mehr zum thema:
scala.ifs.hsr.ch
r2.ifs.hsr.ch

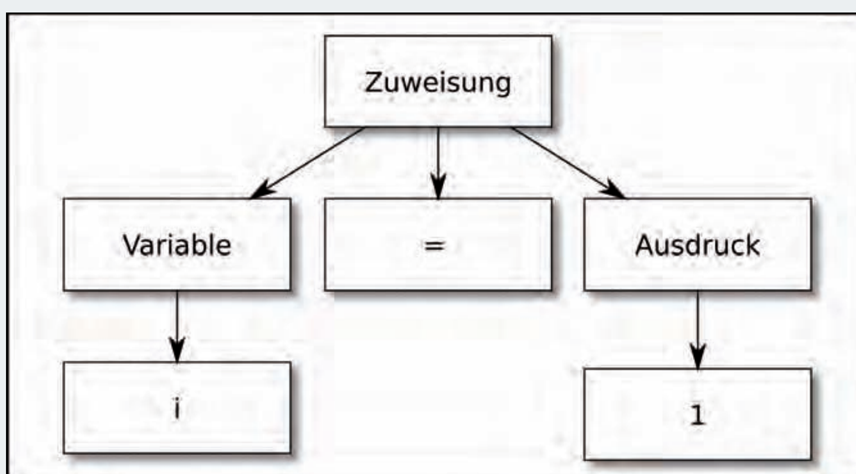
REFAKTORISIERUNGSWERKZEUGE: EIN BLICK HINTER DIE KULISSEN

Weshalb existieren nicht für alle Sprachen so gute Refaktorisierungsautomatisierungen, wie wir sie von Java-Entwicklungsumgebungen kennen? Dieser Frage gehen wir in diesem Beitrag nach. Wir führen Sie durch die typischen Abläufe in einem Refaktorisierungswerkzeug und erläutern Herausforderungen bei der Implementierung. Dank der Umsetzung von Werkzeugen für verschiedenste Programmiersprachen können wir aus erster Hand über funktionierende Lösungsansätze berichten.

Refaktorisieren ist ein essenzieller Bestandteil der täglichen Arbeit eines jeden Softwareentwicklers. Das Spektrum reicht vom simplen Umbenennen einer lokalen Variablen bis zur aufwändigen Restrukturierung ganzer Architekturen. Dabei setzen sich umfangreiche Änderungen häufig aus mehreren kleineren Refaktorisierungen zusammen. Um den gesamten Arbeitsfluss nicht zu stören, dürfen die kleineren Teilschritte den Entwickler nicht von seinem Primärziel, den umfangreichen Änderungen, ablenken. Glücklicherweise lassen sich diese mit integrierten Entwicklungsumgebungen (IDEs) häufig automatisiert in einem einzigen Schritt erledigen.

Das Refaktorisieren ist nicht bloß ein Eckpfeiler agiler Softwareentwicklung, sondern generell ein wichtiges Mittel, um den Programmcode in Form zu halten. Je kleiner der Aufwand ist, um eine Refaktorisierung anzuwenden, desto geringer ist die Hemmschwelle, diese auch einzusetzen. Zudem senkt die Automatisierung durch qualitativ hochwertige Werkzeuge die Fehlerquote.

Ein abstrakter Syntaxbaum, der Informationen über die Struktur des Programms enthält, wird üblicherweise von einem Parser erstellt. Dabei wird die interne Hierarchie der Programmbausteine abgebildet:



Die Knoten des Baumes sind mit weiteren Informationen – wie Quelltext-Position, Typen und Symbol-Referenzen – versehen. In der Regel wird ein AST für statische Analyse, beim Kompilieren oder Interpretieren verwendet. Beim Refaktorisieren wird die statische Analyse ebenfalls basierend auf dem AST durchgeführt. Dabei nutzt man vor allem die Möglichkeiten zur Suche nach spezifischen Knoten und zur Ermittlung von selektierten Elementen.

Kasten 1: *Aufbau eines abstrakten Syntaxbaums (AST).*

die autoren

Thomas Corbat
(E-Mail: tcorbat@hsr.ch)
ist Projektleiter am Institut für Software der HSR Hochschule für Technik, Rapperswil (Schweiz). Seine primäre Tätigkeit liegt im Bereich Werkzeugentwicklung für C++.

Prof. Peter Sommerlad
(E-Mail: psommerl@hsr.ch)
leitet das IFS Institut für Software der HSR Hochschule für Technik, Rapperswil. Er ist Koautor der Bücher „Pattern-Oriented Software Architecture Volume 1: A System of Patterns" und „Security Patterns: Integrating Security and Systems Engineering".

Mirko Stocker
(E-Mail: me@misto.ch)
ist Projektleiter am Institut für Software der HSR Hochschule für Technik, Rapperswil, wo er Refaktorisierungswerkzeuge für moderne Sprachen wie Scala entwickelt.

