Prof. Dr. Friedrich Steimann, Dr. Marcus Frenkel, Dr. Daniela Keller

Kurs 01853

Moderne Programmiertechniken und -methoden

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Vorwort

Was ist modern? Das, was gerade angesagt ist? Dann müßte diese Vorlesung jedes Semester neu geschrieben werden. Was dann?

Manches Wissen der Informatik hat eine erschreckend kurze Halbwertszeit. Das gilt auch für den Bereich der Softwareentwicklung und der Programmiersysteme: Die Programmiersprache der Wahl scheint heute JAVA zu sein, die dazugehörige Entwicklungsumgebung vielleicht ECLIPSE. Extreme Programming und agile Softwareentwicklung erschienen uns gestern als die Zukunft, heute ist es darum schon deutlich ruhiger geworden. Was also gehört in eine Vorlesung, die das Attribut „modern“ trägt?

Meine Antwort darauf heißt: Wissen, das man vor Jahren noch nicht hatte, das Sie aber voraussichtlich auch noch nutzen können, wenn Sie Ihr Studium abgeschlossen haben und wenn Sie dann (wieder) mitten in einem Programmierprojekt stecken. Ihr hier erworbenes Wissen entspricht dann sicher nicht dem neuesten Hype, aber es hat hoffentlich noch eine gewisse Aktualität (während der Hype von heute vielleicht längst als „ganz nette Idee, aber letztlich doch untauglich“ abgeschrieben ist). Um konkreter zu werden: Das Wissen dieses Kurses sollte eine Halbwertszeit von zehn Jahren haben, d. h., die Hälfte dessen, das Sie heute lernen, sollte in zehn Jahren noch gültig und verwertbar sein.

Modern heißt aber auch immer: Noch nicht vor der Geschichte bewährt. Und so habe ich mir erlaubt, in einem Kurstext das Ideal der Einheit von Forschung und Lehre beim Wort zu nehmen und das eine oder andere an meinem Lehrgebiet erzielte Forschungsergebnis in den Text einfließen zu lassen. Da diese Ergebnisse allesamt jüngeren Ursprungs sind, gilt für sie natürlich ganz besonders, daß sie sich noch nicht bewährt haben. Auf der anderen Seite finden Sie ja darin vielleicht einen interessanten Denkansatz und idealerweise sogar ein Thema für eine eigene Abschlußarbeit.

Noch ein Wort zur Sprache: Aufgrund eines Rektoratsbeschlusses der Fernuniversität bin ich gehalten, eine geschlechtsneutrale Sprache zu verwenden oder, wo nicht möglich, beide Geschlechter anzusprechen. Ich kann nicht sagen, ob dies überhaupt praktikabel ist — Proponentinnen und Proponenten mögen sich „Programmierer- und Programmiererinnenproduktivität“ zu Gemüte führen oder versuchen, einen Satz wie „Wer glaubt, das sei einfach, ohne es probiert zuhaben, den kann ich nicht ernstnehmen.“ geschlechtsneutral zu formulieren, ohne daraus ein Ungetüm zu machen —, aber eine verständliche Sprache auf dem Altar der Gleichstellung zu opfern war für mich keine Option. Ich habe mich deshalb dazu entschlossen, ausschließlich die weibliche Form zu verwenden. Diese Entscheidung führt hier und da zu unerwarteten Wendungen, die zeigen, wie sehr das männliche Geschlecht in unserer Sprache verankert ist, vor allem aber dazu, daß *einer* die direkte Ansprache eines Geschlechts überhaupt erst auffällt. Ich hoffe, daß sich dadurch niemand, *die* diesen Text liest, diskriminiert fühlt.

Übersicht

Der Kurs beginnt mit einem etwas eigenwilligen Thema, nämlich der sog. *interfacebasierten Programmierung*. Diese propagiert die Verwendung von Interfaces (als Typen wie in JAVA oder C#) anstelle von Klassen bei der Typisierung von Variablen (also in Variablendeklarationen). Das Konzept und die Verwendung von Interfaces ist ein immer wiederkehrendes Thema in den folgenden Kurseinheiten; die Einführung der interfacebasierten Programmierung gleich zu Anfang scheint daher gerechtfertigt, selbst wenn es sich bei ihr ausdrücklich nicht um ein Standardthema handelt.

Interfaces à la JAVA und C# sind unvollständig. Was Ihnen fehlt, ist eine (formale) Beschreibung dessen, was die Einhaltung des Interfaces über die rein syntaktischen Methodensignaturen hinaus verlangt, gewissermaßen eine Semantik der Methoden oder, anders gesagt, eine genaue, damit verbundene Verhaltensspezifikation. *Design by contract* ist ein besonders griffig formuliertes Prinzip, dieses Defizit auszugleichen: Ihm zufolge wird über ein Interface ein Vertrag geschlossen, nach dem beide Seiten — gewissermaßen über Kreuz — gegenseitige Verpflichtungen und Nutzen haben. Die Einhaltung dieses Vertrages kann über sog. Zusicherungen zur Laufzeit und — in ausgewählten Fällen — auch statisch, also zur Übersetzungszeit, geprüft werden. All dies ist Gegenstand der zweiten Kurseinheit.

Die Überprüfung der Einhaltung von Verträgen sowie allgemeiner der Korrektheit von Code über Zusicherungen ist nur eine Möglichkeit, für Qualität in der Programmierung zu sorgen. Die andere ist das Testen. Zwar ist das Testen nicht besonders beliebt (es hat gewissermaßen destruktiven Charakter — man macht die großen Würfe anderer kaputt, ohne jemals selbst brillieren zu können), doch ist es nach wie vor unverzichtbar. Nicht zuletzt nutzt einer das ganze schöne Design by contract nichts, wenn dessen Zusicherungen bei der Kundin das erste Mal zur Anwendung kommen und dann dort eine Vertragsverletzung melden. Sog. *Unit-Tests*, insbesondere die, die auf dem Framework JUNIT basieren, haben in letzter Zeit das Testen unter Programmiererinnen etwas populärer gemacht, weswegen ihnen auch eine ganze Kurseinheit gewidmet wird.

Die vierte Kurseinheit widmet sich dann der Idee der *Entwurfsmuster* und damit einem deutlich populäreren Thema. Entwurfsmuster bieten zunächst einen Katalog von Standardlösungen für häufig wiederkehrende Entwurfsprobleme; sie bilden aber ganz nebenbei auch ein Vokabular, das es Softwareentwicklerinnen erlaubt, sich kurz und prägnant über Software zu unterhalten, und zwar ohne sich in Details zu verlieren, nämlich unter Verweis auf bekannte Konzepte (ganz so, wie sich Drehbuchautoren und Produzenten in Robert Altmans „The Player“ über Filme unterhalten). Beinahe überflüssig zu sagen, daß Interfaces in Entwurfsmustern eine wichtige Rolle spielen.

Auch wenn es viele nicht wahrhaben wollen: Die Programmierung ist ein evolutionärer Prozeß, bei dem einmal getroffene Entscheidungen ständig auf die Probe gestellt werden und gegebenenfalls wieder geändert werden müssen. Die Ände-

rung von Code ist aber in der Regel eine verzweigte und entsprechend verzwickte Angelegenheit: Nur selten ist es mit einer Änderung an einer Stelle getan. Das führt dann häufig zu der Erkenntnis, daß Software unter Änderung verdirbt. Dem sollen sog. *Refactorings*, standardisierte und teilweise auch automatisierte Änderungen von Code, die dessen Bedeutung nicht verändern, entgegenwirken. Ja noch viel mehr: Mit Hilfe von Refactorings soll sich der Entwurf (und damit die Qualität) existierender Software durch gezielte Änderungen nachträglich verbessern lassen.

Als Abschluß der Programmiertechniken wird noch ein anderes Thema aufgegriffen, das — unter einem neuen Deckmäntelchen namens *aspektorientierte Programmierung* — derzeit für Aufsehen sorgt: die *Metaprogrammierung*. Unter Metaprogrammierung versteht man zunächst die Erstellung von Programmen, die Programme erzeugen oder ändern. Besonders interessant wird die Metaprogrammierung dann, wenn ein Programm sich selbst zu verändern in der Lage ist. Dies ist in gewisser Weise bei der aspektorientierten Programmierung der Fall. Mit einer Behandlung dieser und den ebenfalls erst vor kurzem aktuell gewordenen Annotationen schließt diese Kurseinheit.

Unit-Tests und Refactorings sind zwei Programmiertechniken, die im Rahmen einer bestimmten Programmiermethode größere Bekanntheit erlangt haben, nämlich des *Extreme Programming*. Extreme Programming vereint eine Vielzahl relativ unorthodoxer Herangehensweisen zu einem Gesamtkunstwerk, dessen Praxistauglichkeit in der Vergangenheit viel diskutiert wurde. Ohne daß heute ein abschließendes Ergebnis vorläge, hat das Extreme Programming aber immerhin über das Wort von den „agilen Methoden“ oder Prozessen zu einer immer stärker werdenden Abkehr von schwergewichtigen Softwareentwicklungsansätzen und damit, zumindest aus Sicht der Programmiererinnen, zu einer Art Befreiung geführt. Auch wenn die agile Softwareentwicklung längst nicht auf alle Projekte und Organisationen paßt, so läßt sich doch sicher die eine oder andere Anregung daraus mitnehmen.

5 Refactoring

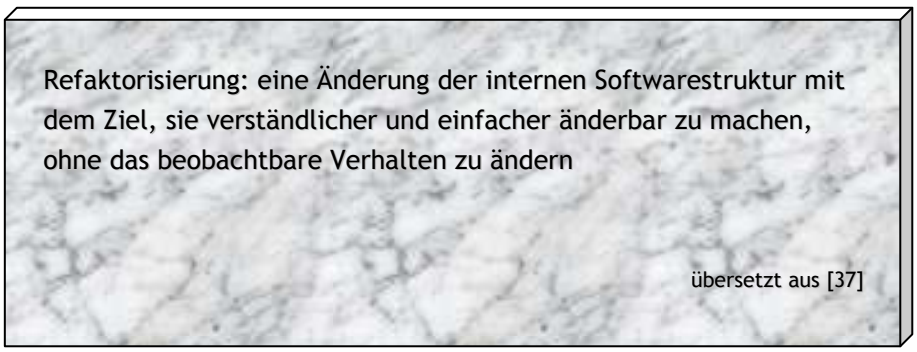
So wie sich in der Software bestimmte Muster ständig wiederholen, so gibt es auch im Prozeß der Programmierung selbst bestimmte stereotype Tätigkeiten, die immer wieder und dabei mit nur leichten Variationen ausgeführt werden müssen. Dies betrifft insbesondere die Änderung von Code, bei der selbst so einfache Dinge wie das Umbenennen einer Klasse eine Masse von manuellen Änderungen nach sich ziehen, die eigentlich vollkommen schematisch und damit automatisch erfolgen können sollten.

