

Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2016 Beuth Hochschule für Technik Berlin

REF - Refactoring



MARTIN FOWLER

(most active publisher concerning refactoring issues)

Refactoring

Lernziele und Überblick

In dieser Lerneinheit wird das verbessernde Umgestalten von Code, das „Refactoring“ theoretisch und praktisch vorgestellt.

Wir beginnen mit der Historie des Refactorings und beschäftigen uns daher mit Programmcode / Quelltext und dessen Verbesserung. Wichtiger als konkrete Refactorings kennen zu lernen ist es, ein Gefühl dafür zu entwickeln, wann Code verbessert werden kann.

Wenn dieses Gefühl durch viel Lesen und praktische Arbeit entwickelt ist, dann sollte man das Handwerkszeug kennen, welches durch eine moderne Entwicklungsumgebungen zur Verfügung gestellt wird. Ohne dieses können selbst einfache Refactorings (wie z. B. ein Umbenennen von Klassen) zum Albtraum werden.

In einer stark fortgeschrittenen Stufe des Refactorings ist man dann in der Lage, bestehenden Code so zu verbessern, dass Design-Patterns integriert werden können.



Lernziele

- Theorie des Refactorings verstehen
- Bad Code Smell identifizieren können
- Refactorings unter Eclipse oder anderer IDE anwenden
- Den Refactoring Katalog kennen
- Die Grenzen des Refactorings kennen



Zeitbedarf und Umfang

Zum Durcharbeiten dieser Lerneinheit benötigen Sie ca. 120 Minuten sowie 60 Minuten für die Bearbeitung der Übungen.



Film

Webkonferenz zur Lerneinheit REF

Softwaretechnik
REF - Refactoring

INHALT INDEX GLOSSAR HILFE

Lernziele und Überblick

In dieser Lerneinheit wird das verbessernde Umgestalten von Code, das „Refactoring“ theoretisch und praktisch vorgestellt.

Wir beginnen mit der Historie des Refactorings und beschäftigen uns daher mit Programmcode / Quelltext und dessen Verbesserung. Wichtiger als konkrete Refactorings kennen zu lernen ist es, ein Gefühl dafür zu entwickeln, wann Code verbessert werden kann.

Wenn dieses Gefühl durch viel Lesen und praktische Arbeit entwickelt ist, dann sollte man das Handwerkszeug kennen, welches durch eine moderne Entwicklungsumgebungen zur Verfügung gestellt wird. Ohne dieses können selbst einfache Refactorings (wie z. B. ein Umbenennen von Klassen) zum Albtraum werden.

In einer stark fortgeschrittenen Stufe des Refactorings ist man dann in der Lage, bestehenden Code so zu verbessern, dass Design-Patterns integriert werden können.

Lernziele

- Theorie des Refactorings verstehen
- Bad Code Smell identifizieren können
- Refactorings unter Eclipse oder anderer IDE anwenden
- Den Refactoring Katalog kennen
- Die Grenzen des Refactorings kennen

Zeitbedarf und Umfang

Zum Durcharbeiten dieser Lerneinheit benötigen Sie ca. 120 Minuten sowie 60 Minuten für die Bearbeitung der Übungen.

0:00 12:09

© Beuth Hochschule Berlin - Dauer: 12:08 Min. - Streaming Media 19.0 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

1 Einleitung

Der **Begriff** Refactoring wurde zum ersten Mal 1990 in dem vorliegenden Zusammenhang in einem Paper von **PALPH JOHNSON** und **WILLIAM OPDYKE** gebraucht:

Refactoring: An aid in designing application frameworks and evolving object-oriented systems.

 [JO90]

OPDYKE promovierte im Jahre 1992 über das Thema Refactoring.

JOHNSON und **OPDYKE** schrieben damals über eine Software-Refactory, die das Umgestalten von Softwareprogrammen erleichtert. Also das „re-factoring“.

„Factoring“ heißt übersetzt „faktorisieren“ und entstammt der Mathematik (Polynomfaktorisierung) bzw. Wirtschaft, in der es auch als Kauf von Forderungen verstanden wird. Doch damit hat der Begriff in der Softwaretechnik nichts zu tun.



Hinweis

In dieser Lerneinheit wird der Begriff gleich noch genauer definiert, jedoch lässt sich jetzt schon sagen, dass das Refactoring den vorhandenen Programmcode umgestaltet und verbessert, dabei jedoch die Funktionalität unverändert bleibt.

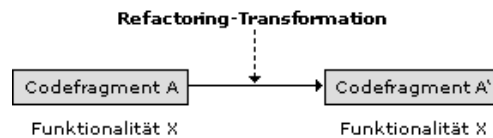




Abb.: Refactoring-Transformation

Wer schon neugierig ist und ein konkretes Refactoring sehen möchte, dem seien die hinteren Kapitel (Refactorings) ans Herz gelegt. Ein kurzes Hin- und Herspringen ist durchaus erwünscht!

2 Literatur

Das Forschungsgebiet des Refactorings ist noch recht jung, daher existiert nicht allzu viel Literatur zu diesem Thema. Jedoch hat diese Thematik (auch dank **MARTIN FOWLER**) einen Schub erfahren, so dass einige qualitativ gute Publikationen vorhanden sind:

Kommentiertes Literaturverzeichnis:	
Fo04  http://www.refactoring.com	Die von FOWLER initiierte Refactoring-Seite ist so etwas wie die Refactoring-Central Seite. Es sind hier eine Unmenge an Informationen und Links zu finden. Am wichtigsten ist die  „Katalog“- Subseite, auf die wir noch mehrmals verweisen werden.
Fo99 MARTIN FOWLER, KENT BECK, JOHN BRANT, WILLIAM OPDYKE, DON ROBERTS (1999): Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, ISBN 0201485672	Ein sehr zu empfehlender Klassiker. Jeder ernsthafte Softwaretechniker oder Softwareentwickler sollte sich auf jeden Fall überlegen, dieses Buch anzuschaffen. Selbst beim Stöbern findet man auch nach Jahren immer wieder gute Hinweise. Dieses Buch ist auch in Deutsch unter der ISBN 3827316308 erhältlich.
Ke04 JOSHUA KERIEVSKY (2004): Refactoring to Patterns, Addison-Wesley Professional, ISBN 0321213351	Ebenfalls ein sehr zu empfehlender Klassiker. Jeder ernsthafte Softwaretechniker oder Softwareentwickler sollte sich auf jeden Fall überlegen, sich dieses Buch anzuschaffen. Selbst beim Stöbern findet man auch nach Jahren immer wieder gute Hinweise.
Wa03 WILLIAM C. WAKE (2003): Refactoring Workbook, Addison-Wesley Professional, ISBN 0321109295	Das Arbeitsbuch zum Thema Refactoring.
BE03 MARTIN BACKSCHAT, STEFAN EDLICH (2003): J2EE-Entwicklung mit Open-Source-Tools, Spektrum Verlag (Elsevier), ISBN 3-8274-1446-6	Dieses Buch enthält im Kapitel 2.4 eine kurze Einführung in das Refactoring.
Ma08 ROBERT C. MARTIN (2008): Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, ISBN-13: 978-0132350884	
RLS06 ROOCK, LIPPERT, STOCKFLETH (2006): Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley & Sons, ISBN-13: 978-0470858929	

Tab.: Kommentiertes
Literaturverzeichnis

3 Definitionen

Mit der Umgestaltung des Programmcodes werden in der Regel wichtige Ziele verfolgt:

- **Lesbarkeit**

Es soll die Lesbarkeit des Quelltextes verbessert werden. Der Programmierer oder auch andere Personen, die sich in den Quelltext hineinarbeiten müssen, sollen den Quelltext besser verstehen. Dies hat zur Folge, dass der Code wartbarer wird und insgesamt schneller agiert werden kann. Änderungen bezüglich der Lesbarkeit betreffen z. B. die Namensgebung oder das Verbergen von Details.

- **Verständlichkeit**

Der Hinweis auf die Lesbarkeit impliziert, dass der Code schneller verständlich wird. Selbst wenn perfekte Variablennamen und Methodennamen vorliegen kann der Code viel zu lang und unorganisiert sein. Verständlichkeit impliziert auch, dass neben der reinen Lesbarkeit des Codes beispielsweise die gesamte Intention des Entwicklers transparent wird.

- **Übersichtlichkeit**

Code der beispielsweise in sinnvolle Codefragmente zergliedert ist und klare und eindeutige Methoden beinhaltet, trägt zur Übersichtlichkeit im Projekt bei.

