**Universität Paderborn**
**Fakultät V**
**Institut für Informatik**

# Seminarbericht
# Refactoring - Transformationen

Martin Freund
bbl@upb.de

Seminar
Refactoring in eXtreme Programming
Betreuer:
Prof. Dr. Uwe Kastens
Dipl. Inform. Jochen Kreimer

Paderborn, 3.3. 2003

# Inhaltsverzeichnis

# Einleitung

Wiederverwendbarkeit von Software ist ein wesentlicher Gesichtspunkt bei der Benutzung objektorientierter Programmiersprachen. Klassen und Bibliotheken sollen nach ihrer Erstellung in neuen Kontexten zum Einsatz kommen.

Da neue Kontexte neue Anforderungen mit sich bringen, ist einfache Wiederverwendung meistens ausgeschlossen, sondern eine Anpassung oder Erweiterung der Software unumgänglich. Was aber, wenn die Anpassung der Software aufgrund schlechten Designs und schlechter Strukturierung fehleranfällig und aufwändig wird? In diesem Fall kann Umstrukturierung helfen. Die Umstrukturierung mit dem Ziel der Designverbesserung unter gleichzeitiger Erhaltung der Funktionalität bezeichnet man als Refactoring.

Refactoring ist ein neues Lieblingswort der Softwareindustrie und kaum eine Programmierhilfe wird im Moment angeboten, die nicht die Unterstützung von Refactoring anpreist.

Dabei ist die erste Arbeit zu diesem Thema von Opdyke [Opdyke92] bereits zehn Jahre alt. Die Verbreitung des Begriffs und die Anwendung in der Praxis geht allerdings einher mit der Verbreitung entsprechender Literatur, wobei Fowlers Buch [Fowler2000] als eines der Standardwerke zum Thema Refactoring für die praktische Anwendung gelten darf.

In enger Anlehnung an Fowlers Buch ist diese Ausarbeitung zum Thema Refactoring-Transformationen entstanden. Sie richtet sich an Leser mit wenig Erfahrung mit Refactoring, z.B. aus dem zugehörigen Seminar [Seminar03] und kann als Einstieg anhand der von Fowler beschriebenen, in der Praxis erprobten Methoden dienen.

Der Text führt die beiden Begriffe Refactoring und Refactoring-Transformationen ein. Anhand eines Überblicks über einen Teil der von Fowler beschriebenen Refactoring-Transformationen wird die Problemabdeckung dargestellt. Die detaillierte Beschreibung der Transformationen *Methode verschieben* und *Bedingte Anweisung durch Polymorphie ersetzen* gibt Aufschluß über Handhabung und Zeitaufwand für das Ausführen von Refactoring-Transformationen.

Es wird auf Unterschiede bei der Beschreibung von Refactoring-Transformationen durch verschiedene Autoren und daraus entstehende Probleme hingewiesen.

Zum Abschluß werden Erfahrungen mit Refactoring beschrieben, wobei ein Projekt zur Erweiterung des Refactoring-Ansatzes und Erfahrungen aus der akademischen Lehre hervorgehoben werden.

# 1 Refactoring-Transformationen

In diesem Kapitel werden die Begriffe Refactoring und Refactoring-Transformation anhand der von Fowler vorgegebenen Definitionen eingeführt. Es werden die Kategorien zur Einteilung der Menge von Refactoring-Transformationen vorgestellt, die William Opdyke und Martin Fowler verwendet haben und deren Unterschiede beschrieben. Zuletzt werden Gründe vorgeschlagen, warum die einheitliche Kategorisierung von Refactoring-Transformationen sinnvoll ist.

## 1.1 Refactoring und Refactoring-Transformationen

Für den Begriff Refactoring gibt Fowler in seinem Buch zwei Definitionen an. Zum einen bezeichnet er als Refactoring eine "Änderung der internen Struktur eines Programms um es verständlicher und einfacher veränderbar zu gestalten, ohne sein beobachtbaren Verhaltens zu ändern."(nach [Fowler2000]). Zum anderen beschreibt er den Vorgang Refactoring "zur Umstrukturierung eines Programms durch Anwendung einer Reihe von Refactorings ohne das beobachtbare Programmverhalten zu ändern".

Umstrukturierung von Software bezeichnet man korrekterweise als Transformation, daher soll der erfolgreiche, abgeschlossene Umstrukturierungsschritt als Refactoring-Transformation bezeichnet werden und wird durch Fowlers erste Definition charakterisiert. Im weiteren Verlauf des Textes werden die Begriffe Refactoring-Transformation und Transformation gebraucht, gemeint ist immer eine Refactoring-Transformation.

Refactoring als Prozeß, der sich aus einzelnen, sinnvoll kombinierten Refactoring-Transformationen zusammensetzt, genügt wiederum der zweiten Definition. Für erfolgreiches Refactoring muß zunächst ein zu erreichendes Ziel definiert werden, welches sich meist unmittelbar aus dem Kontext einer aktuellen Programmieraufgabe ergibt. Zum Erreichen dieses Ziels müssen die dazu notwendigen Refactoring-Transformationen sinnvoll nacheinander ausgeführt werden. Diese Refactoring-Transformationen sind bei Fowler in Art eines Kochbuches beschrieben und nach Kategorien geordnet.

## 1.2 Kategorisierung von Refactoring-Transformationen

Refactoring-Transformationen kann man als einzelne Werkzeuge auffassen, deren Anwendung Verbesserungen an einem existierenden Programm bewirkt. Um die Entschei-

dung für eine bestimmte Transformation zu erleichtern teilen Autoren, welche Refactoring-Transformationen vorstellen, diese in Kategorien ein. Leider sind die jeweils benutzten Einteilungen untereinander nicht immer kompatibel. Zur Verdeutlichung werden die Unterschiede angeführt, welche zwischen der Einteilung nach [Opdyke92], der ersten Arbeit zum Thema Refactoring, und dem Buch von Fowler bestehen.

### 1.2.1 Einteilung nach Opdyke

Opdyke unterteilt seine Doktorarbeit[Opdyke92] nach vier Kategorien für Refactoring-Transformationen. Zunächst beschreibt er 26 atomare Basistransformationen (Low-level refactorings), welche nicht weiter aufteilbar sind. Dazu zählt Opdyke Refactoring-Transformationen für das Erzeugen, Löschen und Ändern von Programmentitäten (program entities), das Verschieben von Variablen und einige komponierte Refactoring-Transformationen. Die drei anderen Kategorien beinhalten Transformationen für das Einführen von Generalisierung (z.B. abstrakter Oberklassen), von Spezialisierung (z.B. Bilden von Unterklassen, Vereinfachen bedingter Anweisungen) und die Verwendung von Aggregation.

Zudem macht sich Opdyke im voraus Gedanken über Programmeigenschaften, welche durch Refactoring-Transformationen leicht verletzt werden können und Formalismen zur Beschreibung der Vorbedingungen, welche für die Durchführung einer Transformation erfüllt sein müssen. Diese Eigenschaften und Formalismen sind Gegenstand der Arbeit von Hermann Wessels zur Komposition von Refactoring-Transformationen im Rahmen des gleichen Seminars, weshalb ich auf die entsprechende Webseite verweise [Seminar03].

