mehr zum thema:
www.feu.de/ps/prjs/CLaRe/

KORREKTE REFAKTORISIERUNGEN: DER BAU VON REFAKTORISIERUNGS-WERKZEUGEN ALS EIGENSTÄNDIGE DISZIPLIN

Angesichts des hohen Qualitätsstandes, den heutige integrierte Entwicklungsumgebungen erreicht haben, sollte man eigentlich glauben, dass die darin enthaltenen Refaktorisierungswerkzeuge korrekt arbeiten. Der Praxistest zeigt jedoch, dass dies längst nicht immer der Fall ist. Vergleicht man den Bau von Refaktorisierungswerkzeugen mit dem von Compilern, fällt auf, dass ersterer noch keine Disziplin ist, die allgemein anerkannte Ansätze bereitstellt, welches Problem wie am besten zu lösen ist. Als Beitrag zu einer solchen Disziplin stelle ich ein einfaches Verfahren zum systematischen Testen von Refaktorisierungswerkzeugen vor und zeige auf, wie Refaktorisierungsprobleme mittels Constraints unabhängig vom Einzelfall dargestellt und gelöst werden können.

▶ der autor

Friedrich Steimann
(E-Mail: steimann@acm.org)
ist Professor für Informatik an der Fernuniversität in Hagen, wo er das Lehrgebiet Programmiersysteme leitet. Er befasst sich seit mehreren Jahren schwerpunktmäßig mit dem Bau von Programmierwerkzeugen zur Steigerung der Programmierproduktivität.

Unter Refaktorisierung versteht man die bedeutungserhaltende Änderung von Programmen. Ihr hauptsächlicher Zweck ist es, bedeutungsverändernde Modifikationen von Programmen vor- oder nachzubereiten, also gewissermaßen die Programme „in Ordnung" (und das ist wörtlich zu verstehen) zu bringen. Das Refaktorisieren wirkt damit der „Softwarefäulnis" entgegen und ist bei der Wartung von Programmen, zumindest auf lange Sicht, unverzichtbar.

Refaktorisierungen findet man vor allem in zwei Formen vor: als verbal spezifizierte Anleitungen zur manuellen Durchführung ([Fow99] ist hier sicher das Standardwerk) oder als voll automatisierte Refaktorisierungswerkzeuge. Die Umsetzung von in Prosa beschriebenen Refaktorisierungen in Refaktorisierungswerkzeuge zeigt allerdings regelmäßig, dass die Beschreibungen unvollständig sind: Viele Voraussetzungen der Anwendung (die so genannten *Vorbedingungen*) werden nicht erwähnt und viele notwendige Schritte in der Umsetzung werden nicht berücksichtigt. Diese Versäumnisse sind nur bedingt als Nachlässigkeit der Autoren einzustufen – wie die Praxis zeigt, ist es alles andere als trivial, selbst einfache Refaktorisierungen für eine Programmiersprache wie Java oder C# richtig hinzubekommen. Auf alle Kombinationen und Varianten von einer Refaktorisierung möglicherweise betroffenen Sprachkonstrukte einzugehen, würde so ziemlich jeden vernünftigen Rahmen sprengen. Vollständige und korrekte Spezifikationen von Refaktorisierungen sind also nur im Rahmen der Entwicklung von Refaktorisierungswerkzeugen sinnvoll.

Leider sind auch die heute verfügbaren Refaktorisierungswerkzeuge alles andere als korrekt: Wie die Beispiele in **Kasten 1** zeigen, bringen sie selbst bei vergleichsweise einfachen Anwendungen Fehler in die refaktorisierten Programme ein, die im günstigen Fall zu einer Fehlermeldung durch den Compiler führen – im weitaus ungünstigeren Fall haben sie eine Bedeutungsänderung zur Folge, die erst im Zuge des Testens aufgedeckt wird oder aber gänzlich unbemerkt bleibt. Wenn also, wie in vereinzelten Untersuchungen (z. B. [Mur09]) herausgefunden, die tatsächliche Benutzung von Refaktorisierungswerkzeugen in der Praxis hinter den Erwartungen zurückbleibt, mag dies auch an mangelnder Qualität liegen.

Wege in die Fehlerfreiheit

Refaktorisierungswerkzeuge sind Programme, die andere Programme zur Ein- und Ausgabe haben. Damit fallen sie in die Kategorie der Metaprogramme, zu denen auch Compiler, Debugger und andere Programmierwerkzeuge zählen. Metaprogramme sind besonders schwer zu schreiben und zu debuggen, da sich Fehler charakteristischerweise vor allem mittelbar – in der Ausführung der von ihnen manipulierten Programme – offenbaren. Zugleich sind die Korrektheitsanforderungen an Programmierwerkzeuge besonders hoch – wer würde einen Compiler einsetzen wollen, von dem er nicht wüsste, dass er fehlerfrei ist? Doch während kommerziell genutzte Compiler heute getrost als korrekt angesehen werden können, gilt dies für Refaktorisierungswerkzeuge nicht.