- **Redundanzen**

Diese können vermieden werden, indem beispielsweise gleiche Codefragmente in nur einer Methode ausgelagert werden. Diese kann auch in die Superklasse verlagert werden. Redundanzfreiheit ist eine wichtige Forderung der effizienten Programmierung, da Änderungen an verschiedenen Codestellen leicht Fehler verursachen.

- **Erweiterbarkeit**

Code der sauber nach den Prinzipien des Refactorings verbessert worden ist, ist leichter erweiterbar. Ist per Refactoring beispielsweise das Strategiepattern in ein Schachprogramm eingebaut worden, so ist der Code sehr leicht um eine neue Schachengine erweiterbar, die anders rechnet.

- **Modularität**

Aus alten Basic-Zeiten ist langer Spaghetticode bekannt. Refactoring setzt auch hier an und versucht, den Code optimal zu modularisieren. Dieses hat auch Auswirkungen auf die Erweiterbarkeit und Testbarkeit.

- **Testbarkeit**

Im Allgemeinen geht es hierbei darum bessere Tests durchführen zu können. Konkret sind dies in der Regel Unit-Tests. Bei diesen Tests spielen natürlich Regressionstests eine wichtige Rolle, da diese beweisen, dass der Code sich danach noch genauso verhält. Es wäre fatal, wenn ein Refactoring in guter Absicht durchgeführt werden würde, sich aber das Verhalten in eine unerwünschte Richtung ändern würde!

Für ein professionelles Refactoring sind objektorientierte Programmierparadigmen sehr wichtig. Viele Refactorings arbeiten mit Patterns, Superklassen und Polymorphie, also alles Konzepte, die ohne eine moderne objektorientierte Programmiersprache nicht denkbar sind.

Es ist weiterhin sehr interessant zu sehen, dass viele Refactorings extrem komplex und änderungsintensiv sein können. Daher ist eine Unterstützung durch IDEs unabdingbar. Moderne Java-IDEs wie IntelliJ IDEA (die im Bereich Refactoring Vorreiter waren und sind), Eclipse oder JBuilder stellen daher einen guten Refactoring-Support zur Verfügung.

Abschließend bleibt festzustellen, dass Refactoring heutzutage ein extrem wichtiger Teil des Redesigns von Software ist. Allzu oft müssen in Projekten Quelltexte umgeschrieben werden. In diesem Fall sollte der Entwickler Erfahrung mit Refactorings haben und entsprechende „Refactoring-Patterns“; gut kennen. Das Refactoring ist oft Teil des Entwicklungszyklus wie beim Test-Driven Development (Kent Beck) oder findet sich sogar in vielen Vorgehensmodellen wieder. Früher war dies Teil eines codefernen „Redesigns“. Heute ist „Refactoring“ ein etabliertes Verfahren für codenahe Verbesserungen.



Achtung

4 Refactoring Theorie

In Zeiten magerer Projektbudgets und Projektmargen ist es wichtig schnell auf Änderungen reagieren zu können. Damit sind sowohl Designänderungen als auch Codeänderungen gemeint, wie beispielsweise neue Features. Wird das Refactoring vom Know-How her und durch die verwendete IDE professionell unterstützt, so sind dies wichtige Wettbewerbsvorteile.

Ein weiteres Ziel des Refactorings ist die Integration von Entwurfsmustern in den Code. Auf dieses Thema wird in den hinteren Kapiteln näher eingegangen. Der Bedarf für den Einsatz von Mustern ergibt sich oftmals erst aus der Entwicklung heraus und kann nicht immer im Design vorhergesehen werden. Der Bedarf für eine **Facade** (Klassenzugriffe zu komplex), ein **Proxy** (hier müssen Aufgaben vorgeschaltet werden) oder **Observer** (hier sind ja noch mehr Klassen an der Zustandsänderung der GUI-Variablen interessiert) entsteht also häufig beim Codieren wenn Architekturänderungen vorgenommen werden sollen. Diese zu integrieren erfordert jedoch eine Menge Erfahrung.



Hinweis

Der Begriff des Refactorings bezieht sich überwiegend auf OO-Code, jedoch nicht immer. Man spricht mittlerweile auch bei der zielgerichteten und wissenschaftlich fundierten Änderung nicht von OO-Code Fragmenten sondern von Refactoring. Prinzipiell kann daher natürlich alles refactored werden (Scriptsprachen, XML-Code, etc.). Die Praxis und die Literatur beziehen sich aber fast immer auf OO-Sprachen.

Die bekanntesten Beispiele in der Literatur, bei denen auch von Refactoring gesprochen wird, oder sogar IDE-Unterstützung vorhanden ist, sind Ant-Dateien und Datenbankschemata.



Abb.:
Auszug der Ant Datei
(links)

Abb.:
Auszug DB Schema
(rechts)

Ant-Buildskripte können groß und unübersichtlich werden (JBoss Version 3 mit vielen Dutzend DIN A4-Seiten) und Datenbankschemata sind oftmals ineffizient. In vielen Projekten bekommen DB-Schemata ein Redesign / Refactoring von einem Profi und die Anwendung ist plötzlich viel schneller.

Weiterhin ist eine Refactoring Unterstützung nicht immer nur mittels einer IDE möglich. Mit JRefactory <http://jrefactory.sourceforge.net> (Sourceforge-Projekt) und RefactorIT www.refactorit.com existieren zwei IDE-unabhängige Java-Tools, die Refactoring unterstützen. Manche lassen sich in IDEs integrieren und bieten interessante Zusatzfeatures an, wie Codeanalyse und Refactoringvorschläge.

Kernpunkt der extrem wichtigen Toolunterstützung ist, dass A) viele Refactorings ohne Tools fast nicht möglich sind und B) die IDE wichtig ist, um sich bei bestimmten Refactorings einfach anzupassen.



Beispiel

Refactoring

A) Eines der simpelsten Refactorings ist das Umbenennen von Namen im Code. Man stelle sich vor, es geht um eine Klasse, die aber im Projekt 42 mal referenziert wird. Handarbeit ist hier unmöglich!

B) Viele Refactorings gestalten Dinge in einer Klasse um. Moderne IDEs denken mit und ändern auch das Interface!

4.1 Risiken und Handhabung

Das Refactoring selbst ist nicht ohne Risiko und daher kritisch zu bewerten. Es können durch ein Refactoring Fehler entstehen. Ein weiteres Problem ist die Reichweite von Refactoring, auf die am Ende nochmal eingegangen wird.

Um Fehler beim Refactoring zu vermeiden, wird in der Literatur oft darauf Wert gelegt, dass Refactorings nur auf fehlerfreien Code angewendet werden. Erst dann liegt ein ordentlicher Zyklus vor: Korrekter Code -> Refactoring -> (immer noch) korrekter Code. Nur so ist man immer auf der sicheren Seite.

Dennoch ist es nicht einsichtig, warum Refactorings nicht auch fehlerhaften Code in korrekten Code transformieren dürfen.

Extrem wichtig ist jedoch, dass Unit-Tests die **Korrektheit** des Refactorings beweisen. Es sollten also eine Reihe von Unit-Tests im Allgemeinen Build-Zyklus (siehe auch Continuous Integration aus der Lerneinheit „Buildmanagement“) integriert sein und diese vor und nach dem Refactoring die Korrektheit sicher stellen.

Ein guter Tipp ist auch, in kleinen Schritten vorzugehen. Das bedeutet, dass auch kleine Änderungen immer per JUnit abzusichern sind. Dadurch entstehen weniger Zerstörung und ein einfacheres Fallback. Komplexe Refactorings, die z. B. das Design oder Patterns betreffen, können auch in kleinere Einheiten zerlegt werden.

Abschließend sei darauf hingewiesen, dass es beim Refactoring wichtig ist, einen sicheren Boden unter den Füßen zu haben! Das bedeutet beispielsweise den Einsatz von **UnDo** und ein **Versionskontrollsystem**. Diese beiden Hilfsmittel geben Sicherheit beim Refactoring.

Viele IDEs können auf jedes Refactoring ein UnDo anwenden (was mitunter eine sehr komplexe Operation sein kann). Kann Ihre IDE das?

Ein Versionskontrollsystem wie CVS oder Subversion ermöglicht, spät erkannte Refactoring-Fehler wieder zu eliminieren. Stellen Sie also sicher, stets mit einem Versionskontrollsystem zu arbeiten. Einen Subversion-Server lokal zu installieren und Subclipse zu installieren dauert keine halbe Stunde.



Achtung




Hinweis

4.2 Der Refactoring „Smell“



Frage von Melanie

Was ist denn eigentlich das Problem, das Refactoring löst?