### 1.2.2 Einteilung nach Fowler

Fowler hingegen stellt Probleme und Refactoring-Transformationen zu deren Lösung gegenüber und verzichtet auf formale Beschreibung von Vor- und Nachbedingungen.

Er identifiziert Problemstellen anhand sogenannter “Codegerüche“ (siehe hierzu die entsprechende Arbeit[Seminar03]) und stellt einen Katalog von Refactoring-Transformationen in sieben Kapiteln vor, welche er dann tabellarisch ihren Codegerüchen zuordnet. Die Kapitel beschreiben insgesamt 72 Refactoring-Transformationen zur Komposition von Methoden, dem Verschieben zwischen Klassen, zur Datenorganisation, dem Vereinfachen bedingter Anweisungen, dem Vereinfachen von Methodenaufrufen, dem Umgang mit Generalisierung, sowie ein Kapitel mit sog. “großen“ Refactoring-Transformationen.

Fowler verzichtet im Gegensatz zu Opdyke auf die explizite Auszeichnung von Basistransformationen und die formale Beschreibung von Vorbedingungen zur Anwendung der Transformationen. Statt dessen beschreibt er die einzelnen Methoden schrittweise nach Art eines Kochrezeptes, bevor er die Ausführung an einem Beispiel erläutert.

### 1.2.3 Unterschiede

Begründet sind diese Unterschiede durch den unterschiedlichen Kontext der beiden Arbeiten. Opdyke erstellte seine Refactorings für C++ und Smalltalk, mit dem Augenmerk

auf spätere formale Komposition und Automatisierbarkeit, während Fowler seine Transformationen auf Java im praktischen Einsatz ausrichtet und die Beschreibungen zur Begleitung der Arbeit von Softwareentwicklern erstellt.

## 1.3 Formal einheitliches Refactoring?

Auch ohne die unterschiedlichen Herangehensweisen der Autoren ist es schwierig, einen allgemeinen Katalog von Transformationen zu beschreiben, dessen Methoden untereinander kompatibel sind. Dazu sind gemeinsame Ansichten bzgl. der Hierarchisierung, z.B. nach Basistransformationen und komponierten Transformationen, sowie dem Umgang mit Unterschieden zwischen Programmiersprachen notwendig[1].

Erstrebenswert ist ein solcher Katalog aufgrund der Wiederverwertbarkeit, welche ja für Programme Ziel des Refactorings ist. Es gibt keinen Grund, warum diese Wiederverwertbarkeit nicht auch für die Transformationen des Refactorings selbst angestrebt werden sollte.

[1] Opdyke selbst gibt an, daß nicht alle Transformationen für C++ auch für Smalltalk anwendbar sind. [Opdyke92], S. 32

# 2 Transformationen aus [Fowler2000]

In diesem Kapitel wird ein Ausschnitt aus dem Katalog von Refactoring-Transformationen genauer vorgestellt, den Fowler in seinem Buch “Refactoring - Improving the design of existing code“ beschreibt. Anschließend werden zwei Refactoring-Transformationen aus diesem Katalog Schritt für Schritt erklärt.

## 2.1 Übersicht

Der Ausschnitt aus [Fowler2000] beschränkt sich auf die Kapitel “Verschieben zwischen Klassen“, “Datenorganisation“, “Vereinfachen bedingter Anweisungen“, welche zusammen 32 Refactoring-Transformationen umfassen. Der Inhalt dieser drei Kapitel läßt sich nach drei Hauptzielen ordnen, welche durch das Ausführen der Refactoring-Transformationen erreicht werden.

1. Funktionale Refactoring-Transformationen
   Diese erleichtern das Hinzufügen neuer Funktionen nach Abschluß der Transformation. Dazu gehört:

   - Verschieben zwischen Klassen
   - Klassenassoziation ändern
   - Bedingten Anweisung durch Polymorphie ersetzen

2. Verbesserung von Programm-Stabilität und -Sicherheit
   Dazu gehören:

   - Datenkapselung
   - Duplikation vermindern/kontrollieren
   - Lesbarkeit erhöhen

3. Workaround-Transformationen
   Die Transformationen dieser Gruppe sind von Fowler selbst mit dem Hinweis versehen worden, daß sie nur eine zeitlich begrenzte Lösung darstellen sollten und werden daher nicht zur Verwendung empfohlen.

Diese Unterteilung in sieben Untergruppen geschieht zur leichteren Beschreibung der Transformationen und um ihrem jeweiligen Zweck besser gerecht werden zu können und trennt die von Fowler vorgenommene Einteilung nach Kapiteln weiter auf. Die einzelnen Refactoring-Transformationen sind zum Vergleich im Anhang nach dem Schema von Fowler und nach dem zweckorientierten Schema aufgelistet.AB

Es folgt die nach Kategorien geordnete Beschreibung der Refactoring-Transformationen.

### 2.1.1 Verschieben zwischen Klassen

Während der Programmentwicklung entstehen immer wieder Methoden, welche hauptsächlich Funktionalität anderer Klassen als derjenigen nutzen, in der sie definiert sind. Um die innere Kohärenz der Klassen wieder herzustellen, sollten die Methode und zugegriffene Variablen bzw. Methoden in eine gemeinsame Klasse verschoben werden. Hierzu gibt es die Transformationen *Methode* und *Variable verschieben.*

Auch das Beseitigen zu kleiner Klassen bzw. Unterklassen durch einreihen ihrer Bestandteile in andere Klassen und das Aufteilen zu groß gewordener Klassen in mehrere kleinere Klassen wird durch Transformationen beschrieben (*Klasse extrahieren/einreihen, Unterklasse durch Variablen ersetzen*).

Sinn von Delegation ist das Übertragen einer Aufgabe an einen Delegaten, ein Objekt einer anderen Klasse. Dies ist z.B. bei Schnittstellenobjekten der Fall, welche den Zugriff auf eine Softwarekomponente regeln und Ergebnisse eigener Methoden durch Aufrufe von Methoden ihrer Delegatenobjekte berechnen lassen. Für das Herstellen einer Delegation bietet Fowler die Transformation *Delegaten verstecken*, für das Entfernen einer nicht mehr erwünschten Delegation die Transformation *Middle man entfernen* an.

### 2.1.2 Klassenassoziation ändern

Um neue Funktionalitäten einführen zu können, ist es bisweilen nötig, zwischen Klassen mit injektiver Assoziation eine bijektive Assoziation einzuführen, z.B. um eine Vertragsliste für einen Kunden zu erstellen, aber nur den Verträgen der jeweilige Kunde bekannt ist. Dazu stellt Fowler die Transformation *Injektive durch bijektive Klassenassoziation ersetzen* vor.

Ist eine vorhandene bijektive Klassenassoziation unnötig geworden und soll ersetzt werden, so läßt sich die Umkehrung *Bijektive durch injektive Klassenassoziation ersetzen* einsetzen. Siehe hierzu Abb. 2.1.

