universitätbonn ROOTS

Ferientutorien Softwaretechnologie 2010

von Eva Stöwe und Jan Nonnen

Disclaimer

- Die Folien
  - stellen lediglich die Basis der jeweiligen Themen dar
  - erheben keinen Anspruch auf Vollständigkeit
- An mehreren Stellen wurden Sachen vereinfacht oder weggelassen.
- Wir raten dringend dazu, auch andere Quellen zur Klausurvorbereitung zu nutzen.

Klausurrelevant sind die Vorlesungsfolien.

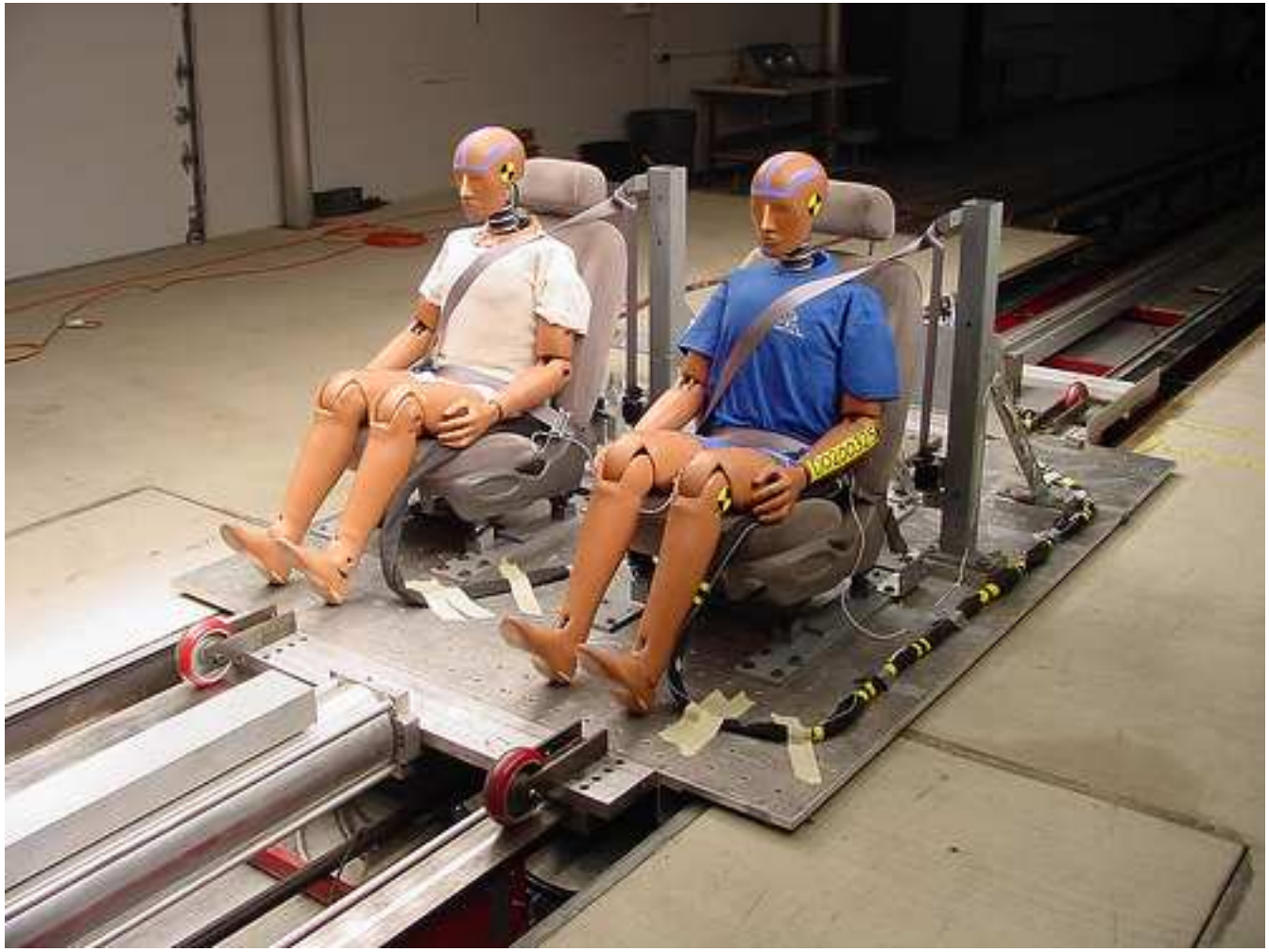
universitätbonn ROOTS

Testing
Code Smells
Refactoring

von Eva Stöwe und Jan Nonnen

16.03.2010

Testing



Fehler ▸ Systematischer Umgang

- **Vermeidung**
  - Verifikation: algorithmische Fehler vermeiden
  - Code-Design: Komplexität reduzieren
  - Code-Reviews: Zusätzliche Qualitätskontrolle
- **Erkennung**
  - Monitoring: Laufzeitdaten protokollieren
  - Debugging: Gezielte Fehlersuche
- **Toleranz**
  - Ausfallsicherheit: Wiederherstellen des letzten konsistenten Zustandes
  - Redundanz: Mehrere Systeme laufen parallel

- **Testen:** Finden von Unterschieden zwischen erwartetem und tatsächlichem Verhalten des System

