutung eines Programms ändern. Die Arbeit präsentiert ein noch imperativ implementiertes Werkzeug, das automatisch die geringst mögliche Zugreifbarkeit von Deklarationen einstellt, ohne dabei Bindungen zu ändern.

Steimann und Thies [36] erweitern die Anwendung von Constraints auf die Probleme der Zugreifbarkeit. Die Autoren leiten Constraintregeln für die Programmiersprachen Java, C# und Eiffel ab, die die Zugreifbarkeitsregeln der Sprachen und ihre komplexen Bindungsregeln angesichts von Überschreiben, Überladen und Verbergen erfassen (vgl. Anhang A, S. 59 ff.). Sie ergänzen damit bis dahin übersehene Vorbedingungen von Refaktorisierungen und erweitern gleichzeitig deren Anwendbarkeit.

Die Regeln verwenden intern einen Zugriffsmodifikator *absent*, der nicht referenzierte Deklarationen markiert und das Verbergen unterbindet. Die Arbeit führt das Konzept der vorausschauenden Anwendung von Constraintregeln ein. Danach werden zusätzliche Constraints für Konstellationen erzeugt, die im Programm vor der Refaktorisierung noch nicht existieren, aber durch die Refaktorisierung entstehen können. Die erschöpfende Anwendung der Constraintregeln erzeugt ein Constraintproblem, das aus sehr vielen Constraints besteht. Seine Lösungen werden mit einem Standard-Constraintlöser berechnet.

Am Beispiel der implementierten MOVE-CLASS- und der PULL-UP-METHOD-Refaktorisierung demonstrieren die Autoren die Eignung von Zugreifbarkeitsconstraints. Sie erzeugen Constraintmengen für Programme, die über eine gute Testabdeckung verfügen, und zeigen anhand ihrer Testfälle, dass alle Lösungen, von einzelnen Ausnahmen abgesehen, zu fehlerfrei kompilierenden Programmen mit unveränderter Bedeutung führen. Dabei haben die Ausnahmen keinen Bezug zu Problemen der Zugreifbarkeit.

Ein Vergleich mit den Werkzeugen von Eclipse ergibt, dass constraintbasierte Refaktorisierung im Gegensatz zur Standardimplementierung korrekte Programme mit erhaltener Bedeutung liefern kann. Aufgrund der automatischen Anpassung von Zugriffsmodifikatoren werden mehr Refaktorisierungen ausführbar.

2.3.3 Vorausschau

Steimann [34] stellt das Problem der vorausschauenden Anwendung von Constraintregeln auf eine neue Grundlage. Bisher enthalten Constraintregeln in ihren Prämissen Eigenschaften, die im Rahmen einer Refaktorisierung veränderlich sind. Das erfordert vorausschauende Anwendung der Regeln in Abhängigkeit von der Art der Refaktorisierung. Die neue Lösung vermeidet diese Abhängigkeit und damit ausufernde Fallunterscheidungen.

Die neu formulierten Constraintregeln sind für eine Programmiersprache allgemeingültig. Ein Filter spezifiziert, welche Programmeigenschaften eine konkrete Refaktorisierung ändern kann. Die Regeln werden unter Anwendung des Filters programmatisch so transformiert, dass ihre Prämissen nur unveränderliche Elemente enthalten, während alle veränderlichen Elemente in der Konklusion der Regeln bleiben. Auf diese Weise generiert die Anwendung der transformierten Regeln auch und genau die vorausschauend erforderlichen Constraints.

Die Formulierung der allgemeingültigen Regeln benutzt zum einen bedingte Constraints. Dazu werden hier sowohl gewöhnliche Constraints gezählt, die Disjunktionen bzw. Implikationen enthalten, als auch Aktivitätsconstraints, die in Abhängigkeit von der Wertebelegung einer Constraintvariablen über die Aufnahme anderer Constraints und deren Variablen in das Constraintproblem entscheiden [41]. Zum anderen führt die Formulierung zu quantifizierten Constraints, die hier über die Menge der Constraintvariablen selbst existenz- oder allquantifiziert sind und nicht nur über die Wertebereiche der Constraintvariablen.

Steimann [34] präsentiert einen Algorithmus, der die Constraints sparsam generiert. In der ersten Phase transformiert der Algorithmus die Regeln unter Anwendung eines Filters über eine Zerlegung in disjunkte Normalform, wie oben erwähnt. Die zweite Phase generiert für jede transformierte Regel die Constraints durch Abfragen auf dem zu refaktorisierenden Programm, eliminiert gelöste Constraints und prüft verbleibende auf Erfüllbarkeit. Die dritte Phase stellt die Ausgangsbelegung der Constraintvariablen her, bestimmt ihre Bereiche und ermittelt Lösungen der Constraintmenge mithilfe eines Standard-Constraintlösers.

Die Anwendung einer Implementierung des Algorithmus auf reale Java-Programme demonstriert am Beispiel der PULL-UP-FIELD-Refaktorisierung die Eignung des Ansatzes.

2.3.4 Integration zur Constraintsprache

Steimann et al. [35] integrieren in einer Arbeit, die parallel zur vorliegenden entstanden ist, oben genannte Ergebnisse zu einem Rahmenwerk für die Spezifikation und Implementierung allgemeiner, constraintbasierter Refaktorisierungswerkzeuge. Kern des Rahmenwerks ist eine deklarative domänenspezifische Sprache (DSL) namens *Refacola*, die Werkzeuge für gängige Entwicklungsumgebungen generiert. Folgende Absätze resümieren wesentliche Resultate.

Bisher waren constraintbasierte Refaktorisierungen entweder auf Typ- oder auf Zugriffsbarkeitsconstraints spezialisiert. Diese beschränkende Trennung hebt das neue Rahmenwerk auf und ergänzt zudem Constraints für weitere Eigenschaften. Dazu gehören andere Modifikatoren und Bezeichner von Deklarationen, Casts sowie Qualifizierungen von Referenzen. Diese können nun simultan mit Typen und Zugriffsmodifikatoren refaktorisiert werden.

Die Arbeit löst eine Reihe technischer und konzeptioneller Schwierigkeiten: Die Wertebereiche der Constraintvariablen werden spezifisch für jedes refaktorierte Programm auf ganzzahlige Bereiche transformiert. Dabei wird nun eine Indirektion realisiert, falls ein Programmelement als Wert der Eigenschaft eines anderen Elements auftritt. Die Arbeit gibt einen Algorithmus an, der selektiv genau die für eine Refaktorisierung benötigten Constraints berechnet. Von vielen möglichen Lösungen eines Constraintproblems kann per Vorgabe einer Zielfunktion eine optimale Lösung ermittelt werden.

Die Constraintsprache *Refacola* ist mithilfe des DSL-Rahmenwerks *Xtext* implementiert. *Refacola* trennt drei Arten deklarativer Spezifikationen: Programmiersprache, Regelsatz und Refaktorisierung. Für eine Programmiersprache wird spezifiziert, welche Arten von Programmelementen mit welchen Eigenschaften vorkommen und wie Programmabfragen aus-

sehen. Die Regelsatz-Spezifikation der Constraintregeln bleibt unabhängig von den Refaktorisierungen. Für jede Refaktorisierung ist schließlich zu definieren, welche Eigenschaften wie geändert werden und welche Daten für eine konkrete Anwendung der Refaktorisierung vorzugeben sind.

Ein Refacola-Compiler generiert den Quellcode der Plugins, die die Refaktorisierungswerkzeuge implementieren. Die Plugins laufen in den Entwicklungsumgebungen Eclipse, *MonoDevelop* oder *EiffelStudio*.

Die Arbeit präsentiert die Spezifikation der elementaren Refaktorisierungen RENAME, CHANGE-ACCESSIBILITY, CHANGE-DECLARED-TYPE und zeigt, wie daraus die CHANGE-METHOD-DECLARATION-Refaktorisierung zusammengesetzt wird.

Steimann et al. [35] führen die genannten elementaren Refaktorisierungen automatisch auf realen Eiffel-Programmen aus und demonstrieren daran die praktische Eignung des Rahmenwerks. Bei geeigneter Suchstrategie des Constraintlösers liegen die Laufzeiten auf einem gewöhnlichen PC in der Größenordnung von Millisekunden und erreichen nur ausnahmsweise bis zu drei Sekunden. Die Anzahl ausführbarer Refaktorisierungen nimmt durch die Kombination von Typ- mit Zugreifbarkeits- und weiteren Constraints zu.

Die Autoren schlussfolgern, der Ansatz, ein Refaktorisierungswerkzeug deklarativ zu spezifizieren und den Mechanismus dem ebenfalls deklarativ festgelegten Constraintsystem zu überlassen, sei tragfähig.

3 Grundlagen

Alle Constraintregeln auf alle möglichen Stellen eines zu refaktorisierenden Programms anzuwenden, führt zu einer sehr großen Anzahl von Constraints. Eine konkrete Refaktorisierungsoperation ist nur von einem Teil dieser Constraints beschränkt. Dieses Kapitel begründet, warum das so ist, und leitet aus einer graphentheoretischen Interpretation der Constraintmenge einen Algorithmus ab, der nur die benötigten Constraints berechnet. Es behandelt außerdem die Komplexität des Algorithmus und die Möglichkeiten, eine Implementierung des Algorithmus zu testen.

3.1 Constraintgraph

Ausgangspunkt der Überlegungen ist die komplette Menge an Constraints, die man erhielte, wendete man die Constraintregeln auf ein vorliegendes Programm erschöpfend an. Dabei sei unterstellt, die Constraintregeln seien korrekt und vollständig. Die erhaltene Menge der Constraints bildet ein Netzwerk, das sich als Graph auffassen und in dieser Darstellung analysieren lässt. Die folgenden Abschnitte führen das Modell ein und erklären, wie man von der gegebenen Constraintmenge zu einem handhabbaren Graphen kommt.

3.1.1 Constraintnetzwerk

Ein Einschränkungproblem wird als das Tripel $\langle X, \mathcal{D}, \mathcal{C} \rangle$ definiert. Dechter [6, S. 25] oder Bessiere [3] nennen das Tripel anschaulich ein *Constraintnetzwerk* (*constraint network*). Darin ist $X = \{x_1, \dots, x_n\}$ die Menge der Constraintvariablen³, $\mathcal{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_n\}$ die Menge ihrer zugeordneten Wertebereiche (*domains*) und \mathcal{C} die Menge der Constraints. Ein *Constraint* $c \in \mathcal{C}$ ist eine Relation auf einer Teilmenge der Constraintvariablen (vgl. Abschnitt A.1, S 58). Einschränkungprobleme, deren Constraintvariablen x_i ausschließlich endliche Wertebereiche \mathcal{D}_i aufweisen, werden als *Constraint-Satisfaction-Problem* (CSP) bezeichnet [14, S. 72]. Die Definition von CSP lässt beliebige Objekte als Elemente der Wertebereiche \mathcal{D}_i zu.

³Der Begriff *Constraintvariable* schließt wie üblich Variablen ein, deren Wertebereich auf einen einzelnen Wert beschränkt ist. Diese werden auch als *fixe Variable* (*fixed variable*) oder als *Konstante* bezeichnet.

Unter einer *Lösung* eines Constraintnetzwerks wird eine Wertebelegung der Constraintvariablen verstanden, die alle Constraints simultan erfüllt. Existiert keine Lösung, ist das Constraintnetzwerk *inkonsistent*. Ein Constraint heißt *gelöst*, wenn alle Kombinationen von Werten der Bereiche seiner Constraintvariablen das Constraint erfüllen.

Obwohl die Constraintmenge nur ein Element des Constraintnetzwerks ist, wird der Begriff *Constraintmenge* oft synonym für den Begriff *Constraintnetzwerk* oder den Begriff *Einschränkungsproblem* verwendet.

3.1.2 Programmelemente als Constraintvariable

Die vorliegende Arbeit verwendet anders als Steimann et al. [35] die Programmelemente als Constraintvariablen. Dieser Abschnitt begründet zunächst diese Entscheidung und erläutert anschließend ihre Konsequenzen. Ein ausführliches Beispiel zeigt der folgende Abschnitt 3.1.3 (S. 15). Abschnitt 6.4.4 (S. 49) diskutiert die Entscheidung.

Constraintvariable sind hier die Programmelemente $e \in DUR$, das heißt Deklarationen $d \in D$ und Referenzen $r \in R$ mit ihren Eigenschaften (vgl. Anhang A.1, S. 56). Die Constraints auf den Programmelementen mit $X = DUR$ und der Zuordnung $DUR = \{d_1, \dots, d_m, r_1, \dots, r_{n-m}\}$ bezeichnet diese Arbeit als *Modell-Constraints*, um sie von den Constraints über ganzzahlige, endliche Wertebereiche zu unterscheiden, die übliche Constraintlöser direkt verarbeiten können. Letztere werden hier *Integer-Constraints* genannt (vgl. Bild 5, S. 26).

Aus folgenden Gründen verwendet diese Arbeit Modell-Constraints mit Programmelementen als Constraintvariablen: Programmelemente repräsentieren die konzeptionellen Einheiten der Problemdomäne und verfügen über Identität. Sie sind die natürlichen Einheiten der Programmanalyse und bieten sich als Datenstruktur an, die die verschiedenen Eigenschaften eines Programmelements zusammenhalten. Programmelemente treten in den Constraints meist mit allen betrachteten Eigenschaften gemeinsam auf. Schließlich ist eine Transformation von Constraints eines Modells in Integer-Constraints zur Lösung des Constraintnetzwerks ohnehin erforderlich (vgl. Abschnitt 4.3, S. 27).

Programmelemente sind durch Eigenschaften charakterisiert (vgl. Anhang A.1, S. 56). Im Allgemeinen ist mehr als eine dieser Eigenschaften im Rahmen der hier betrachteten Refaktorisierung unabhängig von anderen Eigenschaften desselben Programmelements veränderlich. Entsprechend sind die Wertebereiche der Constraintvariablen Tupelmengen. Welche Eigenschaften der Programmelemente vorkommen und welche Tupel die Wertebereiche ausmachen, erläutern folgende Absätze. Zunächst geht es um Referenzen, anschließend um Deklarationen.

Eine Referenz $r \in R$ kommt in den Modell-Constraints mit der variablen Eigenschaft Ort $r.\lambda$ vor. Der Wertebereich der Eigenschaft besteht aus den bei der Refaktorisierung zugelassenen Orten der Referenz. Wie in Abschnitt A.1 (S. 57) angemerkt, ist die Eigenschaft Empfängertyp $r.\rho$ vom Ort $r.\lambda$ der Referenz, vom Ort der zugehörigen Deklaration und von weiteren Bedingungen abhängig. Die Eigenschaft Package $r.\pi$ ist abhängig vom Ort $r.\lambda$ der Referenz. Referenzen haben demnach nur eine unabhängige Eigenschaft, und zwar den Ort $r.\lambda$. Die Tupel des Wertebereichs einer Referenz bestehen folglich nur aus einer Komponente und können durch die Werte dieser Komponente ersetzt werden.

Ein Deklarationselement $d \in D$ kommt mit den unabhängigen Eigenschaften Ort $d.\lambda \in C$ und Zugriffsmodifikator $d.\alpha \in A$ in den Constraints vor. Der zugeordnete Wertebereich ist also die Menge der Paare $\{(c, a) \mid c \in C, a \in A\}$ von Ort und Zugriffsmodifikator oder eine Teilmenge davon, die eine Refaktorisierung zulässt. Die Refaktorisierung verlagert Classifier nicht. Deshalb ist die Eigenschaft Package $d.\pi$ durch die Eigenschaft $d.\lambda$ festgelegt. Die übrigen Eigenschaften sind unveränderlich. Hier haben also nur Constraintvariablen, die Felddeklarationen repräsentieren, einen mehrstelligen Wertebereich.

Die individuellen Wertebereiche jeder Constraintvariablen werden bei ihrer Erzeugung festgelegt. Sie reflektieren die Freiheitsgrade der konkreten PULL-UP-FIELD-Refaktorisierung.

3.1.3 Hypergraph und bipartiter Constraintgraph

Da manche Constraints mehr als zwei Constraintvariablen enthalten, repräsentiert das oben definierte Constraintnetzwerk einen Hypergraphen $H = \langle DUR, \mathcal{C} \rangle$ mit den Programmelementen DUR als Ecken, die über Hyperkanten der Constraints \mathcal{C} miteinander verbunden sind [6, S. 30 f.].

Diesen Hypergraphen schreibe ich der einfacheren Darstellung halber [21, S. 174] als äquivalenten bipartiten Graphen, der auch die Constraints $c \in \mathcal{C}$ als Ecken darstellt.

$$B_H = \langle \mathcal{C} \cup DUR, \{(c, e) \mid c \in \mathcal{C} \wedge e \in DUR \wedge e \in c\} \rangle$$

Die Kanten (c, e) dieses ungerichteten Graphen B_H verbinden die Programmelemente $e \in DUR$ mit den Constraints $c \in \mathcal{C}$, in denen sie vorkommen, symbolisiert durch den Ausdruck $e \in c$.

Ein Beispiel soll den Formalismus veranschaulichen. Bild 1 zeigt den Constraintgraphen in bipartiter Darstellung für folgendes Beispielprogramm. Die Kommentare geben die Bezeichnung der Programmelemente an.

```

1  class A { }           // d_A
2  class B extends A {  // d_B, r_A
3      int i= 0;        // d_i
4      int j= i;        // d_j, r_i
5      void m(){ j++; } // r_j
6  }
```

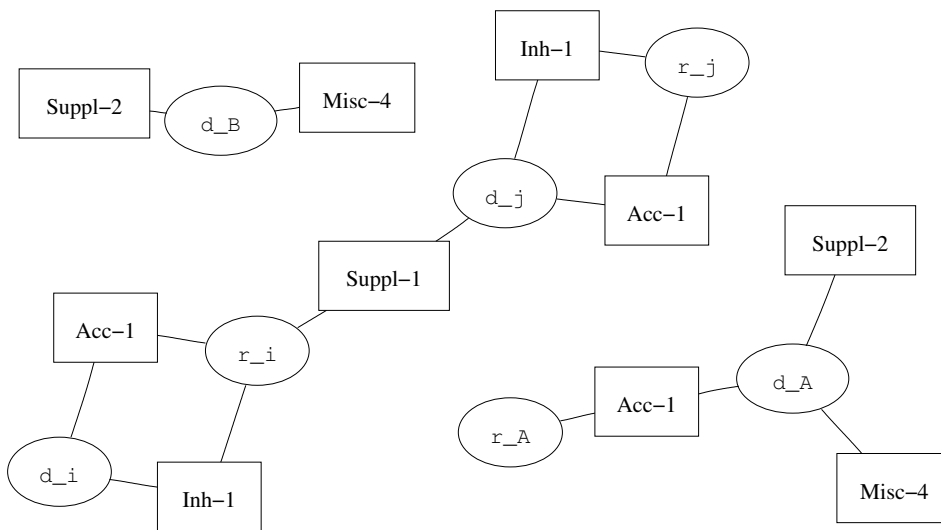


Bild 1 – Beispiel eines bipartiten Constraintgraphen

In dieser Darstellung sind Programmelemente als Ellipsen gezeichnet und Constraints als Rechtecke. Die Bezeichnung der Constraints in Bild 1 entspricht den Bezeichnungen im Anhang A (S. 59 ff.). Dabei steht jede Constraint-Ecke trotz gleicher Beschriftung für ein individuelles Constraint. Tabelle 1 (S. 16) stellt die Bezeichnungen der Programmelemente und ihre jeweils zugeordneten Wertebereiche zusammen. Damit ist das Tripel des Constraintnetzwerks, bestehend aus Programmelementen, Wertebereichen und Constraintmenge, vollständig.

Der Graph in Bild 1 ist nicht zusammenhängend. Er zerfällt in drei Komponenten. Das ist schon einmal ermutigend.

3.2 Komponenten

Der Constraintgraph eines zu refaktorisierenden Programms enthält eine sehr große Anzahl von Constraints. Die Frage ist, welche dieser Constraints für eine Refaktorisierung benötigt

Tabelle 1 – Programmelemente im Constraintgraphen aus Bild 1

Element	Bereich	Erläuterung
d_i	$\{(A, \textit{private}), \dots, (B, \textit{public})\}$	Feld i
d_j	dito	Feld j
r_i	$\{A, B\}$	Referenz auf i in Deklaration von j
r_j	$\{B\}$	Referenz auf j in Methode m
d_A	$\{\textit{private}, \dots, \textit{public}\}$	Klasse A
d_B	dito	Klasse B
r_A	$\{B\}$	Referenz auf A in Deklaration von B

werden. Dieser Abschnitt argumentiert, dass für eine PULL-UP-FIELD-Refaktorisierung nur *eine* zusammenhängende Komponente des Constraintgraphen benötigt wird. Er zeigt weiterhin, wie sich der Constraintgraph in noch kleinere Komponenten zerlegen lässt.

3.2.1 Fortpflanzung von Änderungen im Constraintgraphen

Grundlage der Argumentation ist folgende Beobachtung: Änderungen an Programmelementen pflanzen sich nur über Constraints von einem zum anderen Element fort. Programmelemente, die in verschiedenen Komponenten liegen, können sich gegenseitig nicht beeinflussen. Könnte eine Änderung in einer Komponente Auswirkungen auf Programmelemente in einer anderen Komponente haben, müsste es ein Constraint zwischen den Komponenten geben. Das widerspräche aber der Definition von Komponenten, die besagt, dass Kanten zwischen den Komponenten nicht existieren.

Das Ziel einer Refaktorisierung ist die Änderung weniger Programmelemente. Bei der untersuchten PULL-UP-FIELD-Refaktorisierung geht es nur um *ein* Feld, dessen Ort geändert werden soll. Diese Änderung kann nur Programmelemente berühren, die sich mit dem zu ändernden Element in der gleichen Komponente befinden. Damit ist präzise angegeben, welche Constraints für eine korrekte Refaktorisierung benötigt werden: Es sind genau die Constraints, die mit dem zu ändernden Programmelement gemeinsam eine Komponente bilden.

3.2.2 Eliminierung unveränderlicher Programmelemente

Je kleiner die Komponente, desto besser. Im Rest dieses Abschnitts geht es darum, den Constraintgraphen in kleinere Komponenten zu zerlegen. Dazu wird noch einmal das Programmbeispiel aus dem vorhergehenden Abschnitt bemüht.

Tabelle 1 (S. 16) gibt die Wertebereiche der Constraintvariablen für das Beispiel allgemein an. Eine konkrete Refaktorisierung schränkt diese Bereiche weiter ein. Angenommen, die Aufgabe sei, das Feld i von B nach A hochzuziehen, das Feld j jedoch unangetastet zu lassen. Diese Annahme beschränkt den Wertebereich von d_j auf $\{(B, \textit{package})\}$. Die Constraintvariable d_j ist also auf eine Konstante reduziert.

Die Werte von Konstanten können direkt in die Constraints eingesetzt werden und die entsprechenden Constraintvariablen auf diese Weise eliminiert werden [14, S. 68]. Streicht man aus dem Graphen in Bild 1 (S. 15) die auf Konstanten zurückgeführten Programmelemente d_j , r_j und r_A , dann zerfällt der Graph in noch kleinere Komponenten. Bild 2 (S. 17) zeigt den Graphen, aus dem die konstanten Programmelemente eliminiert sind. Constraints, deren Variablen ganz oder teilweise durch konstante Werte ersetzt wurden, sind darin mit einem Stern markiert.

Das zu refaktorisierende Programmelement d_i hängt in Bild 2 nur noch mit der Programmelement-Ecke r_i über die beiden Constraint-Ecken *Acc-1* und *Inh-1* zusammen. Das Beispiel zeigt, Eliminieren der Ecken konstanter Programmelemente kann zu geringerem Zusammenhang des Graphen, also zu einer größeren Anzahl kleinerer Komponenten führen.

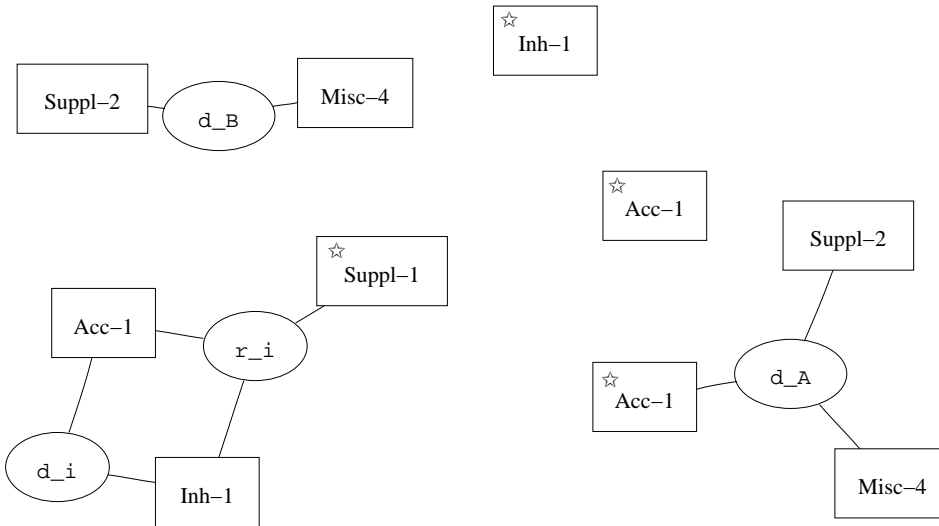


Bild 2 – Constraintgraph aus Bild 1, vermindert um konstante Programmelemente

Dieses Vorgehen ist allgemein anwendbar. Die Eliminierung von konstanten Programmelementen durch Substitution ihrer Werte in die Constraints führt zur Menge der variablen Programmelemente $V = \{v \in D \cup R \wedge |\mathcal{D}_v| > 1\}$ und damit zum reduzierten Constraintgraphen B_H^X , der neben den Constraints nur noch unter der betrachteten Refaktorisierung veränderliche Programmelemente als Ecken enthält:

$$B_H^X = \langle \mathcal{C} \cup V, \{(c, e) \mid c \in \mathcal{C} \wedge e \in V \wedge e \in c\} \rangle$$

Die Menge der Constraints in diesem Graphen ist die gleiche wie im ursprünglichen Graphen B_H . Nur sind unveränderliche Variable durch ihre konstanten Werte jeweils in die angrenzenden Constraints substituiert.

3.2.3 Eliminierung unveränderlicher Eigenschaften

Im Übrigen gelten die Überlegungen nicht nur für ganze Programmelemente, die ausschließlich konstante Eigenschaften haben, sondern sie können analog auf einzelne Eigenschaften übertragen werden.

Ein veränderliches Programmelement d_i kann konstante Eigenschaften haben. Allerdings muss wenigstens eine Eigenschaft veränderlich sein, damit für die Mächtigkeit seines zugeordneten Wertebereichs $|\mathcal{D}_i| > 1$ gilt, sonst wäre das Programmelement konstant.

Jede konstante Eigenschaft eines veränderlichen Programmelements kann durch Substitution aus den Constraints eliminiert werden. Sind auf diese Weise alle veränderlichen Eigenschaften eines Programmelements aus einem Constraint eliminiert, tritt das Programmelement in dem Constraint nicht mehr auf. Damit entfällt die Kante zwischen Programmelement und Constraint. Das kann wiederum zu geringerem Zusammenhang des Graphen führen.

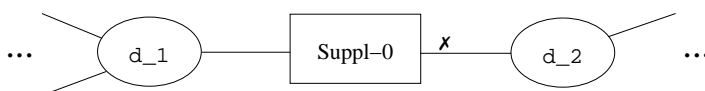


Bild 3 – Ausschnitt eines Constraintgraphen: Eliminierung einer unveränderlichen Eigenschaft

Bild 3 zeigt zur Illustration den Ausschnitt eines Constraintgraphen: Ein Feld d_1 solle in eine Klasse c_2 hochgezogen werden, die bereits ein gleichnamiges, unverlagerbares Feld d_2 deklariere. Nach Constraintregel *Suppl-0* gilt das Constraint $c : d_1.\lambda \neq d_2.\lambda$. Im Constraintgraphen gibt es also Kanten zwischen d_1 und c sowie zwischen c und d_2 . Da $d_2.\lambda = c_2$ konstant

ist, kann $d_2.\lambda$ durch Substitution von c_2 aus dem Constraint eliminiert werden. Es lautet dann $c : d_1.\lambda \neq c_2$. Damit verschwindet im Graphen die Kante zwischen dem Constraint c und dem Programmelement d_2 , die in Bild 3 mit einem Kreuz markiert ist.

3.2.4 Benötigte Constraints

Der vorhergehende Abschnitt 3.1 und dieser Abschnitt begründen zusammengefasst folgende Aussage: Sind die Constraintregeln vollständig, haben sie also alle Abhängigkeiten zwischen Programmelementen erfasst, können Änderungen an Programmelementen in einer zusammenhängenden Komponente des Constraintgraphen keine Auswirkungen auf Programmelemente in anderen Komponenten haben. Die für eine Refaktorisierung benötigten Constraints sind damit gerade diejenigen, die mit den durch die Refaktorisierung zu ändernden Programmelementen eine Komponente bilden.

3.3 Algorithmus und Datenstrukturen

Der vorhergehende Abschnitt hat das Problem, genau die notwendigen Constraints für die Refaktorisierung zu ermitteln, als Aufgabe reformuliert, die Komponente des Constraintgraphen zu ermitteln, zu der ein zu änderndes Programmelement gehört. Damit liegt der Algorithmus auf der Hand: Breiten- oder Tiefendurchlauf vom zu ändernden Programmelement aus liefert genau die Ecken seiner Komponente [1, S. 239]. Allerdings liegt der Constraintgraph noch gar nicht vor, sondern muss beim Durchlauf erst ermittelt werden. Dieser Abschnitt skizziert das Prinzip einer Lösung.

3.3.1 Tiefendurchlauf und Programmanalyse

Tiefendurchlauf hat gegenüber Breitendurchlauf den Vorteil, dass er ohne explizite Datenstruktur für die noch zu bearbeitenden Nachbarecken auskommt und sich elegant rekursiv aufschreiben lässt. Aus diesem Grund entscheide ich mich, den Algorithmus zum Tiefendurchlauf auf die vorliegende Situation zu anzupassen:⁴ Anstatt eine gegebene Adjazenzliste einer Ecke durchzugehen, werden hier die benachbarten Ecken ad hoc durch Programmanalyse ermittelt. Dabei ist nach Sorte der Ecke – Programmelement oder Constraint – zu unterscheiden.

Für jedes Programmelement wird die Liste der Constraintregeln durchgegangen und jeweils überprüft, ob die Regel Constraints generiert, in denen das Programmelement vorkommt. Für ein Constraint werden alle dem Constraint fehlenden Programmelemente im zu refaktorisierenden Programm aufgesucht. Werte konstanter Eigenschaften werden im Constraint eingesetzt, variable Programmelemente repräsentieren Nachbarn, bei denen die Suche fortgesetzt wird. Besuchte Programmelemente werden markiert und Constraints in einer Liste gespeichert.

Der in Bild 4 (S. 19) skizzierte Algorithmus *Tiefendurchlauf* entspricht der Vorlage aus dem Lehrbuch [1, S. 216], nur steht hier die Terminierungsbedingung in Zeile 4 am Anfang.

In Zeile 8 werden alle Programmelemente aus dem Programm \mathcal{P} aufgesucht, die nötig sind, ein Constraint zu erzeugen, das das Programmelement e enthält. Die Programmelemente e_i sind die anderen Nachbarn der Constraint-Ecke c . Die Anweisung der Zeile 8 führt also die Suche nach Programmelementen und eine mehr oder weniger aufwendige Programmanalyse aus. Der rekursive Algorithmus (Zeile 12) terminiert mit dem Grundfall (Zeile 4), weil die Zahl variabler Programmelemente, die über Constraints vom ersten Programmelement e aus erreichbar sind, endlich ist.

Der Algorithmus setzt stillschweigend voraus, dass das Programm \mathcal{P} die Information über das Ziel der Refaktorisierung trägt, die für die Programmanalyse in Zeile 8 benötigt wird.