### 2.1.3 Bedingte Anweisung durch Polymorphie ersetzen

Typvariablen dienen oft dazu, die Ausprägung des Merkmals eines Objektes zu speichern. Dies ist dann unangenehm, wenn das Verhalten des Objektes zur Laufzeit von dieser Ausprägung abhängig ist, was durch das Abfragen der Typvariable in bedingten Anweisungen angezeigt wird. Der erste Nachteil ist das unterschiedliche Verhalten von Objekten der gleichen Klasse zur Laufzeit, der zweite Nachteil ist dadurch gegeben, daß ggf. an mehreren Programmstellen der Zustand der Typvariablen abgefragt wird. Sollen nun neue

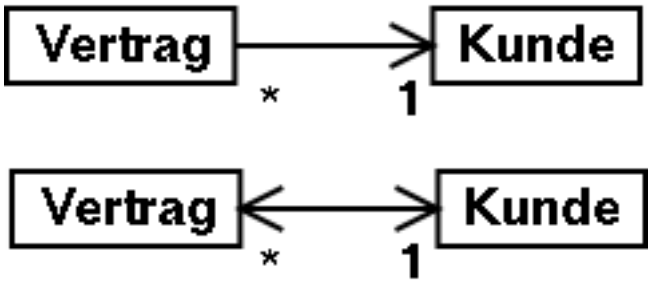


Abbildung 2.1: Klassenassoziation ändern. Oben die injektive, unten die bijektive Assoziation im UML-Diagramm. Wechsel durch Transformation möglich.

Ausprägungen hinzugefügt werden, müssen alle diese Stellen gefunden und modifiziert werden, was schnell zu Fehlern führen kann. Um diese beiden Nachteile zu beheben, stellt Fowler die Transformation *Bedingte Anweisung durch Polymorphie ersetzen* vor. Dabei werden die Typvariable, zugehörige bedingte Anweisungen und mögliche Ausprägungen der Typvariablen entweder durch Unterklassen oder ein Strategie- oder Zustandsmuster ersetzt. Wie diese Transformation genau abläuft, ist im Abschnitt 2.2.2 erläutert.

### 2.1.4 Datenkapselung

Datenkapselung ist ein zentraler Punkt der objektorientierten Programmierung, der hilft, ein Programm stabil und wartbar zu halten.

Öffentlich zugreifbare Daten eines Objektes brechen die Datenkapselung auf, da Änderungen dieser Daten nicht mehr der Kontrolle des Objektes unterliegen. An diesen Stellen muß der öffentliche Zugriff auf die Daten unterbunden und durch kontrollierten Zugriff mittels Methodenaufrufen an das Objekt ersetzt werden. Fowler unterscheidet hier zwischen einfachen Variablen (*Variable kapseln*) und dem Kapseln einer Containervariable (`Collection` *kapseln*).

Das Übertragen von gemeinsam benutzten Variablen in Methodenaufrufen macht nicht nur Programme unleserlich, oft bestehen auch Abhängigkeiten zwischen diesen Variablen. Die gemeinsame Übertragung dieser Variablen wird vereinfacht und Abhängigkeiten können ausgedrückt werden, wenn diese Variablen in eine gemeinsame Klasse verschoben werden, deren Instanzen die alten Variablen ersetzen (*Variablenmenge durch Datenklasse ersetzen*). Das Einführen neuer Objekte zur Datenkapselung ist auch vorteilhaft wenn ein einfaches Datum um weitere Daten oder Funktionen erweitert werden muß (*Datum durch Objekt ersetzen*).

Arrays dienen normalerweise dazu, eine geordnete Menge von gleichartigen Daten zu verwalten. Für den Fall, daß man auf eine Verwendung stößt, wo einzelnen Feldern des Arrays unterschiedliche Bedeutungen zukommen, z.B. Speichern einer Adresse in der Reihenfolge Name, Vorname, Straße, etc., empfiehlt Fowler dieses Array durch ein Objekt zu ersetzen, welches eigene Zugriffsmethoden für die verschiedenen Felder bereitstellt (`Array` *durch Objekt ersetzen*).

Die Verwendung primitiver Datentypen, v.a. des Typs Integer zur Speicherung der Aus-

prägung eines Merkmals eines Objekts ist fehleranfällig, da bei der Parameterübergabe nicht geprüft werden kann, ob das übergebene Integer tatsächlich das gewünschte Merkmal repräsentiert. So wird die Vertauschung der Parameterreihenfolge im folgenden Codebeispiel nicht bei der Compilierung, sondern erst durch fehlerhaftes Programmverhalten zur Laufzeit gefunden, da keine Steuerklasse 10000 existiert.

```
int einkommen = 10000;
int steuerklasse = 1;
Document doc; ...
doc = erstelleSteuerBescheid(einkommen, steuerklasse,...);
...
private Document erstelleSteuerBescheid(int steuerklasse,
    int einkommen,...) {
```

Um diese Fehlerquelle zu beseitigen stellt Fowler die Refactoring-Transformation *Typvariable durch Objekt ersetzen* vor. Diese ist aber nur einsetzbar, wenn eine Typvariable nicht das Objektverhalten beeinflußt. Ein starker Gegenindikator ist das Abfragen der Typvariable in bedingten Anweisungen. In diesem Fall sind die Transformationen des Pakets "Bedingte Anweisung durch Polymorphie ersetzen" gefragt.

Ist es wünschenswert, daß alle Zugriffe auf eine Variable, auch innerhalb der eigenen Klasse, mittels Zugriffsmethoden stattfinden, so kann dieser Zustand mittels der Transformation *Selbstkapselnde Variable* hergestellt werden. Die Frage ob eine privat direkt zugreifbare Variable auch dort nur durch `get` und `set` Methoden zugegriffen darf überläßt Fowler dem persönlichen Geschmack. Fest steht allerdings, daß selbst nach Durchführen der Refactoring-Transformation nicht zu verhindern ist, daß ohne die Zugriffsmethoden direkt auf die Variable zugegriffen wird.

### 2.1.5 Duplikation vermindern/kontrollieren

Duplikation kann in einem Programm zweifach auftreten. Einmal im Quellcode, wo doppelte Codestücke bei der Wartung und Weiterentwicklung mehrfachen Änderungsaufwand und fehlerhaftes Programmverhalten verursachen. Zum anderen zur Laufzeit, wo viele identische Objekte für den gleichen Zweck den Speicherverbrauch erhöhen.

Werden in einem Programm Werte mit spezieller Bedeutung verwendet (sog. magic numbers), z.B. ein Preismultiplikator für die Einkommenssteuer, so empfiehlt Fowler diese Werte durch die Verwendung einer einmalig definierten Konstante zu ersetzen, damit Änderungen des Wertes auf die Definition der Konstante beschränkt werden können (*Magic number durch Konstante ersetzen*).

Finden sich in einer bedingten Anweisung mehrere Anweisungsteile mit identischen Rückgabewerten, so lassen sich diese ggf. zusammenfassen. Sollte keiner der Anweisungsteile weitere Seiteneffekte im Programm haben, so können diese mittels *Bedingte Anweisung konsolidieren* zusammengefaßt werden.