Nun stellt sich die Frage, warum eine gute Refaktorisierungsunterstützung nur in wenigen Entwicklungsumgebungen anzutreffen ist. Bei der Implementierung verschiedenster Refaktorisierungswerkzeuge stellten wir fest, dass deren Implementierung komplexer ist, als es auf den ersten Blick zu erwarten wäre.

Im Folgenden laden wir Sie zu einem Besuch hinter die Kulissen ein: Wir durchleuchten die verschiedenen Komponenten eines typischen Refaktorisierungswerk-

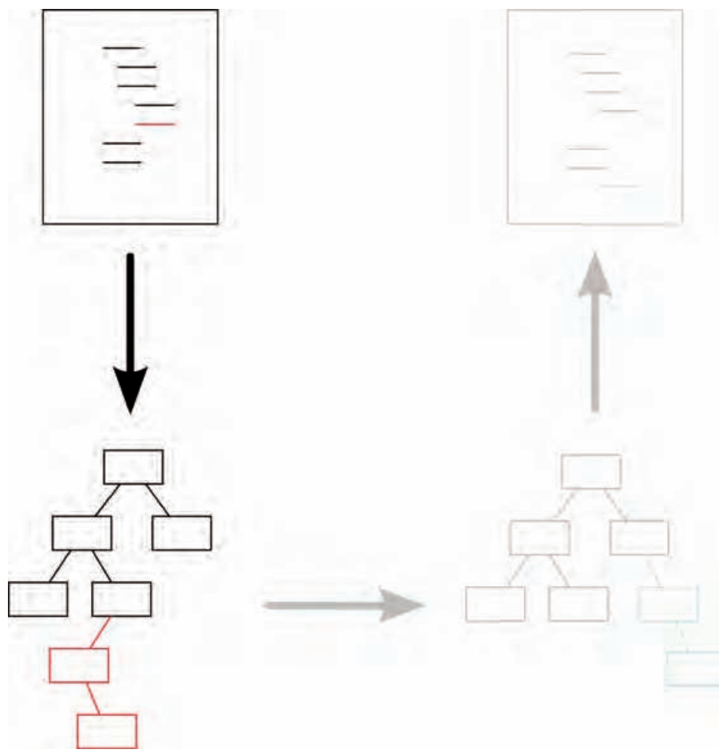


Abb. 1: Der Quellcode wird vor der Refaktorisierung in einen AST überführt.

zeugs und erläutern mögliche Probleme sowie Lösungen, um diese zu überwinden.

Szene 1: Ausgangslage

Ursprünglich wird beim Refaktorisieren kein Werkzeug verwendet. Bill Opdykes und Martin Fowlers Weg bereitende Beschreibungen von Refaktorisierungen (vgl. [Opd92], [Fow99]) sind Schritt-für-Schritt-Anleitungen, die von Hand abgearbeitet werden. Jedoch können bereits sehr einfache Refaktorisierungen, wie zum Beispiel „Umbenennen", nicht zufriedenstellend mit einer primitiven „Suchen&Ersetzen"-Aktion durchgeführt werden. Unbeabsichtigte Stellen könnten fälschlicherweise verändert werden – im schlimmsten Fall ohne offensichtliche Auswirkungen. Ein gutes Refaktorisierungswerkzeug muss daher – ähnlich wie ein Compiler – mehr über die Struktur des Programmes ermitteln können.

Bei der Übersetzung in ein ausführbares Programm wird der Quellcode in einem ersten Schritt von einem Parser analysiert. Die dabei erzeugte Zwischenrepräsentation ist üblicherweise ein abstrakter Syntaxbaum (*Abstract Syntax Tree – AST*, siehe Kasten 1). Dieser enthält verschiedenste Informationen über die Struktur des Programms, die auch für eine Refaktorisierung benötigt werden (siehe Abb. 1).

Für ein Refaktorisierungswerkzeug gibt es mehrere Möglichkeiten, an einen solchen AST zu gelangen:

1. Im einfachsten Fall, wird dieser bereits durch die umgebende IDE zur Verfügung gestellt. Das hat den Vorteil, dass Abhängigkeiten verschiedener Quellcode-Dateien bereits aufgelöst sind und weitere für die Refaktorisierung relevante Informationen vorhanden sind (vgl. [Ecl10]). Die Verwendung einer, oft nicht über eine öffentliche Programmierschnittstelle zugänglichen, Kernkomponente der IDE bildet eine starke Kopplung und macht das Refaktorisierungswerkzeug anfällig für Änderungen.
2. Falls keine adäquate Struktur in der IDE vorhanden ist, kann auf ein externes Softwaremodul zurückgegriffen werden, das die syntaktische Analyse und die Generierung des AST übernimmt, beispielsweise von einem Compiler. Diese sind jedoch oft weniger auf die exakte Repräsentation der textuellen Struktur ausgelegt, sondern häufig bereits für die spätere Verwendung normalisiert (vgl. [JRu]). Ein Beispiel einer solchen Normalisierung ist die Konkatenation von mehreren Zeichenketten, die in einem AST zusammengefasst als ein einziger Knoten abgebildet werden könnte. Zusätzlich entstehen bei der Verwendung von externen Komponenten ebenfalls Abhängigkeiten.
3. Als dritte Möglichkeit bleibt die Implementierung einer eigenen Komponente zur Analyse des Programms. Das ist ein sehr aufwändiges Vorhaben, das jedoch die bestmögliche Abstimmung auf die Anforderungen des Refaktorisierungswerkzeugs garantiert. Beispielsweise enthält ein AST eines Compilers keine genaue Informationen über die Formatierung des Quellcodes. Bei der Verwendung eines bestehenden AST müssen die fehlenden Angaben anderweitig ermittelt werden.