⁴ Wie sich am Ende der Arbeit herausstellt, ist das eine unglückliche Entscheidung (Abschnitt 6.5, S. 50).

```

1 algorithm Tiefendurchlauf( $e, \mathcal{P}, \mathcal{C}$ )
2 input: Programmelement  $e$ , Programm  $\mathcal{P}$ , Menge von Constraints  $\mathcal{C}$  – zunächst leer
3 output: Menge von Constraints  $\mathcal{C}$  ergänzt um notwendige Constraints für  $e$  in  $\mathcal{P}$ 
4 if  $e$  gesehen or  $e$  konstant then
5     return
6 Markiere  $e$  als gesehen
7 for each Constraintregel  $R_c$  mit  $e$  in der Prämisse von  $R_c$  do
8      $T :=$  alle Tupel von Programmelementen  $(e_1, \dots, e_n)$  in  $\mathcal{P}$ , die  $c$  komplettieren
9     for each Tupel  $(e, e_1, \dots, e_n)$  aus den Tupeln in  $T$  do
10         Erzeuge Constraint  $c$  mit Parametern  $e, e_1, \dots, e_n$  und füge es  $\mathcal{C}$  hinzu
11         for each  $e_i$  in  $e_1, \dots, e_n$  do
12             Tiefendurchlauf( $e_i, \mathcal{P}, \mathcal{C}$ ).

```

Bild 4 – Algorithmus zur Ermittlung der notwendigen Constraints

Abschnitt 4.1 (S. 24) spezifiziert die Eingabe des Algorithmus und die Programmanalyse im Detail.

3.3.2 Speichern des Graphen

Der ermittelte Constraintgraph wird in Form einer Kantenliste gespeichert. Und zwar sind es die Kanten des Hypergraphen, also die Constraints. Zur Erinnerung: Der bipartite Graph ist lediglich eine andere Schreibweise und graphische Darstellung des Hypergraphen, der das Constraintproblem repräsentiert (vgl. Abschnitt 3.1, S. 13).

Eine Kantenliste ist hier eine geeignete Datenstruktur, da der Constraintgraph nicht weiter traversiert oder modifiziert wird. Über die Liste wird hingegen mehrfach iteriert.

Obwohl die Kantenliste des Hypergraphen nur eine andere Sicht auf die Constraintmenge ist, wähle ich zu ihrer Speicherung keine Menge. Der Aufwand, über eine Menge zu iterieren, wäre unnötig hoch und die Mengeneigenschaften der Datenstruktur werden gar nicht genutzt.

3.4 Komplexität

Die Laufzeitkomplexität des Algorithmus hängt zunächst mit der Anzahl der Programmelemente m einer Komponente zusammen. Jedes Programmelement wird genau einmal aufgesucht. Die Anzahl der Constraintregeln (Bild 4, Zeile 7) ist fest. Die Anzahl der Tupel wächst exponentiell mit der Stelligkeit n der Constraints. Das ergäbe eine Komplexität des Algorithmus von $O(m^n)$. Das stimmt mit der Erwartung überein, denn die Laufzeitkomplexität des Tiefendurchlaufs ist linear von der Anzahl der Kanten der Komponente abhängig [1, S. 216].

Leider müssen in Zeile 8 im schlechtesten Fall jedoch die Programmelemente des gesamten Programms \mathcal{P} und dann auch noch in ihren Kombinationen angesehen werden, da von vornherein ja nicht klar ist, welche der p Programmelemente des Programms \mathcal{P} zur Komponente gehören. Damit ergibt sich eine Komplexität der Ordnung $O(mp^{n-1})$. Die in Abschnitt A (S. 59 ff.) aufgeführten Constraints sind maximal 3-stellig. Somit erhält man eine Laufzeitkomplexität von $O(mp^2)$; darin ist m die Anzahl von Programmelementen der Komponente und p die Anzahl von Programmelementen des Programms.

Ganz so dramatisch, wie es klingt, ist die Situation vermutlich nicht. Die konstanten Faktoren sind klein. Das hat folgende Gründe: Die Elemente der Tupel können häufig, aber nicht immer, in konstanter Zeit ermittelt werden. So ist beispielsweise die an eine Referenz gebundene Deklaration direkt gegeben. Das wirkt wie eine Verminderung der Stelligkeit des Constraints. Werden die Programmelemente separat nach ihrer Art gespeichert, wird die Zahl jeweils abzusuchender Programmelemente und damit die Mantisse der Komplexität kleiner.

An der Laufzeitkomplexität von $O(mp^2)$ ändern kleine Konstanten jedoch nichts.

3.5 Redundante Constraints

Gleich beim Graphendurchlauf zur Constraintermittlung könnte Knoten- und teilweise Kantenkonsistenz hergestellt oder wenigstens geprüft werden (vgl. Abschnitt 3.6, S. 20). Die vorliegende Arbeit setzt diese Konsistenztechniken jedoch nicht ein (vgl. Abschnitt 1.2, S. 6). Sie eliminiert allerdings redundante Constraints. Warum und wie das geht, beschreibt dieser Abschnitt.

Redundant sind Constraints einer Constraintmenge, die keinen Einfluss auf die Lösungsmenge haben [14, S. 254 f.]. Werden sie nicht gezielt zur Verbesserung der Lösungssuche genutzt, schaden sie nur. Gelöste Constraints (vgl. Abschnitt 3.1, S. 13) sind immer redundant. Wo möglich, werden sie hier gar nicht erst erzeugt und der Durchlauf wird über diese Constraints hinaus nicht fortgesetzt.

Ein Constraint nach Constraintregel *Suppl-2* schreibt vor, dass der Zugriffsmodifikator eines Classifiers nicht vermindert werden darf. Dann kann aber auch ein Constraint nach Regel *Acc-1* für unbewegte Referenzen auf Classifier niemals verletzt werden. Es muss für andere Referenzen auf den gleichen Classifier also gar nicht erst erzeugt werden. Das ist besonders erfreulich, weil damit die Suche nach allen Referenzen auf eine gegebene Classifier-Deklaration unnötig ist.

Constraints können auch ganz unwirksam sein. Sie werden dann gar nicht erst generiert: Das Constraint nach Regel *Misc-8* sorgt dafür, dass Member von Interfaces immer *public* bleiben. Die Constraintregeln *Acc-1* und *Acc-2* brauchen für diese Member also gar nicht erst angewendet zu werden, weil Erfüllen des Constraints nach Regel *Misc-8* ein Erfüllen der Constraints nach Regeln *Acc-1* und *Acc-2* impliziert. Durch Hochziehen kann auch die Vererbung dieser Member nicht verloren gehen, sodass hier auch die Regel *Inh-1* für Member von Interfaces überflüssig ist. Man könnte davon sprechen, dass in diesem Fall Knotenkonsistenz mit dem Constraint nach Regel *Misc-8* ausgenutzt wird, um redundant konsistente Kanten, die Constraints nach Regeln *Acc-1*, *Acc-2* und *Inh-1*, zu eliminieren.

Thies [38] schlägt noch strengere Kriterien für die Selektion von Constraints vor (vgl. Abschnitt 6.5.4, S. 52). Eine Realisierung dieser strengeren Selektion setzt Einschränkungsfortpflanzung parallel zur Constraintterzeugung voraus, die die vorliegende Arbeit jedoch nicht durchführt (Abschnitt 1.2, S. 7).

Die Constraintterzeugung könnte wenigstens sofort terminieren, wenn das gemäß Refaktorisierungsziel geänderte Programmelement bereits alle Constraints erfüllt, in denen es vorkommt. Diese Prüfung auf Konsistenz mit der Ausgangsbelegung ist beim Tiefendurchlauf jedoch nicht während des Durchlaufs möglich, sondern erst im Anschluss daran (vgl. Abschnitt 6.5.2, S. 51).

3.6 Lösen der Constraintmenge

Die Constraintmenge für ein Refaktorisierungsproblem ist mit dem Algorithmus nach Bild 4 (S. 19) zu gewinnen. Es fehlen aber noch Lösungen, die das Constraintnetzwerk erfüllen. Dieser Abschnitt grenzt Verfahren zur Lösung von Constraintproblemen ab, erläutert, warum ein etablierter Constraintlöser eingesetzt wird, und benennt die Voraussetzungen dafür.

3.6.1 Lösungsverfahren

Verfahren zur Lösung von Einschränkungsproblemen auf diskreten, endlichen Wertebereichen beruhen auf Suchverfahren und auf Konsistenztechniken [14, S. 73].

Als Suchverfahren werden verschiedene Varianten der *Suche mit Rücksetzen* (*backtracking*) eingesetzt, oft in Kombination mit Konsistenztechniken. Für spezielle Einschränkungsprobleme kommen auch Suchverfahren in Frage, die unter der Bezeichnung *lokale Suche*

(*local search*) bekannt sind (vgl. Abschnitt 6.5, S. 50). Gemeinsam ist reinen Suchverfahren, dass sie das Einschränkungproblem unverändert lassen.

Für *Konsistenztechniken* sind verwirrend unterschiedliche Bezeichnungen gebräuchlich, darunter *Einschränkungsfortpflanzung* (*constraint propagation*) und *Inferenz* [3]. Konsistenztechniken dienen dazu, den großen Suchraum von Constraintproblemen einzuschränken und die Suche damit zu beschleunigen. Gelegentlich liefern Konsistenztechniken bereits eine Lösung oder zeigen, dass keine Lösung existiert. Populär sind Verfahren, die Knoten- oder Kantenkonsistenz herstellen. Konsistenztechniken schränken den Wertebereich von Constraintvariablen ein, fügen dem Problem weitere Constraints hinzu oder verändern die Struktur des zu lösenden Einschränkungproblems [6, S. 9 f.].

3.6.2 Constraintlöser

Als *Constraintlöser* (*constraint solver*) bezeichnet man eine Bibliothek von Tests und Operationen auf Constraintmengen, mit denen man Erfüllbarkeit prüfen und Lösungen berechnen kann [14, S. 60].

In manchen Fällen könnte eine Lösungssuche gleich beim Durchlauf zur Constraintezeugung eine Lösung des Constraintnetzwerks finden. Im Allgemeinen muss oder möchte man jedoch einen Constraintlöser einsetzen, und zwar aus folgenden Gründen: Liefert eine einfache Suche keine Lösung, müssen fortgeschrittenere Verfahren eingesetzt werden, um überhaupt eine Lösung zu erhalten oder aber um nachzuweisen, dass es keine Lösung gibt. Häufig möchte man selbst in Fällen, in denen man eine Lösung bereits durch einfache Suche ermitteln konnte, alle anderen Lösungen berechnen oder feststellen, dass keine weiteren Lösungen existieren. Eine etablierte Bibliothek hat darüber hinaus den Vorteil, mit hoher Sicherheit korrekte Ergebnisse zu liefern.

3.6.3 Transformation auf ganzzahlige Wertebereiche

Die zu lösende Aufgabe ist ein Einschränkungproblem auf endlichen, diskreten Wertebereichen. Bibliotheken zur Lösung derartiger Aufgaben arbeiten meist mit Constraintvariablen auf Bereichen, die als Intervalle ganzer Zahlen definiert sind. Die in Abschnitt 3.1 (S. 13) definierten Modell-Constraints müssen also in Integer-Constraints transformiert werden. Das heißt, Wertebereiche der Programmelemente müssen auf Intervalle ganzer Zahlen abgebildet werden. Für eine konkrete PULL-UP-FIELD-Refaktorisierung sind die Bereiche der Constraintvariablen und die Definitionsbereiche der Funktionen so klein, dass die erforderlichen Abbildungen jeweils berechnet und extensional angegeben werden können. Wie das geschieht, erläutern folgende Absätze.

Die Abbildung $M_A : A \rightarrow \{0, \dots, 3\}$ übersetzt die Zugreifbarkeit von Deklarationselementen. Sie ist schlicht definiert als

$$M_A \equiv \{(private, 0), (package, 1), (protected, 2), (public, 3)\}.$$

Der Ort von Feldern ist entweder ihr ursprünglicher Ort oder der Ziel-Classifer $d_t \in C$, in den ein Feld hochgezogen werden soll. Das ergibt sich aus der Spezifikation der untersuchten PULL-UP-FIELD-Refaktorisierung (vgl. Abschnitt 4.1.2, S. 24): Felder werden hochgezogen oder bleiben an dem Ort, an dem sie vor der Refaktorisierung waren. Werden Felder hochgezogen, dann werden sie ausschließlich in den Ziel-Classifer hochgezogen und an keinen anderen Ort verlagert. Für jedes bewegliche Programmelement e gibt es entsprechend eine Abbildung $M_e : B \rightarrow \{0, 1\}$, definiert als

$$M_e \equiv \text{if } e.\lambda = \text{old}(e.\lambda) \text{ then } 0 \text{ else } 1,$$

lesbar als „hochgezogen“. Diese Abbildung ist einfacher als eine, die alle vorkommenden Orte ganzzahlig kodiert. Eine zweiwertige Abbildung vereinfacht auch, die vom Ort abhängigen Eigenschaften und Prädikate zu tabellieren.

Die benötigte Eigenschaft Empfängertyp $r.\rho$ von Referenzen, die Funktion $\alpha : C \times C \rightarrow A$ und Prädikate wie „inherits“ werden in Abhängigkeit der vorkommenden Programmelemente für jedes Constraint tabelliert und für jede Art von Constraint in Ausdrücke übertragen, die ein Constraintlöser verarbeiten kann.

Die Abbildungen M_A und M_e sind bijektiv. Ihre Umkehrung wird für die Rücktransformation einer Lösung des Constraintnetzwerks in eine Belegung der Eigenschaften der Programmelemente verwendet.

Technische Details der Übertragung der Modell-Constraints in die vom Constraintlöser verarbeitbaren Integer-Constraints hängen stark von der gewählten Bibliothek ab. Sie werden in Abschnitt 4.5 (S. 31) besprochen.

3.7 Testen des Algorithmus

Die Begründung des Algorithmus zur selektiven Constraintgenerierung (Bild 4, S. 19) benutzt folgende Aussagen: (1) Tiefendurchlauf liefert exakt die Ecken einer Komponente des Constraintgraphen. (2) Constraintnetzwerke, die keine gemeinsamen Variablen enthalten, sind voneinander unabhängig. Diese Aussagen bedürfen keines weiteren Beweises.

Nachzuweisen wäre hingegen die Hypothese, dass der Algorithmus durch Programmanalyse (Bild 4, Zeile 8) korrekt alle benachbarten Ecken und sonst keine ermittelt. Diesen Beweis kann die vorliegende Arbeit nicht erbringen, weil der Aufwand zu hoch wäre und der Gegenstand der Programmanalyse nicht formal spezifiziert ist. Dieser Abschnitt erläutert, wie und in wie weit der Algorithmus wenigstens empirisch geprüft werden kann. Tests können eine formale Verifikation jedoch nicht ersetzen.

3.7.1 Test-Ansätze

Der Algorithmus kann ganz gewöhnlich getestet werden, indem man ihn auf eine bestimmte Eingabe anwendet und seine Ausgabe mit der erwarteten Ausgabe vergleicht. Ein Testfall besteht aus einem Programm, einem hochzuziehenden Feld und einem Ziel-Classifer aus dem Programm sowie einer erwarteten Constraintmenge. Der Vorteil dieses Unit-Testens ist, dass die Tests den Algorithmus separat für sich prüfen. Von Nachteil ist, dass nur unrealistisch kleine Programme als Testbasis dienen können, da die erwartete Constraintmenge von Hand konstruiert werden muss. Art und Anzahl solcher Testfälle sind wegen des erheblichen Aufwands ihrer Konstruktion begrenzt.

Ein Ansatz für Systemtests von Refaktorisierungswerkzeugen ist, eine Refaktorisierungsaufgabe vorzugeben und das Ergebnis der Refaktorisierung in Form des transformierten Programms mit einem vorher bestimmten erwarteten Ergebnis zu vergleichen. Geschickter ist es, diese Testfälle nach vorgegebenen Kriterien in verschiedenen Varianten automatisch zu generieren [5]. Dieses Vorgehen hat den Vorteil, dass Fälle abdeckbar sind, die in der Praxis selten vorkommen. Der Nachteil ist, dass man kaum alle Möglichkeiten bedenken wird.

Diesem Problem begegnen Bouillon et al. [4] und weitere Arbeiten aus Hagen [15, 20, 35, 36], in dem sie das zu testende Refaktorisierungswerkzeug automatisch auf alle denkbare Stellen von realen Programmen anwenden und die Ergebnisse der Refaktorisierung automatisch prüfen. Refaktorierte Programme, die Probanden, müssen weiterhin fehlerfrei kompilieren und keiner der Unit-Tests der Probanden darf nach der Refaktorisierung fehlschlagen. Das Bestehen der Unit-Tests zeigt, dass die Refaktorisierung die Bedeutung des Probanden, die seine Unit-Tests ja spezifizieren, nicht verändert hat [33].

Eine weitere Möglichkeit, den Algorithmus zu prüfen, besteht in einer Art Regressionstest. Das auf dem Algorithmus zur selektiven Constraintgenerierung basierende Refaktorisierungswerkzeug wird mit einem Werkzeug verglichen, das Constraints erschöpfend generiert. Beide Implementierungen müssen die gleichen Refaktorisierungen ermöglichen. Abweichungen können ganz unterschiedliche Gründe haben. Ihre Ursache ist jeweils nur im Einzelfall zu klären.

3.7.2 Fehlersymptome und ihre Ursachen

Folgende Kriterien erfüllt eine Implementierung des Algorithmus zur selektiven Constraint-erzeugung im Systemtest, vorausgesetzt Implementierung *und* Algorithmus sind korrekt. Nicht-erfüllen eines dieser Kriterien lässt die jeweils angegebenen Fehler als Ursache vermuten.

- Die Menge der erzeugten Constraints muss mit der Variablenbelegung, die der Situation vor der Refaktorisierung entspricht, erfüllt sein. Andernfalls wurden falsche Constraints oder eine falsche Belegung der Variablen berechnet.
- Existiert eine Lösung der Constraintmenge für eine Refaktorisierungsoperation, müssen die vor der Refaktorisierung erzeugten Constraints auch nach Ausführen der Refaktorisierung erfüllt sein. Ansonsten hat der Algorithmus eine inkonsistente oder unvollständige Constraintmenge geliefert, ist die Berechnung der Belegung oder die Transformation von Modell- in Integer-Constraints oder die Rücktransformationen der Lösung nicht richtig (vgl. Bild 5, S. 26).
- Die Erfüllbarkeit der Constraints muss in beiden Darstellungen – der Modell- und der Integer-Constraints – identisch sein. Andernfalls ist die Transformation der Modell- in Integer-Constraints oder die Rücktransformation der Lösung falsch.
- In Fällen, in denen keine Lösung der Constraintmenge existiert, darf die Refaktorisierung tatsächlich nicht fehlerfrei ausführbar sein. Sonst wurden fehlerhafte, zu strenge Constraints erzeugt oder die Transformation der Modell- in Integer-Constraints ist falsch.
- Jede Lösung der Constraintmenge muss zu einem Programm führen, das ohne Fehler kompiliert. Andernfalls wurden eine unvollständige Constraintmenge oder falsche Constraints erzeugt oder die Transformation der Modell- in Integer-Constraints bzw. die Rücktransformation ist falsch.
- Jede Lösung der Constraintmenge muss zu einem Programm führen, dessen Testfälle genauso fehlerfrei passieren wie vor der Refaktorisierung. Ist diese Bedingung nicht erfüllt, kommen die gleichen Gründe infrage wie für die zuvor genannten Übersetzungsfehler.

Die Gründe für ein Verletzen der Kriterien sind erwartungsgemäß nicht eindeutig. Ein Nicht-erfüllen der Kriterien kann jedoch auch noch ganz andere Ursachen haben:

- Die Auswahl der Constraintregeln in Anhang A (S. 59 ff.) ist unvollständig oder die Regeln sind schlicht falsch.
- Die Umsetzung der Lösung von den Modell-Constraints in eine Änderung des Quellcodes ist fehlerhaft.
- Es ist eine Situation aufgetreten, die von den Constraintregeln nicht abgedeckt wird.

Die tatsächliche Ursache für ein Versagen der Implementierung des Algorithmus kann also dem Algorithmus zur Constraint-erzeugung nur durch systematische Analyse im Einzelfall zugeordnet werden.

Dieser Abschnitt hat Möglichkeiten eruiert, den Algorithmus zu testen. Auswahl und Durchführung von Tests behandelt Abschnitt 4.7 (S. 35). Festzuhalten bleibt, so umfangreich Tests auch angelegt sein mögen, „Tests können nur Vorhandensein von Fehlern zeigen; Fehlerfreiheit beweisen sie nie.“ [7, S. 6]

4 Implementierung

Das vorhergehende Kapitel hat die Aufgabe herausgearbeitet, einen Tiefendurchlauf auf einem Graphen zu implementieren und das Ergebnis zu testen. Das Besondere des Algorithmus ist, dass der Graph erst während des Durchlaufs durch eine Analyse des zu refaktorisierenden Programms entsteht. Die Implementierung des Algorithmus muss außerdem in ein Refaktorisierungswerkzeug eingebunden werden. Ausgehend von Zielen und Randbedingungen formuliert dieser Abschnitt den Kern des Lösungsansatzes und beschreibt die Lösung der Teilaufgaben.

4.1 Anforderungen

Der in Abschnitt 3.3 (S. 18) hergeleitete Algorithmus spezifiziert bereits funktionale Anforderungen an die Implementierung. Weitere Anforderungen ergeben sich aus dem Ziel, folgende Fragen zu klären, die bereits in der Aufgabenstellung (Abschnitt 1.2, S. 6) bzw. der Darstellung der Möglichkeiten, den Algorithmus zu testen (Abschnitt 3.7, S. 22), benannt wurden.

- Liefert der Algorithmus widerspruchsfreie und vollständige Constraintmengen?
- Sind die Ergebnisse einer auf dem Algorithmus basierenden PULL-UP-FIELD-Refaktorisierung korrekt?
- Ist eine auf dem Algorithmus basierende Implementierung einer Refaktorisierung praktikabel in Bezug auf Laufzeit- und Speicherplatzbedarf?

Damit eine Implementierung diesem Ziel gerecht wird, muss sie richtig, zuverlässig und nachprüfbar sein. Sie darf Anforderungen an Laufzeit- und Speicherplatzeffizienz nicht vernachlässigen, damit sie skalierbar wird. Dabei soll die Implementierung gut lesbar und änderbar sein, damit sie um neue Constraintregeln erweiterbar bleibt, die im Lauf der Entwicklung gefunden werden.

4.1.1 Randbedingungen

Die Aufgabenstellung gibt als Randbedingung vor, die Constraintermittlung und die Refaktorisierung als Eclipse-Plugin auf Basis der *Java-Development-Tools* (JDT) und des *Language-Tool-Kits* (LTK) zu implementieren. Das Plugin soll auf Eclipse in der Version 3.5 (*Galileo*) lauffähig sein und möglichst keine interne Programmierschnittstelle (*application programming interface*, API) verwenden, damit die Implementierung auch auf neueren Eclipse-Versionen läuft.

4.1.2 Spezifikation der Refaktorisierung

Die Untersuchung wird am Beispiel einer PULL-UP-FIELD-Refaktorisierung von Java-Programmen durchgeführt. Folgende Absätze präzisieren die Definition der Refaktorisierung.

Eine PULL-UP-FIELD-Refaktorisierung verlagert die Deklaration eines statischen oder nicht-statischen Felds vom deklarierenden Subtypen in einen seiner Supertypen [9, S. 320]. Annotationen und der Initialisierungsausdruck werden als Teil der Felddeklaration betrachtet und ebenfalls verlagert. Damit die Refaktorisierung allgemeiner anwendbar ist, dürfen beim Hochziehen zusätzliche Änderungen vorgenommen werden:

- Andere Felder dürfen ebenfalls hochgezogen werden: Das Hochziehen ist auf solche Felder beschränkt, die entweder im gleichen Subtypen deklariert sind, von dem aus hochgezogen werden soll, oder die in Typen deklariert sind, die sich in der Typhierarchie zwischen dem Subtypen und dem Supertypen befinden, in den hochgezogen werden soll.
- Zugriffsmodifikatoren von Feldern dürfen beliebig geändert werden.

- Zugriffsmodifikatoren von Classifiern dürfen erhöht werden. *Erhöhen* eines Zugriffsmodifikators bedeutet dabei, dass das ausgezeichnete Programmelement von mehr Orten aus zugreifbar wird.

Die Möglichkeit der zusätzlichen Änderungen kann durch entsprechende Constraints wieder eingeschränkt werden.

Die Angabe eines hochzuziehenden Felds und des *Ziel-Classifiers*, in den das Feld verlagert werden soll, spezifiziert die Eingabe. Mit dem Feld ist auch der *Quell-Classifier* gegeben, aus dem das Feld hochgezogen werden soll und damit implizit das zu refaktorisierende Programm. Es ist das Programm, in dem sich das hochzuziehende Feld befindet.

Folgende Aufzählung nennt die grundlegenden Vorbedingungen für die Anwendung der Refaktorisierung:

- Der Ziel-Classifier ist Supertyp des Quell-Classifiers, der das hochzuziehende Feld deklariert.
- Quell- und Ziel-Classifier sind Teil des zu refaktorisierenden Programms, das in den zu refaktorisierenden Teilen als modifizierbarer Quellcode ungepackt vorliegt.
- Das Programm ist ein gültiges, d. h. fehlerfrei kompilierendes, Java-Programm gemäß der Java-Sprachspezifikation in ihrer 3. Auflage [12], also in Java-Version 1.5 oder 1.6.

Ausgabe der Refaktorisierung ist das refaktorierte Programm. Es gelten folgende Nebenbedingungen: Ist die Refaktorisierung ausführbar, dann ist das Feld im refaktorierten Programm in den angegebenen Ziel-Classifier hochgezogen, möglicherweise sind die oben spezifizierten zusätzlichen Änderungen erfolgt, das Programm ist fehlerfrei kompilierbar und seine Bedeutung ist erhalten – mit den im Anhang A.9 (S. 68) angegebenen Grenzen. Ist die Refaktorisierung nicht ausführbar, bleibt das Programm unverändert.

4.2 Lösungsansatz

Der Lösungsansatz für die Aufgabe, den Tiefendurchlauf und die Konstruktion des Graphen durch Programmanalyse für die Constraintterzeugung in einem Zug durchzuführen, beruht im Kern auf folgenden einfachen Ideen:

- Programmelemente und ihre Eigenschaften werden als Erweiterung des Eclipse-Java-Modells durch Komposition implementiert. Die Programmanalyse verwendet vorzugsweise die *Search-Engine* und das Modell der Typhierarchien der JDT. Sie greift auf das Modell des abstrakten Syntaxbaums (AST) und seiner Bindungen nur dann zurück – und dies auch nur temporär –, wenn die Auflösung des Java-Modells nicht ausreicht.
- Die Suche nach den Programmelementen, die ein gegebenes Modell-Constraint vervollständigen, wird auf meist kleine Bereiche des Programms beschränkt. Diese abzusuchenden Programmteile werden im Folgenden *Einflussbereiche* genannt. Sie sind jeweils durch ein gegebenes Programmelement, die Wertebereiche seiner Eigenschaften und die betrachtete Constraintregel bestimmt.
- Das Constraintnetzwerk wird in Form einer Kantenliste des Constraintgraphen gespeichert, was Programmanalyse und Constraintlösung weitgehend entkoppelt. Zur Lösung werden die ermittelten Modell-Constraints in Integer-Constraints transformiert, die ein Constraintlöser verarbeiten kann.

Bild 5 (S. 26) zeigt die Repräsentation des zu refaktorisierenden Programms auf den verschiedenen Ebenen. Rechtecke symbolisieren die Modell-Ebenen, abgerundete Rechtecke die Transformationen zwischen den Ebenen und Pfeile markieren den Datenfluss.

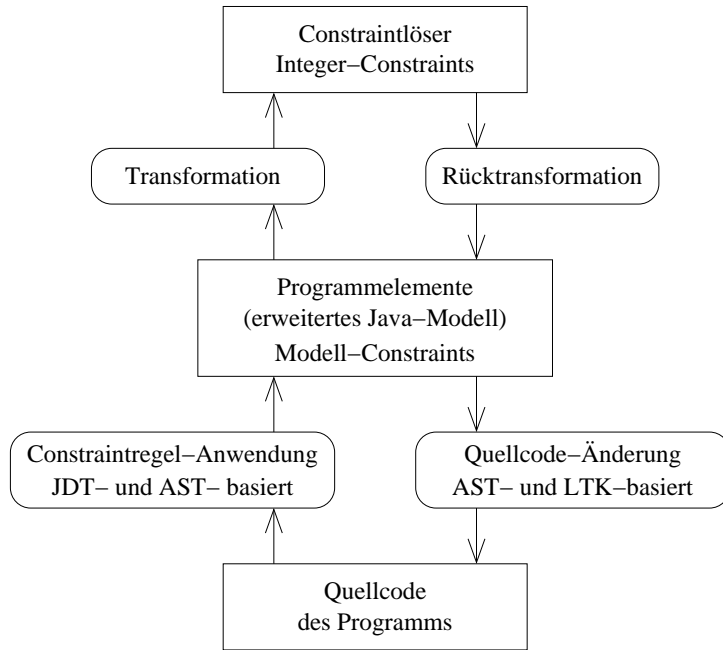


Bild 5 – Ebenen der Repräsentation eines zu refaktorisierenden Programms

Vom Quellcode führt die Anwendung der Constraintregeln auf das zu refaktorisierende Programm (Abschnitt 3.3, S. 18) zu den Modell-Constraints, deren Variablen die Java-Modell-basierten Programmelemente sind (Abschnitt 3.1, S. 13). Die Modell-Constraints werden in Integer-Constraints transformiert, die ein Constraintlöser verarbeiten kann (Abschnitt 3.6, S. 21). Eine Lösung des Constraintproblems wird in Eigenschaften der Programmelemente zurücktransformiert. Die Quellcode-Änderung mithilfe des AST im Rahmen des LTK führt zum refaktorisierten Programm. Die beiden letzten Schritte von der Lösung zum refaktorisierten Programm zusammengefasst bezeichnen Steimann und Thies [36] als *zurückschreiben der Lösung*.

Die Implementierung der gesamten PULL-UP-FIELD-Refaktorisierung erfordert neben der Implementierung des Algorithmus zur Constraintterzeugung per Tiefendurchlauf (vgl. Bild 4, S. 19) weitere Schritte, die Bild 6 noch einmal im Zusammenhang des Ablaufs wiedergibt. Die Erläuterung des Bilds verweist jeweils auf den Abschnitt dieses Kapitels, der die Implementierung des Schritts behandelt.

```

1 algorithm PULL-UP-FIELD( $d, c, \mathcal{P}$ )
2 input: Hochzuziehendes Feld  $d$ , Ziel-Classifer  $c$  und Programm  $\mathcal{P}$ 
3 output: Refaktorisiertes Programm  $\mathcal{P}^*$ 
4 assert  $\mathcal{P}$  kompiliert fehlerfrei  $\wedge d.\lambda, c \in \mathcal{P} \wedge d.\lambda <_I c \wedge d.\lambda, c$  modifizierbar
5  $\mathcal{C} := \emptyset$ 
6 Tiefendurchlauf( $d, \mathcal{P}, \mathcal{C}$ ) { erzeugt Modell-Constraints und akkumuliert sie in  $\mathcal{C}$  }
7  $CP :=$  Transformation von  $\mathcal{C}$  in das Modell des Constraintlösers
8  $SP :=$  Lösung des Constraintproblems  $CP$  berechnet vom Constraintlöser
9 if  $SP$  ist eine Lösung von  $CP$  then
10      $D^* \cup R^* :=$  Rücktransformation der Lösung  $SP$  in Programmelement-Eigenschaften
11      $\mathcal{P}^* :=$  Änderung des Quellcodes gemäß geänderten Eigenschaften von  $D^* \cup R^*$ 
12 else
13     fail.
  
