technische universität
dortmund

Software- Engineering für langlebige Systeme

Thomas Ruhroth | Dortmund
SoSe 2015

VL11

- Refactoring
- Clean Code

Refactoring

- Formal: Evolution bei der, der Evolutionsfilter die Identität ist.
- Informal (Fowler): Refactoring ist der Prozess, ein Softwaresystem so zu verändern, dass das externe Verhalten nicht geändert wird, der Code aber eine bessere interne Struktur erhält.

- Nach Fowlers Definition kann Refactoring als Gegenmaßnahme gegen die Software-Erosion gesehen werden.

Kleine und Große Refactorings

- „Kleine“ Refactorings behandeln nur die Struktur einer Methode
- „Große“ Refactorings verändern eine Klasse oder eine größere Struktur

„Kleine“ Refactorings – Ein Beispiel

Object getFirst (List l) throws EmptyExecption, NullPointerExecption, AssertionException {

```
If( l != null) {
      if (l.size() !=  0) {
          return l.get(0);
      } else {
          throw new EmptyExecption();
      }
} else {
   throw new NullPointerExecption();
}
/* This point should not be reached */
throw new AssertionException();
}
```

Exchange if and else

- Vorher

```
if (a) {
    if-block;
    } else {
    else-block;
    }
```

- Nachher:
-

```
if (!a) {
    else-block;
    } else {
    if-block;
    }
```

Intoduce Guard

- Vorher:

```
if (a) {
    if-block;
    return-or-throw;
} else {
    else-block;
}
```

- Nachher:

```
if (a) {
    if-block;
    return-or-throw;
};
else-block;
```

„Kleine“ Refactorings – Ein Beispiel

```
Object getFirst (List l) throws  EmptyExecption, NullPointerExecption,
AssertionException {

If( l != null) {
      if (l.size() !=  0) {
          return l.get(0);
      } else {
          throw new EmptyExecption();
      }
} else {
   throw new NullPointerExecption();
}
/* This point should not be reached */
throw new AssertionException();
}
```

Ein Beispiel – Refactoring 1

```
Object getFirst (List l) throws  EmptyExecption, NullPointerExecption,
AssertionException {

If( l != null) {
      throw new NullPointerExecption();
} else {
   if (l.size() ==  0) {
         throw new EmptyExecption();
      } else {
           return l.get(0);
        }
}
/* This point should not be reached */
throw new AssertionException();
}
```

Ein Beispiel – Refactoring 2

```
Object getFirst (List l) throws  EmptyExecption, NullPointerExecption,
AssertionException  {

If( l == null) {
      throw new NullPointerExecption();
}

if (l.size() ==  0) {
          throw new EmptyExecption();
      }

return l.get(0);


/* This point should not be reached */
throw new AssertionException();
}
```

Ziel – Verminderung der Softwareerosion

- Refactorings helfen die Struktur einer Software wieder herzustellen
- Code gut verständlich sein
  - siehe auch Wahrnehmungspsychologie
- Refactorings werden von Software unterstützt

Refactorings im Projekt

- Umstrukturierung am Stück
  - Projekt mit hoher Erosion
  - Weiterentwicklung wird eine Zeit stillgelegt
  - Gefahr neuer Probleme
    - Keine inkrementelle Anpassung
- Ständiges Refactoring
  - Bei jeder Änderung wird die Software verbessert
  - Nur besseren Code als vorher einchecken
  - Inkrementelle Anpassung an Bedürfnisse

Neuimplementierung

- Altes System wird gewartet, bis neues System fertig ist
- Vorteile
  - Saubere neue Struktur
  - Anwendung moderner Techniken
- Nachteile
  - Zwei Teams
    - Interner Wettbewerb, obwohl Entscheidung bereits gefallen
    - Mehr Ressourcen notwendig
  - Alle Anforderungen müssen neu entwickelt werden
    - Sind alle Anforderungen explizit bekannt?