Szene 2: Analyse

Mit dem AST besitzen wir eine Repräsentation des Quellcodes für tiefer gehende Analysen. Wie bereits erwähnt, benötigt ein Refaktorisierungswerkzeug mehr Informationen über das Programm als nur dessen Struktur. Beispielsweise muss ein Methodenaufruf der entsprechenden Methodendeklaration zugeordnet werden können. Das ist vor allem bei dynamisch typisierten Sprachen eine große Herausforderung, da Typen vom Programmierer nicht angegeben werden müssen und in diesem Falle erst zur Laufzeit bekannt sind. Wir illustrieren dies an folgendem Programm-Beispiel in der Programmiersprache Ruby:

```
class Lastwagen
  def laden
    ...
  end
end
class Akkumulator
  def laden
    ...
  end
end
def ausliefern(ding)
  ding.laden
  ...
end
```

In dem Beispiel wird der Methodenname laden in zwei Klassen definiert: In beiden Fällen ist dieser nicht sehr aussagekräftig.

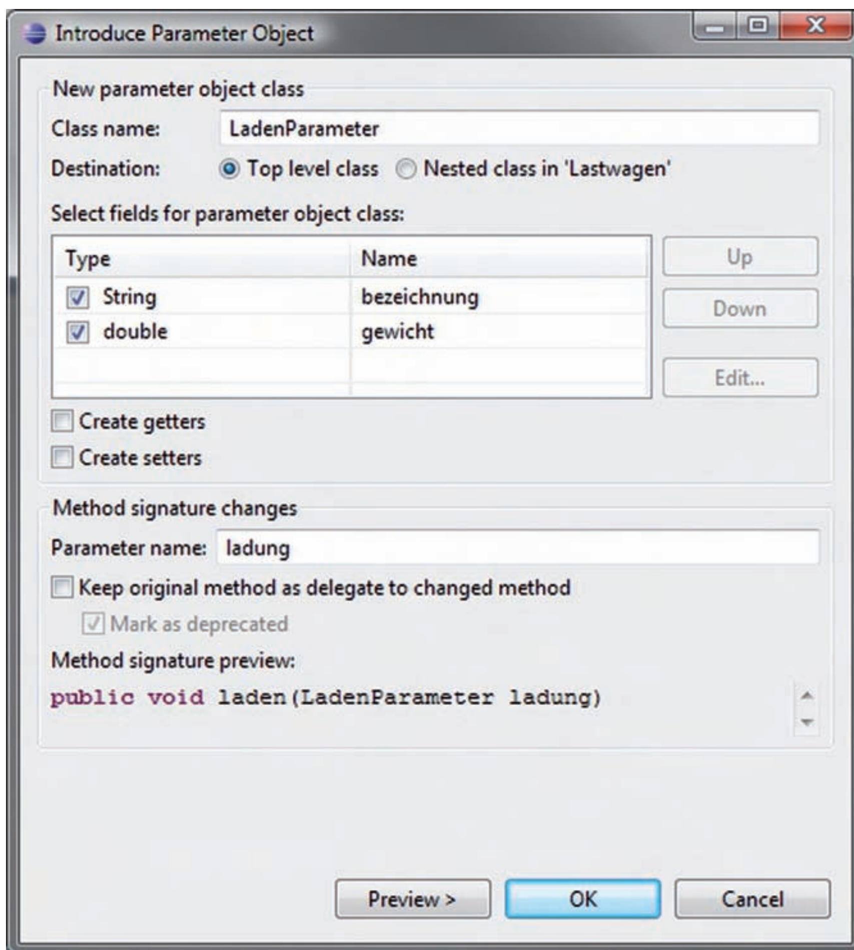


Abb. 2: Eingabemaske zur Konfiguration einer Refaktorisierung.

Entsprechend möchten wir die Methode laden der Klasse Lastwagen umbenennen. Die Bezeichnung beladen trifft besser zu – daher verwenden wir die Refaktorisierung „Methode Umbenennen“. Dazu wird der Name der Methodendeklaration markiert und die Refaktorisierung ausgelöst. Nun steht das Automatisierungswerkzeug – im Gegensatz zum Programmierer – vor einem Problem: Durch die dynamische Typisierung ist nicht eindeutig bestimmbar, welche Aufrufe der Methode laden zur umzubenennenden Deklaration gehören.