Taucht ein Befehl in allen Verzweigungen einer bedingten Anweisung auf, so kann er ggf. vor oder hinter die bedingte Anweisung verschoben und in den Verzweigungen

gelöscht werden. Die Refactoring-Transformation *Doppelte Anweisungen konsolidieren* extrahiert mehrfach vorhandene Befehle und faßt sie zusammen.

Werden viele gleichartige Instanzen einer Klasse erzeugt und müssen zwischen diesen Änderungen ausgetauscht werden, so ist es angezeigt, jede dieser Instanzen durch eine Referenz zu ersetzen (*Wert durch Referenz ersetzen*). Muß hingegen eine Instanz dupliziert werden, um unterschiedliche Zustände anzunehmen, so kann die Umkehrung *Referenz durch Wert ersetzen* genutzt werden.

Neben der ungewollten und fehleranfälligen Duplikation gibt es auch notwendige Duplikation. Z.B. in Client-Server Systemen nach MVC Konzept, wo zur Laufzeit `n` Instanzen der Daten für die View auf Clientseite existieren und eine Dateninstanz für das Modell auf Serverseite. Um sicherzustellen, daß Änderungen der Daten im Modell auch die Aktualisierung der Daten in den Views zur Folge haben, kann das *Beobachter* Muster [GoF1996] eingesetzt werden (*Observierte Daten duplizieren*). Dabei setzt die observierte Dateninstanz auf Serverseite alle beobachtenden Instanzen auf Clientseite von Änderungen in Kenntnis, welche von diesen übernommen werden.

### 2.1.6 Lesbarkeit erhöhen

Für das Entzerren oft unübersichtlicher bedingter Anweisungen gibt Fowler zwei Transformationen an, um diese zu vereinfachen. Dabei unterscheidet er zwischen einfachen bedingten Anweisungen (*Bedingte Anweisung zerlegen*) und geschachtelten bedingten Anweisungen (*Geschachtelte Bedingungen durch guard clauses ersetzen*). Im wesentlichen werden die Bedingungs- und Anweisungsteile der Anweisung dabei in eigene Methoden mit sprechenden Namen extrahiert.

Auch das Ersetzen spezieller Werte durch Konstanten mit sprechenden Namen (*Magic number durch Konstante ersetzen* gehört hierher, wenn es auch schon in Abschnitt 2.1.5) beschrieben ist.

Unübersichtliche Programme kommen nach Fowlers Einschätzung auch durch die Verwendung von Kontrollvariablen in Schleifen zustande und er schlägt vor, diese durch Strukturen mit `break` und `return` zu ersetzen (*Kontrollvariable ersetzen*).

Werden an einer Programmstelle implizite Annahmen über den Zustand einer Variablen gemacht, finden sich diese oft als Kommentar wieder, der bei der Fehlersuche den Softwareentwickler unterstützen soll. Vorteilhafter ist der Gebrauch der Klasse `Assertion`, welche dazu dient, die impliziten Annahmen automatisch abzuprüfen. Zum Herstellen der automatischen Prüfung beschreibt Fowler die Refactoring-Transformation *`Assertion` einführen.*

### 2.1.7 Workaround-Transformationen

Für den Umgang mit nicht modifizierbaren Bibliotheksklassen stellt Fowler zwei Transformationen vor. *Fremdmethode einführen* und *Lokale Erweiterung einführen.* Die Transformationen dienen jeweils dazu, einer nicht modifizierbaren Klasse eine Methode oder einen Wert hinzuzufügen.

Dazu werden die neuen Bestandteile an ihrer Verwendungsstelle bzw. in einer Erweiterung der Bibliotheksklasse definiert. Beide Transformationen führen zu einem starken Fall des Codegeruches *Featureneid* und sind zudem nicht in der Lage, den unsachgemäßen Gebrauch der Oberklasse ohne die Erweiterungen zu unterbinden.

Daher empfiehlt Fowler selbst den Einsatz nur für einen begrenzten Zeitraum und längerfristig die Änderungen von dem Eigentümer der Bibliothek vornehmen zu lassen.

## 2.2 Beispiele

In diesem Abschnitt werden die Refactoring-Transformationen *Methode verschieben* und *Bedingte Anweisung durch Polymorphie ersetzen* Schritt für Schritt beschrieben. Die Transformation *Methode verschieben* wurde ausgewählt, weil sie zu den Transformationen aus dem Katalog Fowlers gehört, welche häufig wieder verwendet werden, z.B. innerhalb von *Klasse extrahieren/einreihen* (S. 8). *Bedingte Anweisung durch Polymorphie ersetzen* hingegen ist ein gutes Beispiel dafür, wie man ablauforientierte Programmierung mit bedingten Anweisungen durch einfache Objektorientierung ersetzen kann.

### 2.2.1 Methode verschieben

Wenn es während der Programmentwicklung vorkommt, daß eine Methode hauptsächlich auf eine andere Klasse zugreift, anstatt Variablen und Methoden der Klasse zu verwenden, in der sie definiert ist, dann sollte diese Methode in die Klasse verschoben werden, auf die sie die meisten Zugriffe macht. Dadurch werden Daten und die auf ihnen definierten Operationen ganz im Sinne des objektorientierten Ansatzes zusammengeführt. Dazu wird zunächst eine ähnliche Methode, Zielmethode genannt, in der Klasse angelegt, die von der Methode am meisten gebraucht wird, der sog. Zielklasse. Die Methode in der alten Klasse, Quellmethode bzw. Quellklasse genannt, bleibt als Delegationsmethode, welche die Methode in der Zielklasse aufruft, ggf. erhalten oder wird ganz entfernt. Aufrufe der verschobenen Methode werden, wenn möglich an Objekte der Zielklasse direkt gerichtet oder von der Delegationsmethode der Quellklasse an die Zielmethode weitergeleitet.

#### Vorüberlegungen

Bevor man die Transformation ausführt gilt es als wichtigsten Punkt zu entscheiden, ob die Quellmethode als Delegationsmethode erhalten bleiben soll oder nicht. Des weiteren muß entschieden werden, ob weitere Variablen und Referenzen ebenfalls verschoben werden sollen: Hierzu sind drei Stellen zu betrachten:

1. **Ober- und Unterklassen**. Wird die zu verschiebende Methode in einer Ober- oder Unterklasse der Quellklasse überschrieben, so muß sie als Delegationsmethode aus Gründen der Semantikerhaltung bestehen bleiben.

2. **Aufrufstellen**. Kann oder soll an den Stellen, wo Aufrufe der zu verschiebenden Methode stattfinden, das entsprechende Objekt der Zielklasse nicht referenziert werden, so muß an diesen Stellen die Delegationsmethode in der Quellklasse weiter verwendet werden.

