Universität Paderborn, AG Kastens „Programmiersprachen und Übersetzer“,
Fakultät für Elektrotechnik, Informatik und Mathematik

Refactoring – eine Einführung

Eine Ausarbeitung im Rahmen des Seminars

„Refactoring in eXtreme Programming“

von Dietrich Travkin

Februar 2003

Veranstalter:

Prof. Dr. Uwe Kastens

Dipl. Inform. Jochen Kreimer

Refactoring – eine Einführung

1. EINFÜHRUNG

In diesem Seminarbericht werden basierend auf [FBB+00] das Refactoring vorgestellt und dessen Anwendungsgebiete aufgezeigt. Einige Transformationen aus dem Katalog von Martin Fowler werden an Beispielen genauer erläutert. Anschließend wird auf die Schwierigkeiten und Grenzen von Refactoring hingewiesen. Schließlich werden die Herkunft und einige Ähnlichkeiten aus anderen Bereichen beleuchtet.

1.1 Definition

Der Begriff Refactoring bezeichnet das Transformieren von bereits existierendem Code, ohne sein „beobachtbares" Verhalten zu ändern. Dabei bedeutet „beobachtbares Verhalten nicht ändern", dass das vorherige sowie das transformierte Programm bei gleicher Eingabe auch die gleiche Ausgabe erzeugen.

Martin Fowler benutzt in seinem Buch die folgenden von ihm formulierten Definitionen für die Substantivform und die Verbform des englischen Wortes refactoring:

> "*Refactoring* (noun): A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing the observable behavior of the software.
>
> *Refactor* (verb): To restructure software by applying a series of refactorings without changing the observable behavior of the software." [FBB+99, S. 53 ff]

Das Substantiv wird als *Refaktorisierung* ins Deutsche übersetzt, während das Verb als *refaktorisieren* übersetzt wird. Im Folgenden werden die deutschen Begriffe verwendet.

Eine Refaktorisierung ändert also die interne Struktur des bereits funktionierenden Programmcodes mit dem Ziel, das Programmverhalten zu erhalten, aber die Lesbarkeit und Flexibilität des Programmcodes zu erhöhen. Das Refaktorisieren transformiert Software mit Hilfe von einer Reihe einzelner Refaktorisierungen.

Refaktorisierungen sind meist kleinere Änderungen an dem Programmcode, können aber auch andere Refaktorisierungen verwenden.

Besonders betont Martin Fowler die Tatsache, dass Refaktorisierungen den Code verständlicher machen sollen. Änderungen am Code, die diesen nicht verständlicher machen, zählt er nicht zu den Refaktorisierungen. Performance-Optimierungen beispielsweise würden den Programmcode meist komplizierter machen und wären deswegen keine Refaktorisierungen.

Da das Refaktorisieren vor allem in der Entwicklung und Weiterentwicklung eine große Rolle spielt, fasst Kent Beck die Eigenschaften des Refaktorisierens wie folgt zusammen:

> „Refaktorisieren bezeichnet den Vorgang, ein funktionsfähiges Programm zu nehmen und seinen Wert zu erhöhen, indem wir sein Verhalten nicht ändern, aber ihm mehr von diesen Eigenschaften geben, die es uns ermöglichen, es mit hohem Tempo weiterzuentwickeln." [FBB+00, S. 49]

Um sicherzustellen, dass das Refaktorisieren das Programmverhalten tatsächlich nicht ändert und die Entwicklung und Weiterentwicklung beschleunigt, ist es sehr wichtig, dass man eine repräsentative Menge von Testfällen implementiert und das Testen weitestgehend automatisiert. Die im Buch von Martin Fowler beschriebenen Refaktorisierungen beinhalten unter anderem das Testen des transformierten Codes, was sicherstellen soll, dass das Programm sein Verhalten wirklich nicht ändert. Die Tests sind Bestandteil der einzelnen Refaktorisierungen.

Damit die Tests das Refaktorisieren aber nicht aufhalten und der Programmierer mangels durchgeführter Tests nicht zu viel Zeit mit der Fehlersuche verbringt, ist es äußerst wichtig, diese zu automatisieren. Automatisierte Tests können bequem, schnell und dadurch ohne Zeitverlust öfter ausgeführt werden als solche Tests, bei denen der Programmierer die Ausgabe des transformierten Programms mit der Ausgabe des vorherigen funktionierenden Programms bzw. mit der erwarteten Ausgabe vergleichen muss. Ein automatisierter Test kann bei nur einer Ausführung auch mehrere Fälle – im Idealfall alle relevanten Fälle – prüfen. Es ist nicht notwendig und oft auch nicht möglich, den Test für jeden Fall einzeln auszuführen.

1.2 Ziele

Refaktorisieren wurde entwickelt, um die Entwicklung und Weiterentwicklung von Software zu vereinfachen und die Kosten für Wartung zu reduzieren. Deswegen ist das Hauptziel von Refaktorisieren, Programmcode so zu transformieren, dass Änderungen an der Software und das Hinzufügen neuer Funktionalität schnell, einfach und damit kostengünstig durchgeführt werden können. Dieses soll auch während der Wartung möglich sein, bei der beim Wasserfallmodell die höchsten Kosten entstehen.

Dieses Ziel versucht man zu erreichen, indem der bereits funktionierende Programmcode durch Verbesserung der Struktur und Umbenennen für spätere Entwickler übersichtlicher und verständlicher gemacht wird. Das ist ein guter Ansatz, weil Code in der Softwareentwicklung meist öfter gelesen als geschrieben wird. Die Einarbeitung in einen verständlicheren Code benötigt einen geringeren Zeitaufwand.
Genauso verhält es sich mit komplexen Programmstrukturen. Es ist schwierig und teuer, die Struktur großer Programme und ihr komplexes Design bei ihrer Weiterentwicklung zu verstehen und zu erhalten. Deswegen werden Änderungen oft kurzfristig, ohne vollständiges Verständnis des Codedesigns und mit möglichst wenig Aufwand gemacht. Das verkompliziert auf Dauer die Struktur und das Design des Codes. Weiterentwicklungen werden immer schwieriger und das Programm unverständlicher. Durch Refaktorisieren soll die Codestruktur langfristig aufrechterhalten und verbessert, sowie die Redundanz eliminiert werden.

Verständliche Programme mit übersichtlicher klarer Struktur erleichtern es den Programmierern, die Fehlerzahl zu reduzieren. Die Aufteilung der Transformationen in kleine einfache und systematisch durchzuführende Schritte sollen zusätzlich dabei helfen. Wenn weniger Fehler gemacht werden, wird auch weniger Zeit für die Fehlersuche benötigt. Beim Refaktorisieren werden häufig Tests ausgeführt, um die für die Fehlersuche benötigte Zeit noch weiter zu reduzieren. Wenn häufig getestet wird, sind die Änderungen zwischen den Tests klein und im Falle eines Fehlers kann der vorherige Zustand leicht wiederhergestellt werden.

Da beim Refaktorisieren der Software keine neue Funktionalität hinzugefügt wird, das Refaktorisieren aber dennoch Zeit erfordert, soll diese Zeit so kurz wie möglich werden und der gesamte Entwicklungsprozess beschleunigt werden. Dazu sollen die häufig benötigten

Tests automatisiert und die Programme selbsttestend gemacht werden. Dadurch, dass weniger Fehler gesucht werden müssen, die Programme übersichtlicher und verständlicher werden und damit weniger Zeit für die Einarbeitung und Fehlersuche benötigt werden, soll sich auch die gesamte Entwicklungszeit verkürzen.

Nimmt man nun alle oben genannten Eigenschaften einer Software zusammen, so stellt man fest, dass sie die Qualität und damit den Wert der Software erhöhen. Software, die flexibel und schnell änderbar, übersichtlich, verständlich, fehlerminimiert, gut strukturiert und somit qualitativ hochwertig ist, wird zum Ziel der meisten Softwareentwickler und deswegen auch zum Ziel des Refaktorisierens gemacht.

2. ANWENDUNG

2.1 Vorgehen

Das Refaktorisieren wird wie ein Werkzeug in der Softwareentwicklung verwendet, um Programmcode verständlicher oder Änderungen leichter zu machen. Dieses Werkzeug kann sowohl beim Programmiervorgang als auch in anderen Fällen, z.B. um Fehler zu suchen, verwendet werden. Wenn in der Softwareentwicklung Refaktorisieren verwendet wird, dann ist der Entwicklungsprozess ein ständiger Wechsel zwischen dem Transformieren von Code, dem Testen und dem anschließenden Hinzufügen neuer oder Ändern vorhandener Funktionalität.

Wann refaktorisieren?