```

Bild 6 – Schema einer constraintbasierten PULL-UP-FIELD-Refaktorisierung

Die Schritte des algorithmischen Schemas in Bild 6 (S. 26) sind: Prüfen allgemeiner Vorbedingungen für die konkrete Refaktorisierung (Zeile 4), die Erzeugung der Modell-Constraints per Tiefendurchlauf (Zeile 6, Abschnitt 4.4, S. 29), eine Transformation der Modell-Constraints in Integer-Constraints (Zeile 7, Abschnitt 4.5, S. 31), Ermitteln einer Lösung des Constraintnetzwerks (Zeile 8), Rücktransformation der Lösung in geänderte Eigenschaften der Programmelemente, die sich in eine Quellcode-Änderung übersetzen lässt (Zeile 10, Abschnitt 4.5, S. 33), und schließlich die entsprechende Änderung des Quellcodes (Zeile 11, Abschnitt 4.6, S. 33). Hat das Constraintnetzwerk keine Lösung, scheitert die Refaktorisierung (Zeile 13) und das Programm bleibt unverändert.

4.3 Erweiterung des Eclipse-Java-Modells

Für die Implementierung ist eine geeignete Repräsentation der in Abschnitt 3.1 (S. 13) eingeführten Programmelemente gesucht, die in den Modell-Constraints vorkommen. Dieser Abschnitt nennt einige Varianten und begründet die Entscheidung für eine Erweiterung der Elemente des Eclipse-Java-Modells, das im Folgenden wie in den JDT einfach *Java-Modell* genannt wird. Der zweite Teil dieses Abschnitts geht auf technische Details ihrer Implementierung ein.

4.3.1 Varianten und Entscheidung

Compiler repräsentieren Programme oder Teile davon in Form abstrakter Syntaxbäume, die um Bindungen von Referenzen an ihre Deklarationen angereichert sind. Im Allgemeinen werden nur kleine Teile eines Programms als AST im Arbeitsspeicher gehalten, weil diese Darstellung sehr viel Speicherplatz benötigt. So empfehlen Azad und Thomann [2], gleichzeitig höchstens einen Java-File (*compilation unit*) in Form eines AST zu referenzieren. Eine Repräsentation der Programmelemente auf Basis des AST oder seiner Knoten scheidet deshalb aus.

Kegel [19] verwendet eindeutige Schlüssel der Bindungen des AST, um Programmelemente bzw. Constraintvariable zu identifizieren. Mithilfe der Schlüssel lassen sich die ursprünglichen Bindungen wiederherstellen, allerdings mit beachtlichem Aufwand an Rechenzeit. Die Bindungen erlauben dann Zugriff auf alle benötigten Eigenschaften der Programmelemente. Da Kegel bereits bei der Programmanalyse eine Lösung der Constraintmenge ermitteln kann, fällt der Aufwand für wiederholtes Erzeugen von Bindungen nicht ins Gewicht. Thies [36] speichert Referenzen auf komplette Bindungen in den Objekten, die Programmelemente repräsentieren. Das ist sehr effizient in Bezug auf die Laufzeit, erfordert aber viel Arbeitsspeicher [2]. Aufgrund der Anforderung, effizient mit Laufzeit *und* Arbeitsspeicher umzugehen, dürfte eine Repräsentation von Programmelementen auf Basis von Bindungen für diese Implementierung weniger geeignet sein.

Zusätzlich zum AST für eine feingranulare Programmanalyse stellen die JDT Programmelemente im Java-Modell zur Verfügung. Die Entwicklungsumgebung benutzt dieses Modell zur Darstellung von Programmen in einer Auflösung vom Package über Member und bis zum lokalen Typ. Die Elemente des Java-Modells erlauben einfache Programmanalysen, das Ermitteln von Typhierarchien und in Verbindung mit der *Search-Engine* auch das Aufsuchen von Referenzen. Seine Elemente sind nach dem Entwurfsmuster *virtuelles Proxy* implementiert. Die entsprechende Aufteilung der Objekte in *handle* und *body* ermöglicht einen besonders sparsamen Umgang mit Speicherplatz [11, S. 311], sodass eine große Anzahl von Objekten des Java-Modells gleichzeitig im Arbeitsspeicher gehalten werden kann.

Die Programmierschnittstelle des Java-Modells bildet viele der hier benötigten Eigenschaften direkt ab, sodass die Implementierung der JDT direkt wiederverwendet werden kann. Aus diesen Gründen wähle ich das Java-Modell als Basis der Implementierung von Programmelementen, dessen Typen ich durch Komposition um Zustand (Attribute und Assoziationen) und Verhalten (Methoden) gegenüber den Java-Modell-Elementen erweitere.

Steimann et al. [35] haben mittlerweile eine abstraktere Repräsentation von Programm-Elementen entwickelt, die auf dem Konzept von *properties* beruht (vgl. Abschnitt 6.4, S. 49). Die vorliegende Arbeit berücksichtigt diese Entwicklung noch nicht.

4.3.2 Repräsentation und Eigenschaften

Auch wenn das gewählte Java-Modell große Teile der benötigten Funktionalität bereits bietet, muss es um einige Features erweitert werden. Die erforderlichen Ergänzungen skizzieren die nächsten Absätze.

Den Java-Elementen `IField` und `IType` fehlt die Möglichkeit, Ort und Zugriffsmodifikator unabhängig von einer Änderung des zugrunde liegenden Quellcodes zu ändern. Die Implementierung führt deshalb ein abstraktes `DeclarationElement` mit konkreten Spezialisierungen für Felder und für Classifier ein, die aus den Java-Elementen und den zusätzlich benötigten Attributen bzw. Assoziationen zusammengesetzt sind. Die Instanzen dieser Deklarationselemente müssen in einer 1:1-Beziehung zu den entsprechenden Objekten des Java-Modells stehen, um die Identität der Deklarationselemente zu wahren. Dafür sorgt eine `ElementFactory`, die die Instanzen in einem *dictionary* mit konstanter Zugriffszeit, einer Hash-Tabelle, vorhält.

Eine Repräsentation von Referenzen fehlt dem Java-Modell ganz. Die vorliegende Arbeit definiert eine abstrakte `ElementReference` mit Spezialisierungen für Feld- und für Classifier-Referenzen. Die Anzahl von Referenzen und damit von Constraints lässt sich deutlich vermindern, wenn je Einheit nur eine Referenz gespeichert wird. Der Transparenz halber werden als Einheiten Member gewählt. Dabei ist allerdings zu berücksichtigen, dass sich Referenzen mit verschiedenen qualifizierenden Ausdrücken unterscheiden und deshalb nicht ohne Weiteres zu einem Objekt zusammengefasst werden können. Referenzen überschreiben die Methoden `hashCode` und `equals` entsprechend, selbstverständlich ohne dabei veränderliche Eigenschaften der Objekte zu benutzen. Referenzen auf ein gegebenes Deklarationselement ermittelt die *Search-Engine* der JDT. Um eine Suche nicht wiederholen zu müssen, speichert die `ElementFactory` die Suchergebnisse in einer Hash-Tabelle.

Zu den Eigenschaften von Referenzen gehört der Empfängertyp (vgl. Abschnitt A.1, S. 57), der nur mithilfe einer Programmanalyse auf Basis des AST ermittelt werden kann. Diese Arbeit kann dafür teilweise Ergebnisse wiederverwenden, die im Rahmen der Vorarbeit [36] entstanden sind. Der Algorithmus ist wesentlich vom API des AST der JDT bestimmt. Er ist in einer Hilfsklasse namens `ReferenceReceiverInspector` gekapselt. Im Fall von *this*- oder *super*-Referenzen ist der Empfängertyp vom Ort der Referenz abhängig. Wird der Ort geändert, muss auch der Empfängertyp neu berechnet werden. Dies muss ohne Nutzung des AST erfolgen, da ein AST für das geänderte Programm noch nicht existiert. Der AST wird auch nicht gebraucht, da der qualifizierende Ausdruck unverändert bleibt und nur sein Typ neu berechnet werden muss.

Bemerkenswerterweise fehlt dem Java-Modell eine Repräsentation einer Felddeklaration, obwohl ein Deklarationsausdruck recht komplex sein kann, besonders wenn er mehrere Felder gemeinsam deklariert. Die vorliegende Implementierung definiert zur Programmanalyse eine Hilfsklasse `FieldDeclarationStatement`, die Felddeklarationen auf Basis des AST analysiert. Die Analyse nutzt das Entwurfsmuster *Visitor*, um den Teil-AST eines Deklarationsausdrucks zu durchlaufen, ganz nach Vorlage aus dem Lehrbuch [11, S. 319 ff.].

Die `ElementFactory` enthält eine weitere Datenstruktur zur Verbesserung der Laufzeiteffizienz: Die Reihenfolge der Programmanalyse ist durch den Tiefendurchlauf des Constraintgraphen bestimmt. Das erfordert ein wiederholtes Zugreifen auf den Syntaxbaum ein und desselben Java-Files. Um den AST nicht jedes Mal neu aus dem Quellcode aufbauen zu müssen, speichert die `ElementFactory` ASTs in einem trivialen *LRU-Cache* zwischen, der jeweils nur einen AST enthält.

4.4 Berechnen der Constraintmenge

Die entscheidenden Datenstrukturen und der Algorithmus (vgl. Abschnitt 3.1, Bild 4, S. 19) sind bereits beschrieben. Damit ist die Implementierung des Algorithmus im Wesentlichen vorgegeben, erschließt sich aber nicht von selbst. Dieser Abschnitt führt durch technische Details der Implementierung, ohne Entscheidungen zu begründen, die so oder auch anders hätten ausfallen können. Er beginnt mit der Repräsentation von Constraints und geht dann auf die Ermittlung der Constraints und darin vorkommender Programmelemente ein und erläutert die Beschränkung der Suche nach Programmelementen auf potenzielle Einflussbereiche.

4.4.1 Repräsentation von Modell-Constraints

Die Implementierung repräsentiert Modell-Constraints als einfache Datenobjekte, die Referenzen auf die beteiligten Programmelemente speichern. Sie sind als Subtypen des Interfaces `JavaModelConstraint` nach ihrer Art klassifiziert, aufgeteilt nach der Anzahl referenzierter Programmelemente. Tabelle 9 (S. 72) im Anhang stellt Formelbezeichnungen der Constraintregeln den Bezeichnern von Constrainttypen der Implementierung gegenüber.

Aus praktischen Gründen implementieren die Modell-Constraints jeweils das Prädikat `isSatisfied`, das angibt, ob das Constraint mit der aktuellen Wertebelegung enthaltener Programmelemente für das zu refaktorisierende Programm erfüllt ist. Eine simple Iteration über eine Liste von Constraints kann damit ermitteln, ob die Constraintmenge erfüllt ist.

Das Refaktorisierungsziel, das Hochziehen des Felds, wird nach Vorbild von Tip et al. [40] mithilfe eines separaten `FixedLocationConstraints` durchgesetzt. Das erlaubt, einfach zu prüfen, ob die Constraints vor der Refaktorisierung erfüllt sind.

4.4.2 Graphendurchlauf

Eine technische Klasse namens `Traveller` im Package `generation`⁵ kapselt die Berechnung der Constraintmenge. Ein `Traveller`-Objekt durchläuft das Programm und zeichnet den Constraintgraphen auf. Es delegiert Spezialaufgaben an hier *Begleiter* (*sputnik*) genannte Objekte innerer Klassen.

Die Klasse `Traveller` hat nur zwei öffentliche Methoden. Die Methode `traverseGraph` berechnet die Constraintmenge und gibt sie in einer Liste gespeichert zurück. Ihre Parameter sind das hochzuziehende Feld und der Classifier, in den das Feld hochgezogen werden soll. Die zurückgegebene Liste enthält das Constraint, das das Refaktorisierungsziel durchsetzt, noch nicht. Dieses Constraint liefert die zweite Methode, genannt `targetFixedLocationConstraint`, separat. Es vervollständigt das Constraintnetzwerk, das die Refaktorisierung komplett spezifiziert. Die Klasse `Traveller` modifiziert das analysierte Java-Projekt nicht.

Die Menge der Classifier, die hochziehbare Felder enthalten (vgl. Abschnitt 4.1, S. 24), ermittelt ein rekursiver Algorithmus, den eine `SupertypeGraphWalker` genannte Klasse realisiert. Die Menge wird benötigt, um zu bestimmen, ob Felddeklarationen verlagerbar sind.

4.4.3 Kontrollfluss

Die Implementierung weicht in Details vom klaren Schema des in Abschnitt 3.1, Bild 4 (S. 19) skizzierten Algorithmus ab. Der eigentliche Einstiegspunkt in den rekursiven Algorithmus ist die Methode `visitField`.

Anstatt Felder direkt als gesehen zu markieren, werden sie einer Datenstruktur mit konstanter Zugriffszeit, einem *Hash-Set*, hinzugefügt (Bild 4, Zeile 6). Der Kern der Implementierung des Algorithmus umfasst die Zeilen 7 bis 9, in denen für alle Constraintregeln alle

⁵Der nach Konvention von einer Internetadresse abgeleitete Präfix `de.erlandm.cgcs` ist jeweils stillschweigend weggelassen.

darin vorkommenden Programmelemente im zu refaktorisierenden Programm aufgesucht werden. Die Implementierung ordnet die Reihenfolge des Durchlaufens der Constraintregeln und des Aufsuchens von Programmelementen in diesen Zeilen des Algorithmus etwas um, um wiederholtes Suchen derselben Programmelemente zu vermeiden.

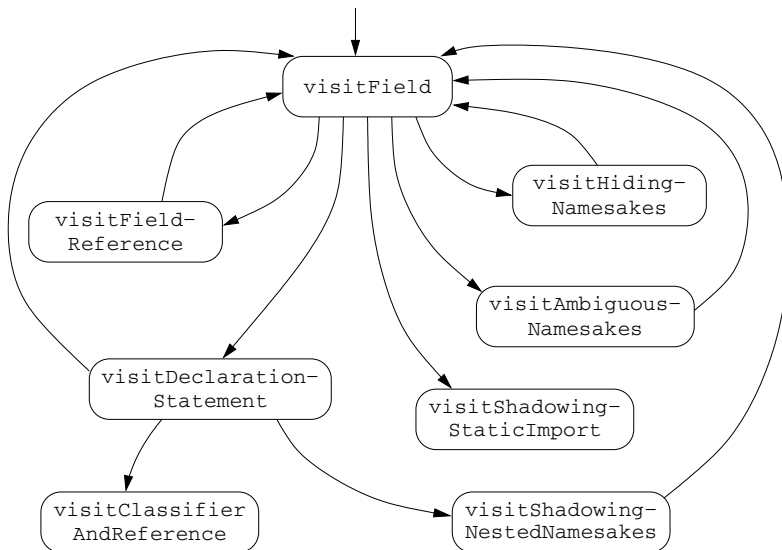


Bild 7 – Skizze des Aufrufgraphen der Implementierung des Algorithmus zur Constraintermittlung

Die folgende Darstellung orientiert sich an der Skizze des Aufrufgraphen in Bild 7. Jede Ecke des Aufrufgraphen steht für eine Methode der Implementierung und gleichzeitig für eine bestimmte Programmkonstellation, die Prämissen von Constraintregeln erfüllen kann. Die Methoden veranlassen das Erzeugen entsprechender Constraints, wenn eine Prämisse der behandelten Constraintregel erfüllt ist.

Für Felder, deren Deklarationen modifizierbar sind, sucht `visitField` das zu refaktorisierende Programm nach *allen* Referenzen ab und ruft für jede dieser Referenzen die Methode `visitFieldReference` auf. Diese veranlasst einen `ConstraintBuilder`, die Constraints nach Regeln *Acc-1*, *Acc-2* und *Inh-1* zu erzeugen. Ist die Referenz Teil einer verlagerbaren Deklaration eines anderen Felds, wird `visitField` mit diesem anderen Feld rekursiv aufgerufen.

Nach Verarbeitung aller Referenzen lässt `visitField` einen `DeclarationStatementSputnik` die Deklarationsanweisung des Felds untersuchen. Felddeklarationen können weitere Felder deklarieren und Referenzen auf Classifier und Felder enthalten. Bei diesen Programmelementen wird der Durchlauf rekursiv fortgesetzt. Die Ermittlung möglicherweise abgeschatteter Felder und Feldreferenzen reicht das `DeclarationStatementSputnik`-Objekt an einen `ShadowingByNestingSputnik` weiter.

4.4.4 Einflussbereich Verbergen

Ein einfaches Beispiel für die Beschränkung der Suche nach Programmelementen auf Einflussbereiche ist das Aufsuchen namensgleicher Felder, das eine innere Klasse, genannt `Hiding-InheritedFieldsSputnik`, implementiert.

Die Prämisse der Regel *Hid-1* verlangt eigentlich, das gesamte Programm nach namensgleichen Feldern abzusuchen. Das ist jedoch gar nicht nötig (vgl. Abschnitt 6.4, S. 49). Das Hochziehen eines Felds oder einer Feldreferenz kann nur Deklarationen verbergen bzw. Referenzen vor Deklarationen verbergen, die in Supertypen des Quell-Classifiers deklariert sind. Deshalb kann die Suche nach namensgleichen Feldern für dieses Constraint auf die Supertypen des Quell-Classifiers beschränkt werden. Die Menge dieser Supertypen ist also der für die Regel *Hid-1* abzusuchende Einflussbereich.

Anders als Schäfer et al. [26] beschränke ich die Suche auf diesen Bereich, weil damit die Zahl der zu erzeugenden Constraints reduziert werden kann. Die Menge der Supertypen und ihre Felddeklarationen liefern Methoden des API des Java-Modells. Jedes gefundene Tripel von Deklaration, Referenz und zweiter namensgleicher Deklarationen, von denen wenigstens ein Element beweglich ist, wird an einen `ConstraintBuilder` weitergereicht.

4.4.5 Einflussbereich *Mehrdeutigkeit*

Verzwickter ist die Suche nach Deklarationen namensgleicher Felder, die zu Mehrdeutigkeit von Referenzen führen können. Sie ist in einer `AmbiguousFieldsSputnik` genannten inneren Klasse implementiert. Nach Prämisse der Regel *Inh-2* müsste, wie bereits für Regel *Hid-1*, das gesamte Programm abgesucht werden, und zwar für jedes in Constraints vorkommende Feld – schon die Erhöhung seines Zugriffsmodifikators kann Mehrdeutigkeit verursachen.

Der Teil des Programms, der auf namensgleiche Felder abgesucht werden muss, ist auch in diesem Fall meist sehr klein. Er ist durch die Konklusion der Constraintregel eingegrenzt: Erben kann ein Empfängertyp nur von seinen Supertypen und eine Deklaration kann Mehrdeutigkeit nur in Subtypen des Orts ihrer Deklaration verursachen, da sie maximal von diesen geerbt wird.

Der Algorithmus zur Ermittlung namensgleicher Deklarationen durchläuft also alle Subtypen des Orts der gegebenen Deklarationen und von diesen jeweils aufwärts alle Supertypen bis zu einer Deklaration gleichen Namens. Dabei werden wie beim `SupertypeGraphWalker` alle Vererbungspfade berücksichtigt.

Das Berechnen der Menge der Subtypen ist verhältnismäßig aufwendig. Deshalb hält eine `AmbiguousFieldsFactory` einmal ermittelte Subtyphierarchien für wiederholte Nutzung vor.

4.4.6 Constraint-Builder

Nach Vorbild des Entwurfsmusters *Builder* ist der Graphendurchlauf von der eigentlichen Constraint-erzeugung (Bild 4 auf S. 19, Zeile 10) getrennt. Das `Traveller`-Objekt spielt die Rolle des *directors*, ein `ConstraintBuilder`-Objekt nimmt die Programmelemente entgegen, generiert die entsprechenden Constraints und speichert sie in einer Ergebnisliste.

Dieses Muster erlaubt prinzipiell, den Builder auszutauschen, um eine andere Repräsentation von Constraints zu erzeugen. Die vorliegende Implementierung wählt jedoch einen anderen Weg und transformiert die gewonnenen Modell-Constraints nachträglich mithilfe eines Adapters in die Integer-Constraints.

4.4.7 Konstruierte Testfälle

Eine größere Zahl von Testfällen sichert die Implementierung des Algorithmus zur Berechnung der Constraintmenge ab. Arbeitspferd dieser Testfälle ist die auf Gamma und Beck [11, S. 371 ff.] zurückgehende *Test-Fixture*, eine Instanz der Klasse `TestProject` aus dem Package `org.eclipse.contribution.junit.test`. Die Testfälle spezifizieren jeweils eine PULL-UP-FIELD-Refaktorisierung auf einem Miniprogramm und sichern zu, dass bestimmte Constraints erzeugt werden.

Wie Abschnitt 3.7 (S. 22) ausführt, ersetzen diese konstruierten Testfälle der Constraint-erzeugung nicht die systematischen Tests der Refaktorisierung auf realen Programmen, die Abschnitt 4.7 (S. 35) behandelt.

4.5 Adapter zur Constraintlösung

Für die berechnete Constraintmenge muss *eine* Lösung gefunden werden, um die Refaktorisierung ausführen zu können. Zur Prüfung der Constraint-erzeugung werden *alle* Lösungen der Constraintmenge benötigt. Abschnitt 3.6 (S. 20) begründet bereits den Einsatz eines etablierten Constraintlösers. In diesem Abschnitt geht es um die Auswahl des Constraintlösers

Choco und um die technische Umsetzung der Transformationen der Modell-Constraints in die Integer-Constraints und zurück (Abschnitt 3.6, S. 21).

4.5.1 Auswahl des Constraintlösers

Choco ist eine quelloffene Java-Bibliothek zur Constraintprogrammierung und Lösung universeller Einschränkungprobleme. Sie zeichnet sich durch eine gut lesbare, deklarativ formulierte Spezifikation von Modellen und eine davon klar getrennte Implementierung und Steuerung der Constraintlösung aus [17].

Steimann [34] setzt den Constraintlöser *Choco* für größere Refaktorisierungsprobleme erfolgreich ein. Der Einfachheit halber entscheide ich mich, dieser Vorlage zu folgen, und wähle die aktuelle Version *Choco-Solver 2.1.1*, ohne weitere Varianten detailliert zu prüfen. Immerhin ist *Chocos* Programmierschnittstelle auf den ersten Blick nicht allzu weit vom gerade entstehenden Standard *JSR-331* [16] entfernt. Ein Austausch gegen einen Constraintlöser, der den Standard implementiert, dürfte deshalb ohne zu großen Aufwand möglich sein.

4.5.2 Kapselung von Lösung und Transformation

Eine `SolverAdapter` genannte Klasse im Package `constraints` implementiert beide Transformationen. Sie fungiert als Fassade für die Klassen des Constraintlösers. Die Methoden `solve` und `solveAll` übergeben die Constraintmenge als Parameter. Der Adapter iteriert über die Menge und verteilt die Constraints per *double dispatch* nach ihrem Typ auf entsprechende Methoden, die jeweils die spezifische Transformation eines Constrainttyps (vgl. Abschnitt 4.4, S. 29 und Tabelle 9, S. 72) vornehmen. Kann der Constraintlöser eine Lösung der transformierten Constraintmenge ermitteln, schreiben die Methoden `solve` und `solveAll` die erste ermittelte Lösung zurück in die Repräsentation der Programmelemente.

Werden für Testzwecke mehrere Lösungen benötigt, kann mit der Methode `nextSolution` die nächste Lösung berechnet und in die Java-Modell-basierten Programmelemente zurückgeschrieben werden.⁶

Die Transformationen nutzen Zwischenobjekte, die Referenzen auf *Chocos* einfache Constraintvariable für jedes Programmelement zusammenfassen. Die Zwischenobjekte sind per Hash-Tabelle den Programmelement-Objekten zugeordnet. Diese Indirektion entkoppelt die Repräsentation der Programmelemente vollständig vom Adapter und den darin gekapselten Modell-Transformationen.

Die Transformation von Modell-Constraints in *Chocos* Integer-Constraints folgt dem in Abschnitt 3.6 (S. 21) beschriebenen allgemeinen Prinzip, ist aber für jeden Typ der Modell-Constraints (vgl. Tabelle 9, S. 72) individuell implementiert.

Manche Constraints können direkt in ein entsprechendes Constraint des Lösers übersetzt werden. Beispielsweise entspricht dem `ReferenceLocationConstraint`, das eine Felddeklaration `d` mit einer darin verwendeten Referenz `r` zusammenhält (vgl. Regel *Suppl-1*, S. 66), direkt einem Gleichheits-Constraint von *Choco*: `eq(d.moved, r.moved)`. Deklaration *und* Referenz werden hochgezogen oder beide nicht.

4.5.3 Fallunterscheidungen

Meist ist die Angelegenheit jedoch komplizierter und erfordert Fallunterscheidungen, die in *Chocos* Repräsentation zu mehr oder weniger tief geschachtelten `ifThenElse`-Constraints führen. Ein vergleichsweise einfaches Beispiel soll das demonstrieren.

Die Transformation eines `AccessibilityConstraints` für eine hochziehbare Classifier-Referenz `r` nach Regel *Acc-1* verlangt zunächst, den erforderlichen Zugriffsmodifikator für beide Fälle zu berechnen, nicht hochgezogen und hochgezogen. Damit kann ein `ifThenElse`-

⁶Die Methode `solveAll` des Constraintlösers erlaubt, anders als vielleicht erwartet, kein Iterieren über alle Lösungen. Möglich ist Iterieren hingegen nach Aufruf von `solve`.

Constraint generiert werden, das in der Bedingung eine Constraintvariable `r.moved` enthält und in den beiden Zweigen die entsprechenden Ungleichungen für die zuvor berechneten erforderlichen Zugriffsmodifikatoren.

In diesem Beispiel sind nur zwei Fälle zu unterscheiden. Sind Referenz und Deklaration beweglich, müssen vier Fälle und bei drei beweglichen Programmelementen bis zu acht Fälle unterschieden werden. Für jeden Fall berechnet der Adapter die Werte der in den Constraints vorkommenden Prädikate und Funktionen. Dazu gehören die erforderliche Zugreifbarkeit α , der Empfängertyp von Referenzen $r.\rho$ und das Prädikat „inherits“ (vgl. Abschnitt 3.6, S. 21).

4.5.4 Verbesserung der Effizienz

Bereits gelöste Constraints, also solche, die für den gesamten Wertebereich ihrer Variablen erfüllt sind, können weggelassen werden (vgl. Abschnitt 3.5, S. 20). Die Fallunterscheidung ermöglicht, bei der Transformation gelöste Constraints zu entdecken. Sie werden dem Constraintmodell des Löser deshalb nicht hinzugefügt

Ein Beispiel dafür ist ein Constraint der Art $d.\alpha \geq_A \text{private}$. Es tritt in den oben genannten Fallunterscheidungen auf – α wird für jeden Fall berechnet und ist daher bei der Transformation bekannt.

Geschachtelte `ifThenElse`-Constraints oder `implies`-Constraints könnten bei der Transformation bereits in elementare Constraints zerlegt werden, die nur Negationen, Konjunktionen und Disjunktionen enthalten. Diese Möglichkeit, die Lösung des Constraintproblems eventuell zu beschleunigen, verfolgt die vorliegende Arbeit nicht. Der Constraintlöser mag die Zerlegung weniger effizient durchführen, kann das aber fehlerfrei. Außerdem bleibt die Formulierung des Constraints ohne Zerlegung leichter nachvollziehbar.

4.5.5 Rücktransformation

Die Rücktransformation einer ermittelten Lösung, also die Rücktransformation von den Werten der Variablen der Integer-Constraints in die Belegung der Programmelement-Eigenschaften der Modell-Constraints ist vergleichsweise einfach. Für alle Programmelemente in der Hash-Tabelle (s. o.), werden die zugeordneten Zwischenobjekte nachgeschlagen und über diese die instanziierten Werte der *Choco*-Constraintvariablen vom Löser abgefragt und in die Eigenschaften der Programmelemente übertragen.

Leicht ist dann auch die Zusicherung der Nachbedingung zu prüfen, dass nach Zurückschreiben der Werte die Constraintmenge erfüllt ist. Man iteriert über die Menge der Modell-Constraints und prüft ihr Prädikat `isSatisfied`.

4.6 Ändern des Quellcodes

Eine Refaktorisierung modifiziert den Quellcode des ursprünglichen Programms. Die eigentliche Code-Transformation besteht bei dieser `PULL-UP-FIELD`-Refaktorisierung aus einer Änderung von Zugriffsmodifikatoren von Feldern und Classifiern, der Qualifizierung von Classifier-Referenzen und der Verlagerung von Felddeklarationsanweisungen von einem Classifier in einen anderen. Ist die Quellcode-Änderung in die Entwicklungsumgebung eingebunden, kann sie umstandslos ausgeführt und getestet werden. Dieser Abschnitt skizziert, wie geänderte Eigenschaften der Programmelemente in die Änderung des Quellcodes transformiert werden und wie die Transformation in das Refaktorisierungs-Rahmenwerk des Eclipse-LTK eingepasst ist. Die Implementierung folgt Widmers Anleitung [42].

4.6.1 Bestandteile der Quellcode-Änderung

Die Änderung von Zugriffsmodifikatoren implementiert eine `AccessModifierModifier` genannte Klasse, die im Package `refactoring` enthalten ist. Ihre einzige öffentliche Methode

`createChange` geht alle Java-Files durch, in denen veränderte Zugriffsmodifikatoren vorkommen, erzeugt den AST und berechnet mithilfe der Eclipse-Klasse `ASTRewrite` die erforderlichen Änderungen des Programmtexts durch Anweisungen zur Modifikation des Syntaxbaums. Der AST bleibt dabei zunächst unverändert. Das zurückgegebene `Change`-Objekt spezifiziert die Änderungen am Quellcode, die das Refaktorisierungs-Rahmenwerk später ausführt.

Für die Verlagerung der Deklarationsanweisungen wird das *Move-Refactoring* der Eclipse-JDT verwendet. Es wird so parametrisiert, dass es Felddeklarationen samt ihrer Deklarationsanweisung verlagert, ohne irgendwelche Zugriffsmodifikatoren anzupassen. Allerdings zerlegt es ungewollt eine gemeinsame Felddeklaration in zwei separate Deklarationen und fügt unerwünscht Import-Anweisungen für verlagerte Classifier-Referenzen ein. Die Zerlegung stört nicht, aber die Folgen der unerwünschten Import-Anweisungen müssen von Fall zu Fall manuell inspiziert werden (vgl. Abschnitt 5.3, S. 39).

Die Klasse `FieldsMover` adaptiert das *Move-Refactoring*, sodass es analog zur Änderung der Zugriffsmodifikatoren mit einer Methode `createChange` aufgerufen werden kann. Die Änderung des Programmtexts wird durch Aufruf der Eclipse-Refaktorisierung für jede zu verlagernde Felddeklaration berechnet. Die Wiedernutzung der Eclipse-Implementierung hat den Nachteil, dass die Code-Transformation nicht so genau gesteuert werden kann, wie es für die Zwecke der vorliegenden Arbeit wünschenswert ist. Offensichtlicher Vorteil ist, dass das komplexere Kopieren von Teil-AST von einem AST in einen anderen nicht neu implementiert werden muss.

4.6.2 Integration in die Entwicklungsumgebung

Zur Einbindung der ganzen Refaktorisierung in die Eclipse-Entwicklungsumgebung erweitert die Klasse `ConstraintBasedPullUpFieldRefactoring` die abstrakte Klasse `Refactoring` des Refaktorisierungs-Rahmenwerks. Die obligatorisch zu implementierende Methode `checkFinalConditions` ermittelt die Constraintmenge und berechnet eine Lösung wie in Abschnitten 4.4 und 4.5 beschrieben.

Die Quellcode-Änderung setzt die ebenfalls obligatorisch zu implementierende Methode `createChange` aus den oben skizzierten Bestandteilen zusammen. Da sich die beiden Textänderungen im Allgemeinen überschneiden, ist ihre Kombination nicht direkt möglich. Die Implementierung der Methode `createChange` bedient sich des unschönen Tricks, die Änderung der Zugriffsmodifikatoren temporär anzuwenden, dann die Verlagerung zu berechnen und anschließend die temporäre Änderung wieder zurückzunehmen. Damit wird ein Konflikt sich überschneidender Textänderungen vermieden.