Testen ▸ Abdeckungen

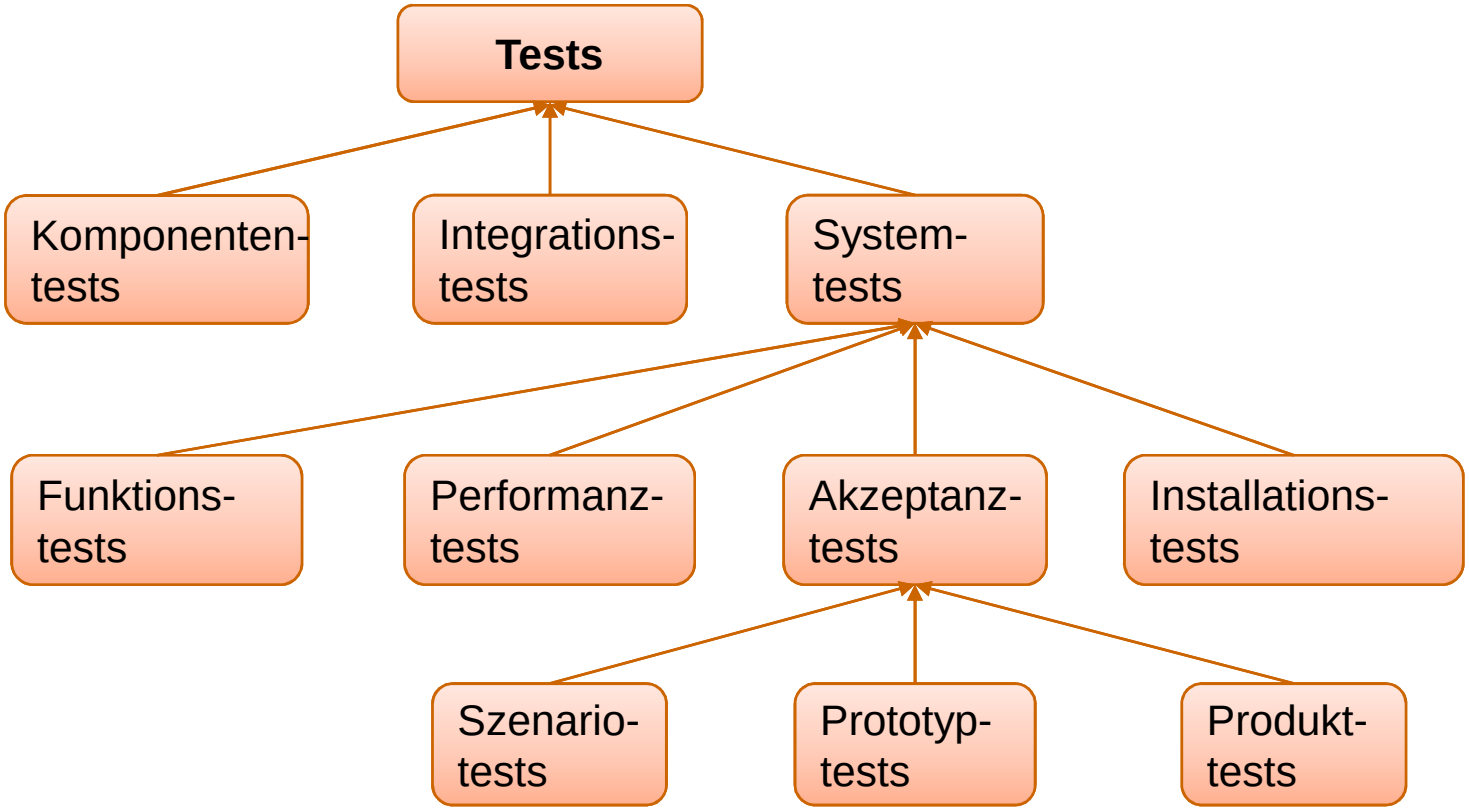
- Fokus: Gründlichkeit (hohe Abdeckung)
  - Abdeckung ist prozentuales Maß der vom Test durchlaufenen Teile der getesteten Komponente im Verhältnis zur Gesamtanzahl solcher Teile.

$$Abdeckung = \frac{Anzahl\ getesteter\ Teile}{Anzahl\ aller\ Teile}$$

- Zu testende „Teile“
  - jede Anweisung → Anweisungsabdeckung
  - jede Verzweigung → Verzweigungsabdeckung
  - jede Schleife → Schleifenabdeckung
  - jeder Pfad → Pfadabdeckung

- Ein „Teil“ gilt als getestet wenn es während des Testlaufs mindestens einmal durchlaufen wurde

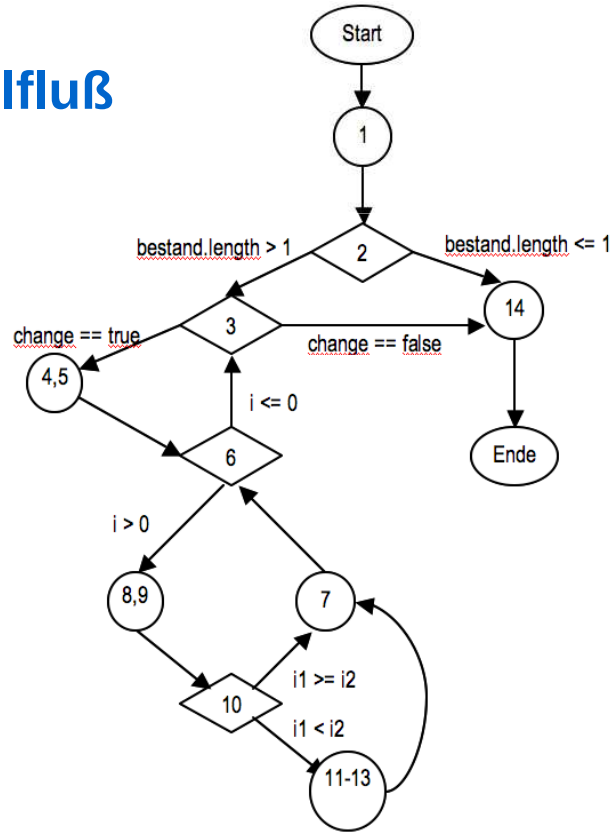
Testen ▸ Arten



Komponenten-Test ▸ Whitebox

- Wissen über Entwurf und Implementierung nutzen um
  - Anzahl von Testfällen zu begrenzen oder
  - gründliche Testabdeckung zu gewährleisten