Der Zeitpunkt für das Refaktorisieren, wird nicht geplant, es ist keine Entwicklungsphase. Das Refaktorisieren ist vielmehr ein Hilfsmittel das während des gesamten Entwicklungsprozesses hilft, den Programmcode so zu transformieren, dass Änderungen einfacher werden. Refaktorisierungen werden meist beim Programmieren vor dem Hinzufügen neuer Funktionalität auf bereits funktionierenden Code angewendet. Martin Fowler schreibt:

> „Sie entscheiden nicht zu refaktorisieren, sondern Sie refaktorisieren, weil Sie etwas anderes machen wollen und das Refaktorisieren Ihnen dabei hilft.“ [FBB+00, S. 46]

Vor dem Hinzufügen neuer Funktionalität

Je nach Programmstruktur kann das Hinzufügen neuer oder das Ändern vorhandener Funktionalität sehr kompliziert sein. Mit Hilfe des Refaktorisierens kann man die Programmstruktur vereinfachen und die Änderungen erleichtern. Einer der Merksätze von Martin Fowler lautet deswegen:

> „Wenn Sie zu einem Programm etwas hinzufügen müssen und die Struktur des Programms erlaubt dies nicht auf einfache Art und Weise, so refaktorisieren Sie zunächst das Programm so, dass Sie die Erweiterung leicht hinzufügen können, und fügen sie anschließend hinzu.“ [FBB+00, S. 6]

Um Code zu verstehen oder Fehler zu finden

Das Refaktorisieren vereinfacht die Einarbeitung in ein unbekanntes Programm und ermöglicht ein höheres Niveau des Verständnisses. Eine unklare unübersichtliche Struktur oder ungeeignete Namen machen ein Programm unnötig kompliziert. Ist ein Programm nur schwer zu verstehen, so kann der verstandene Teil Stück für Stück refaktorisiert und damit vereinfacht werden. Schließlich wird die Struktur des Programms klarer. Auf diese Weise

arbeitet man sich intensiver in den Programmcode ein und versteht meist mehr, als nur durch Lesen von Code.
Dieses Vorgehen ist bei Code-Reviews in kleineren Gruppen nützlich, denn die Reviewer können durch besseres Verständnis auch bessere Änderungsvorschläge machen.

Auch die Fehlersuche wird durch besseres Verständnis des Programms einfacher.

> „Bekommen Sie einen Fehlerbericht, so ist dies ein Zeichen, dass Sie refaktorisieren müssen, weil der Code nicht verständlich genug war, um zu erkennen, dass hier ein Fehler war.“ [FBB+00, S. 47]

Zusätzlich hat man die Möglichkeit, Fehler durch Kapselung einzugrenzen. Die Fehlersuche kann so auf eine oder einige wenige Komponenten beschränkt werden, wenn die anderen Komponenten fehlerfrei funktionieren.

Wie und was refaktorisieren?

Dieser Abschnitt beschreibt die Methodik des Refaktorisierens und gibt Hinweise auf eine Vorkehrung, die das Refaktorisieren beschleunigt und die Fehler minimiert, nämlich das automatisierte Testen.

Tests vor dem Refaktorisieren

Das Refaktorisieren wird besonders sicher, wenn es möglich ist, das Programm nach jeder Transformation zu testen. Bei größeren Refaktorisierungen werden sogar Tests zwischen einzelnen Transformationsschritten nötig. Deswegen ist es sehr wichtig, vor dem eigentlichen Refaktorisieren eine Menge von Testfällen zu implementieren, die sicherstellen soll, dass das transformierte Programm dasselbe Verhalten hat wie vor der Transformation.

Damit das Refaktorisieren nicht durch Tests aufgehalten wird, müssen diese automatisiert werden. Das heißt, das Programm muss selbstüberprüfend sein und die berechneten Ergebnisse mit den erwarteten Ergebnissen eigenständig vergleichen. So verwendet der Programmierer nur wenig Zeit für die Ausführung der Tests, kann diese häufiger durchführen und so die Fehleranzahl reduzieren. Das Refaktorisieren ist dann ein ständiger Wechsel zwischen kleinen Änderungen des Codes und dem Testen.

Da das Testen beim Refaktorisieren eine entscheidende Rolle spielt, wird dieses in einem späteren Kapitel noch genauer erläutert.

Methodik

Das Refaktorisieren ist ein systematisches Vorgehen. Um diese Systematik zu unterstützen und die Anwendung der Refaktorisierungen zu vereinfachen, enthält [FBB+00] einen Katalog von Refaktorisierungen und einen Katalog von sogenannten Codegerüchen. Codegerüche sind Merkmale von Programmcode, die auf schlechten Code hinweisen und dem Programmierer helfen, eine geeignete Anwendungsstelle für eine Refaktorisierung zu finden. Eine Tabelle von Codegerüchen ordnet jedem Geruch Refaktorisierungen zu, die an dieser Stelle angewendet werden können, um das Programm zu verbessern.

Die einzelnen Schritte einer Refaktorisierung sind in dem Katalog von Refaktorisierungen beschrieben. Eine Refaktorisierung benötigt aber häufig die vorherige Ausführung anderer Refaktorisierungen. So kann es beim Refaktorisieren zu einem Zyklus der drei Hauptschritte einer Transformation kommen (siehe folgende Seite).

1. **Identifikation der Anwendungsstelle mit Hilfe der Codegerüche**
2. **Wahl einer geeigneten Transformation anhand der Codegeruch-Tabelle**
3. **Systematisches, schrittweises Anwenden der Transformation und Testen**

Zuerst versucht man mit Hilfe der Codegeruch-Merkmale eine Stelle zu finden, die refaktorisiert werden sollte. Die Codegerüche werden im folgenden Kapitel erklärt. Dann schlägt man in der Tabelle der Codegerüche nach, welche Refaktorisierungen an dieser Stelle geeignet sein könnten und wählt mit Hilfe ihrer Kurzbeschreibungen im Katalog eine für das eigene Vorhaben passende Refaktorisierung aus. Schließlich führt man systematisch die im Katalog beschriebenen Schritte der Refaktorisierung durch. Die Schritte einer Transformation beinhalten auch Tests, die in jeder Transformation mindestens einmal vorkommen.

Im Katalog wird unter anderem beschrieben, worauf bei den einzelnen Schritten zu achten ist. Es kann z.B. vorkommen, dass eine Refaktorisierung sich aufgrund von bestimmten Programmstrukturen nicht anwenden lässt. Für solche Fälle stehen im Katalog Verweise auf andere Refaktorisierungen, die vorher ausgeführt werden können, um die geplante Refaktorisierung doch zu ermöglichen.

Auf diese Weise kommt es beim Anwenden einer Refaktorisierung oft zu einer Kette von hintereinander ausgeführten Refaktorisierungen, die aber alle zusammen in wenigen Minuten ausgeführt werden können. Änderungen des Programmcodes werden danach einfacher und das Programm verständlicher.

2.2 Codegerüche

Bestimmte Programmstrukturen und Programmmerkmale lassen Stellen im Code erkennen, an denen Refaktorisierungen möglich und empfehlenswert sind. Diese werden von Kent Beck und Martin Fowler metaphorisch als übel riechender Code bezeichnet.

In [FBB+00] geben Kent Beck und Martin Fowler in einem Katalog Indizien für Schwierigkeiten an, die durch Refaktorisieren gelöst werden können.

Die Codegerüche haben einen meist griffigen Namen, der direkt oder metaphorisch die Merkmale des Geruchs beschreibt. Bei jedem Geruch wird das Motiv für das Refaktorisieren erläutert, es werden geeignete Refaktorisierungen genannt und deren Anwendung kurz beschrieben. Um das Nachschlagen zu vereinfachen, enthält das Buch eine Tabelle, die jedem Codegeruch geeignete Refaktorisierungen zuordnet. So wird die Suche nach einer passenden Transformation beschleunigt.

Im Folgenden werden drei der 22 beschriebenen Codegerüche kurz vorgestellt.

Duplizierter Code

Redundanz im Programmcode erschwert die Weiterentwicklung. Man muss bei Änderungen alle Vorkommen des redundanten Codes finden und alle diese Vorkommen auf gleiche Art und Weise ändern. Die Suche kann unter Umständen lange dauern, redundante Programmteile können leicht übersehen werden, die Fehleranfälligkeit steigt. Ein Programm wird besser, wenn die Redundanz verschwindet.

Im einfachsten Fall wird in mehreren Methoden einer Klasse das gleiche Codefragment verwendet. Dann kann man die Refaktorisierung *Methode extrahieren* auf diese Fragmente anwenden und diese schließlich durch einen Aufruf der neuen Methode ersetzen. Das Kapitel 2.3 enthält ein Beispiel für diesen Fall.

Lange Methode

Die Länge von Methoden beeinflusst deren Verständnis. Je länger eine Methode ist, desto unübersichtlicher wird sie und desto schwieriger ist sie zu verstehen. Deswegen sollte man lange Methoden in kleinere Methoden mit selbstbeschreibenden Namen aufspalten. Das erspart viele Kommentare, macht den Code verständlicher und erhöht die Wahrscheinlichkeit, dass die Methoden wiederverwendet werden. Bei der Namensgebung sollte man darauf achten, dass der Name möglichst kurz und klar beschreibt, was die Methode tut. Eine Folge von Methodenaufrufen lässt sich dann wie ein Kommentar lesen.

Auch bei diesem Codegeruch kann man in den meisten Fällen *Methode extrahieren* anwenden, um zusammenhängende Codeblöcke zu eigenständigen Methoden zu machen.

