KABA
Ein System zur Refaktorisierung von Java-Programmen

Mirko Streckenbach

strecken@infosun.fmi.uni-passau.de
Universität Passau, Lehrstuhl für Softwaresysteme, Innstr. 33, 94032 Passau

Abstract: Refactoring ist eine bekannte Technik, um verschiedene Aspekte eines objekt-orientierten Programms zu verbessern. Sie ist in den letzten Jahren sehr populär geworden, da sie es erlaubt, Defizite zu beseitigen, die sich in sehr vielen Programmen finden.

Die Größe moderner Software-Systeme macht es unmöglich, Refactoring von Hand durchzuführen. Zwar existieren Werkzeuge, die es ermöglichen Refactorings automatisch anzuwenden, aber sie machen keine Vorschläge, welches Refactoring angewendet werden sollte und warum. Die Snelting/Tip-Analyse ist eine Programm-Analyse, die einen Restrukturierungs-Vorschlag für eine ganze Klassen-Hierarchie macht. Sie basiert auf der Analyse der Verwendung von Klassen-Members.

KABA ist eine Adaption und Erweiterung der Snelting/Tip-Analyse für Java. Ihre Implementierung ist erweitert worden zu einem semantik-erhaltenden, interaktiven Refactoring-System. Fallstudien belegen die Nützlichkeit dieses Systems in der Praxis.

1 Einführung

Das Design einer Klassen-Hierarchie ist schwierig. Der Designer muss alle möglichen Verwendungen der Klassen vorhersehen und alleine aufgrund der Tatsache, dass heutige Software-Systeme aus hunderten von Klassen bestehen, macht dies schwierig. Die Evolution eines Software-Systems verstärkt dies noch: Neue Funktionalität muss eingebaut werden, neue Konzepte kommen hinzu und oft wird die Entropie vergrößert. Auf der anderen Seite soll der diese Klassen benutzende Klienten-Code von den Änderungen überhaupt nicht betroffen sein.

Die Bibliothek des Java Development Kit ist ein Beispiel: Von Java 1.0 bis zu 1.5 hat sich die Anzahl der Klassen im `java.` Namensraum fast verzehnfacht. Enthalten sind inzwischen 2 verschiedene Ansätze für GUI-Programmierung, 3 für I/O und 4 für Container-Klassen.

Eine etablierte Technik gegen Fehler im Design und negative Folgen des Evolutionsprozesses ist *Refactoring*. Refactoring ist ein genereller Begriff, der sich auf Programm-Transformationen bezieht, die Struktur, Lesbarkeit oder Maintainbarkeit eines Programms erhöhen. Der Begriff wurde eingeführt von Opdyke und Johnson [OJ93], erlangte aber erst durch Fowler [Fow99] große Bekanntheit. Sein Buch biete einen Katalog von *Refactoring Patterns*, die auf ein Programm angewendet werden können.

```
class Person {
   public String name;
   public String address;
   public int socialSecurityNumber;
}
class Student extends Person {
   public int studentId;
   public Professor advisor;
   public Student(String sn, String sa, int si) {
      name=sn; address=sa; studentId=si;
   }
   public void setAdvisor(Professor p) {
      advisor=p;
   }
}
class Professor extends Person {
   String workAddress;
   Student assistant;
   public Professor(String n, String wa) {
      name=n; workAddress=wa;
   }
   public void hireAssistant(Student s) {
      assistant=s;
   }
}
```

```
class Client1 {
   static public void main(String args[]) {
      Student s1=new Student("Carl", "here", 12345678);
      Professor p1=new Professor("Prof. X", "there");
      s1.setAdvisor(p1);
   }
}
class Client2 {
   static public void main(String args[]) {
      Student s2=new Student("Mary", "here too", 87654321);
      Professor p2=new Professor("Prof. Y", "not there");
      p2.hireAssistant(s2);
   }
}
```

Abbildung 1: Beispiel: Professoren und Studenten

Auf den ersten Blick ist Refactoring ein manueller Prozess, aber dies ist nur für kleine Programme realistisch. Eine lokale Variable ist einem 10 Zeilen Programm kann man leicht von Hand umbenennen, den Namen einer Member-Variable in einem 100000 Zeilen Programm zu ändern, ist schon sehr viel schwerer.