 Antwort (Siehe Anhang)

Der Refactoring „smell“; ist, wenn folgende Situationen gegeben sind:

- **Duplizierter Code**

Duplizierter Code entsteht meistens durch Copy & Paste und ist daher auch eine der Hauptfehlerquellen bei der Entwicklung. Es gibt daher Tools die nach Copy & Paste Code suchen! Selbst wenn der Code nicht kopiert wurde, ist es in der Regel ein Problem, wenn er mehrfach auftritt. Änderungen in diesem Code müssen dann auch an der anderen Codestelle ausgeführt werden. Das alle Codefragmente immer korrekt editiert werden ist nicht sehr wahrscheinlich.

- **Lange Methoden**

Entwicklern ist das Problem bekannt: Was bisher in Kommentaren in der Methode stand, wird nun ausprogrammiert sodass die Methode länger und länger wird. Und je mehr man schreibt, desto klarer wird, dass ein außenstehender diese Methode immer weniger verstehen könnte. Kann man die Vielzahl der Codefragmente, die man in dieser Methode angeht noch sinnvoller testen?


- **Große Klassen**

Analog gilt dies für große Klassen die zu schwerfällig werden und dadurch die Arbeit behindern. Typische Vertreter sind hier auch GUI-Klassen, die mit ihren Hunderten von nichtssagenden Attributen erfolgreich jedes UML-Diagramm zerstören können.

- **Lange Parameterlisten**

Methoden, die mit Parametern überfüllt sind, hinterlassen ebenfalls einen fragwürdigen Eindruck. Neben dem schlechten Geruch dieser Signatur riecht man aber auch meistens eine mögliche Lösung, wie beispielsweise die Kapselung der Transferdaten in ein Transferobjekt. Zur Lösung derartiger Probleme später mehr.

- **Divergierende Änderungen**


MARTIN FOWLER beschreibt in  [Fo99] ein Phänomen, das entsteht, wenn man Variationen in ein Programm einführt: Man integriert den Zugriff auf eine neue Datenbank oder ein neues Finanzinstrument und muss jedes Mal eine Menge an Methoden in verschiedenen Klassen ändern, was umständlich ist. Mit zunehmender Zeit kommt man zum Schluss, dass es besser wäre, Variationen in nur einer Klasse abzuhandeln, um Änderungen als Parameter behandeln zu können.

- **Neid**

Dieser treffende Begriff beschreibt den Zustand, bei dem zum Beispiel eine Methode X der Klasse A mehr an den Daten M der Klasse B interessiert ist, als an seinen eigenen Daten (der Klasse A). Offenbar ist hier die Aufteilung von Methoden und Attributen nicht sehr glücklich gewählt und bedarf einiger Änderungen.

- **Zuviel Elementare Datentypen**

Es werden zu oft elementare Datentypen verwendet, obwohl die Verwendung von kapselnden Objekten angebrachter wäre.

Weitere mögliche Ursachen für den „Smell“ können Datenklumpen, Switch-Befehle (besser Polymorphie) und falsche Vererbungshierarchien sein. Die hier vorgestellten und noch weitere ausführliche Beispiele finden sie bei FOWLER.  [Fo99]



Beispiel

4.3 Suche und Hinweise auf Refactoring Bedarf

Was tritt bei einem schlechten Design des Codes auf und was kann getan werden, um mehr Hinweise auf Refactorings zu bekommen?

- Wenn immer mehr Zeit damit verbracht wird sich um bereits geschriebenen Code zu kümmern, kann dies ein Hinweis sein, dass dieser nicht gut gestaltet ist und eines Refactorings bedarf.
- In allen etwas umfangreicheren Projekten sollte in regelmäßigen Abständen ein Blick auf das aktuelle UML-Diagramm geworfen werden - am besten regelmäßig ausdrucken und an die Wand kleben! Mit zunehmender Erfahrung fallen spezielle Dinge auf, wie beispielsweise zu volle Klassen oder merkwürdige Vererbungshierarchien.
- Die in den späteren Lerneinheiten beschriebenen **Metriken** zur Code-Analyse wie JDepend liefern deutliche Hinweise auf schlechtes Design. So zeigt JDepend Klassenabhängigkeiten, Zyklen und Maßzahlen für die Abstraktheit des Codes usw., die durch die folgenden Refactoring-Methoden oder auch durch die Einführung von Entwurfsmustern verbessert werden können.

5 Refactorings



Definition

Refactoring-Muster

Kernelement des Refactorings sind so genannte Refactoring-Muster. Diese werden ähnlich wie Entwurfsmuster eingesetzt, um den Code zu verbessern. Ein Refactoring-Muster ist die eingangs beschriebene Transformation des Codes und besteht meistens aus mehreren Arbeitsschritten. Die Arbeitsschritte werden oftmals von der IDE ausgeführt. Der Entwickler hat daher nur noch die Entscheidung zu treffen, auf welche Codefragmente er das Muster anwenden will.

MARTIN FOWLER hat sich die Arbeit gemacht, die Refactorings zu katalogisieren. Dabei wurde er von der Community unterstützt, so dass viele Refactorings von anderen Entwicklern stammen. Dieser Katalog ist unter <http://www.refactoring.com/catalog> zu finden und enthält alle verzeichneten Refactorings.

Viele der dort verzeichneten Refactorings sind bewusst programmiersprachenunabhängig gehalten. Wenn jedoch zur Veranschaulichung Code herangezogen werden muss, dann geschieht dies in einem Java oder C# Dialekt.



Hinweis

Kritischer Hinweis

Ein Katalog hat immer den Anspruch auf Vollständigkeit. Dies bedeutet auch, dass viele Refactorings subjektiv nicht viel Sinn ergeben. Genauso gut kann es sein, dass der geneigte Leser einige Refactorings überhaupt nicht nachvollziehen kann oder jemals einsetzen will. Dies ist völlig in Ordnung.

Es ist wichtig das Spektrum zu kennen, vielleicht machen Sie sich die 10-20 wichtigsten Refactorings eher zu eigen und lassen sich dabei von seiner IDE helfen.

Im Folgenden wird des Öfteren auch auf die IDE-Unterstützung eingegangen und die Refactoring-Fähigkeiten von Eclipse erläutert. Glücklicherweise ist der Katalog der Refactoring Unterstützungen hinreichend abstrakt, so dass sich IDEs wie Eclipse, JBuilder, Netbeans oder IntelliJ IDEA nicht sonderlich groß unterscheiden.

Eclipse 3.1 kennt 31 Refactorings. Hier die Wichtigsten:

- **Rename, Move, Change Method Signature, (Undo, Redo)**
- **Convert Anonymous Class to Nested, Convert Nested Type to Top Level**
- **Push Down, Pull Up**
- **Extract Interface**
- **Use Supertype Where Possible**
- **Inline, Extract Method**
- **Extract Local Variable, Extract Constant**
- **Convert Local Variable to Field, Encapsulate Field**

Da die genannten Refactorings zu den Wichtigsten gehören, ist folgendes Vorgehen empfehlenswert:



















































1. Die Funktionsweise auf den folgenden Seiten oder im [Fowler-Catalog](http://www.refactoring.com/catalog) kurz durchzulesen und zu verstehen.
2. Danach mit der eigenen IDE ausprobieren und das Refactoring-Pattern testen, d. h. mit ihm „warm“ zu werden.














































5.1 Übersicht

Der Übersicht halber sind auf dieser Seite alle 93 Refactorings des Kataloges (Stand Sommer 2005) gelistet. Die nach Meinung des Modulautors besonders wichtigen Refactorings sind hervorgehoben.



93 Refactorings des Fowler-Kataloges

-  [Add Parameter](#)
-  [Change Bidirectional Association to Unidirectional](#)
-  [Change Reference to Value](#)
-  [Change Unidirectional Association to Bidirectional](#)
-  [Change Value to Reference](#)
-  [Collapse Hierarchy](#)
-  [Consolidate Conditional Expression](#)
-  [Consolidate Duplicate Conditional Fragments](#)
-  [Convert Dynamic to Static Construction](#) by GERARD M. DAVISON
-  [Convert Static to Dynamic Construction](#) by GERARD M. DAVISON
-  [Decompose Conditional](#)
-  [Duplicate Observed Data](#)
-  [Eliminate Inter-Entity Bean Communication](#) (Link Only)
-  [Encapsulate Collection](#)
-  [Encapsulate Downcast](#)
-  [Encapsulate Field](#)
-  **[Extract Class](#)**
-  **[Extract Interface](#)**
-  **[Extract Method](#)**
-  [Extract Package](#) by GERARD M. DAVISON
-  [Extract Subclass](#)
-  **[Extract Superclass](#)**
-  [Form Template Method](#)
-  [Hide Delegate](#)
-  [Hide Method](#)
-  [Hide presentation tier-specific details from the business tier](#) (Link Only)
-  **[Inline Class](#)**
-  [Inline Method](#)
-  [Inline Temp](#)
-  [Introduce A Controller](#) (Link Only)
-  **[Introduce Assertion](#)**
-  [Introduce Business Delegate](#) (Link Only)
-  [Introduce Explaining Variable](#)
-  [Introduce Foreign Method](#)
-  [Introduce Local Extension](#)
-  [Introduce Null Object](#)
-  [Introduce Parameter Object](#)
-  [Introduce Synchronizer Token](#) (Link Only)
-  [Localize Disparate Logic](#) (Link Only)
-  [Merge Session Beans](#) (Link Only)
-  [Move Business Logic to Session](#) (Link Only)
-  [Move Class](#) by Gerard M. Davison
-  **[Move Field](#)**
-  **[Move Method](#)**
-  **[Parameterize Method](#)**
-  [Preserve Whole Object](#)
-  [Pull Up Constructor Body](#)
-  **[Pull Up Field](#)**
-  **[Pull Up Method](#)**
-  [Push Down Field](#)