Refactoring im Projekt

- Voraussetzung:
  - Automatisierte Tests
    - z.B. JUnit
  - Revisionsmanagmentsystem
    - Fehlerhafte Refactorings
    - Nichtbeachtung von Nebenbedingungen
    - Menschliche Fehler
- Hilfreich
  - Nutzung von Style-Guides
  - Nutzung von automatisierten Refactorings
  - Nutzung von automatisierten Metriken

Make it work, then make it write

- Refactoring verbessert:
  - Lesbarkeit
  - Übersichtlichkeit
  - Verständlichkeit
  - Erweiterbarkeit
  - Vermeidung von Redundanz
  - Kopplung und Kohäsion
  - Modularität
  - Testbarkeit
- Refactoing verbessert **nicht**:
  - Korrektheit
  - Performance
  - Security

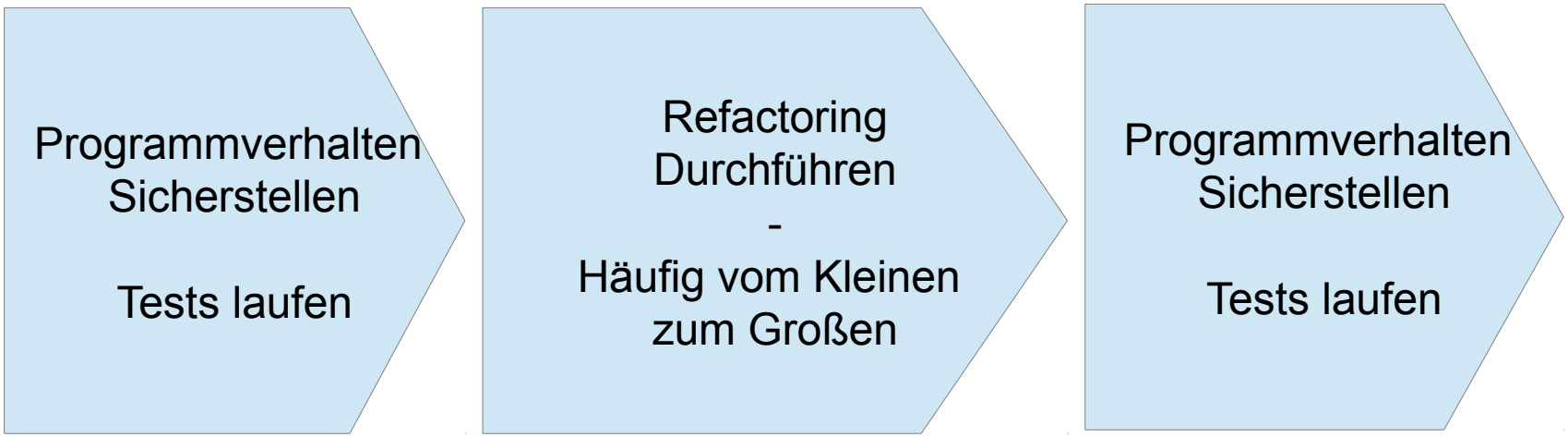
Verbesserung der Testbarkeit

- Einfacher Tests zu erstellen für
  - Methoden mit klarer Aufgabe
  - Methoden mit geringen Aufgabenumfang
- Tests besser strukturiert
  - Ausgelagerter Code braucht nur einmal getestet zu werden
  - Klassen mit wenig Abhängigkeiten
- Aufbau von Testfällen einfacher
  - Weniger große setup und tear down arbeiten

Risiken von Refactorings und Gegenmaßnahmen

- unerwünschte Änderungen und Fehler
  - Unit-Tests
  - Prinzip der kleinen Änderungen
    - Kleine Änderungen → kleine Fehler
    - Fehler schneller auffindbar
- Änderung von Schnittstellen
  - Problem, wenn Schnittstellen öffentlich benutzt
  - Änderung abhängiger Systeme notwendig