3. **Verwendete Elemente der Quellklasse**. Dies bezieht sich nur auf Variablen, Methoden oder Referenzen, welche die Quellmethode verwendet hat, die aber in der Zielmethode nicht zugänglich sind. Diese können entweder mit verschoben oder durch Parameterübergabe in der Zielmethode zugänglich gemacht werden. Verschieben ist nur möglich, wenn die verwendeten Elemente in der Quellklasse nicht weiter benötigt werden. Für verwendete Methoden muß die gesamte Transformation *Methode verschieben* quasi rekursiv durchgeführt werden. Können nicht alle verwendeten Elemente verschoben werden, so muß die Delegationsmethode erhalten bleiben und nicht verschiebbare Elemente in Parameterform übergeben werden.

### Durchführung

Nachdem entschieden wurde, ob eine Delegationsmethode erhalten bleiben soll und welche weiteren Elemente verschoben werden sollen, kann mit der Transformation begonnen werden.

Müssen keine weiteren Elemente verschoben werden und bleibt in der Quellklasse eine Delegationsmethode erhalten, endet die Transformation erfolgreich mit Abschluß von Schritt 4. Anderenfalls endet sie nach Schritt 6 bzw. Schritt 7.

1. **Zielmethode deklarieren**. In der Zielklasse muß zunächst die Signatur der Zielmethode definiert werden. Zusätzlich zu den Parametern der Quellmethode müssen auch alle Elemente übergeben werden, welche in der Zielmethode noch unbekannt sind, egal ob sie noch verschoben werden oder nicht. Kann die Klasse fehlerfrei compiliert werden, war der Schritt erfolgreich.

2. **Funktionalität kopieren und anpassen**. Anschließend wird der Methodenrumpf aus der Quellmethode in die Zielmethode kopiert. Anpassungen müssen nur erfolgen, wenn Parameter der neuen Methode andere Namen haben als ihre entsprechenden Elemente der Quellklasse. Nach einem fehlerfreien Compilieren sollte die Zielmethode bei Funktionstests das gleiche Verhalten zeigen wie die Quellmethode.

3. **Zielmethode in der Quellmethode referenzieren**. Besteht zwischen der Quell- und der Zielklasse noch keine Assoziation, so muß sichergestellt werden, daß in der Quellmethode die neue Methode der Zielklasse aufgerufen werden kann.

4. **Quellmethode in Delegationsmethode wandeln**. Dazu wird der Methodenrumpf durch einen Aufruf der Zielmethode ersetzt. Nach diesem Schritt sollte das Programmverhalten sich nicht geändert haben.

5. **Iteratives und rekursives Verschieben weiterer Elemente der Quellklasse**. Variablen, Referenzen und Methoden der Quellklasse, welche ebenfalls verschoben

werden sollen, können jetzt mittels der Transformationen *Methode/Variable verschieben* in die Zielklasse gebracht werden. Das Verschieben von Methoden kann dabei das Verschieben weiterer Elemente nach sich ziehen. Fowler gibt keine Reihenfolge an, in welcher das Verschieben zu geschehen hat, sondern überläßt diese Entscheidung dem Anwender.

6. **Entfernen temporärer Parameter aus der Signatur der Zielmethode**. Elemente der Quellklasse, welche im letzten Schritt in die Zielklasse verschoben worden sind, können nun direkt referenziert und aus der Parameterliste der Zielmethode entfernt werden.

7. **Anpassen der Aufrufstellen und Entfernen der Delegationsmethode**. Gibt es keine Gründe, die Quellmethode als Delegationsmethode zu erhalten, so sollten alle zugehörigen Aufrufstellen durch Aufrufe der Zielmethode ersetzt und die Methode gelöscht werden.

#### Automatisierbarkeit

In der einfachen Version (Erhalt einer Delegationsmethode, Übergabe aller benötigten Elemente als Parameter) ist die Automatisierung der Transformation einfach. Neben der Beachtung von Randbedingungen wie der Eindeutigkeit vergebener Namen etc. bereitet nur die Namensgebung der zusätzlichen Parameter (Schritt 1) und die Referenzierung der Zielmethode eventuell Schwierigkeiten.

Die Entscheidung zum Beibehalten der Delegationsmethode und das zusätzliche Verschieben weiterer Elemente der Quellklasse läßt sich nicht automatisch entscheiden.

Eine Implementierung für statische Methoden findet sich im Eclipse-Framework.

### 2.2.2 Bedingte Anweisung durch Polymorphie ersetzen

In Abschnitt 2.1.3 wurde beschrieben, warum es sinnvoll ist, Typvariablen und bedingte Anweisungen durch Unterklassen oder ein Strategie- bzw. Zustandsmuster zu ersetzen, wenn aufgrund der Typvariablen das Verhalten des Objektes geändert wird, in dem sie enthalten ist.

Dazu wird zunächst, so notwendig, die betroffene bedingte Anweisung in eine eigene Methode verlegt. Danach muß entschieden werden, ob die einzelnen Verzweigungen der bedingten Anweisung direkt in eigene Unterklassen oder die Unterklassen eines Strategie- bzw. Zustandobjektes verlegt werden sollen. Die bedingte Anweisung wird dann Verzweigung für Verzweigung durch Implementierungen in den gebildeten Unterklassen ersetzt. Für Änderungen an der alten Typvariable werden danach Instanzierungen der neuen Unterklassen eingesetzt.

#### Das Strategie- bzw. Zustandsmuster

Die beiden Muster entstammen dem Buch Entwurfsmuster von Erich Gamma u.a. [GoF1996] und dienen dazu einen Algorithmus oder einen Zustand von den Klassen zu

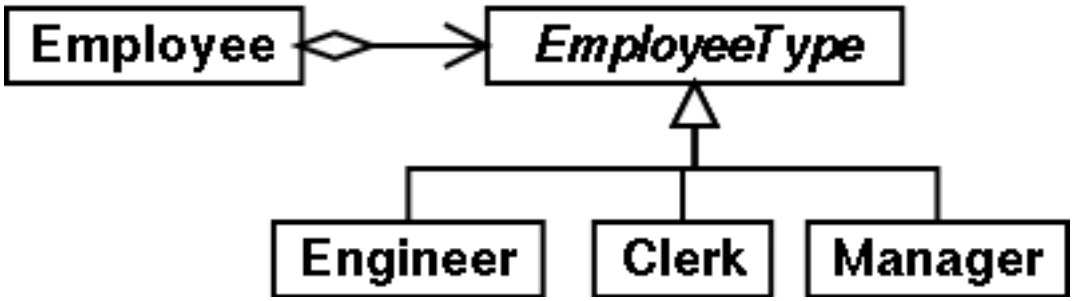


Abbildung 2.2: Zustandsmuster nach [GoF1996]. Der Zustand (Typ) eines `Employee` wird in einer Referenz gehalten. Die Zustandsoberklasse `EmployeeType` ist abstrakt, der Zustand wird durch Instanzieren einer Unterklasse gesetzt.

trennen, in denen sie verwendet werden. Die Umsetzung der beiden Muster ist sehr ähnlich, weshalb Fowler sie immer gemeinsam nennt.