Kommentare

Kommentare sind ebenfalls häufig ein Indiz dafür, das refaktorisiert werden sollte, was aber nicht heißt, dass man Kommentare vermeiden sollte. Kent Beck und Martin Fowler bezeichnen Kommentare sogar als süßen Duft, der aber oft als Deodorant für übel riechenden Code verwendet wird. Oft werden Kommentare dazu benutzt, schlechten Code zu erklären. Refaktorisiert man diesen Code, dann werden die Kommentare meist überflüssig.

Werden Kommentare benutzt, um zu erklären, was ein Codeblock tut, so sollte man *Methode extrahieren* benutzen und den Methodennamen so wählen, dass der Name dieses erklärt.

Im allgemeinen sollte man versuchen, so lange zu refaktorisieren, bis der Kommentar überflüssig wird. Einige Kommentare sind aber sehr sinnvoll: Gute Kommentare gehen über die Beschreibung dessen, was ein Codeblock tut, hinaus. Zum Beispiel kann ein Kommentar Entscheidungen des Programmierers begründen. Informationen über Entwurfsentscheidungen können bei zukünftigen Weiterentwicklungen sehr hilfreich sein und spätere Programmierer davon abhalten, wichtige geplante Programmstrukturen aufgrund von mangelnder Kenntnis des Designs zu ändern.

2.3 Einführendes Beispiel

In diesem Abschnitt der Ausarbeitung wird an einem Beispiel demonstriert, wie Refaktorisieren angewandt wird. Es werden die drei in Kapitel 2.1 erwähnten Hauptschritte der Anwendung einer Refaktorisierung verdeutlicht.

Identifikation der Anwendungsstelle

Bevor ein Programmierer mit dem Refaktorisieren anfangen kann, muss er eine Stelle finden, die er refaktorisieren möchte. Die Codegerüche helfen ihm dabei, indem sie Hinweise auf schlechten Code geben. Einige Codegerüche wurden bereits in dem vorhergehenden Abschnitt vorgestellt.

Die folgenden zwei Methoden produzieren Codegerüche und sind ein Beispiel für schlechten Code. Dieses Beispiel ist zur Vereinfachung klein gehalten.

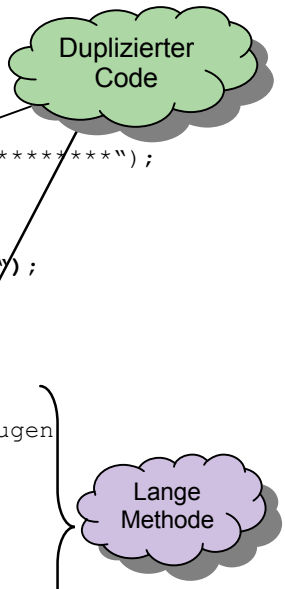
```
class Kunde
{
...
private int nr, offenerBetrag;
private String vorname, nachname;

public void gibKundenDatenMitBannerAus()
{
      //Banner
      System.out.println("***********************************************");

      //Die Daten
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}

private void aktualisiereOffenenBetrag()
{
      //Berechne die Summe der Beträge der noch zu zahlenden Rechnugen
      offenerBetrag = 0;
      for (int i; i<offeneRechnungen.length; i++)
        offenerBetrag += offeneRechnungen[i].getBetrag();

      //Gibt die Daten aus
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}
...
}
```



Die fettgedruckten Codefragmente sind bis auf den Kommentar identisch und arbeiten sogar mit den gleichen Variablen. Dieser Code enthält also Redundanz, die eliminiert werden sollte. Damit ist der Geruch *Duplizierter Code* aufgespürt. Außerdem ist die zweite Methode relativ lang und aufgrund der Tatsache, dass diese Methode zwei unterschiedliche Aufgaben erledigt, sollte diese Methode zerlegt werden. Der Geruch *Lange Methode* ist also auch vorhanden. Beide Gerüche zeigen Stellen im Code auf, die refaktorisiert werden sollten.

In diesem Beispiel soll die Redundanz entfernt werden. Hier sind also zwei Anwendungsstellen, nämlich die im Code bereits fettgedruckten duplizierten Codefragmente.

Wahl der Transformation

Die passende Transformation, um den Codegeruch – in diesem Fall Duplizierter Code – zu beseitigen, kann man mit Hilfe der in [FBB+00] vorhandenen Codegeruch-Tabelle wählen. Der folgende Tabellenausschnitt aus dem Buch gibt mehrere Möglichkeiten vor.

Geruch	Refaktorisierungen
Duplizierter Code, S. 68	Klasse extrahieren (148) Methode extrahieren (106) Methode nach oben verschieben (331) Template-Methode bilden (355)

Für dieses Beispiel ist die Anwendung von *Methode extrahieren* am besten geeignet, um die Redundanz zu entfernen. Das wird erreicht, indem eine neue Methode erstellt wird, die die

Funktionalität der markierten Codefragmente bietet. Anschließend werden die Codefragmente in den beiden Methoden durch einen Aufruf der neuen Methode ersetzt.

In schwierigeren Fällen, kann die Kurzbeschreibung und Motivation der Transformationen im Katalog von Refaktorisierungen helfen, eine geeignete Transformation auszuwählen. Die Zahlen neben den Transformationsnamen in der Codegeruch-Tabelle sind die Seitenzahlen der Transformationsbeschreibungen. Auf Seite 106 und den folgenden in [FBB+00] sind unter anderem die einzelnen Schritte der Transformation *Methode extrahieren* genau beschrieben.

Anwendung der Transformation

Nach der Identifikation der Anwendungsstelle und der Wahl der passenden Transformation – hier *Methode extrahieren* – wird die Transformation systematisch, Schritt für Schritt auf den Code angewendet.

In dem Beispiel muss *Methode extrahieren* wegen dem duplizierten Code auf zwei Stellen angewendet werden, nämlich auf die fettgedruckten Codefragmente in den beiden Methoden. Als Erstes wird die Transformation auf die erste Methode angewandt.

Zunächst erstellt man eine neue Methode, die den zu extrahierenden Code ersetzen soll und gibt ihr einen Namen, aus dem die Funktionalität der Methode klar wird. Dann fügt man den zu extrahierenden Code in den Rumpf der neuen Methode ein (siehe folgende Seite).
Es ist wichtig, bei der Namensgebung einen verständlichen Namen zu wählen. Das erspart unnötige Kommentare, die nur erklären würden, was die Methode tut.

Die Transformationsschritte:
1. Neue Methode erstellen
2. Zu extrahierenden Code in die neue Methode kopieren
3. Nach ungültigen Referenzen suchen
4. Compilieren

```
class Kunde
{
...
private int nr, offenerBetrag;
private String vorname, nachname;

public void gibKundenDatenMitBannerAus()
{
      //Banner
      System.out.println("*********************************************");

      //Die Daten
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}

private void aktualisiereOffenenBetrag()
{
      //Berechne die Summe der Beträge der noch zu zahlenden Rechnugen
      offenerBetrag = 0;
      for (int i; i<offeneRechnungen.length; i++)
        offenerBetrag += offeneRechnungen[i].getBetrag();

      //Gibt die Daten aus
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}

public void gibKundenDatenAus()
{
      //Die Daten
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}
...
}
```

Im nächsten Schritt sucht man nach ungültigen Referenzen im extrahierten Code. Das ist die Verwendung von Variablen in der neuen Methode, die dort nicht sichtbar sind. In diesem einfachen Beispiel sind alle im extrahierten Code verwendeten Variablen in der ganzen Klasse sichtbar, damit existieren keine ungültigen Referenzen.
Falls der extrahierte Code solche Referenzen enthielte, so wären andere zusätzliche Schritte nötig oder die Transformation würde die vorherige Anwendung anderer Transformationen benötigen.

Anschließend wird der gesamte Code compiliert, um eventuell versehentlich entstandene Fehler zu entdecken, z.B. syntaktische Fehler. Dynamisch semantische Fehler werden dabei nicht entdeckt.

Nun wird der extrahierte Code in der ursprünglichen Methode durch einen Aufruf der neuen Methode ersetzt.

```
class Kunde
{
...
private int nr, offenerBetrag;
private String vorname, nachname;

public void gibKundenDatenMitBannerAus()
{
      //Banner
      System.out.println("*********************************************");

      gibKundenDatenAus();
}

private void aktualisiereOffenenBetrag()
{
      //Berechne die Summe der Beträge der noch zu zahlenden Rechnugen
      offenerBetrag = 0;
      for (int i; i<offeneRechnungen.length; i++)
        offenerBetrag += offeneRechnungen[i].getBetrag();

      //Gibt die Daten aus
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}

public void gibKundenDatenAus()
{
      //Die Daten
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}
...
}
```

Die Transformationsschritte:
5. Extrahierten Code durch Aufruf der neuen Methode ersetzen
6. Compilieren und Testen

Abschließend wird das gesamte Programm wieder compiliert und getestet. Das Testen ist dabei sehr wichtig und ist ein Teil des letzten Schrittes. So soll sichergestellt werden, dass das Programm nach der Anwendung der Transformation immer noch das selbe „beobachtbare“ Verhalten hat, wie zuvor.

Damit ist die Anwendung von *Methode extrahieren* auf die erste Methode abgeschlossen. Da dieses Beispiel aber duplizierten Code enthält, muss diese Transformation auch auf die zweite Methode angewendet werden.