	<u>Push Down Method</u>
	<u>Reduce Scope of Variable</u> by MATS HENRICSON
	<u>Refactor Architecture by Tiers</u> (Link Only)
	<u>Remove Assignments to Parameters</u>
	<u>Remove Control Flag</u>
	<u>Remove Double Negative</u> by ASHLEY FRIEZE and MARTIN FOWLER
	<u>Remove Middle Man</u>
	<u>Remove Parameter</u>
	<u>Remove Setting Method</u>
	<u>Rename Method</u>
	<u>Replace Array with Object</u>
	<u>Replace Assignment with Initialization</u> by MATS HENRICSON
	<u>Replace Conditional with Polymorphism</u>
	<u>Replace Conditional with Visitor</u> by IVAN MITROVIC
	<u>Replace Constructor with Factory Method</u>
	<u>Replace Data Value with Object</u>
	<u>Replace Delegation with Inheritance</u>
	<u>Replace Error Code with Exception</u>
	<u>Replace Exception with Test</u>
	<u>Replace Inheritance with Delegation</u>
	<u>Replace Iteration with Recursion</u> by DAVE WHIPP
	<u>Replace Magic Number with Symbolic Constant</u>
	<u>Replace Method with Method Object</u>
	<u>Replace Nested Conditional with Guard Clauses</u>
	<u>Replace Parameter with Explicit Methods</u>
	<u>Replace Parameter with Method</u>
	<u>Replace Record with Data Class</u>
	<u>Replace Recursion with Iteration</u> by IVAN MITROVIC
	<u>Replace Static Variable with Parameter</u> by MARIAN VITTEK
	<u>Replace Subclass with Fields</u>
	<u>Replace Temp with Query</u>
	<u>Replace Type Code with Class</u>
	<u>Replace Type Code with State/Strategy</u>
	<u>Replace Type Code with Subclasses</u>
	<u>Reverse Conditional</u> by BILL MURPHY and MARTIN FOWLER
	<u>Self Encapsulate Field</u>
	<u>Separate Data Access Code</u> (Link Only)
	<u>Separate Query from Modifier</u>
	<u>Split Loop</u> by MARTIN FOWLER
	<u>Split Temporary Variable</u>
	<u>Substitute Algorithm</u>
	<u>Use a Connection Pool</u> (Link Only)
	<u>Wrap entities with session</u> (Link Only)

Viele Refactorings gehen auch in „Best Practices“ über. So könnte man beispielsweise alle Tipps der - sehr hilfreichen - Seite  <http://www.javapractices.com> in Refactorings umwandeln und würde bestimmt noch einige gute neue Refactoring Patterns herausbekommen.

5.2 Problem und Heilung

Bevor nun einige Beispiele von Refactorings (auch unter Eclipse) genauer vorgestellt werden, ist es hilfreich, eine Anleitung zur Hand zu haben, die den „Smell“ und die dazugehörige mögliche „Heilung“ nennt. Diese letzte Referenz aus  [Fo99] kann hier aber nur als grober Leitfaden dienen. Ohne eine genaue Beschreibung des Geruches aus  [Fo99] ist es nicht einfach, die Probleme die unten auf der linken Seite stehen zu identifizieren.



Es ist aber in jedem Falle hilfreich, um sich vorab, unter den auf der linken Seite aufgeführten Problemen, etwas vorstellen zu können. Folglich können die Lösungen auf der rechten Seite der vorigen Refactoring Übersichtsseite, schnell gefunden und geöffnet werden.

Geruch / „Smell“	Refaktorisierung / mögliche Lösung
Alternative Klassen mit verschiedenen Schnittstellen	Methode umbenennen Methode verschieben
Ausgeschlagenes Erbe	Vererbung durch Delegation ersetzen
Datenklassen	Collection kapseln Feld kapseln Methode verschieben
Datenklumpen	Ganzes Objekt übergeben Klasse extrahieren Parameterobjekt einführen
Divergierende Änderungen	Klasse extrahieren
Duplizierter Code	Klasse extrahieren Methode extrahieren Methode nach oben verschieben Template Methode bilden
Faule Klasse	Hierarchie abflachen Klasse integrieren
Große Klasse	Klasse extrahieren Schnittstelle extrahieren Unterklasse extrahieren Wert durch Objekt ersetzen
Kommentare	Methode extrahieren Zusicherung einführen
Lange Methode	Bedingung zerlegen Methode durch Methodenobjekt ersetzen Methode extrahieren Temporäre Variable durch Abfrage ersetzen
Lange Parameterliste	Ganzes Objekt übergeben Parameter durch Methode ersetzen Parameterobjekt einführen
Nachrichtenketten	Delegation verbergen
Neid	Feld verschieben Methode extrahieren Methode verschieben
Neigung zu elementaren Typen	Array durch Objekt ersetzen Klasse extrahieren Parameterobjekt einführen Typenschlüssel durch Klasse ersetzen Typenschlüssel durch Unterklassen ersetzen Typenschlüssel durch Zustand / Strategie ersetzen Wert durch Objekt ersetzen
Parallele Vererbungshierarchien	Feld verschieben Methode verschieben
Schrotkugeln herausoperieren	Feld verschieben Klasse integrieren Methode verschieben
Spekulative Allgemeinheit	Hierarchie abflachen Klasse integrieren Methode umbenennen Parameter entfernen
Switch-Befehle	Bedingten Ausdruck durch Polymorphismus ersetzen Null-Objekt einführen Parameter durch explizite Methoden ersetzen Typenschlüssel durch Unterklassen ersetzen Typenschlüssel durch Zustand / Strategie ersetzen
Temporäre Felder	Klasse extrahieren Null-Objekt einführen
Unangebrachte Intimität	Bidirektionale Assoziation durch gerichtete ersetzen Delegation verbergen Feld verschieben Methode verschieben Vererbung durch Delegation ersetzen
Unvollständige Bibliotheksklasse	Fremde Methode einführen Lokale Erweiterung einführen
Vermittler	Delegation durch Vererbung ersetzen Methode integrieren Vermittler entfernen

Tab.: Smell und Refactoring

```
// Diese Methode
```

Ganz interessant an dieser Tabelle ist, dass **Kommentare** (oder zu viele Kommentare) auch einen negativen „Smell“ haben können und in diesem Fall durch eine einfachere, erklärende Methode oder durch eine klar lesbare Zusicherung ersetzt werden können. Kommentare können daher auch Teil einer Refactoringverbesserung sein.

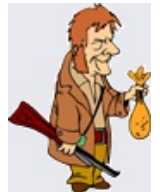
```
switch(o2.getVal())
```

Für Softwaretechniker ist es immer wieder interessant und fast schon ein Sport die **Switch**-Befehle zu untersuchen. Da viele Switch-Befehle durch elegantere Konstrukte - wie die Ausnutzung der Polymorphie - ersetzt werden können, haftet dem Switch-Befehl etwas Anrüchiges an. Der Sport besteht nun darin, eine bessere Variante zu finden.

Was hat FOWLER wohl mit Schrotkugeln gemeint?

Schrotkugeln

Angenommen Sie führen irgendwo im Code eine Änderung (D1) durch. Infolge dieser Änderung - was Sie schon öfter mal gemacht haben - müssen Sie auch an anderen Stellen in anderen Klassen Änderungen / Anpassungen machen (D2 bis DN).



Es drängt sich der Verdacht auf, dass Sie alle Stellen - an denen Sie Änderungen machen (D1...DN), in einer Klasse zusammenfassen könnten. Die Stellen an denen bei Änderungen weitere Änderungen erforderlich sind, wurden von Martin Fowler als Schrotkugeln bezeichnet, die quasi „schmerzhaft“ im Körper sitzen.