Moderne Programmierumgebungen (z.B. Eclipse) bieten bereits Support für Refactoring. Allerdings beschränkt sich dies auf die Anwendung von Refactorings, es werden keine Vorschläge gemacht, wie ein Programm zu verbessern ist. Dafür gibt es verschiedene Gründe. Zum einen haben viele Refactorings komplexe Vorbedingungen, die nicht oder nur schwer automatisch prüfbar sind, zum anderen gibt es für viele Refactorings ein “Gegen-Refactoring”. Zu entscheiden welches passender ist, ist im Einzelfall für einen Menschen schwer, für eine Maschine fast unmöglich.

Die Snelting/Tip-Analyse geht einen anderen Weg. Sie untersucht die reale Verwendung von Members durch Objekte und berechnet eine neue Klassenhierarchie, in der jedes Objekt genau die Members enthält, die es wirklich benutzt. Die neue Hierarchie ist spezifisch für die analysierten Klienten und semantik-erhaltend. Diese wird erreicht durch eine Kombination von Programm-Analyse, Type-Constraints und Begriffsverbänden.

Abb. 1 zeigt ein einfache Beispiel-Hierarchie aus drei Klassen und zwei Klienten, die diese Klassen verwenden. Die oberste Klasse `Person` enthält Members für Namen, Adresse und Sozialversicherungsnummer einer Person. Sie wird von einer Klasse `Student` um Matrikelnummer und einen Professor als Betreuer erweitert. Die Klasse `Professor` ist ebenfalls Unterklasse von `Person` und fügt eine Büro-Adresse sowie einen Studenten als Assistenten hinzu. Beide Klienten erzeugen je einen Professor und einen Studenten und rufen dann zwei verschiedene Methoden auf.

Abb. 2 zeigt das durch Snelting/Tip erstellte Klassen-Hierarchie. Jeder Kasten repräsentiert eine Klasse, in der oberen Hälfte deren Members, in der unteren Hälfte Objekte und Pointer, deren neuer Typ diese Klasse ist, aufgeführt. Mehrere Punkte sind bemerkenswert:

- `java.lang.Object` ist nicht die oberste Klasse. Diese oberste Klasse hat keine

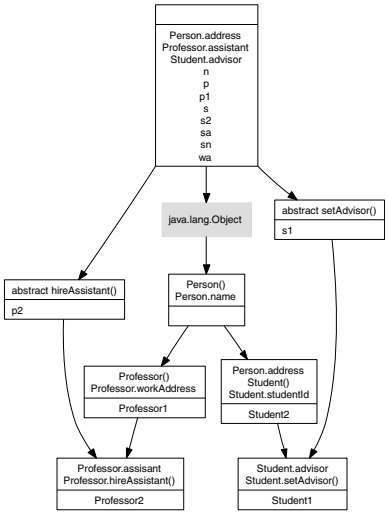

Abbildung 2: Refactoring-Vorschlag zu Abb. 1

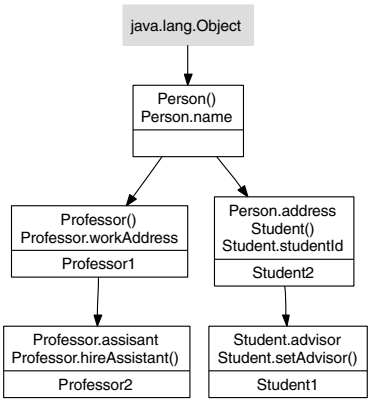

Abbildung 3: Vereinfachte Version von Abb. 2

Members und trotzdem haben mehrere Pointer ihren Typen. Dies liegt in diesem Fall am kleinen Beispiel, ist aber generell ein Anzeichen für einen Programmier-Fehler oder redundante Variablen.

- Die Members der Klasse `Professor` wurde auf zwei Klassen verteilt, die in einer Vererbungsbeziehung stehen. Die Analyse hat festgestellt, dass `Professor1` weder die Methode `hireAssisant` noch `assisant` benutzt und diese deshalb in eine Unterklasse verschoben.
- Auch die Klasse `Student` wurde in zwei Klassen geteilt: Studenten mit Betreuer oder ohne.
- Das Member `address` wurde von `Person` nach `Student` verschoben, da es von Professoren nicht benutzt wurde.
- Für die Pointer `p2` und `s1` wurden eigene Interfaces vorgeschlagen, die nur die von diesen Pointer benutzten Methoden beinhalten. Diese Interfaces werden dann von den Klassen `Professor` bzw. `Student` implementiert.
- Das Member `socialSecurityNumber` wurde ganz aus der Hierarchie entfernt, da sie überhaupt nicht verwendet wurde.