Selbsttestaufgabe 5.1

Was muß alles geändert werden, wenn sich der Bezeichner einer Klasse ändert?

Refaktorisieren und Softwarefäulnis

Der zum Teil erhebliche Aufwand, den selbst solch kleine Änderungen nach sich ziehen, führt allzu häufig dazu, daß nur die unvermeidlichen, weil die Funktionalität betreffenden Änderungen im Code durchgeführt werden, während die Modifikationen, die lediglich dem Erhalt von Struktur, Wartbarkeit und Lesbarkeit des Codes dienen würden, ausbleiben. Der Code verkommt damit in dem Maße, in dem er geändert wird; man spricht hier auch von *Softwarefäulnis*. Das sog. Refaktorisieren wirkt der Softwarefäulnis entgegen.



zum Begriff der Refaktorisierung

Da die mit der Refaktorisierung verbundenen Änderungen auf ein Umbauen des Programms abzielen, ohne dessen Bedeutung zu verändern, heißen sie englisch Refactorings. Dabei ist das Wort „Refactoring“ eine Neuschöpfung, die durch die Analogie erklärbar ist, daß ein Programm ein aus einer Menge von Faktoren bestehendes Produkt, eben faktorisiert ist. Diese Faktorisierung offenbart eine innere Struktur, die ohne sie (die Faktorisierung) nicht so leicht sichtbar wäre. Eine Änderung der Faktorisierung nennt man dann folgerichtig Refaktorisierung.[51]

Mit der systematischen Befassung und der Katalogisierung möglicher Refaktorisierungen hat der Begriff der Refaktorisierung eine Mehrdeutigkeit erfahren: Er bezeichnet nicht nur den Vorgang oder das Ergebnis einer entsprechenden Tä-

[51] Bei der im Deutschen ebenfalls üblichen Übersetzung „Refaktorierung“ handelt es sich um eine sinnberaubende Verstümmelung, die hier bewußt nicht verwendet wird.

tigkeit (an sich schon eine Mehrdeutigkeit), sondern auch das Muster, nach dem die Tätigkeit erfolgt bzw. dem das Ergebnis gleicht. Während man im Deutschen noch eine sprachliche Unterscheidung treffen kann (das „Refaktorisieren“ für die Tätigkeit vs. „die Refaktorisierung“ für das Ergebnis oder das Muster), gelingt dies im Englischen nicht: Dort heißt es in allen Fällen schlicht „refactoring“. Dazu kommt, daß mit Refaktorisierung bzw. Refactoring in zunehmendem Maße auch die Werkzeuge, die eine solche automatisch durchführen, benannt werden. Diese sollten aber zur besseren Unterscheidung Refaktorisierungswerkzeuge bzw. Refactoring tools genannt werden.

5.1 Einordnung

Allgemein dient das Refaktorisieren der Verbesserung des Designs einer Software, *nachdem* diese geschrieben wurde. Dieses Verhalten ist (war) eigentlich verpönt (das Design hat schließlich nach gängigen Vorstellungen von einem geregelten Softwareentwicklungsprozeß vor der Implementierung zu erfolgen), die Notwendigkeit ergibt sich aber faktisch aus der Praxis — sie ist schlichtweg Realität. Inzwischen ist man dazu übergegangen, dies anzuerkennen; nicht zuletzt führen die Änderungen der Anforderungen während der Entwicklungsphase (sog. *Requirements drift*, im Mittel ca. 1% der Anforderungen pro Monat) dazu, daß man sich mit dem Thema auseinandersetzen muß. Die Methode des *Extreme Programming* (Kurseinheit 7) verzichtet sogar vollständig auf ein A-priori-Design und setzt vollständig auf das Refaktorisieren.

Refactorings wurden zuerst in der SMALLTALK-Programmierung eingesetzt. Die erste literaturgeschichtliche Erwähnung ist angeblich von 1990, die erste größere wissenschaftliche Abhandlung ist die Dissertation von Opdyke [36]. Das erste bekannte Werkzeug war der sog. *Refactoring-Browser* für SMALLTALK. Aufgrund seines vorwiegend praktischen Anteils ist das Thema Refactoring in der akademischen Welt bislang nicht besonders vertreten. Gleichwohl gibt es eine ganze Menge offener, nichttrivialer Probleme, die anzugehen sich auch für Wissenschaftlerinnen lohnen kann.

5.1.1 Katalogisierung

Ähnlich wie die Entwurfsmuster aus Kurseinheit 4 lassen sich Refaktorisierungen standardisieren, indem man Schemen in Form strukturierter Beschreibungen vorgibt und diese mit Namen versieht. Diese Namen werden dann zu Synonymen häufig recht komplexer Tätigkeiten und damit Bestandteil des Vokabulars von Programmiererinnen.

Die Beschreibung eines Refactorings geht in der Regel von einem Design aus, das (aus welchem Grund auch immer) verbesserungswürdig erscheint. Es stellt diesem Design ein alternatives gegenüber, das das Ziel des Refactorings darstellt. Die notwendige Transformation des Quellcodes vom alten in das neue Design erfolgt dann mittels einer Reihe von kleinen Schritten, die in der Summe die erwünschte Änderung bewirken.

Die wohl auch heute noch umfassendste Katalogisierung von Refactorings ist die von Martin Fowler in seinem gleichnamigen Buch [37]. Eine Erweiterung dieses Katalogs wird (einigermaßen lieblos) von ihm selbst im Internet gepflegt[52]; eine Neuauflage des Buchs mit den entsprechenden Aktualisierungen ist jedoch nicht geplant. Dies steht m. E. im Gegensatz zu der Wichtigkeit des Themas für die Programmierpraxis: Während von der Weiterentwicklung von Programmiersprachen in den letzten Jahren kaum noch Produktivitätssteigerungen ausgingen, so können zuverlässige Programmierwerkzeuge die Arbeit erheblich vereinfachen, indem sie stereotype Änderungen wie eben das Refaktorisieren von Code automatisieren. Damit diese Werkzeuge jedoch auch eingesetzt werden, ist es notwendig, daß sie über die jeweiligen Produktgrenzen hinaus bekannt sind und in den Softwareentwicklungsprozeß auch gedanklich einbezogen werden (so wie das bei den Entwurfsmustern aus Kurseinheit 4 längst der Fall ist).

große Refactorings

Die katalogisierte Beschreibung von Refactorings erfaßt fast ausschließlich kleine Refaktorisierungen, also solche, die sich als eine überschaubare Menge von Einzelschritten durchführen lassen und deren Erfolg sich leicht überprüfen läßt. Für die Praxis genauso relevant ist aber die Definition und Umsetzung von sog. **großen Refactorings**, also Refactorings, die nicht nur einzelne Programmelemente betreffen, sondern größere Zusammenhänge bis hin zur gesamten Architektur eines Systems. Von einer formalen Vorgehensbeschreibung (oder gar von einer Werkzeugunterstützung) ist man hier aber noch weit entfernt; statt dessen verlegt man sich darauf, große Refactorings als eine Aneinanderreihung kleiner Refactorings zu betrachten. Dagegen spricht jedoch die hohe Fehlerquote, die entsteht, wenn eine große Transformation jedesmal wieder neu in viele kleine zerlegt werden soll, die dann jeweils einzeln durchgeführt werden müssen. Das wird insbesondere dann schwierig, wenn dabei Zwischenprodukte entstehen, die das große Ziel aus den Augen verlieren lassen.

5.1.2 Refaktorisierungen als Algorithmen

Wenn man sich die Beschreibungen von Refaktorisierungen anschaut, fällt auf, daß sie einem (verbal spezifizierten) Algorithmus gleichen: Ausgehend von einem vorgefundenen Design, der Eingabe, wird durch eine Reihe von Schritten ein Zieldesign, die Ausgabe, erzeugt. Dabei kann das Refactoring während seiner Durchführung weitere Eingaben (von der Benutzerin) erwarten oder bereits vor seinem Start mit Parametern versorgt werden. Das besondere an Refactorings ist lediglich, daß es sich bei Ein- und Ausgabe um Programme handelt. Ein Refaktorisierungswerkzeug ist damit ein Programm, das Programme verarbeitet, nämlich ein *Metaprogramm* (s. Kurseinheit 6).

Vor- und Nachbedingungen von Refactorings

Refaktorisierungen können damit genau wie andere Programme spezifiziert werden. Insbesondere hat jedes Refactoring eine Menge von *Vorbedingungen*, die erfüllt sein müssen, damit das Refactoring anwendbar ist, und eine Menge von *Nachbedingungen*, die erfüllt sein müssen, wenn die Refaktorisierung abgeschlos-

52 www.refactoring.com/catalog/

sen ist. Vor- und Nachbedingungen sind insbesondere auch dann interessant, wenn man mehrere Refactorings hintereinander ausführen will, wobei jedes folgende Refactoring eine oder mehrere Nachbedingungen des Vorgängers als Vorbedingungen zur Voraussetzung hat. Allerdings ist der Beweis der Korrektheit von Refactorings, genau wie der Beweis der Korrektheit von Algorithmen, alles andere als einfach.

automatisches Testen der Korrektheit

Eine besondere Nachbedingung jedes Refactorings ist definitionsgemäß, daß es die Bedeutung des Programms nicht ändert. Neben der trivialen Bedingung, daß sich das Programm nach dem Refactoring weiter problemlos übersetzen lassen muß (zumindest wenn dies zuvor der Fall war), gehört auch dazu, daß verfügbare Unit-Tests (Kurseinheit 3) äquivalente Ergebnisse liefern, in der Regel also weiter erfolgreich durchlaufen. Unabhängig von der Verfügbarkeit von Unit-Tests eröffnet die Anwendung von Refactorings zusätzlich die Möglichkeit des sog. *Back-to-back-Testens*: Man kann einfach die Ausgaben des Programms vor und nach der Refaktorisierung gegeneinander vergleichen. Voraussetzung hierfür ist allerdings, daß man über ein Testframework verfügt, das solche Tests (inkl. der automatischen Generierung der dafür notwendigen Eingaben) automatisch durchführt. Der Grundsatz des Testens, daß man damit nur Fehler finden kann und keine Fehlerfreiheit nachweisen, behält aber auch hier Gültigkeit.

