



**h\_da**

HOCHSCHULE DARMSTADT  
UNIVERSITY OF APPLIED SCIENCES

**fbi**

FACHBEREICH INFORMATIK

# Refactoring von Legacy Code

Alexander Dahmen

Hochschule Darmstadt – FB Informatik

Schöfferstr. 8b, 64295 Darmstadt

[Alexander.Dahmen@stud.h-da.de](mailto:Alexander.Dahmen@stud.h-da.de)

Tom Niklas Brell

Hochschule Darmstadt – FB Informatik

Schöfferstr. 8b, 64295 Darmstadt

[Tom.N.Brell@stud.h-da.de](mailto:Tom.N.Brell@stud.h-da.de)

Im Rahmen der Veranstaltung:

„Wissenschaftliches Arbeiten in der Informatik 1“

Sommersemester 2016

Referentin: Prof. Dr. Inge Schestag

[inge.schestag@h-da.de](mailto:inge.schestag@h-da.de)

## **Eidesstattliche Erklärung**

Wir versichern hiermit, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von uns selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 14. Juni 2016

---

## **Abstract**

Diese Ausarbeitung beschäftigt sich mit dem Thema "Refactoring von Legacy-Code".

Das Ziel ist es, dem Leser einen Einblick in die Welt des Refactoring zu geben, ihm dazu allgemeine Grundlagen zu erläutern und diese dann zum Schluss mit Beispielen zu ergänzen.

Der Aufbau der Ausarbeitung setzt sich wie folgt zusammen: Zu Beginn werden die Grundlagen zum Thema Legacy-Code erläutert. Darauf folgt eine Einführung in das Refactoring. Zum Schluss werden einige Beispiele für Tools vorgestellt. Darunter befinden sich sowohl IDE-interne, als auch externe Tools.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>5</b>
<b>2</b>	<b>Grundlagen des Phänomens Legacy Code</b>	<b>6</b>
2.1	Merkmale und Probleme von Legacy Code . . . . .	6
2.2	Gründe für das Arbeiten mit Legacy Code . . . . .	7
<b>3</b>	<b>Code überarbeiten: Einführung ins Refactoring</b>	<b>8</b>
3.1	Test-gesteuertes Entwickeln: Unit Tests . . . . .	8
3.1.1	Wofür braucht man Tests? . . . . .	8
3.1.2	Schritte und Techniken, Tests aufzusetzen . . . . .	9
3.1.3	Test-Driven Development . . . . .	9
3.2	Trennen von Verantwortungen - Dependency-Breaking . . . . .	10
3.2.1	Ein Interface extrahieren . . . . .	11
3.2.2	Einen Parameter zum Konstruktor hinzufügen . . . . .	12
3.2.3	Einen Parameter zu einer Methode hinzufügen . . . . .	12
3.2.4	Einen Aufruf extrahieren und überschreiben . . . . .	12
3.3	Arbeiten in altem Code . . . . .	13
3.3.1	Verstehen von Code . . . . .	13
3.3.2	Hinzufügen von Features . . . . .	14
<b>4</b>	<b>Welche Tools gibt es zum Refactoring von Legacy Code?</b>	<b>17</b>
4.1	IDE-interne Tools . . . . .	17
4.1.1	Rename . . . . .	17
4.1.2	Change Method Signature . . . . .	17
4.1.3	Change Method Signature . . . . .	17
4.1.4	Extract Local Variable bzw. Extract Constant . . . . .	18
4.1.5	Extract Method . . . . .	18
4.1.6	Move . . . . .	18
4.1.7	Inline . . . . .	18
4.1.8	Extract Superclass . . . . .	18
4.1.9	Pull Up / Push Down . . . . .	19
4.1.10	Encapsulate Field . . . . .	19
4.2	Externe Tools . . . . .	19
4.2.1	PMD . . . . .	19
4.2.2	Structure 101 . . . . .	19
<b>5</b>	<b>Zusammenfassung</b>	<b>21</b>
<b>6</b>	<b>Quellen</b>	<b>22</b>

# 1 Einleitung

Üblicherweise wird nicht der gesamte Code eines Projektes von einem selbst geschrieben. Wir übernehmen Code aus Bibliotheken und anderen Projekten, oder die Arbeit findet in einem schon bestehenden Projekt statt.

In solchen Situationen arbeitet man meistens mit „*Legacy Code*“.

Doch was genau zeichnet *Legacy Code* aus? Wie geht man mit *Legacy Code* um, wie kann man ihn verbessern und verändern?

Diese Fragen soll diese Ausarbeitung beantworten und gibt zudem Beispiele für Techniken und automatisierte Tools, welche einem die Lösung dieser Aufgabe ermöglichen. Als Grundlage dieser Arbeit dient hauptsächlich das Buch „Working effectively with Legacy Code“ von Michael C. Feathers aus dem Jahre 2009 [Fea09], in welchem er ausgiebig verschiedene Refactoring-Methoden erläutert und zudem wichtige Begriffe rund um das Thema *Legacy Code* definiert.

## 2 Grundlagen des Phänomens Legacy Code

Legacy-Code ist im Allgemeinen veralteter Code, welcher oft von einer anderen Person geschrieben wurde und mit welchem man weiterarbeitet.

„If you are at all like me, you think of tangled, unintelligible structure, code that you have to change but don't really understand. You think of sleepless nights trying to add in features that should be easy to add, and you think of demoralization, the sense that everyone on the team is so sick of a code base that it seems beyond care, the sort of code that you just wish would die. Part of you feels bad for even thinking about making it better. It seems unworthy of your efforts.“ [Fea09, S. XVI]

Genauer gesagt ist Legacy-Code also schwer veränderbarer Code, da wir ihn nicht verstehen, was wir aber müssen, um ihn zu überarbeiten und weiter mit ihm zu arbeiten.

### 2.1 Merkmale und Probleme von Legacy Code

(Verwendete Quellen für diesen Abschnitt: [Kai14], [Mic02], [Wik16a])

Doch was macht Legacy-Code so schwer verständlich? Welche Fehler können beim Programmieren gemacht werden, um Kollegen oder Nachfolgern schlaflose Nächte zu bereiten?