Das so implementierte Refaktorisierungswerkzeug kann zur Demonstration aus dem Kontextmenü eines Felds im *Outline-View* oder im *Package-Explorer* durch Selektion des ergänzten Menüeintrags „Pull up Field Refactoring“ aufgerufen werden. Bild 10 (S. 74) zeigt ein Beispiel des Konfigurationsdialogs. Er bietet Supertypen, in die das Feld prinzipiell hochgezogen werden kann, in einer *Drop-down-Liste* zur Auswahl an. Wie gewohnt kann über entsprechende Schaltflächen eine Voransicht der berechneten Änderungen gewählt oder die Refaktorisierung direkt ausgeführt werden. Ist die Änderung ausgeführt, kann sie per *Undo* zurückgenommen werden. Die optionale Bedienung des *History-Service* [42] ist nicht implementiert.

Die Implementierung der Code-Transformation dient der Prüfung des Algorithmus zur Berechnung der Constraintmenge. Anforderungen an ein kommerzielles Refaktorisierungswerkzeug erfüllt sie allerdings nicht. Sie ist wegen der Zusammensetzung überschneidender Textänderungen wenig effizient, die kanonische Qualifizierung hochzuziehender Classifier-Referenzen fehlt bisher ganz und die Verlagerung von Felddeklarationen hat folgende, unbeabsichtigte Nebenwirkungen: Die benutzte Eclipse-Refaktorisierung respektiert in bestimmten Situationen die Reihenfolge von Felddeklarationen nicht, fügt (in der Eclipse-Version 3.5) fehlerhafte statische Import-Anweisungen ein und zerlegt gemeinsame Felddeklarationen in individuelle

Deklarationen. Fehler treten infolge dieser Eigenheiten in den im folgenden Abschnitt dargestellten automatisch ausgeführten Tests glücklicherweise so selten auf, dass sie manuell korrigiert werden können (vgl. Abschnitt 5.3, S. 39).

4.7 Testen der Refaktorisierung

Die implementierte Refaktorisierung soll geprüft werden, indem sie automatisch auf alle möglichen Stellen realer Programme angewendet wird (vgl. Abschnitt 3.7, S. 22). Dieser Abschnitt beschreibt, wie alle diese Stellen in Programmen ermittelt werden, wie die Refaktorisierung automatisch ausgeführt und das Ergebnis auf fehlerfreie Übersetzbarkeit geprüft wird. Außerdem wird besprochen, wie der *Refactoring-Tool-Tester* (RTT) [8] dazu gebracht wird, die Testfälle auf refaktorierten Programmen, den Probanden, auszuführen.

4.7.1 Ermitteln hochziehender Felder

Alle Stellen, an denen eine PULL-UP-FIELD-Refaktorisierung prinzipiell möglich ist, sind leicht gefunden: In einem Programm, das hier als Eclipse-Java-Projekt gegeben ist, werden alle Classifier aufgesucht. Sofern sie Felder deklarieren und Supertypen haben, können alle Felder in alle Supertypen hochgezogen werden, die als modifizierbarer Quellcode vorliegen. Die Menge aller Classifier schließt dabei beliebig geschachtelte und lokale Classifier ein, auch solche, die in Initialisierungsblöcken vorkommen. Eine Hash-Tabelle speichert alle potenziellen Quell-Classifier zusammen mit einer Liste ihrer Supertypen für eine weitere Verwendung.

4.7.2 Statistik und Kompilierbarkeit

Das erste hier implementierte Testwerkzeug zählt Refaktorisierungsmöglichkeiten gegebener Programme, misst Laufzeiten für Constraintzeugung und Constraintlösung und überprüft die Ergebnisse der Refaktorisierung auf fehlerfreie Kompilierbarkeit. Es ist aus dem Kontextmenü eines Java-Projekts über den Menüeintrag „Test Run Pull up Field“ aufrufbar. Dafür bindet es sich per Deklaration in der `plugin.xml` an den *Command-Extension-Point* der Eclipse-Entwicklungsumgebung und leitet entsprechend eine konkrete Subklasse von der Klasse `AbstractHandler` der Eclipse-Plattform ab.

Das Package `testing` bündelt die Implementierung dieses Testwerkzeugs. Eine seiner Varianten generiert für alle potenziellen Refaktorisierungsoperationen die Constraints, prüft, ob die Constraints vor der Refaktorisierung erfüllt sind, berechnet eine Lösung der Constraintmenge, führt die Refaktorisierung aus und speichert die benötigten Rechenzeiten.

Eine andere Variante berechnet jeweils bis zu 100 Lösungen je Refaktorisierungsoperation, transformiert die Lösungen in veränderte Eigenschaften der Programmelemente zurück, sichert zu, dass die Modell-Constraints ebenfalls erfüllt sind, führt die Änderung des Quellcodes aus und prüft, ob das Programm damit fehlerfrei kompiliert.

Nach jedem Test wird die Änderung des Quellcodes per *Undo*-Funktion der Refaktorisierung wieder rückgängig gemacht. Die Ergebnisse werden auf einer Konsole der Entwicklungsumgebung protokolliert.

4.7.3 Testen mit dem RTT

Das zweite Testwerkzeug ist ein Adapter an den von El Hosami und Ikkert [8] entwickelten *Refactoring-Tool-Tester*. Der RTT wendet über diesen Adapter die implementierte Refaktorisierung auf Probanden an und prüft deren Ergebnis auf Korrektheit. Folgende Absätze erläutern, wie das geht.

Der RTT ist ein Eclipse-Plugin, das wie ein Rahmenwerk funktioniert. Der RTT steuert den Ablauf des Testens. Er ruft eine Methode einer zu implementierenden Adapterklasse auf. Diese Methode gibt die Spezifikation einer konkreten Refaktorisierungsoperation zurück. Der

RTT führt diese Refaktorisierungsoperation aus und prüft das Ergebnis der Refaktorisierung. Dazu führt der RTT die Testfälle des refaktorierten Probanden aus und protokolliert, ob die Testfälle des Probanden ohne Fehler durchlaufen (vgl. Abschnitt 3.7, S. 22).

Dieser Entwurf des RTT erreicht Unabhängigkeit zwischen RTT und Refaktorisierung. Beide müssen für das Testen der Refaktorisierung nicht mehr angefasst werden [33]. Am RTT ist lediglich zu konfigurieren, welche Testfälle eines Probanden ausgeführt werden sollen und welcher Adapter benutzt werden soll. Für die Spezifikation der Refaktorisierungsoperationen sorgt der Adapter.

Eine Adapterklasse muss eine öffentliche, mit `@Test` annotierte Methode implementieren, die ein `TestProcedureStep`-Objekt zurückgibt [8, S. 29 f.]. Das zurückgegebene Objekt besteht aus der Spezifikation einer Refaktorisierung in Form eines `Refactoring`-Objekts der LTK und einem Wert, der den weiteren Ablauf steuert [15, S. 53].

Die mit `@Test` annotierte Methode der Adapterklasse spezifiziert die auszuführenden Refaktorisierungsoperationen. Dazu muss sie auf zu refaktorisierende Probanden zugreifen. Das gelingt über ein `TestClientSession`-Objekt, das der RTT jedem Objekt der Adapterklasse in ein mit `@TestSession` annotiertes Feld injiziert [8, S. 30]. Eine Methode `scope` der Klasse `TestClientSession` liefert ein Array von Eclipse-Projekten, die Probanden enthalten. Der RTT erzeugt für jeden Aufruf der mit `@Test` annotierten Methode eine neue Instanz der Adapterklasse. Ein Zustand, der einen Aufruf der Methode überdauern soll, muss in statischen Feldern gespeichert werden.

Der Adapter ist in einem separaten Plugin namens `de.erlandm.cgcs.rtt` implementiert, damit die implementierte Refaktorisierung unabhängig vom RTT bleibt. Die `TestCaseRunner` genannte Adapterklasse aus dem Package `testing` iteriert über bis zu 30 Lösungen für alle möglichen PULL-UP-FIELD-Operationen auf dem gegebenen Probanden. Die Adapterklasse hält eine statische Referenz auf ihren Iterator, den sie beim ersten Aufruf der mit `@Test` annotierten Methode erzeugt.

Im Rahmen der vorliegenden Arbeit ist es nicht gelungen, mit dem RTT einige Tausend Refaktorisierungsergebnisse in einem Testlauf auf fehlerfreie Kompilierbarkeit zu prüfen. Die Prüfung scheitert nach wenigen hundert Refaktorisierungen an Speicherplatzgrenzen – ein LRU-Cache der Eclipse-JDT wächst mit jedem geprüften Refaktorisierungsergebnis stark an. Die Ursache des Speicherüberlaufs bei der Prüfung auf Kompilierbarkeit war im Rahmen der Arbeit nicht zu klären, zu beheben oder zu umgehen. Diese Arbeit verwendet den RTT deshalb nur für das Ausführen der Testfälle der Probanden.

4.7.4 Probanden

Geeignete Probanden müssen als Quellcode vorliegen und über eine gute Testabdeckung verfügen [33]. Mithilfe der Evaluation von Ikkert [15, S. 66 ff.] und in Anlehnung an Steimann und Thies [37] wähle ich die in Tabelle 2 (S. 37) angegebenen Probanden für den Test der implementierten PULL-UP-FIELD-Refaktorisierung. Die Anzahl der Codezeilen der Projekte in Tabelle 2 ist mit dem Programm *Sloccount*⁷ gezählt und auf ganze 100 Zeilen gerundet. Die Testabdeckung ist mit dem Eclipse-Plugin *EclEmma*⁸ gemessen. Bei den Probanden handelt es sich um kleinere bis mittlere Projekte mit einem Umfang von 7 000 bis 150 000 Codezeilen. Die Testabdeckung reicht von 30 bis 97 %. Die letzte Spalte der Tabelle 2 gibt die exakte Anzahl von Testfällen an, die mit dem RTT auf den nicht refaktorierten Probanden ohne Fehlschlag ausführbar sind. Das Bestehen dieser Testfälle ist der Maßstab des Testfall-Orakels. Das Projekt *Draw2d* ist ein Eclipse-Plugin. Der RTT kann die Testfälle dieses Projekts deshalb nicht ausführen. Es wird hier nur für das Compiler-Orakel verwendet.

⁷<http://www.dwheeler.com/sloccount/>

⁸<http://www.eclEmma.org/>

Tabelle 2 – Probanden für den Test der PULL-UP-FIELD-Refaktorisierung

Projekt und Version	Codezeilen insgesamt	davon in Tests	Testabde- ckung / %	Anzahl Testfälle
Apache Commons Math 2.1	36 700	20 200	93	1172
Apache Commons Codec 1.4	7 100	4 600	97	191
Apache Ivy 2.0	62 400	5 800	30	265
dom4j 1.6.1	23 800	6 000	46	890
Draw2d 3.7.0	27 400	3 200	32	–
HTML Parser 1.6	35 000	13 200	62	675
Jaxen 1.1.1	20 700	8 300	73	675
JFreeChart 1.2.0	149 200	49 200	53	2194
JHotDraw 6.0	28 400	7 300	32	1216
JUnit 4.8	14 300	8 200	85	477
XOM 1.2.5	47 700	22 100	73	1411

5 Ergebnisse

Die im vorhergehenden Kapitel beschriebene Implementierung der constraintbasierten PULL-UP-FIELD-Refaktorisierung wird systematisch auf den in Tabelle 2 angegebenen Probanden ausgeführt. Diese Testläufe ergeben, dass die Constraintmengen in allen Fällen vor einer Refaktorisierung erfüllt sind. Sie demonstrieren, wie viele Constraints für eine Refaktorisierung erzeugt werden, welche Laufzeit und wie viel Speicherplatz nötig ist und wie die Constraint-erzeugung und -lösung mit der Größe zu refaktorisierender Programme wächst. In wenigen, erklärbaren Fällen ist die ausgeführte Refaktorisierung nicht korrekt. Dieses Kapitel präsentiert die Ergebnisse der Testläufe und gibt den Aufwand für die Implementierung des Refaktorisierungswerkzeugs an.

5.1 Übersicht

Tabelle 3 (S. 38) gibt eine Übersicht der durchgeführten Testläufe und ihres Umfangs. Sie stellt für jeden Probanden die Anzahl prinzipiell möglicher versuchter PULL-UP-FIELD-Operationen der Anzahl von Fällen gegenüber, in denen diese Operationen tatsächlich ausgeführt werden kann. Prinzipiell möglich ist die Operation an jedem Feld eines Subtypen für jeden seiner Subtypen. Ausführbar sind Refaktorisierungsoperationen, wenn eine Lösung der ermittelten Constraintmenge existiert. Außerdem präsentiert Tabelle 3 jeweils das arithmetische Mittel der Anzahl von Constraints und das Mittel der Laufzeit⁹ für Erzeugung, Transformation und Lösung der Constraintmenge über alle Operationen auf einem Probanden. Die Verteilung der gemittelten Werte ist allerdings schief (s. u.); das arithmetische Mittel hat deshalb eingeschränkte Aussagekraft.

Auf den 11 Probanden sind insgesamt etwa 20 000 verschiedene PULL-UP-FIELD-Refaktorisierungen prinzipiell möglich. Davon sind jeweils 36 bis 76 %, im gewichteten Mittel 41 % oder gut 8 300 Refaktorisierungen ausführbar, weil die Constraintmenge eine Lösung hat. Durchschnittlich werden für eine Refaktorisierung 16 Constraints in 220 ms erzeugt und gelöst bzw. festgestellt, dass die Constraintmenge nicht erfüllbar ist. Der Rest des Kapitels stellt diese Ergebnisse detaillierter dar.

⁹Diese und alle folgenden Laufzeiten sind auf einem PC mit einer *Intel-Core2-Duo-CPU @ 3.00 GHz* und 2 GByte Hauptspeicher ermittelt. Darauf läuft ein *Eclipse-SDK* der Version 3.6.1 (*Helios*) in einer *Sun-Java-SE*-Laufzeitumgebung der Version 1.6.0_22 auf einem GNU/Linux-System.

Tabelle 3 – Übersicht getesteter PULL-UP-FIELD-Operationen

Proband	PULL-UP-FIELD-Operationen			Anzahl Constraints im Mittel	Laufzeit im Mittel / ms
	Anzahl versucht	Anzahl ausführbar	Anteil / % ausführbar		
Apache Math	1370	717	52	9.6	102
Apache Codec	216	147	68	14.3	112
Apache Ivy	2830	1017	36	20.7	134
dom4j	528	252	48	17.1	202
Draw2d	1321	586	44	22.5	116
HTML Parser	750	573	76	10.6	177
Jaxen	364	242	66	6.4	74
JFreeChart	10667	3847	36	17.0	303
JHotDraw	1624	685	42	9.1	105
JUnit 4.8	293	106	36	12.7	117
XOM	265	173	65	25.8	202
Summe/gew. Mittel	20228	8345	41	16.3	220

5.2 Constraint- und Lösungsmengen

5.2.1 Anzahl von Constraints

Die Verteilungen der Anzahl ermittelter Constraints und der Anzahl berechneter Lösungen listet Tabelle 4 auf. Die Verteilung der Mächtigkeit der ermittelten Constraintmengen lässt sich durch Extremwerte und Quantile darstellen. Tabelle 4 ergänzt das arithmetische Mittel aus der Übersichtstabelle 3 um Minimum, Maximum, die 20- und 80-%-Quantile ($q_{0.2}$, $q_{0.8}$) und den Median (Zentralwert, $q_{0.5}$). Der Median liegt generell unterhalb des Mittels; die links beschränkten Verteilungen sind jenseits des 80-%-Quantils deutlich rechtsschief. Der Median der Anzahl von Constraints schwankt von Projekt zu Projekt von 2 bis zu 16 Constraints mit einem Median der Mediane von 10 Constraints. Die Maxima der Constraintzahlen liegen meist unter 250, erreichen beim Projekt *Draw2d* bei einer Operation allerdings ein Maximum von 839 Constraints.

Tabelle 4 – Verteilung der Mächtigkeit von Constraintmengen und Anzahl ihrer Lösungen

Proband	Anzahl Constraints					Häufigkeit der Anzahl von Lösungen				
	min	$q_{0.2}$	$q_{0.5}$	$q_{0.8}$	max	1–3	4–10	11–30	31–100	> 100
Apache Math	1	3	7	15	92	536	52	52	2	75
Apache Codec	2	4	8	21	42	103	10	2	0	32
Apache Ivy	1	6	10	15	233	900	31	17	3	66
dom4j	2	8	13	18	257	247	5	0	0	0
Draw2d	1	7	12	20	839	532	21	16	4	13
HTML Parser	3	5	6	12	96	540	2	4	4	23
Jaxen	1	1	2	11	35	241	1	0	0	0
JFreeChart	1	8	16	23	242	3282	377	152	18	18
JHotDraw	1	4	9	12	55	599	37	41	8	0
JUnit 4.8	2	7	11	15	80	106	0	0	0	0
XOM	1	5	10	25	270	158	2	0	0	13

5.2.2 Anzahl von Lösungen

Die Verteilung der Anzahl von Lösungen der Constraintmengen ist so asymmetrisch, dass sie durch Quantile und Extremwerte nicht sinnvoll charakterisiert wird. Tabelle 4 (S. 38) gibt deshalb absolute Häufigkeiten in logarithmisch eingeteilten Häufigkeitsklassen an. Wenn die Constraintmenge Lösungen hat, ist die Anzahl der Lösungen meist klein, am häufigsten sind es 1 bis 3 Lösungen. In einigen Fällen gibt es eine größere Anzahl von Lösungen, selten sind es mehr als 100. In diesen seltenen Fällen sind es aber meist sehr große Zahlen, jenseits von 100 000 Lösungen. Die exakte Zahl wird in diesen Fällen nicht mehr ermittelt, weil die Rechenzeit zur Suche weiterer Lösungen beschränkt ist. Tabelle 11 im Anhang (S. 73) zeigt die Häufigkeiten der Anzahl von Lösungen über einen größeren Wertebereich.

5.3 Testergebnisse

5.3.1 Erfüllen der Constraints vor der Refaktorisierung

Die der Tabelle 4 (S. 38) zugrunde liegenden Constraintmengen sind in allen Fällen vor der Ausführung der Refaktorisierung, also auf den unveränderten Programmen, erfüllt. Die Constraintmengen und ihre Erfüllung kann die Programmanalyse dabei auf allen Probanden und für alle möglichen PULL-UP-FIELD-Operationen berechnen, ohne *exceptions* auszulösen. Bei unvorhergesehenen Konstellationen zu refaktorisierender Programme könnte die Programmanalyse beispielsweise eine `JavaModelException` oder eine `CoreException` werfen, weil die Berechnung des Empfängertypen scheitert oder eine Suche unerwartete Ergebnisse liefert. Programmierfehler in der Programmanalyse würden durch Fehlschlagen einer Zusicherung von Vor- oder Nachbedingungen auffallen.

5.3.2 Kompilierbarkeit

Das automatische Anwenden des implementierten Refaktorisierungswerkzeugs und anschließendes Kompilieren des geänderten Quellcodes hat folgendes Ergebnis: Das Compiler-Orakel deckt Fehler der Implementierung auf. Folgende Absätze erläutern die Fehler.

An 6 Stellen ist das Ergebnis der Refaktorisierung nicht fehlerfrei übersetzbar. Diese Fehler treten unter den Probanden nur am Projekt *Draw2d* auf. Die Ursache ist in allen 6 Fällen eine fehlerhafte Änderung des Quellcodes. Die vorgegebene Reihenfolge der Felddeklarationen wird in 5 dieser 6 Fälle nicht eingehalten. Gibt es Felddeklarationen am Anfang und am Ende einer Klasse, fügt die verwendete MOVE-MEMBER-Refaktorisierung ein neues Feld nicht hinter dem letzten bereits vorhandenen Feld, sondern weiter vorne in der Klasse ein. Ein Beispiel:

```

1   class A {                               class A {
2       int x= 0;                             int x= 0;
3                                           int j= i; // error
4       void m() { }                          void m() { }
5       int i= 1;                             int i= 1;
6   }                                           }
7   class B extends A {                       class B extends A {
8       int j= i;
9   }                                           }
```

Das Feld `j` der Klasse `B` (links) soll in die Klasse `A` hochgezogen werden. Obwohl die Refaktorisierung spezifiziert, das Feld `j` hinter das Feld `i` in Klasse `A` einzuordnen, fügt MOVE-MEMBER das hochgezogene Feld `j` vor dem Feld `i` ein, anstatt es hinter dem Feld `i` einzufügen. Im Beispiel rechts ist die fehlerhaft ausgeführte Verlagerung gezeigt.

Im sechsten Fall erzeugt die Quellcode-Änderung, genauer die verwendete MOVE-MEMBER-Refaktorisierung, eine fehlerhafte Import-Anweisung. Ein ähnlicher Fehler tritt im Projekt *Apache Math* auf. Hier wird der Fehler jedoch bei der Berechnung der Code-Änderung

erkannt und die Refaktorisierung nicht ausgeführt. Dieser Fehler kann auch an weiteren Probanden, die nicht öffentlich zugänglich sind, provoziert werden. Die Quellcode-Änderung erzeugt in solchen Fällen falsche Import-Anweisungen, die zu Übersetzungsfehlern, aber auch zur Änderung der Bedeutung des Programms führen können. Ein Beispiel:

```

10 package p;                               package p;
11 import r.X;                               import r.X;
12                                           import q.X; // error
13 public class A {                          public class A {
14                                           X qx; // error; should
15 }                                           } // be q.X qx;
16
17 package q;                               package q;
18 public class X { }                       public class X { }
19 class B extends p.A {                   class B extends p.A {
20     X qx;                                 }
21 }
22
23 package r;                               package r;
24 public class X { }                       public class X { }

```

Wird das Feld `qx` aus `B` (links) in die Klasse `A` (rechts) hochgezogen, versucht die Quellcode-Änderung fälschlich die Import-Anweisung `import q.X;` in den Java-File der Klasse `A` einzufügen. Die Ursache dieses Fehlers ist die unvollständige Implementierung der Quellcode-Änderung, die eigentlich verlagerte Classifier-Referenzen mit dem Package-Namen qualifizieren sollte.

Die bisher genannten Fehler sind auf Mängel der Implementierung der Quellcode-Änderung zurückzuführen. Das heißt, die Quellcode-Änderung schreibt die richtig berechnete Lösung der Constraintmenge nicht korrekt in den Quellcode zurück. Übersetzungsfehler, die auf eine unvollständige oder fehlerhafte Constraintmenge oder deren Lösung zurückzuführen sind, konnten an Probanden aus „freier Wildbahn“ nicht entdeckt werden. Solche Fehler können jedoch an speziell konstruierten Programmen provoziert werden. Diese konstruierten Beispiele verifizieren die Grenzen der Refaktorisierung, die in Abschnitt A.9 (S. 68) beschrieben sind. Sie sind in Form fehlschlagender Testfälle dokumentiert (siehe `UnsolvedProblemsTest` im Package `generation`).

5.3.3 Bestehen der Testfälle

Der RTT wendet für jede mögliche Refaktorisierungsoperation, deren Constraintmenge überhaupt Lösungen hat, alle berechneten Lösungen bis maximal 30 Lösungen an. Am Projekt *JFreeChart* konnten wegen des hohen Rechenzeitaufwands allerdings nur bis zu 3 Lösungen je Refaktorisierungsoperation geprüft werden. Die Lösungen unterscheiden sich, aber Maßnahmen zur Erzeugung möglichst unterschiedlicher Lösungen wurden nicht ergriffen. Nach der Quellcode-Änderung führt der RTT die Testfälle des Probanden aus und protokolliert deren Ergebnisse (vgl. Abschnitt 4.7, S. 35).

Die vom RTT ausgeführten Testfälle decken weitere Grenzen der implementierten Refaktorisierung auf: Beim Projekt *JFreeChart* laufen an 4 Stellen nach dem Hochziehen eines Feldes nicht alle Testfälle fehlerfrei durch. Es handelt sich jeweils um serialisierbare Felder, die per Introspektion spezifisch serialisiert werden. Wie Abschnitt A.9 (S. 68) anführt, kann in solchen Fällen eine korrekte Refaktorisierung auch nicht garantiert werden.

Sporadisch schlagen Testfälle der Probanden fehl, die auf das Internet zugreifen. Diese Fehlschläge haben jedoch nichts mit der Refaktorisierung zu tun und treten genauso an nicht-refaktorierten Probanden auf. Die Fälle erfordern jedoch manuelles Zuordnen der Ursache des Fehlschlags eines Testfalls.

Weitere Fehler oder Grenzen der Implementierung konnten an den in Tabelle 2 (S. 37) aufgeführten Probanden mit dem RTT nicht nachgewiesen werden.

5.4 Rechen- und Implementierungsaufwand

5.4.1 Rechenzeiten

Die Verteilungen der Rechenzeiten¹⁰ für Constraintterzeugung und ihre Lösung stellt Tabelle 5 dar. Die Mediane mit Werten zwischen 77 und 374 ms liegen jeweils unterhalb der Mittelwerte (vgl. Tabelle 3, S. 38); die Verteilungen sind rechtsschief. Das 80%-Quantil hat einen Maximalwert über alle Probanden von 374 ms. Nur in wenigen Ausnahmefällen übersteigt

Tabelle 5 – Verteilung der Rechenzeiten für Constraintterzeugung und -lösung

Proband	Rechenzeit gesamt / ms					Anteil / % Erzeugung
	min	$q_{0.2}$	$q_{0.5}$	$q_{0.8}$	max	
Apache Math	24	38	64	131	791	98
Apache Codec	27	34	45	95	467	97
Apache Ivy	26	57	96	160	1537	90
dom4j	26	80	138	374	905	96
Draw2d	26	50	76	128	3515	94
HTML Parser	26	67	82	144	920	96
Jaxen	28	59	67	77	299	98
JFreeChart	26	101	175	339	13725	96
JHotDraw	26	55	77	129	756	93
JUnit 4.8	27	60	109	145	660	88
XOM	27	90	148	203	1532	98

die Rechenzeit eine Sekunde; in einem Extremfall werden allerdings 13 s benötigt. Der Anteil der Rechenzeit für die Erzeugung der Constraints liegt über 90%. Die Rechenzeit für Transformation und Lösung der Constraints fällt kaum ins Gewicht.

Die Werte in Tabelle 5 ergeben sich bei der Ermittlung von Constraints auf Projekten, die nicht jedes Mal frisch kompiliert werden. Werden die Projekte direkt vor der Programm-analyse verändert und neu kompiliert, erhöht sich die Rechenzeit im gewichteten Mittel um 60 ms. Warum das so ist, bleibt unklar.

Die Rechenzeit für die eigentliche Änderung des Quellcodes und das Neukompilieren der Programme können nicht sinnvoll separat ermittelt werden, weil die vorläufige Implementierung der Quellcode-Änderung bereits das Kompilieren eines Zwischenstands enthält. Die mittleren Zeiten für die Quellcode-Änderung einschließlich Neukompilieren schwanken zwischen 300 und 1000 ms mit einem gewichteten Mittel von 580 ms. Die Rechenzeit für Quellcode-Änderung und Übersetzung ist damit im Mittel knapp dreimal so groß wie die Rechenzeit für Constraintterzeugung und -lösung.

5.4.2 Speicherplatz

Der Speicherplatzbedarf wird grob abgeschätzt, indem auf repräsentativen Probanden für alle möglichen Stellen, an denen ein Feld hochgezogen werden kann, die Constraintmenge ermittelt und eine Lösung berechnet wird. Wenn eine Lösung existiert, wird der Quellcode entsprechend geändert und kompiliert. Dabei wird der Heap-Speicher, der der Entwicklungsumgebung zur Verfügung steht, beschränkt und beobachtet, ob die Refaktorisierungen ohne Speicherüberlauf ausgeführt werden können.

¹⁰Rechen- und Laufzeit entsprechen einander, da keine anderen Prozesse das System belasten.

Für die kleineren Probanden *Apache Codec* und *XOM* reicht ein Heap-Speicher in einer Größe von 124 MByte aus. Der mittelgroße Proband *JFreeChart* kommt mit 192 MByte Heap zurecht. Zum Vergleich: Der Standardwert der maximalen Heap-Größe einer Eclipse-Entwicklungsumgebung beträgt 256 MByte.

5.4.3 Komplexität

Bild 8 zeigt die Mediane der Rechenzeiten für Constraint-erzeugung und -lösung mit ihren 20- und 80%-Quantilen, aufgetragen gegen die Größe der refaktoriisierten Programme, gemessen in Anzahl der Codezeilen.

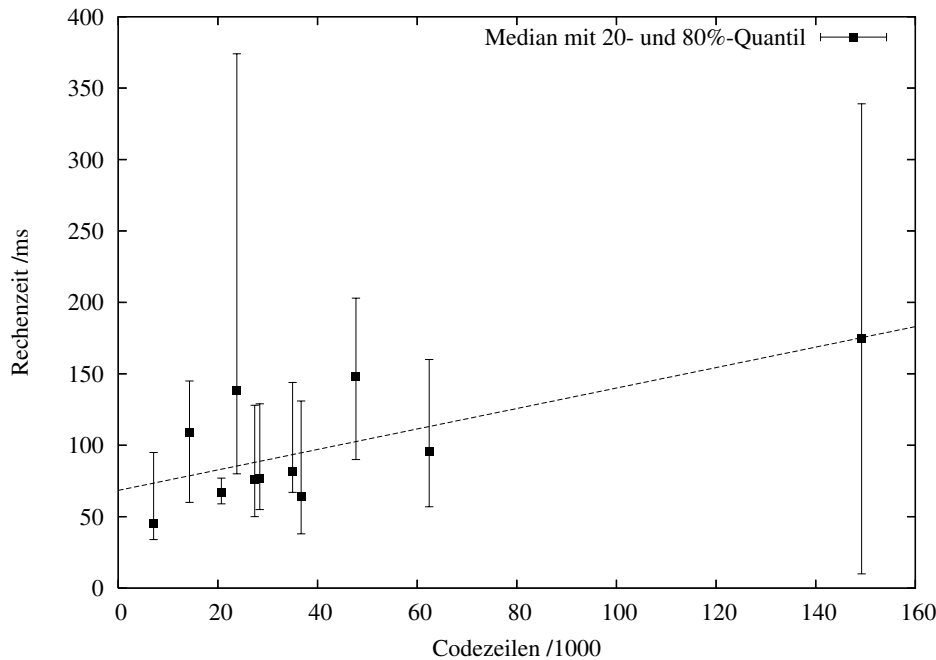


Bild 8 – Rechenzeit für Erzeugung und Lösung der Constraints gegen Größe des Programms

Die Mediane schwanken stark um die eingezeichnete Trendlinie, die eine Steigung von 0.7 ms je 1000 Zeilen Code aufweist. Die Streuung der nicht eingezeichneten arithmetischen Mittel der Rechenzeiten ist größer als die Streuung der Mediane. Eine Normierung der Werte mit der Anzahl von Constraints verringert die Streuung nicht.

Ein Zusammenhang zwischen Größe des Probanden und dem Median der Anzahl von Constraints je Refaktorisierungsoperation ist nicht zu erkennen.

5.4.4 Implementierungsaufwand

Der Aufwandsschätzung für die Implementierung dienen die Anzahlen von Codezeilen, die in Tabelle 10 im Anhang (S. 73) aufgeführt sind. Für die Implementierung der Constraint-erzeugung einschließlich der Programmanalyse sind rund 3300 Zeilen Code nötig, für die Transformation in eine vom Constraintlöser verarbeitbare Form ungefähr weitere 700 Zeilen. Die eigentliche Quellcode-Änderung beansprucht einschließlich Anwendungsschnittstelle zur Demonstration des Refaktorisierungswerkzeugs zusätzliche 700 Zeilen. In diesen Zahlen ist der ungleich höhere Implementierungsaufwand der genutzten Bibliotheken der Eclipse-JDT und des Constraintlösers *Choco* nicht enthalten.

6 Diskussion

Das vorhergehende Kapitel präsentiert die Ergebnisse der Anwendung des implementierten Algorithmus zur selektiven Constraintterzeugung. Hier geht es nun darum, diese Ergebnisse zu bewerten, sie mit denen anderer Arbeiten zu vergleichen und an den Zielen dieser Arbeit zu messen. Die letzten Abschnitte dieses Kapitels benennen Probleme und offene Fragen und fassen die Schlussfolgerungen zusammen.