5.1.3 Refactoring to patterns

Zweck eines Refactorings ist es, ein vorgefundenes Design in ein angestrebtes zu überführen. Angestrebtes objektorientiertes Design ist Ihnen ja schon begegnet, und zwar in Form von Entwurfsmustern (Kurseinheit 4). Was läge also näher, als spezielle Refactorings vorzusehen, die die Verwendung eines Entwurfsmusters zum Ziel haben?

Einer katalogartigen Fassung solcher Refaktorisierungen steht vor allem im Wege, daß die Ausgangslage, also ein Design, das kein Entwurfsmuster verwendet, nahezu beliebig ist. Wenn aber die Form der Eingabe unbekannt ist, ist es schwer, ein allgemeines Verfahren anzugeben, wie sie in die Ausgabe überführt werden kann. Statt dessen kann man nur von der gewünschten Ausgabe (die ja feststeht) rückwärts ausgehend angeben, welche der katalogisierten Refactorings dorthin führen können, und die Auswahl der für die jeweils benötigten Transformationen geeigneten Refactorings der Entwicklerin überlassen. Die dabei auftretenden Probleme entsprechen doch im wesentlichen denen der *großen Refactorings*: Das Refactoring muß jedes mal neu als eine Folge von kleineren Einzel-Refactorings geplant werden und damit hängt der Erfolg ganz wesentlich vom Geschick der Programmiererin ab.

Etwas anderes ist es jedoch, wenn ein Entwurfsmuster (genauer: seine Anwendung) in ein anderes überführt werden soll: Hier sind sowohl Ausgangs- als auch Zielsituation bekannt. Ein Beispiel für ein solches Refactoring werden wir in Abschnitt 5.1.5 ausführlicher durchgehen. Im allgemeinen dienen jedoch verschiedene Entwurfsmuster verschiedenen Zwecken und zwei Entwurfsmuster, die auf dieselbe Problemstellung angewendet dasselbe Programmverhalten bewirken,

sind relativ selten, so daß man auch eher selten von einem Entwurfsmuster in ein anderes refaktorisieren wird.

5.1.4 Werkzeugunterstützung

Die oben erwähnte algorithmische Fassung von Refactorings legt nahe, daß sich Refactorings automatisieren lassen. Allerdings ist es bei fast allen Refactorings (wie schon beim eingangs erwähnten einfachen Beispiel des Umbenennens einer Klasse) nicht mit einem globalen, textuellen Suchen und Ersetzen getan — praktisch immer erfordern die Änderungen eine Interpretation des Programmtextes. Sie benötigen in der Regel den *abstrakten Syntaxbaum* (engl. abstract syntax tree, AST) des Programms.

Nun erlauben moderne IDEs mit ihren APIs in der Regel den Zugriff auf den abstrakten Syntaxbaum eines Programms, so daß die erforderlichen Programmanalysen und -transformationen von einem Refaktorisierungswerkzeug, das auf diesen APIs aufsetzt, direkt auf den benötigten Datenstrukturen durchgeführt werden können. Allerdings sind derartige APIs zum einen nicht sonderlich stabil, zum anderen unterliegen auch die Programmiersprachen, auf deren ASTs die Refactorings ausgeführt werden sollen, ständigen Änderungen, die nicht nur eine Anpassung von Editor und Compiler, sondern auch der Refactorings erfordern. So führte beispielsweise der Übergang von JAVA 2 zu JAVA 5 (mit seinen generischen Typen) dazu, daß viele Refactorings nicht mehr funktionieren und deshalb komplett überarbeitet werden müßten (was aber aufgrund des erheblichen Aufwands nicht immer paßiert). Dazu kommt, daß gleiche Refactorings selbst für syntaktisch stark ähnliche Programmiersprachen (wie etwa JAVA und C#) heute nicht einfach übernommen werden können, da die Implementierungen zu sehr von den konkreten ASTs der jeweiligen Sprache abhängen. Höhere Abstraktionsstufen, die eine allgemeine Formulierung von Refactorings erlauben, sind jedoch noch nicht gefunden (oder zumindest noch nicht etabliert) — eine allgemeine, einheitliche Refactoring-Schnittstelle von IDEs ist (noch) nicht in Sicht.

So kommt es, daß sich heutige IDEs auch darin stark unterscheiden, ob und welche Refactorings sie anbieten. Besonders hervor tut sich auf diesem Gebiet die IDE INTELLIJ IDEA der Firma JETBRAINS, die mittlerweile eine beachtliche Menge an Refactorings anbietet. In der akademischen Gemeinde am sichtbarsten ist aber zur Zeit immer noch ECLIPSE, dessen Refactorings teilweise einen beträchtlichen Funktionsumfang erreicht haben. Entwicklungsumgebungen wie VISUAL STUDIO ziehen hier erst langsam nach. Alleinstehenden Refaktorisierungswerkzeugen scheint, eben aufgrund ihrer mangelnden Integration, in der Praxis keine besondere Bedeutung mehr zuzukommen, selbst wenn ihnen noch am ehesten die Lösung der obengenannten technischen Probleme zuzutrauen wäre.

Zum Schluß noch eine Bemerkung zur Qualität: Trotz des beträchtlichen Potentials zur Automatisierung des Tests von Refaktorisierungswerkzeugen (s. o.) weisen die heute verfügbaren Implementierungen zum Teil erhebliche Mängel auf. So führen selbst einfache Refaktorisierungen in der Praxis schon zu Syntax- oder

Typfehlern, so daß das wichtigste Merkmal heutiger Refaktorisierungswerkzeuge ist, ob die Undo-Funktion zuverlässig funktioniert. Daraus lassen sich zwei Dinge ableiten: Die korrekte Fassung eines Refactorings, einschließlich seiner Vorbedingungen, für alle möglichen Verwendungen ist viel komplexer, als es die oft informellen Beschreibungen glauben machen wollen, und die Verfügbarkeit von zuverlässigen, automatisierten Refaktorisierungen ist weit weniger entscheidend für den Erfolg einer IDE, als man vielleicht erwarten würde. Tatsächlich ergeben Umfragen unter Entwicklerinnen und die Rückmeldungen aus Fehlerdatenbanken, daß verfügbare Refaktorisierungswerkzeuge längst nicht im erwarteten Umfang genutzt werden.

5.1.5 Ein Beispiel

Nach den eher theoretischen Betrachtungen der vorigen Abschnitte wollen wir uns nun etwas ausführlicher mit einem konkreten Beispiel befassen, und zwar einem, das Ihnen aus Abschnitt 4.2 bereits bekannt ist: Vererbung durch Delegation ersetzen. Das dazugehörige Refactoring nennt sich denn auch genau so, auf englisch **REPLACE INHERITANCE WITH DELEGATION**.

In [37] wird dieses Refactoring kurz und knapp wie folgt charakterisiert:

> *A subclass uses only part of a superclasses interface or does not want to inherit data.*
>
> *Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.*

Es folgt eine etwas ausführlichere Motivation und Beschreibung der Schritte, die jedoch nicht über den Inhalt des obigen hinausgeht. Die zur Durchführung des Refactorings konkret notwendigen Schritte werden in einem Abschnitt „Mechanics“ wie folgt ausgeführt:

- *Create a field in the subclass that refers to an instance of the superclass. Initialize this field to this.*
- *Change each method defined in the subclass to use the delegate field. Compile and test after changing each method.*
- *Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.*
- *For each superclass method used by a client, add a simple delegating method.*
- *Compile and test.*

Vorbedingungen

Wir erkennen im ersten Satz die *Vorbedingung* (wenn auch nicht unbedingt als solche formuliert): Eine Klasse ist Subklasse einer anderen und erbt von ihr Dinge, die sie selbst nicht braucht. Der erste Teil der Vorbedingung ist ein hartes Ausschlußkriterium: Wenn eine Klasse keine Subklasse ist, läßt sich das Refactoring nicht anwenden. Der zweite Teil ist etwas weicher: Wenn sie keine Dinge erbt, die sie nicht braucht, ist das Refactoring zwar immer noch anwendbar, bringt aber nicht den beschriebenen Nutzen.

Schritte und Nachbedingungen

Der zweite Satz stellt eine Kombination von grob zusammengefaßten Handlungsanweisungen und *Nachbedingung* dar: Nachdem man das Erforderliche getan hat, kann die Vererbungsbeziehung gelöst werden, die damit nicht mehr besteht. Daß das Programm weiter funktioniert wie bisher, ist im Begriff des Refactorings implizit (wenn sich die Bedeutung ändern würde, wäre es kein Refactoring) und braucht daher nicht extra erwähnt zu werden.

Der aufmerksamen Leserin von Kurseinheit 4 wird nicht entgangen sein, daß das Refactoring *Delegation* einzuführen vorgibt, aber nur *Forwarding* einführt: Für eine echte Delegation fehlt ja der Verweis (das Feld) von der ehemaligen Superklasse zurück zur Klasse. Das bedeutet, daß wenn es vorher offen rekursive Aufrufe in der Superklasse gab, daß dann die refaktorisierte Klasse nach dem Refactoring davon nicht mehr erreicht wird. Es folgt, daß entweder die Vorbedingungen oder die Handlungsanweisungen unvollständig sind.[53] Eine genauere Untersuchung des Refactorings ist also angebracht.

5.1.5.1 Vorbedingungen

Eine triviale Vorbedingung des Refactorings, die in [37] unerwähnt bleibt, ist die, daß die Superklasse nicht abstrakt sein darf, da sonst keine Instanz gebildet werden kann, an die „delegiert" (eigentlich: geforwardet) wird. Zudem müssen alle Konstruktoren der Superklasse, die benötigt werden, um eine brauchbare Instanz zu erhalten, an die delegiert werden kann, zugreifbar sein. Sie dürfen also in JAVA insbesondere nicht `private` oder `protected` deklariert sein. Welche Konstruktoren (neben dem Default-Konstruktor, der in JAVA bei Vererbung automatisch mit aufgerufen wird) man braucht, läßt sich aus den in den Konstruktoren der Klasse vorhandenen Super-Aufrufen ablesen.

Eine dritte, schon weniger offensichtliche Vorbedingung ist die, daß das Programm keine Zuweisungen von Instanzen der zu refaktorisierenden Klasse an Variablen vom Typ der Superklasse oder dessen Supertypen enthalten darf, denn mit der Elimination der Vererbungsbeziehung geht auch die Aufgabe der Subtypenbeziehung einher. Diese ist aber zumindest in Sprachen mit *nominalem Typsystem* (worunter fast alle heute gebräuchlichen Sprachen mit statischer Typprüfung fallen; vgl. Kurs 01814) Voraussetzung für die Zulässigkeit einer Zuweisung. Diese Vorbedingung läßt sich etwas abschwächen, wenn man bereit ist, Subtypenbeziehungen zu Supertypen der Superklasse durch neue Extends- bzw. Implements-Klauseln wiederherzustellen, wobei man sich bei ersteren natürlich wieder Vererbung einkauft, die man ja gerade eliminieren wollte. Wenn aber Zuweisungen an den Typ der Superklasse im Programm vorhanden sind, läßt sich das Refactoring einfach nicht anwenden.