Die Anwendung der ersten vier Transformationsschritte auf die zweite Methode würde die gleiche Methode erzeugen, wie die bereits erstellte Methode `gibKundenDatenAus()`. Deswegen ist es nur noch nötig, den bereits extrahierten Code durch einen Aufruf der neuen Methode zu ersetzen und anschließend zu testen, also die Transformationsschritte fünf und sechs anzuwenden.

```
class Kunde
{
...
private int nr, offenerBetrag;
private String vorname, nachname;

public void gibKundenDatenMitBannerAus()
{
      //Banner
      System.out.println("***********************************************");
      gibKundenDatenAus();
}

private void aktualisiereOffenenBetrag()
{
      //Berechne die Summe der Beträge der noch zu zahlenden Rechnugen
      offenerBetrag = 0;
      for (int i; i<offeneRechnungen.length; i++)
        offenerBetrag += offeneRechnungen[i].getBetrag();

      gibKundenDatenAus();
}

public void gibKundenDatenAus()
{
      //Die Daten
      System.out.print("Knd.-Nr.: " + nr);
      System.out.println(" Name: " + vorname + "," + nachname + ",");
      System.out.println("noch offener Betrag: " + offenerBetrag);
}
...
}
```

Somit ist die Anwendung der Transformation *Methode extrahieren* auf die beiden Methoden abgeschlossen. Dadurch wurde die Redundanz in den Methoden beseitigt, die Methoden wurden verkürzt und die Lesbarkeit wurde erhöht. Beide vorher genannten Codegerüche wurden beseitigt.

2.4 Testen

Wie in den vorhergehenden Kapiteln bereits erwähnt, spielt das Testen beim Refaktorisieren eine entscheidende Rolle. Das Refaktorisieren benötigt Testen, um die Fehlerzahl zu minimieren und sicherzustellen, dass ein Programm nach seiner Transformation das gleiche Verhalten hat wie zuvor. Die Tests müssen automatisiert sein, damit sie das Refaktorisieren nicht bremsen.

Um die Änderungen zwischen den durchgeführten Tests klein und übersichtlich zu halten, müssen die Tests entsprechend häufig ausgeführt werden. Deswegen hat jede Refaktorisierung einen Refaktorisierungsschritt, der das Testen beinhaltet. Bei größeren Refaktorisierungen werden Tests sogar in mehreren Schritten ausgeführt. So lässt sich die letzte Änderung leicht zurücknehmen, falls ein Test Fehler aufzeigt. Auch die Fehlersuche ist einfach, denn es muss nur wenig Code kontrolliert werden.

Arten von Tests

In seinem Buch unterscheidet Martin Fowler zwei Arten von Tests, Komponententests und funktionale Tests.

Komponententests sind Tests, die von den Programmierern geschrieben werden, um ihren eigenen Code zu testen. Diese Tests sind hochgradig lokalisiert, jede Testklasse arbeitet nur in

einem Paket. Es werden die Schnittstellen zu anderen Paketen getestet, aber darüber hinaus wird unterstellt, dass alles andere funktioniert.

Funktionale Tests werden geschrieben, um die Funktionalität der Software als Ganzes sicherzustellen. Sie werden von einem unabhängigen Team entwickelt, das mit Hilfe spezieller Werkzeuge die Software testet. Dabei wird das System möglichst als Blackbox angesehen. Diese Tests untersuchen z.B. wie sich Daten bei bestimmten Eingaben ändern oder testen die Funktionalität einer GUI.

Für das Refaktorisieren sind hauptsächlich die Komponententests von Bedeutung.

Automatisierung

Damit das häufige Testen den Refaktorisierungsvorgang nicht aufhält, müssen die Tests schnell und automatisch ausgeführt werden können.
Normalerweise wird bei einem Test das geänderte Programm ausgeführt und seine Ausgabe mit der erwarteten Ausgabe verglichen. Das kann in vielen Fällen sehr mühselig werden, z.B. wenn man viele Ausgabewerte vergleichen oder diese erst selbst berechnen muss. Auch eine mehrfache Ausführung des Programms, um möglichst viele Fälle abzudecken, benötigt viel Zeit.
Automatisierte Tests beschleunigen das Testen. Dazu wird eine solide Menge von Testfällen implementiert, wobei der Code die erwarteten Ergebnisse enthält und diese mit den tatsächlichen Ergebnissen des Programms eigenständig vergleicht. So können die Tests häufig und ohne Zeitverlust oder großen Aufwand durchgeführt werden, was das Programmiertempo beschleunigt und die Fehler minimiert.

Die Tests sollten eine kurze Ausgabe erzeugen, die entweder bestätigt, das keine Fehler aufgetreten sind oder eine möglichst klare Fehlermeldung liefert. Bei der Ausgabe von Fehlermeldungen sollte zwischen Ausnahmen und falschen Ergebnissen unterschieden werden. Das vereinfacht die Fehlersuche zusätzlich. Eine einfache Fortschrittsanzeige ist bei der Ausführung von mehreren Tests hintereinander ebenfalls hilfreich, z.B. die Punkte in den Ausgaben unten, die jeden durchgeführten Test symbolisieren.

```
...
Time: 0.970

OK (3 tests)
```

```
.F
Time: 0.110

Failures!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead expected: "m" but was: "d"
```

Fehlersuche benötigt normalerweise einen großen Teil der Entwicklungszeit. Das häufige Ausführen automatisierter Tests beim Refaktorisieren vereinfacht und beschleunigt die Fehlersuche so, dass die Zeit für das Programmieren dieser Tests nicht mehr ins Gewicht fällt und die Entwicklungszeit insgesamt verkürzt werden kann. Frameworks wie das von Erich Gamma und Kent Beck entwickelte JUnit-Framework erleichtern die Programmierung der automatisierten Tests. Fehlermeldungen wie oben sind bereits implementiert, die Ausführung mehrerer Tests aus verschiedenen Klassen in nur einem Schritt ist auch möglich. So kann man z.B. täglich einen Großtest durchführen, der alle Testfälle eines größeren Programms beinhaltet.

Testfälle

Das Implementieren von automatisierten Tests kann viel Zeit in Anspruch nehmen, wenn man versucht, den gesamten Code und alle möglichen Fälle zu testen. Gewöhnlicherweise ist die Menge aller Fälle, die getestet werden müssten, sehr groß oder sogar unendlich. So ist es meist nicht möglich alle Fälle abzudecken und sollte risikoorientiert testen. Nur die relevanten Testfälle, bei denen vermutet wird, dass etwas jetzt oder in Zukunft schief gehen könnte, sind zu implementieren. Besonders gut geeignet sind Randbedingungen. Es gibt diverse Verfahren für die Wahl geeigneter Testfälle, die in anderer Literatur genauer behandelt werden.

3. KATALOG VON TRANSFORMATIONEN

3.1 Gliederung des Katalogs

Das Buch [FBB+00] enthält einen Katalog mit insgesamt 72 Refaktorisierungen. Um die Suche nach der passenden Transformation zu erleichtern, wurden die Refaktorisierungen in sieben zusammenhängende Gruppen zusammengefasst. Für jede dieser Gruppen enthält der Katalog ein Kapitel. Die Namen der Kapitel sind so gewählt, dass diese beschreiben, um welche Art Refaktorisierungen es sich darin handelt. Der Katalog soll als Nachschlagewerk dienen und effizientes Refaktorisieren unterstützen, ist aber nach eigenen Angaben des Autors nicht vollständig und nicht ausgereift. [FBB+00, S. 99, 104]

Die folgende Tabelle gibt eine Übersicht über die Kapitel des Katalogs, sowie die Art und Anzahl der behandelten Refaktorisierungen. Die Kapitelnummern entsprechen denen in [FBB+00].

Kapitel	Transformationsarten	Beispielrefaktorisierungen	Anzahl
6. Methoden zusammenstellen	Zerlegen und zusammenstellen von Methoden und diverse Hilfsrefaktorisierungen, Methoden verständlicher machen	*Methode extrahieren, Methode integrieren, Methode durch Methodenobjekt ersetzen, Temporäre Variable zerlegen*	9
7. Eigenschaften zwischen Objekten verschieben	Verteilung der Verantwortlichkeiten ändern, Verantwortung hinzufügen oder entfernen, Eigenschaften oder Berechnungen verschieben	*Methode verschieben, Klasse extrahieren, Klasse integrieren, Delegation verbergen*	8
8. Daten organisieren	Kapselung von Daten, Verwendung eigener Datentypen, Umgang mit sogenannten „magic numbers“	*Feld kapseln, Unterklasse durch Feld ersetzen, Wert durch Objekt ersetzen, Wert durch Referenz ersetzen*	16
9. Bedingte Ausdrücke vereinfachen	Zerlegen von Bedingungen, Fallunterscheidungen mit case bzw. switch durch Polymorphismus ersetzen	*Bedingung zerlegen, Bedingten Ausdruck durch Polymorphismus ersetzen, redundante Bedingungsteile konsolidieren*	8
10. Methodenaufrufe vereinfachen	Vereinfachung von Schnittstellen, Hinzufügen/Entfernen von Parametern, Sichtbarkeit einschränken, Ausnahmebehandlung	*Methode umbenennen, Parameter durch Methode ersetzen, Methode verbergen, Downcast kapseln*	15
11. Umgang mit Generalisierung	Verschieben von Methoden in der Vererbungshierarchie, Bildung von Template-Methoden, Hierarchie abflachen oder ergänzen	*Feld nach oben verschieben, Methode nach oben verschieben, Hierarchie abflachen, Vererbung durch Delegation ersetzen*	12
12. Große Refaktorisierungen	nicht triviale Transformationen, die viel Zeit benötigen, beziehen sich auf gesamtes Programm, vereinfachen z.B. die Vererbungsstruktur	*Vererbungsstrukturen entzerren, Prozedurale Entwürfe in Objekte überführen*	4
		Insgesamt:	**72**

Alle Refaktorisierungen, bis auf die vier aus dem letzten Kapitel, sind in kleine einfache Transformationsschritte unterteilt, können innerhalb kurzer Zeit durchgeführt werden und beziehen sich auf einen relativ kleinen Teil des Programmcodes. Zu vielen dieser Refaktorisierungen gibt es auch Refaktorisierungen, die die Transformation in umgekehrter Richtung durchführen. Diese werden zum Beispiel benötigt, wenn der Code erst in eine Form, die den Code nicht verbessert, transformiert werden muss, um ein bestimmtes Ziel zu erreichen und erst anschließend zurücktransformiert werden kann. So werden manche Ziele nur über Umwege erreicht.