Der offensichtlichste Weg, um Fehler in Refaktorisierungswerkzeugen aufzudecken, ist es zu versuchen, ihre Korrektheit zu beweisen (formale Verifikation). Entsprechende Ansätze gibt es zwar, aber sie leiden – wie andere Korrektheitsbeweise auch – daran, dass eine vollständige Formalisierung der Problemstellung (die in diesem Fall die weitgehende Formalisierung der Programmiersprache einschließt) nicht nur extrem aufwändig, sondern auch ausgesprochen fehleranfällig ist. Daher gibt weder ein Gelingen noch ein Scheitern des Beweises unmittelbar Aufschluss darüber, wie es um die Korrektheit eines Refaktorisierungswerkzeugs tatsächlich bestellt ist. Realistischerweise ist also der formale Beweis der Korrektheit eines Refaktorisierungswerkzeugs nicht zu erwarten – und liegt im übrigen für die allermeisten anderen Programmierwerkzeuge auch nicht vor.

Wo eine formale Verifikation nicht infrage kommt, greift man gewöhnlich auf das Testen zurück. Testen ist grundsätzlich gut dazu geeignet, mit begrenztem Aufwand die gröbsten Fehler schnell zu finden – der Rest bleibt dann den Benutzern und der Zeit überlassen. Allerdings ist das Testen

Ein besonders einfaches Beispiel für eine gängige Refaktorisierung, die nicht korrekt in Werkzeuge umgesetzt ist, ergibt sich aus dem Verschieben der Klasse B des folgenden Java-Programms in ein anderes Paket:

```
package a;
class A {
  B b;
}

package a;
class B {}
```

Damit die Klasse B aus A heraus zugreifbar bleibt, müsste ihre Zugreifbarkeit public sein, was aber nicht der Fall ist. Ein Refaktorisierungswerkzeug müsste das Verschieben also entweder ablehnen (wegen mangelnder Erfüllung einer Vorbedingung) oder die Zugreifbarkeit anpassen (als zusätzlich notwendigen Refaktorisierungsschritt). Stattdessen produzieren aber sowohl „Eclipse JDT" und „NetBeans" als auch „IntelliJ IDEA" fehlerhafte (nicht mehr kompilierende) Programme. Dass das Problem nicht einfach zu lösen ist, zeigt das folgende, leicht erweiterte Beispiel. Wenn in

```
package a;
class A extends B {
  B b;
  protected void m() {…}
  void n() { b.m(); }
}

package a;
class B {
  protected void m() {…}
}
```

wieder die Klasse B in ein anderes Paket verschoben wird, muss neben der Zugreifbarkeit der Klasse B auch die der Methode m() in B auf public erhöht werden. Das aber hat zur Folge, dass auch die Zugreifbarkeit von m() in A erhöht werden muss, da die Zugreifbarkeit einer überschreibenden Methode nicht unter der der überschriebenen liegen darf. Solche und ähnliche Kettenreaktionen können sich leicht durch das ganze Programm ziehen und machen aufwändige Analysen notwendig, die jedoch den meisten heutigen Refaktorisierungswerkzeugen fehlen.
Den beiden obigen Beispielen ist gemeinsam, dass fehlerhafte Umsetzungen der Refaktorisierungen zu nicht mehr kompilierenden Programmen führen. Es handelt sich also gewissermaßen um gutartige Fehler: gutartig, weil man sofort auf den Fehler hingewiesen wird und die Refaktorisierung rückgängig machen kann. Im Gegensatz dazu bleibt der eingeführte Fehler beim Verschieben von B in ein anderes Paket im Beispielprogramm

```
package a;
class A {
  B b;
  void n() { b.m("abc"); }
}

package a;
public class B {
  public void m(Object o) {…}
  void m(String s) {…}
}
```

vom Compiler unbemerkt – es ändert sich lediglich die Bindung des Aufrufs von m("abc") in A von B.m(String) nach B.m(Object). Diese Bedeutungsänderung kann höchstens durch Testen aufgedeckt werden.

Kasten 1: Fehlerhafte Refaktorisierungen.

von Refaktorisierungswerkzeugen nicht ganz so einfach: Hierzu müssen nämlich Beispielprogramme erstellt werden, die nicht nur das zu refaktorisierende Konstrukt enthalten (z. B. eine zu verschiebende Klasse wie in Kasten 1), sondern dies auch in einen Kontext einbetten, der mögliche Fehler des Refaktorisierungswerkzeugs provoziert. Da es in der Regel sehr viele solcher – selbst einfacher – Kontexte gibt (siehe wiederum das Beispiel in Kasten 1), ist das Erstellen einer auch nur halbwegs repräsentativen Test-Suite immer noch sehr aufwändig. Zu viele verbleibende Fehler sind die Folge.