6.1 Constraint- und Lösungsmengen

Dieser Abschnitt diskutiert anhand der Ergebnisse der Testläufe (Abschnitt 5.2, S. 38) die Fragen, wie viele Constraints der Algorithmus berechnet und ob es etwa zu viele oder zu wenige sind. Er vergleicht die Anzahl von Constraints mit den Ergebnissen anderer Arbeiten und geht anschließend der Frage nach, ob die Anzahl berechneter Lösungen realistisch ist.

6.1.1 Anzahl von Constraints

Die Ergebnisse der Testläufe bestätigen, dass die Anzahl von Constraints, die das Hochziehen eines Felds beschränken, meist vergleichsweise klein ist. Oder präziser ausgedrückt: In 80 % der auf den Probanden versuchten PULL-UP-FIELD-Operationen ist die Anzahl von Constraints kleiner als 25. In sehr seltenen Fällen wird eine Zahl von 250 Constraints überschritten. „Sehr selten“ heißt dabei in weniger als 0.5 %₀ der versuchten Refaktorisierungen (vgl. Abschnitt 5.2, S. 38).

Zum Vergleich: Die Vorarbeit [36] ermittelt für das Programm *JHotDraw* erschöpfend über 25 000 Constraints. Dagegen berechnet die selektive Constraintterzeugung für dasselbe Programm für eine PULL-UP-FIELD-Operation maximal 55 Constraints. Obwohl die Werte nur mit Einschränkungen zu vergleichen sind, weil die Vorarbeit andere Constraintregeln verwendet, wird deutlich, dass selektive Constraintterzeugung die Anzahl von Constraints drastisch reduziert. Der Fortpflanzungseffekt hat bei der untersuchten Refaktorisierung keine besonders große Reichweite; mit anderen Worten, der Constraintgraph zerfällt in kleine Komponenten.

Steimann et al. [35] bestätigen diese Aussage für andere Refaktorisierungen, allerdings am Beispiel von Eiffel-Programmen. Die Mediane der Anzahlen notwendiger Constraints, den diese Autoren ermitteln, sind in einer ähnlichen Größenordnung wie die in dieser Arbeit festgestellten Werte. Dagegen sind dort die Verteilungen wesentlich stärker rechtsschief.

6.1.2 Überzählige Constraints

Der Algorithmus berechnet wenige Constraints. Das belegt jedoch nicht, dass darunter nicht überflüssige Constraints sind. Empirisch ist das allerdings nur schwer nachzuweisen.

Nach Vorgabe wird beispielsweise für jedes referenzierte Feld zusätzlich zum *Inh-1*-Constraint ein *Acc-1*-Constraint erzeugt, obwohl das *Inh-1*-Constraint die Beschränkung des *Acc-1*-Constraints bereits einschließt. Für Felder ist die Prämisse der Regeln *Acc-1* und *Inh-1* identisch. Die Konklusion von *Inh-1* impliziert die Konklusion von *Acc-1*. Das *Acc-1*-Constraint könnte also gestrichen werden, ohne dass ungültige Refaktorisierungen entstünden.

Aus diesem Grund kann man nicht einfach auf überzählige Constraints prüfen, indem man einzelne Constraints aus der Constraintmenge entfernt und beobachtet, ob weiterhin eine korrekte Refaktorisierung resultiert. Auch wenn die Refaktorisierung korrekt bleibt, könnte das gestrichene Constraint von der Spezifikation gefordert sein.

Der umgekehrte Schluss wäre möglich. Stellt man nach Entfernen eines Constraints fest, dass die Refaktorisierung nicht mehr korrekt ausgeführt wird, dann war das Constraint nicht überflüssig. Daraus lässt sich jedoch nicht schließen, dass keine überzähligen Constraints erzeugt werden.

Es ist also schwierig, automatisch überflüssige Constraints zu identifizieren. Diese Arbeit beschränkt sich deshalb darauf, diese Frage anhand konstruierter Testfälle zu prüfen. Diese Testfälle sichern zu, dass der Algorithmus genau die erwarteten Constraints erzeugt und keine anderen. Dieses Vorgehen hat allerdings den Nachteil, dass man nicht alle möglichen Programmkonstellationen bedenken wird, die zu überzähligen Constraints führen können.

6.1.3 Anzahl von Lösungen der Constraintmengen

Nur 41 % der auf den Probanden insgesamt versuchten PULL-UP-FIELD-Operationen sind ausführbar, da nur 41 % der Constraintmengen wenigstens eine Lösung haben (vgl. Tabelle 3, S. 38). Dieser Anteil ist nicht unrealistisch. Immerhin wird versucht, jedes Feld in jeden erweiterten Supertypen hochzuziehen. Ist der Ziel-Classifer ein Interface, ist das nur für *static final* deklarierte Felder möglich. Diese Felder sind recht selten, Interfaces als Supertypen dagegen eher verbreitet. Die Variation des Anteils ausführbarer Refaktorisierungen zwischen den Probanden könnte ein unterschiedliches Ausmaß der Verwendung von Interfaces widerspiegeln. Der geringe Anteil an ausführbaren Refaktorisierungen könnte allerdings auch durch zu starke Einschränkungen verursacht sein (vgl. Abschnitt 6.2, S. 45).

Die Verteilung der Anzahlen von Lösungen ist sehr schief. In wenigen Fällen treten extrem große Werte auf (vgl. Abschnitt 5.2, S. 39 und Tabelle 11, S. 73 im Anhang). Das ist so auch zu erwarten. Hängen mehrere Felder über Constraints zusammen, dann kann eine nahezu beliebige Kombination dieser Felder hochgezogen und jeweils mit verschiedenen Zugriffsmodifikatoren ausgezeichnet werden. Die Anzahl der Kombinationen wächst exponentiell mit der Anzahl der Constraintvariablen. Ein Beispiel: Bei nur 6 Feldern, die innerhalb eines Package hochgezogen werden können oder nicht, erhält man schon $(3+4)^6 \approx 120\,000$ Kombinationen. Folgendes merkwürdige Beispiel ist einem Fall aus dem Probanden *Apache Codec* nachempfundenen.

```

1  class A { }
2  class B extends A {
3      int b1; int b2; int b3;
4      int b4; int b5; int b6;
5      int c= f(b1, b2, b3, b4, b5, b6);
6      int f(int... b) { return c; }
7  }
```

Soll in diesem Programm das Feld `b1` in die Klasse `A` hochgezogen werden, dann können die Felder `b2` bis `b6` folgen oder auch nicht. Werden sie nicht hochgezogen, können sie *private* bis *public* ausgezeichnet werden, werden sie hochgezogen, können sie *package* bis *public* ausgezeichnet werden. Constraints müssen für alle Felder erzeugt werden, weil Referenzen in der prinzipiell beweglichen Deklaration des Felds `c` vorkommen.

Interessant ist die Verteilung der Anzahl von Lösungen im Hinblick auf die Wahl der maximalen Anzahl von Lösungen, die beim Testen der Refaktorisierung in den Quellcode zurückgeschrieben werden und dann auf Kompilierbarkeit und fehlerfreies Durchlaufen der Testfälle von Probanden geprüft werden (vgl. Abschnitt 4.7, S 35). Wie Tabelle 11 (S. 73) zeigt, nimmt die Anzahl von Fällen oberhalb von 30 Lösungen stark ab. Dieser Wert bietet sich daher für größere Projekte als Maximum zu prüfender Lösungen für eine Refaktorisierungsoperation an.

6.2 Testergebnisse

Dieser Abschnitt bespricht die Ergebnisse der Testläufe, die Abschnitt 5.3 (S. 39) berichtet. Hier geht es um die Fragen, ob zu stark oder zu schwach einschränkende bzw. zu wenige Constraints erzeugt werden und ob das Verfahren korrekte Refaktorisierungen berechnet. Außerdem behandelt dieser Abschnitt die Konsequenzen der Mängel der Quellcode-Änderung.

6.2.1 Zu starke Einschränkung

Ob falsche, zu strenge Constraints berechnet werden, lässt sich nur bedingt empirisch prüfen (vgl. Abschnitt 3.7, S. 22). Unter einem zu strengen Constraint wird hier auch der äquivalente Fehler verstanden, einer Constraintvariablen einen zu kleinen Wertebereich zuzuweisen.

Wie Abschnitt 5.3 (S. 39) darstellt, sind die Constraints vor Ausführung der Refaktorisierung in allen Fällen erfüllt. Sie sind für alle berechneten Lösungen mit hochgezogenem Feld ebenfalls erfüllt. Deshalb können falsche, zu stark beschränkende Constraints für korrekte Programme nicht erzeugt worden sein.

Abschnitt A.9 (S. 69) begründet aber bereits, dass und warum zu strenge Constraints erzeugt werden. Diese Constraints verlangt die Spezifikation, weil Constraintregeln oder eine Implementierung der Code-Transformation fehlen, die garantieren könnten, dass eine entsprechende Refaktorisierung korrekt ausführbar wäre. So wird beispielsweise das Hochziehen aller Felder unterbunden, deren Initialisierungsausdruck einen Methoden- oder Konstruktoraufruf enthält, weil Constraintregeln nicht berücksichtigt werden, die eine Änderung der Bindungen von Methodenaufrufen verhindern. Dieses Vorgehen hat zwar den Vorteil, dass solche Refaktorisierungen nicht zu Fehlern führen, aber gleichzeitig schließt es Refaktorisierungen aus, die eigentlich korrekt ausführbar wären.

Ob zu strenge Constraints erzeugt werden, die *unbeabsichtigt* Refaktorisierungen unterbinden, die eigentlich ausführbar wären, war im Rahmen dieser Arbeit empirisch nicht prüfbar. Ein Werkzeug, das dieselben Constraintregeln erschöpfend auf alle Stellen des Programms anwendet, steht zum Vergleich nicht zur Verfügung. Ein Vergleich der Anzahlen möglicher Refaktorisierungsoperationen hätte empirisch die Vermutung prüfen können, dass erschöpfende Constraintzeugung mehr Refaktorisierungsoperationen zulässt, etwa weil mehr Constraintvariablen erzeugt werden.

Ein Vergleich mit den Ergebnissen einer Standard-Implementierung der PULL-UP-FIELD-Refaktorisierung erforderte, dass für jeden festgestellten Unterschied manuell geprüft wird, ob der Unterschied beabsichtigt und gerechtfertigt ist oder nicht. Mit dem in dieser Arbeit verwendeten RTT ist es nicht ohne Weiteres gelungen, zu zählen, wie viele korrekte Refaktorisierungen mit der Standard-Implementierung von Eclipse auf den untersuchten Probanden möglich wären. Die vorliegende Arbeit unterlässt deshalb den quantitativen Vergleich von Refaktorisierungen, die mit anderen Werkzeugen ausführbar sind.

Die Frage, ob fehlerhaft ein Hochziehen von Feldern durch zu starke Einschränkung unterbunden wird, prüfen deshalb nur konstruierte Testfälle, die naturgemäß ein unbeabsichtigtes Ausschließen erlaubter Refaktorisierungen schlecht feststellen.

6.2.2 Zu schwache Einschränkung

Die Frage, ob die Implementierung zu wenige Constraints erzeugt und deshalb fehlerhafte Refaktorisierungen ausführt, ist gesichert mit Ja zu beantworten. Abschnitt A.9 (S. 69) führt Gründe und Beispiele dafür an. So fehlen bisher Constraintregeln, nach denen Constraints erzeugt werden, die Verdunkeln (*obscuring*) oder eine Änderungen des Kontrollflusses unterbinden.

Die interessantere Frage ist wiederum, ob es Fälle gibt, in denen *unbeabsichtigt* zu wenige oder zu schwache Constraints erzeugt werden. Zweck des Ausführens von Refaktorisierungen auf Probanden und anschließendes Testen, ob die refaktorierten Programme noch fehlerfrei kompilieren und ihre Testfälle bestehen, ist gerade, diese Fehler aufzudecken. Fehlerhaft zu wenige oder zu schwache Constraints konnten die Testläufe nicht feststellen (vgl. Abschnitt 5.3, S. 39).

Allerdings weisen die auf den Probanden ausgeführten Tests auch ernüchternd wenige der bekannten Lücken der Constraintregeln nach. Lediglich die Grenze der Refaktorisierbarkeit introspezierender Programme entdecken die Test-Suiten der Probanden (vgl. Abschnitt 5.3,

S. 40). Weitere Fehlschläge, die einen Mangel an Constraints ans Tageslicht befördern, sind nicht aufgetreten.

Dieses Phänomen beschränkter Schärfe der Testfall-Orakel berichtet bereits die Vorarbeit [36] am Beispiel von MOVE-CLASS- und PULL-UP-METHOD-Refaktorisierungen. Die Frage, ob fehlerhaft zu wenige oder zu gering einschränkende Constraints erzeugt werden, ist deshalb nicht überzeugend zu beantworten. Die Schlussfolgerung, dass die Constraintenerzeugung jenseits der im Abschnitt A.9 (S. 68) genannten Grenzen vollständig ist, wäre verfrüht. Es zeigt sich dagegen eher, dass die Testbasis zu klein ist.

Es ist nicht ganz einfach, die Testbasis deutlich zu vergrößern, da die Anzahl quelloffener Programme, die über eine gute Testabdeckung verfügen, begrenzt ist. Der Ansatz von Soares et al. [28] könnte helfen, das Problem der mangelnden Testabdeckung auszugleichen. Mit ähnlicher Zielsetzung setzen die Autoren ein Werkzeug ein, das automatisch Test-Suiten für zu refaktorisierende Programme erzeugt. Diese Test-Suiten sollen sicherstellen, dass Refaktorisierungen die Bedeutung von Programmen nicht verändern.

Im Übrigen wäre der Bedarf an Rechenkapazität für das Prüfen der Implementierung auf einer größeren Testbasis nicht unerheblich. Es wäre auch hilfreich, eine Möglichkeit zu implementieren, den Algorithmus ohne grafische Oberfläche auf Probanden ausführen und testen zu können.

6.2.3 Mängel der Quellcode-Änderung

Die Testläufe belegen zwei bekannte Mängel der Implementierung der Änderung des Quellcodes (Abschnitt 4.6, S. 34 bzw. Abschnitt 5.3, S. 39).

- In bestimmten Fällen wird die vorgegebene Reihenfolge von Felddeklarationen nicht eingehalten. Das verursacht einen Übersetzungsfehler, wenn eine Felddeklaration ein anderes Feld referenziert, das versehentlich an das Ende des Java-Files geraten ist.
- Eine vollständige Qualifizierung von verlagerten Classifier-Referenzen ist bisher nicht implementiert. Das verwendete Werkzeug der Eclipse-JDT zur Verlagerung von Feldern erzeugt deshalb ungewollt Import-Anweisungen. Im günstigsten Fall werden die Fehler beim Berechnen der Quellcode-Änderung erkannt. Im schlechtesten Fall führt die unbeabsichtigte Import-Anweisung zu unbemerkter Änderung der Semantik des refaktorierten Programms.

Diese Mängel sind auch in Form fehlschlagender Testfälle der Quellcode-Änderung dokumentiert.

Die Mängel sind nur durch eine Reimplementierung der in dieser Arbeit wiederverwendeten Quellcode-Änderung der Eclipse-JDT zu beheben. Fehler infolge dieser Mängel treten in den Testläufen nur sehr selten auf. Eine Analyse dieser Fälle und ihre manuelle Korrektur zeigt, dass die Fehler bei der Quellcode-Änderung andere Fehler des Algorithmus nicht verdecken. Dennoch bleibt die Situation unbefriedigend. Angesichts der Tatsache, dass die Quellcode-Änderung der Prüfung des Algorithmus dient und nicht Teil seiner Implementierung ist, mögen die Mängel als pragmatischer Kompromiss durchgehen.

6.3 Eignung für die Praxis

Neben der Korrektheit ist der Aufwand ein entscheidendes Kriterium zur Klärung der Frage, ob ein Algorithmus für ein praktisch einsetzbares Refaktorisierungswerkzeug geeignet ist. Diese Frage zu beantworten, ist sekundäres Ziel der Arbeit. Im Folgenden werden der Rechenaufwand, seine Abhängigkeit von der Größe des zu refaktorisierenden Programms und der Implementierungsaufwand besprochen.

6.3.1 Rechenzeit

Eine praktikable Refaktorisierung reagiert augenblicklich und ist nahtlos in das Editieren von Programmen integrierbar [23]. Diese Anforderung erfüllt die Implementierung des Algorithmus: Die Rechenzeiten für das Erzeugen und Lösen der Constraintmengen bleiben, von Ausnahmen abgesehen, deutlich unter dem Aufwand, der für die Änderung des Quellcodes und das inkrementelle Kompilieren erforderlich ist (Abschnitt 5.4, S. 41). Der größte Teil der Rechenzeit wird für das Erzeugen der Constraints, also für die Analyse des zu refaktorisierenden Programms, benötigt. Die Implementierung der Analyse ist bisher nicht auf Rechenzeit optimiert. Eine Verbesserung des Laufzeitverhaltens des Algorithmus, vielleicht auf Kosten der Speicherplatzanforderungen, dürfte also möglich sein.

Für das Projekt *JHotDraw* benötigt die selektive Constraintterzeugung und -lösung im Durchschnitt 100 ms und im schlechtesten Fall rund 750 ms. Dagegen berichtet die Vorarbeit [36] für dasselbe Projekt eine mittlere Rechenzeit allein für die Lösung der Constraints von 6 542 ms – allerdings auf einem etwas weniger leistungsfähigen Rechner als dem hier verwendeten. Die durch selektive Constraintterzeugung erreichte Einsparung an Rechenzeit ist in der drastisch reduzierten Anzahl von Constraints begründet.

Steimann et al. [35] berichten am Beispiel anderer constraintbasierter Refaktorisierungen von Eiffel-Programmen Rechenzeiten für die Constraintlösung in der Größenordnung von Millisekunden. Das stimmt mit den in dieser Arbeit gemessenen Rechenzeiten für die Constraintlösung überein. Das unterstützt die Vermutung, constraintbasierte Refaktorisierung, die nur die benötigten Constraints ermittelt und löst, ist auch für andere als die hier untersuchte PULL-UP-FIELD-Refaktorisierung effizient.

Schäfer und de Moor [27] schreiben, ihrer Refaktorisierung mangle es etwas an Leistungsfähigkeit und selbst auf sehr kleinen Programmen wären bis zu fünf Sekunden Rechenzeit nötig. Im Vergleich scheint der in dieser Arbeit verfolgte Ansatz erfolgversprechend zu sein, jedenfalls für die untersuchte PULL-UP-FIELD-Refaktorisierung.

Die Rechenzeitanforderungen des Verfahrens von Soares et al. [28], das Korrektheit von Refaktorisierungen im Nachhinein durch automatisch erzeugte Unit-Tests absichert, benötigt für eine Refaktorisierungsoperation mehrere Sekunden. Diese Rechenzeit ermitteln die Autoren auf einem ähnlich leistungsfähigen Rechner wie dem hier verwendeten. Das Verfahren ist viel allgemeiner einsetzbar als constraintbasierte Refaktorisierung bisher. Seine vergleichsweise hohen Rechenzeitanforderungen sind jedoch unausweichlich, da die Programme immer erst neu kompiliert werden müssen. Schneller als der Compiler kann das Verfahren von Soares et al. deshalb gar nicht werden.

6.3.2 Speicherplatz

Die Speicherplatzanforderungen der Implementierung des Algorithmus zur selektiven Constraintterzeugung und -lösung sind im Vergleich zu den Anforderungen der Entwicklungsumgebung nicht gravierend und stehen seiner praktischen Anwendung nicht im Weg. Das ist bei einer erschöpfenden Constraintterzeugung, wie sie die Vorarbeit [36] durchführt, natürlich ganz anders und in der unterschiedlichen Zielsetzung begründet.

Ob die ökonomische Verwendung des Speicherplatzes allein in der geringen Anzahl erforderlicher Constraints oder auch in der Entwurfsentscheidung für die vorrangige Verwendung des Java-Modells anstelle des AST und seiner Bindungen begründet ist, bleibt offen. Immerhin ließe der Speicherbedarf Spielraum für zusätzliches Zwischenpuffern von AST in einem größeren LRU-Cache.

6.3.3 Komplexität

Theoretische Überlegungen zeigen, dass der Algorithmus zur selektiven Constraintterzeugung eine Laufzeitkomplexität höherer polynomialer Ordnung aufweist (vgl. Abschnitt 3.4, S. 19).

Die Anzahl von Probanden unterschiedlicher Größe ist jedoch viel zu klein, um diese Hypothese zu bestätigen oder zu widerlegen. Bild 8 (S. 42) zeigt eine wenig dramatische Zunahme der Rechenzeiten mit der Größe der Probanden. Die Steigung der eingezeichneten Regressionslinie ist allerdings stark von dem einzigen Probanden mittlerer Größe von rund 150 000 Codezeilen bestimmt und deshalb wenig gesichert. Da Messungen an großen Probanden fehlen, ist eine Extrapolation auf große Programme ausgeschlossen.

Die Abschätzung des Speicherplatzbedarfs zeigt ebenfalls eine Zunahme der Anforderungen mit der Größe des Probanden (vgl. Abschnitt 5.4, S. 41). Die Speicherplatzanforderungen sind verglichen mit dem Bedarf der Entwicklungsumgebung auch beim Probanden mittlerer Größe unproblematisch. Die Unsicherheit der Schätzung lässt eine Verallgemeinerung auch dieser Werte nicht zu.

Die Ergebnisse der Aufwandsermittlung belegen, dass die Zunahme des Rechenaufwands der selektiven Constraint-erzeugung mit zunehmender Größe des zu refaktorisierenden Programms im Rahmen von kleinen und mittleren Programmen kein Problem ist. Um die Skalierbarkeit des Algorithmus tatsächlich beurteilen zu können, müsste er auf größere Probanden als die in dieser Arbeit untersuchten angewendet werden.

6.3.4 Implementierungsaufwand

Der Implementierungsaufwand, gemessen in Codezeilen, beträgt für die selektive Constraint-erzeugung und die darauf basierende Refaktorisierung rund 4 000 Zeilen (vgl. Abschnitt 5.4, S. 42).

Schäfer und de Moor [27] berichten für ihre PULL-UP-MEMBER-Refaktorisierung einen Implementierungsaufwand von 208 Zeilen. Die Werte sind nur mit Vorbehalt zu vergleichen. Die Werkzeuge benutzen ganz unterschiedliche Bibliotheken. Hier sind es die Eclipse-JDT und der Constraintlöser *Choco*, dort ist es der *JastAddJ*-Compiler. Die Refaktorisierungen sind ebenfalls unterschiedlich. Hier werden Felder verlagert und Zugriffsmodifikatoren verändert, dort werden beliebige Member verlagert und Referenzen neu qualifiziert. Von untergeordneter Bedeutung ist angesichts dieser Unterschiede, dass die Anzahl von Codezeilen allein kein besonders guter Maßstab für den Implementierungsaufwand ist. Dennoch lässt sich angesichts eines Faktors 20 nicht bestreiten, dass Schäfer und de Moors Implementierung durch ihren höheren Abstraktionsgrad den Aufwand erheblich reduziert.

Der Implementierungsaufwand der PULL-UP-MEMBER-Refaktorisierung der Eclipse-JDT, den Schäfer und de Moor [27] mit 1 700 Zeilen angeben, liegt näher an dem Aufwand, der für die hier verwendete selektive Constraint-erzeugung erforderlich ist. Einem Einsatz des Algorithmus in der Praxis steht der Implementierungsaufwand demnach nicht entgegen.

6.4 Probleme der Implementierung

Die Grenzen der Constraintregeln stellt Abschnitt A.9 (S. 68) bereits ausführlich dar. Abschnitt 6.2 (S. 44) bespricht Probleme, die die Testläufe entdecken. Dieser Abschnitt nun diskutiert Probleme der Gestaltung des Algorithmus und der vorgelegten Implementierung, die in Testläufen nicht zutage treten.

6.4.1 Kopplung von Graphendurchlauf und Constraintregeln

Ein Nachteil der Implementierung des Graphendurchlaufs ist, dass die angewendeten Constraintregeln „fest verdrahtet“ sind (vgl. Abschnitt 4.4, S. 29). Die eigentliche Erzeugung der Constraints wird an einen *builder* delegiert (S. 31). Das lindert die starke Abhängigkeit des Durchlaufs von den Constraintregeln jedoch nicht. Die feste Verdrahtung ist nicht mit einem entscheidenden Effizienzgewinn zu rechtfertigen, sondern im Wesentlichen dem Entwicklungsprozess geschuldet.

Die enge Kopplung zwischen Constraintterzeugung und Programmanalyse ist aber auch darin begründet, dass die Programmelement-Eigenschaft Empfängertyp $r.\rho$ und das Prädikat „inherits“ bei Erzeugung und Transformation jeweils durch Programmanalyse ermittelt werden.

6.4.2 Konzept der Einflussbereiche

Problematischer an der Implementierung dürfte die pragmatische, aber wenig formal begründete Behandlung der in dieser Arbeit sogenannten Einflussbereiche sein. Einflussbereiche sind die Bereiche eines Programms, die auf Konstellationen abgesehen werden, die zur Erzeugung eines Constraints für die konkrete Refaktorisierungsoperation nach einer gegebenen Constraintregel führen können (vgl. Abschnitt 4.2, S. 25 und Abschnitt 4.4, S. 30).

Günstiger wäre es vermutlich, diese Beschränkung der Programmanalyse auf Teile eines Programms in der Prämisse der Constraintregel vorzunehmen. Damit würde die Constraintregel jedoch für die PULL-UP-FIELD-Refaktorisierung spezialisiert und ihre Allgemeingültigkeit verlieren.

Eine ähnliche Verfeinerung der Constraintregeln mit dem Ziel, die Anzahl zu behandelnder Programmelemente zu reduzieren, realisiert die vorliegende Arbeit bereits. Beispielsweise schließt die Prämisse der Constraintregel *Acc-2* (vgl. Anhang A.2, S. 59) Felder aus, die statisch sind oder in einer *this expression* referenziert werden, weil für diese Felder das Constraint ohnehin niemals greift. Diese Einschränkung ist für alle Refaktorisierungen gültig, also keine Spezialisierung der Regel für eine bestimmte Refaktorisierung.

Steimann [34] führt ähnliche Spezialisierungen allgemeiner Constraintregeln für eine konkrete Refaktorisierung programmatisch aus, um das Problem der vorausschauenden Anwendung der Regeln zu lösen.

6.4.3 Vorausschau

Das Problem, Constraintregeln vorausschauend anzuwenden zu müssen, tritt bei der vorliegenden Arbeit nicht auf, weil die Regeln von vornherein so formuliert sind, dass alle veränderlichen Eigenschaften von Programmelementen in der Konklusion der Regel vorkommen (vgl. Anhang A, S. 56). Das schränkt die Allgemeingültigkeit der Regeln nicht ein, führt allerdings zu komplexeren Ausdrücken in der Konklusion [34].

Die in der Vorarbeit [36] vorausschauend zusätzlich erzeugten Constraints werden hier also ebenfalls erzeugt. Zu einer Explosion der Anzahl von Constraints kommt es aus folgenden Gründen jedoch nicht: (1) Die Wertebereiche jeder Constraintvariablen sind sehr klein. (2) Sie werden sofort bei der Constraintterzeugung unter Berücksichtigung des Refaktorisierungsziels festgelegt. (3) Die Constraintterzeugung wird über unveränderliche Eigenschaften von Programmelementen hinaus nicht fortgesetzt. (4) Nur Programmelemente innerhalb der im vorhergehenden Abschnitt diskutierten Einflussbereiche werden überhaupt in Betracht gezogen. (5) Gelöste Constraints werden gar nicht erst generiert (vgl. Abschnitt 3.5, S. 20).

Steimann [34] hat das Problem der vorausschauenden Anwendung von Constraintregeln mittlerweile grundsätzlich gelöst (vgl. Abschnitt 2.3, S. 12). Seine Ergebnisse berücksichtigt die hier vorgelegte Implementierung der Constraintterzeugung noch nicht.

6.4.4 Abstraktionsebene

Wie die Vorarbeit auch, transformiert die vorliegende Arbeit erzeugte Constraints in einem separaten Schritt in eine Form, die ein universeller Constraintlöser verarbeiten kann (vgl. Bild 5, S. 26). Diese Arbeit wählt für die zunächst erzeugten Modell-Constraints und die darin vorkommenden Constraintvariablen eine Repräsentation, die der Problemdomäne sehr nahe ist: Erweiterte Elemente des Eclipse-Java-Modells dienen direkt als Constraintvariablen (vgl. Abschnitt 3.1, S. 14).

Im Fall von Felddeklarationen treten bei dieser Repräsentation 2-Tupel als Elemente des Wertebereichs von Constraintvariablen auf. Die Komponenten der Tupel sind Zugreifbarkeit $d.\alpha$ und Ort $d.\lambda$ einer Felddeklaration d .

Diese Wahl der Abstraktionsebene hat die Implementierung des Algorithmus zunächst vereinfacht (vgl. Abschnitt 4.3, S. 27), den Aufwand für die Transformation von Modell-Constraints in Integer-Constraints aber erhöht (vgl. Abschnitt 4.5, S. 31). Ob die größere Nähe der Constraintvariablen zum modellierten Problem die Constraintterzeugung nachvollziehbarer und flexibler macht, bleibt fraglich.

Die Constraintlösung könnte durch eine geschicktere Transformation der Constraints möglicherweise beschleunigt werden. Allerdings ist die Rechenzeit für die Lösung verglichen mit der Zeit für die Erzeugung der Constraints ohnehin sehr gering.

Die Repräsentation von Programmelementen auf Basis von *properties*, die Steimann et al.[35] mittlerweile entwickelt haben, ist näher am Modell ganzzahliger Constraintvariablen (vgl. Abschnitt 3.1, S. 14). Das hat den Vorteil, dass sich die Programmelemente unmittelbarer in die vom Constraintlöser verarbeitbare Form überführen lassen und eine direkte Beziehung zwischen Constraintlösung und Repräsentation des Programms besteht.

6.5 Constraintlösung parallel zur Constraintterzeugung

Diese Arbeit verzichtet auf Constraintpropagierung und Lösungssuche parallel zum Graphendurchlauf für die Constraintterzeugung. Folgende Absätze diskutieren Ansätze zur Lösung der Constraintnetzwerke bzw. zum Erkennen ihrer Inkonsistenz gleich bei der Erzeugung, die mit dem Ergebnis der Arbeit nun auf der Hand liegen.

6.5.1 Begriffe und Einordnung

Dynamische Einschränkungprobleme (*dynamic constraint satisfaction problems*, DCSP) können als Folge statischer Einschränkungprobleme verstanden werden, bei der jedes neue Problem durch kleinere Änderungen der Definition aus dem vorhergehenden Problem entsteht. Die Definition des Problems kann sich dabei durch Hinzufügen, Ändern oder Entfernen von Constraints, von Constraintvariablen oder ihrer Wertebereiche ändern [41]. Von DCSP werden *bedingte Einschränkungprobleme* (*conditional constraint satisfaction problems*) unterschieden, die Probleme behandeln, deren Lösungen sich in ihrer Struktur unterscheiden. Bei diesen Problemen sind Teilmengen der Constraintvariablen und Constraints nur in Abhängigkeit der Wertebelegung anderer Variablen aktiv [41].

Verfahren zur Lösung von DCSP lassen sich genauso wie Verfahren zur Lösung allgemeiner Einschränkungprobleme (siehe Abschnitt 3.6, S. 20) in Suchverfahren und Konsistenztechniken (Constraintpropagierung) sowie Kombinationen davon einteilen. Bei DCSP ist eine Lösung eines ähnlichen, nämlich die des vorhergehenden Problems, bekannt. Deshalb eignen sich zur Lösung von DCSP spezialisierte Suchverfahren, die unter dem Begriff *lokale Suche* (*local search*) bekannt sind [41].