- Stichwort: **Kontrollfluß**



Übung ▸ Kontrollfluß

```
public void methodWithForLoopAndIf()
{
   int a = 4, b = 3, c = 2; // 1-3
   for( int i = 0;          // 4
        i < c;              // 5
        i++)                // 6
   {
     b = c + a;             // 7
     a++;                   // 8
     if( b > a*c ){         // 9
         b /= c;            // 10
     }
   }
}
```

Komponenten-Test ▸ Blackbox

- Werden ohne Kenntnisse über interne Funktionsweise des Systems entwickelt

- **Ziel:**
  - Übereinstimmung eines System mit seiner Spezifikation überprüfen

- **Aber**:
  - kaum geeignet, Fehler
    - ⇨ in bestimmten Komponenten oder
    - ⇨ die fehlerauslösende Komponente
  - zu identifizieren

Testen ▸ Blackbox vs Whitebox

- Vorteile von Black-Box-Tests gegenüber White-Box-Tests:
  - bessere Überprüfung des Gesamtsystems
  - Testen von semantischen Eigenschaften bei geeigneter Spezifikation

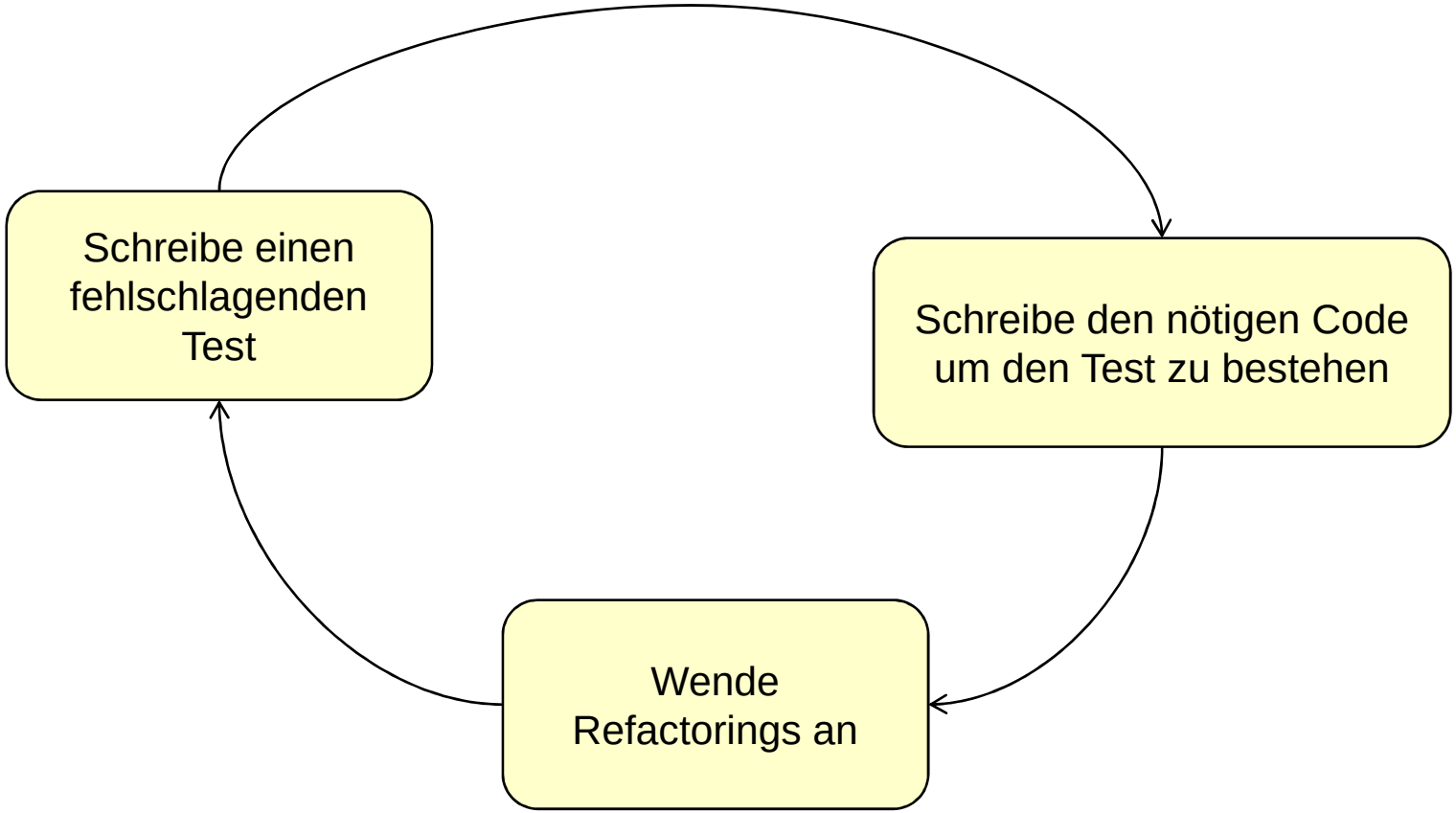
- Nachteile von Black-Box-Tests gegenüber White-Box-Tests:
  - größerer organisatorischer Aufwand
  - zusätzlich eingefügte Funktionen bei der Implementierung werden nur durch Zufall getestet
  - Tests zu einer unzureichenden Spezifikation sind unbrauchbar

