Fachbereich Informatik

Universität Hamburg

Diplomarbeit

Kleine Refactoring-Muster

Marko Schulz
Spannskamp 26
22527 Hamburg

ms@datenreisender.de
http://datenreisender.de/DA/

Mai 2004

Erstbetreuer

Prof. Dr. Heinz Züllighoven

Arbeitsbereich Softwaretechnik
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Zweitbetreuer

Prof. Dr. Horst Lichter

Lehr- und Forschungsgebiet
Informatik III (Softwarekonstruktion)
Fachgruppe Informatik
RWTH Aachen
52056 Aachen

„There is a structure on the patterns, which describe how each pattern is itself a pattern of other smaller patterns“
Christopher Alexander, [Ale79]

Danksagungen:

Früher fand ich Danksagungen in Diplomarbeiten immer etwas befremdlich, da es sich ja schließlich „nur“ um eine Diplomarbeit handelt. Jetzt stehe ich am Ende meiner Diplomarbeit und auch fast am Ende meines Studiums, blicke auf diese beiden zurück und habe eine ganz andere Einstellung zu Danksagungen.

Ich sehe, wie sehr ich die ganze Zeit in einem Netz voller Menschen gelebt habe, die mir persönlich und fachlich zur Seite standen. Ich bin froh darüber. Ich genieße es, nicht gleich dem Klischee eines einsamen Philosophens nur alleine über meinen Gedanken gesessen zu haben, sondern immer in der Gemeinschaft mit anderen Menschen gelebt, gelernt und gearbeitet zu haben.

Zunächst danke ich Prof. Dr. Heinz Züllighoven. In dem Arbeitsbereich SWT, welcher mit von ihm geleitet wird, habe ich durch unzählige Mitarbeiter, Studenten und Veranstaltungen viel gelernt. Besonders anregend war hierbei das Seminar „Ausgewählte Themen der Softwaretechnik“, welches auch gerade zum Gelingen dieser Arbeit beigetragen hat.

Ein besonderer Dank gilt dabei Martin Lippert, mit dem ich viele Konzepte dieser Arbeit diskutiert habe und der mich auch sonst immer mal wieder zur Weiterarbeit angetrieben hat. Ohne die sehr zügige Rückkopplung mit ihm wäre mir diese Arbeit nicht gelungen.

Ich danke auch Dr. Axel Schmolitzky, der vermutlich gar nicht ahnt, wie viele gute Ideen er in mir geweckt hat, während wir gemeinsam im Sommer 2003 eine Praktikumsgruppe geleitet haben und dabei immer wieder unsere Ansichten über objektorientierte Softwareentwicklung mit den Teilnehmern und untereinander diskutierten

Großer Dank gilt Conny Schulz, die mich während des größten Teiles meines Studiums begleitet hat. Ich wäre ohne Dich nicht so weit gekommen.

Mit Thomas Weiss, habe ich ebenfalls – wenn auch ganz anders – den größten Teil des Studiums durchlebt. Von Projekten, über die gemeinsame Studienarbeit, bis hin zu Prüfungen und einem Mittagsstammtisch in der Mensa reicht das Spektrum und gibt doch nur einen kleinen Ausschnitt wieder. Und auch bei dieser Diplomarbeit haben mir viele, kurzfristige Kommentaren unschätzbar geholfen. Ich möchte weder die fachliche noch die private Gemeinschaft missen.

Für die letzten Jahre möchte ich auch noch den Bewohneren der Burse danken. Dank Euch war dies ein Ort, an dem ich nicht nur einige Zeit gewohnt, sondern gerne gelebt habe. Besonders möchte ich Christin, Emily, Henning, Kai, Kati, Laloe, Linda, Thomas und Thomas nennen.

Es gibt noch viele weitere Menschen denen ich viel verdanke; zu jedem könnte ich viel erzählen. Aber dies ist nicht der Ort dafür und so muß es leider dabei bleiben, sie zu nennen: Meine Mutter, mein Vater, Bruni, Anja Ford, Anja Weiss, Allo, Christian, Frank.

Abschließender und besonderer Dank gilt Meike Coldewey. Für die letzten Jahre, für die Begleitung dieser Arbeit, für alles. Du verdienst viel mehr als diese Zeilen. Aber das steht eh jenseits von Worten. Danke. Vielen Dank.

Danke Euch allen, stellvertretend für noch viel mehr Menschen.

Inhaltsverzeichnis

Kapitel 1.

Einleitung

1.1. Motiviation

Die Beherrschung von Refactorings ist zu dem festen Handwerkszeug eines Softwareentwicklers geworden. Dies beruht vor allem auf dem Buch „Refactoring: Improving the Design of Existing Code“ von Martin Fowler, welches vielen erst aufgezeigt hat, wie man Software umgestalten kann und dabei sicherstellt, ihr Verhalten nicht zu verändern.

Das Hauptwerk von Fowler liegt in einem Katalog, in welchem er 68 Refactorings detailliert vorstellt. Um diesen Katalog herum demonstriert er die Ausführung von Refactorings und erklärt die allgemeinen Prinzipien, insbesondere wann welche Refactorings angewendet werden sollten. Er geht auch darauf ein, wie man die Ausführung von Refactorings mit Werkzeugen automatisieren kann.

Tatsächlich bieten immer mehr Entwicklungsumgebungen an, einzelne Refactorings automatisch durchzuführen: Für Java sind dies z.B. Eclipse und IntelliJ IDEA, für C# die nächste Version des MS Visual Studio.

Durch diese Unterstützungen kann man den Eindruck gewinnen, die manuelle Ausführung von Refactorings sei unnötig. Allerdings stellt sich das als Trugschluß heraus, vergleichbar mit der Fehlannahme, man könnte durch CASE-Tools vollständig darauf verzichten, Quelltext zu schreiben.

Es gibt bisher aber nur für eine Auswahl an Refactorings Werkzeugunterstützungen. Bei diesen kann die Ausführung oftmals nicht in allen Fällen vollautomatisch durchgeführt werden, in Zweifelsfällen muß der Programmierer eingreifen oder er wird darauf hingewiesen, daß das Refactoring erst möglich ist, wenn er die Rahmenbedingungen dafür geschaffen hat.

Falls man ein Refactoring durchführen will, welches sich nur teilweise mit jenen deckt, die durch Werkzeuge unterstützt werden, zeigt sich ein weiteres Defizit: Die existierenden Werkzeuge lassen sich meist nur in engen Grenzen an neue Gegebenheiten anpassen. Somit können die Werkzeuge schnell nutzlos für einen Programmierer werden, wenn er ein Problem lösen will, welches von den Entwicklern der Werkzeuge nicht vorgesehen war.

Im ungünstigsten Fall ist es erforderlich ein Refactoring durchzuführen, für das es keinerlei spezielles Werkzeug gibt, welches die Ausführung dieses Refactorings automatisiert. Für diesen Fall bieten die derzeitigen Entwicklungsumgebungen keine Unterstützung, die speziell auf Refactorings abgestimmt ist.

Wenn bei der Ausführung von Refactorings aber manuelles Eingreifen erforderlich bleibt, ist es auch nötig, weiterhin Refactorings selber zu verstehen. Hierfür muß man mit bekannten Refactorings vertraut sein, um diese anwenden zu können. Darüber hinaus muß man aber auch das Wesen von Refactorings verstehen. Das ist die Voraussetzung dafür, bekannte Refactorings anzupassen oder auch gänzlich neue Refactorings zu entwerfen, jeweils wie es einer Situation angemessen ist.

In der existierenden Literatur wird bisher aber das Gewicht darauf gelegt, konkrete Refactorings zu lehren. Es wird nur wenig darauf eingegangen, wie Refactorings allgemein aufgebaut sind und wie man mit Problemen umgeht, die durch eine Restrukturierung gelöst werden sollen, für die es aber kein vorhandenes Refactoring gibt.

Deswegen ist es nötig, Refactorings besser zu verstehen, wozu auch ihre innere Struktur gehört. Es stellt sich vor allem die Frage, nach den grundlegenden Bestandteilen von Refactorings. Wie sind Refactorings in ihrem Inneren aufgebaut und wie kann man sie umbauen oder selber erstellen?

1.2. Ziel der Arbeit

Diese Arbeit soll dazu beitragen, ein besseres Verständnis darüber zu gewinnen, was Refactorings sind und wie sie aufgebaut sind. Dieses Wissen soll nicht als Selbstzweck dienen, vielmehr soll es Softwareentwicklern bei der Arbeit mit Refactorings helfen.

Um das Wesen von Refactorings besser zu verstehen, wird untersucht, ob es legitim ist, sie als Muster zu betrachten. Um ihren inneren Aufbau besser zu verstehen, sollen typische Schritte in Refactorings untersucht und in Form von Mustern festgehalten werden, um dann konkrete Vertreter in einem Musterkatalog zu sammeln.

Dieser Katalog soll keinen Anspruch auf Vollständigkeit erfüllen, aber widerspiegeln, welches die wesentlichen Elemente von Refactorings sind.

1.3. Vorgehen und Struktur

Im Kapitel **Grundlagen: Muster und Refactorings** wird die Basis für diese Arbeit gelegt, indem die beiden grundlegenden Säulen vorgestellt werden: Auf der einen Seite wird dargelegt, welchen Inhalt das Konzept des Musters heute in der Softwaretechnik besitzt, auf der anderen Seite werden Refactorings präsentiert.

Weiterhin wird vorgestellt, welche Veröffentlichungen es bisher zu dem Thema dieser Arbeit – kleine Refactoring-Muster – gab.

Darauf basierend werden im Kapitel **Kleine Refactoring-Muster** die zentralen Begriffe der Arbeit entwickelt und vorgestellt: Refactoring-Muster und kleine Refactoring-Muster.

Für Refactoring-Muster werden die Merkmale von Mustern auf Refactorings abgebildet und es wird überprüft, ob diese Merkmale vorhanden sind. Diese Legitimation führt zu einer Definition von Refactoring-Muster, welche beschreibt, wie Refactorings als Muster betrachtet werden können.

Anschließend werden typischen Schritte in Refactorings als kleine Refactoring-Muster betrachtet und es wird untersucht, wie man diese kleinen Refactoring-Muster von herkömmlichen Refactorings abgrenzen kann. Dieses Wissen wird dann ebenfalls zu einer Definition für kleine Refactoring-Muster verdichtet.

Ein **Katalog kleiner Refactoring-Muster** ist der Inhalt des gleichnamigen, vierten Kapitels. Er ist nicht vollständig, da er nicht sämtliche kleinen Refactoring-Muster beschreibt, dient aber zwei Zwecken: Zum einen wird das Konzept des kleinen Refactoring-Musters aus dem vorherigen Kapitel veranschaulicht, indem es mit konkreten Exemplaren demonstriert wird. Zum anderen können diese Muster unmittelbar von Softwareentwicklern eingesetzt werden.

Die **Anwendung kleiner Refactoring-Muster** wird dann auch in dem fünften Kapitel erläutert. In diesem wird exemplarisch gezeigt, wie man mit kleinen Refactoring-Mustern Refactorings sowohl beschreiben, als auch auf ein spezifisches Problem anpassen kann.

In dem abschließenden, sechsten Kapitel wird ein **Fazit und Ausblick** präsentiert. Das Ergebnis der Arbeit wird zusammengefaßt und es wird verdeutlicht, wo jenseits der Grenzen dieser Arbeit noch weiterer Forschungsbedarf liegt.

1.4. Zur Wahl der Wörter

Ich folge der Auffassung, daß die Wahl von passenden Wörtern nicht nur eine Nebensächlichkeit ist, sondern wesentlich das Denken beeinflußt. Hierin fühle ich mich auch dadurch bestätigt, daß die Autoren von Mustern immer wieder betonen, wie wichtig die Wahl eines passenden Namens für ein Muster ist (siehe Kapitel 2): Ein treffendes Wort unterstreicht einen Gedanken und macht ihn verständlicher. Ein unpassendes Wort verwirrt.

Zur Wahl der Wörter müßte ich bei dieser Arbeit einige Entscheidungen treffen, die ich für so wichtig erachte, daß ich sie hier nennen will. Die Begründungen dazu finden sich im Anhang A, sie hätten den Rahmen dieser Einleitung gesprengt. Hier sind nur kurz die Ergebnisse zusammengefaßt:

- Obwohl dies ein deutschsprachiger Text ist, verwende ich das englische Wort *Refactoring* (plural: *Refactorings*) und gebe ihm damit den Vorzug gegen Übertragungen wie *Refaktorisierung*. Kombinationen mit anderen, deutschen Wörtern geschehen mit Bindestrich, wie auch in dem Wort *Refactoring-Muster* im Titel zu sehen ist.

- Auf den Bindestrich wird – zugunsten der Lesbarkeit – verzichtet, wenn das Wort *Muster* mit einem anderen deutschen Wort kombiniert wird: *Mustersprache*, nicht *Muster-Sprache*. Mit einer Mustersprache ist dann eine Sprache gemeint, deren Elemente Muster sind. Sollte eine musterhafte, also vorbildliche Sprache beschrieben werden, so wird dies durch andere Wörter getan.

- Diese Arbeit orientiert sich an der herkömmlichen Rechtschreibung, verzichtet also auf die Änderungen, welche durch die sogenannte Rechtschreibreform eingeführt wurden.

Kapitel 2.

Grundlagen: Muster und Refactorings

In diesem Kapitel werden die beiden Grundsteine der Arbeit erläutert: Muster und Refactorings. Abschließend wird noch dargelegt, welche Forschungsergebnisse und Erkenntnisse zu dem Thema dieser Arbeit bereits bekannt sind.

2.1. Muster

2.1.1. Der Weg des Mustergedankens in die Softwaretechnik.

Um deutlich zu machen, wie es zu der besonderen, heutigen Ausprägung des Musterkonzeptes in der Softwaretechnik kam, wird hier die Entwicklung dorthin aufgezeigt.

Diese Vorstellung von Mustern basiert auf den Ideen des (nicht Software-!) Architekten Christopher Alexander. Um zu beschreiben, wie man gute Räume, Häuser und ganze Stadtteile entwirft, verfaßte er 253 Muster [AIS77]. Nach diesen konkreten Mustern beschrieb er noch das Modell dahinter [Ale79]: Was sind Muster in der Architektur? Wozu dienen sie? Wie verwendet man sie?