Es ist wichtig, diese Hierarchie nur als Refactoring-Vorschlag zu betrachten. Sie sollte von einem Programmierer der mit dem Code vertraut ist, nachbearbeitet werden. Z.B. könnten geteilte Klassen wieder zusammengefügt werden oder Members, von denen man weiß, dass sie in Zukunft noch benötigt werden, wieder eingefügt werden. KABA bietet die Möglichkeit, diese Änderungen semantik-erhaltend durchzuführen. Abb. 3 zeigt eine sinnvoll nachbearbeitete Version des Beispiels.

2 Die Snelting/Tip-Analyse

Die folgende Darstellung der zugrunde liegenden Snelting/Tip-Analyse und KABA selbst ist extrem kompakt. Für eine Beschreibung dieser Details sei zusätzlich auf auf [ST00] und [SS04] verwiesen.

Die Grundidee der Snelting/Tip-Analyse ist, festzustellen, welche Objekte welche Members wirklich benutzen und von dieser Information aus eine neue Klassenhierarchie zu erstellen, in der alle redundanten Members entfernt wurden.

Die zentrale Datenstruktur ist eine binäre Tabelle. Die Zeilen dieser Tabelle sind mit Pointern und Objekten des Programms beschriftet, wobei Objekte durch ihre *creation site*, d.h. die Stelle ihrer Erzeugung im Quelltext, identifiziert werden. Die Spalten der Tabelle sind mit den Members der Klassen beschriftet. Ein Tabelleneintrag ist 1, wenn ein Objekt ein Member benutzt und sonst 0.

Zu dieser Tabelle lässt sich ein Verband berechnen[GW96]. Dessen graphische Darstellung kann als neue Klassen-Hierarchie interpretiert werden: Knoten entsprechen Klassen und Kanten werden zu Vererbungsbeziehungen. Aus den Beschriftungen ergeben sich die Members der Klassen und die neuen Typen für Objekte.

Diese Hierarchie garantiert äquivalentes Programmverhalten für den analysierten Code.

3 KABA

3.1 Snelting/Tip für JAVA

Die ursprüngliche Snelting/Tip-Analyse konzentriert sich auf den Kern von C++. Es waren Erweiterungen dieser Analyse nötig um Java-Bytecode vollständig behandeln zu können. Dazu gehören die Behandlung von `instanceof`, Type-Casts, Exception-Handling, statischen Klassen-Members, Klassen-Konstruktoren und überladene Methoden. Weiterhin gibt es eine spezielle Behandlung von Bibliotheks-Klassen, die zwar analysiert werden müssen, aber nicht refaktorisiert werden sollen. Alle Erweiterungen werden durch Veränderungen der Tabelle realisiert.

Diese Erweiterungen ermöglichen es, Java-Bytecode ohne Einschränkung zu analysieren. Für Konzepte in Java, die nicht direkt im Bytecode repräsentiert werden (z.B. Inner Classes und Generics) ist keine spezielle Behandlung notwendig.

3.2 Points-To-Analyse

Ein Bestandteil von Snelting/Tip ist die *Points-To-Analyse*. Sie berechnet für jeden Pointer p eines Programms die Menge $\mathrm{PointsTo}(p)$ der Objekte, auf die p zur Laufzeit zeigen kann. Präzise Points-To-Analyse ist unentscheidbar, es gilt das Prinzip der konservativen

Approximation: Points-To-Mengen dürfen zu groß, aber nicht zu klein sein.

Zu Beginn der Arbeit an KABA war Points-To-Analyse für C bereits ein sehr intensiv erforschtes Gebiet, dagegen gab es wenig Arbeiten speziell für Java. KABA enthält daher eine vollständige Implementierung von Andersens[And94] Algorithmus.

Die größte Herausforderung von Points-To-Analyse ist die Größe des analysierten Programms. Selbst kleine Java-Programme benutzen riesige Mengen der Java-Bibliothek[1] und stellen sehr hohe Anforderungen an die Skalierbarkeit der Points-To-Analyse. Weitere Probleme sind *native Methods*, d.h. Methoden, die nicht in Java selbst, sondern in C implementiert wurden, und Reflection, d.h. die Erzeugung von Objekten, deren Typ erst zur Laufzeit festgelegt wird.