---

[53] Wie schon im Kontext von *Design by contract* (in Kurseinheit 2) erwähnt, kann man ein fehlerhaftes Programm dadurch korrigieren, daß man seine Vorbedingungen verschärft. Wenn die Vorbedingungen nicht eingehalten werden, braucht das Programm auch nichts (Sinnvolles) zu tun.

Die vierte Vorbedingung hatten wir bereits erwähnt: Da es sich bei der durch das Refactoring eingeführten „Delegation“ lediglich um schnödes Forwarding handelt, dürfen Instanzen der zu refaktorisierenden Klasse nicht von ihrer Superklasse aus offen rekursiv, also über `this`, aufgerufen werden. Eine Quelle solch offen rekursiver Aufrufe wurde dabei jedoch schon ausgeschlossen: Da die Zuweisungskompatibilität der Klasse mit ihrer ehemaligen Superklasse aufgehoben wird, können Instanzen der Klasse die ihrer Superklasse nicht mehr ersetzen. Das Vorkommen entsprechender Zuweisungen im Programm schließt die dritte Vorbedingung bereits aus. Es bleiben jedoch geforwardete Aufrufe der Klasse selbst an die ehemalige Superklasse, die nun, wie in Abbildung 4.1 zu sehen, nicht mehr die Methoden der Klasse aufrufen können, weil `this` nun eine Instanz der ehemaligen Superklasse bezeichnet. Solche Aufrufe müssen also ebenfalls ausgeschlossen werden, wenn keine Änderung des Programmverhaltens provoziert werden soll.

Eine fünfte Vorbedingung ergibt sich aus dem Umstand, daß in JAVA nur Methodenaufrufe weitergeleitet werden können: Greifen Klientinnen der Klasse auf von der Superklasse geerbte Felder zu, so würden diese Zugriffe nach dem Refactoring ins Leere gehen. Das ließe sich allerdings vermeiden, wenn Feldzugriffe konsequent über Zugriffsmethoden (Accessoren) erfolgen würden, die dann weitergeleitet werden könnten. In EIFFEL ist das übrigens immer der Fall, da dort Feldzugriffe transparent, also für die Zugreiferinnen nicht sichtbar, immer über Accessoren erfolgen; in C# läßt es sich über sog. *Properties* ebenfalls einrichten, ohne die Klientinnen anfassen zu müssen. Man beachte, daß es nicht ausreicht, die betreffenden Felder in der Klasse einfach zu wiederholen (neu zu deklarieren): Die ehemals geerbten Methoden, an die jetzt weitergeleitet wird, hätten nur Zugriff auf ihre eigenen Versionen dieser Felder und eine Synchronisation der Inhalte ist aus technischen Gründen nicht möglich.

Eine sechste Vorbedingung ergibt sich aus den Sichtbarkeitsregeln der jeweils verwendeten Programmiersprache: Wenn beispielsweise in JAVA eine Methode mit dem Access modifier `protected` geerbt wird und vererbende und erbende Klasse nicht im selben Paket sind, dann ist diese Methode nach dem Refactoring nicht mehr zugreifbar und kann auch nicht per Forwarding aufgerufen werden.

Eine siebte Vorbedingung ergibt sich daraus, daß bestimmte Subtypbeziehungen aufrecht erhalten werden müssen, selbst wenn keine sie verlangenden Zuweisungen in einem Programm vorhanden sind: So dürfen beispielsweise keine entsprechenden expliziten Typtests (in JAVA per `instanceof` oder `getClass()`) vorkommen und die Ableitung von sog. Unchecked exceptions wie `Error` und `RuntimeException` darf nicht durch eine von `Throwable` ersetzt werden, damit der Compiler keine fehlenden Exception handler bzw. Throws-Klauseln moniert.

Eine letzte Vorbedingung schließlich ist so unoffensichtlich, daß vermutlich nur die allerwenigsten durch bloßes Überlegen auf sie kommen. Wenn auf den Instanzen einer Subklasse synchronisierte Methoden aufgerufen werden, die teilweise von einer Superklasse geerbt werden, und sich diese Aufrufe nach dem Refactoring auf zwei verschiedene Instanzen verteilen, klappt die Synchronisation u. U. nicht mehr (da jetzt zwei anstelle eines Monitors herangezogen werden).

Wie man sieht, sind die Voraussetzungen für die Anwendung selbst eines scheinbar so simplen Refactorings wie des hier beschriebenen alles andere als banal. Dabei ist man noch gut bedient, wenn der Compiler die Verletzung einer Vorbedingung entdeckt, weil sich nämlich das trotzdem refaktorisierte Programm nicht mehr übersetzen läßt; weit schlimmer ist es, wenn die Verletzung — oder gar die Existenz! — einer Vorbedingung unentdeckt bleibt. Da aber der formale Beweis, daß die für ein Refactoring genannten Vorbedingungen vollständig sind und ausreichen, eine korrekte, bedeutungserhaltende Refaktorisierung durchzuführen, alles andere als auf der Straße liegt, ist man bei allen Refaktorisierungen, und seien sie noch so gut dokumentiert, darauf angewiesen, ihren Erfolg per Testen zu überprüfen. Nicht gerade schmeichelhaft für eine Disziplin, die der Softwareentwicklung eine bedeutende Produktivitätssteigerung bescheren will.

5.1.5.2 Durchführung

Sind die Vorbedingungen erfüllt, kann das Refactoring durchgeführt werden. Doch auch dabei ergibt sich ein nichttriviales Problem: Es ist nämlich gar nicht automatisch klar, welche geerbten Felder und Methoden einer Klasse nicht gebraucht werden. Der Wunsch allein, bestimmte Elemente loszuwerden, reicht dazu nicht aus — es muß auch sichergestellt werden, daß diese Elemente nicht von Klientinnen oder Subklassen der Klasse benötigt werden.

Nun kann man sich auf den Standpunkt stellen, der Compiler wird einem schon sagen, welche Elemente erhalten bleiben müssen — Zugriffe auf nicht mehr vorhandene Programmelemente führen schließlich zu einem Übersetzungsfehler. Aber dieser Ansatz ist der des Trial and error: Man entfernt einfach die Elemente, die man gern loswerden würde, und wenn darunter eines zuviel war, bekommt man dies rückgemeldet. Für die Praxis ist das jedoch wenig befriedigend: Zum einen werden in den typischen Anwendungsfällen dieses Refactorings sehr viele Elemente geerbt (50 und mehr bei Ableitung von Frameworkklassen wie denen des AWT oder SWING) und zum anderen kann sich dabei herausstellen, daß das Refactoring insgesamt keine gute Idee war (nicht den gewünschten Effekt hat), man es also gern komplett wieder rückgängig machen möchte. Dann aber ist schon viel geändert und die Wiederherstellung des Ausgangszustands entsprechend aufwendig. Nicht zuletzt ist es, wenn kein entsprechendes Refaktorisierungswerkzeug vorhanden ist, kaum zumutbar, erst alle Methodenweiterleitungen einzuführen, um diese dann einzeln wieder zu entfernen. Macht man es aber andersherum, beginnt man also ohne eine Methodenweiterleitung und führt diese wie vom Compiler gefordert ein, kann es sein, daß man zunächst so viele Fehlermeldungen bekommt, daß der Überblick verlorengeht.

Was man statt dessen für eine zielgerichtete Anwendung des Refactorings bräuchte, ist eine Programmanalyse, die vorab feststellt, welche Elemente von der zu refaktorisierenden Klasse verlangt werden. Eine solche Analyse wird im Prinzip vom Compiler durchgeführt, während er die Referenzen auflöst — ein entsprechendes Refaktorisierungswerkzeug kann sich diese zunutze machen, wenn es Zugriff auf den AST des Programms hat. Tatsächlich verfügt die Implementierung des REPLACE INHERITANCE WITH DELEGATION Refactorings in INTEL-

LIJ IDEA über eine solche Analyse; leider ist sie nicht vollständig, so daß das Programm nach der Durchführung des Refactorings so manches Mal nicht mehr kompiliert. Das zeigt einmal mehr, wie wenig entwickelt das Thema Refaktorisierungswerkzeuge heute noch ist.

5.1.5.3 Nachbedingungen

Nach einer erfolgreichen Durchführung des Refactorings sollten neben der allgemeinen Nachbedingung für Refactorings, daß das Programm hinterher immer noch dasselbe tut, auch die speziellen Ziele erreicht sein. Das heißt konkret, daß die refaktorisierte Klasse nicht mehr von ihrer ehemaligen Superklasse erbt, sondern statt dessen ein Feld besitzt, mittels dessen ihre Instanzen auf jeweils eine Instanz der ehemaligen Superklasse verweisen, an die sie weiterdelegieren können. Dieses Feld muß bei jeder möglichen Form der Instanziierung der refaktorisierten Klasse automatisch mit einer Instanz der ehemaligen Superklasse versorgt werden. Weiterhin enthält die Klasse für mindestens all die Methoden, die ehemals geerbt wurden und die vom Programm gebraucht werden, entsprechende Weiterleitungsmethoden an die ehemalige Superklasse. Konstruktoren der Superklasse, die zuvor per `super` aufgerufen wurden, werden jetzt zur Erzeugung der Instanz, an die delegiert wird, aufgerufen. Zuletzt ist die Klasse weiterhin Subklasse von Superklassen ihrer ehemaligen Superklasse, nämlich dann, wenn Zuweisungen im Programm eine entsprechende Zuweisungskompatibilität verlangen, und sie implementiert alle Interfaces, die das Programm (wiederum per Existenz entsprechender Zuweisungen) verlangt. Man beachte, daß durch die verlangten Interfaceimplementierungen u. U. auch Methoden in das *Klasseninterface* aufgenommen und weitergeleitet werden müssen, die von keiner Klientin jemals aufgerufen werden; in diesem Fall ist jedoch das entsprechende Interface zu hinterfragen (und ggf. zu refaktorisieren).

Eine allgemeine, in den Kontext anderer Refactorings eingebundene Beschreibung von REPLACE INHERITANCE WITH DELEGATION finden Sie zusammen mit einem Anwendungsbeispiel in Abschnitt 5.2.4.8. Eine Implementierung des Refactorings für JAVA und das ECLIPSE JDT finden Sie unter http://www.fernuni-hagen.de/ps/prjs/RIWD.