Nach einigen Jahren wurden diese Gedanken zu Mustern von Kent Beck und Ward Cunningham entdeckt, auf die Softwareentwicklung angewendet und in einem Papier auf der OOPSLA 87 präsentiert [BC87]. Zunächst blieben diese Ideen aber noch weitgehend unbeachtet [C2 a].

1990 präsentierte Bruce Anderson das Konzept auf der Konferenz TOOLS, wo auch Erich Gamma anwesend war. Er griff das Thema auf und führte in seiner Doktorarbeit den Begriff „Design-Muster" ein, ohne sich dabei auf die Werke von Christopher Alexander zu beziehen [Gam91].

Auf den folgenden OOPSLA-Konferenzen organisierte Bruce Anderson Treffen zu Mustern, wo auch Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides den Entschluß faßten, einen Katalog von Entwurfsmustern zu erstellen. Diese Entscheidung führte schließlich zu dem Buch „Design Patterns: Elements of Reusable Object-Oriented Software"[GHJV94]. Es ist das bekannteste Buch zu Entwurfsmustern und wirkte als Katalysator, indem es die Konzepte vielen Softwareentwicklern bekannt machte, so daß das Wissen um Entwurfsmuster mittlerweile unumgänglich ist.

Im August 1993 organisierten Kent Beck und Grady Booch eine Klausurtagung zu Mustern. Die Teilnehmer beschlossen, auf den Arbeiten von Erich Gamma und Christopher Alexander aufzubauen und gründeten die Hillside Group [Hil]. Diese Gruppe organisierte 1994 die erste Konferenz zu Mustern in der Softwareentwicklung: Pattern Languages of Programs (PLoP). Insbesondere auf diesen Konferenzen wurde das Konzept von Mustern auch immer wieder auf andere Bereiche der Softwareentwicklung angewendet: z.B. Muster für Organisationen [Cop95] oder Muster für Lehre [Ped].

2.1.2. Was ist ein Muster?

Andere Bedeutungen

Das „Deutsche Universalwörterbuch"[Dud03] aus dem Dudenverlag enthält mehrere, allgemeine Definitionen dafür, was ein Muster ist und greift damit das allgemeine Verständnis dieses Begriffes auf:

> **1.** Vorlage, Zeichnung, nach der etw. hergestellt, gemacht wird [...] **2.** etw. in seiner Art Vollkommenes, nachahmenswertes, beispielhaftes Vorbild in Bezug auf etw. Bestimmtes [...]

Wie aber im vorangegangenen Abschnitt schon erläutert, geht das Konzept von Mustern – wie es heute in der Softwaretechnik üblich ist – auf Christopher Alexander zurück, welcher darunter etwas spezielleres beschrieben hat.

In dem Kontext der Informatik muß noch erwähnt werden, daß es in der Bildverarbeitung natürlich auch den Begriff des Musters gibt. Er ist dort aber völlig anders belegt und wird hier nicht behandelt.

Muster nach Christopher Alexander

Grundlegende Merkmale Um es von den allgemeinen und anderen Vorstellungen von Mustern abzugrenzen, wird hier das Konzept eines Musters nach Christopher Alexander vorgestellt. Er schreibt [AIS95, S. X]:

> „Jedes Muster beschreibt zunächst ein in unserer Umwelt immer wieder auftretendes Problem, beschreibt dann den Kern der Lösung dieses Problems, und zwar so, daß man diese Lösung millionenfach anwenden kann, ohne sich je zu wiederholen."

Hiermit sind bereits die drei wichtigsten Merkmale eines Musters erfaßt:

1. Problem
2. Lösung
3. Allgemeine Anwendbarkeit

In „The Timeless Way of Building"[Ale79, S. 247] wird dies weiter ausgeführt:

Ein Problem tritt in einem bestimmten Kontext auf und zeigt sich dadurch, daß in ihm unterschiedliche Kräfte aufeinander wirken. Durch die Lösung gelingt es einem, diese Kräfte in ein Gleichgewicht zu bringen.

Die allgemeine Anwendbarkeit bedeutet: Ein einmaliges Problem – mit dazugehöriger Lösung – oder gar etwas theoretisch Erdachtes kann noch kein Muster darstellen. Deswegen ist es zum allgemein anerkannten Kriterium geworden, daß ein Muster nicht erfunden, sondern nur entdeckt werden kann. Man muß mehrfach beobachten, wie eine ähnliche Lösung auf ein ähnliches Problem angewendet wurde, um ein Muster zu erkennen. In [Zül98] heißt es hierzu:

> "Ein Muster ist, allgemein betrachtet, ein Konzept, das auf unserer Erfahrung beruht. Auf der Grundlage von Erfahrung und der Reflexion darüber können wir wiederkehrende Muster erkennen."

Obwohl Muster allgemein einsetzbar sind, so werden sie doch immer in einem konkreten Kontext angewendet. Dieser Kontext bestimmt die spezielle Form der Anwendung. So ist jedes Muster allgemein anwendbar und jede seiner Anwendung doch einzigartig [Ale79, S. 462 f.].

Namen Um ein Muster wirklich be*greif*bar und *hand*habbar zu machen, so daß man es auch mit anderen teilen kann, muß es einen Namen erhalten. Einen Namen zu finden, ist häufig der letzte Schritt, wenn man ein Muster festhält. Gleichzeitig ist es aber auch eine besonders schwierige und wichtige Aufgabe: Ein schwacher Name zeigt, daß man den Kern eines Musters noch nicht gefunden hat; ein treffender Name dagegen macht ein Muster besser benutzbar [Ale79, S. 267 f.].

Durch ihre Namen sind Muster „sprachbildend“ und „da Muster unsere Wahrnehmung prägen, dienen sie uns zur Orientierung in unserer Umwelt“ [Zül98]. Dies deutet bereits auf die Mustersprachen hin, die in Abschnitt 2.1.4 erklärt werden.

Ordnungen Muster gibt es in unterschieden Größen [Ale79, S. 247 f.], [AIS95, S. XII + XIX ff.]: Von Mustern zu der Auswahl von Farben, über die Form von Häusern, über die Gestaltung von Stadtvierteln, bis zur Verteilung von Städten. Es kommt aber auch vor, daß Muster in einer anderen Größenordnung angewendet werden, als für die sie ursprünglich entdeckt wurden [AIS95, S. XXXVI].

Nach Alexander wendet man ein Muster nach dem anderen an – sequentiell, nicht parallel. Würde man versuchen, zu einem Zeitpunkt mehrere Muster umzusetzen, würde man keinem gerecht werden [Ale79, S. 399 f.]. Somit besteht das Arbeiten mit Mustern aus einer Vielzahl kleinster Schritte [Ale79, S. 354].

Die Reihenfolge, in der Muster angewendet werden, muß für jedes Vorhaben individuell gefunden werden, bestimmt das Endergebnis und orientiert sich an der Größe der Muster: Zunächst benutzt man Muster, welche einen weiterreichenden Einfluß haben und wendet dann die kleineren Muster an [Ale79, S. 379 f.][AIS95, S. XII].

Unterschiedliche Arten Christopher Alexander beschreibt, daß es für unterschiedlichste Dinge Muster gibt [Ale79, S. 62 ff.]. In seiner Domäne der Architektur hebt er besonders Muster von Ereignissen (engl. *Patterns of events*) und räumliche Muster (engl. *Patterns of space*) hervor, welche wechselseitig aufeinander wirken: Eine Bank in einem Park hat Einfluß auf die Ereignisse dort und wenn an einem Weg besonders viele Menschen entlanglaufen, hat dies Einfluß auf die baulichen Strukturen dort. Konkret zählt

Alexander in [AIS95] dann nur räumliche Muster auf. Denn obwohl er beschreibt, daß die Ereignisse prägenderen Einfluß auf die Menschen haben, so gestalten Architekten eben doch die räumlichen Gebilde.

2.1.3. Muster in der Softwaretechnik

Wie schon in der geschichtlichen Entwicklung in 2.1.1 gesagt, gibt es Muster für viele Aspekte der Softwareentwicklung. Den größten Augenmerk haben aber immer Muster über den inneren Aufbau von Software erhalten. Je nach Größe des Musters werden diese meist – von klein nach groß – als Programmiermuster (engl. *idioms*), Entwurfsmuster (engl. *design patterns*) und Architekturmuster (engl. *architectural patterns*) bezeichnet [BMR+96, S. 12]. Hierbei sind Entwurfsmuster mit Abstand am bekanntesten und werden am meisten behandelt.

In „Design Patterns: Elements of Reusable Object-Oriented Software" werden vier unerläßliche Bestandteile von Entwurfsmustern genannt [GHJV94, S. 3]:

- Der *Mustername*
- Das *Problem*, auf welches das Muster angewendet werden kann
- Die *Lösung*, die das Muster für das Problem bietet
- Die *Konsequenzen*, die mit dem Muster noch einher gehen

Diese Teile finden sich auch in anderen Büchern zu Softwaremustern ständig wieder (z.B. [BMR+96]). Oftmals etwas feiner aufgeteilt oder um zusätzliche, erklärende Punkte angereichert, wie auch in [GHJV94, S. 6 f.] selber.

Man erkennt bei diesen Bestandteilen sofort die Merkmale von Alexander wieder. „Konsequenzen" wird mit hinzugenommen, da es sich in der Arbeit mit Entwurfsmustern gezeigt hat, daß die weiteren Auswirkungen eines Musters ein wichtiges Entscheidungskriterium dafür sind, ob man ein Muster einsetzt. Sie sind ein so besonderer Teil der Lösung, daß sie herausgehoben werden.

Beschreibungen von Mustern

In viele Mustersammlungen werden feste Schablonen (engl. *Template*) benutzt, um die Muster zu beschreiben. Lesern hilft dies, die unterschiedlichen Bestandteile eines Musters schnell zu erkennen; Autoren werden durch Schablonen dazu angehalten, die Struktur eines Musters klar herauszuarbeiten.

Viele Autoren übernehmen aber auch einfach bekannte Formen, z.B. aus [GHJV94], [BMR+96] oder [Ale79], ohne dafür einen genauen Grund zu haben: „Als wesentliches Ergebnis läßt sich feststellen, daß die Wahl der geeigneten Form einer Musterbeschreibung vom angestrebten Zweck abhängt. Dies ist sicher nicht überraschend, doch offenbar nur wenigen der Musterautoren wirklich klar geworden.“[Zül98]

Vor allem zwei Zwecke bei der Benutzung von Mustern kann man unterscheiden: Entweder es sollen neue Exemplare von Mustern erzeugt werden oder es sollen vorhandene Ausprägungen von Mustern erkannt werden.

Falls Muster angewendet werden sollen, so sollte die Musterbeschreibung „konstruktiv“ sein: Der Leser muß mit Hilfe der Problembeschreibung erkennen können, wann es möglich und sinnvoll ist, ein Muster umzusetzen. Des weiteren soll beschrieben werden, wie diese Umsetzung dann vollzogen werden kann, z.B. durch eine schrittweise Anleitung.

Falls aber existierende Exemplare eines Muster erkannt werden, ist eine Anleitung nicht hilfreich. Vielmehr müssen besondere Merkmale des umgesetzten Musters beschrieben werden, um seine Ausprägungen auffindbar zu machen. Solche Beschreibungen bezeichnet man als „deskriptiv“ [Zül98].

Eine Musterbeschreibung muß aber auch berücksichtigen, falls beide Zwecke zusammenfallen: So werden Entwurfsmuster verwendet, um ein Entwurfsproblem zu lösen; gleichermaßen werden sie aber auch zum Verständnis eines vorhandenen Entwurfes benutzt.

2.1.4. Mustersprachen

Eine besondere Eigenschaft von Mustern ist deren Synergieeffekt: Sie ergänzen und verstärken sich gegenseitig, so daß sie in einem Verbund noch stärkere Auswirkungen haben.

Christopher Alexander geht so weit und behauptet, daß man architektonische Entwürfe vollständig nur mit Mustern beschreiben kann [Ale79, S.

309 ff.]. Die Menge an Mustern, die man dabei verwendet, bezeichnet er als Sprache. Er beschreibt selber in „Eine Mustersprache“[AIS95] *eine* solche Sprache. Jeder Mensch hat aber seine eigene Mustersprache und auch für jedes Projekt gibt es eine eigene Sprache, die die jeweiligen Erfordernisse aufgreift. Des weiteren sind solche Sprachen auch nicht fix, sondern entwickeln sich fort [Ale79, S. 337 ff. + 432 ff.].

In einer Sprache sind Muster unterschiedlicher Größe miteinander verwoben: Ein Muster hängt von den kleineren Mustern ab, die es enthält, und auch von den größeren Mustern, in die es selber eingebettet ist [Ale79, S. 312].

Bei der Übernahme von Mustern in die Softwaretechnik wurden auch hier viele Zusammenstellungen von Mustern gesammelt, jeweils für ein spezielles Gebiet. Diese wurden aber nur selten als Sprachen bezeichnet, oftmals wurden andere Bezeichnungen verwendet: z.B. Katalog [GHJV94], System [BMR+96], Handbuch [RZ96] oder Sammlung [Zül98].

2.2. Refactorings

2.2.1. Was sind Refactorings?

Refactorings sind unter Softwareentwicklern weitgehend bekannt, zum einen durch das Buch „Refactoring: Improving the Design of Existing Code“[Fow99] von Martin Fowler, zum anderen dadurch, daß Refactorings in agilen Softwareentwicklungsmethoden – allen voran Extreme Programming – meist eine fundamentaler Bestandteil sind.

Dort wird ein Refactoring definiert als:

> “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.“

Diese Definition besteht aus vier Teilen, wobei „a change“ und „without changing its observable behavior“ besonders wichtig sind:

a change ... Refactorings sind eine deutliche Abkehr von Ansichten wie „Never change a running system“, wie sie viele Programmierer lange gepflegt haben, da sie befürchteten, durch eine Veränderung des Quelltextes einen Fehler in diesem zu erzeugen. Refactorings sind aber nichts anderes als eine reine Veränderung und vor dieser soll deswegen nicht mehr zurückgeschreckt werden; wenn es einen Grund für sie gibt, ist auch die Bereitschaft zur Veränderung vorhanden.