Der „Refactoring Tool Tester"

Es geht aber auch wesentlich einfacher. Der Vorteil von Refaktorisierungswerkzeugen als zu testenden Programmen ist nämlich, dass die Tests gewissermaßen auf der Straße liegen: Jedes Programm muss nach seiner Refaktorisierung noch ausführbar sein und dabei definitionsgemäß ein unverändertes beobachtbares Verhalten aufweisen. Für beide Kriterien gibt es nun aber kostenlose Testorakel: Die Ausführbarkeit prüft der Compiler, indem er das Programm zu übersetzen versucht; das unveränderte Verhalten lässt sich durch einen Vergleich der Ausgaben des refaktorisierten Programms mit denen des Originalprogramms überprüfen (so genanntes *Back-to-back*-Testen). Letzteres ist dann besonders einfach, wenn das Verhalten des Originalprogramms in automatischen Tests (Regressionstests à la „JUnit") dokumentiert ist: In diesem Fall genügt es, diese Tests für das refaktorisierte Programm durchzuführen, um durch die Refaktorisierung eingebrachte Verhaltensänderungen (die ja einen Fehler in der Refaktorisierung bedeuten) finden zu können.

Dieser einfache Sachverhalt lässt sich zum systematischen Testen von Refaktorisierungswerkzeugen wie folgt ausnutzen: Programme, die durch Tests gut abgedeckt sind, werden als Probanden eingesetzt. Bei diesen wendet man eine Refaktorisierung an allen dafür infrage kommenden Stellen automatisch an und prüft nach jeder Anwendung, ob das refaktorisierte Programm noch kompiliert und ob es noch alle seine Tests besteht. Der Algorithmus eines solchen, vollautomatischen Refaktorisierungswerkzeug-Testers ist in Kasten 2 dargestellt. Dieser Algo-

```
teste(Refaktorisierung r)
| für jeden Probanden p im Repository
| | für jede Code-Stelle c in p, an der r anwendbar ist
| | | für jeden Parameter a von r an c
| | | | führe r mit a an c durch
| | | | wenn p nicht mehr kompiliert
| | | | | schreibe Fehlerlog mit (r, c, a)
| | | | sonst
| | | | | wenn die Testfälle von p nicht mehr durchlaufen
| | | | | | schreibe Fehlerlog mit (r, c, a)
| | | | mache r rückgängig
```

Kasten 2: Der Algorithmus des „Refactoring Tool Testers".

rithmus ist in dem von uns entwickelten und eingesetzten *Refactoring Tool Tester (RTT)* implementiert.

Abbildung 1 zeigt die Architektur des RTT. Die innerhalb der gestrichelten Kontur befindlichen Komponenten sind feste Bestandteile des RTT und bleiben immer gleich:

- Der Compiler stammt aus der integrierten Entwicklungsumgebung (IDE), in die der RTT integriert ist (Eclipse).
- Das Test-Framework wird zur Ausführung der Regressionstests der Probanden benötigt (in der Regel JUnit).
- Das Probanden-Repository ist im Wesentlichen ein Archiv (beispielsweise CVS), in dem der Quellcode der Probanden (beliebige Projekte mit ihren Regressionstests) hinterlegt werden kann.
  Die Probanden können immer gleich bleiben, sind jedoch eigenständige Projekte und nicht Bestandteile des RTTs. Die Refaktorisierungswerkzeuge sind die eigentlich zu testenden Programme und sind – genau wie die Probanden – eigenständige Projekte; sie müssen in der Regel nicht angefasst werden, insbesondere dann nicht, wenn sie über eine Plug-In-Schnittstelle in dieselbe IDE eingebunden werden können, in die auch der RTT integriert ist.

So ist für den Test eines konkreten Refaktorisierungswerkzeugs lediglich ein spezieller Adapter, der dieses in den RTT einbindet, zu schreiben. Ein solcher Adapter besteht typischerweise aus weniger als 50 Zeilen Code. Als Probanden kommen grundsätzlich alle quelloffenen Programme infrage. Je höher allerdings die Testabdeckung durch die mitgelieferten Test-Suiten ist, desto größer ist die Wahrscheinlichkeit, durch eine Refaktorisierung eingebrachte Verhaltensänderungen zu entdecken.

