Johannes Link

jl@johanneslink.net

Bevor ich es vergesse...

- Dank an Malte Finsterwalder
- Dank an Jens Coldewey (coldewey.com)
- Ich suche einen studentischen Mitarbeiter

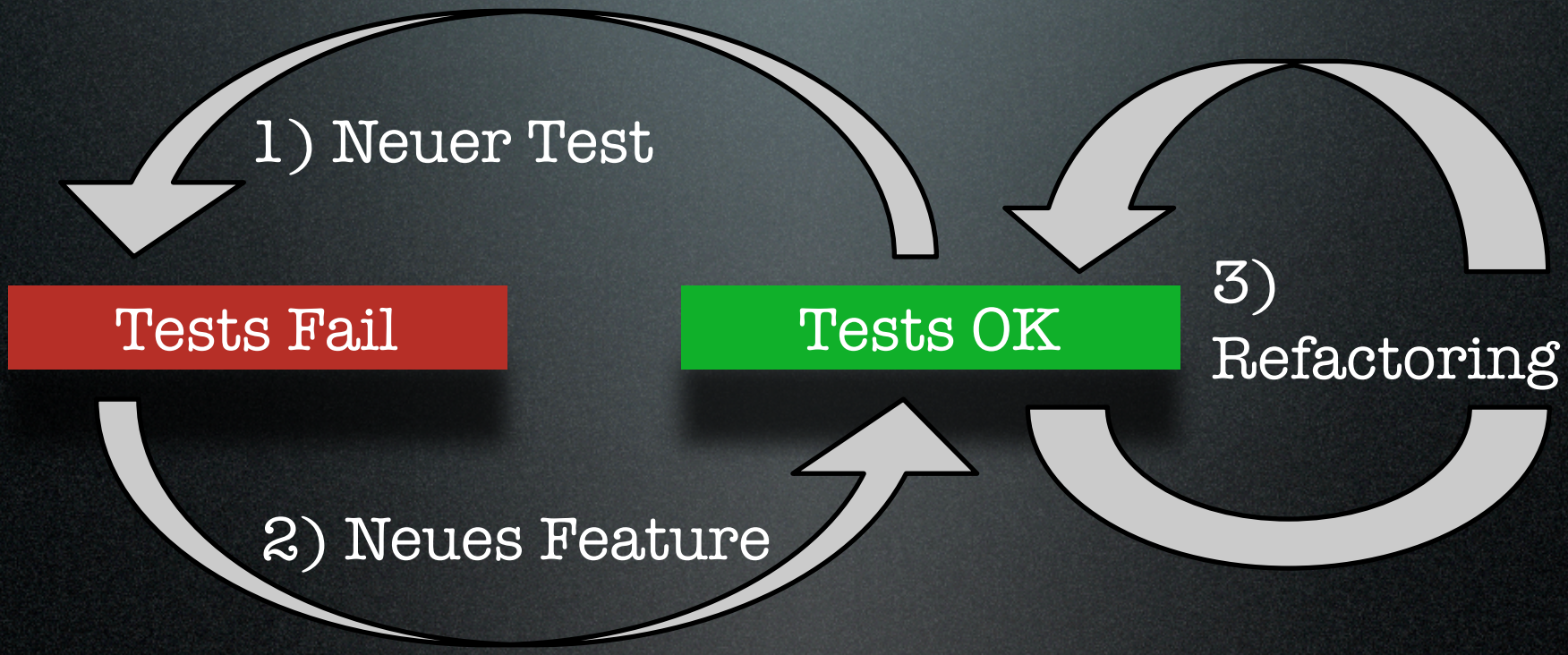
Refactoring

anhand von Beispielen

Evolutionäres Design

- **Geplantes Design** versucht, heutige und mögliche zukünftige Anforderungen auf einen Schlag abzudecken.
- Viele Annahmen treten jedoch nicht ein, so dass der Code unnötig komplex wird.
- Unvorhergesehene Anforderungen kommen trotzdem
- Mit **evolutionärem Design** können wir die heutigen Anforderungen in kleinen Schritten erfüllen.
- Wir vermeiden den vorausplanenden Blick in die unsichere Zukunft.
- **Refactoring** hält das Design für zukünftige Anforderungen offen.