Refaktorisierung lässt sich als Folge zweier ähnlicher Constraintprobleme auffassen. Das erste Constraintproblem ist das bereits gelöste Constraintnetzwerk, das dem Programm vor der Refaktorisierung entspricht. Die Refaktorisierungsoperation schreibt für wenige Constraintvariablen neue Wertebereiche vor bzw. fügt einstellige Constraints hinzu. Sie spezifiziert also ein zweites, geringfügig verändertes Constraintproblem, dessen Lösung zu suchen oder abzuleiten ist. Das zu lösende Problem fällt damit in die Klasse der DCSP, für die effiziente Verfahren zur Lösung beschrieben sind.

Constraintpropagierung läuft auf eine Änderung des Constraintproblems hinaus (vgl. Abschnitt 3.6, S. 20). Sie könnte parallel zur Constraintterzeugung durchgeführt werden, um eine anschließende Suche nach einer Lösung zu beschleunigen. Die vorliegende Arbeit betrachtet diese Möglichkeit wegen des damit verbundenen Aufwands nicht. Es stellt sich aber die Frage,

ob eine Anwendung der für DCSP geeigneten Suchverfahren oder eine Konsistenzprüfung von Constraints parallel zur Constraintterzeugung zu einem effizienteren Algorithmus führt.

6.5.2 Suchen einer Lösung

Diese Arbeit prüft zum Test des Algorithmus, ob das erzeugte Constraintnetzwerk mit der ermittelten Ausgangsbelegung der Constraintvariablen erfüllt ist (vgl. Abschnitt 5.3, S. 39). Das Constraint, das das Ziel der Refaktorisierung durchsetzt, wird erst danach zum Constraintnetzwerk hinzugefügt (Abschnitt 4.4, S. 29). An dieser Stelle könnte ohne Weiteres die Belegung der Constraintvariablen, die das hochgezogene Feld repräsentiert, so angepasst werden, dass der Ort des Felds dem Ziel der Refaktorisierung entspricht und seine Zugreifbarkeit vom ursprünglichen Ort aus garantiert ist. Ergäbe dann eine Prüfung der Wertebelegung auf Konsistenz, dass das Constraintnetzwerk erfüllt ist, wäre man an dieser Stelle fertig, ohne erst einen Constraintlöser zu bemühen. Die vorliegende Arbeit nutzt diese Möglichkeit nicht, weil die Constraintlösung im Gegensatz zur Constraintterzeugung vergleichsweise wenig Rechenzeit benötigt und ohnehin ein Constraintlöser zum Test des Algorithmus eingesetzt werden muss.

Interessanter wäre die Möglichkeit, die Constraintterzeugung zu beenden, wenn eine dem Refaktorisierungsziel entsprechend modifizierte Ausgangsbelegung der Constraintvariablen alle Constraints erfüllt, an denen das veränderte Programmelement direkt beteiligt ist. Ein entsprechender Algorithmus liegt nun auf der Hand: Per Breitendurchlauf werden ähnlich wie bei dem in Bild 4 (S. 19) dargestellten Algorithmus alle Constraints ermittelt, in denen das hochzuziehende Feld vorkommt. Diese Constraints werden mit allen weiteren darin vorkommenden Programmelementen vervollständigt. An dieser Stelle wird die Ausgangsbelegung der Eigenschaften des hochzuziehenden Felds so angepasst, dass das Ziel der Refaktorisierung erreicht ist. Sind damit immer noch alle Constraints erfüllt, liegt eine Lösung des Problems vor und der Algorithmus kann terminieren, bevor eine vollständige Komponente des Constraintgraphen ermittelt ist.

Diesen skizzierten Algorithmus könnte man in Anlehnung an die oben genannten Verfahren lokaler Reparatur verfeinern. Sind die Constraints nicht erfüllt, würde man den Ablauf der Constraintterzeugung damit fortsetzen, die Belegung der Programmelemente anzupassen, die zu einer Verletzung von Constraints führen, und den Durchlauf bei den veränderten Programmelementen fortsetzen. Auf diese Weise wird die Änderung der Wertebelegung fortgepflanzt, die Constraints werden aber nicht propagiert.

Die Entscheidung für einen Tiefendurchlauf anstelle eines Breitendurchlaufs hat die Chance einer früheren Terminierung des Algorithmus zur Constraintterzeugung bei Auffinden einer Lösung des Constraintnetzwerks verbaut. Erst wenn der Tiefendurchlauf terminiert, sind alle Constraints, an denen das durch die Refaktorisierung zu ändernde Programmelement beteiligt ist, gefunden. Die Frage, wie groß der Effizienzgewinn durch frühere Terminierung der Constraintterzeugung wäre, kann die vorliegende Arbeit nicht beantworten, weil eine Umstellung der Implementierung von Tiefendurchlauf auf Breitendurchlauf im Rahmen der Arbeit nicht mehr machbar war.

6.5.3 Prüfen auf Konsistenz

Die vorgelegte Implementierung des Tiefendurchlaufs terminiert an grundsätzlich gelösten oder an redundanten Constraints (vgl. Abschnitt 3.5, S. 20). Beim Durchlauf wird jedoch nicht mehr jedes Constraint individuell daraufhin geprüft, ob es gelöst ist. Ob der nicht unerhebliche Aufwand für diese Prüfung beim Durchlauf lohnt, bleibt fraglich.

Während des Durchlaufs wird auch nicht geprüft, ob ein erzeugtes Constraint überhaupt erfüllbar oder mit seinen benachbarten Constraints konsistent ist. Knoten- und Kanten-Konsistenz wird also weder hergestellt noch geprüft. Deshalb wird eine Inkonsistenz des erzeugten

Constraintnetzwerks erst festgestellt, wenn der Constraintlöser keine Lösung des Constraintproblems findet. Der Algorithmus könnte jedoch sofort terminieren, wenn er auf ein inkonsistentes Constraint stößt. Die Frage, ob eine Konsistenzprüfung der Constraints parallel zu ihrer Erzeugung den Algorithmus merklich beschleunigt, untersucht die vorliegende Arbeit nicht (vgl. Abschnitt 1.2, S. 7).

6.5.4 Strengere Selektion von Constraints

Thies [38] schlägt strengere Kriterien für die Selektion von Constraints vor, als die vorliegende Arbeit realisieren konnte (vgl. Abschnitt 3.5, S. 20). Thies [38] schreibt, sollten Lösungen eines Constraintnetzwerks möglich sein, die Belegungen aller Variablen einzelner Constraints unverändert lassen, dann seien diese Constraints unnötig generiert worden.

Zu bedenken ist allerdings, dass man zum Zeitpunkt der Ermittlung der Constraints die Lösung noch nicht kennt. Man kann deshalb nicht ohne Weiteres voraussehen, welche Ausgangsbelegungen von Variablen mit einer Lösung konsistent sein werden. Das gilt im Allgemeinen auch für die oben (Abschnitt 6.5.2, S. 51) diskutierte Lösungssuche parallel zur Constraintzerzeugung, weil die dort besprochenen Suchverfahren ein Rücksetzen nicht ausschließen.

Es gibt allerdings Verfahren der Einschränkungsfortpflanzung (vgl. Abschnitt 3.6, S. 20), die garantieren, dass eine anschließende Lösungssuche rücksetzungsfrei verläuft. Die Verbindung der Constraintzerzeugung mit einer solchen Einschränkungsfortpflanzung wäre eine Voraussetzung dafür, Thies' strengere Selektionskriterien umzusetzen.

Die Frage, ob Einschränkungsfortpflanzung parallel zur Constraintzerzeugung die Anzahl erzeugter Constraints stärker reduzieren kann, als es die oben angesprochenen Suchverfahren parallel zur Constraintzerzeugung können, bleibt offen.

6.6 Offene Fragen

Dieser Abschnitt ergänzt weitere offene Fragen, die die Untersuchungsfrage der Arbeit berühren, aber nicht zu ihrem Kern gehören. Er schließt mit einer Zusammenfassung der Probleme, deren Bearbeitung die hier durchgeführte Untersuchung weiterbringen dürfte.

6.6.1 Normalisierung des Constraintnetzwerks

Die vorliegende Arbeit normalisiert die Constraintmenge nicht. Normalisieren der Menge bedeutete hier, jeweils die Constraints, die die gleichen Programmelemente enthalten, von vornherein konjunktiv zu einem zusammengesetzten Constraint zu kombinieren [3].

Die separate Behandlung der Constraints kostet zwar etwas Speicherplatz, erleichtert einer Erklärungskomponente aber, Gründe für das Fehlschlagen der Suche nach einer Lösung für das Constraintnetzwerk darzustellen. Die Frage, ob aus Refaktorisierungsaufgaben resultierende Constraintnetzwerke effizienter lösbar sind, wenn die Constraintmengen normalisiert werden, bleibt offen.

6.6.2 Optimierung der Lösung

Der eingesetzte Constraintlöser beherrscht keine Verfahren wie lokale Reparatur oder lokale Suche (*local search*), die die DCSP-Eigenschaft des Problems ausnutzen und zur Ausgangsbelegung der Constraintvariablen ähnliche Lösungen berechnen. Deshalb unterscheidet sich die hier ermittelte erstbeste Lösung oft stark von der Ausgangsbelegung, die dem Programm vor der Refaktorisierung entspricht. Das führt dazu, dass die Programme bei der Refaktorisierung unerwartet stark verändert werden.

Der gewählte Constraintlöser *Choco* beginnt die Suche nach einer Lösung an den unteren Grenzen der Wertebereiche. Die Transformation von Modell- in Integer-Constraints

berücksichtigt diese Eigenschaft des Constraintlösers. Sie ist so gestaltet, dass die Zugriffsmodifikatoren möglichst klein bestimmt und möglichst wenige Felder verlagert werden (vgl. Abschnitt 3.6, S. 21). Damit wird implizit eine Zielfunktion implementiert.

Um eine Refaktorisierungsoperation unterschiedlichen Anforderungen besser anpassen zu können, müsste eine Zielfunktion definiert werden, anhand derer die optimale Lösung gesucht werden kann. Optimierungsverfahren sind allerdings deutlich aufwendiger als Verfahren, die nur eine beliebige Lösung ermitteln [35]. Fragen der Definition geeigneter Zielfunktionen und Optimierung der Lösungen untersucht die vorliegende Arbeit nicht.

6.6.3 Qualifizierung statischer Felder

Ungelöst bleibt auch das Problem der Verlagerung nicht kanonisch qualifizierter, statischer Felder. Die angedachte [38] Requalifizierung statischer Felder war im Rahmen dieser Arbeit nicht umsetzbar. Im folgenden Beispiel könnte das statische Feld `C.i` in die Klasse `A` hochgezogen werden, würden seine Referenzen in Methode `n` kanonisch qualifiziert – das Feld `i` in Klasse `B` verbirgt nach dem Hochziehen das gleichnamige, statische Feld `A.i`.

```

1  class A { }
2  class B extends A {
3      int i= 0;
4  }
5  class C extends B {
6      static int i= 1;
7      C m(){ /*...*/
8          return this;
9      }
10     void n(){
11         while ( m().i>0 && m().i<9 ){ /*...*/ }
12     }
13 }
```

Das Beispiel zeigt aber auch, dass ein Algorithmus zur kanonischen Qualifizierung nicht trivial zu realisieren ist. Die Qualifizierung der Feldzugriffe auf `i` müsste so modifiziert werden, dass Anzahl und Reihenfolge der Aufrufe der Methode `m` und der Feldzugriffe erhalten bleiben. Eine Änderung des Rückgabetyps von `m` oder *upcasts* der Rückgabewerte an der Aufrufstelle wären einfacher, lösten die Aufgabe aber nicht. Eclipse' *Quick-Fix* gelingt die Lösung der Aufgabe jedenfalls nicht korrekt.

6.6.4 Die nächsten Aufgaben

Die vorliegende Arbeit wirft eine Reihe neuer Fragen auf und lässt andere Fragen unbeantwortet. Die folgende Aufzählung fasst die drängenderen Probleme zusammen. Manche könnten Gegenstand zukünftiger Arbeiten sein.

- Die bisherige Implementierung der Quellcode-Änderung greift teilweise auf die Code-Verlagerung durch die JDT zurück, die jedoch nur begrenzt steuerbar ist. Eine eigene, vollständige Implementierung der Quellcode-Änderung einschließlich einer Qualifizierung verlagertes Classifier-Referenzen kann einige Fehler beheben, die die Wiederverwendung bedingt.
- Constraintregeln, die die Bindungsregeln von Methodenaufrufen und Typvariablen beschreiben, werden bisher nicht berücksichtigt. Deshalb müssen vorläufig verhältnismäßig viele Refaktorisierungsoperationen ausgeschlossen werden, die eigentlich möglich wären. Auf Grundlage zusätzlicher Constraintregeln könnten auch andere Refaktorisierungen als `PULL-UP-FIELD` realisiert werden.

- Constraintregeln, Einflussbereiche und Graphendurchlauf sind in der vorliegenden Implementierung stark verwoben. Das vorläufige Konzept der Einflussbereiche dieser Arbeit ließe sich vermutlich mit Steimanns Transformation von Constraintregeln [34] kombinieren. Eine Reimplementierung des Algorithmus auf dieser Grundlage mit einer flexibleren Einbindung von Constraintregeln dürfte die Erweiterung des Algorithmus um neue Constraintregeln erheblich erleichtern.
- Verschiedene Wege der Optimierung des Algorithmus und seiner Implementierung sind bisher nicht untersucht. Besonders Erfolg versprechend dürfte der oben beschriebene Ansatz sein, den Durchlauf von Tiefen- auf Breitendurchlauf umzustellen und parallel zum Durchlauf nach einer Lösung zu suchen. Als Suchverfahren bieten sich die für DCSP entwickelten Algorithmen an. Es ist zu erwarten, dass die Kombination von Constraintinterzeugung und Lösungssuche die Anzahl zu ermittelnder Constraints weiter reduziert.
- Umgekehrt könnte eine Konsistenzprüfung beim Durchlauf frühzeitig Inkonsistenz des Constraintproblems ergeben und damit eine Fortsetzung der Constraintinterzeugung erübrigen. Ob die Laufzeiteinsparungen durch schnellere Feststellung der Undurchführbarkeit einer Refaktorisierungsoperation den Aufwand für Konsistenztechniken (Constraintpropagierung) rechtfertigt und die Constraintinterzeugung und -lösung insgesamt beschleunigt, bleibt fraglich.

6.7 Schlussfolgerungen

Die Vorarbeit [36] formuliert die Idee, für eine geplante Refaktorisierung nur gerade diejenigen Constraints zu berechnen, die diese Refaktorisierung beschränken. Alle anderen Constraints, die nicht berührt werden, müssten gar nicht erzeugt werden, da sie vor und nach der Refaktorisierung unverändert erfüllt bleiben. Primäres Ziel der vorliegenden Arbeit war, diesen Vorschlag der Vorarbeit umzusetzen und einen Algorithmus zur selektiven Constraintinterzeugung abzuleiten, zu implementieren und zu prüfen. Das ist gelungen. Die Idee der Vorarbeit hat sich als tragfähig und außerordentlich effektiv erwiesen.

Das Ergebnis der Testläufe unterstützt die Hypothese, der Algorithmus und seine Implementierung berechneten in den Grenzen der gegebenen Constraintregeln alle notwendigen Constraints und jede Lösung des erhaltenen Constraintnetzwerks ergebe eine korrekte Refaktorisierung. Vorbehaltlich der begrenzten Aussagekraft von Tests schließe ich, der Ansatz der Vorarbeit [36], nur die beschränkenden Constraintregeln zu ermitteln, und der hier abgeleitete Algorithmus zur Ermittlung einer Komponente sowie seine Implementierung sind richtig.

Am Beispiel der PULL-UP-FIELD-Refaktorisierung zeigt sich, dass die Anzahl benötigter Constraints meist vergleichsweise klein ist und die Rechenzeit für Erzeugung und Lösung der Constraints die Zeit unterschreitet, die inkrementelles Kompilieren des refaktorierten Programms erfordert. Die Speicherplatzanforderungen der Implementierung sind verglichen mit den Anforderungen der Entwicklungsumgebung ebenfalls moderat. Das Verfahren der selektiven Constraintinterzeugung eignet sich deshalb für die Praxis.

Im Ergebnis der Arbeit wird ein Weg offensichtlich, Constraintinterzeugung und Lösungssuche parallel durchzuführen. Das dürfte die Anzahl benötigter Constraints weiter verringern und könnte den Algorithmus beschleunigen. Eine Konsistenzprüfung während des Durchlaufs erlaubte hingegen, die Constraintinterzeugung vorzeitig abzubrechen, wenn sich die Refaktorisierung als nicht ausführbar erweist.

Eine Ergänzung von Constraintregeln erfordert keine grundsätzliche Änderung des abgeleiteten Algorithmus. Deshalb ist die Vermutung begründet, dass selektive Constraintinterzeugung auf andere als die in dieser Arbeit untersuchte Refaktorisierungen übertragbar und vom Rechenaufwand her auch praktikabel ist.

Diese Arbeit hat die in der Vorarbeit entwickelten Constraintregeln verfeinert und ergänzt, sodass nun auch geschachtelte Classifier und Mehrfachvererbung berücksichtigt werden. Außerdem gibt sie die Grenzen korrekter Refaktorisierbarkeit auf Grundlage der gewählten Constraintregeln explizit an und setzt viele dieser Grenzen in Constraints um, die eine Refaktorisierung unterbinden, wenn kein korrektes Ergebnis garantiert werden kann. Ableiten und Ergänzen fehlender Constraintregeln werden in Zukunft den Anwendungsbereich constraint-basierter Refaktorisierung erweitern.

Anhang

A Constraintregeln

Constraintzeugung setzt eine Spezifikation von Constraintregeln voraus. Dieser Anhang stellt die benötigten Constraintregeln zusammen, begründet Änderungen des Regelsatzes gegenüber der Vorarbeit [36] und gibt explizit die Grenzen korrekter Refaktorisierung auf Basis dieser Regeln an.

Die Constraintregeln sind für eine PULL-UP-FIELD-Refaktorisierung zusammengestellt. Diese Refaktorisierung nimmt neben dem eigentlichen Hochziehen eines Felds bei Bedarf weitere Änderungen am zu refaktorisierenden Programm vor. Welche das sind, definiert Abschnitt 4.1 (S. 24). Dort sind auch allgemeine Vorbedingungen aufgeführt.

Die Constraintregeln sind so formuliert, dass alle veränderlichen Eigenschaften von Programmelementen in der Konklusion der Regel stehen (vgl. Abschnitt 6.4.3, S. 49).

A.1 Notation

Seit der Veröffentlichung der Vorarbeit [36] haben Steimann und Thies die Schreibweise von Constraintregeln grundlegend überarbeitet [32, 34]. Die vorliegende Arbeit orientiert sich an der neueren Schreibweise. Folgende Absätze führen die Notation ein und definieren wesentliche Begriffe.

Trägermengen

Grundlage sind die Trägermengen. Refaktorisierungen behandeln *Programmelemente*, die im Wesentlichen der Menge der *Deklarationselemente* D und der Menge der *Referenzen* R zugeordnet sind. Zu den Deklarationselementen zählen *Classifier* $C \subseteq D$. In Java sind dies Klassen, Interfaces, Aufzählungstypen und Annotationen.¹¹

Packages P_j kommen in den Constraintregeln dieser Arbeit nur indirekt beim Ermitteln der Zugreifbarkeit von Deklarationselementen vor. Java kennt die *Zugreifbarkeiten* $A = \{\text{private}, \text{package}, \text{protected}, \text{public}\}$. Die *Arten (kinds)* vorkommender Programmelemente beschreibt die Menge $K = \{\text{field}, \text{classifier}, \text{compilation unit}, \text{local variable}\}$. Schließlich vereinfacht eine Definition der Menge der Wahrheitswerte $B = \{\text{true}, \text{false}\}$ die Schreibweise der Constraintregeln und darin benutzter Relationen.

Eigenschaften

Programmelemente sind durch *Eigenschaften (properties)* charakterisiert. Eigenschaften werden mit griechischen Buchstaben bezeichnet und vom Programmelement, das die Eigenschaft aufweist, durch einen Punkt getrennt. Eine Refaktorisierung ändert Eigenschaften von Programmelementen. Die Identität der Programmelemente bleibt erhalten.

Nicht jede Art von Programmelement hat jede in Tabelle 6 (S. 57) angegebene Eigenschaft. *Zugreifbarkeit* α ist nur für Deklarationselemente sinnvoll, während nur Referenzen eine *Bindung* β an ihr jeweiliges Deklarationselement haben. Die Eigenschaft δ wird nur für Felder verwendet und gibt die Deklarationsanweisung an, in der das Feld deklariert ist. Alle Programmelemente haben einen *Bezeichner* ι , der im Fall von Anonymen Klassen auch eine leere Zeichenkette sein kann.

Das Konzept des *Orts (location)* ist gegenüber den Vorarbeiten etwas abgewandelt und vereinfacht. Den Ort eines Programmelements (Import-Deklarationen ausgenommen) definiere ich als den direkt einschließenden Classifier. Im Fall von Top-Level-Classifiern sei dies der Top-Level-Classifier selbst. Die Funktion „ $\text{tlc} : C \rightarrow C$ “ gibt entsprechend zu jedem Classifier

¹¹Explizites Benennen der Teilmenge $C = \{d \in D \mid d.\kappa = \text{classifier}\}$ erleichtert weitere Definitionen.

Tabelle 6 – Eigenschaften von Programmelementen

Kürzel	Bereiche	Eigenschaft
α	$D \rightarrow A$	Zugreifbarkeit (<i>accessibility</i>)
β	$R \rightarrow D$	Bindung einer Referenz an das zugehörige Deklarationselement
δ	–	Deklarationsanweisung
ι	–	Bezeichner (<i>identifier</i>)
κ	$D \rightarrow K$	Art (<i>kind</i>)
λ	$D \cup R \rightarrow C$	Ort (<i>location</i>)
π	$D \rightarrow P_j$	(Java-)Package
ρ	$R \rightarrow C$	Empfängertyp (<i>receiver type</i>)

den einschließenden Top-Level-Classifier zurück. Der Ort von Import-Deklarationen ist ein *Java-File* (*compilation unit*). Die gleichnamige Funktion „*compilation-unit*“ liefert für Classifier und für Import-Deklarationen den Java-File, in dem sie enthalten sind.

Die Eigenschaft π eines Classifiers gibt das Package an, in dem er enthalten ist. Sie wird nicht direkt manipuliert, da die untersuchte Refaktorisierung Classifier nicht verlagert. Sie ist nur für die Ermittlung der Zugreifbarkeit nötig. In welchem Package Felddeklarationen und Referenzen enthalten sind, ist indirekt über ihren Ort bestimmbar.

Empfängertyp

Nicht-statische und statische Feldzugriffe erfolgen in Java explizit qualifiziert oder implizit *this*-qualifiziert – im letzteren Fall allein über den Bezeichner des Felds [12, § 6.5.6.1, S. 134]. Gebunden wird die Referenz an den statischen Typ des Empfängerausdrucks [12, § 15.11.1, S. 436]. Für die Zwecke dieser Arbeit ist es ausreichend, den Typ mit dem Classifier gleichzusetzen.

Der *Empfängertyp* $r.\rho$ einer Feldreferenz r ist bei expliziter Qualifizierung der Typ des qualifizierenden Ausdrucks.

Im Fall einer implizit *this*-qualifizierten Referenz ist der Empfängertyp der Typ, der das referenzierte Feld deklariert oder erbt [12, § 13.1, S. 336]. Es ist also genau der Typ, den man explizit mit einem qualifizierten *this* (siehe [12, § 15.8.4, S. 422]) angeben würde.¹² In inneren Klassen kann der Empfängertyp einer implizit *this*-qualifizierten Referenz vom Ort der Referenz verschieden sein. Ein Beispiel:

```

1  class A {
2
3      class B extends A {
4          private int i; // d_1
5          int j= i;      // r_1
6      }
7  }

```

```

class A {
    private int i; // d_2
    class B extends A {
        int j= i; // r_2
    }
}

```

Der Ort der Deklaration des Felds i in der links dargestellten Konstellation ist die innere Klasse B: $d_1.\lambda = B$. Das Empfängerobjekt der Referenz r_1 auf das Feld i in der Klasse B ist eine Instanz der Klasse B. Damit ist der Empfängertyp $r_1.\rho = B$.

Zieht man das Feld i von B in die einschließende Klasse A hoch, erhält man die rechts dargestellte Konstellation mit $d_2.\lambda = A$. Die innere Klasse B erbt das Feld $d_2 = i$ nicht, da es *private* deklariert ist. Das im Initialisierungsausdruck referenzierte Objekt ist nun die Instanz

¹²Ein qualifiziertes *this*, das mit dem Namen des direkt einschließenden Classifiers qualifiziert ist, ist äquivalent zum nicht-qualifizierten *this* am gleichen Ort.

der einschließenden Klasse A. Der Empfängertyp $r_2.\rho$ der Referenz r_2 ist die Klasse A; der Ort $r_2.\lambda$ der Referenz r_2 ist unverändert die Klasse B. Also gilt hier $r_2.\rho = A \neq r_2.\lambda = B$.

Erhöht man die Zugreifbarkeit des Felds i in der rechts dargestellten Konstellation auf *package*, dann erbt die innere Klasse das Feld i und der Empfängertyp ändert sich wieder zur inneren Klasse B.

Das Beispiel zeigt, der Empfängertyp $r.\rho$ ist im Fall implizit *this*-qualifizierter Referenzen eine Funktion des Orts $r.\lambda$, an dem sich die Referenz r befindet. In geschachtelten Classifiern ist der Empfängertyp eine Funktion des Orts $r.\lambda$ und zusätzlich des Orts $d.\lambda$ der referenzierten Deklaration d sowie weiterer Bedingungen, die die Vererbung der Deklaration bestimmen.

Der Zugriff auf statische Felder ist außerdem über eine Qualifizierung mit dem Namen des deklarierenden oder eines erbenden Classifiers möglich [12, § 6.5.6.2, S. 135]. Die Qualifizierung mit dem Namen des deklarierenden Classifiers ist *die* empfohlene Art, auf statische Felder zuzugreifen, die nicht mit ihrem einfachen Namen referenziert werden können. Der Empfängertyp wird hier mit dem Classifier gleichgesetzt, mit dessen Namen die Referenz qualifiziert ist.

Referenzen auf Top-Level-Classifier können ausschließlich mit dem Namen des Packages qualifiziert werden, in dem sie enthalten sind. Referenzen auf geschachtelte Classifier können mit dem Namen des deklarierenden Classifiers qualifiziert werden [12, § 6.5.5, S. 132]. Von einem *Empfängertyp* kann man in diesem Zusammenhang nicht sprechen.

Prädikate

Die Constraintregeln verwenden intuitiv lesbare Prädikate wie „ $\text{final} : D \rightarrow B$ “, „ $\text{static} : D \rightarrow B$ “, „ $\text{implicit-this} : R \rightarrow B$ “ und „ $\text{interface} : C \rightarrow B$ “. Dabei kommt es nicht darauf an, ob ein Deklarationselement mit dem entsprechenden Schlüsselwort ausgezeichnet ist, sondern ob es die Bedingung, möglicherweise implizit, erfüllt. Weitere, etwas komplexere Funktionen führt der folgende Abschnitt jeweils in ihrem Zusammenhang ein.

Relationen

Die Formulierung der Regeln benötigt verschiedene Relationen. Die einfachste ist die totale Ordnungsrelation der Zugreifbarkeiten „ $<_A \subseteq A \times A$ “, wobei *private* $<_A$ *package* $<_A$ *protected* $<_A$ *public* gelte. Wie üblich wird zur Abkürzung $\leq_A \equiv (<_A \vee =_A)$ vereinbart.

Die partielle Ordnung „ $\leq_I \subseteq C \times C$ “ bezeichnet die Sub-/Supertypbeziehung. Zur Abkürzung wird die Äquivalenz $<_I \equiv (\leq_I \wedge \neq_I)$ definiert. $A <_I B$ ist zu lesen als „Classifier A ist ein Subtyp von Classifier B“ bzw. umgekehrt „B ist ein Supertyp von A“. Die nicht-transitive Relation, die die direkte, unmittelbare Sub-/Supertypbeziehung angibt, schreibe ich mit dem Zeichen „ \prec_I “. Der Ausdruck $A \prec_I B$ bedeutet entsprechend „A ist direkter Subtyp von B“, anders ausgedrückt „A *extends* B“ bzw. „A *implements* B“. Selbstverständlich gilt $\prec_I \subseteq \leq_I$.

Schachtelung (*nesting*) von Classifiern beschreibt die Relation „ $<_N \subseteq C \times C$ “, die angibt, ob ein Classifier in einem anderen Classifier geschachtelt ist. $A <_N B$ bedeutet, dass Classifier A in Classifier B enthalten ist. Auch hier gilt $\leq_N \equiv (<_N \vee =_N)$.

Begriffe und Schreibweise

Der Begriff *Constraint* wird bereits in Abschnitt 3.1 (S. 13) als Relation auf einer Teilmenge der Constraintvariablen eingeführt. Ein Constraint ist präziser als eine Teilmenge des Kreuzprodukts der Wertebereiche seiner Constraintvariablen definiert [6, S. 25].

Constraintregeln sind Vorschriften zur Erzeugung von Constraints. Constraintregeln bestehen aus Prämisse und Konklusion. In Anlehnung an die Vorarbeit [36] trennt das Zeichen „ \Rightarrow “ die Prämisse von der Konklusion. Ist die Prämisse erfüllt, wird das in der Konklusion angegebene Constraint erzeugt. Eine logische Implikation wird als einfacher Pfeil „ \rightarrow “ geschrieben, um sie vom Trennzeichen einer Constraintregel zu unterscheiden.

Bezeichner von Programmelementen werden im Text wie Programmauszüge in einer nicht-proportionalen Schriftart gesetzt.

A.2 Zugreifen

Unverändert aus der Vorarbeit [36] übernommen ist die Constraintregel *Acc-1*, die sicherstellt, dass ein referenziertes Programmelement vom Ort seiner Referenz aus zugreifbar bleibt:

$$r.\beta = d \wedge d.\kappa \in \{field, classifier\} \Rightarrow d.\alpha \geq_A \alpha(r.\lambda, d.\lambda) \quad (\text{Acc-1})$$

Die in *Acc-1* und den folgenden Regeln verwendete Funktion „ $\alpha : C \times C \rightarrow A$ “ berücksichtigt nun auch den Fall geschachtelter Classifier. Sie ist folgendermaßen definiert:

$$\begin{aligned} \alpha(r.\lambda, d.\lambda) \equiv & \text{if } \text{tlc}(r.\lambda) = \text{tlc}(d.\lambda) \text{ then } \textit{private} \\ & \text{else if } (r.\lambda).\pi = (d.\lambda).\pi \text{ then } \textit{package} \\ & \text{else if } \exists c \in C \mid c \geq_N r.\lambda \wedge c <_I d.\lambda \text{ then } \textit{protected} \\ & \text{else } \textit{public} \end{aligned}$$

Darin liefert die Hilfsfunktion „ $\text{tlc} : C \rightarrow C$ “ den Top-Level-Classifier eines Classifiers und die Eigenschaft π das den Classifier enthaltende Package.

Die Funktion α ist als *Makro* zu verstehen, das in den Constraintregeln jeweils expandiert gedacht ist. Mit anderen Worten, α wird nicht nur für das Programm vor der Refaktorisierung berechnet, sondern α wird für jede betrachtete Belegung der Eigenschaften vorkommender Programmelemente neu ausgewertet.