5.2 Eine Auswahl von Refactorings

Leider gibt es für längst nicht alle bekannten Refactorings heute schon Implementierungen, die den Ablauf automatisieren. Wie das vorangegangene ausführliche Beispiel klargemacht haben sollte, liegt das nicht am mangelnden Interesse der Programmiererinnen an diesen Refactorings — vielmehr zeigt erst der Versuch, ein Refactoring in einem Werkzeug umzusetzen, auf, was alles berücksichtigt werden muß und worin die eigentlichen Schwierigkeiten dabei liegen. Die meiste Änderungsarbeit muß daher immer noch von Hand gemacht werden. Dennoch haben auch Refactorings ohne Implementierung einen Wert für sich: Sie geben nämlich — in strukturierter Form — praktische Beispiele dafür, wie schlechter Code aussieht, wie guter Code aussieht und wie man vom einen zum anderen gelangt. In diesem Sinne trifft die nachfolgende Vorstellung einzelner

Refactorings auch eine Auswahl auf Basis dessen, was eine „gute“ objektorientierte Idiomatik (also eine Art, sich in einem Programm objektorientiert auszudrücken) darstellt.

Ähnlich wie Entwurfsmuster lassen sich auch Refactorings nach ihrem Inhalt klassifizieren. Im folgenden werden Beispiele aus den folgenden Kategorien besprochen:

1. Bedingungen vereinfachen
2. Lesbarkeit verbessern
3. Daten organisieren
4. Generalisierung einsetzen
5. Methoden organisieren

Die Darstellung orientiert sich dabei an [37].

5.2.1 Bedingungen vereinfachen

Die Kontrollogik eines Programms verursacht häufig einen großen Teil seiner Komplexität. Dabei muß nach Fred Brooks zwischen der natürlichen Komplexität, die in der Natur des Problems begründet ist, und der künstlichen Komplexität, die durch eine nichtideale Umsetzung entsteht, unterschieden werden.[54] Letztere kann reduziert werden, indem man die Bedingungen, die die Kontrollogik eines Programms ausmachen, vereinfacht.

5.2.1.1 Verschachtelte Bedingungen durch Wächter ersetzen (REPLACE NESTED CONDITIONAL WITH GUARD CLAUSES)

Wer hat das Problem noch nicht selbst erlebt: Eine zunächst einfache Fallunterscheidung muß immer mehr Spezialfälle berücksichtigen, so daß immer neue Else-if-Zweige hinzugefügt und bestehende Bedingungen mit Und-, Oder, und/oder Nicht-Ausdrücken verfeinert werden müssen. Am Ende steht man vor einem unüberschaubaren Wust aus Zweigen, der sich trotz bester Einrückungspraxis nicht mehr nachvollziehen läßt und in dem Fehler zu beheben wie eine Sisyphusarbeit anmutet: Kaum paßt die eine Bedingung, stimmt es an einer anderen Stelle nicht mehr. Das folgende Beispiel mag in dieser Hinsicht als harmlos angesehen werden:

```
1089  double getPayAmount() {
1090        double result;
1091        if (_isDead) result = deadAmount();
1092        else {
1093           if (_isSeparated) result = separatedAmount();
1094           else {
1095              if (_isRetired) result = retiredAmount();
1096              else result = normalPayAmount();
1097           };
1098        }
```

54 FP Brooks „No silver bullet: essence and accidents of software engineering“ *IEEE Computer* 20:4 (1987) 10–19.

```
1099        return result;
1100    };
```

Eine einfache, aber recht effektive Art, dieses Problem zu lösen, ist, die Else- und Else-if-Teile aufzulösen und statt dessen nur noch Ifs zu verwenden. Deren Bedingungen müssen natürlich komplexer sein, da sie die beim Else implizite Negation des vorangegangenen Ifs wiederholen müssen. Auf der anderen Seite ist so für jeden Zweig die Bedingung, die für dessen Ausführung erfüllt sein muß, unmittelbar dem Zweig zugeordnet (und muß nicht umständlich und fehleranfällig aus dem gesamten Verzweigungskomplex zusammengesucht werden). Da die Bedingung die Anweisungen gewissermaßen (vor der Ausführung) schützt, spricht man auch von einem Guard bzw. von *Guarded commands* (so genannt von Dijkstra). Untereinander hingeschrieben entsprechen die Guarded commands den Einträgen in einer Wahrheitstabelle, wobei im Ergebnisteil natürlich beliebige Anweisungen (Commands) (anstelle von Wahrheitswerten) eingetragen sein können und all die Zeilen, in denen keine Anweisung steht, weggelassen werden. Eine solche Wahrheitstabelle (durch Einfügung von Don't cares weiter vereinfacht) ist die nachfolgende:

_isDead	_isSeparated	_isRetired	Anweisungen
true	don't care	don't care	result = deadAmount()
false	true	don't care	result = separatedAmount()
false	false	true	result = retiredAmount()
false	false	false	result = normalPayAmount()

Diese Tabelle läßt sich in folgenden Code übersetzen:

```
1101    if (_isDead) result = deadAmount();
1102    if (! _isDead && _isSeparated) result = separatedAmount();
1103    if (! _isDead && !_isSeparated && _isRetired) result =
                                                        retiredAmount();
1104    if (! _isDead && !_isSeparated && !_isRetired) result =
                                                      normalPayAmount();
```

Im gegebenen Beispiel läßt sich zusätzlich ausnutzen, daß die Ausführung einer Methode jederzeit durch Rückgabe (Return) abgebrochen werden kann. Zwar läßt sich diskutieren, ob ein Return inmitten einer Methode (oder an mehreren Stellen einer Methode) noch der strukturierten Programmierung entspricht (nach der jedes Konstrukt genau einen Eingang und einen Ausgang haben sollte), man wird aber wohl zustimmen, daß die Lesbarkeit hier keinen Schaden nimmt.

```
1105    double getPayAmount() {
1106        if (_isDead) return deadAmount();
1107        if (_isSeparated) return separatedAmount();
1108        if (_isRetired) return retiredAmount();
1109        return normalPayAmount();
1110    };
```

5.2.1.2 Bedingung zerlegen (DECOMPOSE CONDITIONAL)

Manchmal ist schon *eine* Bedingung so kompliziert, daß die Lesbarkeit darunter leidet. Wenn dann noch dazukommt, daß die bedingten Aktionen nicht selbsterklärend sind, kann dieses Refactoring helfen. Aus

```
1111  if (date.before (SUMMER_START) || date.after(SUMMER_END))
1112        charge = quantity * _winterRate + _winterServiceCharge;
1113  else
1114        charge = quantity * _summerRate;
```

wird dann

```
1115  if (notSummer(date))
1116        charge = winterCharge(quantity);
1117  else
1118        charge = summerCharge (quantity);

1119  private boolean notSummer(Date date) {
1120        return date.before(SUMMER_START) || date.after(SUMMER_END);
1121  }

1122  private double summerCharge(int quantity) {
1123        return quantity * summerRate;
1124  }

1125  private double winterCharge(int quantity) {
1126        return quantity * _winterRate + _winterServiceCharge;
1127  }
```

Die Namen der eingefügten Methoden haben den Charakter ausführbarer Kommentare. Das Refactoring löst damit auf einfache Weise das Problem, daß man bei eingefügten Kommentaren oft nicht so genau weiß, auf welchen Teil des Quellcodes sie sich beziehen, ein Problem, das man nur durch längliche Formulierungen oder Wiederholung von Teilen des Quellcodes im Kommentar lösen kann.

Man könnte argumentieren, daß die bessere Lesbarkeit teuer, nämlich um den Preis der verlangsamten Ausführung erkauft wird. Ein geschickter Compiler wird jedoch versuchen, die Methodenaufrufe zu inlinen (also an Ort und Stelle — in der Zeile — einzufügen), so daß keine Verschlechterung des Laufzeitverhaltens entsteht. Außerdem hilft die durchgeführte Fragmentierung, doppelten Code — der von manchen als die Wurzel allen Übels angesehen wird — zu vermeiden: Wenn an anderer Stelle dieselbe Bedingung oder Aktion noch einmal benötigt werden sollte, kann man direkt darauf zurückgreifen. Ist das jedoch nicht der Fall, wird die Klasse allerdings mit Methoden überfrachtet, deren Nutzen außerhalb des Kontextes der Bedingung nicht ersichtlich ist, was die Lesbarkeit in gewisser Weise verschlechtert. Eine Lösung könnte hier die Möglichkeit der Definition von lokalen Methoden (Methoden, die innerhalb von Methoden deklariert sind) bieten; diese besteht in JAVA aber leider nicht.

Dieses Refactoring ist übrigens eine konkrete Anwendung des Methode-extrahieren-Refactorings, das in Abschnitt 5.2.5.1 behandelt wird.

5.2.1.3 Bedingung durch Polymorphismus ersetzen (REPLACE CONDITIONAL WITH POLYMORPHISM)

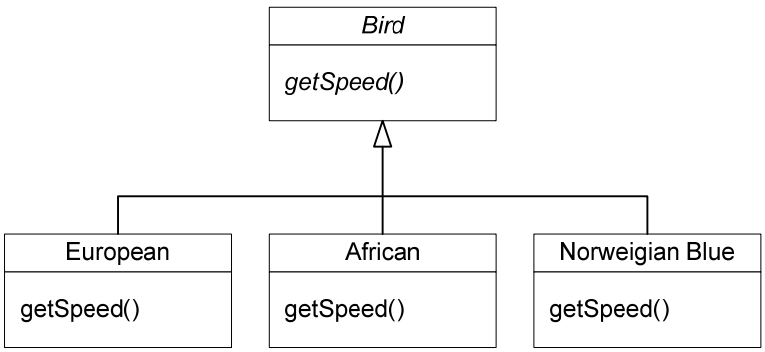
Nicht selten hängt eine Fallunterscheidung am Typ eines Objekts. Im folgenden (nicht ganz ernstgemeinten) Beispiel (übernommen aus [37]) ist das der Fall:

```
1128  double getSpeed() {
```

```
1129    switch (_type) {
1130      case EUROPEAN:
1131        return getBaseSpeed();
1132      case AFRICAN:
1133        return getBaseSpeed() - getLoadFactor() *
1134          _numberOfCoconuts;
1135      case NORWEGIAN_BLUE:
1136        return (_isNailed) ? 0 : getBaseSpeed(_voltage);
1137    }
1138    throw new RuntimeException ("Should be unreachable");
1139  }
```

Die Variable `_type` soll hier offensichtlich nur einen von drei verschiedenen Werten annehmen können: `EUROPEAN`, `AFRICAN` oder `NORWEIGIAN_BLUE`. In Abhängigkeit vom jeweiligen Wert, der wohl eine Vogelart repräsentiert, wird der Wert für eine Variable `speed` berechnet und zurückgegeben. Da es sich bei der Variable `_type` um ein Attribut (Feld) des Objektes handelt, das die Methode `getSpeed()` implementiert (dessen Speed also berechnet werden soll), ist davon auszugehen, daß es sich bei dem Objekt um einen Vogel handelt, der eben von einem der drei genannten Arten (Typen) sein kann.