Der Vorteil des im RTT implementierten Verfahrens ist, dass man keinen einzigen Testfall für das Refaktorisierungswerkzeug selbst schreiben muss, da dieses ja nur indirekt getestet wird – die Tests, die Verhaltensänderungen der Probanden aufspüren, gehören zu den Probanden, nicht zur zu testenden Refaktorisierung. Der einzige manuelle Testaufwand entsteht, wie gesagt, beim Schreiben des Adapters, der das zu testende Refaktorisierungswerkzeug in den Testalgorithmus einbindet. Dabei ist die Bestimmung der Stellen in den Probanden, an denen eine Refaktorisierung ausgeführt werden kann, das geringere Problem – hierfür reicht ein Ablaufen des abstrakten Syntaxbaums der Probanden sowie die Anwendung eines einfachen Filters, der mögliche Anwendungsstellen aus den Knoten des Syntaxbaums herausfiltert. Weitaus schwieriger kann sich die Wahl der Parameter der Refaktorisierung gestalten, die einen Fehler offenbaren sollen. So ist es beispielsweise beim Test der *Rename*-Refaktorisierung nicht möglich, alle eventuellen neuen Namen eines umzubenennenden Programmelements durchzuprobieren (dies sind nämlich potentiell unendlich viele). Stattdessen sollten Namen ausgewählt werden, deren Verwendung Fehler provoziert, zum Beispiel solche, die schon im Programm vorhanden sind, sodass es zu Verdeckungen oder ungewollten Überschreibungen kommen kann. Im Extremfall (beispielsweise bei der *Rename*-Refaktorisierung) kann die Wahl der Parameter der expliziten Spezifikation von Testfällen gleichkommen, sodass durch den Ansatz wenig gewonnen ist. Im anderen Extrem entfällt dieser Aufwand jedoch vollständig, nämlich wenn eine Refaktorisierung parameterlos ist. Grundsätzlich ist der Aufwand um so geringer, je kleiner die Anzahl der Parameter und ihrer möglichen Werte ist.

Tatsächlich ergibt sich die Mächtigkeit dieses Testverfahrens vor allem aus der Anwendung einer Refaktorisierung mit gleichen (oder auf die gleiche Weise bestimmten) Parameterwerten in sehr vielen unterschiedlichen Kontexten, wie sie sich aus den Probanden ergeben. Je repräsentativer also die Probanden für die Anwendungsfälle der Refaktorisierung sind, desto aussagekräftiger ist die erzielte Testabdeckung.

In der Praxis hat sich dieser Ansatz des systematischen Testens von Refaktorisierungswerkzeugen als ausgesprochen effektiv erwiesen: Durch seinen konsequenten Einsatz bei allen von uns entwickelten Refaktorisierungswerkzeugen konnten neben zahlreichen Programmierfehlern auch eine ganze Reihe von Vorbedingungen für die Anwendung der Refaktorisierungen identifiziert werden, die uns bei unseren vorausgegangenen theoretischen Überlegungen entgangen waren. Kasten 3 nennt ein paar Beispiele für Vorbedingungen für die Refaktorisierung „Vererbung durch Delegation ersetzen" (vgl. [Fow99], [Keg08]), von denen zumindest einige nicht ganz offensichtlich sind.

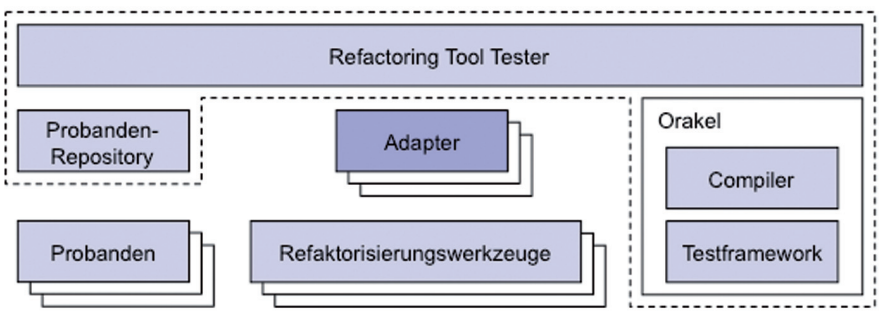

Abb. 1: Die Architektur des „Refactoring Tool Testers".

Der Bau von Refaktorisierungswerkzeugen als Disziplin

Anstatt sich damit zu beschäftigen, wie man Fehler in Refaktorisierungswerkzeugen findet, könnte man ja auch versuchen, den Bau von Refaktorisierungswerkzeugen – analog zum Compilerbau – als eigene Disziplin zu etablieren, die so gut verstanden ist, dass ihre Produkte gar keine oder nur wenige Fehler enthalten. Voraussetzung für das Entstehen einer solchen Disziplin ist aber, dass man analog zum Compilerbau eine Theorie zur Verfügung hat, derer sich die Definition von Refaktorisierungen bedienen kann und