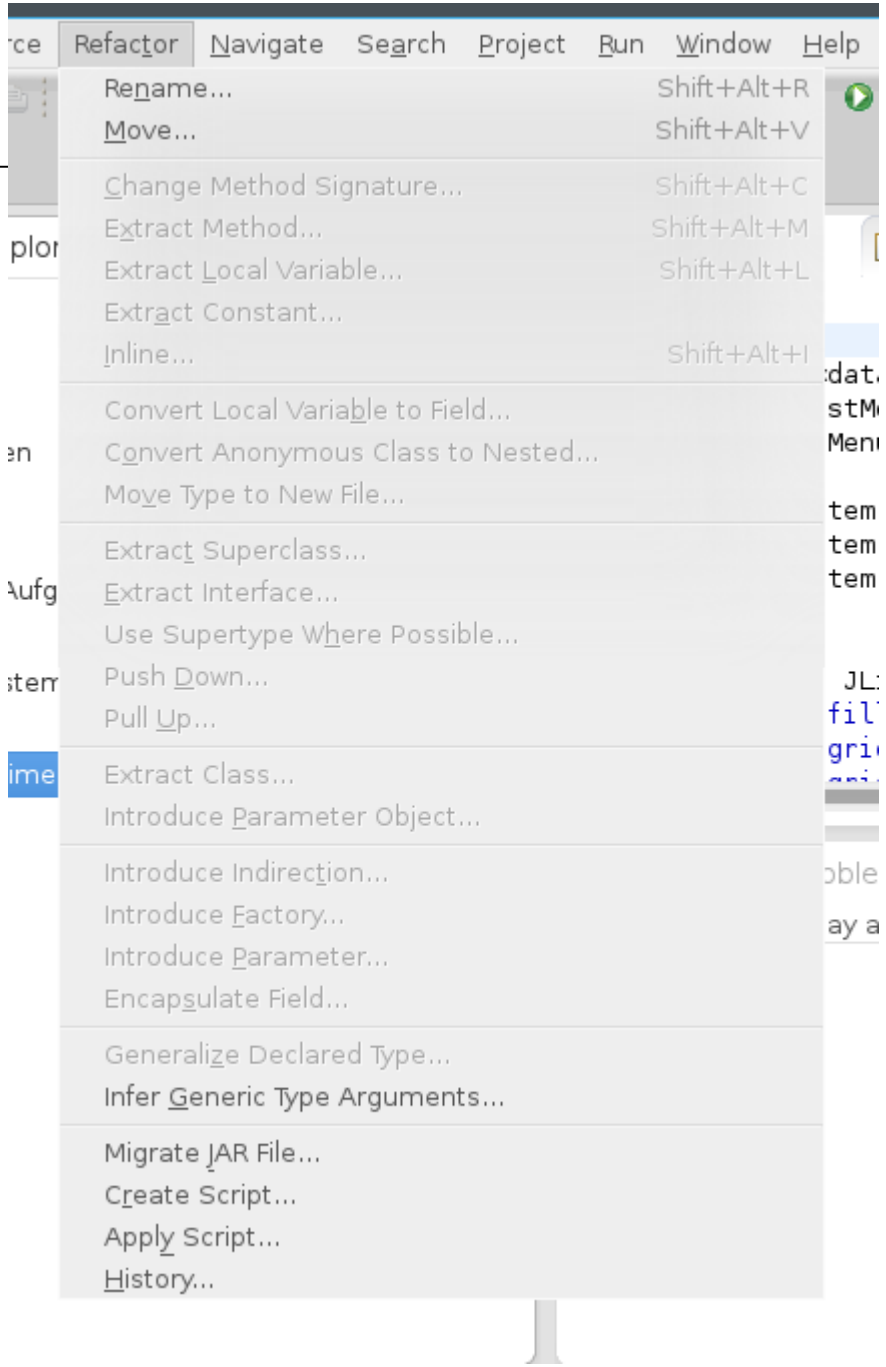
Vorgehen



- Manche Strukturrefactorings erst möglich, wenn Klassen-Strukturen sauber
  - Set von Query trennen
  - Parameterlisten
- Achtung: Make it work, then make it write

Tools

- Refactorings können von Tools automatisiert durchgeführt werden
- Weniger fehleranfällig
- Vorsicht: Nicht fehlerfrei
- Immer auf die Tests achten