Testen ▸ Integrationstests

- Finden von Fehlern in der Zusammenarbeit von Komponenten
  - Aufbauend auf Modul/Komponenten-Tests

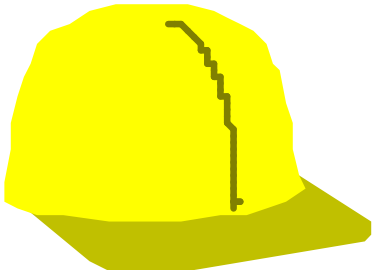
- **Teststrategie :**
  - Reihenfolge, in der Subsysteme für Tests ausgewählt werden
    - ⇨ *Big-bang-Integration*
    - ⇨ *Bottom-up-Integration*
    - ⇨ *Top-down-Integration*
    - ⇨ *Sandwich Testing*

Testen ▸ Test Driven Development



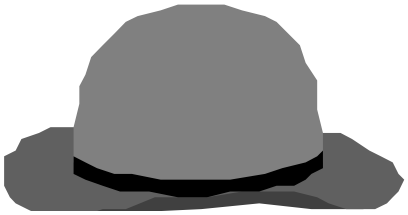
Kent Beck's "Hüte-Metapher"

- Was will ich **umstrukturieren**?
- Gibt es einen Test? → Test schreiben!
- Refactoring durchführen
- Testen → Fehler beheben!



Refactoring Hat

Entwickler



Function Adding Hat

- Was will ich **hinzufügen**?
- Test schreiben
- Funktionserweiterung durchführen
- Testen → Fehler beheben!

Testen ▸ Test Driven Development

- Testen geschieht parallel zur gesamten Entwicklung
- Code ist zu jeder Zeit vollständig mit Tests abgedeckt
- Refactorings leicht und ohne Angst ausführbar
- Kunde hat einen Überblick welche seiner Features vorhanden sind
- Persönliche Erfahrung: TDD fördert gutes Design und „clean code“

Testen ▸ Test Driven Development

1. Füge einen Test hinzu
   - Jedes Feature beginnt mit einem Test
2. Führe alle Tests aus und beobachte ob der neue Fehlschlägt
   - Überprüfung dass das Feature nicht schon abgedeckt ist
3. Füge Programm Code hinzu
   - Es soll nur genau soviel Code geschrieben werden, wie der Test zum bestehen benötigt.
     - ⇨ Es kommt nicht auf Code Eleganz an
4. Führe alle Tests aus
5. Refaktorisiere den Code
   - Nachdem der Test erfolgreich war, verbessere die Code Qualität mit Refactorings

universitätbonn ROOTS

„First make it work
Than make it right“

Refactorings

von Eva Stöwe und Jan Nonnen

Refactorings ▸ Zitat

“ By continuously improving the design of code, we make it easier and easier to work with.

This is in sharp contrast to what typically happens:
- little refactoring and
- a great deal of attention paid to expediently adding new features.

If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code. ”

Joshua Kerievsky, Refactoring to Patterns

Refactoring ▸ Definition

- **Refactoring** ist die
  - systematische Umstrukturierung der internen Struktur des Codes
  - ohne das Verhalten nach außen zu ändern

- Nutzen
  - bessere Lesbarkeit, leichteres Verständnis
  - besseres Design

➔ bessere Wartbarkeit und Wiederverwendbarkeit

Refactorings ▸ Martin Fowler

- Martin Fowler, Refactoring Home Page:
  - „Der Refactoring-Prozess besteht aus einer Serie von kleinen, verhaltenserhaltenden Transformationen.
  - Jede Transformation ändert wenig, aber eine Sequenz von Transformationen kann zu einer signifikante Restrukturierung führen.
  - Da jedes Refactoring klein ist, ist es weniger wahrscheinlich dass es Fehler produziert.
  - Das System bleibt nach jedem kleinen Refactoring lauffähig, was die Gefahr reduziert, dass es bei der Restrukturierung kaputt geht.“

Refactoring ▸ Katalog (Auszüge)

- Methoden zusammenstellen
  - Extract Method
  - Inline Method
  - Inline Temp
- Eigenschaften zwischen Objekten verschieben
  - Move Method
  - Move Field
  - Extract Class
  - Introduce Local Extension
- Daten organisieren
  - Self Encapsulate Field
  - Change Reference to Value
- Bedingte Ausdrücke vereinfachen
  - Consolidate Conditional Expression
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
- Methodenaufrufe vereinfachen
  - Rename Method
  - Add Parameter
  - Replace Exception with Test
- Umgang mit Generalisierung
  - Pull Up Method
  - Extract Superclass
  - Collapse Hierarchy
- Weiteres
  - Convert Static to Dynamic Construction
  - Extract Package
- Enterprise Java
  - Eliminate Inter-Entity Bean Communication
  - Merge Session Beans

Refactorings ▸ Beispiele aus der Vorlesung

- **Extract Method**
  - leichter verständlich
  - Teile eventuell in anderen Kontexten wieder verwendbar