Die Offenheit gegenüber Veränderungen wird auch dadurch unterstrichen, daß das Manifest zu Extreme Programming[Bec00] im Untertitel „Embrace Change“ heißt. In den Prinzipien hinter dem Manifest für agile Softwareentwicklung[Agi] heißt es ebenfalls, daß Veränderungen zum Vorteil des Kunden ergriffen werden sollen.

... made to the internal structure of software ... Dieser Definitionsteil ist eigentlich unnötig; woran die Veränderungen vorgenommen werden, ergibt sich aus dem vierten Teil, welcher eine schärfere Forderung darstellt.

Wenn man etwas anderes verändert als die interne Struktur, etwa die Benutzeroberfläche, verändert man auch das Verhalten der Software, da die Änderung ja eben nicht intern war. Trotzdem hilft einem dieser Satzteil bei dem Verständnis, Refactorings gegenüber anderen Änderungen am Quelltext abzugrenzen.

... to make it easier to understand and cheaper to modify ... Diese Rechtfertigung für Refactorings ist ebenfalls fragwürdig. Es stimmt zwar, daß dies zwei gewichtige Gründe für Refactorings sind. In der Praxis hat sich aber gezeigt, daß Refactorings von Softwareentwicklern als ein Werkzeug zu vielerlei Zwecken angewendet werden, ganz individuell zur situativen Problemlösung.

In [Bec00, S. 179] wird die Notwendigkeit für Refactoring ebenfalls allgemeiner gesehen. Dort heißt es, daß man mit einem Refactoring eine nichtfunktionale Qualität verbessert, z.B. Schlichtheit, Flexibilität, Verständlichkeit oder Performanz.

Es ist aber wichtig, daß in einer Definition von Refactorings angegeben wird, daß diese *überhaupt* auf einen Nutzen hin ausgerichtet sind und nicht nur zum Selbstzweck durchgeführt werden. Dagegen definiert Don Roberts in seiner Doktorarbeit Refactorings so technisch und allgemein, daß jegliche Änderungen an einem Programm darunter fallen könnten, egal was sie beinhalten und warum sie ausgeführt werden. [Rob99, S. 25]

... without changing its observable behavior. Dies ist wohl der wichtigste Teil der Definition. Refactorings sind eben nicht eine *beliebige* Veränderung an Software. Nach ihrer Durchführung zeigt die Software wieder das selbe Verhalten wie zuvor. Dies bedeutet inbesondere auch, daß durch das Refactoring keine Fehler in die Software gelangt sind.

In dieser Beibehaltung des Verhaltens liegt gleichermaßen die Besonderheit von Refactorings, als auch eine starke Ungenauigkeit:

Diese Eigenschaft macht Refactorings besonders, da sonst Programme nur bearbeitet werden um ihre Eigenschaften zu verändern: Meist um die Funktionalität zu erweitern oder Fehler zu entfernen. Deswegen wurde zunächst der Bedarf für Refactorings auch oft nicht erkannt. Wie in dem dritten Teil der Definition beschrieben wurde, gibt es aber vielfältige Gründe für Refactorings.

Dieser Teil der Definition ist unscharf, da das „observable behavior" nicht klar abgrenzbar ist. Streng genommen ändern fast alle Refactorings das Verhalten von Software, da sich schon geringe Änderungen minimal auf die Laufzeiten auswirken und sich dies mit speziellen Werkzeug beobachten läßt. In den meisten Fällen wird dies aber keine relevante Größenordnung annehmen, so daß *scheinbar* das Verhalten gleich bleibt. Bei einer Echtzeitanwendung kann aber schon eine geringe Änderung der Laufzeiten einen Fehler darstellen.

Dies zeigt, daß das „beobachtbare Verhalten" richtig ausgelegt werden muß. Hierbei ist besonders die Perspektive wichtig, denn bei „beobachtbar" stellt sich die Frage: Von wem beobachtbar? Meist ist hier die Perspektive eines Endanwenders gemeint, der nicht bemerkt, wenn sich der Name einer Methode verändert. Wenn es sich bei dem Produkt aber um ein Rahmenwerk handelt, so werden Softwareentwickler, welche dieses Rahmenwerk verwenden, sehr wohl diese Veränderung bemerken, falls die Methode öffentlich sichtbar ist.

Mit der Perspektive geht auch die Relevanz einher: Wenn sich die Laufzeit eines Programmes um eine Millisekunde verändert, so ist dies für Endanwender meist nicht relevant. Deswegen beachten auch viele Refactorings Änderungen der Laufzeiten nicht.

Wenn sich in einem Rahmenwerk ein Klassenname ändert, so mag dies für die meisten Programmierer, die dieses Rahmenwerk verwenden, ebenfalls nicht relevant sein, wenn die Klasse nur innerhalb des Rahmenwerk verwendet werden soll. Ein Programmierer, der dieser Richtlinie nicht folgt und doch von außerhalb auf diese Klasse zugreift, wird durch die Änderung dann doch betroffen. Auswirkungen auf solchen Mißbrauch des Rahmenwerkes wird dessen Hersteller aber meist nicht als relevant erachten.

Die mehrfachen Bedeutungen des Wortes „Refactoring"

Leider wird das Wort „Refactoring" mit unterschiedlichen Bedeutungen verwendet. Als Beispiel betrachte man diese drei Sätze:

- „‚Rename Method' ist ein sehr häufig angewandtes Refactoring."
- „Wir mußten das Refactoring rückgängig machen, da wir nicht nachvollziehbare Fehler gemacht hatten."
- „Refactoring gehört heute zum gängigen Handwerkszeug eines Programmierers."

In allen drei Sätzen wird das Wort mit einer anderen Bedeutung verwendet: Im ersten Satz bezeichnet „Refactoring" ein Schema, welches auf Software angewendet werden kann, um diese umzugestalten. Im zweiten Satz bezeichnet „Refactoring" genau die konkrete Anwendung eines solchen Schemas. Im dritten Satz wiederum, bezeichnet „Refactoring" die allgemeine Disziplin, solche Schemata zu kennen und mit ihnen umzugehen.

Diese mehrfache Bedeutung ist unglücklich. Sie mag deswegen weit verbreitet sein, weil sich in vielen Fällen die exakte Auslegung des Begriffes aus dem Kontext ergibt oder nicht relevant ist.

Für diese Arbeit gilt, daß mit „Refactoring" die erste Bedeutung gemeint ist. Die zweite wird als eine „Refactoring-Ausführung" bezeichnet oder – weil dies wahrhaft ein Ungetüm ist – umschrieben mit der „Ausführung eines Refactorings". Die dritte Bedeutung wird in dieser Arbeit nicht verwendet.

2.2.2. Abgrenzung zwischen Refactorings und Restrukturierungen

Die Definition zu Refactorings paßt eigentlich auch auf beliebige Restrukturierungen [CC90]. Refactorings sind aber nicht eine beliebige Art, Quelltext zu restrukturieren, sondern eine sehr spezielle Technik [Fow04].

Refactorings sind so ausgelegt, daß man mit ihnen größtmögliche Sicherheit erhält, durch die Durchführung nicht das bestehende Verhalten der Software zu verändern. Die Mittel mit denen dies erreicht wird, sind vielfältig: Sie reichen von entsprechend gut konzipierten Refactorings (z.B. mit möglichst kleinen Schritten), über den Einsatz von Tests, welche einem als Anzeiger für ein korrektes Verhalten der Software dienen, bis hin zu vollautomatischen Refactoring-Werkzeugen, die durch Analyse des Quelltextes sicherstellen, daß ein Refactoring ausgeführt werden kann, ohne daß sich das Verhalten der Software ändert.

2.2.3. Was sind *keine* Refactorings?

Es ist wichtig abzugrenzen, was *kein* Refactoring ist, insbesondere da dies schon zu einem Buzzword geworden ist. [Orr00]

Einige Softwareentwickler bezeichnen jede Restrukturierung als Refactorings, auch wenn gar nicht die Kriterien zutreffen, welche Refactorings von beliebigen Restrukturierungen abgrenzen. Andere gehen sogar so frei mit dem Begriff um, daß fast jedwede Änderung an Software als Refactoring bezeichnet wird, sobald es dabei nicht das vorrangige Ziel ist, sie um neue Funktionalität zu erweitern, obwohl einige grundlegende Merkmale von Refactorings verletzt werden:

- Es wird nicht in kleinen Schritten vorgegangen.
- Das System wird nicht ständig in einem korrekt lauffähigem Zustand gehalten.
- Parallel zu der Restrukturierung werden womöglich auch noch kleinere Änderungen vorgenommen, die das Verhalten der Software verändern.

Auch wenn diese mitunter so genannt werden, sind es *keine* Refactorings. [Fow04]

2.3. Bisherige Arbeiten zu kleinen Refactoring-Mustern

In diesem Abschnitt wird vorgestellt, wie diese Arbeit zu anderen Forschungen über kleine Refactorings und Refactoring-Muster in Bezug steht:

William F. Opdyke

In seiner Doktorarbeit zählt William Opdyke „Low-Level Refactorings“ auf, wovon sich einige in dieser Arbeit auch als kleine Refactoring-Muster wiederfinden. Die kleinen Refactorings werden von ihm aber nur benutzt, um über Refactorings schreiben zu können, die größer sind als diese. Es wird nicht erläutert, was die einzelnen low-level Refactorings darüber hinaus für ein Zweck haben und wie sie genutzt werden kann. Er schreibt selber, daß er sie nur der Vollständigkeit halber so ausführlich darstellt, was von den meisten Lesern aber übersprungen werden kann [Opd92, S. 22 f. und S. 38 ff.].

Mel Ó Cinnéide

Mel Ó Cinnéide führt „Primitive Refactorings“ ein, die in Art und Absicht stark an Opdykes low-level Refactorings erinnern: Wie jene sind sie auch mehr ein Hilfsmittel für die logisch-mathematischen Schlüsse, wenn er die primitive Refactorings kombiniert. Ó Cinnéide versucht deswegen auch nicht solche Refactorings zu finden, die sich als nützlich erweisen können, sondern beschreibt nur solche, die in der Arbeit notwendig wurden.

Des weiteren führt Ó Cinnéide Minitransformationen ein, welche er zu größeren Transformationen zusammensetzt. Diese Minitransformationen sind bereits deutlich größer als die kleinen Refactoring-Muster, wie sie in dieser Arbeit vorgestellt werden. Das Schema – normale Transformationen aus kleineren Bausteinen zusammenzufügen – ist ähnlich wie in dieser Arbeit. Auch hier gilt jedoch wieder, daß Ó Cinnéide nicht untersucht, wie Softwareentwickler durch das Wissen um diese kleineren Bausteine profitieren können. Sein Fokus liegt auf dem Größeren [Cin01, S. 33 ff. und S. 71 ff.].

Kent Beck

In „Test-Driven Development: By Example“ beschreibt Kent Beck in einem Kapitel unterschiedliche Refactorings, wie sie in testgetriebener Entwicklung auftreten [Bec02, S. 181 ff.]. Er bezeichnet diese Refactorings ausdrücklich als Muster, auch wenn er dabei zu ihrer Beschreibung nur reinen Fließtext benutzt und keine Schablonen, wie sie sonst bei Mustern oft üblich sind.

Die von ihm beschriebenen Refactoring-Muster haben unterschiedliche Größen, sind nach meiner Einteilung aber eher mittlere Refactoring-Muster.

C2 WikiWikiWeb

In dem C2 Wiki, in welchem auch frühzeitig über Muster oder agile Softwareentwicklung diskutiert wurde, gibt es eine Seite zu Refactoring-Mustern [C2 b]. Dort gibt es aber nicht mehr als vage Ideen, welche bisher eine inhomogene Sammlung aus Hinweisen zum guten Vorgehen bei Refactorings ist. Über deren Größe sollte man einige der dortigen Muster auch besser mit den Refactorings von Martin Fowler vergleichen.

Reengineering-Muster

In dem FAMOOS-Projekt [FAM, DDN03] und bei Forschungen zu „Systems Reengineering Patterns“ an der University of Edinburgh [Sys, DLPS99] hat man die Verbindung zwischen Reengineering, Refactoring und Mustern gezogen. Dort wurde jedoch nicht das Vorgehen innerhalb einzelner Refactorings betrachtet. Vielmehr wurde dort nach Mustern in dem übergeordneten Reengineering-Prozeß gesucht, in welchem dann Refactorings als kleinere Bestandteile eingebettet sein können.

Lifecycle and Refactoring Patterns

Nachdem Opdyke in seiner Doktorarbeit Refactorings noch nicht als Muster betrachtet hatte, geht er diesen Schritt in dem Papier „Lifecycle and Refactoring Patterns That Support Evolution and Reuse“ gemeinsam mit Brian Foote [FO95].

Zusätzlich wird hier im Entwicklungs- und Refactorings-Prozeß eine geschichtete Sammlung von Mustern gesehen. Die Muster in den oberen Schichten entsprechen dem was andere – wie zuvor beschrieben – als Reengineering-Muster bezeichnen, die in der untersten Schicht werden als Refactoring-Muster bezeichnet.

Damit wird in meiner Arbeit eine Schicht mit Mustern aufgezeigt, die noch unmittelbar unter denen von Foote und Opdyke vorgeschlagenen Schichten liegt.

Kapitel 3.

Kleine Refactoring-Muster

In diesem Kapitel wird das Konzept kleiner Refactoring-Muster dargestellt.

Hierzu wird in dem ersten Abschnitt aufgezeigt, wann es gerechtfertigt ist, Refactorings als Muster zu betrachten. Insbesondere wird dargelegt, wieso auch die von Fowler vorgestellten Refactorings Muster sind.

Anschließend wird betrachtet, wie diese Muster in unterschiedlichen Größen auftreten, wonach dann kleine Refactoring-Muster – als Bestandteile von Refactorings – von größeren Refactorings-Muster abgegrenzt werden. Dies mündet in einer eigenständigen Definition von kleinen Refactoring-Mustern.

3.1. Refactorings als Muster

In seiner Doktorarbeit bezeichnete William Opdyke Refactorings nicht als Muster und er präsentierte sie auch nicht in einer Form, die an Muster erinnern [Opd92]. Zwei Jahre später auf der ersten PLoP jedoch präsentierte er mit Brian Foote zwei Refactorings als Muster [FO95].