In der objektorientierten Programmierung würde man genau diesen Umstand, nämlich daß es drei verschiedene Arten von Vögeln gibt, dadurch ausdrücken, daß man eine abstrakte Klasse `Bird` vorsieht und von ihr drei konkrete Klassen, `European`, `African` und `NorweigianBlue`, ableitet, wie im nachfolgenden UML-Diagramm dargestellt. Die abstrakte Methode `getSpeed()`, die in der gemeinsamen Superklasse deklariert ist, wird dann in den drei Unterklassen konkret überschrieben. Die explizite Verzweigung der Switch-Anweisung wird damit durch die implizite Verzweigung des *dynamischen Bindens* (der technischen Umsetzung der Polymorphie) ersetzt.



Refaktorisiert sieht die obige Methode dann so aus:

```
1140  abstract class Bird {
1141        abstract double getSpeed();
1142  }

1143  class European extends Bird {
1144        double getSpeed() {
1145           return getBaseSpeed();
1146        }
1147  }

1148  class African extends Bird {
1149        double getSpeed() {
1150           return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
```

```
1151        }
1152    }

1153    class NorweigianBlue extends Bird {
1154        double getSpeed() {
1155            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
1156        }
1157    }
```

Die refaktorisierte Version hat gleich mehrere Vorteile:

1. Mit der Anzahl der zu unterscheidenden Fälle (Typen) steigt die Länge der Switch-Anweisungen. Einzelne Methoden können dadurch sehr lang werden, was in der objektorientierten Programmierung einigermaßen verpönt ist (vgl. Kurs 01814). Bei der polymorphismusbasierten Lösung wird hingegen jeder Fall einzeln (in einer anderen Klasse) abgehandelt und die Methoden bleiben entsprechend kurz.
2. Häufig bleibt es nicht bei *einer* Fallunterscheidung nach dem Typ: Switch-Anweisungen der obigen Art, die große Redundanzen enthalten, durchziehen dann den Code. Dies ist insbesondere dann ein Problem, wenn sich die Zahl der zu unterscheidenden Arten (Typen) irgendwann einmal ändert: Dann müssen alle Switch-Anweisungen nachgepflegt werden — wehe der, die eine vergißt! Bei der polymorphismusbasierten Lösung hingegen wird beim Einfügen eines neuen Typs (repräsentiert durch eine neue konkrete Klasse) vom Compiler erzwungen, daß alle in der Superklasse abstrakt deklarierten Methoden auch implementiert werden und die Fallunterscheidung damit stets vollständig getroffen wird.
3. Nicht zuletzt ergibt sich aus dem Polymorphismusansatz eine bessere Erweiterbarkeit: Während bei einer Erweiterung des Aufzählungstyps und entsprechend der Switch-Anweisungen bereits bestehender Quellcode (auf den man unter Umständen gar keinen oder nur eingeschränkten Zugriff hat) geändert werden muß, kann man im anderen Fall einfach eine neue Klasse hinzufügen — im günstigsten Fall muß an bereits bestehendem Code überhaupt nichts geändert werden.

Selbsttestaufgabe 5.2

Unter welchen Voraussetzungen muß am Code nichts geändert werden?

Dem gegenüber steht aber auch ein durchaus gravierender Nachteil:

- Da die einzelnen Fälle und damit die Fallunterscheidung als Ganzes auf mehrere Klassen verteilt ist, ist es schwierig, sich einen Überblick über alle Alternativen (die ja gegebenenfalls noch nicht einmal vollständig bekannt sind; siehe Punkt 3 oben) zu verschaffen: Man muß schon alle Klassen nebeneinander legen, um zu sehen, worin sich die verschiedenen Fälle unterscheiden. Damit einhergehend (und nicht selten als Hauptnachteil der objektorientierten Programmierung bezeichnet) ist der Umstand, daß das Programm bei der schrittweisen Ausführung (beim Debuggen) wild hin

und her springt und man schnell den Überblick darüber verliert, wo man gerade ist (und wie man dort hingekommen ist; vgl. Kurs 01814).

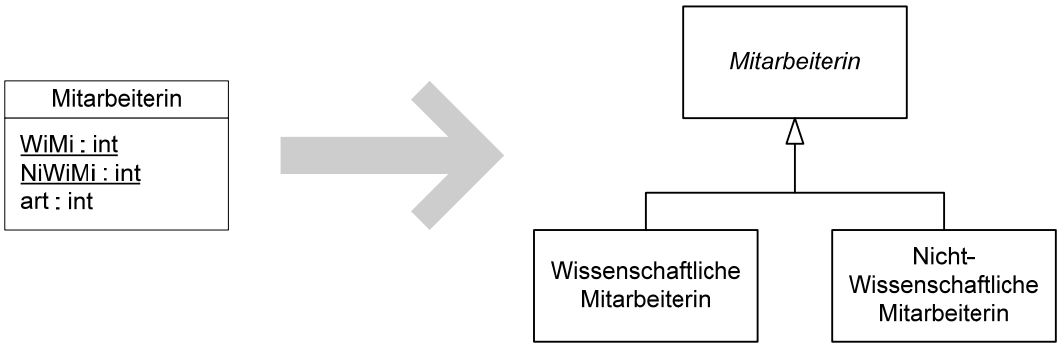
Zusammenfassung

Es ergibt sich also, daß die Fallunterscheidungen jeweils anders gruppiert werden: Bei einer (expliziten) Verzweigung stehen alle Fälle an einer Stelle, aber mehrere Fallunterscheidungen mit den gleichen Alternativen über den Code verstreut; bei der Polymorphie stehen die Fälle an verschiedenen Stellen (Klassen), aber die sonst verstreuten Fallunterscheidungen stehen (nach Alternativen geordnet) zusammen (in Klassen, die die Alternativen repräsentieren).

konsequente Anwendung in SMALLTALK

Das Prinzip, den Polymorphismus an die Stelle der Bedingung bzw. der bedingten Verzweigung zu setzen, wurde übrigens in der Programmiersprache SMALLTALK auf die Spitze getrieben: Da dort alles, also auch die Wahrheitswerte `true` und `false`, ein Objekt ist, kann man auf die If-Anweisung ganz verzichten, einfach indem man eine abstrakte Klasse `Boolean` mit zwei konkreten Subklassen `True` und `False` vorsieht, die jeweils verschiedene Implementierungen für die (in `Boolean` abstrakt deklarierten) Methoden `ifTrue` und `ifFalse` vorsehen (wobei der vom Ergebnis abhängig auszuführende Code als Parameter an die jeweilige Methode übergeben werden muß).

Anmerkung: Stark mit diesem Refactoring verwandt ist „Typcode durch Subtypen ersetzen“ (REPLACE TYPE CODE WITH SUBCLASSES):



5.2.1.4 Einführung eines Nullobjekts (INTRODUCE NULL OBJECT)

Gewissermaßen ein Spezialfall des vorgenannten Refactorings stellt die Einführung eines Nullobjekts dar. Häufig ist es zulässig (entspricht es der Realität), daß ein Objekt als Attributwert anstelle eines anderen eben auch kein Objekt haben kann. So kann es beispielsweise sein, daß jemand nicht verheiratet ist (und damit eben keine Ehepartnerin hat), daß eine Kundin kein Konto hat etc. In solchen Fällen muß vor einem Zugriff auf das Attributobjekt geprüft werden, ob es überhaupt vorhanden ist oder ob statt dessen ein Nullwert vorliegt. Der Nullwert wäre dann entsprechend anders zu behandeln.

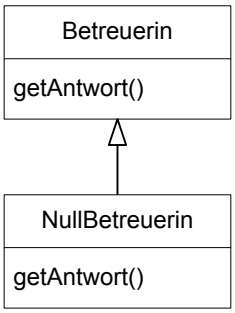
Das nachfolgende Codefragment zeigt ein Beispiel aus einem gedachten Kursbetreuungssystem:

```
1158  Betreuerin betreuerin = kurs.betreuerin;
1159  if (betreuerin == null)
1160        antwort = "leider keine da, die eine Antwort wüßte";
1161  else
```

```
1162        antwort = betreuerin.getAntwort();
```

Wesentlich an dem Beispiel ist, daß für den Fall des Vorliegens eines Nullwertes eine wirkliche Alternative vorgesehen ist und nicht nur ein für den Normalfall gedachter Codeblock übersprungen wird.

Diese Fallunterscheidung läßt sich nun eliminieren (korrekter: durch Polymorphismus ersetzen), indem man eine spezielle Klasse `NullBetreuerin` einführt, die angibt, was zu tun ist, wenn zu einem Kurs keine Betreuerin (die Nullbetreuerin) vorliegt. Das dazugehörige Klassendiagramm kann so aussehen:



Anmerkung: Es könnte aber auch eine abstrakte Klasse `Betreuerin` mit zwei oder mehr konkreten Subklassen, darunter `EchteBetreuerin` und `NullBetreuerin` vorgesehen werden. Da im gegebenen Fall die Klasse `NullBetreuerin` ein Singleton ist (nur genau eine Instanz hat; s. Hinweis unten), ist dies vielleicht etwas übertrieben.

Das obige Codefragment würde damit auf das folgende zusammenschrumpfen:

```
1163  Betreuerin betreuerin = kurs.betreuerin;
1164  antwort = betreuerin.getAntwort();
```

Dafür wäre dann noch das folgende zu implementieren:

```
1165  class NullBetreuerin extends Betreuerin {
1166        String getAntwort() {
1167           return "leider keine da, die eine Antwort wüßte";
1168        }
1169  }
```

Diese Klasse würde dann ggf. noch um zusätzliche Methoden zur Behandlung von weiteren Spezialfällen bei nicht vorhandenen Betreuerinnen ergänzt. Im Gegensatz zu den möglichen Problemen des allgemeineren Refactorings oben (Bedingung durch Polymorphismus ersetzen) hat dies nicht den Nachteil, daß der Code weit verstreut wird: es gibt nur den einen Sonderfall (den Nullwert) und die Behandlung dessen wird an einer Stelle konzentriert.