5.3 Refactoring unter Eclipse

Bei Eclipse ist der Refactoring Support leicht über das Menü der rechten Maustaste zu erreichen. Dabei ist zu beachten, dass das Menü kontextsensitiv ist - also davon abhängt was gerade markiert ist (Klasse, Methode, Attribute, Codefragment, etc.).

Es lohnt sich daher immer, mit der rechten Maustaste verschiedene Bereiche zu markieren.



Hinweis

Refactor	Alt+Shift+T			
Local History			Rename...	Alt+Shift+R
References			Move...	Alt+Shift+V
Declarations			Change Method Signature...	Alt+Shift+C
Occurrences in File	Ctrl+Shift+U		Pull Up...	
Run As			Push Down...	
Debug As			Extract Interface...	
Team			Generalize Type...	
Compare With			Use Supertype Where Possible...	
Replace With			Inline...	Alt+Shift+I
Preferences...			Extract Method...	Alt+Shift+M
			Extract Local Variable...	Alt+Shift+L
			Extract Constant...	
			Introduce Parameter...	
			Convert Local Variable to Field...	Alt+Shift+F

Abb.: Screenshot aus Eclipse 3.1 Final

6 Refactoring-Patterns

In diesem Kapitel werden beispielhaft einige Refactorings und einige Beispiele aus dem Refactoring Katalog erklärt, um einen praxisnahen Zugang zu der Materie des Refactorings zu bekommen.

Das eigene Experimentieren mittels einer IDE ist jedoch durch nichts zu ersetzen!

6.1 Extract Method



Definition

Extract Method

Extract Method schneidet aus einer größeren Methode kleinere heraus. Dies hat den Effekt, dass der resultierende Code lesbarer und testbarer wird. Aber auch die anderen Ziele aus dem Kapitel 3 können hier positiv wirken.

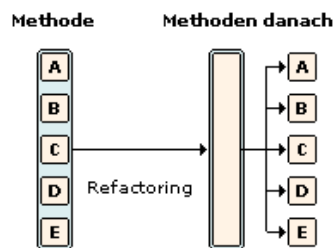


Abb.: Refactoring bei Methoden



Beispiel

Berechnung einer Hypothek

Soll beispielsweise in einer Methode eine Hypothek berechnet werden, so können die folgenden Aufgaben auch in einer Methode erledigt werden:

- Entnehme Stammdaten
- Hole Darlehenssumme, Jahreszins, Tilgung, Sondertilgung
- Berechne monatliche Rate
- Berechne Laufzeit bis zur vollständigen Tilgung
- Geleistete Zinszahlungen in diesem Zeitraum
- Gesamter Aufwand für das Darlehen
- Erstelle Finanz-PDF für jedes einzelne Jahr

Es ist jedoch sofort ersichtlich dass eine Aufteilung in kleinere Einheiten sinnvoll ist:




Quellcode

Methode zur Bearbeitung von Hypotheken

```
... getStammdaten(...)
... addFinancialData(...)
... calcMonthRare(...)
... calcTimeToFinish(...)
... calcInterest4Period(...)
... calcTotalExpense(...)
... createPDF4EachYear(...)
```

Eine solche Methode dürfte sich sehr viel einfacher lesen, als in dem Fall, in dem die Methode gleich alles selbst erledigt. Und selbst wenn die hier gezeigten Methoden u. U. privates einer Klasse sind, kann man sie leicht mit geeigneten Werkzeugen testen (siehe Testkapitel).

Der Korrektheit halber sei hier angemerkt, dass die oben gezeigten Methoden unter Umständen verschiedene Schichten oder Zuständigkeitsbereiche adressieren. D. h. es ist sowieso ein Refactoring angebracht, dass hier vielleicht sogar Klassen extrahiert. Z. B. I/O-Klassen für die Dateneingabe und die Datenausgabe wie den PDF Report!

MARTIN FOWLER zeigt ein noch einfacheres Beispiel  [Fo99]:



Quellcode

Quellcode vorher <pre> printOwing(double amount) { printBanner(); //print details System.out.println("name:" + _name); System.out.println("name:" + _name); } </pre>	
Quellcode nachher <pre> printOwing(double amount) { printBanner(); void printDetails(amount); } </pre>	
Quellcode <pre> void printDetails(double amount) { System.out.println("name:" + _name); System.out.println("name:" + _name); } </pre>	

Dieses Beispiel erscheint schon fast zu trivial um sinnvoll zu sein. Wieso sollte ich zwei lächerliche Druckbefehle extra aufwendig in eine Methode auslagern? Das kostet ja sogar Laufzeit!

Dennoch zeigt das Beispiel worum es geht. Der Quellcode wird nachher lesbarer. Falls einmal weitere Ausgaben hinzugefügt werden müssen, kann dies lokal sauber geschehen, ohne `printOwing` zu beeinflussen. `PrintDetails` kann (bei komplexeren Methoden auch besser getestet werden). Man kann `printDetails` z. B. auch mit Aspekten versehen.

Bei längerem Nachdenken findet man weitere Vorteile.

Wegen all dieser Gründe ist **Extract Method** sicher eines der wichtigsten Refactorings, das von jeder IDE unterstützt wird (die diesen Namen IDE auch verdient).

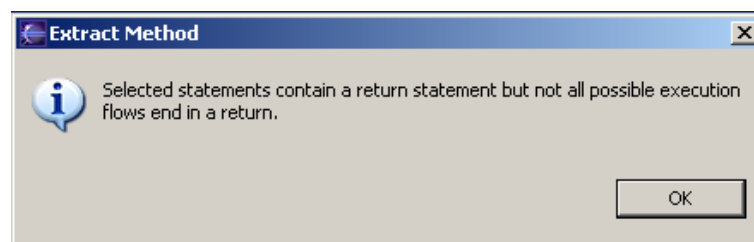
Aus mehreren Gründen ist aber Vorsicht geboten:

- Was passiert mit lokalen Variablen der Methode, die im Codesegment enthalten sind, das man in der IDE markiert hat? Dieses Problem wird auch von FOWLER erkannt und beschrieben.

Lösung: Diese müssten u.U. als Parameter übergeben werden. Das ist leicht, wenn diese Variablen nur lesend verwendet werden. Schwieriger wird es, wenn diese auch verändert werden! Dann müsste man die Ergebnisse wieder hochreichen. Hier muss man genau schauen, ob man nicht einen „Kampf“ mit Variablen eingeht, der u. U. verloren gehen kann. Gut ist dennoch, dass auch IDEs einem bei der Definition der Variablen unterstützen!

- Noch lästiger sind Codeabschnitte, die z. B. ein `return` in einem `if` kapseln (hier gibt es Schwächen bei FOWLER, die er selbst nicht erwähnt). Ein Refactoring würde hier komplett den Sinn entstellen, selbst wenn alle Variablen korrekt gehandhabt würden! Wohin soll die Submethode springen? Wieder in die Hauptmethode? Das war nicht Sinn der Sache. Eine lästige Lösung ist hier evtl. nur, den Aufruf der Submethode über ein Bool selbst wieder in ein `if` mit `return` zu packen. Vielleicht gibt es aber auch noch bessere Lösungsvorschläge von Ihnen!

Zum Glück warnen die meisten IDEs schon vorher - hier Eclipse 3.1.



6.2 Inline Method



Definition

Inline Method

Unter Inline Method versteht man die inverse Operation zu **Extract Method**. In bestimmten Fällen ist der Overhead für einen Methodenaufruf auch aus Gründen der Lesbarkeit nicht angebracht. In diesem Fall wird der Inhalt der Methode zurück in ihren Aufruf integriert. In fast allen Fällen ist nicht Performance sondern Lesbarkeit der Grund für den Aufruf von **Inline**.



Quellcode

Quellcode vorher

```
int getRating() {
    return ( moreThanFiveLateDeliveries() ) ? 2 : 1;
}

boolean moreThanFiveDeliveries()
    return _numberOfLateDeliveries > 5;
}
```

Quellcode nachher

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Wie man erkennt, liefert die Methode `moreThanFiveLateDeliveries` keinen Mehrwert, da sie quasi nur eine Membervariable kapselt, die genauso heißt.