Das letzte Kapitel des Katalogs, große Refaktorisierungen, befasst sich mit vier komplexen Transformationen, die sich auf das gesamte Programm beziehen und entsprechend mehr Zeitaufwand für ihre Durchführung benötigen. Martin Fowler schreibt von Projekten, bei denen eine solche Transformation Monate oder gar Jahre benötigt hat. Diese vier Transformationen sind auch weniger genau formuliert, weil das Abstraktionsniveau hier viel höher liegt, als bei den anderen Transformationen. Diese Transformationen können nur schwer verallgemeinert werden.

Auch bei der Namensgebung für die einzelnen Refaktorisierungen wurde Wert darauf gelegt, dass diese eindeutig, leicht zu merken und selbsterklärend sind. Auf diese Weise ist es einfach, die passende Refaktorisierung aus einer in [FBB+00] ebenfalls enthaltenen Liste zu wählen.

Die Beschreibung einer Refaktorisierung im Katalog besteht aus dem Namen, einer Kurzbeschreibung, der Motivation, der Vorgehensweise und einigen Beispielen zur Verdeutlichung.

> Die **Kurzbeschreibung** soll helfen, schnell zu verstehen, was die Transformation tut und in welcher Situation sie angewendet wird. Martin Fowler verwendet dazu Codefragmente, UML-Diagramme oder beides, um dieses zu verdeutlichen.
>
> In der **Motivation** wird genauer erklärt, in welchen Fällen eine Refaktorisierung angewendet werden soll bzw. wann darauf verzichtet werden sollte.
>
> Die **Vorgehensweise** enthält eine genaue Beschreibung der Transformationsschritte, allerdings ohne zu erklären, warum diese so und in dieser Reihenfolge angewendet werden sollen.
>
> Die **Beispiele** werden schließlich dazu verwendet, die Transformation genauer zu erklären und Sonderfälle und eventuelle Probleme mit der Transformation zu betrachten.

3.2 Refaktorisierungen aus dem Kapitel „Methoden zusammenstellen“

Das Kapitel „Methoden zusammenstellen“ im Katalog von Transformationen in [FBB+00] beschreibt Refaktorisierungen, die Methoden vereinfachen und sie verständlicher machen. Hier werden sie nur kurz vorgestellt, einige davon werden (bzw. wurden bereits in dem Kapitel 2.3) an einem Beispiel genauer erläutert.

Lange Methoden sind meist unübersichtlich und nur schwer zu verstehen. Diese enthalten oft nicht leicht einsehbare Informationen über das Programm. Deswegen dienen viele Refaktorisierungen dazu, die Methoden zu reorganisieren und in kleinere Methoden zu unterteilen. Kurze Methoden sind verständlicher und wenn die Namen geschickt gewählt sind, lassen sich ihre Aufrufe wie Kommentare lesen. Zusätzlich steigt durch kürzere Methoden die Wahrscheinlichkeit, dass diese wiederverwendet werden können.

Die wichtigste Transformation in dem Kapitel „Methoden zusammenstellen“ ist deswegen *Methode extrahieren*. Diese Transformation ermöglicht es, zusammenhängende Codefragmente aus einer Methode in eine neue Methode zu extrahieren und so die Methoden zu verkürzen und die Lesbarkeit zu erhöhen. Diese Transformation wurde an dem einführenden Beispiel in Kapitel 2.3 bereits vorgestellt.

Das Kapitel „Methoden zusammenstellen“ enthält die folgenden 9 Transformationen. Ihre Funktion wird hier nur kurz erläutert. Die Transformationen *Methode extrahieren* und *Methode durch Methodenobjekt ersetzen* werden anschließend genauer beschrieben.

Refaktorisierung	**Kurzbeschreibung**
Methode extrahieren	Extrahiert ein zusammenhängendes Codefragment in eine neue Methode.
Methode integrieren	Rumpf einer Methode in den Rumpf der aufrufenden Methode verschieben und die ursprüngliche Methode löschen.
Temporäre Variable integrieren	Referenzen auf eine temporäre Variable durch den zugewiesenen Ausdruck ersetzen.
Temporäre Variable durch Abfrage ersetzen	Berechnung der temporären Variable in eine neue Methode extrahieren und Referenzen auf diese Variable durch einen Aufruf der neuen Methode ersetzen.
Temporäre Variable zerlegen	Eine temporäre Variable, die mit verschiedenen Bedeutungen verwendet wird, durch mehrere temporäre Variablen für jede verwendete Bedeutung ersetzen.
Erklärende Variable einführen	Komplizierten Ausdruck einer neuen temporären Variable mit erklärendem Namen zuweisen.
Zuweisungen zu Parametern entfernen	Eine Zuweisung zu einem Parameter durch eine Zuweisung zu einer neuen temporären Variable ersetzen.
Methode durch Methodenobjekt ersetzen	Die Funktionalität einer Methode an ein Objekt einer neuen Klasse delegieren und alle lokalen Variablen der ursprünglichen Methode als Attribute des Objekts verwenden.
Algorithmus ersetzen	Den Rumpf einer einen Algorithmus ausführenden Methode durch einen Rumpf ersetzen, der einen anderen Algorithmus verwendet.

Methode extrahieren

In diesem Abschnitt wird diese Transformation nur kurz auf die Art und Weise vorgestellt, wie es in dem Katalog getan wird. Die Beispiele werden hier aber weggelassen.

Kurzbeschreibung:

Wie bereits in Kapitel 2.3 am einführenden Beispiel vorgestellt, wird diese Refaktorisierung dazu verwendet, zusammenhängenden Code aus einer Methode in eine neue Methode zu extrahieren.

Skizze aus [FBB+00]:

```
void printOwing(double amount)
{
    printBanner();

    //print details
    System.out.println("name: " + name);
    System.out.println("amount " + amount);
}
```

```
void printOwing(double amount)
{
    printBanner();
    printDetails(amount);
}

void printDetails(double amount)
{
    System.out.println("name: " + name);
    System.out.println("amount " + amount);
}
```

Motivation:

Lange Methoden werden verkürzt und unverständlicher Code wird lesbarer. Kurze Methoden können besser wiederverwendet werden. Durch die Wahl eines verständlichen Namens für die neue Methode können Kommentare, die das Codefragment beschreiben, vermieden werden. Eine Reihe vom Methodenaufrufen kann dann wie ein Kommentar gelesen werden.

Vorgehen:

1. Neue Methode mit einem selbsterklärendem Namen erstellen.
2. Das ausgewählte Codefragment in die neue Methode kopieren.
3. Den extrahierten Code nach Referenzen auf Variablen durchsuchen, die nur in der ursprünglichen Methode gültig sind.
4. Falls temporäre Variablen nur im extrahierten Code verwendet werden, dann diese in der neuen Methode als temporäre Variablen deklarieren (und falls diese außerhalb des extrahierten Codes deklariert wurden, dann diese Deklaration entfernen).
5. Prüfen, ob eine Variable, die nur in der ursprünglichen Methode gültig ist, im extrahierten Code verändert wird. Ist das der Fall, dann untersuchen, ob die neue Methode den neuen Wert dieser Variable als Ergebnis zurückgeben kann, sodass dieser Wert der Variablen zugewiesen werden kann. Wenn das nicht möglich ist oder mehrere Variablen verändert werden, dann ist *Methode extrahieren* so nicht möglich.
6. Die nur in der ursprünglichen Methode gültigen Variablen, die vom extrahierten Code gelesen werden, der neuen Methode als Parameter übergeben.

7. Falls alle lokalen Variablen der ursprünglichen Methode, die in dem extrahierten Code verwendet werden, abgearbeitet sind, dann den Code compilieren.
8. In der ursprünglichen Methode den extrahierten Code durch einen Aufruf der neuen Methode ersetzen.
9. Code erneut compilieren und testen.