Auch bei anderen Autoren finden sich immer wieder Beispiele, daß Refactorings als Muster aufgefaßt werden (z.B. in [Fra02, S. 6]). Insbesondere hat auch Kent Beck in [Bec02] eine Sammlung von Refactorings präsentiert und diese als Muster bezeichnet. Dies hilft aber noch nicht bei der Entscheidung, ob diese Titulierung legitim ist, da – wie auch in 2.2.3 erwähnt – Begriffe mitunter benutzt werden, selbst wenn dies durch ihre Bedeutung gar nicht angebracht ist.

Deswegen muß genauer untersucht werden, ob die Refactorings aus [Fow99] einen Mustercharakter haben. Erleichtert wird dies durch die Form, in welcher Fowler seine Refactorings präsentiert. Er folgt einer festen Schablone, welches bereits ein schwaches Indiz für ihren Charakter ist. Diese Form erleichtert es, die Grundbestandteile eines Musters zu erkennen:

Zunächst gibt Fowler jedem Refactoring einen *Namen*, um es identifizieren zu können. Jedes Refactoring ist auf ein *Problem* ausgerichtet, welches er im Abschnitt „Motivation“ erläutert. Die *Lösung* für dieses Problem wird in dem Abschnitt „Mechanics“ durch eine schrittweise Anleitung beschrieben, welche gleich eine konstruktive Beschreibungsform erkennen läßt.

Die Refactorings sind so beschrieben, daß sie *allgemein anwendbar* sind, da sie sich bei passendem Kontext auf unterschiedliche Situationen anwenden lassen. Dies wird von vielen in der Mustergemeinde aber oftmals noch nicht als ausreichend anerkannt. Es gilt als überzeugender, wenn man aufzeigen kann, daß ein Muster tatsächlich schon von unterschiedlichen Personen in unterschiedlichen Kontexten angewandt *wurde*. Fowler geht darauf nicht direkt ein, da dies nicht der Hauptaugenmerk seines Buches ist. Er beschreibt aber, daß seine Refactorings durchaus retrospektiv entwickelt wurden, indem sie aus den Notizen hervorgegangen sind, wie er sie sich beim täglichen Arbeiten mit Refactorings entwickelt hat. Des weiteren wurden diese Refactorings auch in dem C3 Projekt bei Chrysler durchgeführt [Fow99, S. xxi und 106]. Die Verbreitung dieser Refactorings zeigt sich auch darin, daß sie in zahlreichen Arbeiten wiederholt erklärt oder erwähnt werden, angefangen bei [Opd92].

Im Gegensatz zu Entwurfsmustern wären Refactoring-Muster keine Muster von Artefakten, sondern Muster von Vorgängen. Wie jedoch schon im vorherigen Kapitel erläutert, ist dies kein Grund, einige Refactorings nicht als Muster zu betrachten: Es gibt von unterschiedlichen Dingen Muster, nicht nur von Artefakten, sondern z.B. auch von Vorgängen.

Abschließend ist es auch ein Hinweis, daß Fowler eine ganze Sammlung an Refactorings beschreibt, deren einzelne Elemente sich kombinieren lassen, z.B. um größere Refactorings durchzuführen.

Somit finden sich sämtliche Eigenschaften von Mustern auch in Refactorings wieder – so wie sie von Fowler vorgestellt wurden. Dies erlaubt die folgende, schlichte Definition für Refactoring-Muster:

> *Definition:* **Refactoring-Muster**
> Ein Refactoring, welches den Charakter von einem Muster besitzt und als solches beschrieben wird.

Auch wenn man alle Refactorings von Fowler als Muster betrachten kann, wird hiermit *nicht* behauptet, daß jedes Refactoring auch ein Refactoring-Muster ist: Insbesondere stellen sowohl Beck und Fowler [Fow99, S. 359], als auch Lippert [Lip03] fest, daß sich große Refactorings meist individuell deutlich voneinander unterscheiden. Sie müssen also oft zu sehr in ihrem jeweiligen Kontext betrachtet werden, was eine allgemein Beschreibung verhindert. Große Refactorings werden vermutlich nur selten Refactoring-Muster sein.

3.2. Refactoring-Muster unterschiedlicher Größe

Muster treten oftmals in unterschiedlichen Größen auf: Wie in 2.1 beschrieben wurde, findet man dies sowohl in den Mustern der Architektur als auch in den Mustern von Softwarestrukturen (Programmiermuster, Entwurfsmuster, Architekturmuster) wieder.

Eine solche Hierarchie entdeckt man auch bei Refactoring-Mustern: Refactoring-Muster mittlerer Größe lassen sich kombinieren zu großen Refactorings – auch wenn diese meist keinen Mustercharakter besitzen. Bei der Untersuchung von mittleren Refactorings stellt man aber auch fest, daß diese ihrerseits wieder aus kleineren Refactoring-Mustern aufgebaut sind.

Im weiteren werden Refactoring-Muster, die eine ähnliche Größe besitzen wie jene, die von Fowler beschrieben werden, als mittlere Refactoring-Muster bezeichnet. Große und kleine Refactoring-Muster ergeben sich entsprechend, wobei eine klarere Abgrenzung nötig ist. Für große Refactorings bietet [Lip04] ein Unterscheidungskriterium, für kleine Refactoring-Muster wird dieses im nächsten Abschnitt gefunden.

„Größer“ und „kleiner“ ist natürlich jeweils relativ zu sehen: „Größer“ als kleine Refactoring-Muster sind sowohl mittlere, als auch große Refactoring-Muster.

3.3. Kleine Refactoring-Muster

Bisher wurde bereits angedeutet, daß sich kleine Refactoring-Muster in den Bestandteilen größerer Refactorings finden und einsetzen lassen. Eine klare Abgrenzung und Definition fehlt aber bisher noch.

In diesem Abschnitt werden kleine mit größeren Refactoring-Muster verglichen und voneinander abgegrenzt, um damit zu einer Definition zu gelangen.

Um die Erklärungen in diesem Abschnitt nicht rein abstrakt zu gestalten, wird hier in Tabelle 3.1 bereits eine Übersicht über kleine Refactoring-Muster vorangestellt. Der Katalog der Muster mit ausführlichen Beschreibungen folgt nach der Definition in Kapitel 4.

- *Teile und herrsche* – Spalte eine umfangreicheres Refactoring in sichere Teilschritte auf.
- *Weiterleitung einführen* – Führe eine Weiterleitung ein, so daß die Funktionalität einer Stelle auch an einer anderen zugreifbar ist.
- *Bearbeite alle* – Suche alle Stellen, auf die ein Kriterium zutrifft, und bearbeite sie gleichartig.
- *Finde Schlüsselelement* – Finde eine Element, welches zentral für ein Refactoring ist und dessen Ausführung leitet.
- *Kopieren* – Kopiere einen Teil des Quelltextes um ihn sicher von einer alten an eine neue Stelle zu bewegen.
- *Struktur aufbauen* – Erzeuge neue Strukturen, die für die Durchführung oder Erfüllung eines Refactoring nötig sind.
- *Aufräumen* – Entferne unnötig gewordene Strukturen.
- *Deprecate* – Unterstütze die Durchführung eines Refactorings, indem der Compiler betroffene Codestellen markiert.

Tabelle 3.1.: Zusammenfassung der kleinen Refactoring-Muster

Ausführungsdauer

Kleine Refactoring-Muster von mittleren und großen abzugrenzen, indem man die Zeit für ihre Ausführung betrachtet, ist problematisch. Schon bei Fowlers Refactorings ist die Zeit, die für ihre Ausführung benötigt wird, sehr unterschiedlich:

- Ein „Rename Method“ läßt sich – insbesondere mit geeigneter Werkzeugunterstützung – vielleicht innerhalb weniger Sekunden ausführen.
- Wenn einem aber kein gut geeignetes Werkzeug zur Verfügung steht und man in einem großen System eine Methode umbenennen will, die an vielen Stellen benutzt wird, so kann dies mehrere Stunden in Anspruch nehmen.
- Wenn man dieses Refactoring an einem System durchführen will, bei dem es die Methode mit demselben Namen an vielen anderen Klassen auch gibt (z.B. toString()), in welchen sie nicht umbenannt werden soll, kann das Refactoring sogar mehrere Tage in Anspruch nehmen, da man an vielen Stellen genau untersuchen muß, von welchem Typ die Objekte sind, an denen diese Methode gerufen wird.

Einem bestimmten Refactoring wohnt also keine feste Ausführungsdauer inne.

Beibehaltung des Verhaltens

Natürlich teilen kleine Refactoring-Muster sämtliche Eigenschaften von Refactorings mit ihren größeren Pendants. Besonders sei hier nochmals hervorgehoben, daß sich auch durch die Anwendung eines kleinen Refactoring-Muster das beobachtbare Verhalten eines Programmes nicht verändern darf.

Zweck

Fowler nennt in seiner Definition Gründe zur Ausführung von Refactorings („to make it easier to understand and cheaper to modify“[Fow99, S. 53]), andere fassen deren Zwecke weiter (z.B. [Bec02, S. 181]). In jedem Fall liegt aber in dieser Motivation das wichtigste Unterscheidungskriterium zwischen kleinen und mittleren Refactoring-Mustern:

Mittlere Refactoring-Muster werden nicht um ihrer selbst willen eingesetzt, sondern um einen Zweck zu erfüllen, der außerhalb der Refactorings liegt: So kann z.B. ein „Extract Method“ durchgeführt werden um den Quelltext verständlicher zu machen, indem eine Gruppe von Anweisungen einen Namen erhält.

Kleine Refactoring-Muster dagegen haben ihren Zweck *innerhalb* eines Refactorings: So wird durch das Muster „Teile und Herrsche“ ein Refactoring in kleinere Schritte zerteilt, welche *die Ausführung des Refactorings selber sicherer machen*. Durch die Anwendung dieses Musters wird kein äußeres Ziel angestrebt, wie z.B. die Struktur des Programmes in geeigneter Weise zu verbessern.

Da kleine Refactoring-Muster ihren Zweck immer nur innerhalb eines Refactorings besitzen, werden sie meist nur in Kombination mit anderen kleinen Refactoring-Mustern verwendet. Dort in der Gemeinschaft wird dann ein Ziel erreicht, welches außerhalb der Möglichkeiten jedes einzelnen kleinen Refactoring-Musters liegt. Dies verdeutlicht die Synergieeffekte der Muster: Durch ihre Kombination wird etwas erreicht, was durch die einzelnen Teile alleine nicht erreichbar war.

Als Beispiel kann hier „Kopieren“ dienen: Wenn man mit diesem Muster Quelltext von einer Stelle zu einer anderen kopiert, entsteht duplizierter Code, was von vielen Softwareentwicklern als ein schlechtes Design angesehen wird (siehe [Fow99, S. 76]). Wenn man aber direkt im Anschluß mit „Weiterleitung einführen“ den Quelltext an der alten Stelle durch eine Weiterleitung auf die neue Stelle ersetzt, so wird hierdurch ein Design geschaffen, daß man Funktionalität an eine neue Stelle verschoben hat, ohne duplizierten Code zu besitzen. Da die Funktionalität durch die Weiterleitung auch an der alten Stelle noch zugreifbar ist, können andere Stelle, an denen diese Funktionalität genutzt wird, schrittweise von der alten auf die neue Stelle umgestellt werden, und so dem Muster „Teile und herrsche“ zu folgen: Diese drei Muster können also einfach kombiniert werden, um gemeinsam einen größeren Nutzen zu erzielen, als dies durch ein Muster alleine möglich wäre.

Kombination von Refactorings

Refactorings – auch Refactoring-Muster – werden oft kombiniert, um ein umfassenderes Refactoring durchzuführen. Diese Beziehung sei so bezeichnet:

> *Definition:* **Eingebettete und umgebende Refactorings**
> Wenn mehrere Refactorings zu einem größeren Refactoring kombiniert werden, so sind sie in dieses umgebende Refactoring eingebettet.

Wie im vorherigen Abschnitt erklärt, treten kleine Refactoring-Muster stets nur in Kombinationen auf, um so für einen Zweck verwendet zu werden, welchen sie einzeln nicht erfüllen könnten. Mit dieser neuen Definition läßt sich dies auch so umschreiben: Kleine Refactorings treten nur eingebettet in ein umgebendes Refactoring auf.

Definition

Zusammengenommen führen die Ergebnisse der letzten Abschnitte zu dieser Definition:

> *Definition:* **Kleines Refactoring-Muster**
> Ein Refactoring-Muster, welches in Refactorings eingebettet verwendet wird, um die dort üblichen Probleme zu lösen.

3.4. Zusammenfassung

In diesem Abschnitt wurden die zentralen Konzepte dieser Arbeit beschrieben. Für *Refactoring-Muster* wurde vor allem auf bekanntes Wissen zurückgegriffen und dieses zusammengetragen. Zur Unterstützung wurden *eingebettete und umgebende Refactorings* beschrieben. Für *kleine Refactoring-Muster* mußte ein eigenes Konzept geschaffen werden. Letzteres beruht auf den Erfahrungen, wie sie mit Refactorings gemacht wurden und wie sie Refactoring-Autoren niedergeschrieben haben. Im folgenden Kapitel wird gezeigt, in welchen konkreten kleinen Refactoring-Mustern sich diese Erfahrung widerspiegelt.

Kapitel 4.

Katalog kleiner Refactoring-Muster

Für den Katalog kleiner Refactoring-Muster wird eine Schablone mit den folgenden Abschnitten verwendet. Wenn man in einem Refactoring mit einem Problem konfrontiert ist, hilft diese einheitliche Struktur zu erkennnen, ob es dafür ein passendes kleines Refactoring-Muster gibt, um dann die dort beschriebene Lösung anwenden zu können.

Name
: Der Name des Musters beschreibt seinen Kern kurz und bündig. Wie im Abschnitt 2.1 erläutert, ist es eine wichtige Aufgabe, einem Muster einen treffenden Namen zu geben.

Zusammenfassung
: Eine kurze Zusammenfassung, die einem einen Überblick über das Muster verschaffen soll, wann und wie es eingesetzt wird.

Problem
: Für welches Problem bietet das Muster eine Lösung an?