Als eines der wichtigen Kriterien gilt vor allem das Fehlen von Tests. Ohne Tests kann nicht gut nachvollzogen werden, was welche Funktion oder Klasse macht und ob sie überhaupt funktioniert.

Zudem ist ein weiteres Kriterium, dass der Quellcode nicht ausreichend mit Kommentaren dokumentiert wurde. Gerade an Stellen, an denen komplizierte Methoden oder Funktionen nicht wegzulassen sind, erschwert dies die Nachvollziehbarkeit enorm.

Außerdem finden sich in Legacy Code oft unnötig komplizierte Stellen (tief verschachtelte if-Anweisungen oder schlecht definierte while-Schleifen), welche zudem oft zusammen mit totem Code (z.B. ungenutzte Variablen) in Erscheinung treten. Auch die Architektur an sich ist ein weiteres Kriterium. Bestehen zu viele Abhängigkeiten zwischen Elementen im Projekt (z.B. zwischen Packages), kann dies später die Weiterarbeit erschweren (siehe 4.2 „Externe Tools“ (Seite 19)). Legacy Code kann aber auch ganz einfach Code sein, welcher in einer sehr alten oder nicht weit verbreiteten Sprache geschrieben wurde.

Was Legacy Code auf der anderen Seite auch ausmacht ist die Tatsache, dass er besonders wertvoll ist. Sonst würde sich die Frage stellen, warum man sich mit Code, welcher derart viele Probleme beherbergen kann, so zeitintensiv auseinandersetzt (siehe Kapitel 3 „Einführung ins Refactoring“ (Seite 8)). Was Legacy Code so wertvoll machen kann, wird im nachfolgenden Abschnitt (2.2 „Gründe für das Arbeiten mit Legacy Code“ (Seite 7)) erklärt.

## 2.2 Gründe für das Arbeiten mit Legacy Code

Nachdem man sich die Liste an Merkmalen von Legacy Code angesehen hat, kann man sich die Frage stellen, warum man nicht einfach einen komplett neuen Code schreibt.

Der Grund dafür ist einfach, dass der bestehende Code seine Aufgaben erfüllt und das System funktioniert. Wenn er einmal verstanden und strukturiert ist, kann er extrem flexibel und somit äußerst wertvoll sein.

In der Regel will man kein komplett neues Projekt, welches sich schnell als fehleranfällig erweisen kann. Im Normalfall möchte man das alte Projekt einfach verändern und verbessern.

Zum Beispiel funktioniert das Programm, doch jemand wünscht sich ein zusätzliches Feature. Muss aber dafür das Programm komplett neu geschrieben werden? Wenn das Programm erstmal verstanden und verbessert wurde, ist der Vorgang des Hinzufügens in Zukunft deutlich kürzer als das Schreiben einer neuen Software.

Oder aber das Verhalten der Software soll beibehalten werden, doch die Struktur des Quellcodes soll wartungsfreundlicher gemacht werden. Wenn dies bei einer funktionierenden Software erfolgt, können auch Kollegen oder Nachfolger erfolgreich mit der Software weiter arbeiten.

Diese Ziele kann man durch das sogenannte **Refactoring** erreichen. Wie genau dies funktioniert und welche Tools dabei zum Einsatz kommen können, wird in den folgenden Kapiteln genauer erläutert.

## 3 Code überarbeiten: Einführung ins Refactoring

Refactoring von Code ist, einfach gesagt, der Prozess, Code zu verbessern, ohne das Verhalten zu ändern. Wie kann man solchen alten Code besser machen? Das scheint die Hauptfrage zu sein, wenn man sich mit altem Code befasst. Man möchte ihn entweder strukturell klarer machen, Features hinzufügen oder sonstige Verbesserungen vornehmen.

Während dies in Kürze behandelt wird, betrachten wir als ersten Schritt des Refactorings das Testen. Tests sind nicht spannend, nicht etwas, was man gerne schreibt. Trotzdem muss man mit dem Testen anfangen, um den Prozess des Refactorings zu beschreiben.

Sobald man die wichtigen Teile seines Systems, die man verbessern will, unter Tests stehen hat, kann die eigentliche Arbeit des Refactorings beginnen.

### 3.1 Test-gesteuertes Entwickeln: Unit Tests

„Unit Tests“ sind, einfach gesagt, Überprüfungen, ob einzelne Teile oder Komponenten unseres Gesamtsystems ihre Aufgabe korrekt erfüllen. Diese Überprüfungen laufen in der Regel automatisch ab, d.h. man schreibt sie einmal und kann sie beliebig oft laufen lassen. Wichtig ist dabei auch, dass diese Komponenten in Isolation laufen sollten - daher das „Unit“ (zu Deutsch: „Einheit“) im „Unit Test“.

#### 3.1.1 Wofür braucht man Tests?

Tests stellen sicher, dass man beim Ändern von Code nicht das Verhalten ändert. Tests sind gewissermaßen ein „Schraubstock“, welcher um das Verhalten unserer Programmkomponenten angelegt wird. Falls wir beim Verändern des Codes, sei dies Refactoring zum Verbessern der Struktur oder Hinzufügen von Funktionalitäten, auch das Verhalten verändern, werden Tests uns sagen, wo und wie viele Fehler wir gemacht haben. Des weiteren benötigen Komponenten oft weitere Komponenten zum Funktionieren, und so wäre das Auffinden eines Fehlers ohne eine Lokalisation durch Tests eine schwere Aufgabe.

[Übersetzung Feathers]

Hier sind die Qualitäten guter Unit Tests:

1. Sie laufen schnell
2. Sie helfen uns, Probleme zu lokalisieren

[...] Ein UnitTest, der 0,1 Sekunde dauert, ist ein langsamer UnitTest.