Damit auf ein Feld zugegriffen werden kann, muss es nicht nur selbst mit wenigstens der erforderlichen Zugreifbarkeit ausgezeichnet sein, sondern auch der Typ des Ausdrucks, über den auf das Feld zugegriffen wird, also der Empfängertyp, muss zugreifbar sein [12, § 6.6.1, S. 139]. Bei (implizit) *this*- oder *super*-qualifizierten Zugriffen ist diese Bedingung automatisch erfüllt. Für die übrigen Feldreferenzen erzeugt die folgende Ergänzung der Regel *Acc-1* ein Constraint für die implizite Referenz auf den Empfängertyp:

$$\begin{aligned} r.\beta = d \wedge d.\kappa = field \wedge c = r.\rho \wedge \neg \text{this-expression}(r) \wedge \neg \text{super-access}(r) \Rightarrow \\ c.\alpha \geq_A \alpha(r.\lambda, c.\lambda) \end{aligned} \quad (\text{Acc-1}^*)$$

Folgendes Code-Beispiel erläutert die Ergänzung der Regel. Der Initialisierungsausdruck des Felds *j* in der Klasse *B* enthält eine implizite Referenz auf die innere Klasse *X*: Der Initialisierungsausdruck referenziert das Feld *i* über das Feld *x*. Der Typ von *x*, der Empfängertyp der Referenz auf *i*, ist die Klasse *X*.

```

8   class A {
9       class X { int i; }
10      X x= new X();
11  }
12  class B extends A {
13      int j= x.i;
14  }
```

Die Anwendung des ersten Teils der Regel *Acc-1* erzeugt die Constraints, die dafür sorgen, dass die Felder *i* und *x* wenigstens die Zugreifbarkeit *package* erhalten, damit sie von Klasse *B* aus zugreifbar sind. Erst der zweite Teil der Regel, *Acc-1**, erzeugt das Constraint, das für die Klasse *X* ebenfalls wenigstens *package*-Zugreifbarkeit durchsetzt.

Java erlaubt den Zugriff auf als *protected* deklarierte Instanzvariablen über qualifizierende Ausdrücke nur dem Code, der für die Implementierung des Typs des qualifizierenden

Ausdrucks zuständig ist [12, § 6.6.2.1, S. 139, und § 6.6.7, S. 143 f.]. Das sind neben der das Feld deklarierenden Klasse diejenigen Subklassen, die außerdem vom Typ oder ein Supertyp des qualifizierenden Ausdrucks sind. Bemerkenswerterweise gehören dazu außerdem alle von diesen Klassen eingeschlossenen Classifier, unabhängig davon, ob sie Teil der Vererbungshierarchie sind oder nicht. Regel *Acc-2* drückt die Beschränkung dieses privilegierten Zugriffs aus.

$$r.\beta = d \wedge d.\kappa = \text{field} \wedge \neg \text{static}(d) \wedge \neg \text{this-expression}(r) \Rightarrow \quad (\text{Acc-2})$$

$$\alpha(r.\lambda, d.\lambda) =_A \text{protected} \wedge (\nexists c \in C \mid c \geq_N r.\lambda \wedge c \geq_I r.\rho) \rightarrow d.\alpha =_A \text{public}$$

Das Prädikat „this-expression : $R \rightarrow B$ “ prüft, ob es sich bei der Referenz um einen (implizit) *this*-qualifizierten Ausdruck handelt. Die Klausel ist zu der Prämisse der Regel ergänzt, damit für die häufigen *this*-Referenzen das Constraint gar nicht erst erzeugt wird. Es würde ohnehin nicht greifen, da der Nicht-Existenz-Ausdruck der Bedingung in der Konklusion der Regel für *this*-Referenzen nie erfüllt ist. Der Ausdruck ist für *this*-Referenzen immer falsch, weil eine *this*-Referenz gerade die Existenz eines einschließenden Classifiers voraussetzt, der das referenzierte Feld deklariert oder erbt. Ohne Berücksichtigung von Schachtelung vereinfachte sich dieser Ausdruck zu $r.\lambda \not\geq_I r.\rho$.

Folgendes, auf Kahl [18, S. 16] zurückgehende Beispiel zeigt, dass alle durch die Refaktorisierung änderbaren Eigenschaften in der Konklusion der Regel stehen müssen. Beide Klauseln der Bedingung der Implikation können ihren Wert ändern.

15	package p;	package p;
16	public class A {	public class A {
17	}	public int i;
18	}	}
19		
20	package q;	package q;
21	class B extends p.A {	class B extends p.A {
22	int i;	}
23	}	class C extends B {
24	class C extends B {	void m(B b){ b.i++; }
25	void m(B b){ b.i++; }	}
26	}	}

Im Code vor der Refaktorisierung (links) sind die Bedingungen der Prämisse erfüllt. Damit generiert die Regel das Constraint. Es verlangt für die Situation nach dem Hochziehen des Felds *i* von B nach A (rechts), das Feld *i* nun *public* zu deklarieren: Der Typ des Empfänger-Ausdrucks $r.\rho$ ist B, also nicht Subtyp des Orts der Referenz $r.\lambda = C$. Die Funktion α wertet nach der Refaktorisierung zu *protected* aus. Die Bedingungen der Implikation sind somit erfüllt, und es folgt $d.\alpha = \text{public}$. An diesem Beispiel lässt sich weiter verifizieren, dass auch der zweite Term der Konklusion seinen Wert in Folge einer Refaktorisierung ändern kann. Führt man die Refaktorisierung weiter und zieht nun zusätzlich die Methode *m* von C nach B hoch, dann wird $r.\lambda =_I r.\rho$. Der Term wertet zu „falsch“ aus, und damit reicht *protected* als Zugreifbarkeit für das Feld *i* in Klasse A.

A.3 Erben

Zugriff auf geerbte Felder beschreibt die folgende Constraintregel *Inh-1*, die in zweierlei Hinsicht gegenüber [36] erweitert ist. Die ergänzte Klausel $r.\rho <_I d.\lambda$ repräsentiert ein Typconstraint aus der Arbeit von Tip et al. [40]. Es stellt sicher, dass für den Empfängertyp $r.\rho$ das Feld *d* überhaupt verfügbar ist und bleibt [12, § 8.2, S. 190].

Die zusätzliche Klausel $d.\alpha \geq_A \max(\alpha(c_i, d.\lambda))$ sorgt dafür, dass das Feld tatsächlich in der Kette von Supertyp zu Subtyp durchgehend vererbbar ist. Nur die Member der direkten

Supertypen werden geerbt und auch nur dann, wenn sie zugreifbar und nicht privat sind [12, § 6.4.3, S. 123, bzw. § 8.4.8, S. 224] – Verbergen behandelt eine separate Constraintregel.

$$\begin{aligned} r.\beta = d \wedge d.\kappa = \text{field} \wedge c_i \in C \Rightarrow r.\rho \neq_I d.\lambda \rightarrow & \quad (\text{Inh-1}) \\ r.\rho <_I d.\lambda \wedge r.\rho \leq_I c_i <_I d.\lambda \wedge d.\alpha >_A \text{private} \wedge d.\alpha \geq_A \max(\alpha(c_i, d.\lambda)) & \end{aligned}$$

Da der Term $\max(\alpha(c_i, d.\lambda))$ im Fall von geschachtelten Classifiern zu *private* auswerten kann, ist die zusätzliche Schranke $d.\alpha >_A \text{private}$ erforderlich. Mehrfachvererbung muss hier nicht gesondert behandelt werden, da in Interfaces deklarierte Felder ohnehin *public* sind. Folgendes Beispiel illustriert die Regel. Das Feld *i* sei aus Klasse *C*, links, in Klasse *A* hochgezogen, rechts. Obwohl die Referenz auf *i* in der Initialisierung des Felds *j* aus dem gleichen Package erfolgt, muss nach der Refaktorisierung *i* *protected* deklariert werden, damit es aus *C* zugreifbar bleibt.

27	package p;		package p;
28	public class A {		public class A {
29			protected int i= 0;
30	}		}
31	class C extends q.B {		class C extends q.B {
32	int i= 0;		
33	int j= i;		int j= i;
34	}		}
35			
36	package q;		package q;
37	public class B extends p.A { }		public class B extends p.A { }

Zöge man in diesem Beispiel zuerst das Feld *j* hoch, dann erzwänge die erste Klausel der Konklusion, dass das Feld *i* folgen müsste und ebenfalls hochgezogen würde.

Mehrfachvererbung macht die Sache kompliziert und – wie sich herausstellt, durch Kombination von Vererbung und Verbergen – komplizierter als zunächst angenommen. Mehrdeutigkeit von Referenzen wird in Java dadurch vermieden, dass verschiedene von Supertypen geerbte Member gleichen Namens nicht referenziert werden dürfen [12, § 8.3.3.3, S. 207, und § 15.11.1, S. 435]. Mangelnde Zugreifbarkeit (*Inh-1*) und Verbergen (*Hid-1*) verhindern Vererbung; allerdings müssen alle Wege, auf denen geerbt werden kann, versperrt sein.

Ich führe zunächst ein zweistelliges Prädikat „inherits : $C \times D \rightarrow B$ “ ein, das angibt, ob der Empfängertyp $r.\rho$ einer Referenz r das Feld d erbt: Sei $G = (V, E)$ ein gerichteter Graph auf einer Teilmenge der Classifier $V \subseteq C$ mit $(v_1, v_2) \in E(G) \Leftrightarrow v_1 \prec_I v_2$.¹³ Seien weiterhin

$$\begin{aligned} d, d_h \in D \text{ mit } d \neq d_h \wedge d.\kappa = d_h.\kappa = \text{field} \wedge d.\iota = d_h.\iota \\ \text{sowie } H := \{v \in V \mid v = d_h.\lambda \vee d.\alpha <_A \alpha(v, d.\lambda)\}, \end{aligned}$$

dann ist „inherits“ wie folgt definiert:

$$\begin{aligned} \text{inherits}(r.\rho, d) \equiv_{\text{def}} d.\alpha >_A \text{private} \wedge \text{es gibt einen Weg } P = v_0 \dots v_k \text{ in } G \\ \text{mit } k \in \mathbb{N} \wedge v_0 = r.\rho \wedge v_k = d.\lambda \text{ sodass } V(P) \cap H = \emptyset. \end{aligned}$$

Das heißt, ein Empfängertyp $r.\rho$ erbt das Feld d genau dann, wenn es nicht *private* deklariert ist und wenn es einen Weg im Vererbungsgraphen vom Empfängertypen bis zum deklarierenden Typen gibt, der nicht durch ein gleichnamiges Feld d_h blockiert ist, und von dessen Ecken aus das Feld zugreifbar ist. Dieses Prädikat „inherits“ ist, wie bereits die Funktion α , als Makro zu verstehen. Es muss für jede betrachtete Belegung der Eigenschaften von Programmelementen ausgewertet werden.

¹³ Wie üblich bezeichnet $V(G)$ die Eckenmenge und $E(G)$ die Kantenmenge eines Graphen G .

Mithilfe dieses Prädikats lässt sich die Constraintregel, die Mehrdeutigkeit von Feldreferenzen verhindert, nun recht einfach schreiben. Das zweite Feld darf vom Empfängertyp nicht geerbt werden.

$$r.\beta = d_1 \wedge d_1.\kappa = d_2.\kappa = \text{field} \wedge d_1.\iota = d_2.\iota \wedge d_1 \neq d_2 \wedge (\text{static}(d_1) \vee \text{static}(d_2)) \Rightarrow \neg \text{inherits}(r.\rho, d_2) \quad (\text{Inh-2})$$

Wenigstens eins der beiden Felder d_1 und d_2 muss statisch sein, da höchstens ein nicht-statisches Feld gleichen Namens geerbt werden kann. Ohne weiteres können aber beide Felder statisch sein. So lange keine mehrdeutige Referenz existiert, ist das auch kein Fehler.

Ein Code-Beispiel von Heinrich-Kerber [13] illustriert das Zusammenwirken von Mehrdeutigkeit und Verbergen im Zusammenhang von Mehrdeutigkeit einer Feldreferenz.

38	class A {		class A {
39	}		int i;
40	interface I { int i= 0; }		}
41	class B extends A implements I {		interface I { int i= 0; }
42	int i;		class B extends A implements I {
43	void m(){ i++; }		void m() { i++; } //error
44	}		}
45			

Das Hochziehen der Instanzvariablen i aus B (links) nach A (rechts) führt dazu, dass das statische Feld $I.i$ nicht länger verbergen bleibt. Die Referenz in Methode m wird damit mehrdeutig und verursacht einen Übersetzungsfehler. Die Constraintregel *Inh-2* würde diese Refaktorisierung unterbinden.

A.4 Verbergen

Der vorhergehende Abschnitt hat das Phänomen des *Verbergens* schon angeschnitten. Ein geerbtes Member kann durch eine neue Deklaration der gleichen Art mit gleichem Namen *verbergen* werden (*hiding* [12, § 8.3, S. 196]). Dabei kommt es nicht darauf an, ob die neue Deklaration vom betrachteten Ort aus überhaupt zugreifbar ist; sie kann auch privat deklariert sein. Die hier angegebene Formalisierung der Constraintregel *Hid-1* verzichtet auf die Möglichkeit, eine Deklaration als *absent* auszuzeichnen. Eine PULL-UP-FIELD-Refaktorisierung soll kein Feld verschwinden lassen, selbst wenn es nicht referenziert ist. Die Regel verlangt stattdessen die Abwesenheit einer verbergenden Deklaration. Außerdem berücksichtigt die Regel nun, dass es wegen Mehrfachvererbung mehrere Vererbungswege geben könnte. Die betrachtete Refaktorisierung kann nicht zum Verbergen von Classifiern führen, da sie Classifierdeklarationen nicht verschiebt. Die Regel ist hier deshalb auf Felder beschränkt. Sei $G = (V, E)$ ein gerichteter Graph mit $V \subseteq C$ und $(v_1, v_2) \in E(G) \Leftrightarrow v_1 \prec_I v_2$, dann gilt die Constraintregel:

$$r.\beta = d \wedge d \neq d_h \wedge d.\kappa = d_h.\kappa = \text{field} \wedge d.\iota = d_h.\iota \Rightarrow \begin{array}{l} \text{es gibt einen Weg } P = v_0 \dots v_k \text{ in } G \text{ mit } k \in \mathbb{N} \wedge v_0 = r.\rho \wedge v_k = d.\lambda \\ \text{sodass } \forall v \in V(P) : v \neq d_h.\lambda. \end{array} \quad (\text{Hid-1})$$

Das heißt, es gibt im Vererbungsgraphen G einen Weg vom Empfängertypen $r.\rho$ zum deklarierenden Typen $d.\lambda$ des Felds d , der nicht durch ein gleichnamiges Feld d_h blockiert ist.

Bild 9 (S. 63) zeigt links eine Situation, in der die Klasse A das Interface I implementiert und das Feld $d_h = i$ *private* deklariert. Die Subklasse B von A deklariert ein gleichnamiges Feld $d = B.i$, das im Initialisierungsausdruck des Felds j referenziert ist (r). Die Constraintregel *Hid-1* verhindert ein Hochziehen des Felds $d = B.i$ von B in das Interface I ($d.\lambda = I$) auf

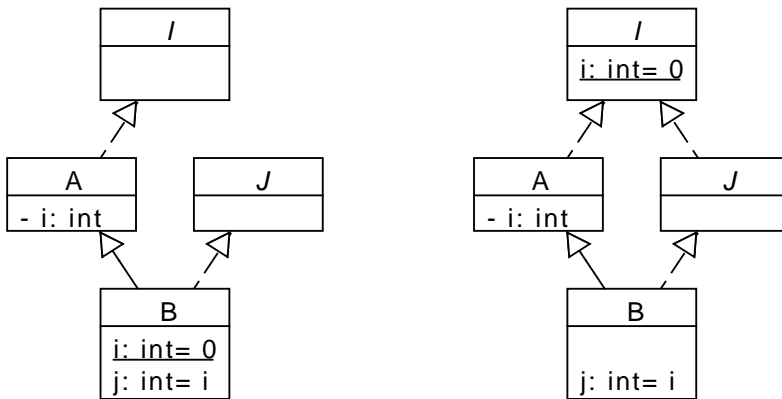


Bild 9 – Verbergen und Mehrfachvererbung

folgende Weise: Die Prämisse der Regel ist offenbar erfüllt. Das erzeugte Constraint verlangt, es gebe einen Weg von $v_0 = B$ nach $v_k = I$, der nicht über die Ecke A führt. So ein Weg existiert offensichtlich nicht. Das Constraint ist somit nicht erfüllbar.

Die rechte Seite von Bild 9 stellt eine Anordnung dar, in der ein Hochziehen des Felds B.i aus Klasse B in das Interface I möglich ist. Für die rechte Seite des Bilds existiert folgender Weg P im Vererbungsgraphen: $P = (v_0 v_1 v_2)$ mit $v_0 = B \wedge v_1 = J \wedge v_2 = I$ und an allen Ecken v gilt $v \neq d_h.\lambda = A$. Somit ist das Constraint erfüllt.

A.5 Abschatten

Nicht-qualifizierte Referenzen

Implizit *this*-qualifizierte Referenzen bzw. nicht-qualifizierte Referenzen mit einfachem Namen (*simple name*), die in geschachtelten (*nested*) Classifiern vorkommen, können durch Deklarationen von Mitgliedern im geschachtelten Classifier selbst oder in einem seiner Supertypen *abgeschattet* werden (*shadowing* [12, § 8.3, S. 196, § 6.3.1, S. 119]) – Fehlermeldungen der Compiler unterscheiden nicht unbedingt zwischen *hiding* und *shadowing*. Abschatten ist anders als Verbergen jedoch abhängig von Ort und Zugriffsmodifikator der abschattenden Deklaration. Die folgende Constraintregel *Hid-2* unterbindet das Abschatten eines referenzierten Felds durch eine PULL-UP-FIELD-Refaktorisierung.

$$\begin{aligned}
 & r.\beta = d_1 \wedge \text{implicit-this}(r) \wedge d_1.\kappa \in \{\text{field}, \text{local variable}\} \wedge & \text{(Hid-2)} \\
 & d_2.\kappa = \text{field} \wedge d_1.\iota = d_2.\iota \wedge d_1 \neq d_2 \wedge c \in C \Rightarrow \\
 & r.\rho >_N c \geq_N r.\lambda \rightarrow c \neq d_2.\lambda \wedge \neg \text{inherits}(c, d_2)
 \end{aligned}$$

Ein weiteres Beispiel aus Heinrich-Kerbers Sammlung [13] soll die Constraintregel begründen.

```

46  class A {
47      int i= 0;
48      class B extends S {
49          int m(){ return i; }
50      }
51  }
52
53  class S { }
54  class T extends S {
55      int i= 1;
56  }

```

Die Referenz r in der inneren Klasse B auf das Feld $d_1 = i$ in der einschließenden Klasse A ist implizit *this*-qualifiziert. Es gibt ein zweites Feld $d_2 = i$ mit gleichem Namen in Klasse T, das in die gemeinsame Superklasse S hochgezogen werden soll. Die Prämisse der Constraintregel *Hid-2* ist also erfüllt, und entsprechend wird das Constraint generiert. Es besagt, wenn es einen die Referenz einschließenden Classifier c zwischen Empfänger der Referenz $r.\rho = A$ und Ort der Referenz $r.\lambda = B$ gibt, dann darf dieser Classifier c kein Feld d_2 mit gleichem Namen deklarieren oder erben. Zöge man im gegebenen Beispiel das Feld $d_2 = i$ aus der Klasse T in die Klasse S hoch, würde das Constraint verletzt, da Klasse B dann das Feld i von S erbt. Die Methode m gäbe nach dem Hochziehen 1 statt 0 zurück.

Statische Import-Deklarationen

Eine ähnliche Art von Abschattung betrifft statische Import-Deklarationen. Jede direkte oder geerbte Deklaration eines neuen Felds schattet eine gleichnamige statische Import-Deklaration im Java-File ab. Das ist aus der Sprachspezifikation [12, § 6.3.1, S. 119] nur schwer herauszulesen, lässt sich aber an einem Java-Compiler nachvollziehen. Folgende neue Constraintregel unterbindet eine Änderung der Bindung von Feldreferenzen durch Abschattung eines statischen Imports, solange statische Import-Deklarationen selbst nicht verlagert werden.

$$\begin{aligned} r.\beta = d_1 \wedge \text{implicit-this}(r) \wedge \text{static}(d_1) \wedge r_i.\beta = d_1 \wedge \text{static-import}(r_i) \wedge & \quad (\text{Hid-3}) \\ d_1.\kappa = d_2.\kappa = \text{field} \wedge d_1.\iota = d_2.\iota \wedge d_1 \neq d_2 \wedge r_i.\lambda = \text{compilation-unit}(r.\lambda) \\ \wedge c \in C \Rightarrow \text{tlc}(r.\lambda) \geq_N c \geq_N r.\lambda \wedge c \neq d_2.\lambda \wedge \neg \text{inherits}(c, d_2) \end{aligned}$$

Das Constraint gilt für statisch importierte, über den statischen Import referenzierte Felder, wenn es ein gleichnamiges Feld gibt. Es besagt, dass eine die Referenz einschließende Klasse das Feld auch nach der Refaktorisierung nicht deklarieren oder erben darf. Ein etwas perfides Beispiel soll helfen, die Constraintregel zu erklären.

```

57 package p;
58 public class A { int i= 1; }
59
60 package q;
61 import static q.I.i;
62 class B extends p.A {
63     class C {
64         int m(){ return i; }
65     }
66 }
67
68 package q;
69 interface I { int i= 0; }

```

Ein Erhöhen der Zugreifbarkeit des Felds i in Klasse A von *package* auf *protected* führt dazu, dass die Methode m in der zunächst ganz unbeteiligt erscheinenden Klasse C nun 1 statt 0 zurückgibt. Wie verhindert die Constraintregel *Hid-3* diese ungewollte Änderung?

Die implizit *this*-qualifizierte Referenz r in Methode m der inneren Klasse C ist über die statische Import-Deklaration r_i an die Deklaration des Felds $d_1 = I.i$ in Interface I gebunden. Der Java-File der Import-Deklaration ist gleich dem Java-File, der die Referenz enthält ($r_i.\lambda = \text{compilation-unit}(r.\lambda)$). Nun gibt es ein zweites, gleichnamiges Feld $d_2 = i$ in der Klasse A. Damit ist die Prämisse der Regel erfüllt, und das Constraint wird erzeugt. Es schreibt vor, dass d_2 nicht von der einschließenden Klasse B geerbt werden darf, die Zugreifbarkeit also nicht größer oder gleich *protected* sein darf.

Geschachtelte Classifier

Ein analoges Problem kann die Änderung der Zugriffsmodifikatoren geschachtelter Classifier verursachen. Ein durch die Refaktorisierung vererbbar gewordener Classifier kann einen importierten Classifier oder andere im Geltungsbereich deklarierte Classifier abschatten. In Ermangelung einer geeigneteren Constraintregel (siehe Abschnitt A.9, S. 68) verhindert vorläufig Regel *Hid-4* auf recht grobe Art neues Abschatten bestehender Import-Deklarationen oder anderer bestehender Bindungen.

$$\begin{aligned}
 d_1.\kappa = d_2.\kappa = \textit{classifier} \wedge d_1 \neq \textit{tlc}(d_1) \wedge d_1.\alpha <_A \textit{public} \wedge & \quad (\text{Hid-4}) \\
 d_2 <_I d_1.\lambda \wedge r.\lambda \leq_N d_2 \wedge r.\iota = d_1.\iota \wedge d_1.\alpha <_A \alpha(r.\lambda, d_1) \Rightarrow & \\
 d_1.\alpha <_A \alpha(r.\lambda, d_1) &
 \end{aligned}$$

Es sei angemerkt: Die Klausel $d_1.\alpha <_A \alpha(r.\lambda, d_1)$ impliziert $r.\beta \neq d_1$. Für alle geschachtelten Classifier, die nicht ohnehin schon *public* deklariert sind und die Subklassen haben, in denen andere gleichnamige Classifier referenziert werden, verhindert *Hid-4*, dass der Zugriffsmodifikator des Classifiers über den bereits vorhandenen hinaus so erhöht wird, dass er vom Ort der Referenz aus zugreifbar würde. Da verlagerte Classifier-Referenzen als voll qualifiziert (angenommen) sind, kann *Hid-4* sie ignorieren. Folgendes Beispiel begründet den Bedarf einer derartigen Regel.

```

70 package p;
71 public class X {
72     public static int m(){ return 0; }
73 }
74
75 package q;
76 import p.X;
77 class B extends A {
78     int n(){ return X.m(); }
79 }
80
81 package q;
82 class A {
83     private static class X {
84         static int m(){ return 1; }
85     }
86 }

```

Die Methode *n* in Klasse *B* gibt in dem gegebenen Programm den Wert 0 zurück. Erhöht man in Klasse *A* jedoch den Zugriffsmodifikator der geschachtelten Klasse *B.X* von *private* auf *package*, dann schattet der Classifier *B.X* den Import der Klasse *p.X* ab, und *n* gibt nun den Wert 1 zurück (der Kürze halber zeigt das Beispiel nicht, wie diese Aufweitung durch eine PULL-UP-FIELD-Refaktorisierung ausgelöst werden kann). *Hid-4* verhindert diese unzulässige Änderung. Die Funktion $\alpha(B, A.X)$ wertet zu *package* aus. Das Constraint verlangt, dass der Zugriffsmodifikator von *A.X* kleiner bleibt als dieser Wert.

A.6 Verschiedenes

Aus der Vorarbeit [36] sind folgende mit dem Kürzel *Misc* gekennzeichnete Constraintregeln übernommen, die ganz unterschiedliche Aspekte der Java-Spezifikation behandeln. Die Nummerierung entspricht der Reihenfolge in der Vorlage.

Mehrere Felder können in einer gemeinsamen Anweisung deklariert werden, die durch die Refaktorisierung nicht in individuelle Deklaration zerlegt werden. Die Zugreifbarkeit der

Felder, die in der Deklarationsanweisung δ gemeinsam deklariert sind, ist für alle Felder identisch [12, § 8.3, S. 196]. Das Gleiche gilt auch für ihren Ort vor und nach der Refaktorisierung. Die Regel *Misc-2* fasst beide Bedingungen zusammen:

$$d_1.\delta = d_2.\delta \wedge d_1.\kappa = d_2.\kappa = \text{field} \Rightarrow d_1.\alpha =_A d_2.\alpha \wedge d_1.\lambda = d_2.\lambda \quad (\text{Misc-2})$$

Zulässige Zugriffsmodifikatoren von Top-Level-Classifiern sind auf *package* oder *public* beschränkt [12, § 8.1.1, S. 175]. Die Constraintregel *Misc-4* gewährleistet, dass diese Bedingung erhalten bleibt. Sie sorgt auch dafür, dass nur ein Top-Level-Classifier öffentlich sein darf, dessen Name mit dem Namen des Java-Files (*compilation unit*) übereinstimmt [12, § 7.6, S. 167].

$$d.\kappa = \text{classifier} \wedge \text{tlc}(d) = d \Rightarrow d.\alpha \in \{\text{package}, \text{public}\} \quad (\text{Misc-4})$$

$$d_1.\kappa = \text{classifier} \wedge d_2.\kappa = \text{compilation unit} \wedge d_1.\iota \neq d_2.\iota \wedge \text{compilation-unit}(d_1) = d_2 \wedge \text{tlc}(d_1) = d_1 \Rightarrow d_1.\alpha =_A \text{package}$$

Einzeln, statisch importierte Felder müssen am Ort ihres Imports, also für einen Java-File, zugreifbar bleiben. *Misc-5* setzt diese Anforderung der Sprachspezifikation [12, § 7.5.3, S. 164] auch für importierte (statische) Felder durch, die ansonsten im Java-File gar nicht referenziert werden. Einzeln importierte Classifier deckt die Constraintregel *Suppl-2* ab. Genutzte Importe bleiben wegen *Acc-1* ohnehin zugreifbar.

$$r_i.\beta = d \wedge \text{static-import}(r_i) \wedge d.\kappa = \text{field} \Rightarrow \quad (\text{Misc-5})$$

$$r_i.\pi = (d.\lambda).\pi \wedge d.\alpha \geq_A \text{package} \vee d.\alpha =_A \text{public}$$

Besondere Anforderungen müssen Member von Interfaces erfüllen. Die Constraintregel *Misc-8* setzt durch, dass Felder, die in Interfaces hochgezogen werden, (implizit) *public*, *static* und *final* sind [12, § 9.3, S. 264]. Damit sie hochgezogen werden können, müssen sie bereits vor der Refaktorisierung *static* und *final* deklariert sein, da eine Änderung dieser Eigenschaften durch die Refaktorisierung nicht vorgesehen ist.

$$d.\kappa = \text{field} \Rightarrow \text{interface}(d.\lambda) \rightarrow d.\alpha =_A \text{public} \quad (\text{Misc-8})$$

$$d.\kappa = \text{field} \wedge \neg(\text{static}(d) \wedge \text{final}(d)) \Rightarrow \neg \text{interface}(d.\lambda)$$

A.7 Ergänztes

Es folgen eine Reihe einfacher Regeln, die eher technischer Natur sind. Constraintregel *Suppl-0* verhindert, dass zwei Member gleichen Namens und gleicher Art an denselben Ort gelangen [12, § 8.3, S. 196].

$$d_1.\iota = d_2.\iota \wedge d_1.\kappa = d_2.\kappa \Rightarrow d_1.\lambda \neq d_2.\lambda \quad (\text{Suppl-0})$$

Die Regel *Suppl-1* hält solche Felddeklarationen mit Referenzen r an einem Ort zusammen, die in der Deklaration selbst, in einer Annotation oder in den Initialisierungsausdrücken der Deklarationsanweisung δ vorkommen.

$$r.\delta = d.\delta \Rightarrow r.\lambda = d.\lambda \quad (\text{Suppl-1})$$

Für die untersuchte Refaktorisierung gab es keine Regel, die eine Verminderung der Zugreifbarkeit von Classifiern auslöst. Die Regel *Suppl-2* setzt das erwartete Verhalten der Refaktorisierung durch, Zugriffsmodifikatoren von Classifiern nicht zu vermindern. *Suppl-2* ersetzt für Classifier die in der Vorarbeit [36] unter *Verschiedenes* aufgeführte dritte und siebte Regel, die verlangen, dass alle Classifier mit *main*-Methoden sowie mit *@API*, *@Test* oder ähnlich

annotierte Deklarationen öffentlich zugreifbar bleiben, wenn sie das vor der Refaktorisierung bereits waren.

$$d.\kappa = \text{classifier} \Rightarrow d.\alpha \geq_A \text{old}(d.\alpha) \quad (\text{Suppl-2})$$

Die Funktion „old : $A \rightarrow A$ “ liefert hier den Zugriffsmodifikator, mit dem die Deklaration d vor der Refaktorisierung ausgezeichnet ist.

Nicht-statische, geschachtelte Klassen bezeichnet die Java-Sprachspezifikation als *innere Klassen*. Sie dürfen keine statischen Member deklarieren, die nicht zur Compile-Zeit auswertbare Konstanten sind [12, § 8.1.3, S. 181]. Constraintregel *Suppl-3* verhindert, etwas vereinfachend, generell das Hochziehen statischer Felder in innere Klassen, weil nicht ohne Weiteres zu ermitteln ist, ob der Initialisierungsausdruck zur Compile-Zeit auswertbar ist.

$$d.\kappa = \text{field} \wedge \text{static}(d) \Rightarrow \neg \text{static}(d.\lambda) \rightarrow d.\lambda = \text{tlc}(d.\lambda) \quad (\text{Suppl-3})$$

Werden nicht-transiente Felder, die in serialisierbaren Klassen deklariert sind, in nicht-serialisierbare Klassen hochgezogen, werden Objekte nach der Refaktorisierung anders serialisiert als davor [12, § 8.3.1.3, S. 199]. Constraintregel *Suppl-4* verhindert solch eine Verhaltensänderung.

$$\begin{aligned} d.\kappa = \text{field} \wedge \neg \text{transient}(d) \wedge d.\lambda <_I \text{java.io.Serializable} \Rightarrow \\ d.\lambda <_I \text{java.io.Serializable} \end{aligned} \quad (\text{Suppl-4})$$

Es bleiben weitere technische Gründe, aus denen ein Feld mit der hier untersuchten PULL-UP-FIELD-Refaktorisierung nicht verlagert werden kann (siehe Abschnitt A.9, S. 68):

- Ein *final* deklariertes Feld, das erst im Konstruktor bzw. in einem *Initializer*-Block initialisiert wird, kann nicht hochgezogen werden, ohne dass die Initialisierung ebenfalls in die Superklasse verlagert wird [12, § 8.3.1.2, S. 199]. Ein Verlagern der Initialisierung in den Konstruktor der Superklasse bzw. in einen Initialisierungsausdruck oder -block sieht diese Arbeit jedoch nicht vor.
- Felder, deren Initialisierungsausdruck einen Methoden- oder Konstruktoraufruf enthält, können nicht hochgezogen werden, da die vorliegende Arbeit Constraintregeln für die Verlagerung von Methoden- und Konstruktoraufrufen nicht anwendet.
- Für die Behandlung ungebundener Typparameter in Felddeklarationen fehlen ebenfalls Constraintregeln. Deshalb können Felddeklarationen, die ungebundene Typparameter enthalten, nicht verschoben werden.
- Felddeklarationen, die statisch importierte Felder referenzieren, können nicht verschoben werden, da die hier untersuchte Refaktorisierung keine Verlagerung von statischen Import-Deklarationen abdeckt und auch keine kanonische Qualifizierung von Feldreferenzen vornimmt.
- Felddeklarationen, die lokale Variable referenzieren, werden hier ebenfalls von der Refaktorisierung ausgeschlossen. Ein Hochziehen wäre ohnehin nur möglich, wenn auch die Superklasse im Geltungsbereich (*scope*) der lokalen Variable läge.