Die Anzahl der Transformationsschritte ist hier höher als in dem Beispiel in Kapitel 2.3. Das liegt daran, dass in dem einführenden Beispiel die Schritte 4, 5 und 6 nicht durchgeführt werden mussten, weil der zu extrahierende Code keine lokalen Variablen enthielt.

Methode durch Methodenobjekt ersetzen

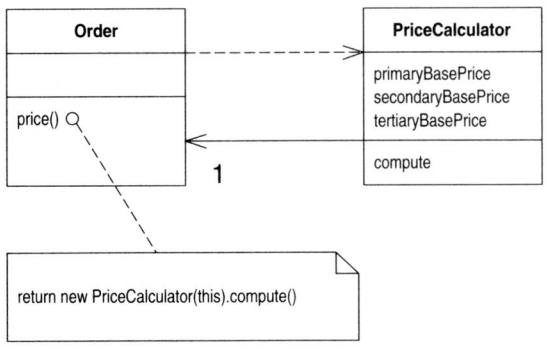
Diese Transformation dient dazu, die Transformation *Methode extrahieren* auch in schwierigen Fällen zu ermöglichen. Die Anwendung von *Methode extrahieren* kann durch eine bestimmte Art und Weise der Verwendung von lokalen Variablen erheblich erschwert oder gar unmöglich gemacht werden.

Um die Anwendung von *Methode extrahieren* zu ermöglichen, wird eine neue Klasse erstellt. Diese Klasse übernimmt alle lokalen Variablen der Methode, aus der extrahiert werden sollte, als Attribute von Objekten dieser Klasse. Die ursprüngliche Methode delegiert dann ihre Funktionalität an ein Objekt der neuen Klasse.

Skizze aus [FBB+00]:

```
class Order
{...
    double price()
    {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;

        //long computation
        ...
    }
}
```



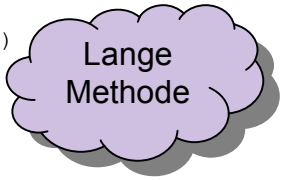
Anwendungsbeispiel und Motivation:

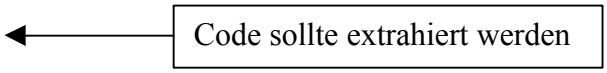
Die Methode im nachfolgenden Beispiel ist ziemlich lang und enthält zusammenhängenden Code, der extrahiert werden sollte.

```
class Kunde
{
    ...
    public int barPreisFuer(Artikel einArtikel, int menge)
    {
        int preis = menge * einArtikel.getVerkaufsPreis;
        float rabatt = getMinRabatt();
        int preisnachlass = 0;

        //Berechne gesamten Preisnachlass
        if (menge > 10)
            rabatt += 5;
        if (menge > 100)
            rabatt += 10;
        preisnachlass = (int) ( preis * rabatt / 100 );

        System.out.println("Kunde " + this + " bekommt " + rabatt + "% Rabatt.");
        return (preis - preisnachlass);
    }
}
```





Die Transformation *Methode extrahieren* wäre üblicherweise geeignet, den Codegeruch an dieser Stelle zu entfernen. In diesem Fall ist das aber sehr problematisch.
Das Hauptproblem dabei ist die Verwendung von mehreren temporären Variablen, die in dem zu extrahierenden Codefragment verändert und danach nochmals verwendet werden (siehe unten). Bei nur einer solchen Variable wäre es möglich, den Code zu extrahieren, indem man der neuen Methode die lokalen Variablen als Parameter übergibt und den Wert der veränderten temporären Variable als Ergebnis zurückgibt. Hier werden aber mehrere Variablen verändert und danach mit ihrem neuen Wert benötigt. Die Anwendung von *Methode extrahieren* ist also nicht ohne Weiteres möglich. Im Folgenden sind die entsprechenden temporären Variablen durch Fettdruck markiert.

```
class Kunde
{
    ...
    public int barPreisFuer(Artikel einArtikel, int menge)
    {
        int preis = menge * einArtikel.getVerkaufsPreis;
        float rabatt = getMinRabatt();
        int preisnachlass = 0;

        //Berechne gesamten Preisnachlass
        if (menge > 10)
            rabatt += 5;
        if (menge > 100)
            rabatt += 10;
        preisnachlass = (int) ( preis * rabatt / 100 );

        System.out.println("Kunde " + this + " bekommt " + rabatt + "% Rabatt.");
        return (preis - preisnachlass);
    }
}
```

Um das Extrahieren des Codes doch noch zu ermöglichen und die Methode dadurch zu verkürzen, kann man die Transformation *Methode durch Methodenobjekt ersetzen* einsetzen.

Anwendung:

1. Zunächst erstellt man eine neue Klasse. Dabei ist auch hier ein sinnvoller verständlicher Name zu wählen. Der Name soll beschreiben, was die ursprüngliche Methode getan hat, denn ein Objekt der neuen Klasse wird die Funktionalität dieser Methode übernehmen.

2. Als Nächstes wird eine Referenz auf ein Objekt der ursprünglichen Klasse hinzugefügt. Diese Referenz ermöglicht es, Methoden und Variablen des ursprünglichen Objektes zu nutzen. Diese Referenz ist hier als final deklariert, weil ein Objekt der neuen Klasse für genau ein Objekt der ursprünglichen Klasse Berechnungen durchführen soll und nicht falsche Daten, die das Objekt in seinen Attributen speichern wird, für eine weitere Berechnung benutzen kann.

3. Dann fügt man alle Parameter und temporären Variablen der ursprünglichen Methode der neuen Klasse als Attribute hinzu. Dann sind alle diese Variablen in der gesamten neuen Klasse sichtbar und erleichtern die spätere Anwendung von *Methode extrahieren*.

```
class Kunde
{
    ...
    public int barPreisFuer(Artikel einArtikel, int menge)
    {
        int preis = menge * einArtikel.getVerkaufsPreis;

        float rabatt = getMinRabatt();
        int preisnachlass = 0;
        ...
    }
}

class BarPreisBerechner
{
    private final Kunde kunde;
    private Artikel einArtikel;
    private int menge, preis, preisnachlass;
    private float rabatt;

    BarPreisBerechner(Kunde iKunde, Artikel iArtikel, int iMenge)
    {
        kunde = iKunde; einArtikel = iArtikel; menge = iMenge;
    }
}
```

Die Transformationsschritte:
1. Neue Klasse erstellen
2. Referenz auf ursprüngliches Objekt hinzufügen
3. Parameter und temporäre Variablen als Attribute hinzufügen
4. Konstruktor hinzufügen

4. Im nächsten Schritt wird der neuen Klasse ein Konstruktor hinzugefügt. Dieser bekommt alle Parameter der ursprünglichen Methode und eine Referenz auf das ursprüngliche Objekt als Parameter übergeben.

5. & 6. Dann wird eine Methode namens `compute()` erstellt und der Rumpf der ursprünglichen Methode in die Methode `compute()` eingefügt. Nun muss man alle ungültigen Aufrufe von Methoden und Referenzen auf Variablen der ursprünglichen Klasse finden und den Zugriff mit Hilfe der Referenz auf das ursprüngliche Objekt ermöglichen. Die Methode `compute()` hat jetzt die gleiche Funktionalität, wie die ursprüngliche Methode.

7. Als Nächstes wird der gesamte Code compiliert, um eventuell aufgetretene Fehler jetzt schon zu finden. Es werden aber nicht alle Fehler gefunden, z.B. werden dynamisch semantische Fehler nicht gefunden.

```
class Kunde
{
    ...
    public int barPreisFuer(Artikel einArtikel, int menge)
    {
        int preis = menge * einArtikel.getVerkaufsPreis;
        float rabatt = getMinRabatt();
        int preisnachlass = 0;
        ...
    }
}

class BarPreisBerechner
{
    ...
    int compute()
    {
        int preis = menge * einArtikel.getVerkaufsPreis;
        float rabatt = kunde.getMinRabatt();
        int preisnachlass = 0;

        //Berechne gesamten Preisnachlass
        if (menge > 10)
            rabatt += 5;
        if (menge > 100)
            rabatt += 10;
        preisnachlass = (int) ( preis * rabatt / 100 );

        System.out.println("Kunde " + this + " bekommt " + rabatt + "% Rabatt.");
        return (preis - preisnachlass);
    }
}
```

Die Transformationsschritte:
5. Methode `compute()` erstellen
6. Rumpf der ursprünglichen Methode einfügen
7. Compilieren

8. Anschließend wird in der ursprünglichen Methode der Rumpf entfernt und stattdessen ein Objekt der neuen Klasse erstellt. Dieses Objekt soll die Berechnungen der Methode übernehmen und bekommt alle nötigen Parameter übergeben. Die Methode `compute()` wird auf diesem Objekt angewandt und das Ergebnis wird als Ergebnis der ursprünglichen Methode zurückgegeben.