[Fea09, S. 13]

Wie oben beschrieben ist eine weitere Eigenschaft von UnitTests, dass sie isoliert laufen. UnitTests sollen nie über ein Netzwerk kommunizieren, auf ein Dateisystem zugreifen oder mit einer Datenbank in Verbindung treten.



### 3.1.2 Schritte und Techniken, Tests aufzusetzen

Wenn man einen UnitTest mit einem automatisierten Test-Harness wie JUnit, CppUnit, oder weiteren Frameworks aufsetzen will, muss man die Komponenten zunächst in Isolation laufen lassen. Dabei ergibt sich oft das Problem, dass diese Komponenten nicht in Isolation funktionieren: Sie sind abhängig von weiteren Komponenten, externen Bibliotheken, Datenbanken oder spezieller Hardware.

Falls dies der Fall ist, muss man zunächst die Abhängigkeiten aufbrechen. Genaue Techniken dazu werden im Abschnitt 3.2 „Trennen von Verantwortungen - Dependency-Breaking“ (S. 10) besprochen.

Sobald man seine Klassen und Methoden so weit geordnet hat, dass alle problematischen Abhängigkeiten und Zugriffe auf externe Ressourcen so abgegrenzt sind, dass man diese durch Techniken des Dependency-Breakings aus dem Test entfernen kann, schreibt man die eigentlichen Tests.

Die meisten Test-Frameworks bieten eine Handvoll von Funktionen, um einen Test für eine Klasse aufzusetzen:

- Setup: Benötigte Objekte aufsetzen.
- Method Tests: Öffentliche Methoden testen.
- Teardown: Meist optional, nach dem Test nicht benötigte Objekte zerstören.

Bei einem Testfall bereitet man ein erwartetes Ergebnis vor, führt die Methode aus und vergleicht die tatsächlichen Ergebnisse mit den erwarteten. Hierbei wird der Erfolg des Tests mit einem sogenannten „*assert*“ der Testumgebung mitgeteilt.

Als Faustregel sollte man in einer Klasse alle Methoden testen, welche folgenden Regeln entsprechen:

- Sie sind öffentlich (public)
- Sie verrichten Arbeit (d.h. sie sind nicht nur Getter oder Setter)
- Sie bieten nicht ausschließlich einen Zugriff auf eine Bibliothek oder Ressource.

Jede Methode, die diesen Regeln entspricht, sollte unter mindestens einen Test gestellt werden. *Mindestens* bedeutet hierbei, dass man auch mehrere Tests für eine Methode schreiben kann - je nachdem, wie viele verschiedene Arten von Ergebnis man von der Methode erwartet.

### 3.1.3 Test-Driven Development

Im Gegensatz zum Schreiben neuer Tests für alte Methoden sollte man bei neu geschriebenen Methoden das sogenannte „Test-Driven Development“ anwenden. Der Kernpunkt hierbei ist, dass man Tests schreibt, bevor die Funktionalität der Methode implementiert wird. So wird das Implementieren der Methode durch den Test gesteuert.

Der „Alogrithmus“ des Test-Driven Development wird von Feathers wie folgt beschrieben:

”Test-driven development uses a little algorithm that goes like this:

1. Write a failing test case.
2. Get it to compile.
3. Make it pass.
4. Remove duplication.
5. Repeat.”

[Fea09, S. 88]

Anders gesagt schreibt man zuerst einen Test zu einer Methode, welche nicht existiert. Dieser Test wird natürlich nicht compilieren, also schreibt man eine Methode welche leer ist, aber dafür sorgt, dass das Projekt überhaupt läuft. Im Folgenden findet der Hauptteil der Arbeit statt, das Implementieren. Fertig mit der Implementation ist man, sobald die Ausführung des Tests erfolgreich verläuft. Implementiert man mehr als eine Methode auf einmal durch Test-Driven Development, kann es oft vorkommen, dass Code in mehreren Methoden ähnlich aussieht oder sogar redundant ist. Ist dies der Fall, kann man eine weitere, meist private Methode schreiben, welche die Redundanz auflöst und von den anderen Methoden verwendet wird. Selbst wenn man bei diesen Schritten einen Fehler macht, wird dies keine negativen Konsequenzen haben - der Test wird einfach fehlschlagen und man kann weiterentwickeln, bis alles korrekt abläuft.

## 3.2 Trennen von Verantwortungen - Dependency-Breaking

Dependencies, zu Deutsch „Abhängigkeiten“, bezeichnen die Situation, wenn eine Komponente eines Systems nicht allein existieren kann, sondern weitere Komponenten, externe Hardware oder Verbindungen zu externen Geräten oder Services braucht. Solche Abhängigkeiten sind eines der größten Probleme, wenn man mit Legacy Code arbeitet. Tests sind schwerer aufzusetzen, da man für eigentlich isolierte Teile des Systems, für die man einen Test schreiben will, weitere Komponenten instantiieren oder verwenden muss. Diese Komponenten können einige Probleme mit sich bringen:

- Sie sind nicht einfach zu erzeugen, weil sie wiederum weitere Komponenten benötigen. So ergibt sich eine Kette, deren Länge schwer abschätzbar ist. Dies wirkt sich auch negativ auf die Ausführungszeit der Tests aus, welche wie im Abschnitt 3.1.1 „Wofür braucht man Tests?“ (S. 8) so schnell wie möglich laufen sollen.
- Sie können versteckte Zugriffe auf andere Teile des Systems beinhalten, was nicht im Sinne eines UnitTests ist.
- Sie können integrierter Teil einer externen Ressource sein und somit nicht frei für den Test zur Verfügung stehen.

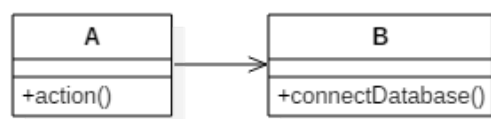
Im Folgenden werden einige der allgemeinen Techniken aufgelistet, um Abhängigkeiten von Komponenten (in objektorientierten Sprachen also Klassen und Objekten) innerhalb eines Systems sowie Abhängigkeiten zu externen Ressourcen aufzubrechen. Es gibt viel zu viele Strategien, um alle hier zu nennen. Daher werden hier die gebräuchlichsten und einfachsten Techniken besprochen.