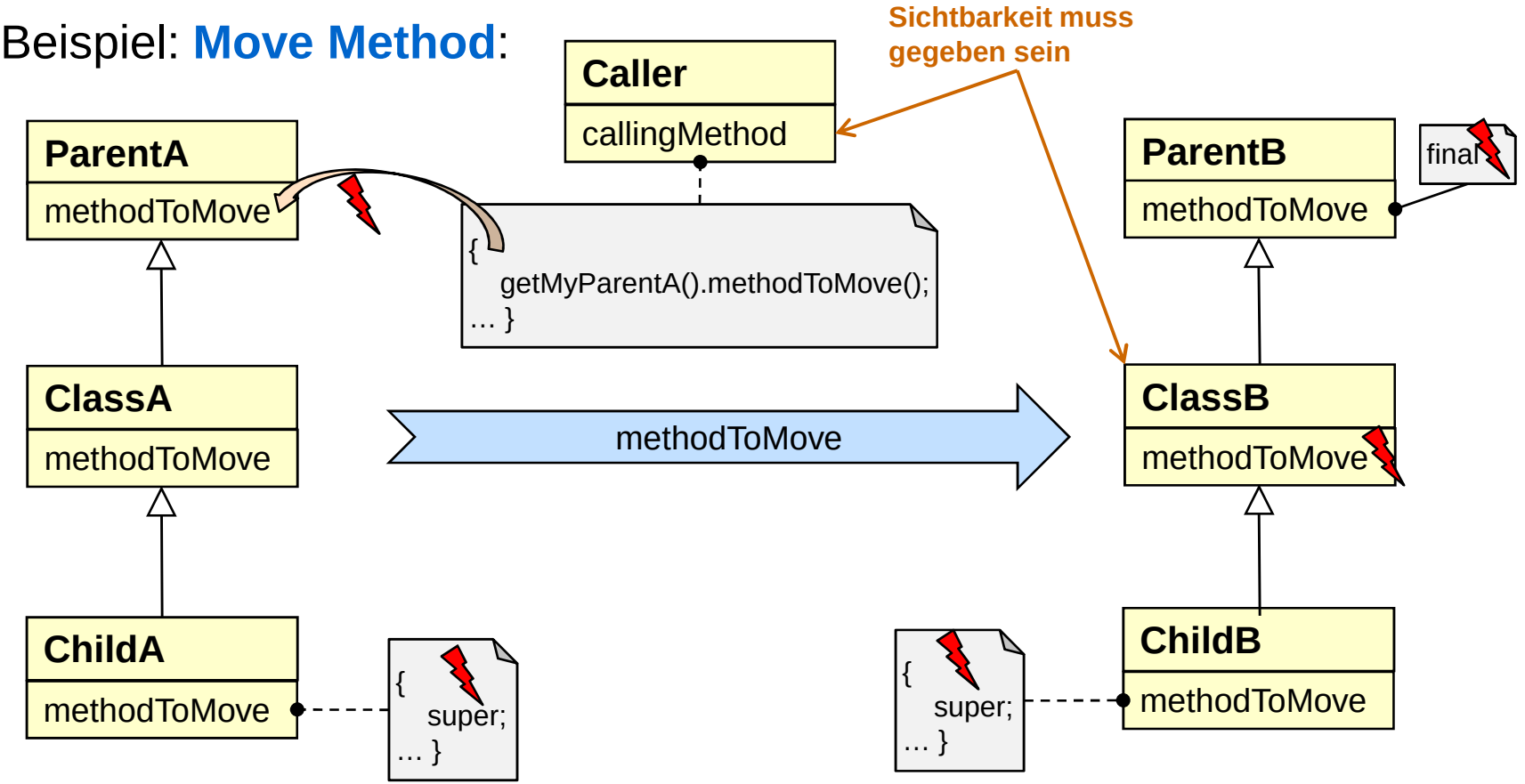
- **Move Method**
  - Methoden näher zu "ihren" Daten bringen
  - weniger Abhängigkeiten zwischen Klassen

- **Replace Temp with Query**
  - Auslagerung von Methoden erleichtern

- **Replace Conditional with Polymorphism**
  - kleinere, klarere Methoden
  - Erweiterbarkeit um zusätzliche Fälle ohne Änderung der Klienten einer Klasse

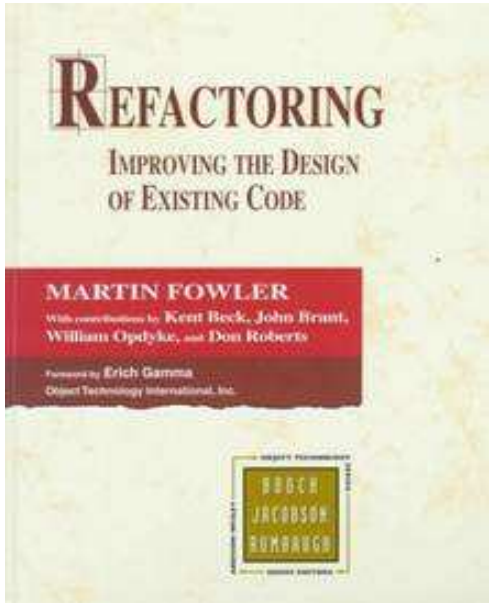
Refactoring ▸ Wozu diese Kataloge?