Für dieses Problem sind verschiedenste Lösungsansätze denkbar:

- Das Refaktorisierungswerkzeug kann die Mehrdeutigkeiten nicht auflösen, jedoch erkennen. Sobald solche gefunden werden, erhält der Programmierer eine Meldung und es wird keine Refaktorisierung vorgenommen. Das ist nicht besonders hilfreich, stellt jedoch sicher, dass das Programm semantisch korrekt bleibt.
- Alternativ können alle gleichen (Methoden-)Namen zusammen umbenannt werden. Dadurch bleibt zwar die Lauffähigkeit des Programms unverändert, jedoch schränkt das den Nutzen der Refaktorisierung ein. Das wäre auch in obigem Beispiel von geringem Nutzen und ist auch nur dann möglich, wenn keine Methoden aus Bibliotheken betroffen sind.
- Alle möglichen Aufrufe werden gesammelt und der Benutzer wird in die Entscheidung, welche umbenannt werden, miteinbezogen. Das stellt einen Kompromiss zwischen den beiden obigen Lösungen dar (vgl. [RDT]).
- Der komfortabelste Ansatz ist der Einsatz von so genannter Typinferenz. Dabei werden die vom Programmierer nicht spezifizierten Typen algorithmisch hergeleitet, ohne dass das Programm ausgeführt werden muss. Leider ist diese Art der Bestimmung nicht trivial umzusetzen und führt nicht zwingend zu einer nützlichen Lösung (vgl. [Pep]).

Ähnliche Probleme existieren beim Refaktorisieren von Code, der mit *Reflection* arbeitet.

Szene 3: Konfiguration

Mit den Analysemöglichkeiten sind wir an einem Punkt angelangt, an dem der Benutzer das weitere Vorgehen bestimmt. Die Entwicklungsumgebung kann dem Programmierer nun die möglichen Refaktorisierungen anbieten. Die Auswahl ist vom Kontext der aktuellen Selektion und der Text-Cursor-Position abhängig. Beispielsweise ist die Refaktorisierung „Methode Extrahieren“ nicht anwendbar, wenn der Name einer Klasse selektiert ist.

Sobald der Entwickler eine konkrete Refaktorisierung ausgewählt hat, führt das Werkzeug eine detaillierte Prüfung der Vorbedingungen durch. Davon werden die für die Refaktorisierung benötigten Parameter abgeleitet und dem Benutzer präsentiert. Durch den aktuellen Kontext können Einschränkungen in der Verwendung der Refaktorisierungen entstehen. Ein Beispiel dafür ist das bereits erwähnte Umbenennen von mehrdeutigen Methodennamen in dynamisch typisierten Sprachen.

Der Programmierer konfiguriert, entsprechend seinen Absichten, die ausgewählte Refaktorisierung. Die Möglichkeiten hängen stark von der Refaktorisierung ab: Sehr einfache Änderungen können ohne weitere Konfiguration direkt im Editor der Entwicklungsumgebung vorgenommen werden, während andere vielfältige Einstellungen erlauben (siehe Abb. 2).

Die Eingaben werden fortlaufend auf mögliche Probleme geprüft. Treten beispielsweise Namenskonflikte auf, wird eine Rückmeldung gegeben. Leider können nicht alle Nachbedingungen in Echtzeit geprüft werden, da dies zu ressourcenintensiv wäre.

Szene 4: Manipulation

Wenn die Konfiguration abgeschlossen ist, steht die Refaktorisierung vor dem aufwändigsten Teil ihrer Aufgabe: der Ermittlung und Durchführung der notwendigen Änderungen an der Programmstruktur. Hierzu wird erneut auf die zu Grunde liegende Repräsentation, den AST, zurückgegriffen. Am Beispiel von „Temporäre Variable entfernen“ erläutern wir diese beiden Schritte. Die Ausgangslage ist folgende Methodendefinition:

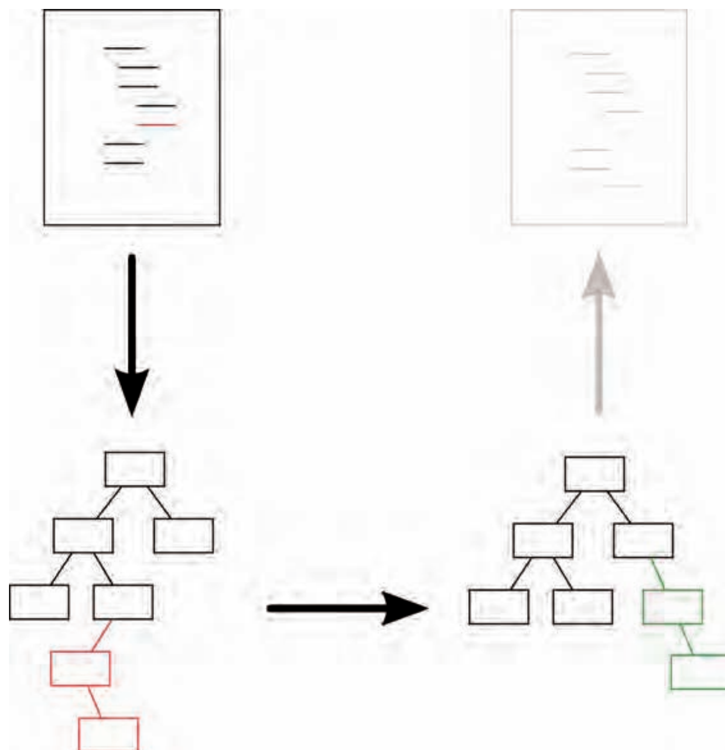


Abb. 3: *Die Refaktorisierung manipuliert den AST.*

```
def volumen
  grundflaeche = laenge * breite
  grundflaeche * hoehe
end
```

Unser Ziel ist es, die temporäre Variable grundflaeche zu entfernen und alle Verwendungen durch den Initialisierungsausdruck der Definition wie folgt zu ersetzen:

```
def volumen
  laenge * breite * hoehe
end
```

Die Refaktorisierung besteht aus zwei Änderungen:

- Die Definition von grundflaeche löschen.
- Die Verwendung von grundflaeche ersetzen.

Diese Änderungen können auf zwei Arten realisiert werden:

1. *Durch eine direkte Manipulation der Repräsentation:* Hierbei werden Knoten im Baum gelöscht bzw. ersetzt oder neue Knoten eingefügt (siehe Abb. 3). Das Resultat ist eine abstrakte Repräsentation des refaktorisierten Programms.
2. *Eine Beschreibung der Änderung („Lösche Knoten X"):* Bei dieser Variante gehen keine Informationen über die ursprüngliche Struktur verloren – im Gegensatz zu der direkten Manipulation. Nachfolgende Schritte können so Rückschlüsse auf die vorgenommenen Änderung machen. Das ermöglicht beispielsweise eine detaillierte Ansicht der Teilmodifikationen für den Benutzer. Dieser Ansatz ist jedoch komplexer umzusetzen, wenn eine Refaktorisierung verschachtelte Änderungen vornehmen muss.

Für Refaktorisierungen, die keine strukturellen Veränderungen aufweisen, wie zum Beispiel „Umbenennen", ist dieser Schritt signifikant einfacher, da keine Manipulation am AST vorgenommen werden muss.

Szene 5: Generierung

Nach der Manipulation der abstrakten Repräsentation muss diese zurück in konkreten Quellcode umgewandelt werden. Ausgehend von einem modifizierten AST kann das Werkzeug diesen auf einfache Weise generieren (siehe Abb. 4). Dazu wird der abstrakte Syntaxbaum traversiert und jeder Knoten wird in eine zugehörige Textform überführt. Gepaart mit der Formatierungsunterstützung der Entwicklungsumgebung ergibt dies einen gut lesbaren Quellcode.

Problematisch bei dieser Lösung ist die unnötige komplette Neugenerierung des gesamten Programms. Dadurch verlieren wir sämtliche Informationen über die ursprüngliche Formatierung. Dieses uner-

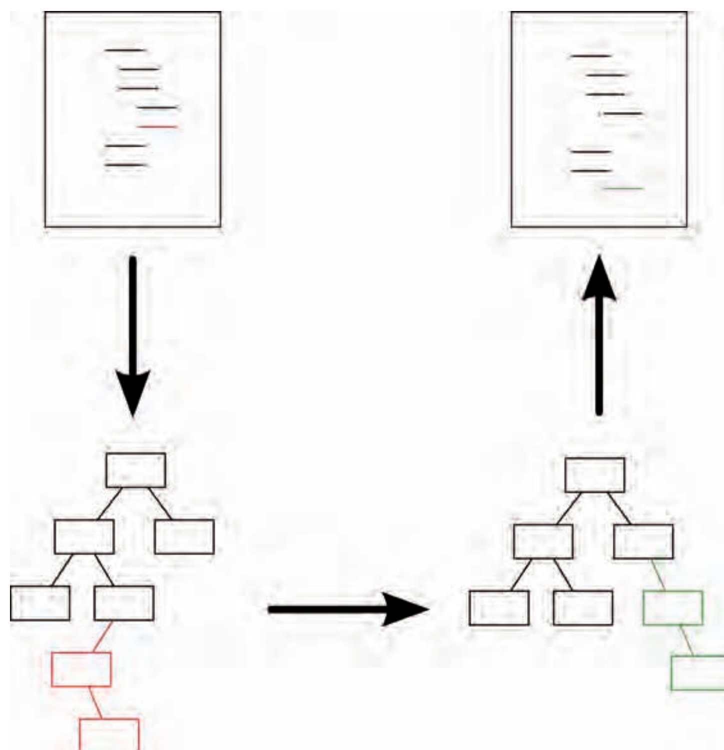


Abb. 4: *Generierung des Quellcodes anhand der modifizierten Struktur.*

wünschte Verhalten ist in den meisten Fällen sehr störend und kann im Extremfall die automatisierte Refaktorisierungsunterstützung unbrauchbar machen.