Parallel zur Entwicklung von KABA entstanden bessere skalierende Implementierungen von Points-To-Analyse [RMR01, LH03] für Java. Die Implementierung in KABA enthält dafür Unterstützung für die wichtigsten native Methods und die gebräuchlichsten Features von Reflection.

3.3 Dynamische Analyse

Snelting/Tip ist ursprünglich als statische Programm-Analyse konzipiert. Um den genannten Problemen mit der Skalierbarkeit der Points-To-Analyse entgegen zu treten, wurde eine alternative dynamische Analyse entwickelt. Anstelle das Programm zu analysieren, werden Member-Zugriffe zur Laufzeit des Programms aufgezeichnet.

Die Ergebnisse sind viel kleinere und schwächer gefüllte Tabellen, die zu kleineren und einfacheren Refactoring-Vorschlägen führt. Allerdings sind diese Vorschläge nur noch für die Programm-Läufe, die zur Aufzeichnung der Member-Zugriffe benutzt worden, korrekt, wogegen die statische Analyse Korrektheit für alle Programm-Läufe garantiert. Als Einsatzgebiet ist also primär Code geeignet, für den eine große Testsuite vorhanden ist.

3.4 Interaktives Refactoring

Da die von KABA erzeugten Klassen-Hierarchien nur als Vorschläge oder Anregungen verstanden werden sollen, ist es in den meisten Fällen notwendig, sie noch zu ändern. KABA bietet dafür einen interaktiven Refactoring-Editor, der feststellen kann, ob ein manuelle Änderung das Programm-Verhalten ändert oder nicht. Somit kann auch nach manuellem Bearbeiten der Klassen-Hierarchie die Korrektheit der neuen Hierarchie garantiert werden.

Zusätzlich wurden Heuristiken entwickelt, um Größe und Detailgrad von Refactoring-Vorschlägen automatisch zu reduzieren. Die *automatische Vereinfachung* verschmilzt zwei Klassen, wenn eine die Oberklasse der anderen ist und beide ausschließlich Members aus der gleichen Original-Klasse enthalten. Eine andere Heuristik entfernt vorkommende Mehrfachvererbung, in dem sie Klassen sucht, die die gleiche Oberklasse mehrfach

[1] Das JDK 1.5 lädt für ein “Hello, World!” Programm 295 Klassen

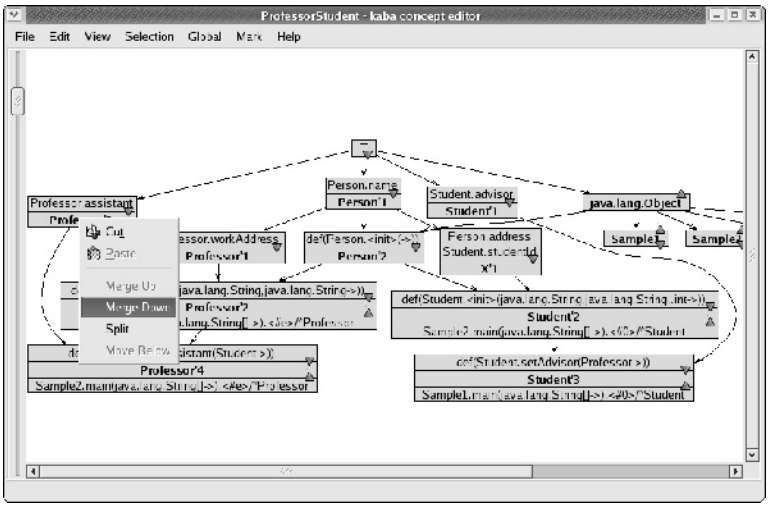

Abbildung 4: Der KABA Refactoring Editor

erben und entsprechende Substrukturen kollabiert.

4 Der KABA Refactoring Editor

Abbildung 4 zeigt einen Screenshot von KABA mit dem Beispiel aus der Einleitung. Der Refactoring-Vorschlag wird als Klassenhierarchie gezeigt, jeder Kasten entspricht einer Klasse und Vererbung wird durch die Verbindungen der Kästen angezeigt. Für jede Klasse wird ein Namensvorschlag (hervorgehoben) anzeigt, sowie die Members dieser Klasse (über dem Namen) oder die Objekte dieser Klasse (darunter). Pointer sind zur besseren Übersichtlichkeit nicht angezeigt.