Das Refactoring inline selbst kann nicht nur auf Methoden angewendet werden. Im Refactoring-Katalog sind noch zwei weitere Anwendungen verzeichnet:

- **Inline Temp** eliminiert eine (evtl. sogar zuvor) temporäre Variable und ersetzt diese durch ihre Zuweisung:

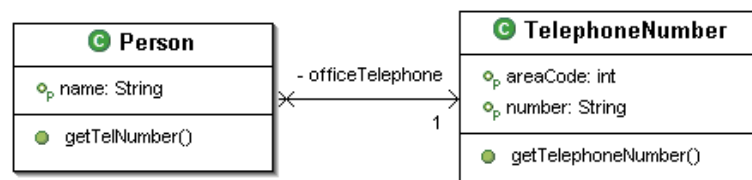
```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

Wird danach einfach:

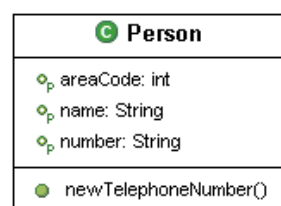
```
return (anOrder.basePrice() > 1000)
```

Offensichtlich ändert sich auch hier nicht viel an Lesbarkeit, Testbarkeit, Refactorbarkeit, etc.

- **Inline Class** verschmelzt eine - unter Umständen - unnötige Klasse mit einer anderen Klasse.



Wird zu



Dieses Refactoring sieht man häufiger nach dem Modellieren von Klassendiagrammen wenn ohne Attribute und Methoden modelliert wurde. Viele Klassen sind dann so „dünn“ dass sie in andere Klassen integriert werden können.

6.3 Triviale Refactorings (Eclipse)

Die folgenden Refactorings sind architektonisch nicht besonders anspruchsvoll, aber trotzdem extrem wichtig.

- **Undo**: Es klingt unglaublich, aber auch die Undo-Funktion sollte in der Lage sein, das letzte Refactoring rückgängig machen zu können. Kann Ihre IDE dies?



Frage von Melanie

Wieso ist **Undo** auch ein Refactoring und warum ist es so wichtig?

 Antwort (Siehe Anhang)

- **Redo**: Es wurde festgestellt, dass das Löschen des **Undo** des letzten Refactorings doch ein Fehler war und man das Refactoring braucht und jetzt gerne wieder hätte.
- **Rename**: Das Umbenennen von Klassen, Methode, Attributen, etc. ist ebenfalls Teil des Refactoring-Kataloges und wahrscheinlich das meist Verwendete. Der Punkt dabei ist, dass das Umbenennen weder „von Hand“ noch auf Dateisystemebene durchgeführt werden darf. Dabei fängt es schon bei der Namensgleichheit von Datei- und Klassenname innerhalb von Java an, die es demzufolge zu beachten gilt. Da aber eine Methode noch 100 mal woanders referenziert werden kann (oder nach Murphy wird), will dies niemand von Hand ändern. Hier ist jeder dankbar über die Maschine (=IDE) die einem diese Sisyphusarbeit abnimmt.
- **Move**: Auch das Verschieben von Klassen (zwischen Packages), Methode, Attributen, etc. ist nicht immer einfach per Hand durchzuführen, weshalb alle guten IDEs diesen Vorgang unterstützen.
- **Change Method Signature**: Dieses Refactoring ändert die Methodensignatur in der Definition und in allen Aufrufen.

6.4 Extract Interface



Definition

Extract Interface


Erstellt ein Interface aus der aktuellen Klasse. Weiterhin wird die originäre Klasse so geändert, dass diese Klasse jetzt dieses neu erstellte Interface implementiert.



Achtung

- Wichtig ist auch hier wieder die Änderung aller **Referenzen**! Das ist eigentlich die wichtigste Aufgabe dieses Refactorings. Alle Klassen die vorher auf die Klasse verwiesen haben, implementieren jetzt gegen Interfaces, was in der Regel sehr sinnvoll ist.

Leider haben hier viele IDEs ein unterschiedliches Verhalten: So nimmt Eclipse beispielsweise nicht die bereits vorhandenen Methodensignaturen in das Interface mit. Dies wünscht man sich ja eigentlich und möchte dies nicht von Hand machen. Ein Default-Mitnehmen und danach Löschen von Methoden wäre u. U. einfacher.

Der Refactoring-Katalog kennt weiterhin ein **extract** Package  www.refactoring.com.

GERARD M. DAVIDSON schreibt hierzu: „*Ein Package hat entweder zu viele Klassen um verständlich zu sein oder es riecht verworren / buntgewürfelt*“.

Beim Browsen von Packages hat bestimmt jeder schon einmal folgenden „smell“ gesehen (z. B. `java.io` mit 83 Klassen in Java 1.5).



6.5 Weitere Refactorings



Definition

Pull Up / Push Down

Verschiebt Felder und Methoden zwischen Klasse und Superklasse.

- Dieses Refactoring ist auf ein oder auf mehrere Elemente anwendbar.

In fast allen IDEs müssen hier die Variablen mit der Maus richtig markiert werden, sonst kann dieses Refactoring nicht richtig automatisch vollzogen werden.

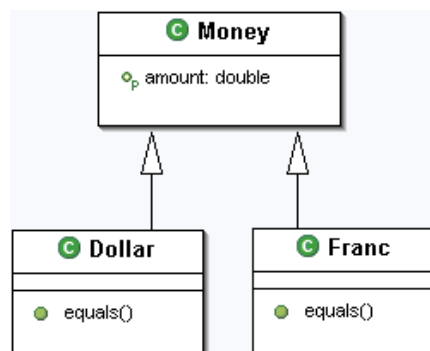


Beispiel

Beispiel

Superklasse Money

In einem bekannten Beispiel von KENT BECK [Be03] werden die Klassen **Dollar** und **Franc** um eine Superklasse **Money** angereichert. Dann ist klar, dass doppelte Methoden wie das bekannte `equals()` hochgezogen werden müssen um Duplikate zu beseitigen.



Definition

Encapsulate Field

Setzt die Sichtbarkeit eines Attributes auf `privat` und fügt `getter` und `setter` hinzu.

Ein Refactoring Feature, das aus UML-Modellierungstools schon länger bekannt ist, sich aber in einer IDE sehr nützlich macht.



Beispiel

Benachrichtigung bei Änderung einer Klasse

In einer Anwendung war ein Feld bisher `public`. Nun sollen aber andere Klassen bei Änderung dieser Klasse benachrichtigt werden:

```

public void setNice(int nice) {
    System.out.println("Interessant ist: " + nice);
    this.nice = nice;

    // "nice" bekannt machen (java.util.Observable)
    setChanged();
    notifyObservers();
}
  
```

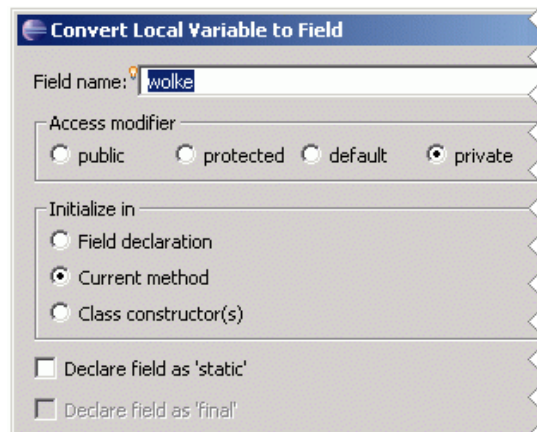


Definition

Convert Local Variable to Field

Erhebt eine lokale Variable in den Status einer Klassenvariablen.

Dies ist des Öfteren der Fall wenn der Inhalt dieser Variablen in anderen Methoden sinnvoll weiter verwendet werden kann. Ein praktisches Beispiel wurde bereits in **Extract Method** gezeigt.



6.6 Weak Refactorings

Bevor abschließend auf zwei klassische Refactoringbeispiele eingegangen wird, hier noch zwei weniger wichtige Refactorings, die eher an Source-Code Veränderungen erinnern. Daher sind diese Refactorings eher in IDEs zu finden und weniger in den „heiligen“ Refactoring-Katalogen.

Interessant ist, dass das Refactoring **Spektrum** von einer kleinen Source-Code Manipulation, bis hin zu komplexen Refactorings wie beispielsweise in eine bestehende Architektur Entwurfsmuster „einzuweben“ reicht.



Definition

- **Introduce Parameter:** Eine Expression wird durch einen Parameter ersetzt. Alle diese Methode aufrufenden anderen Methoden werden entsprechend abgeändert, dass die Expression `22*getTickRate()` als Parameter eingefügt wird.