Kontext
: In welchem Kontext tritt das Problem üblicherweise auf und wo wird das Muster meist angewandt?

Wie in [BMR+96, S. 9] beschrieben, ist es oft schwierig, den richtigen Kontext zu finden. Versucht man alle möglichen Situationen zu erfassen, in denen ein Muster auftritt, wird der Kontext so vage beschrieben, daß er nichts mehr aussagt. Wird man zu spezifisch, klammert man vielleicht zu viele Situationen aus. Deswegen werden hier die Kontexte beschrieben, in welchen das Muster beobachtet wurde, ohne dabei auszuschließen, daß es auch noch in anderen auftritt.

Wie sich aus ihrer Definition bereits ergibt, treten kleinen Refactoring-Muster nur eingebettet in andere Refactorings auf. Dies wird deswegen nicht ausdrücklich bei jedem Muster nochmals erwähnt.

Lösung

Wie geht man bei der Lösung des Problems vor, wenn man das Muster anwendet?

Anwendung des Musters

Abschließend folgen Hinweise für die Anwendung des Musters und Bezüge zu anderen kleinen Refactoring-Mustern. Dies beinhaltet auch bekannte Auftreten der Muster, wobei hier nur wenige beispielhaft genannt werden. Man findet jedes dieser Muster in vielen Refactorings wieder.

In den Beschreibungen der Muster wird oft auf das Buch „Refactoring: Improving the Design of Existing Code“[Fow99] von Martin Fowler verwiesen, da dies der bekannteste Katalog von Refactorings ist.

Teile und herrsche

Name

Teile und herrsche

Zusammenfassung

Ein Refactoring wird beherrschbar gemacht, indem man es in kleine Schritte unterteilt. Wenn jeder Schritt das Verhalten der Software nicht ändert, so ist man sich sicher, daß auch das gesamte Refactoring das Verhalten nicht verändert.

Problem

Es ist gleichermaßen verführerisch, wie auch problematisch bei Refactorings in großen Schritten vorzugehen.

Verführerisch erscheint dies vielen Programmierern, weil sie meinen, bei kleinen Schritten – oder durch Refactoring generell – langsamer voranzukommen und unnötige Umwege zu gehen. Auch wenn dies wohl ein Trugschluß ist, ist dieses Gefühl noch immer weit verbreitet [Fow99, S. 57] und [Ker04, Abschnitt „Small Steps“ auf S. 16 f.].

Große Schritte können in Refactorings zu unterschiedlichen Problemen führen. Beispiele hierfür sind:

- Es wird ein Fehler bei der Durchführung eines Schrittes gemacht. Je größer der Schritt war, desto weniger kann man eingrenzen, wo der Fehler gemacht wurde.
- Während eines Schrittes kann man den Quelltext nicht – oder nur deutlich erschwert – mit anderen Programmierern abgleichen, da man kein korrekt lauffähiges Programm hat. Ein Abgleich sollte aber regelmäßig stattfinden. In agilen Methoden zur Softwareentwicklung wird vorgeschlagen, daß dies mindestens täglich geschieht. Falls Refactorings so lange andauern, daß sie sich zeitlich über einen Abgleich des Quelltext hinaus erstrecken, führt dies zu mannigfaltigen Problemen, welche von [Lip04] detaillierter beschrieben sind.

Kontext

Die Notwendigkeit, eine sichere Unterteilung für ein Refactoring zu finden, tritt insbesondere in drei Zusammenhängen auf:

- Man plant ein neues Refactoring, für welches noch kein Vorgehen bekannt ist und will dieses beherrschbar machen, indem man dafür einen Refactoring-Plan erstellt [Lip04].
- Man will nachträglich einen sicheren Weg für ein Refactoring festhalten, welches man kennengelernt hat. In dieser Art hat Martin Fowler die Abschnitte „Vorgehen" (engl. *Mechanics*) zu seinen Refactorings erstellt [Fow99].
- Man hat ein Refactoring begonnen, ist dabei aber auf hartnäckige Probleme gestoßen, da man in zu großen Schritten vorangegangen ist. Deswegen macht man den problematischen Schritt rückgängig und unterteilt diesen in kleinere Teilschritte [Fow99, S. 104].

Lösung

Spalte das Refactoring in kleinere Schritte auf, so daß sich nach jedem kleinen Schritt das Verhalten der Software nicht verändert hat.

Wenn man nun jeden kleinen Schritt sicher ausführen kann, so hat man dadurch auch das gesamte Refactoring sicher ausgeführt.

Anwendung des Musters

Die Aufteilung von Refactorings in kleine Schritte mag einem schon als grundlegendes Prinzip erscheinen. Tatsächlich ist es aber ein Muster, welches nur gezielt angewendet werden braucht. Fowler beschreibt, daß er die Vorgehen in seinem Buch deswegen so fein aufgeteilt hat, um einen sicheren Weg zu betonen, auf dem dieses Refactoring erreicht werden kann. Während der normalen Arbeit geht auch er oft in größeren Schritten vor und greift auf die kleineren Schritte zurück, wenn die gröbere Unterteilung zu Problemen führt [Fow99, S. 104].

Falls man Werkzeuge verwendet, welche Refactorings zuverlässig und automatisiert durchführen, so ist eine Unterteilung dieser Refactorings in Schritte ebenfalls nicht nötig, da die Sicherheit durch andere Mittel – hier ein Werkzeug – gewährleistet wird.

Die einzelnen Schritte können sowohl unterschiedlich (zunächst Kopieren von Quelltext, anschließend Einführen einer Weiterleitung), als auch gleichartig (zunächst Anpassen einiger Aufrufe eine Methode, anschließend Anpassen der restlichen Aufrufe) sein. Für Aufteilungen in gleichartige Schritte kann einem das Muster „Bearbeite alle“ helfen.

Als geeignete Schritte für ein Refactoring können die anderen kleinen Refactoring-Muster verwendet werden. Eine gute Aufteilung in Schritte bleibt jedoch eine Aufgabe, welche Erfahrung verlangt, was durch das Studium bekannter Vorgehen (etwas aus [Fow99]) unterstützt werden kann.

Die Durchführung von „Teile und herrsche“ wird unterstützt, wenn man Mittel besitzt, mit denen man leicht überprüfen kann, ob sich das Verhalten der Software geändert hat. Hierzu werden oft Komponententests (engl. *unit tests*) eingesetzt [Lin02].

Weiterleitung einführen

Name

Weiterleitung einführen

Zusammenfassung

Es wird eine Weiterleitung eingeführt, so daß sämtliche Aufrufe an einer Stelle einfach weitergeleitet werden.

Problem

Funktionalität, welche bereits an einer Stelle des Programmes vorhanden ist, wird an einer anderen Stelle benötigt. Man will den Quelltext nicht einfach von der einen zur anderen Stelle kopieren, da hierdurch duplizierter Code entsteht, was oftmals nicht erwünscht ist [Fow99, S. 76].

Kontext

Weiterleitungen werden häufig während eines Refactorings vorübergehend eingeführt, um das Aufteilen eines Refactorings zu ermöglichen. Wenn z.B. eine Methode verschoben werden soll, so kann man zunächst an der neuen Stelle eine Weiterleitung einrichten und die Aufrufe dieser Methode allmählich von der alten Methode auf die neue umstellen, ohne daß die Software dadurch in ihrem Verhalten beeinträchtigt wird.

Eine Weiterleitung kann aber auch dazu dienen, vorhandene Funktionalität dauerhaft an einer neuen Stelle zugreifbar zu machen. Den Bedarf dafür stellt man vielleicht erst während des Refactorings fest und läßt die Weiterleitung dann auch nach dem Refactoring noch bestehen.

Lösung

Verweise an der Zielstelle mit einer Weiterleitung auf die Ursprungsstelle.

Nachdem eine Weiterleitung ihren Dienst während des Refactorings erfüllt hat, sollte man prüfen ob sie noch benötigt wird oder wieder entfernt werden kann.

Anwendung des Musters

Dieses Muster wird für ähnliche Probleme angewandt wie „Kopieren“. Mitunter können die beiden Muster aber auch nacheinander ausgeführt werden, indem zunächst Quelltext kopiert und anschließend das Original durch eine Weiterleitung auf die Kopie ersetzt wird. Martin Fowler nutzt dies in „Rename Method“, wo er zunächst den Rumpf aus der alten in die neue Methode kopiert, um anschließend den duplizierten Code in der alten Methode durch eine Weiterleitung auf die neue Methode zu ersetzen [Fow99, S. 274 f.].

Manchmal hat man bereits duplizierten Quelltext und löst diesen Mißstand, indem man den Code an der einen Stelle durch eine Weiterleitung an die andere Stelle ersetzt. Das Muster, zunächst zwei Teile des

Quelltextes so zu verändern, daß sie identisch sind, wird von Kent Beck als „Reconcile Differences“ bezeichnet [Bec02, S. 181].

Tammo Freese demonstriert, wie man Weiterleitungen einfach wieder entfernen kann, indem man später das Refactoring „Inline Method“ durchführt [Fre03].

Man bezeichnet Weiterleitungen auch oft als Delegationen oder Indirektionen. Dies sind jedoch beides speziellere, technische Konzepte, die verwendet werden können, um eine Weiterleitung zu realisieren. Deswegen soll im Kontext dieser Refactoring-Muster die Bezeichnung „Weiterleitung“ beibehalten werden.

Bearbeite alle

Name

Bearbeite alle

Zusammenfassung

In einem Refactoring erfordert eine Änderungen, daß viele, verteilte Stellen des Quelltextes an diese Änderung angepaßt werden.

Problem

Es ist ein hehres Ziel der objektorientierten Programmierung, das eine Veränderung möglichst nur wenige andere, lokal begrenzte Änderungen nach sich zieht.

Leider ist dies oftmals nicht möglich und gerade einige Refactorings führen dazu, daß Veränderungen durchgeführt werden müssen, die über weite Teile des Quelltextes verteilt sind. Wenn man z.B. den Namen einer oft verwendeten Klasse ändert, muß man diesen Name auch überall anpassen, wo diese Klasse benutzt wird.

Kontext

Dieses Muster wird im Laufe eines Refactorings oft angewandt, wenn sich das Refactoring auf viele getrennte Codeteile auswirkt.

Häufig wurde vor der Anwendung dieses Musters eine passende Struktur geschaffen, so daß das „Bearbeite alle“ nicht vollständig durchgeführt werden muß, sondern es möglich ist, daß die Stellen in beliebiger Reihenfolge und auch nur teilweise bearbeitet werden.

Lösung

Bestimme ein Kriterium (z.B. „ruft die Methode setStatus() an einem Exemplar der Klasse Step.“), anhand dessen möglicherweise betroffenen Stellen im Quelltext gefunden werden können. Suche mit diesem Kriterium, untersuche jede Fundstelle und bearbeite diese gegebenenfalls entsprechend.

Anwendung des Musters

Auch dieses Muster findet sich wieder in vielen Refactorings, so wie sie von Fowler beschrieben wurden. So muß in dem Refactoring „Encapsulate Field“ nach jedem Zugriff auf das betroffene Attribut gesucht werden und durch den Aufruf einer Methode ersetzt werden, welche das Attribut ausliest oder setzt [Fow99, S.206 f.].

Oftmals wird für die Suche nach allen betroffenen Stellen ein Schlüsselelement verwendet (z.B. eine Methode, die für das Refactoring zentral ist), um dann mit einem Werkzeug alle Stellen zu suchen, an der dieses Schlüsselelement verwendet wird (z.B. die Methode aufgerufen oder überschrieben). Die Suche nach einem solchen zentralen Element ist in dem Muster „Finde Schlüsselelement“ beschrieben.

Manchmal kann dieses Muster auch so verwendet werden, daß zunächst nur ein Teil der betroffenen Stellen bearbeitet wird und weitere erst später. Damit wird das Muster „Teile und herrsche“ unterstützt. Eine solche Aufteilung ist z.B. in dem Refactoring „Move Method“ meist möglich.

Es kann bei der Ausführung dieses Musters sehr aufwendig sein, alle betroffenen Stellen zu finden. Fowler weist hierfür darauf hin, sich bei der Suche durch Werkzeuge oder den Compiler unterstützen zu lassen [Fow99, S. 105 f.]. Die Unterstützung durch den Compiler wird in dem Muster „Deprecate“ beschrieben.

Bei dem Einsatz dieses Refactorings ist besonders zu beachten, daß sich eventuell nicht alle betroffenen Stellen nur im ausführbaren Quelltext wiederfinden. Refactorings können sich auch auf andere Dinge auswirken, wie z.B. Dokumentation, Makefiles oder Plugin-Beschreibungs-Dateien.

Finde Schlüsselelement

Name

Finde Schlüsselelement

Zusammenfassung

Finde ein Element, welches zentral für ein Refactoring ist. Dieses Element wird im weiteren Refactoring eine Schlüsselposition und Orientierungshilfe sein.

Problem

Refactorings erfordern mitunter Änderungen an vielen, verteilten Stellen des Quelltextes (siehe Muster „Bearbeite alle“) und man läuft dadurch Gefahr, sich in diesen weit verstreuten Stellen zu verlieren.

Kontext

Oft kommt dieses Muster schon mit dem Start eines Refactorings zum Einsatz, da man einen Mißstand entdeckt hat (auch als „Code Smell“ bezeichnet [Fow99, S. 75 ff.]) und dieser eine Schlüsselposition darstellt. Ansonsten wird das Muster aber auch früh im umgebenden Refactoring angewandt, damit das Schlüsselelement im weiteren Verlauf seine Rolle als Orientierungshilfe ausüben kann.

Lösung

Finde ein Element, welches für das weitere Refactoring bestimmend ist und als Orientierungshilfe benutzt werden kann, z.B. indem man Verweise auf dieses Element für das Muster „Bearbeite alle“ verwendet.

Anwendung des Musters

Ein Fall, in dem sich das Schlüsselelement schon aus dem „Code Smell“ ergibt, ist das Refactoring „Remove Middle Man“. Den „Middle Man“ hat man als unerwünscht erkannt und geht von seinen delegierenden Methoden aus, um jede Benutzung dieser zu finden und durch einen Aufruf ohne Delegation zu ersetzen [Fow99, S. 85 u. 160 f.].