- Auch wenn es sich um einfach klingende Transformationen handelt, gibt es viel Fehlerpotential
- Beispiel: **Move Method**:



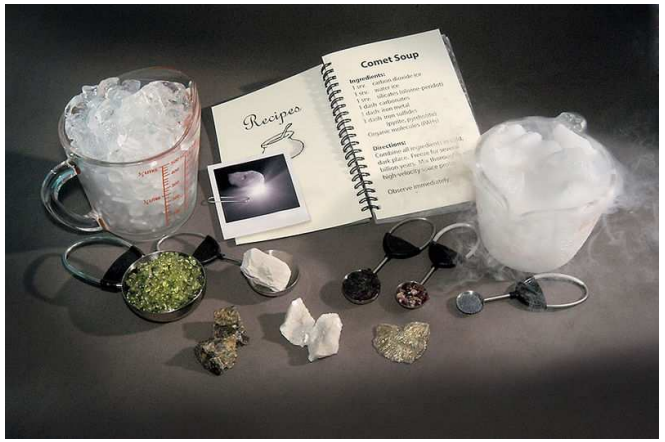
Refactoring ▸ Woraus besteht ein Rezept

- Eine Refactoring-Definition besteht aus:
  - Namen des Refactorings
  - Kurzbeschreibung
  - Allgemeine Vorher- Nachher-Beschreibung
    - ⇨ Codefragment, Klassendiagram, …
  - Motivation (Idee)
    - ⇨ In welchen Situationen soll das Refactoring helfen
    - ⇨ was tut es?
  - Mechanik
    - ⇨ Rezeptartige Auflistung, was in welcher Reihenfolge zu tun ist
  - Beispiel
  - zusätzliche Kommentare
    - ⇨ optional



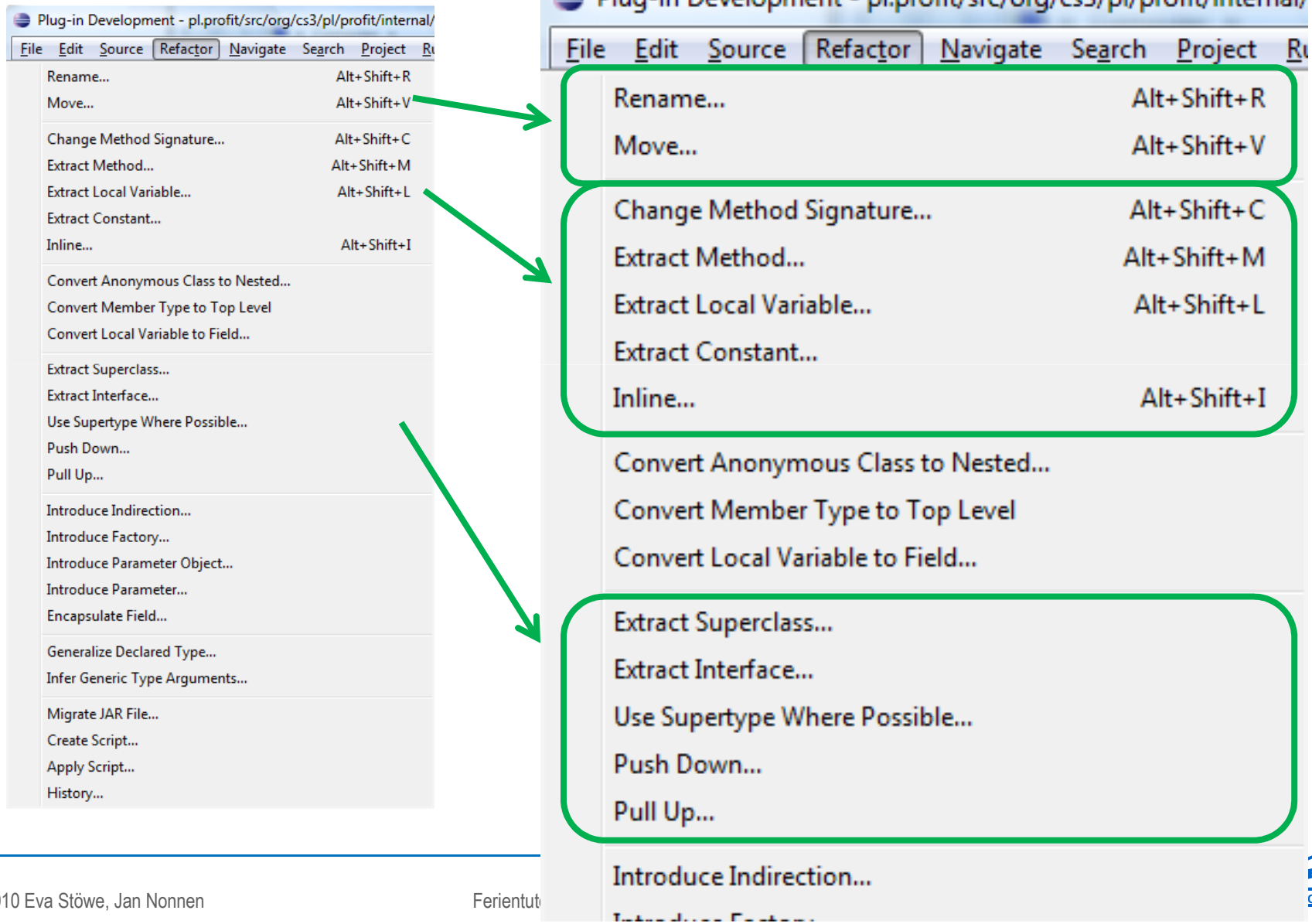
Refactoring ▸ Beispiel-Rezept
Pull Up Field

- Inspect all uses of the candidate fields to ensure they are used in the same way.
- If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
- Compile and test.
- Create a field in the superclass.
  - *If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.*
- Delete the subclass fields.
- Compile and test.



- Consider using *Self Encapsulate Field* on the new field.