Hinweis: Da es im allgemeinen nicht sinnvoll sein wird, mehrere Instanzen einer solchen Nullklasse zuzulassen, wird man diese in Form des *SINGLETON Pattern* realisieren (siehe [39]). Da `NullBetreuerin` zudem eng an `Betreuerin` gebunden ist (und außerhalb des Kontexts von `Betreuerin` wohl kaum Verwendung finden wird), kann man sich überlegen, ob man erstere nicht als innere Klasse

implementieren will. Die äußere Klasse (im gegebenen Fall `Betreuerin`) würde dann nur noch eine Konstante (das Nullobjekt) öffentlich machen.

5.2.1.5 Zusicherung einfügen (INTRODUCE ASSERTION)

Das Design by contract verlangt, daß jede Methode die Voraussetzungen für ihr Gelingen mittels Vorbedingung explizit macht. Das sog. *defensive Programmieren* verlangt dagegen, daß mögliche Zustände (Eingaben etc.), mit denen eine Methode nicht zurechtkommen würde, in dieser Methode geprüft werden und bei Vorliegen entsprechend (fehlertolerant) reagiert wird[55]. In der Praxis wird jedoch häufig weder das eine noch das andere gemacht.

Wann immer man findet, daß eine bestimmte, im Code implizite Bedingung Voraussetzung für die fehlerfreie Ausführung des Codes (einer Codesequenz) ist, sollte man diese Bedingung explizit machen, wenn schon nicht durch Vorbedingungen à la Design by contract oder Guards, dann durch das Einstreuen von allgemeinen Zusicherungen (Asserts). Der nachfolgende Code, der nur funktioniert, wenn ein Angestellter eine Spesenbeschränkung oder das Projekt, dem er primär zugeordnet ist, eine solche hat, enthält eine solche implizite Bedingung (im Kommentar ausgedrückt):

```
1170  class Employee {
1171        private static final double NULL_EXPENSE = -1.0;
1172        private double _expenseLimit = NULL_EXPENSE;
1173        private Project _primaryProject;

1174        double getExpenseLimit() {
1175           // should have either expense limit or primary project
1176           return (_expenseLimit != NULL_EXPENSE) ?
1177              _expenseLimit :
1178              _primaryProject.getMemberExpenseLimit();
1179        }
1180  }
```

Er wird zu

```
1181        double getExpenseLimit() {
1182           assert (_expenseLimit != NULL_EXPENSE
1183                 || _primaryProject != null);
1184           return (_expenseLimit != NULL_EXPENSE) ?
1185              _expenseLimit :
1186              _primaryProject.getMemberExpenseLimit();
1187        }
```

5.2.2 Lesbarkeit verbessern

5.2.2.1 Methode oder Variable umbenennen (RENAME)

Es ist leider immer noch ein weit verbreitetes Übel, daß Programmiererinnen für die Bezeichner in ihren Programmen Abkürzungen wählen, die außer ihnen selbst kaum jemandem (und manchmal nach gewisser Zeit selbst ihnen nicht mehr) zugänglich sind. Dies mag hier und da gute Gründe haben (ausführliche

[55] was jedoch nicht immer möglich ist; s. Abschnitt 2.4

Namen werden unter Umständen zu lang und führen deswegen zu unerwünschten Zeilenumbrüchen, bestimmte Abkürzungen haben sich eingebürgert), aber die Lesbarkeit des Programms sollte eigentlich für gar nichts geopfert werden. Überhaupt nicht hinzunehmen sind Argumente wie „Ja glauben Sie denn, ich tippe mir die Finger wund?“: Heutige Entwicklungsumgebungen nehmen einem durch die Funktion der automatischen Vervollständigung (sog. Code completion) die meiste Schreibarbeit sowieso schon ab, und bei der Einführung neuer Bezeichner wird man einem gebildeten Mitmenschen noch abverlangen können, sich allgemeinverständlich auszudrücken. Bei der Verwendung kryptischer Abkürzungen hingegen kann niemand mehr durch bloßes Hinsehen entscheiden, ob sich die Programmiererin vielleicht im Bezeichner geirrt hat und aus Versehen die falsche Methode aufruft oder das falsche Feld referenziert. Wann immer also jemand einen unverständlichen oder irreführenden Bezeichner in einem Programm findet, sollte sie ihn schleunigst ändern.

Eine Methode sollte immer danach benannt werden, *was* sie tut, nicht danach, *wie* sie es tut. Wie sie es tut ist nur von Interesse, wenn man etwas daran ändern will. Dazu müssen Sie aber an die Definitionsstelle der Methode gehen, sich die Implementierung ansehen und diese verstehen. Wenn Sie die Implementierung verstanden haben, wissen Sie auch, wie die Methode etwas tut, und zwar viel genauer, als das irgendein Name sagen könnte. An der Aufrufstelle einer Methode ist hingegen nur von Interesse, was sie tut; so bleibt der Fokus in der Umgebung der Aufrufstelle, die die Implementierung der *aufrufenden* Methode darstellt, auf der Erklärung von deren Wie.

Leider ist das mit dem Ändern von Bezeichnern nicht ganz so einfach. Ändert man nämlich den Namen einer Methode nur lokal, so können die Aufrufe der Methode an anderen Stellen nicht mehr gebunden werden. Außerdem muß geprüft werden, ob diese Methode eine andere, geerbte überschreibt, und ob sie ggf. (in Unterklassen) überschrieben wird. Schließlich muß überprüft werden, ob es nicht schon eine Methode mit gleichem Namen gibt; wenn dies überhaupt erlaubt ist, kann die so neu entstehende Überladung zu Problemen führen, die vorher nicht da waren[56]. All diese Feinheiten sind auch der Grund dafür, warum man tunlichst davon Abstand nehmen sollte, den Namen einer Methode per globalem Suchen und Ersetzen zu ändern. Das würde nämlich schon daran scheitern, daß einfaches Suchen und Ersetzen zufällige von beabsichtigter Namensgleichheit nicht unterscheiden kann.

Glücklicherweise besitzen moderne Entwicklungsumgebungen bereits ein Refactoring, das die Umbenennung automatisch vornimmt.

5.2.2.2 Parameterklasse einführen (INTRODUCE PARAMETER OBJECT)

Wenn eine Methode mit vielen Parametern aufgerufen werden muß, die inhaltlich stark zusammenhängen, oder wenn mehrere Methoden mit genau den gleichen Parametern aufgerufen werden, kann es sinnvoll sein, diese Parameter in

[56] z. B. einen Method-ambiguous-Error

einem Record zusammenzuführen. In der objektorientierten Programmierung wird man dafür eine Klasse nehmen, die keine eigenen Methoden hat, da ihr einziger Zweck ist, die Parameter zu gruppieren.

5.2.2.3 Konstruktor durch eine Factory-Methode ersetzen (REPLACE CONSTRUCTOR WITH FACTORY METHOD)

In SMALLTALK sind Konstruktoren Klassenmethoden (also Methoden, die in JAVA oder C# als `static` deklariert würden), die eine neue Instanz der Klasse zurückgeben (letztendlich realisiert von der Klassenmethode `new` in der Klasse `Object`). In JAVA und C# hingegen sind Konstruktoren besondere Methoden, deren Name und Rückgabetyp durch die Klasse, in der sie definiert werden, festgelegt ist. Nicht selten kommt es jedoch vor, daß der Typ einer zurückgegebenen Instanz von anderen Faktoren abhängt, etwa den Parametern des Konstruktoraufrufs, oder daß vielleicht gar kein neues Objekt zurückgegeben werden soll, sondern vielmehr eines, das bereits existiert (das *SINGLETON Pattern* [39]). Konstruktoren sind dafür nicht flexibel genug.

Allgemein wird es gelegentlich als guter Stil erachtet, anstelle von Konstruktoren statische Methoden zu verwenden, die eine neue Instanz zurückgeben. Da diese Methoden wesentlich flexibler sind, was das zurückgegebene Objekt (insbesondere seinen Typ) angeht, sie insbesondere das Objekt nach den Vorgaben der Parameter zusammenbauen können, nennt man sie auch Factory-Methoden (vgl. Abschnitt 4.4.6). Natürlich können diese Factory-Methoden ein neues Objekt nicht selbst erzeugen; sie müssen dazu einen Konstruktor der Klasse aufrufen. Es reicht dafür jedoch der Standardkonstruktor. So wird beispielsweise aus dem Konstruktor

```
1188        Employee (int type) {
1189           _type = type;
1190        }
```

die Factory-Methode

```
1191        static Employee create(int type) {
1192           return new Employee(type);
1193        }
```

die den ursprünglichen Konstruktor oder, besser noch,

```
1194        static Employee create(int type) {
1195           Employee employee = new Employee();
1196           employee._type = type;
1197           return employee;
1198        }
```

die den Standardkonstruktor aufruft. Der Standardkonstruktor kann dann `protected` deklariert werden, um die Umleitung aller Instanzerzeugungen auf die Factory-Methode zu erzwingen.

Richtig interessant wird die Einführung einer Factory-Methode allerdings erst dann, wenn die Parameter ausgewertet und davon abhängig Objekte verschiedener Typen zurückgegeben werden, wie etwa in

```
1199        static Employee create(int type) {
1200           switch (type) {
1201              case 1: return new UnskilledEmployee();
1202              case 2: return new Worker();
1203              case 3: return new Manager();
1204           }
1205        }
```

Dieser Code läßt sich noch weiter verkürzen, indem man die *Reflection*-Fähigkeit von JAVA ausnutzt (vgl. Abschnitt 6.1):

```
1206  static Employee create (String name) {
1207        try {
1208           return (Employee) Class.forName(name).newInstance();
1209        }
1210        catch (Exception e) {
1211           throw new IllegalArgumentException("Unable to instantiate");
1212        }
1213  }
```

5.2.2.4 Fehlercode durch Ausnahme ersetzen (REPLACE ERROR CODE WITH EXCEPTION)

Viele gängige Bibliotheksroutinen (wie auch Betriebssystemfunktionen) verwenden sog. Fehlercodes, um anzuzeigen, daß mit der Ausführung etwas nicht geklappt hat. Diese Fehlercodes können gemischt mit den richtigen Rückgabewerten einer Methode auftreten (sie stellen also quasi eine Erweiterung des Wertebereichs einer Funktion dar), oder sie verbannen den eigentlichen Rückgabewert einer Methode in die Liste der Parameter, wobei dann ein spezieller Rückgabewert (oder Fehlercode) anzeigt, daß keine Fehler aufgetreten sind.