Test / Code / Refactor



Refactoring?

„Eine Änderung an der internen Struktur eines Programms, um es leichter verständlich und besser modifizierbar zu machen, ohne dabei sein beobachtbares Verhalten zu ändern.“

[Fowler 99]

„If you want to refactor,
the essential precondition
is having solid tests.“

Martin Fowler

Umformung mathematischer Formeln:

$$(a + b)^2 =$$

$$\Downarrow$$

$$(a + b)(a + b) =$$
$$a^2 + ab + ba + b^2 =$$
$$a^2 + ab + ab + b^2 =$$

$$\Downarrow$$

$$a^2 + 2ab + b^2$$

Umformung von Code:

```
void foo() {…}
…
x = foo();
```

⇩

```
void guterName() {…}
…
x = guterName();
```

☞ Mathematische Umformungen: Die Formel verändern, und den Wert der Formel unverändert lassen

☞ Refaktorisieren: Den Code ändern, ohne sein Verhalten (Semantik) zu ändern

Größere Refaktorisierungen kann man mit mathematischen Beweisen vergleichen

- Vom Start zum Ziel braucht man viele kleine (oft triviale) Zwischenschritte
- Jeder einzelne Schritt erhält die Semantik
- Zwischenschritte sind oft komplexer als der Start
- Es gibt Strategien für mittelgroße Schritte
- Gelegentlich läuft man in Sackgassen
- Gelegentlich macht man Fehler

☞ **Unterschied:** Beim Refaktorisieren kann man mit Unit-Tests zu jedem Zeitpunkt feststellen, ob man Fehler gemacht hat

Refactoring erhält strukturelle Qualität

- Wir refaktorisieren,
  - um das Design zu verbessern
  - um das Programm leichter verständlich zu machen
  - um zukünftige Änderungen am Code zu erleichtern
  - um der Entropie entgegen zu wirken

In einem abgeschlossenen System nimmt die Unordnung nicht ab.

2. Hauptsatz der Thermodynamik

Das gilt auch für Software...

Der Komplexitätstod eines Softwaresystems

Ohne Gegenmaßnahmen nimmt die Komplexität eines Systems immer mehr zu...

...und senkt die Produktivität...

...bis die Wartung irgendwann unwirtschaftlich wird

Refaktorisieren nimmt Komplexität aus dem bestehenden System

Refactoring-Ziel: Das einfachste Design

Design ist einfach, wenn der Code ...

- ... alle seine Tests erfüllt.
- ... jede Intention der Programmierer ausdrückt.
- ... keine duplizierte Logik enthält.
- ... möglichst wenig Klassen und Methoden umfasst.

Reihenfolge entscheidend!

Refactoring - Vorgehen

- Refactoring findet ständig statt.
- Klein(st)e Schritte
- Entweder Refactoring oder neue Funktionalität
- Unit Tests sind das Fangnetz des Refactoring.
- Oft ist auch ein Refactoring der Tests notwendig.

Warum ständiges Refactoring?

- „Nicht optimales“ Design zu dulden, ist wie Schulden machen
- Große Refactorings bremsen das ganze Team
- Refactoring-Phasen lassen sich nur schwer verkaufen

➡ Refactoring gehört zum professionellen Entwickeln und sollte nicht zur „Verhandlungsmasse“ gehören

Übelriechender Code

Code Smells:
Indizien für (eventuell) notwendiges Refactoring

- duplizierte Logik
- lange Funktionen
- Kommentare
- switch-Ketten, verschachtelte if-then-else
- Code, der seine Intention nicht ausdrückt
- Neid auf die Features einer anderen Klasse
- Datenklassen ohne wirkliches Verhalten
- „Eigentlich…“
- …

Refactorings

- Rename
- Extract Method / Variable
- Inline Method / Variable
- Encapsulate Field
- Change Method Signature
- Move Method
- Introduce Null Object
- Replace Conditional with Polymorphism
- Replace Inheritance with Delegation
- ...