Arten von Refactorings

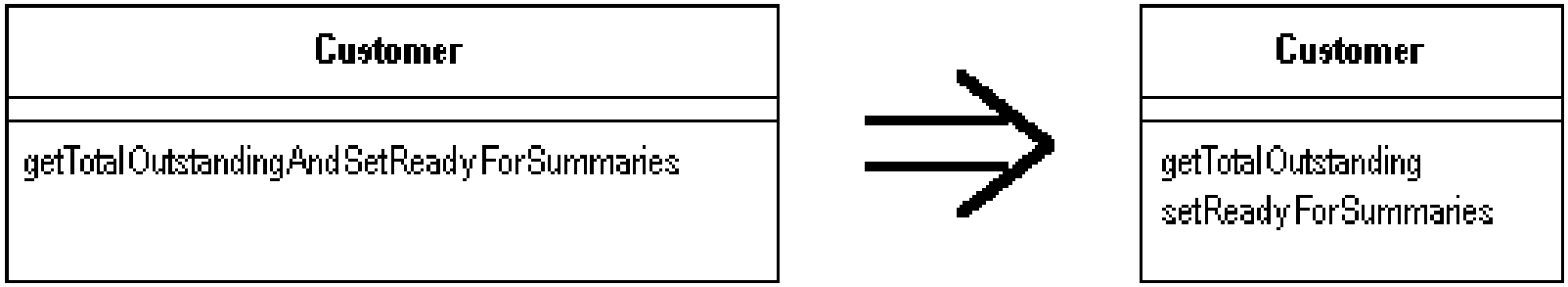
- Methoden
  - Methodenaufrufe vereinfachen
  - Bedingte Ausdrücke vereinfachen
  - Methoden zusammenstellen
- Klassensysteme
  - Eigenschaften zwischen Objekten verschieben
  - Daten organisieren
  - Umgang mit Generalisierung

Refactorings auf Methoden Ebene

- Ziel:
  - Struktur einer Methode verbessern
- Methodenaufrufe vereinfachen (Beispiele)
  - Benenne Methode um
  - Trenne Anfrage vom Verändern
  - Parametrisiere Methode
- Bedingte Ausdrücke vereinfachen (Beispiele)
  - Kontroll-Flags entfernen
  - Geschachtelte Bedingungen durch Wächter ersetzen
  - Ersetze Fallunterscheidungen durch Polymorphie
  - Null-Objekt einführen
- Methoden zusammenstellen (Beispiele)
  - Methode extrahieren
  - Methode inline setzen
  - Temporäre Variable entfernen

Beispiel: Trenne Anfrage vom Verändern

- Eine Methode liefert einen Wert zurück und ändert gleichzeitig den Zustand des Objekts.
- Schreibe zwei Methoden. Eine für die Anfrage und eine zur Modifikation.