Refactorings ▸ In Eclipse (Java)



Refactoring ▸ Genereller Ablauf

- Bedarf für Refactoring feststellen (Smell)

- Vollständige Testabdeckung sicherstellen
- Fehlerfreiheit garantieren
- Vorbedingung überprüfen
- Refactoring durchführen
- Überprüfen, dass Fehlerfreiheit weiterhin gewährleistet ist

- Überprüfen, ob Smell behoben ist

Refactoring ▸ Indirektion-Diskussion

- Refactoring bedeutet häufig Indirektion
  - Zerlegung großer Methoden in kleine Methoden
  - Zerlegung großer Objekte in kleine Objekte

- Vorteile von Indirektion
  - „sharing“ von Programmlogik
    - ⇨ Methoden aus Oberklassen
  - Lokalisation von Änderungen
    - ⇨ Unterklasse
  - Ersatz für Fallunterscheidungen
    - ⇨ Nachrichten

- **„Problem“**: Indirektion kann Komplexität erhöhen
  - Performance-Probleme?
  - ➔ nächste Folie

Refactoring ▸ Performance-Probleme?

Empfehlung:

- Zuerst auf gutes Design konzentrieren
  - Auch “Refactoring“ gehört dazu
- Grundregeln der Optimierung (nach Dijkstra)
  - 1. Don‘t do it.
  - 2. Don‘t do it yet.
- Nur häufig durchlaufene Programmteile („hot spots“) optimieren
  - Statistik: für ca. 80% der Laufzeit sind 10-20% des Codes verantwortlich
  - den Rest zu optimieren ist Zeitverschwendung
  - Profiling-Tools nutzen
  - Precompiler optimieren Code oft automatisch

Refactoring ▸ API-Änderungen

- Dürfen Refactorings Komponenten-Schnittstellen (APIs) ändern?
  - Wenn alle Klienten bekannt und erreichbar
    - ⇨ diese mit anpassen
- Sonst: „Reparatur“
  - Methoden: Forwarding-Methode mit alter Signatur einfügen
  - Variablen: als deprecated markieren und später entfernen
- Vorbeugung
  - „published Interface“ schlank halten
    - ⇨ Felder und Methoden so lange wie möglich „privat“ belassen
    - ⇨ Keine direkten Feldzugriffe in APIs

Refactoring ▸ Wann?

- Refactoring ist **ständiger Teil** des Entwicklungsprozesses:

- Während man neue Funktionen hinzufügt
  - Neue Struktur ist naheliegender für neue Funktionalität

- Während man einen Fehler sucht
  - Code vereinfachen um leichter Bugs zu finden

- Während man im Code „nachschlägt“
  - hilft beim Verständnis

Refactoring ▸ Wann?

Refactoring ▸ "Scouts Rule"

„Code schöner hinterlassen, als man ihn vorgefunden hat“

universitätbonn ROOTS

Code Smells

von Eva Stöwe und Jan Nonnen

Code Smells ▸ Was ist das?

- Symptom im Code das Überarbeitung nahelegt
- Keine Programmfehler sondern schlecht strukturierter Code
- Können auf tiefere Probleme im Code hinweisen, die in der schlechten Struktur verborgen liegen
- Indikatoren für Refactorings

Code Smells ▸ Einteilung

- Klassische Unterteilung
  - Innerhalb eines Typs
  - Zwischen Typen

- Alternative Unterteilung
  - The Bloaters
    - ⇨ Etwas ist so groß geworden, dass es nicht mehr handhabbar ist
  - The Object-Orientation Abusers
    - ⇨ Zustand benutzt noch nicht alle OO-Möglichkeiten
  - The Change Preventers
    - ⇨ Zustand verhindert Änderungen oder Weiterentwicklung
  - The Dispensables
    - ⇨ Etwas ist ungenutzt und sollte entfernt werden
  - The Couplers
    - ⇨ Kopplungs-bezogen

Code Smells ▸ Code Duplication & DRY

- **Don‘t Repeat Yourself (DRY) Prinzip**
  - Prinzip besagt Redundanz zu vermeiden oder zumindest zu reduzieren
  - Redundante Informationen sind schwierig zu pflegen (Konsistenz)
- Der Smell **Code Duplication** entsteht durch Kopieren vorhandenes Codes
- Code Duplication gilt als einer der häufigsten Code Smells
- Kann durch Refactoring (z.B. Extract Method Refactoring) einfach aufgelöst werden

Code Smells ▸ Methoden & SLAP

- Richtwerte für Methoden (nichteinhalten deutet auf Smells hin):
  - Sollten klein sein (Richtwert **max 15 Zeilen** in Java)
  - Sollten genau eine Aufgabe haben und gut erledigen
  - Sollten genau ein Abstraktions-Level beinhalten
    - ⇨ **Single Level of Abstraction Principle (SLAP)**
  - Sollten im „Lese-Fluss“ geschrieben sein
  - Sollten aussagekräftige passende Namen haben

„Methods should do one thing.
They should do it well.
They should do it only.“

Code Smells ▸ Kommentare

- Sind Kommentare nicht gut? Sollte man nicht alles kommentieren?
  - Öffentliche Schnittstellen sollte man mit *x*Doc **sinnvoll dokumentieren**
  - In Methoden haben sie nichts verloren und stellen einen Code Smell dar
  - Wenn sie reinen Code widerspiegeln führen sie zu redundanten Informationen
  - Kommentare veraltern und werden nicht gepflegt
  - Zuviel Kommentare führen zu **„scary noise“** und verstecken den Code

- Im Normalfall führen aussagekräftige Bezeichner zur besten Dokumentation

„Don‘t use a comment when you can use a function or a variable“

Code Smells ▸ Dead Code

- Problem: Nicht benutzter Code
- Lösung: Konsequentes löschen von ungenutztem Code
  - SCM speichert die Historie und ermöglicht die Wiederherstellung
- Andere Ausprägungen sind auskommentierte Zeilen
  - Sollten ebenso direkt gelöscht werden