Wie komplex die Vorbedingungen einer Refaktorisierung ausfallen können, kann man sich am Beispiel der Refaktorisierung „Vererbung durch Delegation ersetzen" (vgl. [Fow99]) vor Augen führen. Im Standardbeispiel für diese Refaktorisierung erbt zunächst eine Klasse Stack von einer Klasse Vector. Diese Vererbungsbeziehung, die in Java mit einer Subtyp-Beziehung einhergeht (sodass der Compiler überall da, wo eine Referenz vom Typ Vector erwartet wird, auch eine vom Typ Stack akzeptiert), gilt jedoch als inadäquat, da Stack von Vector auch Eigenschaften erbt, die für einen Stack gar nicht vorgesehen sind. Dies lässt sich vermeiden, indem man die Vererbung durch eine Objektkomposition ersetzt, in der dann genau die für Stack vorgesehen Methoden an Vector delegiert werden:

```
// vor der Refaktorisierung
public class Stack extends Vector {
        public Stack() {}
        public Object push(Object item) {
                addElement(item);
                return item;
        }
        ...
                // Methode size() wird
                geerbt
}
```

```
// nach der Refaktorisierung
public class Stack {
protected Vector delegatee;
public Stack() {
delegatee = new Vector();
}
public Object push(Object item) {
delegatee.addElement(item);
return item;
}
...
public int size() {
return delegatee.size();
}
}
```

Allerdings ist längst nicht immer möglich. Tatsächlich haben wir bei der Implementierung eines entsprechenden Refaktorisierungswerkzeugs eine stattliche Reihe von teilweise schwer zu prüfenden Vorbedingungen identifiziert (vgl. [Keg08]), von denen ich im Folgenden nur einige aufzähle. Entscheiden Sie selbst, welche davon als offensichtlich zu bezeichnen sind (keine davon findet sich übrigens in [Fow99]):

- Die Superklasse darf nicht abstrakt sein, da sonst keine Instanzen von ihr als Delegate gebildet werden können.
- Methodenaufrufe und Zuweisungen im Programm dürfen nicht verlangen, dass die Subklasse zur Superklasse zuweisungskompatibel ist, da die Subtypen-Beziehung mit der Vererbung entfernt wird; entsprechende Typtests (mit instanceof) dürfen ebenfalls nicht vorkommen.
- Klienten der Subklasse dürfen auf keine von der Superklasse geerbten Felder zugreifen, da Feldzugriffe in Java nicht delegierbar sind.
- Es dürfen keine in der Subklasse überschriebenen Methoden auf this in der Superklasse (oder auf Variablen, denen this von der Superklasse aus zugewiesen wurde) aufgerufen werden, da die dynamische Bindung an die überschreibenden Methoden nach der Refaktorisierung nicht mehr funktioniert.
- Es dürfen keine Tests auf Identität (mit ==) von Instanzen der Subklasse über Methoden der Superklasse angestoßen oder durchgeführt werden, da this in der Superklasse nach der Refaktorisierung ein anderes Objekt referenziert.
- Es dürfen keine synchronisierten, vor der Refaktorisierung von der Superklasse geerbten Methoden zusammen mit eigenen synchronisierten Methoden aufgerufen werden, da der Monitor nach der Refaktorisierung auf zwei Objekte aufgeteilt ist.

Kasten 3: Wie komplex Vorbedingungen werden können.

deren praktische Umsetzung die Konstruktion von Refaktorisierungswerkzeugen leitet. Die Implementierung von Refaktorisierungswerkzeugen in imperativer Form, wie sie durch die verbalen Beschreibungen der Refaktorisierungen à la [Fow99] nahegelegt wird und die in der Praxis von einer unüberschaubaren Menge von Fallunterscheidungen geprägt ist, hat sich jedenfalls nicht bewährt.

Als Basis einer geeigneten Theorie bieten sich verschiedene Kandidaten an.

Ein naheliegender Ansatz fasst Programme als Graphen auf und verwendet Graph-Transformationen, um eine Refaktorisierung durchzuführen. Dabei wird die Überführung eines (als Quelltext vorliegenden) Programms in einen Graphen vom Compiler geleistet: Der als Zwischenergebnis erzeugte abstrakte Syntaxbaum ist bereits ein Graph, der durch die Auflösung von Referenzen leicht so angereichert werden kann, dass die für eine Refaktorisierung benötigten Informationen vollständig aus ihm abgelesen werden können. Das Wissen über die Vorbedingungen und darüber, welche Transformationen für eine Refaktorisierung durchzuführen sind, wird dabei deklarativ in Form so genannter Graph-Transformationsregeln festgehalten, deren Anwendung zum gewünschten Ergebnis führen soll. Der Teufel steckt hier aber, wie so oft, im Detail: Um eine zielgerichtete Anwendung der Regeln zu garantieren, setzen die in der Literatur bisher beschriebenen Umsetzungen umfangreiche prozedurale Komponenten ein (vgl. [Men05]), die die Vorteile gegenüber imperativen Formulierungen mit ihren Fallunterscheidungen dahinschmelzen lassen.