```
public int erlebnis() {
    int a = 22*getTickRate();
    return processTickRate(a);
}
```

Die Ersetzung in der Quellmethode lautet dann:

```
public int erlebnis(int magicNumber) {
    int a = magicNumber;
    return processTickRate(a);
}
```

Eine aufrufende Methode `private void doSomething() {erlebnis();}` würde dann zu `private void doSomething() {erlebnis(22*getTickRate());}` geändert werden, d. h. die Expression selbst übergeben werden. Auch hier wird dieses Refactoring eingesetzt, um die Lesbarkeit zu erhöhen.

- **Extract Local Variable:** Soll die Expression kein Parameter sein, sondern einfach eine lokale Variable eingeführt werden, so hilft dieses Refactoring. Es erinnert stark an **Introduce Explaining Variable** www.refactoring.com aus dem Pattern Katalog von MARTIN FOWLER.

Quellcode vorher

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 )
{ // do something
}
```


Wird zu:

Quellcode nachher

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;
if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
    // do something
}
```

Auch hier wird der Code lesbarer, da die meisten Codeleser nur die Zeile im `if` lesen werden und die Definitionen einfach „glauben“ werden.

- **Extract Constant:** Hier wird aus einer lokalen Konstante ein Klassenattribut gemacht, welches optional auch noch als `static final` deklariert werden kann (z. B. für pi).
- **Use Supertype where Possible:** Hilft bei dem Versuch, statt eines Typen, den Typ der Superklasse zu verwenden. Dies kann gerade bei Ausnutzung von Polymorphie-Eigenschaften immer sinnvoll sein.

6.7 Architektur-Refactorings und erklärende Refactorings

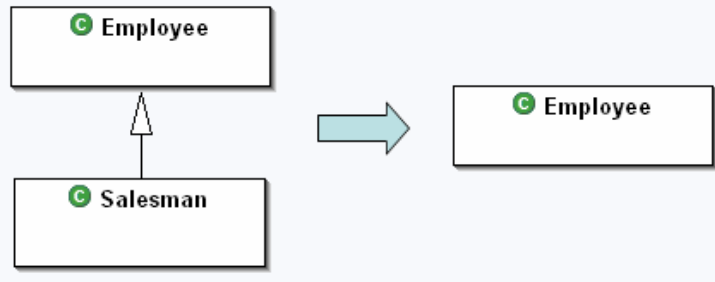
Ein Großteil der Refactorings des Kataloges sind Architektur-Refactorings die in der Regel Klassenstrukturen oder Teile von Klassendiagrammen verändern oder verschieben.



Beispiel

Beispiel für Architektur-Refactoring

Ein klassisches Beispiel ist **Collapse Hierarchy** und ähnelt dem eingangs besprochenen **Inline Class Refactoring** aus Eclipse. Hier zeigen die Superklasse und die Subklasse keine großen Unterschiede und sind klein genug, um zu verschmelzen.



Und so lebt der Salesman vielleicht nur als Attribut in Employee weiter.

Sehr viele Refactorings sind allerdings erklärend. Während auf den vorigen Seiten z. B. eine erklärende temporäre lokale Variable eingeführt wurde, kann man genauso gut eine erklärende Methode einführen. So z. B. **Decompose Conditional**



Quellcode

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

wird zu

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Dies ist kein **extract Method**, weil keine existierenden Codeteile herausgeschnitten werden können, aber es scheint ähnlich zu sein, da hier erklärend ersetzt und vereinfacht wird.

6.8 Fazit

- Refactoring-Patterns sind mittlerweile ein wichtiges und unverzichtbares Werkzeug bei der Softwareentwicklung. Dies gilt besonders für alle Zyklen, die TDD (Test Driven Development) ähnlich sind. Auch in XP und allen agilen Methoden der Softwaretechnik ist Refactoring ein fester Bestandteil, um dem Endziel mit einem frühen Prototypen so schnell wie möglich näher zu kommen.
- Bei dem heutigen extremen Kostendruck ist es daher Pflicht, sich schnell anpassen zu können und den Code oder seine Architektur zu ändern.
- Wichtig ist, Refactoring nicht blind einzusetzen, sondern sich ständig auch Gedanken über den Erfolg oder die Leistungsfähigkeit zu machen: Erreichen die Refactorings auch alle Dokumente? Erreichen sie Deskriptoren? Alle XML-Dateien? Was passiert wenn ich eine wichtige Methode umbenenne, diese aber leider per JNDI im `ejb-jar.xml` referenziert wird? Es ist also eine Frage der **Reichweite** des Refactorings!

Solche Fälle sind besonders böse, da sie unter Umständen erst spät in Integrationstests erkannt werden und daher besonders teuer sind! Daher müssen alle Dateien, die nicht im Quellbaum hängen oder JavaDocs oder sonstige Meta-Informationen mit in Betracht gezogen werden.

Glücklicherweise sind die meisten Refactorings in kleinen Projekten eher ungefährlich und zeigen nur lokale Auswirkungen.

- Dank moderner IDEs sind auch viele komplexe Refactorings mittlerweile entzaubert. Es lohnt also immer sich mit den Refactoring-Features der IDE auseinanderzusetzen.



Achtung



Zusammenfassung

- Mit der Umgestaltung des Programmcodes werden in der Regel einige wichtige Ziele verfolgt: Lesbarkeit, Verständlichkeit, Übersichtlichkeit, Vermeidung von Redundanzen, Erweiterbarkeit, Modularität und Testbarkeit.
- Für ein professionelles Refactoring sind objektorientierte Programmierparadigmen sehr wichtig.
- Ein Ziel des Refactorings ist es, Entwurfsmuster in den Code zu integrieren.
- Der Begriff des Refactorings bezieht sich überwiegend auf OO-Code.
- Kernpunkt der extrem wichtigen Toolunterstützung ist, dass
 - A) viele Refactorings ohne Tools fast nicht möglich sind und
 - B) die IDE wichtig ist, um bei bestimmten Refactorings einfach aufzupassen.
- Refactorings haben Grenzen und können Fehler erzeugen, wenn die Reichweite nicht ausreicht.
- Beim Refactoring ist es wichtig, sich durch UnDo und Versionskontrollsystemen abzusichern.
- Quellcode ist oftmals verbesserungsbedürftig, er hat einen schlechten Geruch („Smell“). Dieser muss erkannt und definiert werden, um durch „Heilung“ gezielt beseitigt werden zu können.
- Kernelement des Refactorings sind so genannte Refactoring-Muster. Diese werden ähnlich wie Entwurfsmuster eingesetzt, um den Code zu verbessern.
- Einige der Refactorings wie UnDo, ReDo, Rename, usw. sind architektonisch nicht besonders anspruchsvoll, aber extrem wichtig.
- Ein Großteil der Refactorings des Kataloges sind Architektur-Refactorings, die in der Regel Klassenstrukturen verändern oder Teile von Klassendiagrammen verändern oder verschieben.
- Refactorings sollten nicht blind eingesetzt werden. Man sollte sich ständig auch Gedanken über den Erfolg oder die Leistungsfähigkeit machen.
- Bei dem heutigen extremen Kostendruck ist es daher Pflicht, sich schnell anpassen zu können und den Code oder seine Architektur zu ändern.

Sie sind am Ende dieser Lerneinheit angelangt. Auf der folgenden Seite finden Sie noch die Übungen zur Wissensüberprüfung.

Übungen



Formulieren

Übung REF-01

Refactoring-Katalog

Schauen Sie sich alle Refactorings des Katalogs von Martin Fowler unter der Adresse <http://www.refactoring.com/catalog> an. Spielen Sie mit einigen Refactorings. Probieren Sie diese in der IDE aus. Schreiben Sie etwas über die drei Refactorings, die Sie besonders beeindruckt haben, und über je eines, dass Sie für überflüssig halten und eines, dass Sie nicht verstanden haben!

Ihre Ausarbeitung schicken Sie bitte über das Lernraumsystem an Ihre Kursbetreuung.

Bearbeitungsdauer: 40 Minuten



Formulieren

Übung REF-02

The Smell of Real Code

Schauen Sie sich bitte den folgenden Code an: [Democode \(Siehe Anhang\)](#)

- Welche „smells“ riechen Sie?
- Welche Heilungen schlagen Sie vor?
- Wie würden Sie den nachstehenden Code umgestalten?
- Welche Probleme könnten dabei auftreten?

Bereiten Sie Ihre Antworten so vor, dass Sie in der Lage sind diese - entweder in der nächsten Audiokonferenz oder der nächsten Präsenzveranstaltung - vorzutragen und sich an der Diskussion zu beteiligen..

Bearbeitungsdauer: 10 Minuten



Formulieren

Übung REF-03

Projektaufgabe - Refactoring

Welche Refactorings kann Ihre IDE? Gibt es dazu Dokumentation? Es kostet Sie sicherlich extrem viel Zeit, alle Refactorings zu beherrschen. Aber bitte haben Sie selbst den Anspruch, möglichst viele Refactorings Ihrer IDE zu können! Welche zwei Refactorings haben Sie in Ihrem Softwareprojekt mit der IDE am meisten verwendet?

Beschreiben Sie in Ihrer Projektdokumentation mehrere (nicht nur triviale) Refactorings (also nicht nur **rename**), die Sie auch tatsächlich ausgeführt haben.

Bearbeitungsdauer: 20 Minuten

Appendix

Frage von Melanie

Melanie:

Was ist denn eigentlich das Problem das Refactoring löst?

Antwort:

Bisher wurde gesagt, dass Refactoring Code verbessert. Es ist daher in der Regel so, dass der Code vielleicht nicht direkt fehlerhaft ist, aber er ist irgendwie schlecht. Vielleicht ist er schlecht designed. Um die Definition dieses schlechten Codes wird es im Folgenden gehen. In der englischsprachigen Literatur hat sich hier der Begriff des „Smells“ eingebürgert. Quellcode hat also einen „schlechten Geruch“. Diesen zu definieren und zu erkennen ist daher unser Anliegen.

Wir werden im späteren Verlauf viele „Refactoring-Patterns“ also Muster kennenlernen, was natürlich wichtig ist. Dennoch ist es fast viel wichtiger, die Probleme im Code, d. h. den „üblen Geruch“ zu erkennen, als hunderte von Mustern auswendig zu lernen!

Frage von Melanie

Melanie:

In der Literatur wird `undo` manchmal als Refactoring-Pattern mit aufgenommen obwohl es vielleicht keines ist. Es ist ja lediglich eine Funktion oder eine Art Pattern, alle letzten Aktionen transaktional zu verarbeiten und bei Bedarf rückgängig zu machen.

Antwort:

`undo` ist deshalb wichtig, weil es einige Refactorings gibt, die viele Codestellen und Dateien anfässt. Der Autor kennt Beispiele (z. B. bei einem einfachen `rename` einer Klassenmethode (was ja wieder ein zweifelhaftes Refactoring ist) wo Hunderte von Codestellen und Dutzende von Dateien geändert werden. Hat man aber bei diesem Refactoring Fehler gemacht oder ist das Refactoring ungünstig, dann ist ein „Rückgängigmachen“ von „Hand“ quasi unmöglich. Hier braucht man ein `undo`. Fehlt dies, so wurde hoffentlich regelmäßig ein gutes Versionsmanagementsystem (siehe entsprechende Lerneinheit) verwendet, was einem einen sicheren Boden bietet.

Democode