Refactorings finden sich in Kontextmenüs, im Beispiel hat der Benutzer für eine Klasse, die nur ein Member hat und von der es keine Objekte gibt, ausgewählt, diese Klasse mit ihrer Unterklasse zu verschmelzen. Nach Auswahl der Aktion führt der Editor die Prüfung aus, ob dadurch das Programm-Verhalten geändert wird. Sollte dies der Fall sein, bekommt der Benutzer eine entsprechende Fehlermeldung und die Hierarchie bleibt unverändert.

5 Eine Fallstudie

KABA ist mit zahlreichen realen Programmen getestet worden, im folgenden soll ein einzelnes im Detail besprochen werden. Um Klassen-Hierarchien möglichst kompakt darstellen zu können, wurde ein sehr reduziertes Format gewählt. Klassen werden durch Kästen

dargestellt, ihre Beschriftung entspricht der Namen der Original-Klassen der Members, die sie enthalten. Nummern auf der linken Seite dienen zur Referenzierung, eine Nummer rechts unten die Anzahl der Objekte, die diesen Typ haben.

5.1 Der `antlr` Parser-Generator

Das analysierte Programm ist `antlr`, ein bekannter Parser-Generator. Er wurde mit der dynamischen Analyse analysiert, als Testläufe dienten die mitgelieferten Beispiel-Grammatiken (84 Testläufe). Die generierten Parser benutzen selbst einen Teil von `antlr`, sie wurden nicht analysiert.

Abbildung 5 zeigt einen Teil der Klassen-Hierarchie von `antlr` und Abbildung 6 den Refactoring-Vorschlag von KABA. Schon auf den ersten Blick ist zu erkennen, dass hier zahlreiche Änderungen vorgenommen wurden.

1. Die Members von 6 der 20 Klassen wurden auf mehr als eine Klasse verteilt; dabei herausragend sind die Klassen `AlternateBlock`, die in 9 Klassen geteilt wurde, sowie `AlternativeElement` (in 6) und `GrammarElement` (in 5).

2. In der Original-Hierarchie sind `GrammarElement` und `AlternativeElement` Oberklassen aller anderen Klassen. In der neuen Hierarchie gibt es immer noch eine oberste Klasse, aber sie enthält Members beider Klassen. Dies deutet an, dass die Unterscheidung dieser Klassen redundant ist.

3. Im Original war `AlternativeBlock` Unterklasse von `AlternativeElement`. Diese Beziehung ist schwächer in der neuen Hierarchie, da viele Members von `AlternativeElement` nicht in Klassen enthalten sind, die auch Members aus `AlternativeBlock` haben. Diese stellt die Vererbungsbeziehung in Frage und macht sie zu keinem Kandidaten für tiefergehende, manuelle Inspektion.

4. Die isolierte Klasse 23 enthält nur ein statisches Member. Da auf statische Members nicht über Objekte zugegriffen wird, erscheinen sie an isolierten Klassen und können manuell in jede beliebige andere Klasse geschoben werden.

Dieser Vorschlag ist sehr fein-granular und somit auch sehr kompliziert. Ein realistischer Vorschlag ist er sowieso nicht, da er noch Mehrfachvererbung enthält. Trotzdem ist hier abzulesen, dass das Design aus Abbildung 5 nicht nach dem Prinzip funktioneller Kohäsion entwickelt wurde und dass es in Bezug auf die Punkte oben ohne Änderung des Verhaltens refaktorisiert werden kann.

Abbildung 7 zeigt das Resultat nach automatischer Vereinfachung. Dieser Vorschlag sieht gleich viel überzeugender aus! Die oben genannten Punkte sind immer noch vorhanden, aber die generelle Struktur ist der Original-Hierarchie schon viel ähnlich, obwohl immer noch Klassen geteilt oder kollabiert und Members verschoben wurden. Dies führt zu einer deutlichen Erhöhung der funktionellen Kohäsion.