### 3.2.1 Ein Interface extrahieren

Diese Technik ist einer der sichersten und einfachsten Wege, Abhängigkeiten aufzubrechen. Interfaces sind, einfach gesagt, Klassen ohne Attribute und mit ausschließlich leeren Methoden. Sie sind den abstrakten Klassen ähnlich, aber haben einen konzeptuellen Unterschied:

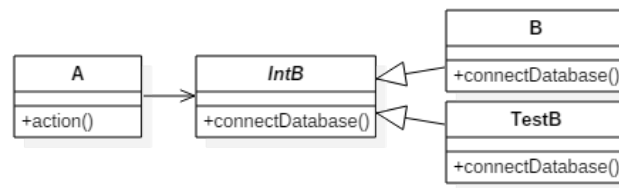
- **Abstrakte Klassen** sind ein gemeinsamer Kern für eine Familie von Objekten. Als Beispiel kann man sich die abstrakte Klasse *Fortbewegungsmittel* denken, von der die Klassen *Auto*, *Boot* und *Flugzeug* erben.
- **Interfaces** sind eine gemeinsame Art, auf Objekte verschiedenster Klassen zuzugreifen. Als Beispiel kann man sich das Interface *Photographierbar* denken, welches die Klassen *Auto*, *Mikrowelle* und *Hängematte* implementieren.

Wie kann man Interfaces nun verwenden, um Abhängigkeiten aufzubrechen? Hat man eine *Klasse A*, welche unter einen Test gestellt werden soll, welche zur Instantiierung ein Objekt einer *Klasse B* benötigt, so scheint der Test einfach: Erzeuge ein Objekt der *Klasse B* und gebe dies an den Konstruktor weiter, welcher das Objekt der *Klasse A* erzeugt, das anschließend getestet werden soll. Ein Problem, das hierbei auftreten kann, ist, dass eine Methode in *Klasse B*, welche von *Klasse A* verwendet wird, eine Verbindung zu einer Datenbank aufbauen kann oder wiederum eine Menge weiterer problematischer Objekte braucht, um instantiiert zu werden.



Was man nun tun kann ist, ein Interface zu schreiben, welches die Methoden aus *Klasse B* enthält (genannt *IntB*). Nun lässt man *Klasse B* dieses mit den existierenden Methoden implementieren und ändert schließlich den Konstruktor von *Klasse A* so ab, dass er nicht mehr ein Objekt der *Klasse B* erwartet, sondern eine Implementierung des eben erstellten Interfaces *IntB*.

Nun kann man *IntB* von einer weiteren Klasse implementieren lassen, in unserem Fall zum Beispiel *TestB*, welches für die Funktion von *Klasse A* notwendige Ein- und Ausgaben zur Verfügung stellt, die keine Datenbank oder sonstige externe Ressource benötigen.



### 3.2.2 Einen Parameter zum Konstruktor hinzufügen

Oft passiert es im Code, dass der Konstruktor einer Klasse Objekte weiterer Klassen erzeugt, welche für das konstruierte Objekt benötigt werden. Dies bringt das Problem mit sich, dass die erzeugten Objekte festgelegt sind und nicht für Tests ausgetauscht werden können.

Die Lösung ist relativ simple: Man erzeugt das benötigte Objekt nicht neu im Konstruktor, sondern übergibt es als Parameter und weist es der Instanzvariable zu.

Dadurch ergibt sich aber ein weiteres Problem: Alle Klassen, die diesen Konstruktor verwenden, werden ihn nicht mehr verwenden können, da sich die Signatur des Konstruktors geändert hat. Glücklicherweise gibt es eine Lösung: In Sprachen wie Java oder C# fügt man einen weiteren Konstruktor mit derselben Signatur wie der vorherige Konstruktor hinzu und ruft von dort aus den parameterisierten Konstruktor mit einem wie vorher instantiierten benötigten Objekt auf. Falls die verwendete Sprache keinen gegenseitigen Aufruf von Konstruktoren erlaubt (wie zum Beispiel C++), muss man den gesamten Konstruktor kopieren und den Parameter dort hinzufügen.

Auf diese Art und Weise können alle Klienten der Klasse diese weiterhin wie gehabt verwenden, aber man hat auch einen Ansatzpunkt, Testobjekte in der Klasse zu verwenden.

Einen Nachteil hat diese Technik aber doch: Im endgültigen Code steht der Konstruktor mit zusätzlichem Parameter ebenfalls zur Verfügung und kann verwendet werden, um noch verworrenere Abhängigkeiten zu schaffen. Dies ist ein kleines Risiko, sollte aber nicht vernachlässigt werden.

### 3.2.3 Einen Parameter zu einer Methode hinzufügen

Ähnlich wie den Konstruktor kann man in den meisten modernen Sprachen auch Methoden überladen. So kann man auch Methoden, die neue Objekte erzeugen, mit denselben Schritten wie in 3.2.2 „Einen Parameter zum Konstruktor hinzufügen“ einen zusätzlichen Parameter geben.

### 3.2.4 Einen Aufruf extrahieren und überschreiben

Fast immer ruft eine Methode, welche man testen möchte, weitere Methoden auf. Eine solche aufgerufene Methode muss nicht einmal in der selben Klasse oder Instanzmethode eines Objektes sein, es kann eine statische Methode einer anderen Klasse oder eine globale Variable (ggf. durch Singleton-Pattern) sein.

Sollte dies der Fall sein und wir können aufgrund dieses Aufrufes keinen Test schreiben, ist es möglich, diesen Aufruf zu extrahieren. Hierzu nimmt man sich den Aufruf der in Frage stehenden Methode und verschiebt ihn in eine weitere Methode innerhalb derselben Klasse. Als Zugriffsberechtigung sollte man hierbei „protected“ einstellen.