Ein Algorithmus oder ein Zustand wird durch die Referenz auf ein abstraktes Zustands- oder Strategieobjekt ersetzt. Algorithmus bzw. Zustand werden definiert, in dem eine konkrete Unterklasse des abstrakten Zustands/Strategie-Objekts instanziert wird. Entscheidungen, welche von diesem Zustand oder dieser Strategie abhängen, werden dann an das instanzierte Objekt delegiert. Ein Beispiel hierfür ist in Abb. 2.2 gegeben.

**Vorüberlegungen** Die einzige Vorüberlegung betrifft die Entscheidung zwischen dem Bilden direkter Unterklassen und dem Einführen eines Zustandsobjektes nach dem o.g. Muster.

Das direkte Verwenden von Unterklassen macht am wenigsten Arbeit, doch muß auf das Zustandsmuster zurückgegriffen werden, wenn folgendes zutrifft:

- **Die zu erweiternde Klasse besitzt bereits Unterklassen**. In diesem Fall kann die bedingte Anweisung nicht einfach durch weitere Unterklassen ersetzt werden. Oder:
- **Nach der Instanzierung eines Objektes kann die Typvariable geändert werden**. Da in diesem Fall das Verhalten des instanzierten Objektes mitgeändert werden muß, scheiden Unterklassen als Lösung aus. Gamma selbst empfiehlt gerade für diese Fälle die Verwendung des Zustandsmusters ([GoF1996], S. 398).

### Durchführung

Einzig die Frage, ob neu erzeugte Unterklassen direkt oder von einer neuen Zustandsklasse abgeleitet werden, beeinflußt den Ablauf dieser Transformation. Sie endet auf jeden Fall erst, wenn die Instanzierungsstellen der erweiterten Klasse modifiziert worden sind.

1. **Anweisung in eigene Methode verlegen**. Wenn die bedingte Anweisung Teil einer größeren Methode ist, muß diese erst in eine eigene Methode verlegt werden. Fow-

ler greift dazu auf seine Transformation *Methode extrahieren*,([Fowler2000], Kap. 6) zurück.

2. **Vererbungshierarchie erstellen**. Für das korrekte Erstellen der Unterklassen (und des Zustandsobjektes) beschreibt Fowler zwei Transformationen: *Typvariable durch Unterklassen ersetzen* oder *Typvariable durch Zustandsmuster ersetzen*. Jeder Verzweigung der bedingten Anweisung wird dabei eine Unterklasse zugeordnet.

   Der Ablauf der Transformationen soll nicht genauer beschrieben werden, doch soll festgehalten werden, daß beide Transformationen dafür sorgen, daß die Typvariable korrekt gekapselt wird und die benötigten Unterklassen erzeugt werden.

   Werden Unterklassen verwendet stellt die Transformation sicher, daß an den Instanzierungsstellen die korrekten Unterklassen gebildet werden.

   Wird das Zustandsmuster benutzt wird sichergestellt, daß die nun kontrollierten Zugriffe auf die Typvariable an die Instanz des Zustandsobjektes weitergeleitet werden.

3. **Methode mit Anweisung in Zustandsobjekt verschieben**. Dies ist nur notwendig, wenn tatsächlich das Zustandsmuster benutzt wird.

4. **Anweisungsteile in Unterklassen verschieben**. Die Methode mit der bedingten Anweisung (ob verschoben oder nicht) muß in einer der neu erzeugten Unterklassen überschrieben werden.

   Die Verzweigung der bedingten Anweisung, welche dieser Unterklasse in Schritt 2 zugeordnet wurde, wird nun als Methodenrumpf in die neue Methode der Unterklasse kopiert. Damit die Methode fehlerfrei arbeiten kann, müssen ggf. noch Teile der Oberklasse als `protected` deklariert werden.

   Läuft die Compilierung der Unterklasse fehlerfrei ab kann die Verzweigung aus der bedingten Anweisung der Oberklasse entfernt werden.

   Dieser Schritt wird für alle Verzweigungen wiederholt.

5. **Methode in Oberklasse `abstract` deklarieren**. Dies geschieht um zu verhindern, daß versehentlich auf die leere Implementierung der Methode in der Oberklasse zugegriffen wird. Damit werden zugleich Instanzierungen der Klasse selbst verboten.

6. **An Instanzierungsstellen Unterklassen verwenden**. Wurde das Zustandsmuster verwendet, müssen in diesem Schritt nur die Instanzierungsstellen des Zustandsobjektes angepaßt werden. Da Zugriffe auf die Typvariable in Schritt 2 gekapselt worden sind, befinden sich alle diese Instanzierungsstellen in der unter 2 erstellten `set` Methode für die Typvariable. War das Zustandsmuster nicht notwendig, müssen alle Instanzierungsstellen der Originalklasse modifiziert werden. Diese lassen sich automatisch finden, da diese inzwischen als `abstract` deklariert ist, weshalb bei der Compilierung Instanzierungen dieser Klasse als Fehler angezeigt werden.

An diesen Stellen muß entsprechend der angegebenen Typvariable die korrekte Unterklasse instanziert werden.

### Automatisierbarkeit

Die Automatisierbarkeit dieser Transformation ist nicht trivial. Dies betrifft z.B. die Identifikation aller betroffenen bedingten Anweisungen. Fowler hat nicht bedacht, daß das automatische Ersetzen eines Typcodes durch Unterklassen ggf. viele bedingte Anweisungen zugleich betreffen kann. Ggf. müsste man den Typcode auch in den Unterklassen mitführen, um kompatibel zu bleiben, bis der letzte lesende Zugriff auf die Typvariable ersetzt worden ist.

# 3 Erfahrungen mit Refactoring

Nach dem Einstieg in den Katalog von Refactoring-Transformationen nach Fowler werden in diesem Kapitel Erfahrungen mit Refactoring beschrieben werden. Zunächst werden Eindrücke aus der Literatur behandelt, danach soll an zwei Beispielen zum einen die Weiterentwicklung von Refactoring und zum anderen der Einsatz von Refactoring in der Lehre dargestellt werden.

## 3.1 Literatur

Refactoring ist kein vollständiger Software-Entwicklungsprozeß, sondern führt nur in Kombination mit anderen Techniken zu einem erfolgreichen Projektabschluß. Daher wird es nicht verwundern, daß Berichte über die Erfahrung mit Refactoring Teil von Berichten sind, welche die Umsetzung neuer Softwareprozesse beschreiben, v.a. eXtreme Programming. [XP1],[XP2], [XPClassroom], [XPTeaching1], [XPTeaching2], [XPTeaching3]

Allgemein wurden keine negativen Erfahrungen mit Refactoring dokumentiert, der Einsatz neuer Entwicklungsprozesse (incl. Refactoring) insgesamt positiv beurteilt.

Leider fehlen generell Untersuchungen zu den verwendeten Refactoring-Transformationen und ihrem praktischen Einsatz. Auch wurde nicht untersucht, was die Einführung von Refactoring in einen bereits praktizierten Softwareprozeß ohne weitere Neuerungen bewirkt. In diesem Szenario ließe sich die Wertsteigerung ohne Nebeneffekte durch andere neue Techniken beobachten. Untersuchungen zu beiden Themen stehen noch aus. Der positive Effekt von Refactoring wird so zwar allseits betont, ist aber nicht quantifizierbar.