Manchmal muß man ein Schlüsselelement zunächst mittels des Musters „Struktur aufbauen“ erzeugen, um sich anschließend daran orientieren zu können. Joshua Kerievsky demonstriert dies in dem Refactoring

„Replace Conditional Calculations with Strategy“. Dort wird zu Beginn eine neue Klasse erzeugt, in welche anschließend der betroffene Quelltext verschoben wird und dann Ausgangsbasis für die Bildung von Subklassen ist [Ker04].

Kopieren

Name

Kopieren

Zusammenfassung

Während der normalen Softwareentwicklung wird duplizierter Quelltext argwöhnisch betrachtet, weswegen man es auch vermeiden soll, Quelltext zu kopieren. Im Verlauf eines Refactorings kann dies aber gezielt als ein Mittel eingesetzt werden, um eine schrittweise Migration zu ermöglichen.

Problem

Quelltext ist bereits an einer Stelle vorhanden, wird aber an einer anderen Stelle benötigt. Es ist nicht wünschenswert oder möglich, daß an der einen Stelle einfach eine Weiterleitung auf die andere Stelle eingeführt wird.

Kontext

Dieses Muster tritt oft inmitten eines Refactorings auf, wenn man Quelltext von einer Stelle an eine andere bewegen will, es aber ein zu großer Schritt wäre, diesen zu verschieben.

Ein anderer Grund für das Kopieren von Quelltext ist der Aufbau einer Schattenstruktur, bei der Elemente (vorübergehend) doppelt vorhanden sind, ohne aufeinander zu verweisen.

Lösung

Deswegen: Akzeptiere vorübergehend das schlechtere Design, da man sich in einem laufenden Refactoring befindet und kopiere den Quelltext.

Sorge dafür, daß später im laufenden, umgebenden Refactoring der duplizierte Quelltext wieder entfernt wird.

Anwendung des Musters

Dieses Muster wird für ähnliche Probleme verwendet, wie „Weiterleitung einführen“. Im praktischen Einsatz wird jedoch oft schon durch den Kontext deutlich, welches Muster zur Anwendung kommt: Ist es das Ziel, den Quelltext an die neue Stelle zu verschieben oder soll er an der neuen Stelle anschließend verändert werden, so ist das Muster „Kopieren“ passender. Soll dagegen an beiden Stellen auf die selben Daten zugegriffen werden oder vielleicht die an mehreren Stellen aufrufbare Funktionalität auch länger Bestand haben, so ist das Muster „Weiterleitung einführen“ besser.

Fowler nennt „Duplicate Code“ als ersten „Code Smell“, trotzdem wird in vielen seiner Refactorings Quelltext kopiert. So wird in „Extract Method“ der zu extrahierende Teil des Quelltextes in eine neue Methode kopiert. Dadurch kann die neue Methode erst vollständig eingerichtet werden kann (unter Beachtung lokaler Variablen und Parametern), während das Programm weiterhin mit seinem bisherigen Verhalten lauffähig bleibt, bevor an der Ursprungsstelle der Teil des Quelltextes durch einen Methodenaufruf ersetzt wird. Wenn man den Teil unmittelbar verschieben und durch einen Aufruf der neuen Methode ersetzen würde, wäre das Programm während dieser Schritte vermutlich noch nicht einmal lauffähig [Fow99, S. 76 u. 111].

Für das Refactoring „Push Down Method“ wird eine Methode zunächst in jede Subklasse kopiert, um das Programm lauffähig zu halten. Anschließend kann jede Kopie einzeln untersucht werden, ob sie noch angepaßt werden muß oder nicht mehr benötigt wird und entfernt werden kann [Fow99, S. 328].

Es ist oftmals verlockend, Quelltext nicht zu *kopieren* sondern zu *verschieben*. Das Verschieben bringt aber so viele Probleme mit sich, daß man es in Anlehnung an [BMMM98] schon fast als „Anti-Refactoring-Muster“ bezeichnen kann und nur im kleinen Rahmen anwenden sollte:

Anti-Refactoring-Muster: Verschieben

Etwas zu verschieben ist ein riskanter Schritt, da etwas plötzlich an einer Stelle fehlt, wo es zuvor noch vorhanden war. Vermutlich wurde es an der alten Stelle auch noch verwendet, so daß das Programm fehlerhaft ist, bis man alles auf die neue Stelle angepaßt hat. Macht man diese Anpassungen vorher, ist das Programm ebenfalls während der Übergangszeit fehlerhaft, da an der neuen Stelle der Quelltext noch nicht bereitsteht. Viel einfacher ist es, zunächst zu kopieren, dadurch ein weiterhin lauffähiges Programm erhalten, die Umstellung auf die neue Stelle vollziehen und abschließend die überflüssig gewordene Stelle entfernen.

Als konkretes Beispiel kann man hierzu das Refactoring „Extract Method“ betrachten: Man verschiebt oder kopiert den Quelltext von der ursprünglichen Stelle in eine neue Methode und fügt an der ursprünglichen Stelle einen Aufruf der neuen Methode ein. Wenn das Erstellen der extrahierten Methode aufwendiger wird als erwartet, so ist das Programm währenddessen trotzdem noch korrekt, wenn der Quelltext nur kopiert wurde; wenn er aber verschoben wurde, fehlt das Verhalten an der alten Stelle und das Programm ist fehlerhaft, bis das Refactoring abgeschlossen werden konnte.

Da man durch Verschieben meist einen Fehler provoziert, muß man diesen unmittelbar korrigieren, oft durch eine Weiterleitung. Wenn man nur kopiert ist man flexibler und kann oftmals in kleineren Schritten vorgehen. Somit unterstützt „Kopieren“ das Muster „Teile und herrsche“.

Struktur aufbauen

Name

Struktur aufbauen

Zusammenfassung

Erzeuge neue Strukturen frühzeitig und lasse Dich im Refactoring von ihnen leiten.

Problem

Für die Bewältigung eines Refactorings sind zusätzliche Strukturen (z.B. Klassen, Methoden oder Attribute) notwendig. Dadurch soll aber das bestehende Verhalten des Programmes nicht beeinflußt werden.

Kontext

Neue Strukturen werden meist frühzeitig in einem Refactoring aufgebaut, sie können dann noch als Orientierungshilfen dienen und so das Muster „Finde Schlüsselelement" unterstützen.

Lösung

Erzeuge die neuen Strukturen. Beachte dabei, daß es zu keinen Auswirkungen auf das bereits vorhandene Programm kommt.

Anwendung des Musters

Dieses Muster ist zunächst etwas unscheinbar, selbst wenn es in vielen Refactorings eingesetzt wird. So wird in „Introduce Parameter Object" zu Beginn eine neue Klasse eingeführt, welche für das weitere Refactoring einen Fixpunkt bildet, so wie in dem Muster „Finde Schlüsselelement" beschrieben. Die Klasse löst im restlichen Programm eine typische Parameterkombination ab und an sie kann abschließend noch Methoden angelagert werden, welche typisch für diese Kombination sind [Fow99, S. 295 f.].

Die genaue Untersuchung, ob ein neues Element keine Auswirkungen hat, sprengt hier den Rahmen. Beispielhaft werden nur vier, typische Fälle gezeigt: zwei für die Einführung einer neuen Methode, zwei für eine neue Klasse. Diese Fälle sind für die Programmiersprache Java sicher, soweit man Reflection außer Acht läßt.

Es ist sicher, eine neue Methode einzuführen, falls

- es in der Vererbungshierarchie keine andere Methode mit der selben Signatur gibt. Die neue Methode kann hierbei einen beliebigen Inhalt haben.
- es zwar schon Methoden mit der selben Signatur in der Vererbungshierarchie gibt, diese Methoden unmittelbar ober- und unterhalb in der Hierarchie aber alle den selben Methodenrumpf besitzen und die neue Methode ebenfalls diesen Methodenrumpf erhält.

Es ist sicher, eine neue Klasse einzuführen, falls

- diese nur unten an die Klassenhierarchie angehängt wird, also keine andere Klasse von ihr erbt. Die neue Klasse kann dabei einen beliebigen Inhalt haben.
- die Klasse keinerlei Elemente enthält und die Klasse so in eine Klassenhierarchie eingefügt wird, daß andere Klassen von ihr erben.

Aufräumen

Name

Aufräumen

Zusammenfassung

Eine nicht mehr benötigte Struktur wird entfernt.

Problem

Während eines Refactorings werden Strukturen überflüssig. Würden diese bestehen bleiben, können sich nach dem Refactoring Programmierern unsicher sein, ob und wozu diese Strukturen noch nötig sind.

Kontext

Dieses Muster wird vor allem gegen Ende von Refactorings angewendet.

Lösung

Untersuche, ob das Element wirklich entfernt werden kann, ohne daß Verhalten des Programmes zu verändern. Falls dies der Fall ist, entferne das Element.

Anwendung des Musters

Ebenso wie „Struktur aufbauen“ ist dies ein einfaches aber doch wichtiges Muster. Insbesondere da Refactorings häufig auf Schlichtheit und Verständlichkeit hin zielen, ist es wichtig, nach Abschluß eines Refactorings keine unnötigen Strukturen zu hinterlassen.

Um herauszufinden, ob etwas noch benutzt wird, kann man das Muster „Deprecate“ benutzen. Dies kann auch verwendet werden, um eine bevorstehende Löschung anzukündigen.

In „Collapse Hierarchy“ werden aus einer Klasse sämtliche Methoden und Parameter entfernt, damit die Klasse am Ende ohne Probleme entfernt werden kann. Das Refactoring zielt also genau darauf ab, daß am Ende dieses Muster angewendet werden kann [Fow99, S. 344].

Deprecate

Name

Deprecate

Zusammenfassung

Dieses Muster unterstützt andere, indem ein Element so markiert wird, daß beim Compilieren nur Warnungen, aber noch keine Fehler erzeugt werden.

Problem

Wenn man ein überflüssiges Element löscht, obwohl es noch benutzt wird, führt dies zu Fehlern.

Kontext

Das Problem tritt besonders dann auf, wenn man nicht den gesamten Quelltext unter Kontrolle hat und sich das Refactoring über eine längere Zeit hinzieht.

Lösung

Benutze einen Mechanismus der Programmiersprache, damit jede Benutzung des Elements eine Warnung erzeugt.

Anwendung des Musters

In Java kann man zur Kennzeichnung der Elemente die @deprecate-Marken verwenden, woher das Muster auch seinen Namen erhalten hat. In C++ kann man für die Kennzeichnung Pragmas benutzen und in [DLPS99] wird berichtet, daß dieses Muster auch in Emacs Lisp eingesetzt wird.

Wenn der Compiler für dieses Muster keine Unterstützung anbietet, kann man auch noch externe Werkzeuge benutzen, welche Kommentare nach vereinbarten Marken absuchen. Diese Werkzeuge sind aber meistens nicht so weitgehend und bietet damit eine schlechtere Lösung, als dies bei einer Unterstützung durch den Compiler der Fall wäre [Fow99, S. 105 f.].

Problematisch ist, wenn man einen Deprecate-Mechanismus für mehrere Zwecke benutzt: etwa innerhalb eines eigenen Refactorings, während gleichzeitig das laufende große Refactoring eines anderen Programmierers die gleichen Marken hinterläßt und man auch noch die gleichen Warnungen durch ein benutztes Rahmenwerk bekommt. In solchen Fällen sollte man die Marken so beschriften, daß die Absicht einer Marke ersichtlich wird oder sogar eindeutig zuordnungsbare IDs verwendet werden.

Neben dem oben beschriebenen Kontext kann man dieses Muster auch als Hilfsmittel für „Bearbeite alle“ benutzen, selbst wenn das Element am Ende gar nicht entfernt wird. Hier gilt aber noch mehr das eben erwähnte Problem, daß unterschiedliche Nutzungen des Deprecate-Mechanismus sich gegenseitig behindern können.

Kapitel 5.

Anwendung kleiner Refactoring-Muster

In diesem Kapitel wird anhand des Refactorings „Replace Array with Object“ gezeigt, wie sich kleine Refactoring-Muster in Refactorings wiederfinden und sie zu deren Beschreibung verwendet werden können.

Anschließend wird mit dem selben Beispiel demonstriert, wie mit kleinen Refactoring-Muster ein bekanntes Refactoring angepaßt werden kann, falls dieses in einer Ausführung nötig wird.

5.1. Beschreibung von Refactorings

Martin Fowler beschreibt das Vorgehen für „Replace Array with Object“ folgendermaßen [Fow99, S. 186 f.]:

- Create a new class to represent the information in the array. Give it a public field for the array.
- Change all users of the array to use the new class.
- Compile and test.
- One by one, add getters and setters for each element of the array. Name the accessors after the purpose of the array element. Change the clients to use the accessors. Compile and test after each change.
- When all array accesses are replaced by methods, make the array private.

- Compile.
- For each element of the array, create a field in the class and change the accessors to use the field.
- Compile and test after each element is changed.
- When all elements have been replaced with fields, delete the array.

Wenn man dieses Refactoring mit der Hilfe von kleinen Refactoring-Mustern beschreibt, so stellt das zu ersetzende Feld ein Schlüsselelement des Refactorings dar (siehe Muster *Finde Schlüsselelement*). In dem Vorgehen werden die folgenden Muster eingesetzt, wobei die Aufforderungen zum Compilieren vernachlässigt werden:

- *Struktur aufbauen*: Erzeuge die Zielklasse und ein öffentlich sichtbares Feld in ihr.
- *Bearbeite alle*: Finde alle Stellen, in welchen das bisherige Feld verwendet wurde, und stelle sie so um, daß sie stattdessen das Feld in einem Exemplar der neuen Klasse verwenden.
- *Bearbeite alle*: Mache für jedes Element des Feldes zwei Schritte:
  - *Struktur aufbauen*: Erzeuge schreibende und lesende Zugriffsmethoden.
  - *Bearbeite alle*: Suche jede Stelle, an der auf dieses Element des Feldes zugegriffen wird und ändere die Stelle so, daß die Zugriffsmethoden verwendet werden.
- Wenn auf des Feld nicht mehr von außerhalb der Klasse zugegriffen wird, schränke seine Sichtbarkeit entsprechend ein.
- *Bearbeite alle*: Erzeuge für jedes Element des Feldes ein Attribut in der Klasse. Stelle die Zugriffsmethoden von dem Element auf das Attribut um.
- *Aufräumen*: Wenn alle Elemente des Feldes durch Attribute ersetzt wurden, lösche das Feld.