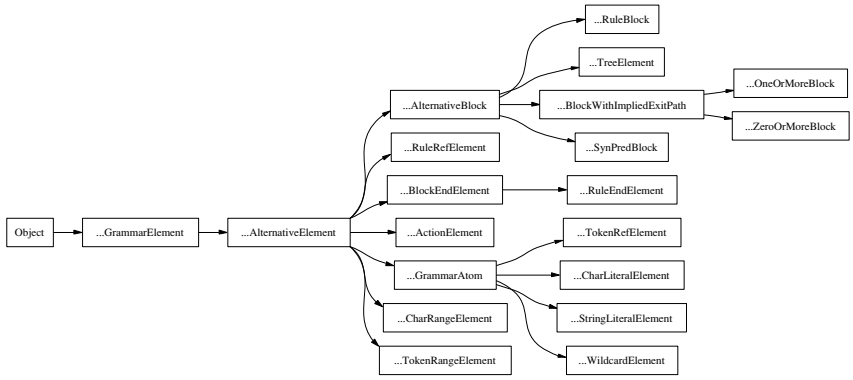

Abbildung 5: Original-Hierarchie eines Teils von `antlr`

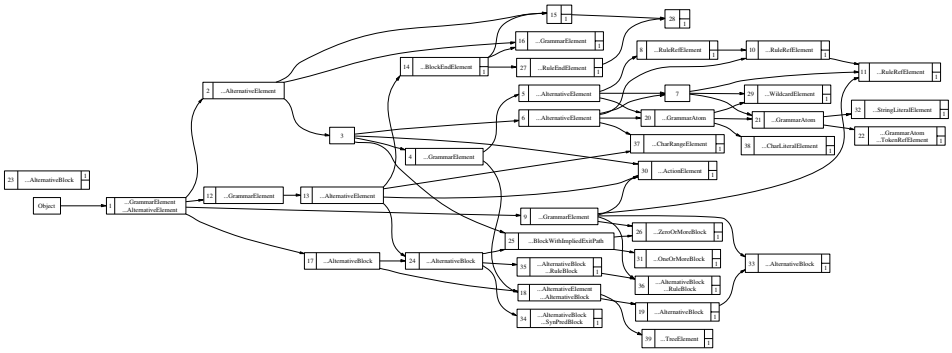

Abbildung 6: Fein-granularer KABA Refactoring-Vorschlag

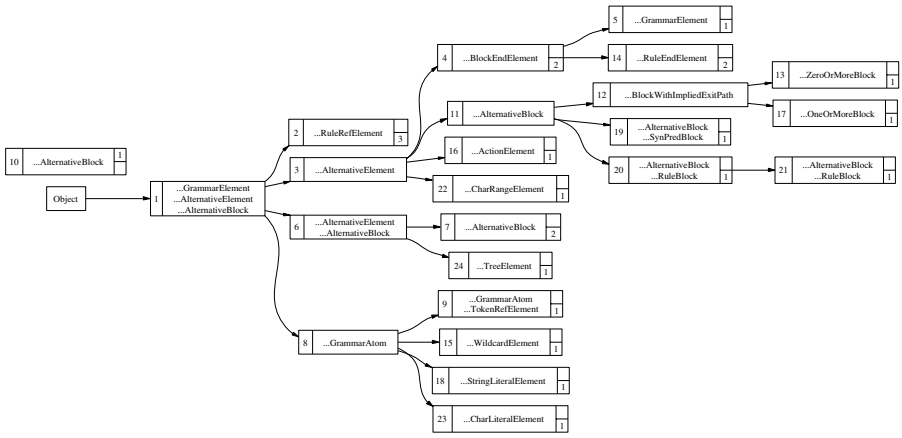

Abbildung 7: Endgültiger KABA Refactoring-Vorschlag nach Entfernung von Mehrfachvererbung

5.2 Der Java-Compiler `javac`

Eine weitere intensiv untersuchte Fallstudie ist der Java-Compiler `javac` aus SUNs JDK. Leider ist es aus Platzmangel nicht möglich entsprechende Klassenhierarchie zu zeigen.

Im Gegensatz zu dem gezeigten Beispiel schlägt KABA bei diesem Programm kaum Änderungen vor. Einige wenige Members werden verschoben, und manche Teile der Hierarchie bleiben sogar vollständig unverändert. In keinem Fall werden so tiefgehende Änderungen wie bei `antlr` vorgeschlagen. Dies bedeutet, dass hier die funktionale Kohäsion bei diesem Programm viel stärker ist und die Benutzung der Klassen nahezu vollständig ihrer Deklaration entspricht.