Hat man nun diesen Aufruf extrahiert ist es möglich, die zu testende Methode in einer vererbenden Testklasse laufen zu lassen, welche lediglich den Aufruf in der extrahierten Methode durch einen Platzhalter oder ähnliche, testbare Aufrufe ersetzt.

### 3.3 Arbeiten in altem Code

Nachdem man Tests aufgesetzt hat, um sicher sein zu können, dass Änderungen am Code nicht das Verhalten ändern, kann die eigentliche Arbeit beginnen: Man beginnt, den Code zu verbessern. Sei es strukturell, für höhere Effizienz oder zum Hinzufügen eines Features.

So gilt es nun, den Code zu verstehen, Tests zu schreiben, den Code zu verbessern und Features hinzuzufügen.

#### 3.3.1 Verstehen von Code

Oft passiert es, dass man vor Code sitzt, den man nicht versteht, mit dem man aber arbeiten muss. Die Dokumentation lässt zu wünschen übrig und es sind keine Tests angelegt. Wie kann man vorgehen, um unverständlichen Code zu verstehen? Im Folgenden werden einige Tipps aufgelistet, um sich in Legacy Code zurecht zu finden.

Die hier angesprochenen Vorgehensweisen orientieren sich an Michael Feather's Arbeit in „Working effectively with Legacy Code“ [Fea09].

**Skizzenhafte Notizen** Oft hilft es, wenn man sich schlichtweg mit einem Stück Papier und einem Bleistift, optional noch mit einem Kollegen, einige Notizen über den Code macht, um welchen es geht. Dies muss kein Text sein, es muss nicht der UML entsprechen - es können einfach nur abstrakte Rechtecke, Namen und Pfeile sein. Ein Beispiel für eine solche Skizze ist Abbildung 1 auf Seite 14: Es folgt keiner Norm, sondern ist eine spontane Visualisierung des Systems. Auch, wenn diese Skizzen im Nachhinein nicht mehr nützlich sind, helfen sie im Moment, in dem sie angefertigt werden, beim Verstehen des Systems weiter.

**Code Ausdrucken** Vor allem nützlich bei übermäßig langen Methoden ist es, sich das Listing des Quellcodes auszudrucken und dann darin herumzuzeichnen. Man kann Linien ziehen, um sich Abhängigkeiten oder Aufrufe zu verdeutlichen, man kann Teile umkreisen, um sich zu markieren, wo man Methoden extrahieren will, und man kann Linien zwischen Zeilen ziehen, um Stellen zu markieren, an denen Verantwortungen getrennt werden sollten.

Es ist nicht festgeschrieben, was man wie zeichnet - solange man eine bessere Idee bekommt, wie der Code funktioniert.

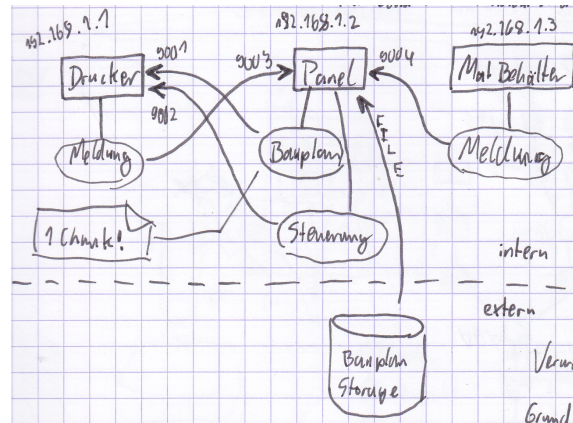


Abbildung 1: Beispiel für eine skizzenhafte Notiz

**Scratch Refactoring** Dies ist eine etwas bizarre, aber effektive Möglichkeit, ungetesteten Code zu verstehen. Man legt sich eine Kopie des zu verstehenden Codes an (z.B. durch Auschecken von einem Versionskontrollsystem) und beginnt, Code herumzubewegen und umzuschreiben. Der springende Punkt ist, dass die Kopie danach **verworfen** wird - keinesfalls den ungetestet modifizierten Code wieder einchecken! Es mag nach Verschwendung klingen, an Code zu arbeiten und diese Arbeit dann einfach zu verwerfen. Der eigentliche Gewinn liegt aber nicht in der Arbeit, die man an dem Code angewendet hat, sondern an dem gewonnenen Verständnis für das System.

### 3.3.2 Hinzufügen von Features

Im Nachfolgenden werden selbst erstellte Graphiken verwendet, um die Vorgänge einfacher darzustellen. Die Legende für diese Darstellungen ist in Abbildung 2 auf Seite 14 zu sehen.

Legende:



Abbildung 2: Legende für das Hinzufügen von Features

Wie kann man nun ein Feature zum bisherigen Code hinzufügen? Die einfachste Möglichkeit wäre natürlich, die neue Funktionalität in dieselbe Methode wie die bisherige Funktionalität zu schreiben. Dies ist jedoch eine ausdrücklich schlechte Vorgehensweise, da hiermit die Klarheit der Methode abnimmt und der Code dieser Methode keine eindeutige Verantwortungstrennung mehr hat. Zu sehen ist dies in Abbildung 3 auf Seite 15.

Problem :

Schlechte Vorgehensweise :

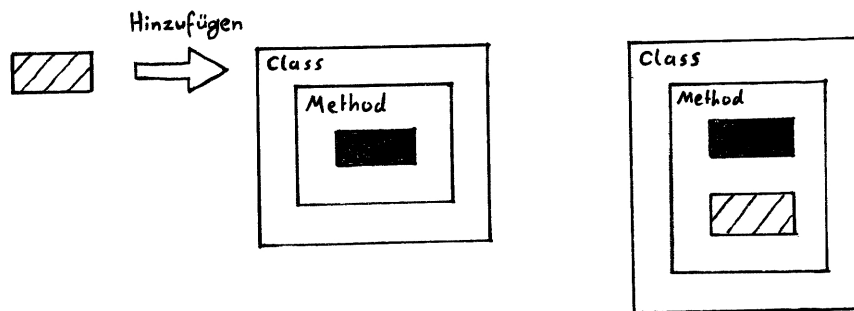


Abbildung 3: Schlechte Vorgehensweise

Im Folgenden werden einige bessere Vorgehensweisen besprochen, welche auf der Arbeit von Michael Feather's „Working Effectively with Legacy Code“ [Fea09] basieren. Hierbei ist anzumerken, dass alle neu hinzukommenden Methoden nach dem Prinzip des 3.1.3 „Test-Driven Developments“ (Seite 9) geschrieben werden sollten.