Ein anderer, ebenfalls deklarativer Ansatz scheint dagegen erfolgversprechender: der der *constraint*-basierten Refaktorisierung. Auf ihm beruht eine ganze Reihe bereits praktisch eingesetzter Refaktorisierungswerkzeuge, zum Beispiel:

- die typbezogenen Refaktorisierungen von Eclipse, wie „Generalize Declared Type", „Use Supertype Where Possible", „Introduce Type Parameter" und „Infer Generic Type Arguments" (vgl. [Tip07])
- unsere eigenen Refaktorisierungen „Infer Type", „Replace Inheritance with Delegation", „Change Access Modifier", „Introduce Role Object" (vgl. [Keg07], [Keg08], [Ste09])

Die Funktionsweise des *Constraint*-Ansatzes ist in **Kasten 4** an einem einfachen Beispiel dargestellt. Dieser Ansatz fasst ein Programm als ein mathematisches System von Variablen auf, deren Werte durch Relationen – z. B. Gleichungen oder Ungleichungen – zueinander ins Verhältnis gesetzt werden. Die Variablen stehen für bestimmte Eigenschaften eines Programms, z. B. den Typ eines Feldes oder die Klasse,

Beim *constraint*-basierten Ansatz wird ein Programm intern als eine Menge von so genannten *Constraint*-Variablen dargestellt, die die Freiheitsgrade (Veränderlichen) einer Refaktorisierung repräsentieren. Jede *Constraint*-Variable steht dabei für eine bestimmte Eigenschaft eines Programmelements, z. B. für den Ort der Deklaration einer Klasse, oder für deren deklarierte Zugreifbarkeit (public, protected usw.). Im Programm

```
package a;
class A {
  B b;
}

package a;
class B {}
```

aus **Kasten 1** könnte beispielsweise eine Variable x1 für den Ort der Klasse B stehen und eine Variable x2 für die deklarierte Zugreifbarkeit von B. Die initialen Werte der beiden Variablen, die sich ebenfalls aus dem Programm ableiten lassen, sind package a (für x1, den Ort der Klasse B) bzw. default (für x2, die deklarierte Zugreifbarkeit von B). Wollte man den Ort von B oder ihre deklarierte Zugreifbarkeit im Rahmen einer Refaktorisierung ändern, würde sich das in einer Änderung des entsprechenden Variablenwerts widerspiegeln.

Allerdings unterliegen die im Rahmen einer Refaktorisierung zuweisbaren neuen Variablenwerte gewissen Einschränkungen, den *Constraints*, die durch das Programm in seiner bestehenden Form vorgegeben sind. So verlangt beispielsweise der Zugriff auf B aus der Klasse A, dass B mit für A ausreichender Zugreifbarkeit deklariert ist. Welcher Zugriffsmodifizierer von B (repräsentiert durch den Wert von x2) dazu mindestens notwendig ist, ist eine Funktion z der Variablen x1 und x3, wobei x3 für den Ort des Zugriffs (hier Klasse A in Paket a) steht. Das sich daraus ergebende *Constraint*

$x2 \geq z(x1, x3)$

ist für das bestehende Programm mit x2 = default und z(x1, x3) = default erfüllt. Der Wert von x2 darf zudem im Rahmen einer Refaktorisierung angehoben werden (also die Zugreifbarkeit von B erhöht werden), da das *Constraint* dadurch nicht verletzt wird, verringert werden darf er jedoch nicht. Wird hingegen der Ort von B in ein anderes Paket geändert, ändert sich der Wert von x1 und damit der Wert von z(x1, x3) zu public, sodass entweder der Wert von x2 auf public erhöht oder die Klasse A ebenfalls in das andere Paket werden verschoben muss, damit sich der Wert von z zurück in default ändert und das *Constraint* wieder erfüllt ist.

Die *Constraints* und ihre Variablen werden aus einem Programm mittels *Constraint*-Regeln abgeleitet. Die *Constraint*-Regel, aus der das obige *Constraint* hervorgegangen ist, hat beispielsweise die Form: „Wenn ein Programmelement von Ort X1 ein anderes Programmelement an Ort X2 referenziert und das referenzierte Programmelement die Zugreifbarkeit X3 hat, dann muss gelten: X3 >= z(X1, X2)". Eine weitere *Constraint*-Regel, die im zweiten Beispiel von **Kasten 1** zur Anwendung kommen würde, hat die Form: „Wird eine Methode mit deklarierter Zugreifbarkeit X1 durch eine andere Methode mit deklarierter Zugreifbarkeit X2 überschrieben, dann muss gelten: X2 >= X1". Dabei stehen die Platzhalter X1, X2 und X3 für beliebige *Constraint*-Variablen, die in einer Regelanwendung durch konkrete *Constraint*-Variablen (wie x2, x1 und x3 oben) ersetzt werden. Kommt eine konkrete *Constraint*-Variable in mehreren erzeugten *Constraints* vor, steht sie in allen für den gleichen Wert. Änderungen der Variablen, die im Zuge der Erfüllung eines *Constraints* notwendig werden, propagieren sich damit zu anderen *Constraints*. Auf diese Weise werden auch Kettenreaktionen, wie sie durch manche Refaktorisierungen notwendig werden (**vgl. Kasten 1**), ohne zusätzlichen Aufwand umgesetzt.