Eine bessere Variante der Quellcode-Anpassung beschränkt sich auf die effektiv geänderten Programmteile. Das erfordert jedoch detaillierte Kenntnisse der durchgeführten Änderungen. Hierbei ist es von Vorteil, wenn nicht nur die finale Repräsentation, sondern eine Beschreibung der Änderungen vorhanden ist. Wir unterscheiden drei Fälle von Manipulationen:

- *Löschen:* Der zugehörige Codeteil wird entfernt.
- *Einfügen:* Neu generierter Code wird an der definierten Position eingefügt.
- *Ersetzen:* Hierbei handelt es sich um eine Kombination aus Löschen und Einfügen, bei welcher der ersetzende Code ebenfalls generiert wird.

Diese Manipulationen minimieren die

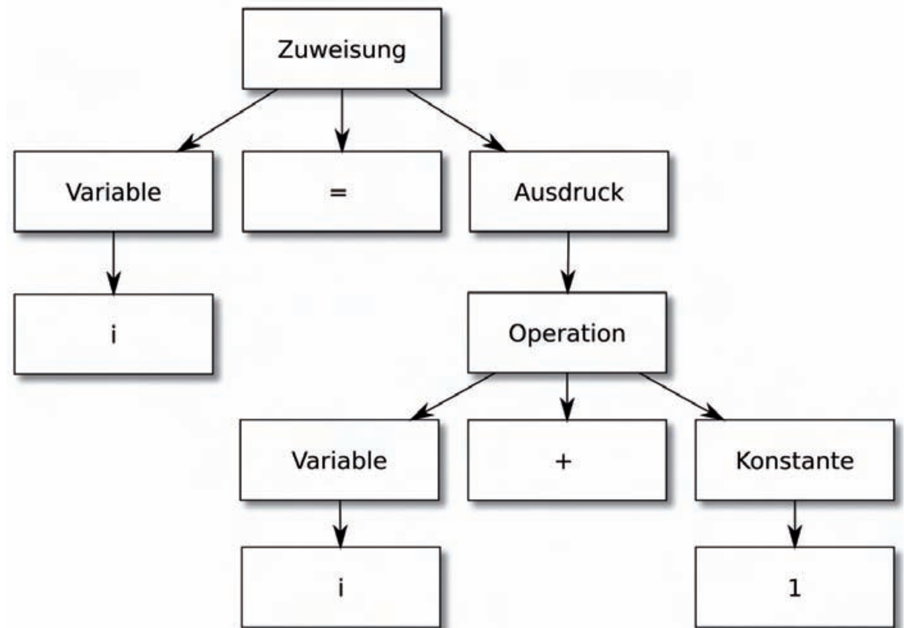


Abb. 5: *Der AST für die Anweisungen „i += 1“ und „i = i + 1“.*

Die Behandlung von Kommentaren beim Refaktorisieren ist eine besondere Herausforderung. Dabei gibt es zwei grundlegende Probleme:

1. Aus Sicht einer Programmiersprache werden Kommentare häufig mit Leerzeichen gleichgestellt (C++) und haben syntaktisch keine Bedeutung. Bereits im ersten Schritt der Übersetzung, der lexikalischen Analyse, werden sie ignoriert. Folglich werden sie auch in einer Repräsentation wie dem AST nur selten abgebildet.
2. Zusätzlich besteht ein weiteres Problem in der Zuordnung von Kommentaren zu Programmelementen. Da dem Refaktorisierungswerkzeug die Absicht hinter einem Kommentar verborgen bleibt, kann nicht deterministisch bestimmt werden, welcher Teil des Quellcodes aus Programmierersicht kommentiert wird.

Wir halten es für wichtig, in einem benutzerfreundlichen Refaktorisierungswerkzeug die Kommentare des Programmierers trotz der genannten Schwierigkeiten zu erhalten. Wenn diese nicht im AST abgebildet werden, gehen sie bei der Neugenerierung verloren. Aus mehreren Lösungsansätzen kristallisierte sich folgender als flexibelster heraus.

Statt die Kommentare bei der lexikalischen Analyse des Programmcodes wie Leerzeichen zu überspringen, werden diese ebenfalls durch eine entsprechende Regel erkannt. Jedoch werden die Kommentare nicht als so genannte Tokens an den Parser weitergegeben, sondern in einer Liste für die spätere Weiterverarbeitung eingesammelt. Auch wenn ein existierendes Werkzeug zur Analyse eines Programms dies nicht von sich aus unterstützt, ist die Änderung am Erkenner meist nur geringfügig, da Strukturen zur Erkennung von Kommentaren üblicherweise bereits vorhanden sind (vgl. [RDT]).