6 Zusammenfassung

KABA erstellt einen Refactoring-Vorschlag basierend auf der realen Verwendung von Members durch Objekte. Ein interaktiver Editor bietet weiterhin die Möglichkeit, diese Vorschläge zu bearbeiten, ohne das Programm-Verhalten zu verändern. Diese Vorschläge sind extrem detailliert, können aber durch automatische Vereinfachung und die Entfernung von Mehrfachvererbung in praktikable Form gebracht werden. In jedem Fall geben sie wertvolle Hinweise, um das Design zu verbessern.

Dabei ist immer wieder zu betonen, dass es sich nur um Vorschläge handelt; um Vorschläge, die garantiert das Verhalten des Programms nicht ändern. KABA ist kein Ersatz für einen mit dem Code vertrauten Designer oder Programmierer, der Entscheidungen über die Struktur des Programms trifft. Es kann immer andere Gründe geben, warum die Struktur nicht so sein soll, wie KABA sie vorschlägt, z.B. zukünftige Erweiterbarkeit. KABA zeigt, was man tun *kann*, nicht notwendigerweise, was man tun *soll*.

7 Andere Arbeiten

Refactoring ist seit mehreren Jahren ein aktives Forschungsgebiet, aber KABA unterscheidet sich von fast allen anderen Arbeiten in wesentlichen Punkten.

KABA führt globale Refactorings wie “move method” oder “extract class” durch. Viele andere Arbeiten (z.B. Kataoka et al. [KEGN01]) bieten lokale Refactorings, die innerhalb von Methoden arbeiten. Diese Arbeiten behandelt eine Menge von Refactorings disjunkt zu den von KABA durchgeführten Refactorings.

Casais[Cas92] hat ähnliche Ziele wie KABA, er versucht Redundanz in der Klassenhierarchie zu entfernen. Leider gibt dieser Algorithmus keine Garantien über Verhaltensäquivalenz.

Tip et al. [TKB03] beschreiben Refactoring, das genau wie KABA keine Änderungen des Verhaltens garantiert, allerdings macht dieses System keine Refactoring-Vorschläge,

sondern erlaubt nur das Anwenden von Refactorings.

KABA am ähnlichsten ist die Arbeit von Moore[Moo96]. Er beschreibt Guru, ein Tool, das Restrukturierung von Klassen-Hierarchien und Refactoring von Methoden in Self erlaubt. Da Self eine dynamisch typisierte Sprache ist, sind die vorgestellten Techniken allerdings nur sehr bedingt auf Java übertragbar.

Literatur

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Dissertation, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[Cas92] Eduardo Casais. An Incremental Class Reorganization Approach. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, Seiten 114–132. Springer-Verlag, 1992.

[Fow99] Martin Fowler. *Refactoring*. Addison-Wesley, 1999.

[GW96] Bernhard Ganter und Rudolf Wille. *Formale Begriffsanalyse : mathemathische Grundlagen*. Springer Verlag, 1996.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold und David Notkin. Automated Support for Program Refactoring Using Invariants. In *ICSM*, Seiten 736–743, 2001.

[LH03] Ondřej Lhoták und Laurie Hendren. Scaling Java points-to analysis using SPARK. In G. Hedin, Hrsg., *Compiler Construction: 12th International Conference*, 2003.

[Moo96] Ivan Moore. *Automatic Restructuring of Object-Oriented Programs*. Dissertation, Manchester University, 1996.

[OJ93] William F. Opdyke und Ralph E. Johnson. Creating abstract superclasses by refactoring. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, Seiten 66–73. ACM Press, 1993.

[RMR01] Atanas Rountev, Ana Milanova und Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, Seiten 43–55. ACM Press, 2001.

[SS04] Mirko Streckenbach und Gregor Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, Seiten 315–330. ACM Press, 2004.

[ST00] Gregor Snelting und Frank Tip. Understanding class hierarchies using concept analysis. *ACM Trans. Program. Lang. Syst.*, 22(3):540–582, 2000.

[TKB03] Frank Tip, Adam Kiezun und Dirk Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, Seiten 13–26. ACM Press, 2003.

Mirko Streckenbach wurde 1972 in Hildesheim geboren. Er studierte Informatik an der TU Braunschweig von 1993-1999. Seit April 1999 ist er am Lehrstuhl für Software-Systeme der Universität Passau tätig, wo er im April 2005 promovierte.