Kasten 4: *Der constraint-basierte Ansatz zur Refaktorisierung.*

zu der eine Methode gehört. Eine Änderung des Werts einer solchen Variablen steht für eine entsprechende Änderung des Programms. Die Relationen spiegeln die Regeln der Sprache (syntaktische Regeln, Binderegeln usw.) wider. Ihre Einhaltung stellt sicher, dass nur solche Wertänderungen von Variablen akzeptiert werden, die zu weiterhin kompilierenden Programmen mit unverändertem Verhalten führen. Ein aus einem vorhandenen Programm abgeleitetes *Constraint*-System, bestehend aus Variablen und Relationen, spannt somit einen Lösungsraum auf, dessen Elemente (konkrete Variablenbelegungen) allesamt korrekte Refaktorisierungen des Programms darstellen. Indem der Benutzer für bestimmte Variablen neue Werte vorgibt und für andere Variablen festlegt, ob sich ihr Wert im Zuge der Refaktorisierung ändern darf, legt er fest, welche Refaktorisierung konkret unter welchen Nebenbedingungen durchgeführt werden soll. Den Rest erledigt ein Algorithmus zur Lösung des *Constraint*-Systems.

Gegenüber imperativen Formulierungen hat der *Constraint*-Ansatz gleich mehrere Vorteile: Zum einen lassen sich mit ihm die Vorbedingungen einer Refaktorisierung und die zur Durchführung notwendigen Schritte auf die gleiche Weise ausdrücken, nämlich durch aus dem Programm abgeleitete *Constraints*. Ob ein *Constraint* eine Vorbedingung prüft oder einen Schritt berechnet, unterscheidet sich lediglich darin, wie viele seiner *Constraint*-Variablen veränderlich sind: Ist es nur eine, dann entscheidet die Frage, ob das *Constraint* durch die mit der geplanten Refaktorisierung verbundene Wertänderung der Variablen weiter erfüllt ist, darüber, ob eine Vorbedingung eingehalten ist. Sind es mehrere, kann das Ändern einer Variablen zwar zu einer *Constraint*-Verletzung führen, die aber unter Umständen durch die anschließende Änderung einer weiteren Variablen wieder geheilt werden kann (**siehe Kasten 4**). Solche abhängigen Änderungen stellen weitere, zur Durchführung der Refaktorisierung notwendige Schritte dar.

Ein damit unmittelbar zusammenhängender Vorteil ist der Umstand, dass die im Rahmen einer beabsichtigten Refaktorisierung unerfüllbaren *Constraints* (zusammen mit den Regelanwendungen, aus denen sie hervorgegangen sind) Aufschluss darüber geben, warum die Refaktorisierung nicht durchführbar ist. Allerdings ist dabei zu bedenken, dass die verschiedenen Pfade, die ein *Constraint*-Lösungs-

algorithmus exploriert, nicht immer an den gleichen *Constraints* scheitern, sodass die Schuldigen nicht immer eindeutig identifizierbar sind. In solchen Fällen wäre es aber immer noch möglich, dem Benutzer eine interaktive Erklärungskomponente zur Verfügung zu stellen, mit der er die Pfade selbst durchspielen kann und so die Unmöglichkeit seines Refaktorisierungsvorhabens erkennt.

Ein weiterer Vorteil ist, dass die Regeln, nach denen *Constraints* aus einem vorliegenden Programm generiert werden (und die im Wesentlichen die Syntax- und Semantikregeln einer Programmiersprache repräsentieren), inhärent modular sind: Gibt die Menge der *Constraint*-Regeln eine Sprachdefinition erst einmal vollständig wieder, dann verlangt keine – wie auch immer geartete – Kombination von Sprachkonstrukten eine Sonderfallbehandlung, wie sie die imperativen Formulierungen von Refaktorisierungen durchziehen und deren Unvollständigkeit die Ursache vieler Fehler von Refaktorisierungswerkzeugen ist.

Ein vierter Vorteil ist, dass sich die *Constraint*-Regeln, von denen die Korrektheit der Refaktorisierungen abhängt, leicht und unabhängig von einer konkreten Refaktorisierung testen lassen: Alle aus einem Programm mittels einer zu testenden Regel abgeleiteten *Constraints* müssen mit den ebenfalls aus ihm abgeleiteten Variablenwerten erfüllt sein – ansonsten ist die Regel nicht korrekt. Auf ähnliche Weise lässt sich eine Menge von *Constraint*-Regeln als vollständig testen: Alle Variablenzuweisungen, die alle *Constraints* erfüllen, müssen korrekten (kompilierbaren und ihre Tests erfüllenden) Programmen entsprechen oder es fehlen *Constraints*, die die Variablenwerte weiter einschränken.