9. Schließlich wird nochmals compiliert und getestet, um sicherzustellen, dass das „beobachtbare“ Verhalten nach der Transformation das gleiche ist, wie zuvor. Das Testen ist auch hier ein Teil des Schrittes und muss durchgeführt werden.

```
class Kunde
{
    ...
    public int barPreisFuer(Artikel einArtikel, int menge)
    {
        return new BarPreisBerechner(this, einArtikel, menge).compute();
    }
}

class BarPreisBerechner
{
    ...
}
```

Die Transformationsschritte:
8. Den Rumpf der ursprünglichen Methode entfernen, ein Objekt der neuen Klasse erzeugen und `compute()` darauf ausführen
9. Compilieren und testen

Ergebnis der Transformation

Nach der Transformation sind die lokalen Variablen, die vorher Probleme bei der Anwendung von *Methode extrahieren* gemacht haben, in der gesamten neuen Klasse sichtbar. In der Methode `compute()` sind sie nicht mehr lokal und können von allen Methoden der neuen

Klasse verändert werden. Das macht die Anwendung von *Methode extrahieren* besonders einfach.

Man erstellt eine neue Methode, kopiert den zu extrahierenden Code – nachfolgend fettgedruckt – in diese Methode und compiliert. Das Prüfen auf ungültige Referenzen entfällt. Dann ersetzt man den extrahierten Code in der ursprünglichen Methode durch einen Aufruf der neuen Methode, compiliert nochmals und testet das Programm.

***Methode extrahieren* jetzt einfach:**
1. Methode erstellen
2. Zu extrahierenden Code kopieren
3. Compilieren
4. Ursprünglichen Code durch Methodenaufruf ersetzen
5. Compilieren und testen

```
class BarPreisBerechner
{   ...
    int compute()
    {
        int preis = menge * einArtikel.getVerkaufsPreis;
        float rabatt = kunde.getMinRabatt();
        int preisnachlass = 0;

        berechneGesamtenPreisnachlass();

        System.out.println("Kunde " + this + " bekommt " + rabatt + "% Rabatt.");
        return (preis - preisnachlass);
    }

    void berechneGesamtenPreisnachlass()
    {
        //Berechne gesamten Preisnachlass
        if (menge > 10)
            rabatt += 5;
        if (menge > 100)
            rabatt += 10;
        preisnachlass = (int) ( preis * rabatt / 100 );
    }
}
```

extrahierter Code

Dieses Beispiel zeigt, wie lokale Variablen die Anwendung von *Methode extrahieren* erschweren können und wie die dadurch entstehenden Probleme durch die Transformation *Methode durch Methodenobjekt ersetzen* behoben werden können. Außerdem ist das ein Beispiel dafür, dass einige Transformationen in bestimmten Fällen eine vorherige Anwendung anderer Transformationen benötigen und deswegen mehrere Transformationen häufig direkt hintereinander durchgeführt werden.

Die Durchführung dieser beiden Transformationen kann, wenn der Programmierer geübt ist, in nur wenigen Minuten erfolgen. Die einzelnen Schritte sind einfach und die Fehleranfälligkeit dadurch gering. Das Testen und Compilieren bei jeder Transformation ist eine weitere Sicherheit, die die Fehler schnell erkennen lässt und die Fehlerzahl reduziert.

4. SCHLUSSÜBERLEGUNGEN

4.1 Refaktorisieren und Entwurf

> „Im Design kann ich sehr schnell denken, aber meine Gedanken sind voller kleiner Löcher.“ Alistair Cockburn [FBB+00, S. 57]

Refaktorisieren kann als eine Alternative zum Entwurf vor dem Implementieren gesehen werden. Man könnte direkt mit dem Programmieren beginnen, den Code funktionsfähig machen und durch das Refaktorisieren in eine gute Form bringen. Das ist aber meist nicht effizient.

Beim Extreme Programming (XP) wird auch erst nach dem Erstellen eines plausiblen Entwurfs programmiert und dann refaktorisiert.

Das Entwerfen einer ausgereiften Programmstruktur ist sehr aufwendig und sollte ursprünglich das noch aufwendigere spätere Redesign verringern. Laut Fowler ist es aber meist weniger aufwendig und damit effizienter, einen guten aber nicht unbedingt ausgereiften Entwurf vor der Implementierung zu erstellen und dann beim Programmieren zu refaktorisieren. Beim Programmieren lernt man mehr über das Problem, an dem man arbeitet und merkt, dass eine ausgereifte Lösung sich von der ursprünglich geplanten unterscheidet. Hat man einen teuren und aufwendigen Entwurf erstellt, so ist eine spätere Änderung der Programmstruktur ebenfalls aufwendig und teuer. Beim Refaktorisieren ist eine nachträgliche Änderung der Struktur nicht aufwendig und damit nicht teuer.

Versucht man von Anfang an eine sehr flexible Lösung zu finden, so ist der Entwurf und die spätere Wartung des fertigen Programms sehr teuer. Deswegen versucht man beim Refaktorisieren, vor der Implementierung nur einen einfachen Entwurf zu erstellen und eine einfache Lösung zu finden, die leicht in eine flexiblere Lösung refaktorisiert werden kann. Laut Fowler verlieren die Programme dann nicht an Flexibilität, der Entwurfsprozess und die Wartung werden aber einfacher.

4.2 Refaktorisieren und Performance

Beim Refaktorisieren wird der Programmcode oft so transformiert, dass nach der Transformation mehr Objekte instanziiert werden, Methoden öfter aufgerufen werden und dadurch einige Algorithmen oder Berechnungen öfter durchgeführt werden. So kann schnell der Eindruck entstehen, dass Refaktorisieren die Laufzeit und den Speicherbedarf der Programme erheblich erhöht.
Das stimmt nur bedingt. In den meisten Fällen wird die Laufzeit und der Speicherbedarf zwar erhöht, jedoch ist der Unterschied zu einem optimierten Programm gering. Dadurch, dass das Programm aber viel verständlicher ist, kann es zum Schluss der Entwicklung bezüglich Performance leichter optimiert werden.

Um Performance zu erreichen, kann man stets „aufmerksam“ programmieren und den Code möglichst effizient gestalten. Die meisten Programme nutzen aber aufgrund ihrer hohen Lokalität im größten Teil der Zeit nur einen kleinen Teil des Codes, so dass ca. 90% der Laufzeitoptimierungen keinen merkbaren Performance-Schub erbringen. Die Zeit für diese Optimierungen ist dann verschwendet.

Es ist einfacher, den Code zu schreiben, ohne auf Performance zu achten und erst am Ende der Entwicklung den Code bezüglich Laufzeit und Speicherbedarf zu optimieren. In diesem Fall konzentriert man sich aber nur auf die kritischen Bereiche und optimiert diese mit gleicher Vorsicht, wie beim Refaktorisieren. Man ändert nur kleine Teile, testet, ob das Programm dasselbe leistet und prüft die Performance. Falls es keine Performance-Verbesserung gibt, macht man den letzten Schritt wieder rückgängig.

Diese Art der Entwicklung braucht weniger Zeit und es kann mehr Zeit für Performance-Optimierung verwendet werden. Das fertige Programm ist viel verständlicher, die Komponenten sind besser gekapselt und die gesamte Programmstruktur ist feinkörnig. Bei der Performance-Optimierung kann man sich dann auf kleine Teile konzentrieren und es ist leichter abzuschätzen, welche Tuningmaßnahmen wirken werden. Verschiedene Werkzeuge unterstützen den Optimierungsvorgang und helfen, die relevanten Stellen im Code zu finden.

Fazit ist also, dass Refaktorisieren es nicht erschwert, Programme mit guter Performance zu entwickeln, da diese zum Schluss der Entwicklung schnell und einfach optimiert werden können.

4.3 Schwierigkeiten und Grenzen

Finden von Referenzierungen

Bei vielen Refaktorisierungen muss man alle Referenzen oder Verwendungen von einer Methode, einem Attribut oder einer Klasse finden. Aufgrund von dynamischer Methodenbindung, Vererbung, verschiedener Sichtbarkeiten ist das oft keine leichte Aufgabe. In kleinen übersichtlichen Programmen ist die Suche meist einfach, aber in einem großen Softwareprojekt ist das schwierig.

Bei der manuellen Suche kann schnell eine Referenz übersehen werden, deswegen wird die Suche meist dem Rechner überlassen. Die Textsuche ist schneller und übersieht keine Vorkommen des eingegebenen Suchbegriffs, aber auch das ist keine Garantie dafür, dass alle Referenzen gefunden werden. Bei der Verwendung der Suchen-und-Ersetzen-Funktion von Text-Editoren kann es aufgrund von dynamischer Methodenbindung zu falschen Ersetzungen kommen, wenn diese vom Programmierer vorher nicht überprüft werden. Es könnten zum Beispiel überschreibende Methoden oder Variablen gleichen Namens in verschiedenen Sichtbarkeitsbereichen fälschlicherweise verändert werden. Aufgrund einer auf nur eine Datei beschränkten Suche könnten Referenzen in anderen Klassen übersehen werden.

Der Compiler kann auch dazu verwendet werden, Referenzen, die geändert werden müssen, zu finden, aber auch damit werden nicht immer alle Referenzen gefunden. Die Suche nach einer mehrfach überschriebenen Methode mit Hilfe des Compilers ist sehr schwierig. Außerdem kann das Compilieren von großen Programmen viel Zeit benötigen.