Code Smells ▸ Kohäsion & Kopplung

- Kohäsion = Maß der Abhängigkeiten innerhalb der Kapselungsgrenzen
  - Hier Kapselungsgrenzen: Typen
  - Starke Kohäsion: Die Methoden in einem Typ haben ähnliche Aufgaben und sind untereinander verknüpft
- Kopplung = Maß der Abhängigkeiten zwischen den Kapselungsgrenzen
  - Hier Kapselungsgrenzen: Typen
  - Starker Kopplung: Modifikation eines Typs hat gravierende Auswirkungen auf die anderen
- Gut entwickelte Typen haben
  - **maximale Kohäsion**
  - **minimale Kopplung**

Code Smells ▸ Feature Envy(Neid)

- Eine Methode interessiert sich vorwiegend für Methoden oder Zustände einer anderen Klasse
  - ⇨ Geringe Kohäsion in der aktuellen Klasse
  - ⇨ Hohe Kopplung zu der „beneideten“ Klasse

- „Methode möchte am liebsten in der anderen Klasse sein“
  - ◆ Lösung durch Move Method Refactoring
- Falls nur Teile der Methode Neid aufweisen, sollte zuerst Method Extraction und dann Move Method ausgeführt werden
- **Aber:** Es gibt genug Patterns oder Design-Entscheidungen die bewusste den Smell erzeugen

Code Smells ▸ Refactorings (by Fowler)

Smell	Refactoring
Duplicated Code	Extract Method, Extract Class, Hide Delegate, Remove Middle Man
Comments	Extract Method, Introduce Assertion
Switch Statements	Replace Type Code with State/Strategy, Replace Type Code with Subclasses, Introduce Null Object, Replace Conditional with Polymorphism, Replace Parameter with Explicit Methods
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Primitive Obsession	Extract Class, Replace Data Value with Object, Replace Type Code with Class (Enumeration), Introduce Parameter Object, Replace Array with Object, Replace Type Code with State/Strategy, Replace Type Code with Subclasses
Long Parameter List	Introduce Parameter Object, Preserve Whole Object, Replace Parameter with Method
Long Method	Extract Subclass, Decompose Conditional, Replace Inheritance with Delegation, Replace Temp with Query
Large Class	Extract Class, Extract Subclass, Extract Interface, Duplicate Observed Data

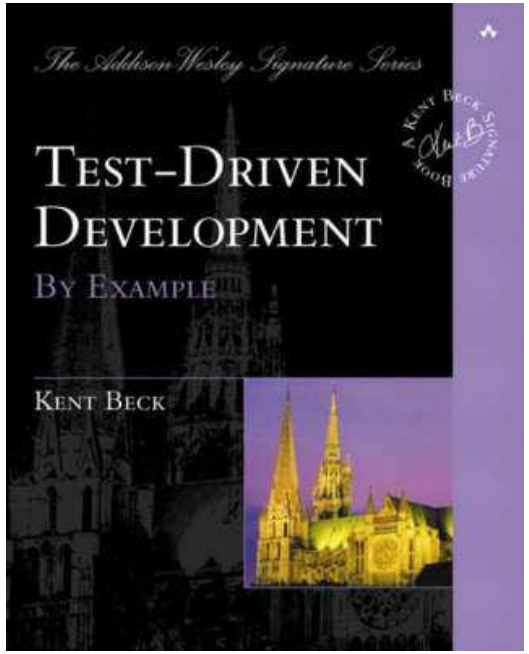
TDD ▸ Literaturempfehlungen

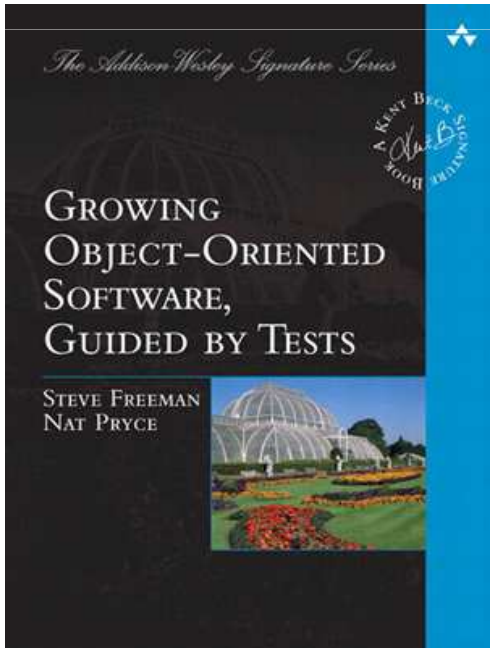
- Steve Freeman & Nat Pryce

  Growing Object-Oriented Software Guided by Tests

- Kent Beck

  Test Driven Development. By Example





Refactoring ▸ Literaturempfehlungen

- Martin Fowler: „Refactoring – Improving the Design of Existing Code“, Addison-Wesley, 1999.
  - Grundlage dieses Kapitels der Vorlesung
  - Das einführende Beispiel und alle anderen Inhalte stammen daraus.
  - Umfangreicher Katalog von Refactorings und Bad Smells

- Martin Fowler: www.refactoring.com
  - Trägt viel nützliche Informationen zum Thema zusammen, u.a.
    - ⇨ erweiterter Refactoring-Katalog
    - ⇨ Links zu verwandten Seiten

- Newsgroup
  - groups.yahoo.com/group/refactoring

Smells ▸ Literaturempfehlungen

- Robert C. Martin

  Clean Code: A Handbook of Agile Software Craftsmanship

- Michele Lanza & Radu Marinescu

  Object-Oriented Metrics in Practice