## 3.2 Das Know-It-All Projekt

Refactoring-Transformationen beschränken sich bisher auf Änderungen auf der Ebene des Programmcodes und der Klassenhierarchie. Refactoring-Transformationen auch auf anderen Designebenen einzusetzen und Änderungen zwischen verschiedenen Modellierungsebenen korrekt zu übertragen ist Teil des Know-It-All Projektes an der Concordia University (CU).[KnowItAll]

Im Projekt sollen allgemein Methoden für die Framework-Evolution entwickelt werden. Frameworks empfehlen sich für evolutionäre Ansätze, da sie i.A. eine hohe Lebensdauer besitzen und die Anforderungen neuer zu hostender Anwendungen neue Anforderungen an das Framework mit sich bringen. Als funktionsfähiges und praxisnahes Beispiel

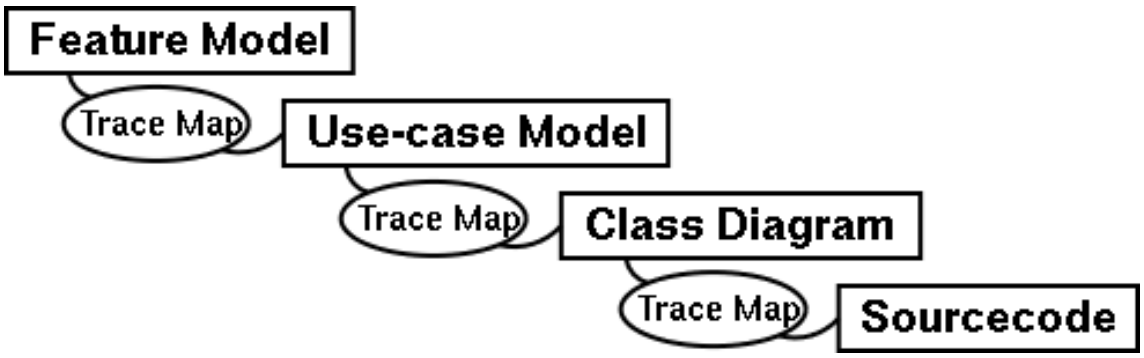


Abbildung 3.1: Die Modellierungsebenen für das Know-It-All Projekt der Concordia University als Rechtecke. Dazwischen als Ellipsen die Trace maps, welche beim Refactoring das kaskadierende Anstoßen von Refactoring-Transformationen auf unteren Modellierungsebenen erlauben sollen.

wird im Rahmen des Projektes an einem Framework für Datenbankmanagementsysteme gearbeitet.

Für den Entwurf des DBMS-Frameworks wurde die konsequente Nutzung mehrerer Modellierungsebenen angestrebt. Die verschiedenen Modellierungsebenen sind in Abb. 3.1 wiedergegeben.

Es ist intuitiv klar, daß Änderungen am Entwurf auf höheren Modellierungsebenen Änderungen in darunter gelegenen Ebenen nach sich ziehen müssen, was die Frage nach der formalen Verbindung zwischen diesen Änderungen mit sich bringt. Für den Transfer von Änderungen zwischen verschiedenen Ebenen werden an der CU sog. Trace maps entwickelt.

Parallel werden Refactoring-Transformationen für die höheren Designebenen entwickelt, durch die mittels der zugehörigen Trace maps weitere Transformationen auf unteren Ebenen angestoßen werden. Dies bezeichnen die Entwickler als cascading refactoring, da sich Transformationen erwartungsgemäß bis auf die Ebene des Quellcodes erstrecken.

So könnte z.B. das Verschieben eines Methodennamens im Klassendiagramm per Drag-and-Drop die automatische Ausführung einer Transformation *Methode verschieben* auf Sourcecode-Ebene, ähnlich wie im Abschnitt 2.2.1 beschrieben, über eine zugehörige Trace map anstoßen.

Die Refactoring-Transformationen für höhere Modellierungsebenen und der Formalismus für Trace maps befinden sich erst am Anfang der Entwicklungsphase, der Ansatz verspricht aber eine Menge komponierbarer, formal beschriebener und automatisierbarer Refactoring-Transformationen zum Ergebnis zu haben, v.a. aber Transformationen, welche sich auf Ebenen oberhalb des Klassendiagramms nutzen lassen.

## 3.3 Refactoring in der Lehre

Neue Programmiertechniken werden schnell in die Lehre der Softwareentwicklung übernommen, so daß auch zu Refactoring in der Lehre bereits mehrere Untersuchungen vor-

liegen, wiederum als Teil von eXtreme Programming vermittelt.

Z.B. hat sich Refactoring an der Duke University, Durham NC, USA, bei der Einführung von Entwurfsmustern als vorteilhaft erwiesen.[XPClassroom]

Dazu wurden den Studenten kleine, aufeinander aufbauende Aufgaben gestellt, welche die Implementierung der Klassenhierarchie des Pinball-Spieles aus Timothy Budd's Buch: "Understanding object-oriented Design in Java" zum Ziel hatten.[Budd98]

Bei den Programmentwürfen der Studierenden wurden auch fehlerhafte Entwürfe für die jeweils nächste Aufgabe zugelassen, bevor eine designtechnisch bessere Modellierung mittels Entwurfsmustern am Beispiel vorgeführt wurde. Nach dem notwendigen Refactoring des eigenen Programms anhand des Beispiels waren die Studierenden der formalen Einführung des Entwurfsmuster gegenüber wesentlich aufgeschlossener.

Der positive Effekt von eXtreme Programming und Refactoring wird auch in anderen Studien bestätigt.[XPClassroom][XPTeaching1][XPTeaching2][XPTeaching3]

## 3.4 Fazit

Es hat sich gezeigt, daß der Effekt reinen Refactorings wenig untersucht ist, Refactoring sich aber als Teil neuer Softwareprozesse verbreitet und gut aufgenommen wird.

Bestrebungen zielen darauf, das Anwendungsgebiet auf Quellcode- und Klassenhierarchieebene zu verlassen und Refactoring soll auch auf höheren Modellierungsebenen zum Einsatz zu bringen. Die dafür notwendigen Formalismen befinden sich aber noch in der Entwicklung.

Schließlich wurden positive Effekt von XP (und damit auch Refactoring) in der akademischen Lehre festgestellt, was der weiteren Verbreitung dieser Technik zugute kommen wird.

# 4 Zusammenfassung

Im vorliegenden Text wurden die Begriffe Refactoring und Refactoring-Transformation erläutert und Unterschiede bei der bisherigen Kategorisierung und Beschreibung von Refactoring-Transformationen anhand der Ergebnisse von Opdyke und Fowler dargelegt. Es wurde festgestellt, daß diese Unterschiede der Übertragbarkeit, Verknüpfung und Automatisierung von Transformationen unterschiedlicher Herkunft im Wege stehen.