Einige Entwicklungsumgebungen liefern Programmanalyse-Werkzeuge, die Abhilfe bei der Suche nach Referenzen schaffen. Diese sind der manuellen und der Textsuche, sowie der Compiler-Suche weit überlegen und können alle in Frage kommenden Referenzen finden. Aus der Menge der Referenzen kann der Programmierer dann diejenigen aussuchen, die tatsächlich geändert werden sollen. Die Suche mit diesen Werkzeugen ist auch effizienter, da die Programmstruktur in einer Datenbank im Hauptspeicher verwaltet wird, anstatt Textdateien zu durchsuchen.

Änderung von Schnittstellen

Refaktorisierungen ändern auch Schnittstellen, was zu Problemen führt, wenn diese Schnittstellen öffentlich oder gar veröffentlicht sind.

Öffentliche Schnittstellen sind überall sichtbar und können so von jedem benutzt werden. Wenn man alle Benutzer dieser Schnittstelle erreicht, kann man die Nutzung durch Änderungen von Code selbst anpassen, was aber meistens ziemlich aufwendig ist.

Veröffentlichte Schnittstellen sind ebenfalls überall sichtbar, aber man kann nicht mehr selbst alle Benutzer dieser Schnittstelle erreichen. Zum Beispiel ist das der Fall nach der Veröffentlichung eine Bibliothek wie der Java-Bibliothek. Man kann die Nutzung der darin verwendeten Schnittstellen nicht mehr anpassen.
In diesem Fall kann man die alte Schnittstelle behalten, eine neue entwickeln und diese die alte Schnittstelle nutzen lassen. Damit wird Redundanz vermieden und es treten keine Probleme bei Nutzern der alten Schnittstelle auf. In Java sollte man zusätzlich die alte Schnittstelle als „deprecated“ (veraltet) kennzeichnen.
Besser ist es – wenn möglich – keine unausgereiften Schnittstellen zu veröffentlichen.

Nebenläufigkeit

Die Refaktorisierungen im Katalog von Martin Fowler wurden für Programme mit nur einem Prozess entwickelt. Refaktorisierungen in nebenläufigen oder verteilten Programmen wären andere. Man müsste z.B. zusätzlich darauf achten, dass Methodenaufrufe und der Datentransfer minimiert werden.

4.4 Fazit

Die Refaktorisierungen von Martin Fowler sind Transformationen von objektorientiertem Programmcode zur Steigerung der Lesbarkeit. Sie sind aus der Programmiererfahrung entstanden, aber es wurde nicht bewiesen, dass sie in allen Fällen verhaltenserhaltend transformieren. Um dieses dennoch für die meisten Fälle sicherzustellen, werden umfangreiche Tests beim Refaktorisieren durchgeführt. Die einzelnen Refaktorisierungen benötigen nur wenig Zeit und können bei geeigneter Anwendung die Lesbarkeit erhöhen und so die Effektivität steigern.
Die Kataloge von Codegerüchen und Refaktorisierungen sind nicht vollständig und nicht ausgereift, tragen aber als Nachschlagewerk zur Effizienzsteigerung beim Refaktorisieren bei und sollen außerdem das Interesse an Refaktorisierungen und der Forschung in diesem Bereich wecken.

5. HERKUNFT UND ÄHNLICHKEITEN

5.1 Herkunft

Die Herkunft des Refaktorisierens lässt sich nicht eindeutig festlegen. In der Smalltalk-Gemeinde wurde schon seit den 80er Jahren an einem Entwicklungsprozess gearbeitet, der die Wartungskosten reduzieren sollte. Dabei entstand aus der Arbeit von Ward Cunningham und Kent Beck das Extreme Programming, welches das Refaktorisieren zur Produktivitätssteigerung benutzt.
Bei einer Zusammenarbeit an einem Projekt mit Kent Beck hat Martin Fowler die Vorteile von Refaktorisieren kennen gelernt und anschließend sein Buch [FBB+99] geschrieben, weil kein Buch zu diesem Thema existierte. Durch dieses Buch wurde Refaktorisieren bekannt.

5.2 Ähnlichkeiten

Auch in anderen Bereichen wurden Transformationen entwickelt, die Programme verhaltenserhaltend transformieren. Im Folgenden werden einige Transformationen angesprochen, um Ähnlichkeiten zum Refaktorisieren aufzuzeigen. Sie werden alle systematisch in kleinen Schritten und nach bestimmten Regeln angewandt.

Transformationen in der funktionalen Programmierung

Auch bei der funktionalen Programmierung werden Transformationsregeln verwendet. Zum Beispiel kann jede imperative Schleife in eine end-rekursive Funktion transformiert werden und eine end-rekursive Funktion kann in eine imperative Form transformiert werden.

Fold und Unfold sind zwei weitere Transformationen aus der funktionalen Programmierung. Sie sind aus dem Fold/Unfold-System [Erw99, S. 75 ff], das insgesamt sechs Transformationen enthält und sind zueinander invers. Während Unfold Funktionsanwendungen durch entsprechende Definitionen ersetzt, ersetzt Fold einen Ausdruck durch einen Funktionsaufruf. Das gesamte Fold/Unfold-System kann zum Beispiel die Funktionsauswertung effektiver machen.
Die Transformationen des Fold/Unfold-Systems sind streng spezifiziert, es ist aber keine Reihenfolge festgelegt, in der sie angewendet werden müssen.

Transformationen des Projekts CIP

In den 70er Jahren wurde im Rahmen des „Computer-Aided Intuition-Guided Programming"-Projektes (CIP) unter der Leitung von F. L. Bauer von der Technischen Universität München eine auf Regeln basierende Sprache zur Beschreibung von Transformationen entwickelt. Mit Hilfe dieser Sprache sollte eine abstrakte, algebraisch formulierte Spezifikation in ein ausführbares effizientes Programm transformiert werden.

Zu den in dieser Sprache formulierten Transformationen gehören unter anderem auch verhaltenserhaltende Transformationen ähnlich denen beim Refaktorisieren. Darunter sind Transformationsregeln zur algebraischen Optimierung, Kontrollstrukturmanipulation, fold und unfold von Funktionen und zum Ändern von Datenstrukturen. Sie wurden durch ein abstraktes Schema spezifiziert und benutzen sogenannte Schemavariablen, um Teile der Zielsprache zu ersetzen.

In den folgenden zwei Beispielen aus [PhR01, S. 6] sind die Großbuchstaben die Schemavariablen, welche Teile eines zu transformierenden Ausdrucks repräsentieren. Jeweils rechts steht die Bedingung, unter der die Transformation angewendet werden darf.

Beispiele für CIP-Transformationsregeln:

$$\frac{E + E}{2{*}E} \;\Big[\!\Big[\; \text{Ausdruck } E \text{ ist seiteneffektfrei und deterministisch}$$

Regel (a)

$$\frac{V := E;\ S}{S;\ V := E} \;\Big[\!\Big[\; \text{Variable } V \text{ wird in der Aussage } S \text{ nicht verwendet und Ausdruck } E \text{ ist seiteneffektfrei}$$

Regel (b)

Die Regel (a) ist eine simple algebraische Transformation. Mit dieser Regel lässt sich 3+3 durch 2*3 ersetzen. Es kann aber nicht auf (i++)+(i++) angewendet werden, weil das nicht seiteneffektfrei ist und auch nicht auf Math.random()+Math.random(), da das nicht deterministisch ist.

Die Regel (b) ist ein Beispiel für Kontrollstrukturmanipulation und behandelt die Umordnung von Aussagen.
An beiden Beispielen kann man auch erkennen, dass diese in beide Richtungen angewendet werden können, wie viele andere CIP-Transformationen auch.

Die Transformationsregeln beim CIP sind viel strenger formuliert und oft viel feinkörniger als die Refaktorisierungen von Martin Fowler, aber sie sind dennoch ähnlich.

Mathematische Transformationen

In der Mathematik werden zum Beispiel Polynomgleichungen mit Hilfe von Algebra-Regeln nach verschiedenen Rechnungen wieder faktorisiert, also in ihre Faktoren zerlegt. Diese faktorisierten Gleichungen lassen auf einfache Art und Weise die Nullstellen erkennen und sind in diesem Sinne verständlicher.

$$0 = x^2 + x - 2 \quad \Rightarrow \quad 0 = (x - 1)\,(x + 2)$$

Die Faktorisierung in der Mathematik ist auch eine Art Transformation. Es gibt in der Mathematik noch unzählige weitere Transformationen, aber Faktorisieren, oder wieder Faktorisieren heißt auf Englisch refactor, was möglicherweise der Grund für den Namen „Refactoring“ war.
Sieht man die Menge der Refaktorisierungen als Algebra an, so ist das Transformieren von Programmcode, die Anwendung dieser Algebra-Regeln. Natürlich müsste die Richtigkeit dieser Regeln erst noch bewiesen werden, was Fowler in [FBB+00] nicht getan hat.

6. LITERATUR

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts mit dem Vorwort von Erich Gamma. *Refactoring. Improving The Design Of Existing Code.* Addison-Wesley, 1999

[FBB+00] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts mit dem Vorwort von Erich Gamma. *Refactoring. Wie Sie das Design vorhandener Software verbessern.* Addison-Wesley, 2000 (Übersetzung von [FBB+99])

[PhR01] Jan Philipps, Bernhard Rumpe: *Roots of Refactoring.* 2001

[Erw99].Martin Erwig. *Grundlagen funktionaler Programmierung.* R. Oldenburg, 1999