**Sprout** Eine Möglichkeit zum strukturierten Hinzufügen von Funktionalität ist eine „Sprout Methode“ bzw. eine „Sprout Klasse“. Hierbei wird eine neue Methode bzw. eine neue Klasse hinzugefügt, welche die neue Funktionalität enthält, und anschließend von der alten Methode aus diese neue Funktionalität aufgerufen. Siehe Abbildung 4 auf Seite 15.

SPROUT :

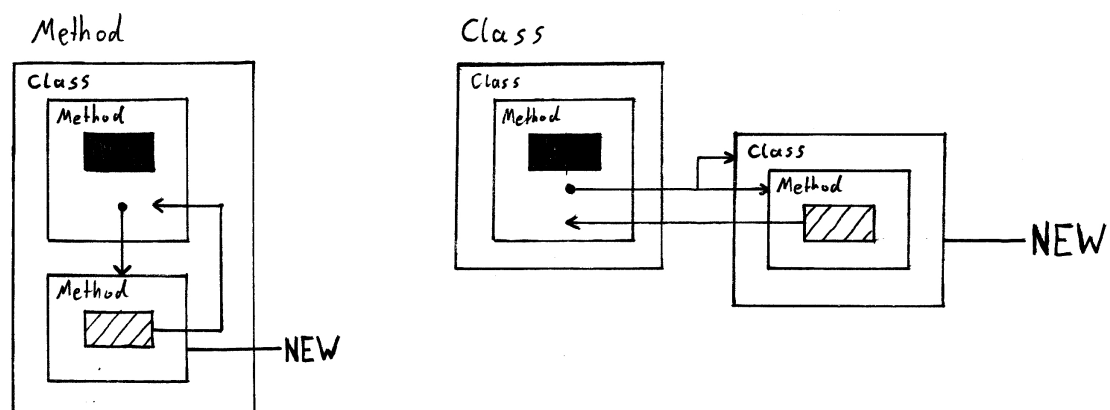


Abbildung 4: Sprout Methode und Sprout Klasse

WRAP :

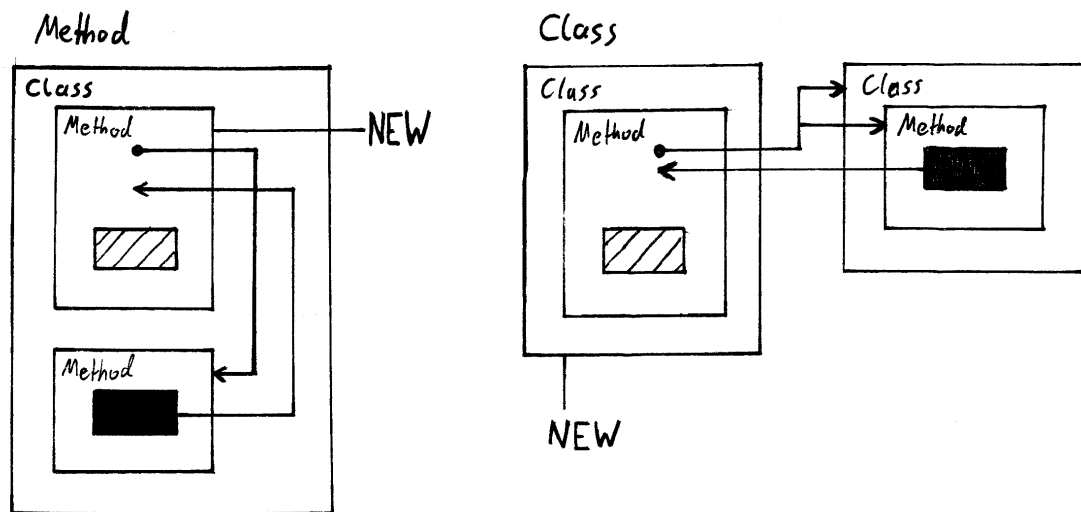


Abbildung 5: Wrap Methode und Wrap Klasse

**Wrap** Eine weitere Möglichkeit zum strukturierten Hinzufügen von Funktionalität ist eine „Wrap Methode“ bzw. eine „Wrap Klasse“. Hierbei wird, ähnlich wie bei einem Sprout, eine neue Methode bzw. eine neue Klasse hinzugefügt, welche die neue Funktionalität enthält. Anders als beim Sprout wird jedoch die alte Funktionalität von der neuen Funktionalität aufgerufen - die Hierarchie der Aufrufe ist also umgekehrt. Siehe Abbildung 5 auf Seite 16.

Ein weiteres Merkmal von Sprout Klassen insbesondere ist, dass man ggf. aus mehreren solcher Klassen eine abstrakte Elternklasse extrahieren kann und ein sogenanntes „Decorator Pattern“ anwenden kann, in welchem eine Vielzahl von Funktionalitäten, welche sich Parameter teilen, ineinander verschachtelt sind.

**Entscheidung: Wrap oder Sprout?** Zur Frage, ob man einen Wrap oder Sprout verwenden sollte, gibt es leider keine eindeutige Antwort - es kommt immer auf die gegebene Situation an. Macht es mehr Sinn, eine Funktion an einem bestimmten Punkt in alten Code einzufügen, oder ist es besser, den alten Code als Teil eines nun größeren Kontextes anzusehen?

Die Entscheidung hängt immer von dem aktuellen Problem ab, mit welchem man sich gerade befasst.