Nachdem alle Kommentare gesammelt wurden, müssen diese mit den entsprechenden Knoten im AST assoziiert werden. Leider werden für eine korrekte Zuordnung Informationen über die Absicht des Kommentars benötigt. Da wir nicht davon ausgehen, dass eine entsprechende Analyse erfolgen kann, greifen wir auf die relativen Positionen zurück. Die entsprechende Heuristik basiert auf folgenden Annahmen:

Allein stehende Kommentare betreffen den nachfolgenden Quellcode:

```
//Definition von i
int i = 0;
```

Kommentare hinter einem Programm-Element beziehen sich auf diese:

```
int i = 0; //i ist 0
```

Alleinstehende Kommentare ohne nachfolgenden Quellcode, beschreiben das vorhergehende Element. Mit diesem Ansatz stellen wir sicher, dass keine Kommentare verloren gehen. Eine detaillierte Beschreibung der Zuordnungsheuristik ist in [Som08] beschrieben.

Kasten 2: *Die Behandlung von Kommentaren.*

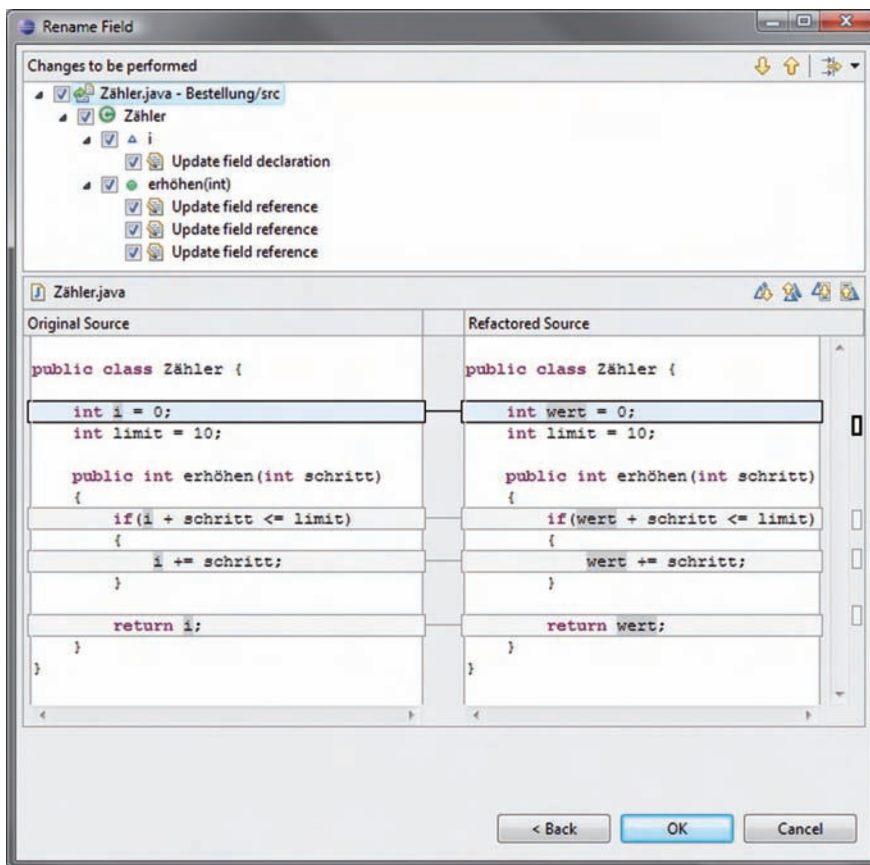


Abb. 6: Vorschau der Änderungen.

unbeabsichtigte Neuformatierung auf die veränderten Stellen. Es gibt jedoch noch weitere Unzulänglichkeiten, die sich nicht allein durch den beschriebenen Ansatz lösen lassen. Beispiele dafür sind die korrekte Behandlung von Kommentaren (siehe Kasten 2) und die bereits erwähnten Normalisierungen im AST, bei denen ursprünglich unterschiedliche Anweisungen zur gleichen Abbildung führen. Abhängig von der Programmiersprache können folgende zwei Anweisungen – trotz unterschiedlicher Syntax – semantisch identisch sein (vgl. [JRu]):

```
i += 1
i = i + 1
```

Beide Anweisungen können dieselbe Repräsentation im abstrakten Syntaxbaum besitzen (siehe Abb. 5). Bei der Neu-Generierung ist ohne weitere Kontextinformationen unklar, welches die ursprüngliche Textform war.

Spezialfälle wie diese erschweren die Generierung und sprechen für den Einsatz einer reichhaltigeren Datenstruktur, die nicht die abstrakte Syntax, sondern die Struktur des Quellcodes in den Vordergrund stellt – also den Syntaxbaum. In der Praxis ist dieser jedoch selten in der gewünschten Form anzutreffen und muss selbst geschaffen werden. Die Vor- und Nachteile davon wurden bereits im Abschnitt „Szene 1“ erwähnt.

Szene 6: Abschluss

Aufgeteilt in einzelne Änderungen – jeweils versehen mit einer kurzen Beschreibung – steht das Resultat der Refaktorisierung jetzt zur Verfügung. Sollte die Konfiguration die Semantik des Programms verändern, wird dies dem Benutzer mitgeteilt. Möglicherweise kann dadurch die Refaktorisierung gar nicht durgeführt werden.