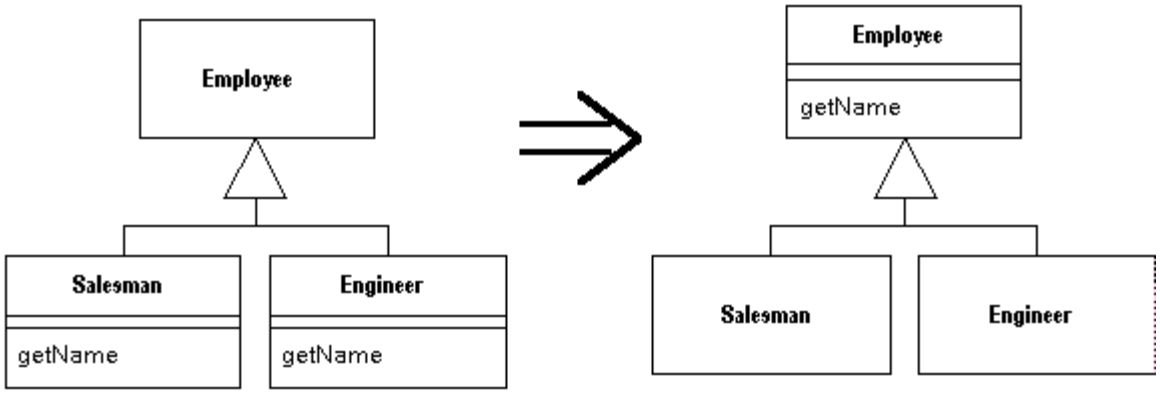


Refactorings auf Klassen-Ebene

- Methoden zusammenstellen (Beispiele)
  - Verschiebe Methode
  - Verschiebe Attribut
  - Extrahiere Klasse
  - Setze Klasse inline
- Umgang mit Generalisierung (Beispiele)
  - Attribut hochziehen
  - Methode nach unten verlegen
  - Extrahiere Unterklasse
  - Extrahiere Oberklasse
  - Hierarchien abbauen
- Daten organisieren (Beispiele)
  - Kapsele eigene Attributzugriffe
  - Ersetze Feld durch ein Objekt
  - Beobachtete Daten verdoppeln
  - Ersetze magische Zahl durch symbolische Konstante
  - Kapsele Attribut
  - Ersetze einen Datensatz durch eine Datenklasse
  - Ersetzte Typ-spezifischen Code durch Unterklassen

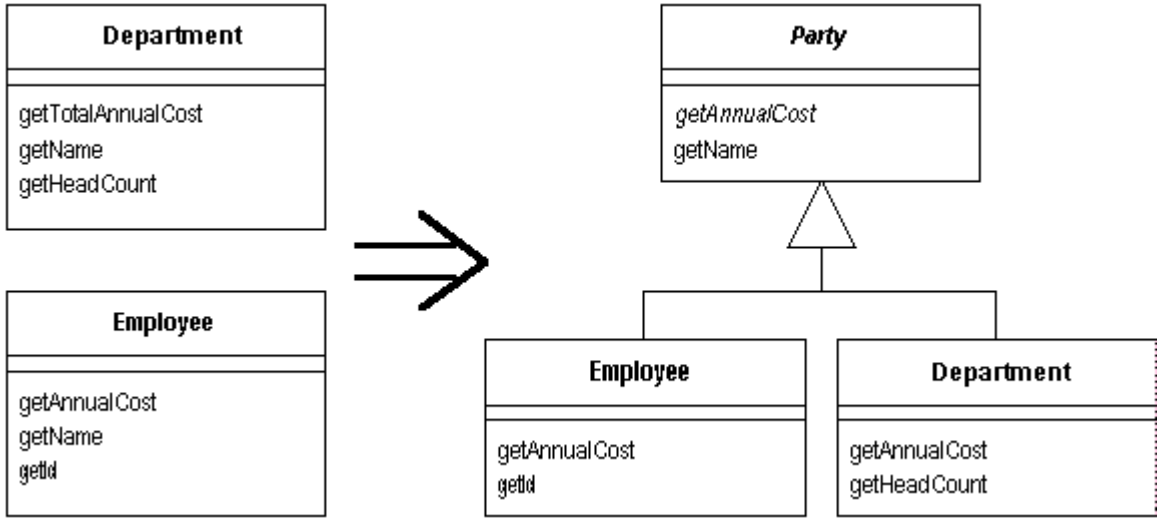
Methode Hochziehen

- Es gibt Methoden mit identischen Ergebnissen in den Unterklassen.
- Verschiebe sie in die Oberklassen



Extract Superclass

- Es gibt zwei Klassen mit gemeinsamen Eigenschaften.
- Erzeuge eine neue Oberklasse und lege die gemeinsamen Eigenschaften in diese neue Klasse.



Verbessern von unten:
Der Weg des Clean Code

Verbesserung von unten

- Problem:
  - Auftraggeber/Chefs/… geben ungern Ressourcen für eine Umstrukturierung frei
  - Erosion verlangsamt Entwicklung
- Idee:
  - Zeit die durch die Erosion verschenkt wird, für Clean Code aufwenden
  - Je besser die Struktur (=weniger Erosion) je mehr Zeit wird frei
  - Zeit die für die Diskussion über Probleme genutzt würde für den Start verwenden
- Aktion:
  - Jeder Programmierer sorgt ohne Auftrag von „oben“ für besseren Code
  - Werben untereinander