## 4 Welche Tools gibt es zum Refactoring von Legacy Code?

In der modernen Softwareentwicklung muss man nicht immer alles manuell umschreiben. Das Gute an automatisierten Tools zum Refactoring ist nicht nur, dass sie es uns bequemer machen, Änderungen an mehreren Stellen vorzunehmen, sondern auch die „menschlichen Fehler“, welche jeder von uns macht, zu verhindern. Natürlich sind Refactoring-Tools selbst auch von Menschen geschrieben und können somit selbst keine Fehlerfreiheit garantieren - sie sind mit Vorsicht zu genießen. Im folgenden Abschnitt werden einige Arten und Beispiele von Refactoring-Tools näher erläutert.

### 4.1 IDE-Interne Tools

IDEs wie *NetBeans*, *Eclipse* und *Android Studio* stellen standardmäßig eine Anzahl von Refactoring-Tools zur Verfügung, welche dem Programmierer eine Menge fehleranfälliger Schreiarbeit abnehmen.

Nachfolgend werden einige der wichtigsten Tools vorgestellt, es existieren aber deutlich mehr. Um den Überblick zu behalten, und, da sich die meisten bekannten Editoren in vielen Hinsichten ähneln, basieren die folgenden Erläuterungen auf einem einzelnen Editor: *Eclipse*. Als Quelle hierzu diene das Eclipse-Wiki der Fernuni Hagen [Fer16].

#### 4.1.1 Rename

Jegliche Variablen, Klassen, Methoden, Packages, Projekte etc. lassen sich umbenennen. Dies kann vor allem zur besseren Verständlichkeit von komplexem und/oder schlecht geführtem Quellcode führen.

Ein einfaches Beispiel wäre, dass der Code viele Variablen mit dem Namen „tmp“ oder ähnlichen, wenig aussagenden Objektnamen enthält.

#### 4.1.2 Change Method Signature

Dieses Tool kann jegliche Parameter einer Methode ändern und zudem den Access-Modifier ändern. So können Methoden aktualisiert werden, ohne große Eingriffe in den Quellcode an sich vornehmen zu müssen.

#### 4.1.3 Change Method Signature

Dieses Tool kann jegliche Parameter einer Methode ändern und zudem den Access-Modifier neu setzen. So können Methoden aktualisiert werden, ohne große Eingriffe in den Quellcode an sich vornehmen zu müssen.

#### 4.1.4 Extract Local Variable bzw. Extract Constant

Beliebige Ausdrücke (zum Beispiel eine Subtraktion) können einer lokalen Variable bzw. Feld-Konstante zugeordnet werden. So ist es auf leichte Weise möglich, einen komplizierten Ausdruck wie zum Beispiel `2 + 4 * 3 / 2 + Variable1 % 5` in eine Variable zu extrahieren. Der eben genannte Ausdruck kann so zum Beispiel einfach in eine neue Variable (z.B. `Variable2`) gespeichert werden.

#### 4.1.5 Extract Method

Hierdurch kann ein markierter Codeblock in eine eigene Methode ausgelagert werden. Zudem werden die benötigten Parameter des markierten Codeblocks als Methoden-Parameter deklariert. Alle Vorkommnisse des Codeblocks werden durch den Aufruf der neuen Methode ersetzt.

Dieses automatisierte Refactoring-Tool kann eingesetzt werden, um eine Abhängigkeit mittels 3.2.4 „Einen Aufruf extrahieren und überschreiben“ automatisiert durchzuführen.

#### 4.1.6 Move

Sowohl Klassen als auch Methoden können in andere Projekte verschoben werden (Methoden zudem in andere Klassen des gleichen Projekts). Bei „nicht immer sinnvollen Aktionen“, wie zum Beispiel das Verschieben eines Konstruktors in eine andere Klasse, gibt Eclipse zumindest eine Warnung aus - in manchen Fällen kann die Aktion sogar verweigert werden.

Eine Stärke dieses Tools ist, dass Abhängigkeiten vom bewegten Code, wie Nutzung von Methoden oder Imports vom Klassen, automatisch erkannt und wenn möglich auch automatisch angepasst werden.

#### 4.1.7 Inline

Fügt den Wert einer Variable anstatt der Variable an sich ein. Kleines Beispiel:

```
int wert1 = 2 + 2;  
int wert2 = wert1 + 3;
```

wird zu

```
int wert2 = 2 + 2 + 3;
```

#### 4.1.8 Extract Superclass

Hierdurch kann eine alte Oberklasse erweitert werden, indem man für eine markierte Klasse eine Oberklasse erzeugt. Passende Konstruktoren werden zudem automatisch für die neue Oberklasse erzeugt. Methoden und Felder lassen sich auswählen und können somit in die neue Oberklasse übernommen werden.

#### 4.1.9 Pull Up / Push Down

Hiermit können Variablen und Felder von einer Subklasse in die Oberklasse verschoben werden (Pull Up) bzw. umgekehrt von einer Oberklasse in die Subklasse verschoben werden (Push Down).

#### 4.1.10 Encapsulate Field

Dieses Tool erzeugt für Variablen einer Klasse Getter- und Setter-Methoden.

### 4.2 Externe Tools

(Verwendete Quellen für diesen Abschnitt: [QA 14], [Kai14], [Wik16b] )

Auch außerhalb von Editoren, wie *Eclipse* oder *NetBeans*, gibt es zahlreiche Möglichkeiten, Quellcode zu analysieren, um ihn gegebenenfalls anschließend zu modifizieren. Oft sind diese externen Tools besser auf einen bestimmten Bereich des Refactorings spezialisiert.

Im folgenden Kapitel werden *PMD* (der Name hat keine Bedeutung), welches vordefiniert und meistens innerhalb einer Klasse oder Methode verwendet wird, und *Structure 101*, welches die Architektur und Struktur an sich analysiert, vorgestellt.

#### 4.2.1 PMD

*PMD* ist ein plattformunabhängiges Werkzeug, welches *Java*, *JavaScript*, *XML*, *XLS* und davon abgeleitete Dialekte unterstützt. *PMD* sucht an sich keine echten Fehler, sondern hilft, ineffizienten Code zu verbessern.