Nicht zuletzt ist ein großer Vorteil des *Constraint*-Ansatzes der Fakt, dass seine *Constraint*-Regeln wiederverwendbar sind: Für eine bestimmte, in ein Werkzeug zu gießende Refaktorisierung müssen im Wesentlichen nur die Regeln ausgewählt werden, die die von der Refaktorisierung direkt oder indirekt zu ändernden Eigenschaften der Programmelemente beschränken. Für die Lösung der *Constraints*, die durch die Anwendung der Regeln aus einem Programm erzeugt werden, können Standardalgorithmen eingesetzt werden.

Die Schwierigkeiten des *Constraint*-Ansatzes bestehen darin, die Regeln, die für eine Refaktorisierung relevant sind, vollständig zu bestimmen und bei ihrer konkreten Anwendung im einzelnen Fall nur die *Constraints* zu erzeugen, die für eine geplante Refaktorisierung relevant sind. Für ersteres leistet der oben beschriebene RTT gute Dienste, an letzterem muss noch geforscht werden.

Zukünftige Herausforderungen

Wenn Refaktorisierung als eigenständige Disziplin wahrgenommen und anerkannt werden will, müssen Wege gefunden werden, wie der Bau von Refaktorisierungswerkzeugen standardisiert werden kann – nur so wird sich auf Dauer die notwendige Qualität erzielen lassen. Anstatt – wie bisher – die Vorbedingungen und die zur Durchführung notwendigen Schritte für jede Refaktorisierung getrennt zu formulieren und anschließend mit einigem Aufwand in den Kontext der jeweiligen Entwicklungsumgebungen einzubetten, wäre es hilfreich, wenn Refaktorisierungswerkzeuge nach dem Baukastenprinzip aus erprobten Komponenten zusammengestellt werden könnten und wenn sie von den Entwicklungsumgebungen stets die gleiche Infrastruktur benötigen. Aufgrund seiner inhärenten Modularität verspricht der *constraint*-basierte Ansatz, ein wichtiger Schritt in diese Richtung zu sein.

Eine weitere große Herausforderung ist die Entwicklung sprachübergreifender Refaktorisierungswerkzeuge. Die zunehmende Popularität von Plattformen wie der „Java Virtual Machine" oder „.NET" in der kommerziellen Softwareentwicklung führt dazu, dass Programme immer häufiger aus Komponenten zusammengesetzt werden, die in verschiedenen Sprachen geschrieben wurden. Refaktorisierungen, wie das Umbenennen von Programmelementen oder die Änderung von Typdeklarationen, machen aber vor Komponentengrenzen nicht halt: Ein Refaktorisierungswerkzeug, das die Vorbedingungen und notwendigen Schritte, die sich aus in anderen Sprachen geschriebenen Programmteilen ergeben, ignoriert, sodass die verschiedenen Komponenten hinterher nicht mehr korrekt zusammenspielen, ist nur eine halbe Sache. Auch hier sollte der *Constraint*-Ansatz bestehende Probleme lösen können. Wo genau dabei die Grenzen liegen, untersuchen wir derzeit in einem zweijährigen Forschungsprojekt, das von der Deutschen Forschungsgemeinschaft (unter der Kennung „Ste 906/4-1") gefördert wird (siehe: www.feu.de/ps/prjs/CLaRe).

Fazit

Refaktorisierungen und die Werkzeuge, die sie automatisieren, versprechen einen wichtigen Beitrag zur Softwareentwicklung zu leisten. Damit dieses Versprechen eingelöst werden kann, müssen sie korrekt sein, d. h. nicht nur die gewünschten Änderungen im Programm durchführen, sondern auch garantieren, dass sich durch diese Änderungen die Bedeutung des Programms nicht ändert. Letzteres sicherzustellen ist keine leichte Aufgabe, aber wenn Refaktorisierungswerkzeuge zukünftig in großem Umfang eingesetzt werden sollen, muss sie unbedingt angegangen werden. ■

Literatur & Links

[Fow99] M. Fowler, Refactoring, Addison-Wesley, 1999

[Keg07] H. Kegel, Constraint-basierte Typinferenz für Java 5, Diplomarbeit, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen

[Keg08] H. Kegel, F. Steimann, Systematically refactoring inheritance to delegation in Java, in: Proc. of International Conference on Software Engineering (ICSE), 2008, 431-440

[Men05] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens, Formalizing refactorings with graph transformations, Journal of Software Maintenance and Evolution 17:4, 2005, 247-276

[Mur09] E.R. Murphy-Hill, C. Parnin, A.P. Black, How we refactor, and how we know it, in: International Conference on Software Engineering (ICSE), 2009, 287-297

[Ste09] F. Steimann, A. Thies, From public to private to absent: Refactoring Java programs under constrained accessibility, in: Proc. of European Conference on Object-Oriented Programming (ECOOP), 2009, 419-443

[Tip07] F. Tip, Refactoring using type constraints, in: Proc. of International Symposium on Static Analysis (SAS), 2007, 1-17