Clean Code Initiative oder
Der Weg zum guten Programmierer

- Idee:
  - Verbessre dich Schritt für Schritt
  - Nehme dir ein Thema nach dem anderen vor
  - Verbessere dich und behalte gute Eigenschaften
- Code Qualität in den Alltag einbauen
- Leben von „gutem Code“ überzeugt mehr, als jede Diskussion
- Sofort Gut ist besser als reparieren
- Bekämpfen „Mangel an Disziplin beim Programmieren“
- Verbesserung von Unten
- Stufen - Katas

Die Tugenden im Clean Code - Prinzipielles

- Schätze Variation (Value Variation (VV))
  - Werte: Evolvierbarkeit, Kontinuierliche Verbesserung
- Tue nur das Nötigste (Do Only What´s Neccessary (DOWN))
  - Werte: Produktionseffizienz, Evolvierbarkeit
  - Vorsicht vor Optimierungen! (Prinzip des roten Grads)
  - You Ain´t Gonna Need It (YAGNI) (Prinzip des blauen Grads)
  - Keep it simple, stupid (KISS) (Prinzip des roten Grads)
- Isoliere Aspekte (Isolate Aspects (IA))
  - Werte: Evolvierbarkeit
  - Don´t Repeat Yourself (DRY) (Prinzip des roten Grads)
  - Separation of Concerns (SoC) (Prinzip des orangenen Grads)
  - Single Level of Abstraction (SLA) (Prinzip des orangenen Grads)
  - Single Responsibility Principle (SRP) (Prinzip des orangenen Grads)
  - Interface Segregation Principle (ISP) (Prinzip des gelben Grads)
  - Entwurf und Implementation überlappen nicht (Prinzip des blauen Grads)

Die Tugenden im Clean Code - Prinzipielles

- Minimiere Abhängigkeiten (Minimize Dependencies (MD))
  - Werte: Evolvierbarkeit
  - Dependency Inversion Principle
  - Information Hiding Principle
  - Law of Demeter
  - Open Closed Principle
  - Tell, don´t ask
  - Interface Segregation Principle (ISP)
- Halte Versprechen ein (Honor Pledges (HP)
  - Werte: Evolvierbarkeit oder auch: Minimize Surprises
  - Liskov Substitution Principle
  - Principle of Least Astonishment
  - Implementation spiegelt Entwurf
  - Favour Composition over Inheritance (FCoI)

Die Tugenden im Clean Code - Praktisches

- Umarme Unsicherheit (Embrace Uncertainty (EU))
  - Werte: Evolvierbarkeit, Kontinuierliche Verbesserung
  - Ein Versionskontrollsystem einsetzen
  - Automatisierte Integrationstests
  - Automatisierte Unit Tests
  - Mockups (Testattrappen)
  - Continuous Integration
  - Inversion of Control Container
- Fokussiere (Focus (F))
  - Werte: Produktionseffizienz
  - Komponentenorientierung
  - Test first
  - Limit WIP
- Wertschätze Qualität (Value Quality (VQ))
  - Werte: Produktionseffizienz
  - Akzeptiere nur hohe Qualität
  - Automatisierte Unit Tests
  - Reviews

Die Tugenden im Clean Code - Praktisches

- Mach fertig (Get Things Done (GTD))
  - Werte: Produktionseffizienz
  - Iterative Entwicklung
  - Continuous Delivery
  - Limit WIP
- Halte Ordnung (Stay Clean (SC))
  - Werte: Evolvierbarkeit, Korrektheit, Produktionseffizienz
  - Die Pfadfinderregel beachten (Hinterlasse einen Ort immer in einem besseren Zustand als du ihn vorgefunden hast.)
  - Komplexe Refaktorisierungen
  - Einfache Refaktorisierungsmuster anwenden
  - Statische Codeanalyse (Metriken)
  - Code Coverage Analyse
  - Source Code Konventionen
- Bleib am Ball (Keep Moving (KM))
  - Werte: Kontinuierliche Verbesserung
  - Lesen, Lesen, Lesen
  - Teilnahme an Fachveranstaltungen
  - Erfahrung weitergeben
  - Täglich reflektieren
  - Root Cause Analysis
  - Messen von Fehlern
  - Issue Tracking
  - Regelmäßige Retrospektiven