Unter Anderem sucht es leere try/catch/switch- Blöcke, welche möglicherweise Bugs verursachen. Außerdem macht es ungenutzte lokale Variablen, Parameter und private Methoden ausfindig.

Neben den genannten Methoden, welche toten Code ausfindig machen, ist das Werkzeug in der Lage dazu, unnötig komplizierte Ausdrücke und Funktionen (unnötige bzw. schlecht definierte Schleifen oder Bedingungen) zu vereinfachen.

Neben diesen festgelegten Methoden und Funktionalitäten des Tools ist es auch möglich, diesen Satz an Regeln zu erweitern und an seine Bedürfnisse anzupassen.

#### 4.2.2 Structure 101

*Structure 101* ist ein Werkzeug, welches sich im Vergleich zu *PMD* den Code/das Package/das Projekt im Gesamtbild anguckt und analysiert.

Abhängigkeiten und die Komplexität von Strukturen werden dann in Grafiken und Modellen veranschaulicht und eingegrenzt. Mithilfe dieser Grafiken unterstützt es den Nutzer dabei unstrukturierten Code zu entwirren und gibt Vorschläge auf Grundlage von allgemeinen Vorgaben.

Berechnet wird die Komplexität mit zwei Metriken:

- Zum einen gibt es den sog. ***Fat***-Wert. Dieser basiert auf der Anzahl der Kanten im Abhängigkeitsgraph, welcher auf verschiedene Ebenen eingestellt werden kann. So kann man die Abhängigkeit von der Methode bis zum Package ermitteln.
- Zum anderen gibt es sogenannte ***Design Tangles***. Diese sind zyklische Abhängigkeiten auf Package-Ebene, welche Entwicklung, Test und Release einzelner Packages erschweren können.

Zudem gibt es bei *Structure 101* die Möglichkeit, bestimmte Kriterien und Vorgaben bezüglich der Architektur zu definieren und diese dann mit der eigenen Implementierung zu vergleichen.

## 5 Zusammenfassung

Zusammengefasst kann man einige Schlüsse zu Legacy Code und dessen Refactoring ziehen.

**Grundlagen** Legacy-Code ist schwer veränderbarer Code, den man nicht versteht, was man aber muss, um weiter mit ihm zu arbeiten. Ihn zeichnet vor allem eine schlechte Struktur aus, welche sich zum Beispiel durch unnötig komplizierte Abschnitte oder einfach durch das Fehlen von Tests bemerkbar macht.

Wenn der Code aber verstanden und verbessert wurde, ist er äußerst flexibel und somit wertvoll.

**Einführung ins Refactoring** Wenn man einen Teil seines Systems bearbeiten will (sei es, um ihn effizienter zu machen, Features hinzuzufügen oder die Struktur zu verbessern), so nennt man diesen Vorgang „Refactoring“.

Beim Refactoring ist einer der wichtigsten Faktoren, dass es Tests für die Klassen gibt, welche man verändern will. Sollte dies nicht der Fall sein, sollte man diese stets schreiben, bevor man beim Code Hand anlegt. Sollte dies nicht möglich sein, so sollte man Techniken anwenden, um diese Abhängigkeiten aufzubrechen (3.2, „Trennen von Verantwortungen - Dependency-Breaking“ Seite 10).

**Refactoring Tools** Die meisten Editoren wie zum Beispiel *Eclipse* stellen intern eine Reihe wichtiger Refactoring-Tools zur Verfügung.

Diese reichen von relativ simplen Methoden wie *Rename*, bis hin zu komplizierteren Methoden, wie zum Beispiel *Extract Superclass*.

Sie erleichtern uns die Arbeit, sind aber immer mit Vorsicht zu genießen, da sie nicht immer ein einhundert Prozent korrektes Refactoring garantieren können.

Auch außerhalb von Editoren gibt es Software, welche den Nutzer bei diversen Problemen unterstützen kann. Es gibt sowohl Tools, welche ineffiziente Stellen anhand von vordefinierten Regeln innerhalb von Methoden oder Klassen ausfindig machen (*PMD*), als auch Tools, welche die Struktur des kompletten Codes im Gesamtbild analysieren. (*Structure 101*).

## 6 Quellen

### Literatur

- [Fea09] Michael C. Feathers. *Working effectively with legacy code*. 10. print. Robert C. Martin series. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2009. ISBN: 978-0-13-117705-5 (siehe S. 5 f., 8, 10, 13, 15).
- [Fer16] Fernuni Hagen, Hrsg. *Refactoring-Tools in Eclipse – Eclipse*. 10.05.2016. URL: [https://wiki.fernuni-hagen.de/eclipse/index.php/Refactoring-Tools\\_in\\_Eclipse](https://wiki.fernuni-hagen.de/eclipse/index.php/Refactoring-Tools_in_Eclipse) (siehe S. 17).
- [Kai14] Kai Spichale. *Kontinuierliche Architekturvalidierung: Auf die inneren Werte kommt es an*. Hrsg. von JAXenter. 2014. URL: <https://jaxenter.de/kontinuierliche-architekturvalidierung-auf-die-inneren-werte-kommt-es-an-375> (siehe S. 6, 19).
- [Mic02] Michael Feathers. *Working Effectively With Legacy Code*. 9.4.2002. URL: <http://www.netobjectives.com/system/files/WorkingEffectivelyWithLegacyCode.pdf> (siehe S. 6).
- [QA 14] QA Systems, Hrsg. *Code-Strukturen erfolgreich analysieren mit Structure101*. 24.03.2014. URL: <http://www.qa-systems.de/produkte/structure101.html> (siehe S. 19).
- [Wik16a] Wikipedia, Hrsg. *Legacy code - Wikipedia, the free encyclopedia*. 12.06.2016. URL: <https://en.wikipedia.org/w/index.php?oldid=713059641> (siehe S. 6).
- [Wik16b] Wikipedia, Hrsg. *PMD (Software)*. 12.06.2016. URL: <https://de.wikipedia.org/w/index.php?oldid=152729505> (siehe S. 19).