In allen in dieser Aufzählung genannten Fällen gelte das Prädikat „fix-location : $D \rightarrow B$ “ als erfüllt.

$$d.\kappa = \text{field} \wedge \text{fix-location}(d) \Rightarrow d.\lambda = \text{old}(d.\lambda) \quad (\text{Suppl-5})$$

Die Constraintregel *Suppl-5* macht den Ort eines Felds unveränderlich, wenn das Prädikat erfüllt ist, also eine der oben genannten Bedingungen keine korrekte Refaktorisierung garantiert.

A.8 Ausgelassenes

Steimann und Thies [36] identifizieren weitere Constraintregeln, die bisher nicht erwähnt wurden. Dieser Abschnitt benennt diese Regeln und begründet, warum diese Regeln für die untersuchte PULL-UP-FIELD-Refaktorisierung nicht relevant sind.

Die Refaktorisierung tastet Zugriffsmodifikatoren von Methoden oder Konstruktoren nicht an. Sie verlagert weder die Deklaration von Methoden oder Konstruktoren noch verändert sie den Ort ihrer Aufrufe. Der Algorithmus zur Auswahl der *most specific method* ignoriert die hier möglicherweise veränderte Zugreifbarkeit von Parametertypen am Ort des Aufrufs [12, § 15.12.2, S. 445 ff.]. Deshalb können Constraints nicht verletzt werden, die Überschreiben, dynamisches Binden oder Überladen von Methoden und Konstruktoren betreffen – vorausgesetzt, die Bindung von Classifier-Referenzen bleibt erhalten. Die in der Vorarbeit *Sub*, *Dyn* und *Ovr* bezeichneten Constraintregeln behandeln entsprechend nur Methoden und Konstruktoren. Sie können unter der genannten Voraussetzung auslassen werden.

Die Voraussetzung ist aus folgenden Gründen gerechtfertigt. Den Erhalt der Bindung von Classifier-Referenzen an ihre Deklaration stellen die Constraintregeln *Acc-1* und *Suppl-2* sicher. Da die untersuchte Refaktorisierung Classifier-Deklarationen nicht verlagert, kann Verbergen (*hiding*) nicht entstehen oder aufgehoben werden. Regel *Hid-4* verhindert, dass Abschatten durch Erhöhen der Zugreifbarkeit von Classifiern entsteht. Umgekehrt könnten neue Import-Deklarationen gleichnamige Classifier-Deklarationen im gleichen Package abschatten. Dieses Problem wird hier dadurch umgangen, dass keine Import-Deklarationen verlagert oder neu erzeugt werden. Stattdessen werden verschobene Classifier-Referenzen voll qualifiziert angenommen.

A.9 Grenzen

Nicht alle Lücken der Constraintregeln konnten im Rahmen dieser Arbeit geschlossen werden. Deshalb werden Refaktorisierungen gezielt unterbunden, die eigentlich durchführbar wären. Umgekehrt werden in seltenen Fällen Refaktorisierungen ausgeführt, obwohl der Erhalt der Bedeutung des Programms nicht garantiert ist. Dieser Abschnitt fasst die Einschränkungen an einer Stelle zusammen.

Grundsätzliches

Constraintbasierte Refaktorisierung ist bisher auf statische Programmanalyse beschränkt. Deshalb kann sie die Korrektheit der Refaktorisierung von Programmen nicht garantieren, deren Ablauf auf Introspektion oder Reflexion beruht. Objektorientierte Programmierung leidet unter starker, nicht dokumentierter Kopplung von Sub- und Superklassen [30, S. 302 f.]. Da die Subklassen oft gar nicht bekannt oder zugänglich sind, kann eine Refaktorisierung der Superklassen zu unbemerkten Veränderungen der Funktion von Programmen führen, die Subklassen ableiten.

Die Untersuchung wird am Beispiel einer PULL-UP-FIELD-Refaktorisierung durchgeführt. Die Constraintregeln und der Algorithmus zur selektiven Constrainerzeugung sind auf diese Refaktorisierung beschränkt. Eine Änderung der Qualifizierung von Feldern ist dabei nicht vorgesehen.

Die Anwendung der Refaktorisierung ist weiter darin begrenzt, dass als Quelle und Ziel des Hochziehens von Feldern nur Klassen und Interfaces infrage kommen. Aufzählungstypen (*enums*) werden hier nicht behandelt, sie werden deshalb nicht refaktorisiert. Als Typ eines Felds sind sie jedoch zugelassen. Annotationstypen haben selbst zwar keine Sub- oder Supertypen [12, § 9.6, S. 271], können aber Interfaces und Klassen enthalten. Auch diese in Annotationstypen geschachtelten Classifier werden hier nicht betrachtet. Annotierte Felder und als Annotationsparameter vorkommende Classifier- oder Feldreferenzen werden jedoch korrekt refaktorisiert.

Zu starke Constraintregeln

Die oben angeführten Constraintregeln decken eine Reihe möglicher Konstellationen in zu refaktorisierenden Programmen nicht ab, oder die Art und Weise der vorgesehenen Code-Transformation unterliegt einer bestimmten Einschränkung. Das führt dazu, dass in folgenden Situationen Refaktorisierungen nicht ausgeführt werden können:

- Felder, deren Initialisierungsausdruck Methoden- oder Konstruktoraufrufe enthält, können nicht hochgezogen werden, da diese Arbeit Constraintregeln für dynamisches Binden und Überladen nicht berücksichtigt.
- Felder, deren Deklaration ungebundene Typparameter enthalten, können nicht hochgezogen werden, da entsprechende Constraintregeln fehlen.
- Verlagerte Classifier-Referenzen werden voll qualifiziert angenommen, da Constraintregeln für eine korrekte Verlagerung oder Generierung von Import-Deklarationen fehlen.
- Die Refaktorisierung sieht eine Aufteilung in einer Anweisung gemeinsam deklarerter Felder auf separate Deklarationen beim Hochziehen nicht vor.
- Die Initialisierung von Feldern wird nicht reorganisiert. Deshalb können *final* deklarierte Felder, die im Konstruktor oder Initializer-Block initialisiert werden, nicht hochgezogen werden.
- Felder, deren Deklaration statisch importierte Felder referenziert, können nicht hochgezogen werden, da die Refaktorisierung Importdeklarationen nicht verlagert oder erzeugt und die Feldreferenzen auch nachträglich nicht qualifiziert.
- Felder, deren Deklaration lokale Variablen referenziert, sind ebenfalls vom Hochziehen ausgeschlossen.
- Die Constraintregel für Abschattung durch geschachtelte Classifier (*Hid-4*) ist stark vereinfacht und kann daher Hochziehen verhindern, obwohl es möglich wäre.
- Eine Verlagerung statischer Felder in innere Klassen wird vereinfachend generell unterbunden (vgl. *Suppl-3*), obwohl dies erlaubt wäre, wenn es sich um eine zur Compile-Zeit auswertbare Konstante handelt.

In den hier genannten Fällen werden Refaktorisierungen unterbunden, obwohl sie möglicherweise korrekt durchführbar wären.

Fehlende Constraintregeln

Es gibt offensichtlich auch Situationen, in denen die Refaktorisierung aus anderen, nicht abgefangenen Gründen zu Fehlern führen kann.

Verdunkeln Felder können, je nach Ort und Zugreifbarkeit, Classifier und Packages *verdunkeln* (*obscure* [12, § 6.3.2, S. 122]). Hochziehen von Feldern oder Erhöhen der Zugreifbarkeit von Feldern oder Classifiern kann das Problem verursachen, insbesondere wenn Javas Konventionen zum Benennen von Packages, Classifiern und Mitgliedern nicht eingehalten sind. Eine Constraintregel, die Verdunkeln verhindert, fehlt bisher. Folgender Codeausschnitt demonstriert das Problem.

```

87  class A {
88      int m(){ return c.i; }
89  }
90  class B extends A {
91      C c= new C();
92      int n(){ return c.i; }
93  }
94  class C {
95      int i= 1;
96  }
97  class c {
98      static int i= 0;
99  }

```

Zieht man das Feld `c` von Klasse `B` in die Klasse `A` hoch, verdunkelt die neue Deklaration des Felds `c` in `A` die Deklaration der Klasse `c`. Nach dieser Refaktorisierung gibt die Methode `m` den Wert 1 statt 0 zurück. Reduziert man nun den Zugriffsmodifikator des Felds `c` von *package* auf *private*, ändert Methode `n` ihren Rückgabewert von 1 auf 0.

Kontrollfluss Das Hochziehen von Feldern kann auch den Kontrollfluss eines Programms und damit sein Verhalten ändern. Das Problem betrifft vor allem Felder, deren Initialisierungsausdrücke Methodenaufrufe oder Feldzugriffe mit Seiteneffekten enthalten. Folgendes Beispiel zeigt, dass ungewollte Änderungen auch ohne Seiteneffekte in Initialisierungsausdrücken entstehen können.

<pre> 100 class A { 101 int i= 0; 102 103 A() { i= 1; } 104 } 105 class B extends A { 106 int j= i; 107 } </pre>	<pre> class A { int i= 0; int j= i; A() { i= 1; } } class B extends A { } </pre>
---	--

Zieht man das Feld `j` von Klasse `B` (links) in die Klasse `A` hoch (rechts), dann liefert der Ausdruck `new B().j` nach dem Hochziehen den Wert 0, im ursprünglichen Programm aber den Wert 1. Das Hochziehen vertauscht die Reihenfolge der Ausführung des Konstruktors der Klasse `A` und der Initialisierung des Felds `j`. Bisher fehlen Constraintregeln, die diese Art von Verhaltensänderung verhindern. Diese Probleme bleiben im Rahmen der vorliegenden Arbeit ungelöst.

API Es fehlt schließlich eine Constraintregel, die verhindert, dass die Zugreifbarkeit von solchen Feldern vermindert wird, die mit `@API` oder mit einer vergleichbaren Annotation ausgezeichnet und in einer Klasse deklariert sind. Bei Classifiern sorgt Regel *Suppl-2* für den Erhalt der Zugreifbarkeit. Die Zugreifbarkeit weiterer Member wird von der betrachteten Refaktorisierung nicht verändert. Der Bedarf einer entsprechenden Constraintregel für diese annotierten Felder von Klassen wurde hier schlicht übersehen.

B Tabellen

B.1 Verwendete Formelzeichen

Die exakte Bedeutung der Formelzeichen wird jeweils bei ihrer ersten Verwendung eingeführt. Die folgende tabellarische Zusammenstellung gibt eine knappe Erläuterung, die das Lesen der Arbeit erleichtern soll und verweist auf die Seite der einführenden Verwendung.

Tabelle 7 – Formelzeichen

Symbol	Bedeutung	Seite
A	Menge von Zugreifbarkeiten: $\{private, package, protected, public\}$	56
B	Menge der Wahrheitswerte (<i>Boolean</i>)	56
c	Constraint	13
\mathcal{C}	Menge von Constraints	13
c	Classifier	59
C	Menge von Classifiern	56
\mathcal{D}	Wertebereich einer Constraintvariablen	13
D	Menge von Deklarationselementen	56
d	Deklarationselement	59
$D \cup R$	Menge von Programmelementen	56
e	Programmelement, eine Deklaration oder eine Referenz	15
E	Menge der Kanten eines Graphen (<i>edges</i>)	61
G	Graph	61
K	Menge der Arten von Programmelementen (<i>kinds</i>)	56
P	Weg in einem Graphen (<i>path</i>)	61
\mathcal{P}	Programm	19
P_j	Menge von (Java-)Packages	56
q	Quantil	38
R	Menge von Referenzen	56
r	Referenz	59
v	Ecke eines Graphen (<i>vertex</i>)	61
V	Menge der Ecken eines Graphen (<i>vertices</i>)	61
X	Menge von Constraintvariablen	13
α	Eigenschaft: Zugriffsmodifikator einer Deklaration (<i>accessibility</i>)	57
	Funktion: Minimal erforderliche Zugreifbarkeit	59
β	Bindung einer Referenz an die Deklaration	57
δ	Deklarationsanweisung	57
ι	Bezeichner (<i>identifier</i>)	57
κ	Art eines Programmelements (<i>kind</i>)	57
λ	Ort eines Programmelements (<i>location</i>)	57
π	(Java-)Package	57
ρ	Empfängertyp einer Referenz (<i>receiver type</i>)	57
$<_A$	Ordnungsrelation der Zugreifbarkeiten	58
$<_I$	Subtypbeziehung	58
\prec_I	direkter Subtyp (<i>implements</i> bzw. <i>extends</i>)	58
$<_N$	Schachtelungsbeziehung (<i>nesting</i>)	58
\Rightarrow	Trennung von Prämisse und Konklusion einer Constraintregel	58
\rightarrow	In Constraintregel: Implikation	58

B.2 Abkürzungen

Folgende Tabelle listet im Text wiederkehrend verwendete Abkürzungen auf und gibt die Seite ihrer Einführung an.

Tabelle 8 – Abkürzungen

Abkürzung	Bedeutung	Seite
API	Programmierschnittstelle (<i>application programming interface</i>)	24
AST	Abstrakter Syntaxbaum (<i>abstract syntax tree</i>)	25
CSP	Constraint-Satisfaction-Problem	13
DCSP	Dynamisches Einschränkungproblem (<i>dynamic constraint satisfaction problem</i>)	50
DSL	Domänenspezifische Sprache (<i>domain specific language</i>)	12
JDT	Java-Development-Tools (Eclipse)	24
LTK	Language-Tool-Kit (Eclipse)	24
RTT	Refactoring-Tool-Tester	35

B.3 Bezeichnung der Constraintregeln

Die vorliegende Arbeit verwendet im Theorieteil (Abschnitt 3) und im Implementierungsteil (Abschnitt 4) verschiedene Bezeichnungen für Constraintregeln und für die entsprechenden Modell-Constraints, um die Trennung von *Theorie* und *Implementierung* nicht zu verwässern. Die folgende Tabelle soll es erleichtern, den unabhängigen Theorieteil mit dem davon abhängigen Implementierungsteil in Beziehung zu setzen.

Tabelle 9 – Constraintregeln und entsprechende Constrainttypen

Bezeichnung	Bezeichner der Klasse
Acc-1	<code>AccessibilityConstraint</code>
Acc-2	<code>PrivilegedAccessConstraint</code>
Inh-1	<code>InheritanceConstraint</code>
Inh-2	<code>AmbiguousFieldReferenceConstraint</code>
Hid-1	<code>HidingMemberConstraint</code>
Hid-2	<code>ShadowingImplicitThisConstraint</code>
Hid-3	<code>ShadowingStaticImportConstraint</code>
Hid-4	<code>ShadowingClassifierConstraint</code>
Misc-2	<code>CommonFieldDeclarationConstraint</code>
Misc-4	<code>TopLevelClassifierAccConstraint</code>
Misc-5	<code>ImportAccessibilityConstraint</code>
Misc-8	<code>InterfaceMemberConstraint</code>
Suppl-0	<code>UniqueIdentifierConstraint</code>
Suppl-1	<code>ReferenceLocationConstraint</code>
Suppl-2	<code>AccessibilityPreservationConstraint</code>
Suppl-3	per <code>FixedLocationConstraint</code>
Suppl-4	per <code>FixedLocationConstraint</code>
Suppl-5	<code>FixedLocationConstraint</code>

B.4 Gliederung der Implementierung

Die Implementierung des Refaktorisierungswerkzeugs ist in zwei Eclipse-Plugins (s. o.) und in fünf Packages gegliedert. In Tabelle 10 ist der gemeinsame Präfix aller genannten Packa-

ges `de.erlandm.cgcs.` weggelassen. Das Package `de.erlandm.cgcs.testing` ist auf beide Eclipse-Plugins verteilt. Die in der Tabelle angegebenen Codezeilen sind mit dem Programm *Sloccount* gezählt und auf ganze 100 Zeilen gerundet. Die Zahlen sollen eine grobe Abschätzung des Aufwands der Implementierung ermöglichen (vgl. Abschnitt 5).

Tabelle 10 – Aufteilung der Implementierung auf Packages

Name	Codezeilen in		Zuständigkeiten
	Impl.	Tests	
<code>elements</code>	1700	1100	Programmelemente und Hilfsklassen zur Ermittlung ihrer Eigenschaften
<code>constraints</code>	1200	700	Constraints mit ihrem Adapter an den Constraintlöser
<code>generation</code>	1100	2600	Constraintertezeugung per Graphendurchlauf und Hilfsklassen zur Ermittlung abzusuchender Typhierarchien
<code>refactoring</code>	700	200	Code-Modifikation, Einbindung der Refaktorisierung in das <i>Language-Tool-Kit</i> , Benutzungsinterface zur Demonstration der Refaktorisierung
<code>testing</code>	1100	100	Werkzeuge zum automatischen Ausführen der Refaktorisierung auf Probanden samt Benutzungsinterface

B.5 Anzahlen von Lösungen der Constraintmengen

Die Verteilung der Anzahlen von Lösungen der Constraintmengen ist so schief, dass erst eine Darstellung der Häufigkeiten von Anzahlen der Lösungen mit logarithmisch eingeteilten Häufigkeitsklassen einen Eindruck der Verteilung vermittelt (vgl. Abschnitt 5). Tabelle 11 gibt die Häufigkeiten von Anzahlen der Lösungen aller auf den Probanden versuchten Refaktorisierungs-Operationen wieder.

Tabelle 11 – Verteilung der Anzahlen von Lösungen der Constraintmengen

Proband	0	1	2-3	4-10	11-30	31-100	101-300	301-1000	>1000
Apache Math	653	215	321	52	52	2	10	48	17
Apache Codec	69	94	9	10	2	0	0	0	32
Apache Ivy	1813	211	689	31	17	3	0	0	66
dom4j	276	13	234	5	0	0	0	0	0
Draw2d	735	85	447	21	16	4	7	6	0
HTML Parser	177	182	358	2	4	4	4	0	19
Jaxen	122	192	49	1	0	0	0	0	0
JFreeChart	6820	1284	1998	377	152	18	6	12	0
JHotDraw	939	193	406	37	41	8	0	0	0
JUnit 4.8	187	15	91	0	0	0	0	0	0
XOM	92	3	155	2	0	0	0	0	13
Summe	11883	2487	4757	538	284	39	27	66	147

C Bedienung und Installation

C.1 Demo-Refaktorisierung

Zur Demonstration der implementierten Refaktorisierung ist eine minimale Benutzungs-schnittstelle implementiert. Die Refaktorisierung kann aus dem Kontextmenü eines Felds

im Outline-View oder Package-Explorer über den zusätzlichen Eintrag „Pull up Field Refactoring“ aufgerufen werden. Dieser Eintrag führt zu dem in Bild 10 wiedergegebenen Konfigurationsdialog. Feld und deklarierende Klasse sind darin vorgegeben. Aus dem Drop-down-Menü kann ein Typ ausgewählt werden, in den das Feld hochgezogen werden soll. Die Liste enthält nur die Supertypen, deren Quellcode editierbar ist. Die Buttons haben die bei Eclipse-Refaktorisierungen übliche Wirkung und zeigen eine Vorschau, brechen die Refaktorisierung ab oder führen sie direkt aus. Hat die Constraintmenge keine Lösung, erscheint der bekannte Fehlerdialog mit dem Hinweis, dass keine Lösung für die Constraintmenge gefunden werden konnte. Die Refaktorisierung ist dann nicht ausführbar. Andernfalls, wird die Refaktorisierung der ersten ermittelten Lösung der Constraintmenge entsprechend ausgeführt.

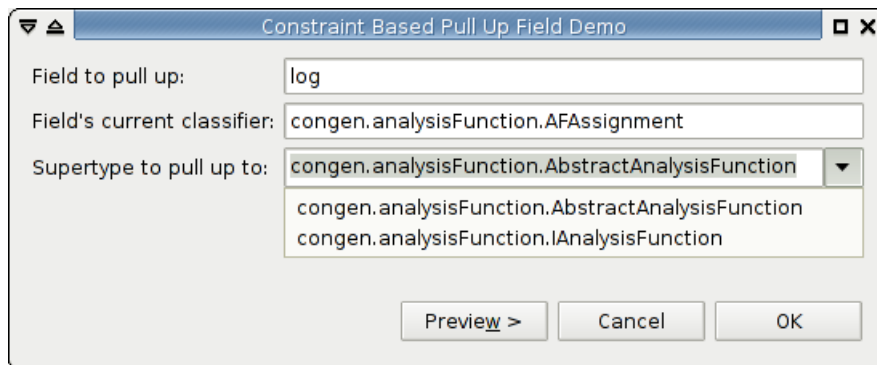


Bild 10 – Konfigurationsdialog der Demo-PULL-UP-FIELD-Refaktorisierung

C.2 Quellcode der Implementierung

Der Quellcode des implementierten Refaktorisierungswerkzeugs ist als Eclipse-Plugin-Projekt von der Seite <http://www.stud.fernuni-hagen.de/q7248008/cgcs/> herunterladbar. Kompilieren und Ausführen des Projekts setzt eine Eclipse-Java-Entwicklungsumgebung in einer Version ab 3.5 (*Galileo*) und eine Java-Laufzeitumgebung in einer Version ab 1.6 voraus.

Das Zip-Archiv „de.erlandm.cgcs-0.1.6.zip“ enthält zwei Plugin-Projekte, die zusammen in einen Eclipse-Workspace importiert werden können (File → Import → General/Existing Project into Workspace → Next → Select archive file ...): Das Projekt *de.erlandm.cgcs* enthält die Implementierung der Constraintzerzeugung samt Testfällen. Dieses Plugin ist abhängig vom zweiten im Archiv enthaltenen Plugin *org.eclipse.contribution.junit.test* [11, S. 371 ff.], das die Klassen für die Test-Fixtures bereitstellt.

Die Test-Suite des Plugins *de.erlandm.cgcs* ist aus dem Workspace, in den das Projekt importiert wird, direkt ausführbar (Kontextmenü des Verzeichnisses *tests* im Projekt *de.erlandm.cgcs* → Run As → JUnit Plug-in Test). Das Plugin selbst und die darin enthaltene Demo-Refaktorisierung kann in einer Tochter-Instanz der Entwicklungsumgebung ausgeführt werden (Run → Run Configurations → Eclipse Application → Run a product: „org.eclipse.sdk.ide“ → Run). Im Kontextmenü eines Felds im Outline-View, Package-Explorer oder im Java-Editor erscheint nun zusätzlich der Eintrag „Pull up Field Refactoring“. Im Kontextmenü von Java-Projekten im Package- oder Project-Explorer erscheint zusätzlich der Eintrag „Test Run Pull up Field“.

Das Zip-Archiv „de.erlandm.cgcs.rtt-0.0.2.zip“ enthält das Library-Plugin *de.erlandm.cgcs.rtt* mit dem Adapter zur Anwendung des *Refactoring Tool Testers* (RTT). Es ist von einer Installation des RTT abhängig, die El Hosami [8, S. 73 ff.] ausführlich beschreibt. Dort ist auch angegeben, wie das Refaktorisierungswerkzeug mithilfe des Adapters auf Probanden ausgeführt werden kann.

Den Exemplaren der Arbeit für das Prüfungsamt liegt jeweils eine CD mit den oben genannten Zip-Archiven und den in Tabelle 2 (S. 37) angegebenen Probanden bei.

Literatur

- [1] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullmann: *Data Structures and Algorithms*; Addison-Wesley; Reading, MS, USA; 1983.
- [2] Deepak Azad und Olivier Thomann: *JDT fundamentals – Become a JDT tool smith*; EclipseCon 2010, Santa Clara CA, March 22–25; 2010; <http://www.eclipsecon.org/2010/sessions/sessions?id=1339>, 2010-07-04.
- [3] Christian Bessiere: *Constraint Propagation*; in *Handbook of Constraint Programming* (Hg. Francesca Rossi, Peter van Beek und Toby Walsh); Kap. 3, S. 29–84; Elsevier; Amsterdam; 2006.
- [4] Philipp Bouillon, Eric Großkinsky und Friedrich Steimann: *Controlling Accessibility in Agile Projects with the Access Modifier Modifier*; in *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 – July 4, 2008*; Bd. 11 von *Lecture Notes in Business Information Processing*; S. 41–59. Springer; 2008.
- [5] Brett Daniel, Danny Dig et al.: *Automated Testing of Refactoring Engines*; in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Dubrovnik, Croatia, September 3–7, 2007 (ESEC/SIGSOFT FSE)*; S. 185–194; 2007.
- [6] Rina Dechter: *Constraint Processing*; Morgan Kaufmann; San Francisco, SF, USA; 2003.
- [7] Edsger W. Dijkstra: *Notes on Structured Programming*; in *Structured Programming* (Hg. Ole-Johan Dahl, Edsger W. Dijkstra und C. A. R. Hoare); S. 1–82; Academic Press; London; 1972.
- [8] Osama El Hosami: *Implementierung eines Eclipse-Plugins zum automatisierten Testen von Refaktorisierungswerkzeugen*; Masterarbeit, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen; 2010.
- [9] Martin Fowler: *Refactoring, Improving the Design of Existing Code*; Addison-Wesley; Boston, MS, USA; 1999.
- [10] Eugene Freuder: *In Pursuit of the Holy Grail*; *ACM Computing Surveys*; 28(4); 1996.
- [11] Erich Gamma und Kent Beck: *Contributing to Eclipse: Principles, Patterns, and Plugins*; Addison Wesley Longman; Redwood City, CA, USA; 2003.
- [12] James Gosling, Bill Joy et al.: *The Java Language Specification*; The Java Series; Addison-Wesley Professional; Upper Saddle River, NJ, USA; 3. Aufl.; 2005; <http://java.sun.com/docs/books/jls/index.html>, 2010-05-20.
- [13] Nadia Heinrich-Kerber: *Fallbeispiel „Pull Up Field“-Refactoring*; unveröffentlicht; Sept. 2010.
- [14] Petra Hofstedt und Armin Wolf: *Einführung in die Constraint-Programmierung, Grundlagen, Methoden, Sprachen, Anwendungen*; Springer; Berlin, Heidelberg; 2007.
- [15] Sergei Ikkert: *Untersuchung der Eclipse-JDT-Refaktorisierungen mit Hilfe des Refactoring Tool Testers*; Masterarbeit, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen; 2010.
- [16] *JSR-331, Java Constraint Programming API*; Specification, Version 0.7.1, Java Community Process; Okt. 2010; <http://jcp.org/en/jsr/summary?id=331>, 2010-12-23.
- [17] Narendra Jussien, Guillaume Rochart und Xavier Lorca: *Choco: an Open Source Java Constraint Programming Library*; in *CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08), Paris, May 20–23*; 2008; <http://hal.archives-ouvertes.fr/hal-00483090>, 2010-08-26.
- [18] Andreas Kahl: *Untersuchung der Eclipse Refactoringtools hinsichtlich der Beachtung von Java Zugreifbarkeitsregeln*; Masterarbeit, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen; 2010.
- [19] Hannes Kegel: *Constraint-basierte Typinferenz für Java 5*; Diplomarbeit, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen; 2007.
- [20] Hannes Kegel und Friedrich Steimann: *Systematically Refactoring Inheritance to Delegation in Java*; in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008*; S. 431–440. ACM; 2008.

- [21] Kurt Mehlhorn und Peter Sanders: *Algorithms and Data Structures, The Basic Toolbox*; Springer; Berlin, Heidelberg; 2008.
- [22] Tom Mens und Tom Tourwé: *A Survey of Software Refactoring*; *IEEE Transactions on Software Engineering*; 30(2), S. 126–139; 2004.
- [23] Emerson Murphy-Hill und Andrew P. Black: *Refactoring Tools: Fitness for Purpose*; *IEEE Software*; 25(5), S. 38–44; 2008.
- [24] Jens Palsberg und Michael I. Schwartzbach: *Object-Oriented Type Systems*; Wiley; 1994.
- [25] Don Roberts und John Brant: *Refactoring Tools*; in [9]; Kap. 14, S. 401–407; 1999.
- [26] Max Schäfer, Torbjörn Ekman und Oege de Moor: *Sound and Extensible Renaming for Java*; in *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19–23, 2008, Nashville, TN, USA*; S. 277–294. ACM; 2008.
- [27] Max Schäfer und Oege de Moor: *Specifying and Implementing Refactorings*; in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17–21, 2010, Reno, NV, USA*; S. 286–301. ACM; 2010.
- [28] Gustavo Soares, Rohit Gheyi et al.: *Making Program Refactoring Safer*; *IEEE Software*; 27(4), S. 52–57; 2010.
- [29] Friedrich Steimann: *The Infer Type Refactoring and its Use for Interface-Based Programming*; *Journal of Object Technology*; 6(2), S. 99–120; 2007.
- [30] Friedrich Steimann: *Kurs 01814: Objektorientierte Programmierung*; Kurstext, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen, Nr. 01814-6-01-S 1, unveröffentlicht; 2009.
- [31] Friedrich Steimann: *Kurs 01853: Moderne Programmiermethoden und -methoden*; Kurstext, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen, Nr. 01853-5-01-S 1, unveröffentlicht; 2009.
- [32] Friedrich Steimann: *Cross-Language Refactoring Cookbook*; unveröffentlicht; Juni 2010.
- [33] Friedrich Steimann: *Korrekte Refaktorisierungen: Der Bau von Refaktorisierungswerkzeugen als eigenständige Disziplin*; *Objektspektrum*; 2010(4), S. 24–29; 2010.
- [34] Friedrich Steimann: *Refactoring with Foresight*; Manuskript; Aug. 2010; <http://www.feu.de/ps/docs/Foresight.pdf>, 2011-02-24.
- [35] Friedrich Steimann, Christian Kollee und Jens von Pilgrim: *A Refactoring Constraint Language and its Application to Eiffel*; Manuskript, akzeptiert zur ECOOP 2011; Dez. 2010.
- [36] Friedrich Steimann und Andreas Thies: *From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility*; in *ECOOP 2009 – Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6–10, 2009. Proceedings*; Bd. 5653 von *Lecture Notes in Computer Science*; S. 419–443. Springer; 2009.
- [37] Friedrich Steimann und Andreas Thies: *From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation*; in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, Cape Town, SA, 1–8 May 2010*. ACM; 2010.
- [38] Andreas Thies: *Re: Thema Abschlußarbeit*; E-Mail vom 29.05.2010, unveröffentlicht; Mai 2010.
- [39] Frank Tip: *Refactoring Using Type Constraints*; in *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22–24, 2007*; Bd. 4634 von *Lecture Notes in Computer Science*; S. 1–17. Springer; 2007.
- [40] Frank Tip, Adam Kiezun und Dirk Bäumler: *Refactoring for generalization using type constraints*; in *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26–30, 2003, Anaheim, CA, USA*; S. 13–26. ACM; 2003.
- [41] Gérard Verfaillie und Narendra Jussien: *Constraint Solving in Uncertain and Dynamic Environments: A Survey*; *Constraints*; 10(3), S. 253–281; 2005.
- [42] Tobias Widmer: *Unleashing the Power of Refactoring*; Eclipse Corner Article; Febr. 2007; <http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html>, 2008-11-04.