Die Modifikationen werden dem Benutzer in Form von textuellen Unterschieden des Quellcodes vorgeschlagen. Die finale Entscheidung, ob die Refaktorisierung den Absichten entspricht, muss nun getroffen werden (siehe Abb. 6).

Die Arbeit des Werkzeugentwicklers ist hiermit vollbracht. Üblicherweise stellt die Entwicklungsumgebung die weiteren Schritte sicher (siehe Kasten 3) und ist für die Anwendung der Änderung sowie die Handhabung einer Funktion „Rückgängig machen“ verantwortlich.

Idealerweise ist der Ablauf zur Implementierung einer Refaktorisierungsautomatik bereits von der Entwicklungsumgebung vorgegeben. Das ist beispielsweise beim Eclipse Language Toolkit (vgl. [LTK]) der Fall. Dieses unterstützt den Refaktorisierungsentwickler mit folgenden Hilfsmitteln:

- Rahmen für den ganzen Ablauf
- Anzeigen und Anwenden der effektiven Änderungen am Quelltext
- Verwaltung der Funktion „Rückgängig machen“
- Anzeigen von Warnungen und Fehlern

Das LTK stellt eine sprachunabhängige Basis für alle Refaktorisierungen zur Verfügung und hilft bei der Vereinheitlichung des Ablaufs. Vergleichbare Unterstützung wird durch die Visual Studio Erweiterung „ReSharper“ auch für .NET-Sprachen angeboten.

Kasten 3: Unterstützung des Implementierungsablaufs.

Die in diesem Artikel beschriebenen Probleme und vorgeschlagenen Lösungen sind keine rein theoretischen Produkte, sondern resultieren aus unseren Arbeiten an verschiedensten Refaktorisierungswerkzeugen. Die Beispielerfahrungen stammen aus den Refaktorisierungsprojekten für die Sprachen Ruby, C++ und Scala. Letztere befinden sich noch in der Weiterentwicklung:

- Ruby [RDT]
- C++ [CDT]
- Scala [Sca]

Tiefer gehende Informationen zur Implementierung von Refaktorisierungswerkzeugen sind in den Dokumentationen der genannten Projekte zu finden.

Kasten 4: Implementierungen von Refaktorisierungswerkzeugen.

Fazit

Hiermit schließen wir unseren Blick hinter die Kulissen ab – und hoffen, eine klärende Sicht auf die Hintergründe eines typischen Refaktorisierungswerkzeugs vermittelt zu haben. Die zu Beginn gestellte Frage, weshalb gute Refaktorisierungsunterstützung nur selten anzutreffen ist, erklärt sich durch die – auf den ersten Blick verborgene – Komplexität eines solchen Werkzeugs.

Die besprochenen, wieder kehrenden Schritte treten unabhängig von der Programmiersprache auf – jeder Werkzeugentwickler sieht sich früher oder später damit konfrontiert. Sobald die ersten Hürden gemeistert sind, können weitere Funktionen implementiert werden. Dabei sollte der Fokus nicht auf der Implementierung aller möglichen Refaktorisierungen liegen, sondern auf der Integration der bestehenden in den Arbeitsprozess des Entwicklers. Denkbar wäre beispielsweise die automatische Erkennung von *Smells* (fragliche Programmkonstrukte) und dazugehöriger Vorschläge für deren Bereinigung.

Für die Zukunft erhoffen wir uns, dass eine gute Refaktorisierungsautomatisierung – egal für welche Programmiersprache – zum Standard-Repertoire jeder Entwicklungsumgebung gehören wird. Einen Grundstein dafür haben wir mit einigen Projekten bereits gelegt (siehe Kasten 4). ■

Literatur & Links

[Ecl10] The Eclipse Foundation, C++ Development Tools, 2010, siehe: www. eclipse.org/cdt

[Fow99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999

[Fre06] L. Frenzel, The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs, in: Eclipse Magazin Vol. 5, January 2006, siehe: www.eclipse.org/articles/Article-LTK/ltk.html

[Jet] JetBrains, ReSharper, siehe: www.jetbrains.com/resharper

[JRu] JRuby, Ruby Implementation für die JVM, siehe: www.jruby.org

[Opd92] W. Opdyke, Refactoring Object Oriented Frameworks, 1992, siehe: www.laputan.org/pub/papers/opdyke-thesis.pdf

[Pep] Python Eclipse Plug-in für Codeverbesserungen, siehe: peptic.ifs.hsr.ch

[RDT] Refaktorisierungen für die Ruby Development Tools, siehe: r2.ifs.hsr.ch

[Sca] Refactoring für Scala, siehe: scala.ifs.hsr.ch

[Som08] P. Sommerlad, G. Zgraggen, T. Corbat, L. Felber, Retaining Comments when Refactoring Code, OOPSLA Companion, 2008