➡ http://refactoring.com/catalog/index.html

Was tut dieser Code?

```
public class A {
   private int b = 0;
   public int gb() {
      return b;
   }
   public void d(int a) {
      c(a);
      b += a;
   }
   private void c(int a) {
      if (a <= 0.0) {
         throw new IAE();
      }
   }
}
```

Namen sind mehr als Schall und Rauch: Das Rename-Refactoring

```
public class Konto {
    private int saldo = 0;

    public int getSaldo() {
        return saldo;
    }

    public void zahleEin(int betrag) {
        ueberpruefeDassBetragNichtNegativ(betrag);
        saldo += betrag;
    }

    private void ueberpruefeDassBetragNichtNegativ(int amount) {
        if (betrag <= 0.0) {
            throw new IllegalArgumentException();
        }
    }
}
```

„Der Thesaurus ist mein wichtigstes Programmier-Werkzeug“

Ward Cunningham

Suchen und Ersetzen genügen nicht!

```
public class Foo {
  int foo;
  int foo(int b) {
    int foo = b * this.foo;
    return foo;
  }
}

public class FooUser {
  Foo foo;

  void foo(int c) {
    foo.foo(c);
  }

}
```

- Sichtbarkeit und Verschattung müssen berücksichtigt werden
- Nötig: Auswertung interner Informationen des Compilers

☞ Brauchbare Refactoring-Browser sind eng mit dem Compiler verbunden

Gute Namen machen viele Kommentare überflüssig

Vorher:

```
/**
 * Ueberweise betrag in Euro
 * auf anderes konto
 */
public void ueberweise(double betrag,
  Konto konto) {
      konto.saldo += betrag;
      this.saldo -= betrag;
}
```

Nachher:

```
public void ueberweiseAuf(double betragInEuro,
  Konto zielKonto) {
   zielKonto.saldo += betragInEuro;
   this.saldo -= betragInEuro;
}
```

- Kommentare sieht man nur in der Deklaration, gute Namen überall
- Um gute Namen zu finden braucht man oft mehrere Versuche
- „Kann ich aus diesem Kommentar einen Namen machen?“

Wann ist Umbenennen sinnvoll?

- Um die Verständlichkeit des Codes zu erhöhen
- Wenn sich das eigene Verständnis des Codes verbessert hat
- Wenn sich die Zuständigkeiten der Klasse verändert haben (z.B. Generalisierung)
- Um Redundanz sichtbar zu machen
- Um APIs zu vereinheitlichen
- Um Verschattungen zu eliminieren

Divide and Conquer: Extract Method

Java – Uebung 1 – Loesung/src/tictactoe/loesung

Konto.java | TicTacToe.java

```
            EMPTY_FIELD };

    private char lastPlayed = EMPTY_FIELD;

    public String printBoard() {
        StringBuffer buffer = new StringBuffe
        for (int fieldIndex = 1; fieldIndex <
            buffer.append(getFieldChar(fieldI
            if (fieldIndex % EDGE_SIZE == 0)
                buffer.append(LINE_DELIMITER)
            } else {
                buffer.append(' ');
            }
        }
        return buffer.toString();
    }

    private char getFieldChar(int fieldIndex)
        return board[fieldIndex - 1];
    }
```

Writable | Smar...sert | Build

Extract Method

Method name:

Access modifier: public protected default private

Parameters:

Type	Name
StringBuffer	buffer
int	fieldIndex

Edit... Up Down

Declare thrown runtime exceptions

Generate method comment

Replace all occurrences of statements with method

Method signature preview:

```
private void extracted(StringBuffer buffer,int fieldIndex)
```

Preview > Cancel OK

Parameter und Rückgabetypen werden automatisch bestimmt

```
private void paintCard(Graphics g) {
  Image image = null;
  if (card.getType().equals("Problem")) {
    image = explanations.getGameUI().problem;
  } else if (card.getType().equals("Solution")) {
    image = explanations.getGameUI().solution;
  } else if (card.getType().equals("Value")) {
    image = explanations.getGameUI().value;
  }
  g.drawImage(image, 0, 0, explanations.getGameUI());
  if (shouldHighlight())
    paintCardHighlight(g);
  paintCardText(g);
}
```

```
private void paintCard(Graphics g) {
  paintCardImage(g);
  if (shouldHighlight())
    paintCardHighlight(g);
  paintCardText(g);
}
```