Im Beispiel eines Kontos (`Account`) und der Methode Abheben (`withdraw`) sieht das so aus:

```
1214  class Account {
1215        private int _balance;

1216        int withdraw(int amount) {
1217           if (amount > _balance)
1218              return -1;
1219           else {
1220              _balance -= amount;
1221              return 0;
1222           }
1223        }
      …
```

An der Aufrufstelle findet man dann solchen Code:

```
1224  if (account.withdraw(amount) == -1)
1225        handleOverdrawn();
1226  else
1227        doTheUsualThing();
```

Diese Praxis, so etabliert sie auch sein mag, hat verheerende Auswirkungen auf die Lesbarkeit von Programmen. Das größte Problem dabei ist, daß der Code, der den regulären Ablauf einer Routine beschreibt, mit Code zur Fehlerbehandlung durchsetzt wird, ohne daß das eine vom anderen klar getrennt wäre. Das zweite

Problem ist, daß Fehler, die in einer geschachtelten Struktur irgendwo tief unten auftreten, unter Umständen zum Abbruch vieler äußerer Blöcke führen, so daß große Codestrecken, die unmittelbar gar nichts mit dem auftretenden Fehler zu tun haben, mit Bedingungen versehen (geschützt; *Guarded commands*!) werden müssen, um eine Ausführung nach Auftreten des Fehlers zu verhindern bzw. um alternative Maßnahmen vorzusehen.

Um genau dies zu verhindern, also um den Code zur regulären Ausführung von dem zur Fehlerbehandlung syntaktisch klar zu trennen und um sich die Durchsetzung von Code mit Verzweigungen zur Berücksichtigung von Fehlercodes (in der Regel zum Abbruch eines Blocks) zu ersparen, wurde das Konzept der Exceptions und des Exception handlings eingeführt.

Im einfachsten Fall wird aus obiger Methode `withdraw` mit einem Fehlercode als Rückgabe eine Methode ohne Rückgabewert, dafür mit der Möglichkeit, eine Exception zu werfen:

```
1228        void withdraw(int amount) throws BalanceException {
1229           if (amount > _balance)
1230              throw new BalanceException();
1231           _balance -= amount;
1232        }
```

Verzweigungen auf der Basis von If-Anweisungen, die den Fehlercode auswerten, entfallen damit vollständig, sie werden durch Catch-Klauseln oder durch weitere Throws-Deklarationen im Kopf der aufrufenden Methoden ersetzt. Der resultierende Code an der Aufrufstelle ist also entweder

```
1233  try {
1234        account.withdraw(amount);
1235        doTheUsualThing();
1236  } catch (BalanceException e) {
1237        handleOverdrawn();
1238  }
```

oder

```
1239  void someMethod() throws BalanceException {
1240        account.withdraw(amount);
1241        doTheUsualThing();
1242  }
```

In gewisser Weise erweitert die Deklaration, (möglicherweise) Exceptions zu werfen, die Menge der möglichen Rückgabewerte einer Methode. Folgerichtig müßte die Menge der deklarierten Exceptions einer Methode zu ihrer Signatur zählen, wenn in JAVA nur der Rückgabetyp auch zur Signatur gehörte.

Das Exception handling stellt einen der größten Fortschritte in der Programmierung seit der Einführung der strukturierten Programmierung dar. Auch wenn es manchmal lästig erscheint (psychologisch schon allein deswegen, weil es ja nur die Fälle behandelt, die man eigentlich gar nicht haben möchte), sollte man es sich zur Gewohnheit machen, beim Schreiben einer Methode auch an die möglichen Ausnahmen zu denken und diese entsprechend zu deklarieren. Im schlimmsten Fall, also wenn man sich wirklich nicht um die Ausnahme küm-

mern möchte oder keine Ahnung hat, wie man dies sinnvoll tun könnte, kann man im Exception handler dann immer noch eine Fehlermeldung im Klartext ausgeben und somit nachfolgenden Programmiererinnen Hinweise geben, wo der Fehler aufgetreten ist und daß man ihn vielleicht noch abstellen müßte. (Siehe in diesem Zusammenhang auch das nachfolgende Refactoring „Ausnahme durch Vorbedingung ersetzen“).

Man sieht gelegentlich, daß das Exception handling mißbraucht wird, um Blöcke abzubrechen, obwohl das verwendete Abbruchkriterium ein regelmäßiges ist, also nichts mit einer Ausnahme zu tun hat. So kommt es beispielsweise vor, daß Programmiererinnen über die Elemente eines Arrays iterieren, ohne ein Abbruchkriterium für das Erreichen der oberen Array-Grenze vorzusehen, und statt dessen die Index-out-of-bounds-Exception ausnutzen, um die Schleife zu beenden:

```
1243  try {
1244        for (i=0;; i++)
1245           System.out.println(line[i]);
1246  } catch (IndexOutOfBoundsException e) {}
```

Dies ist natürlich ganz schlechter Stil und sollte sich jeder von selbst verbieten. Man trifft jedoch auch auf weniger offensichtliche Fälle; hierfür ist dann das übernächste Refactoring „Ausnahme durch Test ersetzen“ gedacht.

5.2.2.5 Ausnahme durch Vorbedingung ersetzen (REPLACE EXCEPTION WITH PRECONDITION)

Nicht immer ist die Signalisierung einer Ausnahme gerechtfertigt. So kann man beispielsweise argumentieren, daß der Versuch, sein Bankkonto zu überziehen, regulären Charakter hat und das Werfen einer Ausnahme einen Mißbrauch des Konzepts darstellt (wenn auch nicht ganz so kraß wie in den Programmzeilen 1243–1246 in Abschnitt 5.2.2.4). Ähnlich zum obigen Beispiel ersetze man dann in

```
1247  class Account {
1248        private int _balance;

1249        int withdraw(int amount) {
1250           if (amount > _balance)
1251              return -1;
1252           else {
1253              _balance -= amount;
1254              return 0;
1255           }
1256        }
      …
```

die Methode `withdraw(.)` durch

```
1257  void withdraw(int amount) {
1258        assert canWithdraw(amount) : "Amount too large";
1259        _balance -= amount;
1260  }

1261  boolean canWithdraw(amount) {
1262        return amount > _balance;
1263  }
```

An der Aufrufstelle muß dann die Einhaltung der Vorbedingung sichergestellt werden:

```
1264  if (!account.canWithdraw(amount))
1265        handleOverdrawn();
1266  else {
1267        account.withdraw(amount);
1268        doTheUsualThing();
1269  }
```

Die Aufrufstelle unterscheidet sich damit nicht übermäßig von der mit Fehlercode (Programmzeilen 1224–1227 in Abschnitt 5.2.2.4); zwar ist die Überprüfung auf Fehler (`!account.canWithdraw(amount)`) von der Ausführung der eigentlichen Aktion getrennt (`account.withdraw(amount)`), aber die Fehlerbehandlung steht direkt neben der Behandlung des Regelfalls, ist insbesondere nicht syntaktisch (durch einen Catch-Block) hervorgehoben. Dies entspricht im gegebenen Fall aber gerade der Absicht (es sollte sich bei der Überziehung ja auch um einen regulären Fall handeln).

5.2.2.6 Ausnahme durch Test ersetzen (REPLACE EXCEPTION WITH TEST)

Im nachfolgenden Fall ist die Provokation einer Exception und die Verwendung eines Catch-Blocks mißbräuchlich:

```
1270  class ResourcePool {
1271        Stack<Resource> _available;
1272        Stack<Resource> _allocated;

1273        Resource getResource() {
1274           Resource result;
1275           try {
1276              result = _available.pop();
1277              _allocated.push(result);
1278              return result;`
1279           } catch (EmptyStackException e) {
1280              result = new Resource();
1281              _allocated.push(result);
1282              return result;
1283           }
1284        }
```

Daß die Ressourcen irgendwann erschöpft sind (der Stack `_available` irgendwann leer ist), ist ganz klar zu erwarten. Die refaktorisierte Variante der Methode sieht so aus:

```
1285  Resource getResource() {
1286        Resource result;
1287        if (_available.isEmpty())
1288           result = new Resource();
1289        else
1290           result = _available.pop();
1291        _allocated.push(result);
1292        return result;
1293  }
```

Man beachte, daß `_allocated.push(result)` aus den beiden alternativen Pfaden herausfaktorisiert wurde (ein weiteres Refactoring).

5.4 Weiterführende Literatur

Refactorings sind ein sehr pragmatisches Thema, dessen theoretische Grundlagen dennoch anspruchsvoll sind. Bislang ist es allerdings nicht Gegenstand allzu vieler Forschungsarbeiten geworden, was vermutlich daran liegt, daß die damit verbundenen Probleme und Lösungen mehr oder weniger bekannt sind und man lediglich in Detailfragen noch einen Erkenntnisgewinn erzielen kann. Der Klassiker ist sicher das Buch von Fowler [37] (der einmal mehr den richtigen Riecher gehabt hat); die darin beschriebenen Refactorings, von denen ein Gutteil hier (inklusive der Beispiele) wiedergegeben werden, sind zum Teil jedoch nur recht oberflächlich beschrieben und werden sich in der Praxis nicht immer so (leicht) durchführen lassen. Daß das tatsächlich so ist, wurde in [38] klar gezeigt.

Interessant sind sicher noch die hier ausgelassenen sog. *großen Refactorings* (ebenfalls in [37] beschrieben) sowie der systematische Einsatz von Refactorings zur Verwendung von Entwurfsmustern im Code (*Refactoring to patterns*) [40] (wobei letzteres Werk wohl erst noch einer zweiten Ausgabe bedarf). Ein einsichtsreiches Buch ist auch [41], das allerdings, da es sich auf SMALLTALK bezieht, nur für wirklich Interessierte (und Fans von Kent Beck) sinnvoll zu lesen ist.

Das sehr mächtige Refactoring INFER TYPE, das Grundlage weiterer Refaktorisierungswerkzeuge wie REPLACE INHERITANCE WITH DELEGATION ist, wurde in [42] und [43] ausführlich beschrieben.

[36] W Opdyke *Refactoring Object-Oriented Frameworks* PhD thesis (Urbana-Champaign, IL, USA, 1992).

[37] M Fowler *Refactorings — Improving the Design of Existing Code* (Addison-Wesley, 1999).

[38] H Kegel, F Steimann „Systematically replacing inheritance with delegation in Java“ in: *International Conference on Software Engineering* (2008) 431–440.

[39] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns — Elements of Reusable Software* (Addison-Wesley, 1995). Dublette zu [1].

[40] J Kerievsky *Refactoring to Patterns* (Addison Wesley Professional, 2004).

[41] K Beck *Smalltalk Best Practice Patterns* (Prentice Hall 1996).

[42] F Steimann „The Infer Type refactoring and its use for interface-based programming“, in: *Journal of Object Technology* 6:2 (2007) 67–89.

[43] H Kegel „Constraint-basierte Typinferenz für Java 5“ (Diplomarbeit, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, 2007).