Anhand des Ausschnittes aus dem Transformationen-Katalog von Fowler in Kapitel 2 sollte der Leser ein Gefühl für die Problemabdeckung entwickelt haben, welche bei diesen Katalogen angestrebt wird. Anhand der Beispiele *Methode verschieben* und *Bedingte Anweisung durch Polymorphie* aus dem gleichen Kapitel wurde die Ablaufgeschwindigkeit aber auch die Komplexität bei der manuellen Durchführung der Transformationen aufgezeigt.

Publikationen zu Erfahrungen mit Refactoring zeigen, daß Refactoring nicht allein, sondern als Teil neuer Softwareprozesse, v.a. eXtreme Programming, auftritt. Diese neuen Softwareprozesse begleitet Refactoring auch auf dem Weg in die akademische Lehre und wurde dort gut aufgenommen. Des weiteren sind Bestrebungen im Gange, Refactoring-Transformationen zu beschreiben, die außerhalb der bisher betrachteten Programmtexte und Klassendiagramme angewandt werden können. Die notwendigen Formalismen, damit diese Transformationen aber schlußendlich wieder auf den Programmtext zurückwirken können, befinden sich aber noch im Entwicklungsstadium.

Ähnliche Gedanken finden sich auch in einem aktuellen Papier zum Stand des Refactorings, welches aus Zeitgründen nicht mehr Teil dieses Textes werden konnte.
[Refactoring03]

# A Einteilung nach [Fowler2000]

**Kapitel 7 - Verschieben zwischen Klassen**

*Methode verschieben*
*Variable verschieben*
*Klasse extrahieren*
*Klasse einreihen*
*Delegaten verstecken*
*Middle man entfernen*
*Fremdmethode einführen*
*Lokale Erweiterung einführen*

**Kapitel 8 - Daten organisieren**

*Selbstkapselnde Variable*
*Variable durch Objekt ersetzen*
*Wert durch Referenz ersetzen*
*Referenz durch Wert ersetzen*
`Array` *durch Objekt ersetzen*
*Observierte Daten duplizieren*
*Injektive durch bijektive Assoziation ersetzen*
*Bijektive durch injektive Assoziation ersetzen*
*Magic number durch Konstante ersetzen*
*Variable kapseln*
`Collection` *kapseln*
*Variablenmenge durch Datenklasse ersetzen*
*Typvariable durch Klasse ersetzen*
*Typvariable durch Unterklassen ersetzen*
*Typvariable durch* `Strategie`*- oder* `Zustandsmuster` *ersetzen*
*Unterklasse durch Variablen ersetzen*

**Kapitel 9 - Bedingte Anweisungen vereinfachen**

*Bedingte Anweisung zerlegen*
*Bedingte Anweisung konsolidieren*
*Doppelte Anweisungen konsolidieren*
*Kontrollvariable entfernen*
*Geschachtelte Bedingungen durch guard clauses ersetzen*
*Bedingte Anweisung durch Polymorphie ersetzen*
`null` *Objekt einführen*
`Assertion` *einführen*

# B Zweckorientierte Einteilung

**Verschieben zwischen Klassen**
*Methode verschieben*
*Variable verschieben*
*Klasse extrahieren*
*Klasse einreihen*
*Delegaten verstecken*
*Middle man entfernen*
*Unterklasse durch Variablen ersetzen*

**Klassenassoziation ändern**
*Injektive durch bijektive Assoziation ersetzen*
*Bijektive durch injektive Assoziation ersetzen*

**Bedingte Anweisung durch Polymorphie ersetzen**
*Bedingte Anweisung durch Polymorphie ersetzen*
*Typvariable durch Unterklassen ersetzen*
*Typvariable durch* `Strategie`*- oder* `Zustandsmuster` *ersetzen*

**Datenkapselung**
*Selbstkapselnde Variable*
*Variable durch Objekt ersetzen*
`Array` *durch Objekt ersetzen*
*Variable kapseln*
`Collection` *kapseln*
*Variablenmenge durch Datenklasse ersetzen*
*Typvariable durch Klasse ersetzen*
`null` *Objekt einführen*

**Duplikation vermindern/kontrollieren**
*Wert durch Referenz ersetzen*
*Referenz durch Wert ersetzen*
*Observierte Daten duplizieren*
*Magic number durch Konstante ersetzen*
*Bedingte Anweisung konsolidieren*
*Doppelte Anweisungen konsolidieren*

**Lesbarkeit erhöhen**
*Magic number durch Konstante ersetzen*
*Kontrollvariable entfernen*
*Bedingte Anweisung zerlegen*
*Geschachtelte Bedingungen durch guard clauses ersetzen*
`Assertion` *einführen*

**Workaround-Transformationen**
*Fremdmethode einführen*
*Lokale Erweiterung einführen*

# Literaturverzeichnis


[Budd98] Timothy Budd: Understanding Object-Oriented Programming with Java. Addison-Wesley 1998 20

[Fowler2000] Fowler, Martin; Refactoring: Improving the design of existing code; 9th printing July 2002, Addison-Wesley
ISBN 0-201-48567-2 2, 3, 4, 7, 16, 22

[GoF1996] Gamma, Erich: Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software; 5. korrigierter Nachdruck der 1. Auflage, Addison-Wesley 2001
ISBN 3-8273-1862-9 11, 14, 15

[KnowItAll] The Know-It-All project
http://www.cs.concordia.ca/∼faculty/gregb/home/PDF/dosd-chapter.pdf 18

[Opdyke92] William F. Opdyke: Refactoring object-oriented frameworks, Ph.D. Thesis, University of Illinois, 1992
http://citeseer.nj.nec.com/cache/papers/cs/14208/ftp:zSzzSzwww.laputan.orgzSzpubzSzpaperszSzopdyke-thesis.pdf/opdyke92refactoring.pdf 3, 5, 6

[Refactoring03] Tom Mens u.a.: Refactoring - Current Research and Future Trends, Electronic Notes in Theoretical computer science 82 No3 (2003)
http://win-www.uia.ac.be/u/lore/refactoringProject/publications/refactoring-entcs2.pdf 21

[Seminar03] http://www.upb.de/cs/ag-kastens/lehre/index.htm, Material zum Seminar “Refactoring in eXtreme Programming“ 3, 5

[XP1] http://www.xpuniverse.com/2001/pdfs/EP202.pdf 18

[XP2] http://www.xp2003.org/conference/papers/Chapter27-Dunsmore+alii.pdf 18

[XPClassroom] Owen Astrachan, Robert C. Duvall, Eugene Wallingford: Bringing XP to the classroom
http://www.agilealliance.org/articles/articles/BringingXPToTheClassroom.pdf 18, 20

[XPTeaching1] Frank Keenan: Teaching and Learning XP, Dundalk Institute of Technology, Ireland
http://www.agilealliance.org/articles/articles/
FrankKeenan- -TeachingAndLearningXP.pdf 18, 20

[XPTeaching2] http://www.xpuniverse.com/2001/pdfs/Edu05.pdf 18, 20

[XPTeaching3] http://www.agilealliance.org/articles/articles/
PeterLappo- -ObservationsonTeachingandMentoringXP.pdf 18, 20