Auch Ausdrücke können extrahiert werden

```
if (card.getType().equals("Problem")) {
```

Extract Method

```
private String problem() {
  return "Problem";
}
...
if (card.getType().equals(problem())) {
```

Extract Local Variable

```
String problem = "Problem";
if (card.getType().equals(problem)) {
```

Extract Constant

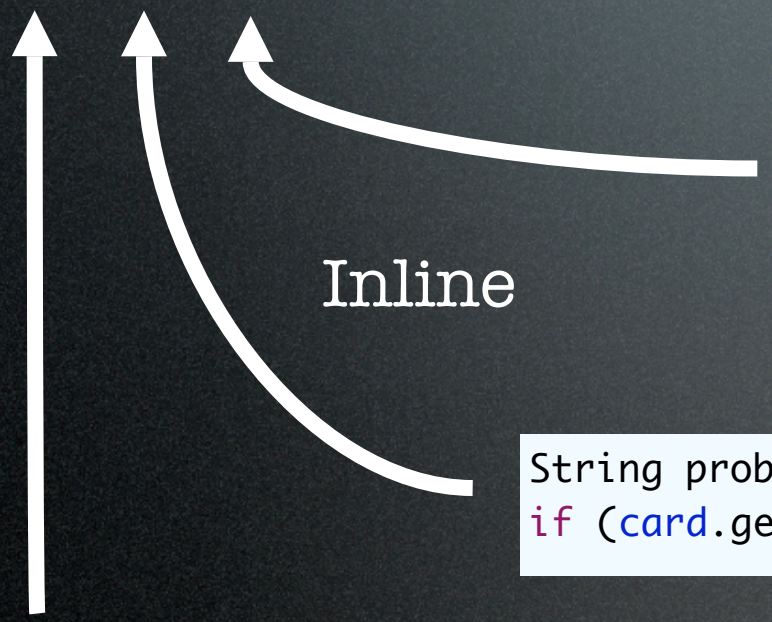
```
private static final String PROBLEM = "Problem";
...
if (card.getType().equals(problem())) {
```

Wann ist es sinnvoll, zu extrahieren?

- Um die Lesbarkeit zu verbessern und Kommentare loszuwerden
- Um alle Aufrufe auf der gleichen Detailstufe zu haben
- Um redundante Abschnitte zusammen zu fassen
- Zur Vorbereitung weiterer Umstellungen:
  - Um Codestücke zu extrahieren, die alle an ein (anderes) Objekt gehen
  - Um Teile an der API bereit zu stellen
  - Um abweichendes Verhalten in Subklassen überschreiben zu können

Die Umkehroperation zu Extract ist Inline

```
if (card.getType().equals("Problem")) {
```




```
private String problem() {
  return "Problem";
}
...
if (card.getType().equals(problem())) {
```

```
String problem = "Problem";
if (card.getType().equals(problem)) {
```

```
private static final String PROBLEM = "Problem";
...
if (card.getType().equals(problem())) {
```

Wann ist Inline sinnvoll?

- Um überflüssige Variablen zu eliminieren
- Um reine „Durchreich-Methoden“ zu eliminieren
- Um alle Methoden auf der gleichen Detailebene zu haben
- Wenn viele Aufrufer gleichzeitig umgestellt werden müssen
- Für API-Änderungen:
  - deprecated Aufrufe eliminieren
  - Änderungen der Signatur automatisieren

Feature Envy: Ich bin nicht zuständig

- Wenn alle (oder die meisten) Aufrufe innerhalb einer Methode an ein anderes Objekt gehen, dann sollte die Methode auch dort angesiedelt sein:

```
private Storage storage;
protected void appendWithStorageCheck(ElementType element) {
  if (storage.noMoreSpaceLeft()) {
    if (storage.compactingMakesSense())
      storage.compactStorageArray();
    else
      storage.extendStorageArray();
  }
  storage.append(element);
}
```

- Diese Methode gehört in die Klasse ***Storage***

Zuständigekeiten verschieben: Move Method

```
private Storage storage;
protected void appendWithStorageCheck(ElementType element) {
  if (storage.noMoreSpaceLeft()) {
    ...
  }
  storage.append(element);
}
```

```
protected void appendWithStorageCheck(ElementType element) {
  storage.appendWithCheck(element);
}

class Storage...
 public void appendWithCheck(ElementType element) {
   if (noMoreSpaceLeft()) {
     ...
   }
   append((ElementType) element);
 }
```

Wann kann man Move einsetzen?