Wie man sehen kann, gibt es in diesem Refactoring nur einen Schritt, der keinem kleinen Refactoring-Muster folgt. Die Vorgehen insgesamt folgt natürlich dem Muster *Teile und herrsche*.

5.2. Anpassung von Refactorings

5.2.1. Eine Refactoring-Ausführung nach Fowler

Fowler demonstriert eine Ausführung dieses Refactorings folgendermaßen:

Das zu entfernende Feld wird deklariert und initialisiert mit

```
String[] row = new String[3];
```

Stellen, an denen dieses Feld benutzt wird, sehen bisher etwa so aus:

```
row [0] = "Liverpool";
row [1] = "15";

String name = row[0];
int wins = Integer.parseInt(row[1]);
```

Für den ersten Schritt erzeugt er eine neue Klasse und in dieser ein Attribut, welches das Feld hält:

```
class Performance {
    public String[] _data = new String[3];
}
```

Im zweiten Schritt ersetzt er das Feld durch ein Exemplar der neuen Klasse:

```
Performance row = new Performance();
```

Und paßt jede Verwendung des bisherigen Feldes so an, daß das Feld in dem neuen Objekt benutzt wird:

```
row._data [0] = "Liverpool";
row._data [1] = "15";

String name = row._data[0];
int wins = Integer.parseInt(row._data[1]);
```

So weit das Refactoring, wie von Martin Fowler beschrieben.

Falls dieser letzte Schritt (die Anpassung aller Stellen auf die Verwendung der neue Klasse) besonders langwierig oder schwierig ist, möchte man dem Muster *Teile und herrsche* folgen und diesen Schritt weiter unterteilen.

So wie das Vorgehen für das Refactoring beschrieben ist, ist dieser Fall aber nicht vorgesehen: Dadurch, daß die Variable `row` bereits auf ein Exemplar der neuen Klasse verweist, stellen alle Orte Compilerfehler dar, an welchen noch davon ausgegangen wird, das die Variable auf ein Feld verweist. Dies bedeutet, daß erst sämtliche Zugriffe auf die Variable umgestellt werden müssen, bevor das Programm wieder lauffähig ist.

5.2.2. Anpassung der Ausführung auf eine besondere Gegebenheit

Möchte man den zweiten Schritt des Refactorings dennoch aufteilen, so kann man dies durch die folgende Anpassung tun:

Anstatt die Variable `row` auf die neue Klasse umzustellen, erzeugt man hilfsweise eine zusätzliche Variable, welche ein Exemplar der neuen Klasse enthält (Muster *Struktur aufbauen*).

```
Performance rowObject = new Performance();
```

Für die Variable `row` wird nun kein Feld mehr erzeugt, stattdessen verweist diese auf das Feld in dem zuvor erzeugten Objekt (Muster *Weiterleitung einführen*).

```
String[] row = rowObject._data;
```

Nun kann – wie zuvor – das Muster *Bearbeite alle* alle angewendet werden, jedoch mit dem Vorteil, daß die einzelnen Stellen unabhängig voneinander und in beliebiger Reihenfolge umgestellt werden können. Das Programm bleibt auch dann mit seinem bisherigen Verhalten lauffähig, wenn bisher nur einige der Stellen angepaßt wurden:

```
rowObject._data [0] = "Liverpool";
rowObject._data [1] = "15";
```

```
String name = row[0];
int wins = Integer.parseInt(row[1]);
```

Nachdem alle Stellen auf die neue Struktur angepaßt wurden, kann man die überflüssige Variable `row` entfernen (Muster *Aufräumen*) und eventuell die Variable `rowObject` umbenennen.

Kapitel 6.

Fazit und Ausblick

6.1. Inhalt der Arbeit

Als zentrales Ergebnis dieser Arbeit ist das Konzept von kleinen Refactoring-Mustern erarbeitet und demonstriert worden.

Die Basis wurde hierzu gelegt, indem die Konzepte von Mustern und Refactorings vorgestellt wurden. Hierfür ist dargestellt worden, was das heutige Verständnis von Mustern in der Softwaretechnik ist. Es wurde erläutert, wie dieses auf den Gedanken von Christopher Alexander beruht, allmählich auf die Softwareentwicklung übertragen wurde und welchen Inhalt es besitzt.

Anschließend wurden gleichermaßen Refactorings vorgestellt und von anderen Konzepten der Restrukturierung von Software abgegrenzt.

Als letzte Grundlage ist untersucht worden, welche Erkenntnisse es bereits in dem Gebiet dieser Arbeit gibt. Dabei wurde festgestellt, daß dieses Thema von anderen Autoren stets nur am Rande behandelt wurde; entweder haben sie es nicht zusammenfassend behandelt oder sogar ganz ausgespart.

In Kapitel 3 wurde untersucht, ob und wann man Refactorings als Muster erachten kann, was insbesondere an den Refactorings von Martin Fowler betrachtet wurde. Hierfür sind vor allem Erkenntnisse anderer zusammengefaßt worden, die bisher nur verstreut vorlagen. Abgeschlossen wurde dies durch eine Definition für Refactoring-Muster.

Anschließend wurden kleine Refactoring-Muster untersucht, welche nur als Bestandteile von Refactorings auftreten. Hier lag zunächst der besondere Augenmerk darauf, wie man diese von größeren Refactoring-Mustern abgrenzen kann. Diese Unterscheidung mündete in einer Definition von kleinen Refactoring-Mustern.

Im vierten Kapitel wurde ein Katalog kleiner Refactoring-Muster vorgestellt, womit die Konzepte aus dem vorherigen Kapitel an konkreten Beispielen festgemacht werden. Dieser Katalog ist nicht vollständig, erfaßt aber die wichtigsten Muster, wie sie sich in den derzeit verbreiteten Vorgehen zu Refactorings finden. Mit diesen Mustern ist es möglich, die Erkenntnisse über kleine Refactoring-Muster umzusetzen.

Diese Anwendung wurde in Kapitel 5 demonstriert. Anhand eines Beispieles ist dort dargestellt worden, wie man mit kleinen Refactoring-Mustern andere Refactorings begreifen, beschreiben und anpassen kann.

6.2. Ergebnisse

In dieser Arbeit wurden die Erkenntnisse über *Refactoring-Muster* zusammengetragen und zu einer Definition verdichtet. Weiter wurde ein Konzept von *kleinen Refactoring-Mustern* erarbeitet und ein *Katalog* von solchen Mustern erstellt. Hierdurch ist es gelungen, ein besseres Verständnis zu gewinnen, wie Refactorings aufgebaut sind und wie sie durchgeführt werden können.

Dieses Wissen kann verwendet werden, wenn man Refactorings beschreiben will: sowohl wenn man rückblickend Refactorings festhält, welche einem bekannt sind, als auch wenn man vorausschauend ein neues Refactoring plant. Wie demonstriert wurde, können die Erkenntnisse aber auch dazu dienen, bekannte Refactorings auf eine Situation anzupassen.

6.3. Ausblick

Die vorliegende Arbeit basiert auf den Erkenntnissen, die bisher mit Refactorings gemacht wurden. Genauere Studien, zu welchen Auswirkungen das bewußte Arbeiten mit kleinen Refactoring-Mustern führt, gehen über diese Arbeit hinaus und verbleiben für die Zukunft.

Wie schon mehrmals beschrieben, erhebt der Katalog von kleinen Refactoring-Mustern keinen Anspruch auf Vollständigkeit. Somit besteht der Bedarf, nach weiteren Mustern zu suchen. Es ist aber auch noch zu erforschen, wann, warum und welche Schritte in Refactorings sich nicht durch Muster erfassen lassen.

Werkzeuge für kleine Refactoring-Muster

Eine bisher noch nicht untersuchte Anwendung kleiner Refactoring-Muster könnte in Werkzeugen liegen, welche die Anwendung dieser Muster vereinfachen.

Zur Unterstützung des Refactoring-Prozesses gibt es bisher nur für ausgewählte Refactorings Werkzeuge, welche diese Refactoring vollständig durchführen [Fow99, S. 401 ff.]. Für kleine Refactoring-Muster könnte man Werkzeuge entwickeln, die auch dann verwendbar sind, wenn ein solcher Automat nicht verfügbar ist. Beispiele hierfür wären:

- Ein Refactoring-Planer, mit dem man eine Aufteilung festhält, die man durch das Muster *Teile und herrsche* gefunden hat. Durch ein solches Werkzeug könnte man einen Refactoring-Plan versioniert speichern, elektronisch weitergeben und mit Verweisen auf den Quelltext versehen. Die Konzepte von Refactoring-Plänen und ihrer möglichen Unterstützung durch Software beschreibt Martin Lippert in [Lip04].
- Das Muster *Bearbeite alle* könnte durch verbesserte Suchwerkzeuge unterstützt werden. Es ist bereits ein Fortschritt, in modernen Entwicklungsumgebungen nicht nur textuell suchen zu können, sondern z.B. auch nach jeder Verwendung einer Klasse.

  Aber schon die Kombination mehrerer Suchen per Schnittmenge oder Vereinigung ist oft nicht möglich. Wünschenswert wären aber noch weitergehende Funktionen: Wenn man eine Suche mit ihren Ergebnissen abspeichern kann, läßt sich ein „Bearbeite alle“ leichter aufteilen, so daß es von unterschiedlichen Programmierern oder in unterschiedlichen Refactoringschritten durchgeführt werden kann. Außerdem sollte man die Suchergebnisse einzeln markieren können (z.B. als noch zu erledigen oder bereits bearbeitet).

Neben der Entwicklung solcher Werkzeuge verbleibt aber auch ihre Evaluation noch für die Zukunft. Wie sie sich in der Praxis einsetzen lassen und welche Auswirkungen sie haben, liegt abermals außerhalb dieser Arbeit.

Anhang A.

Zur Wahl der Wörter

In diesem Anhang werden drei Entscheidungen zur Wortwahl in dieser Arbeit begründet: Die Verwendung des Wortes *Refactoring*, den Verzicht auf Bindestrichen bei Kombination des Wortes *Muster* mit anderen Wörtern und die Wahl der herkömmlichen Rechtschreibung.

A.1. Das Kind beim Namen nennen: Refactoring, Refaktorisierung, Restrukturierung oder ...?

Da *Refactoring* ein englisches Fachwort ist, stellt sich die Frage, wie man damit in einem deutschen Text umgeht. Übernimmt man das englische Wort oder ersetzt es durch ein deutsches? Wenn man ein deutsches Wort wählen möchte, welches nimmt man?

Ich werde in diesem Abschnitt darlegen, welche Überlegungen bei mir dazu geführt haben, den Begriff *Refactoring* direkt aus dem Englischen zu übernehmen.

Refactoring

Refactoring ist ein Kunstwort, vermutlich in den achtziger Jahren entsprungen aus der Smalltalk-Gemeinde und basierend auf dem mathematischen Begriff *to factor* (dt. *faktorisieren*) [Ing81],[Rob99, S. 14]. Der Begriff wurde dann auch von Opdyke in seiner maßgebenden Dissertation benutzt und Fowler hat ihn für den Titel seines grundlegenden Buches verwendet [Opd92], [Fow99].

So ist die Bezeichnung *Refactoring* heutzutage im englischsprachigen Raum weitverbreitet. Im Englischen lautet das Verb dazu *to refactor* und weitere Wörter hierzu können einfach gebildet werden: *refactoring step*, *refactoring tool*. Hierüber besteht keinerlei Diskussionsbedarf.

In Deutschland wurde das Substantiv *Refactoring* von vielen Informatikern und Autoren direkt übernommen. Die englisch gebildete Mehrzahl *Refactorings* fügt sich bereits etwas schwerfälliger ein, wird aber auch noch von vielen verwendet. Wird die Bezeichnung mit einem deutschen Wort kombiniert, so geschieht diese unglückliche Kombination aus zwei Wörtern unterschiedlicher Sprachen meist mit einem Bindestrich: *Refactoring-Schritt*, *Refactoring-Werkzeug*.

Problematisch wird es im Deutschen, wenn man zu *Refactoring* ein Verb bilden möchte. Während man in der Umgangssprache mitunter ein leicht eingedeutschtes *refactorn* hört, umgehen Autoren dieses Wort meist, indem sie Umschreibungen wählen: z.B. *umstrukturieren* oder schlicht *ein Refactoring ausführen*.

Refaktorisierung

In der deutschen Übersetzung von Fowlers Buch taucht im Titel noch das Wort *Refactoring* auf, ansonsten wird aber sehr konsequent das Wort *Refaktorisierung* und dazu passend das Verb *refaktorisieren* benutzt. Weitere Wörter können mit dieser Basis auch leicht gebildet werden: *Refaktorisierungswerkzeuge*. In einer Überschrift wird man nur überrascht, da bei *Faktorisierungsschritt* ohne ersichtlichen Grund die Vorsilbe *Re* fallengelassen wurde [Fow00].

Jenseits der Übersetzung dieses Buches findet der Begriff *Refaktorisierung* aber wenig Verwendung. Andere Autoren benutzen ihn kaum und auch jenseits der Literatur wird er kaum benutzt.

Tatsächlich wirkt der Begriff *Refaktorisierung* holprig. Vielleicht wird er aber auch so wenig benutzt, weil viele meinen, das Wort wäre ein „Falscher Freund“: So als ob man *to become* mit *bekommen* übersetzt oder auch die weitverbreitete, unglückliche Übersetzung von *instance* im objektorientierten Bereich zu *Instanz*. Wolf Schneider schreibt hierzu: „Wir übersetzen nur scheinbar; in Wahrheit äffen wir das englische Original in deutschen Silben nach“[Sch94, S. 64].

Hier trügt der Schein aber; *Refaktorisierung* ist als Wortwahl keineswegs so unglücklich, wie dies zunächst scheint: Wie oben erläutert wurde, ist das englische Ursprungswort *Refactoring* ein künstlicher Begriff, basierend auf *to refactor*. Wenn nun das ursprüngliche Wort künstlich erschaffen wurde, erscheint es nur legitim, für die deutsche Übersetzung ebenfalls ein neues Wort zu erfinden und hierzu die korrekte Übersetzung des Wortstammes zu benutzen. Dies entspricht auch dem Vorschlag von Rechenberg „Mutig neue deutsche Begriffe zu prägen“ [Rec02, S.71].