```
//(C) Prof. Dr. Stefan Edlich
```

```
package de.edlich.air;
```

```
import java.io.IOException;
import java.lang.reflect.Field;
import java.util.Enumeration;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletContext;
import javax.servlet.http.*;
```

```
// Referenced classes of package de.edlich.air:
// AirUtils, Action, ValiError, ActionError
```

```
public class Dispatcher extends HttpServlet
{
```

```
    AirUtils dispUtils;
```

```
    public Dispatcher()
    {
        dispUtils = new AirUtils();
    }
```

```
    public void service(HttpServletRequest req, HttpServletResponse res)
    {
        String targetActionClass = req.getParameter("ACTIONCLASS");
        dispUtils.log("AIR-MESSAGE: targetActionClass=" + targetActionClass);
        if(targetActionClass == null)
        {
            dispUtils.log("### AIR-ERROR: No Servlet-Mapping defined. Parameter ACTIONCLASS ist not set!");
            dispUtils.log2Console(res, "No Servlet-Mapping defined. Parameter ACTIONCLASS ist not set!");
            return;
        }
        Class tmpC = null;
        Action callAction = null;
        try
        {
            tmpC = Class.forName(targetActionClass);
            callAction = (Action)tmpC.newInstance();
        }
        catch(Exception e)
        {
            dispUtils.log("### AIR-ERROR: Cannot load class. Wrong mapping: " + targetActionClass);
            e.printStackTrace();
            dispUtils.log2Console(res, "Cannot load class. Wrong mapping: " + targetActionClass);
            return;
        }
        for(Enumeration emu = req.getParameterNames(); emu.hasMoreElements();)
        {
            String tmpS = (String)emu.nextElement();
            String tmpAttr = req.getParameter(tmpS);
            String tmpAttrs[] = req.getParameterValues(tmpS);
```



```

Field theAttribut = null;
if(!tmpS.equals("ACTIONCLASS") && !tmpS.equals("JSP"))
{
    dispUtils.log("AIR-MESSAGE: Processing: " + tmpS + "=" +
tmpAttr);
    try
    {
        theAttribut = tmpC.getField(tmpS);
        Class fieldType = theAttribut.getType();
        if(fieldType.getName().equals("java.lang.String")) {
            theAttribut.set(callAction, tmpAttr);
        } else
        if(fieldType == Integer.TYPE) {
            theAttribut.setInt(callAction,
Integer.parseInt(tmpAttr));
        } else
        if(fieldType == Float.TYPE) {
            theAttribut.setFloat(callAction,
Float.parseFloat(tmpAttr));
        } else
        if(fieldType == Double.TYPE){
            theAttribut.setDouble(callAction,
Double.parseDouble(tmpAttr));
        } else
        if(fieldType == Long.TYPE){
            theAttribut.setLong(callAction, Long.parseLong(tmpAttr));
        } else
        if(fieldType == Byte.TYPE){
            theAttribut.setByte(callAction, Byte.parseByte(tmpAttr));
        } else
        if(fieldType.isArray()) {
            theAttribut.set(callAction, tmpAttrs);
            for(int i = 0; i < tmpAttrs.length; i++) {
                dispUtils.log("AIR-MESSAGE: Processing: " + tmpS + "="
+ tmpAttrs[i]);
            }
        } else{
            throw new IllegalArgumentException("Unbekannter Datentyp!");
        }
    }
    catch(NoSuchFieldException nsf){
        dispUtils.log("AIR-MESSAGE: Can not set Field " + tmpS + ".
It's not there!");
    }
    catch(IllegalAccessException iae) {
        dispUtils.log("AIR-MESSAGE: Field " + tmpS + " has wrong
access rights!");
    } catch(IllegalArgumentException iae) {
        dispUtils.log("AIR-MESSAGE: Tried to convert Field " + tmpS
+ " but failed. (E.g. you entered 'abc' to be put into an int?!);");
    }
}

String sourceJSP = req.getParameter("JSP");
dispUtils.log("AIR-MESSAGE: sourceJSP=" + sourceJSP);
if(sourceJSP == null) {
    dispUtils.log("### AIR-ERROR: JSP is not a valid parameter in your
JSP. Can't call it back!");
}

```

```

        dispUtils.log2Console(res, "JSP is not a valid parameter in your
JSP. Can't call it back!");
        return;
    }
    ServletContext sc = getServletContext();
    if(!callAction.validate(req, res))
    {
        req.setAttribute("ACTIONCLASS", callAction);
        dispUtils.log("AIR-MESSAGE: Validate returned: " +
((ValiError)req.getAttribute("VALIERROR")).vErr);
        try {
            RequestDispatcher rd = sc.getRequestDispatcher("/") + sourceJSP);
            rd.forward(req, res);
        }
        catch(Exception e)
        {
            dispUtils.log("### AIR-ERROR: Can't call JSP " + sourceJSP);
            e.printStackTrace();
            dispUtils.log2Console(res, "Can't call JSP " + sourceJSP);
        }
        return;
    }
    String jspToFollow = callAction.execute(req, res);
    dispUtils.log("AIR-MESSAGE: Calling Mapping String=" + jspToFollow);
    if(jspToFollow == null) {
        ActionError executeError =
(ActionError)req.getAttribute("ACTIONERROR");
        if(executeError.actErr != null) {
            dispUtils.log("### AIR-ERROR: Servlet returned no valid target-JSP: "
+ executeError.actErr);
            dispUtils.log2Console(res, "Servlet returned no valid target-JSP: "
+ executeError.actErr);
        } else {
            dispUtils.log2Console(res, "The Action-execute returned null=error
but set no action error object!");
        }
        return;
    }
    req.setAttribute("ACTFIELDS", callAction);
    dispUtils.log("AIR-MESSAGE: Calling JSP:" + jspToFollow);
    if(jspToFollow == null) {
        dispUtils.log("AIR-MESSAGE: No fine mapping defined for target: " +
jspToFollow);
        dispUtils.log2Console(res, "AIR-MESSAGE: No fine mapping defined for
target: " + jspToFollow);
    }
    try {
        RequestDispatcher rd = sc.getRequestDispatcher("/") + jspToFollow);
        rd.forward(req, res);
    }
    catch(Exception e){
        dispUtils.log("### AIR-ERROR: Can't call to " + jspToFollow);
        e.printStackTrace();
    }
    try {
        res.sendError(6, "### AIR-ERROR: Can't call to " + jspToFollow);
    } catch(IOException ie) {
        ie.printStackTrace();
    }
}

```

```
    return;  
  }  
}  
}
```