- Um Zuständigkeiten an einer Stelle zu sammeln („Eine Klasse, ein Konzept“)
- Um Klassen aufzuspalten
- Um „überfettete“ Klassen aufzuräumen
- Teilschritt zahlreicher komplexer Refactorings, z.B.:
  - Replace Inheritance with Delegation
  - Replace Conditional with Polymorphism

Replace Conditional with Polymorphism

```
public class Konto...
  private boolean istBetragGedeckt(double betrag) {
    switch (kontotyp) {
    case GIRO:
      return betrag <= saldo + dispo;
    case SPAR:
      return betrag <= saldo;
    }
    throw new RuntimeException("Unbekannter Kontotyp");
  }
```

```
public abstract class Konto...
  abstract protected boolean istBetragGedeckt(double betrag);
```

```
public class Sparkonto extends Konto...
   protected boolean istBetragGedeckt(double betrag) {
       return betrag <= saldo;
   }
```

```
public class Girokonto extends Konto...
   protected boolean istBetragGedeckt(double betrag) {
       return betrag <= saldo + dispo;
   }
```

Tipps zum Refaktorisieren

- „Aufräumen“ geht ad hoc, strukturelle Änderungen sollte man diskutieren
- Zwischenprodukte über (Verstoß gegen) Namenskonvention oder deprecated kennzeichnen
- Nicht alles auf einmal machen: Brauchbare Zwischenstände synchronisieren und einchecken
- Mut zum Rückschritt: Manchmal verrennt man sich
- Ehrgeiz: Das Design sollte nur aktuellen Anforderungen widerspiegeln, *weder* die Historie *noch* (Annahmen über) die Zukunft

Technische Grenzen des automatischen Refactorings

- Es kann nur Code geändert werden, der auch geladen ist
- Nur wo der Compiler Abhängigkeiten erkennt, kann er konsistent ändern: Vorsicht mit Reflection!
- Es gibt noch kaum sprachübergreifenden Ansätze, z.B. Java/SQL, Java/JavaScript, Java/XML
- Viele nennenswerte Refactorings sind nicht automatisierbar, weil sie Wissen über die Absicht des Codes verlangen

Organisatorische Grenzen des Refactorings

- Besitzdenken über Code verhindert oft notwendige Umbauten
- Verteilte Entwicklung verhindert notwendige Abstimmungen
- Refactoring braucht manchmal Mut – der muss auch in die Kultur passen
- Wenn „Produktivität“ zu feingranular gemessen wird, ist keine Zeit mehr für „unproduktives“ Refaktorisieren – und die Produktivität sinkt
- „Wenn Sie es gleich richtig gemacht hätten, müssten Sie jetzt keine Zeit für Umbauten verschwenden!“
- Conway‘s Law: „Architecture follows organization“

Große Umbauten erfordern Planung

- Je größer ein Umbau ist, umso schwerer ist er zu schätzen
- Unvollendetes Refactoring hinterlässt den Code oft schlimmer, als vorher
- Temporäre „Behelfsbrücken“, um konsistente Zwischenstände zu haben
- Zunächst auf Entkopplung von Systemteilen konzentrieren (Interfaces!)
- Große Umbauten möglichst durch frühes und ständiges Refactoring vermeiden

Referenzen

- Kent Beck: Test-Driven Development By Example. Addison-Wesley, 2003.
- Martin Fowler: Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1999.
- Joshua Kerievsky: Refactoring to Patterns. Addison-Wesley, 2004.
- Stefan Roock & Martin Lippert: Refactorings in großen Softwareprojekten. dpunkt, 2004.