Restrukturierung

Anstelle des holprigen Ausdrucks *Refaktorisierung* könnte man auch einen etwas größeren Schritt wagen, indem man ein Wort wie *Um-* oder *Restrukurierung* verwendet. Ähnlich wie *Refaktorisierung* kann man dieses Wort einfach in einen deutschen Text einbetten, auch da es leicht möglich ist, ein Verb dazu zu bilden.

Allerdings hat diese Wortwahl zwei gravierende Nachteile: Zum einen wird in der Informatik auch der englische Ausdruck *Restructuring* bereits mit einer leicht anderen Bedeutung verwendet (siehe Abschnitt 2.2.2), so daß ein deutscher Leser hier an ein anderes Konzept erinnert wird; zum anderen hat dieser deutsche Ausdruck kaum Verbreitung im deutschen Sprachraum, so daß ein Text mit ihm für einen Leser schwerer verständlich ist.

Nur Verben wie *umstrukturieren* findet man gelegentlich, wenn Autoren als Substantiv *Refactoring* benutzen und mangels eines deutschen Verbes zu Umschreibungen greifen.

Die Entscheidung für *Refactoring*

Wie man bereits am Titel der Arbeit ablesen kann, habe ich mich als Begriff für *Refactoring* entschieden. Ich habe eine zeitlang überlegt, den Begriff *Refaktorisierung* zu benutzen und zuvor auch *Restrukturierung* probeweise benutzt. Ich merkte aber, wie schwer mir deren Benutzung „durch die Finger ging“, sie „fühlten“ sich verkehrt an.

Somit bin ich – trotz seiner Nachteile – zu dem Ausdruck *Refactoring* zurückgekehrt, insbesondere um diese Arbeit verständlicher zu machen. Denn

Refactoring ist der Begriff, der im deutschen Sprachraum weitgehend benutzt wird. Dies ist naheliegend, denn wie auch Rechenberg über offene Amerikanismen der Informatik schrieb: „[...] wenn man immerzu 'Domain', 'host', 'instance' [...] liest, *denkt* man auch in diesen Wörtern [...]"[Rec02, S.71]

Letztendlich fiel diese Entscheidung also vor allem mit Rücksicht auf die Gewohnheiten des Lesers.

A.2. Mustersprache oder Muster-Sprache?

Bei der Übersetzung des Buches „A Pattern Language"[AIS77] ins Deutsche haben sich die Übersetzer dazu entschieden, den Titel mit „Eine Muster-Sprache" zu übersetzen [AIS95]. Sie haben nicht das Wort *Mustersprache* gewählt, um ein Mißverständnis zu vermeiden: Zum einen kann eine Mustersprache eine Sprache sein, die Muster enthält. Zum anderen könnte eine Mustersprache aber auch eine Sprache sein, welche musterhaft – im Sinne von vorbildlich – ist.

Wenn man bei dem Wort *Muster-Sprache* eine Schreibweise mit Bindestrich verwendet, müßte man dies aber auch bei anderen Wörtern wie *Muster-Beschreibung* tun. Dies führt zu Wortungetümen und schadet der Lesbarkeit.

Deswegen wird in dieser Arbeit bei Kombinationen aus *Muster* und einem deutschen Wort stets die Schreibweise ohne Bindestrich verwendet. Die Bedeutung dieser Wörter ist immer so, daß mit einer *Musterbeschreibung* die Beschreibung eines Musters gemeint ist. Sollte eine Beschreibung als vorbildlich gekennzeichnet werden, geschieht dies durch eine andere Wortwahl.

A.3. Das rechte Schreiben

1996 haben Deutschland, Österreich und die Schweiz durch die Wiener Absichtserklärung zur „Neuregelung der deutschen Rechtschreibung" beschlossen, bis zum 1. 8. 1998 eine überarbeitete Rechtschreibung einzuführen, was spätestens zu diesem Termin auch an allen Schulen geschehen ist.

Ich sehe diese Rechtschreibreform nach wie vor skeptisch, insbesondere die neuen Regeln zur Getrenntschreibung von Wörtern. Der Sprache werden

Ausdrucksmöglichkeiten genommen, wenn man nicht mehr zwischen *wieder sehen* und *wiedersehen* unterscheiden kann. In den letzten Jahren hat sich auch immer mehr eine Beliebigkeit in der Rechtschreibung durchgesetzt, die irgendwo zwischen der herkömmlichen und der neuen Rechtschreibung liegt.

In dieser Arbeit werde ich mich weitgehend an die herkömmliche Rechtschreibung halten und hierzu den Schreibweisen aus dem Wörterbuch von Prof. Theodor Ickler folgen [Ick00].

Literaturverzeichnis

Es ist unerläßlich geworden, auch Quellen aus dem Internet zu verwenden. Dies bleibt aber problematisch, da man keine Aussagen darüber treffen kann, wie lange Adressen gültig sind. Auf den Umgang mit dieser Problematik wird hier in einer kurzen Notiz eingegangen:

Die URLs, die in diesem Literaturverzeichnis angegeben sind, wurden zuletzt am 11.5.2004 überprüft. Von einigen Seiten finden sich Kopien unter `http://datenreisender.de/DA/`, so dies urheberrechtlich möglich ist und bei diesen Seiten zu befürchten ist, daß sie sich schnell ändern könnten. Man kann von vielen Webseiten, selbst wenn diese verändert oder entfernt wurden, ältere Fassungen auch unter `http://web.archive.org/` abrufen.

Wie schnell URLs veralten können, zeigte sich auch im Laufe dieser Arbeit: Eine der Gruppen, die an dem ausgelaufenen FAMOOS-Projekt[FAM] beteiligt war, wurde von einer anderen Firma aufgekauft. Im Zuge dessen wurden deren Webseiten komplett umgestaltet und diejenigen Seiten des FAMOOS-Projektes, welche unter `http://dis.sema.es/projects/FAMOOS/` lagen, sind leider nicht mehr verfügbar.

[Agi] *Principles behind the Agile Manifesto.* – `http://www.agilemanifesto.org/principles.html`

[AIS77] Alexander, Christopher ; Ishikawa, Sara ; Silverstein, Murray: *Center for Environmental Structure Series.* Bd. 2: *A Pattern Language: towns, buildings, construction.* New York : Oxford University Press, 1977

[AIS95] Alexander, Christopher ; Ishikawa, Sara ; Silverstein, Murray: *Eine Muster-Sprache: Städte, Gebäude, Konstruktionen.* Löcker Verlag, 1995. – ISBN 3–85409–179–6

[Ale79] ALEXANDER, Christopher: *Center for Environmental Structure Series*. Bd. 1: *The Timeless Way of Building*. New York : Oxford University Press, 1979. – ISBN 0–19–502402–8

[BC87] BECK, Kent ; CUNNINGHAM, Ward: Using Pattern Languages for Object-Oriented Programs / Computer Research Laboratory, Tektronix, Inc. 1987 (CR-87–43). – Technical Report. `http://c2.com/doc/oopsla87.html`

[Bec00] BECK, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Inc., 2000. – ISBN 0–201–61641–6

[Bec02] BECK, Kent: *Test-Driven Development: By Example*. Addison-Wesley, Boston, 2002. – ISBN 0–321–14653–0

[BMMM98] BROWN, William J. ; MALVEAU, Raphael C. ; MCCORMICK III, Hays W. ; MOWBRAY, Thomas J.: *Anti Patterns: Refactoring Software, Architectures and Projects in Crisis*. John Wiley & Sons, 1998. – ISBN: 0-471-19713-0

[BMR+96] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-oriented software architecture: a system of patterns*. Wiley, Chichester [u.a.], 1996. – ISBN 0–471–95869–7

[C2 a] *History of Patterns*. – `http://c2.com/cgi/wiki?HistoryOfPatterns`

[C2 b] *Refactoring Patterns*. – `http://c2.com/cgi/wiki?RefactoringPatterns`

[CC90] CHIKOFSKY, E. J. ; CROSS, J. H.: Reverse engineering and design recovery - a taxonomy. In: *IEEE Software* 7 (1990), January, Nr. 1, S. 13–17

[Cin01] Ó CINNÉIDE, Mel: *Automated Application of Design Patterns: A Refactoring Approach*, University of Dublin, Trinity College, Diss., 2001. – `http://www.cs.ucd.ie/staff/meloc/home/papers/thesis/thesis.htm`

[Cop95] COPLIEN, James O.: A Generative Development-Process Pattern Language. In: COPLIEN, James O. (Hrsg.) ; SCHMIDT, Douglas C. (Hrsg.): *Pattern Languages of Program Design 1*. Addison Wesley, 1995, S. 183–237 – `ftp://st.cs.uiuc.edu/pub/patterns/plop-papers/ProcessPatterns.ps`

[DDN03] DEMEYER, Serge ; DUCASSE, Stéphane ; NIERSTRASZ, Oscar: *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003. – ISBN 1–558–60639–4

[DLPS99] DEWAR, Rick ; LKOYD, Ashley D. ; POOLEY, Rob ; STEVENS, Perdita: Identifying and communicating expertise in systems reengineering: a patterns approach. In: *IEE Proceedings - Software*, 1999, S. 145–152 – `http://www.dcs.ed.ac.uk/home/pxs/journalPatterns.ps`

[Dud03] DUDENREDAKTION (Hrsg.): *Duden - Deutsches Universalwörterbuch*. 5. Auflage. Dudenverlag, 2003. – ISBN 3–411–05505–7

[FAM] *The FAMOOS Project*. – `http://www.iam.unibe.ch/~famoos/`

[FO95] FOOTE, Brian ; OPDYKE, William F.: Lifecycle and Refactoring Patterns That Support Evolution and Reuse. In: COPLIEN, James O. (Hrsg.) ; SCHMIDT, Douglas C. (Hrsg.): *Pattern Languages of Program Design*. Reading, MA : Addison-Wesley, 1995, S. 239–257 – `http://www.laputan.org/lifecycle/lifecycle.html`

[Fow99] FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999. – ISBN 0–201–48567–2

[Fow00] FOWLER, Martin: *Refactoring: Wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, München, 2000. – ISBN 3–8273–1630–8

[Fow04] FOWLER, Martin: RefactoringMalapropism. (2004). – `http://martinfowler.com/bliki/RefactoringMalapropism.html`

[Fra02] FRAGEMANN, Per: *Refactoring von UML-Modellen*, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, 2002

[Fre03] FREESE, Tammo: Inline Method Considered Helpful: An Approach to Interface Evolution. In: *Extreme Programming and Agile Processes in Software Engineering*, Springer-Verlag Heidelberg, 2003, S. 271–278

[Gam91] GAMMA, Erich: *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*, Universität Zürich, Diss., 1991

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts : Addison Wesley, 1994. – ISBN 0–201–63361–2

[Hil] *Hillside History*. – `http://hillside.net/history.html`

[Ick00] ICKLER, Theodor: *Das Rechtschreibwörterbuch: Sinnvoll schreiben, trennen, Zeichen setzen*. Leibniz-Verlag, St. Goar, 2000. – ISBN 3–931155–14–5

[Ing81] INGALLS, Daniel H. H.: Design Principles Behind Smalltalk. In: *BYTE Magazine* (1981), August. – `http://users.ipa.net/~dwighth/smalltalk/byte_aug81/design_principles_behind_smalltalk.html`

[Ker04] KERIEVSKY, Joshua: *Refactoring to Patterns*. 2004. – Das Buch soll im August 2004 veröffentlicht werden. Ich beziehe mich auf die Vorversion 0.17 des Buches, welche auf der Webseite `http://www.industriallogic.com/xp/refactoring/` zur Verfügung stand.

[Lin02] LINK, Johannes: *Unit Tests mit Java*. dpunkt-Verlag, 2002. – ISBN 3–89864–150–3

[Lip03] LIPPERT, Martin: *Large Refactorings in Agile Software Development"*. 2003. – Position Paper for the Workshop on „Beyond Green-Field Software Development: Strategies for Reengineering and Evolution“ at OOPSLA 2003, `http://www.martinlippert.com/diss/Lippert-PositionPaper-OOPSLA-2003-Workshop.pdf`

[Lip04] LIPPERT, Martin: Towards a Proper Integration of Large Refactorings in Agile Software Development. In: *Proceedings of XP 2004 Conference*, Springer LNCS, 2004

[Opd92] OPDYKE, William F.: *Refactoring Object-oriented Framworks*, University of Illinois at Urbana-Champaign, Diss., 1992

[Orr00] ORR, Ken: Data Refactoring. In: *Agile Project Management Advisory Service* (2000), February, S. 10–12. – `http://www.cutter.com/freestuff/ead0002.pdf`

[Ped] *The Pedagogical Patterns Project.* – `http://www.pedagogicalpatterns.org/`

[Rec02] RECHENBERG, Peter: *Technisches Schreiben: (nicht nur) für Informatiker.* 1. Auflage. Carl Hanser Verlag, München, 2002. – ISBN 3–446–21944–7

[Rob99] ROBERTS, Donald B.: *Practical Analysis for Refactoring*, University of Illinois at Urbana-Champaign, Diss., 1999

[RZ96] RIEHLE, Dirk ; ZÜLLIGHOVEN, Heinz: Understanding and using patterns in software development. In: *Theory and Practice of Object Systems* 2 (1996), Nr. 1, S. 3–13. – `http://www.riehle.org/computer-science/research/1996/tapos-1996-survey.html`.

[Sch94] SCHNEIDER, Wolf: *Deutsch fürs Leben: Was die Schule zu lehren vergaß.* 10. Auflage. Rowohlt, Reinbek bei Hamburg, 1994. – ISBN 3–499–19695–6

[Sys] *Systems Reengineering Patterns.* – `http://reengineering.ed.ac.uk/System%20Reengineering%20Patterns.htm`

[Zül98] ZÜLLIGHOVEN, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz.* dpunkt.verlag, Heidelberg, 1998. – ISBN 3–932588–05–3

Erklärung

Ich versichere, daß ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Desweiteren bin ich mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 25. Mai 2004

Marko Schulz