Die Grade

- Schwarzer 0. Grad
  - Der Suchende – arbeitet noch nicht nach Clean Code, lernt aber aktiv
- Roter 1. Grad
  - Tägliche Übungspraxis
  - Wenige Tugenden werden angewandt
  - Grundlage für den Prozess der Kontinuierlichen Verbesserung
- Oranger 2. Grad
- Gelber 3. Grad
- Grüner 4. Grad
- Blauer 5. Grad
- Weißer 6. Grad
  - Anwendung aller Tugenden zu jeder Zeit

Roter Grad

- Prinzipie
  - Don´t Repeat Yourself (DRY)
  - Keep it simple, stupid (KISS)
  - Vorsicht vor Optimierungen!
  - Favour Composition over Inheritance (FCoI)
- Praktiken
  - Die Pfadfinderregel beachten
  - Root Cause Analysis
  - Ein Versionskontrollsystem einsetzen
  - Einfache Refaktorisierungsmuster anwenden
  - Täglich reflektieren

Ergänzende Prinzipien

Code aufräumen vor und nach Erweiterung

- Vor jeder Erweiterung
- Nach jeder Erweiterung
- Arbeiten in Clean Code ist einfacher
- Schlechter Code ist schlecht wartbar

- Im Handwerk ist aufräumen normal!

Pair Programming

- Zwei Programmierer arbeiten zusammen
  - Am gleichen Rechner
  - Am gleichen Code
- Rollenteilung
  - Person 1
    - Programmiert
  - Person 2
    - Kontrolliert den Code während er geschrieben wird
    - Spricht Probleme an
    - Denkt über die Lösung nach
  - Beide
    - Diskutieren Möglichkeiten

Pair Programming

- Vorteile
  - Weniger Fehler (ca. 15-20% weniger)
  - Kleinere Programme (ca. 20% kleiner)
  - Besserer Code
  - Wissensvermittlung
- Nachteile
  - Kosten
  - Teamfindung

Morgen in der Übung!
Bitte Rechner mitbringen
Java Entwicklungsumgebung

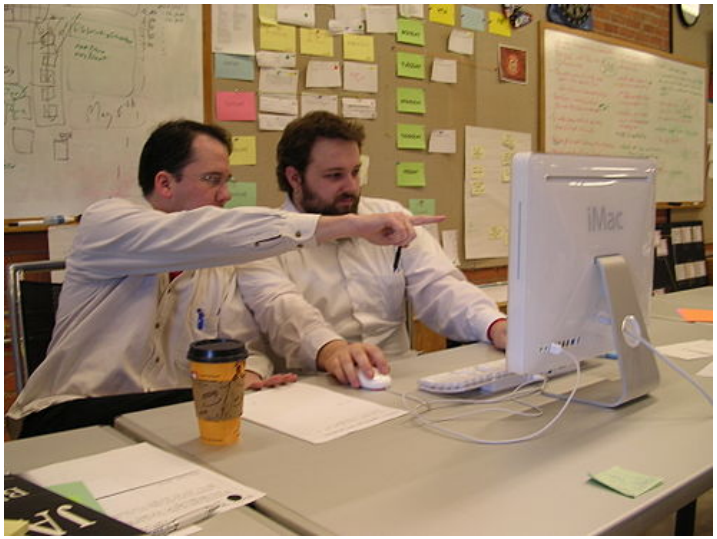


Bild: Wikipedia: Lisamarie Babik - Ted & Ian Uploaded by Edward

Pair Programming beim „Verbesserung von unten“

- Vollzeit nicht möglich
- Schon zwei Stunden die Woche bring einen weiter
  - Wissensvermittlung
  - Kennenlernen der eigenen Schwächen
  - Stärkung der Kommunikation
- Besonders gut bei schwierigen/kniffeligen Herausforderungen
  - Kann dann als „Kollegenhilfe“ deklariert werden

Nächste Woche:
WarpUp