

Refactoring

Programmierungsmethodik

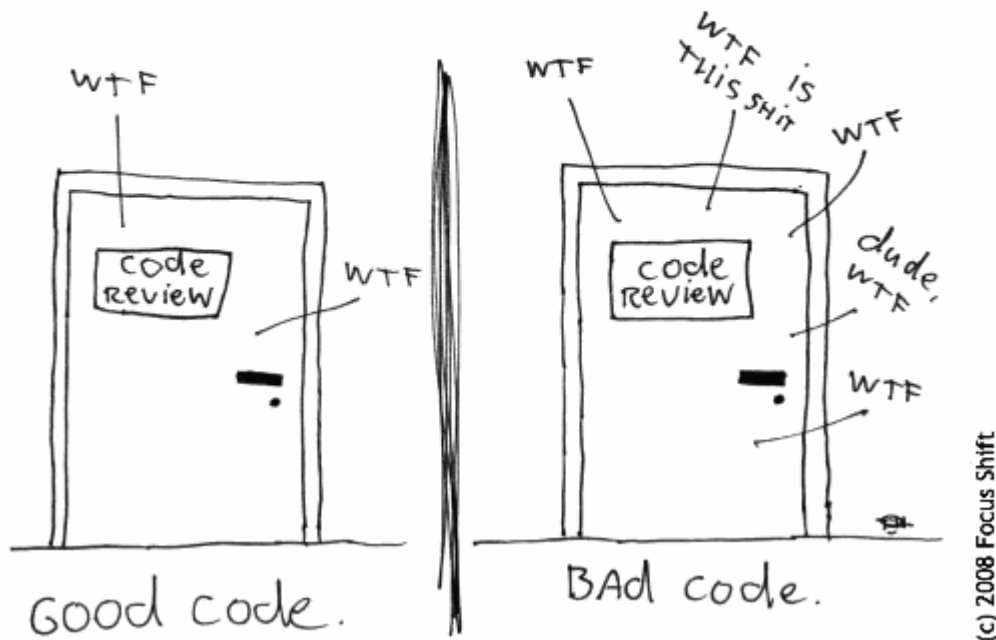
Lukas Kaltenbrunner, Simon Priller

Universität Innsbruck



Motivation

The ONLY valid measurement
of code quality: WTFs/minute



(c) 2008 Focus Shift

Clean Code

- Leicht lesbar und verständlich
- Leicht wartbar und erweiterbar
- Vorhersehbar – keine unerwarteten Seiteneffekte
- Getestet

Warum ist Clean Code wichtig?

- Produktivität
- „Unsauberer Code“ ist teuer.
- Produktivität sinkt mit der Zeit – „technical debt“ steigt.

Namensgebung

- Name sollte keine Fragen offen lassen.
- Wenn ein Kommentar für eine Variable oder bei der Verwendung einer Methode erforderlich ist, sind die gewählten Namen meist nicht aussagekräftig genug.

```
int d; // elapsed time in days
```

- Fehlinformation vermeiden (u.A. Abkürzungen)

```
Map<String, Account> accList;
```

- Unterschiede deutlich machen

```
copyChars(char[] a1, char[] a2)  
→ copyChars(char[] source, char[] destination)
```

- Aussprechbare Namen verwenden
- Namen verwenden nach denen effizient gesucht werden kann
- Methoden: beginnen mit Verben
- Klassen: Nomen
- Ein Wort pro Konzept festlegen (z.B. get vs. retrieve vs. fetch)

Beispiel (1)

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList) {  
        if (x[0] == 4) {  
            list1.add(x);  
        }  
    }  
    return list1;  
}
```



```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<>();  
    for (Cell cell : gameBoard) {  
        if (cell.isFlagged()) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

Beispiel (2)

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<>();  
    for (Cell cell : gameBoard) {  
        if (cell.isFlagged()) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```



```
public List<Cell> getFlaggedCells() {  
    return gameBoard.stream()  
        .filter(Cell::isFlagged)  
        .collect(Collectors.toList());  
}
```


Kommentare (1)

- „Kommentieren Sie schlechten Code nicht – schreiben Sie ihn um“
(Brian W. Kernighan, P.J. Plaugher)
- Erklärung im und durch Code

```
// Check to see if the employee is eligible for full benefits  
if ((employee.getFlags() & HOURLY_FLAG) && (employee.getAge() > 65))
```



```
if (employee.isEligibleForFullBenefits())
```

Kommentare (2)

- Gute Kommentare
 - Juristische Kommentare (Copyrights, etc.)
 - Erklärung der Absicht („warum“)
 - Warnung vor Konsequenzen
 - TODOs
 - Dokumentationskommentare für die öffentliche Schnittstelle
- Schlechte Kommentare
 - Kommentare die noch Fragen offen lassen bzw. mehr Fragen aufwerfen.
 - Redundante Kommentare (redundant zu Code)
 - Irreführende Kommentare
 - Veraltete Kommentare
 - Tagebuch-Kommentare (Changelogs in Quellcodedateien)
 - Auskommentierter Code

Methoden

- Eine Aufgabe pro Methode
- So klein wie möglich
- Beschreibende Namen (die mit Verben beginnen)
- DRY – Prinzip befolgen
- Unerwartete Seiteneffekte vermeiden

```
public boolean checkPassword(String username, String password) {  
    boolean authorized = isAuthorized(username, password);  
    if (authorized) {  
        session.start(); // side effect  
    }  
    return authorized;  
}
```

Unit-Tests (Wiederholung)

- Ein Konzept pro Test überprüfen.
- FIRST-Prinzipien
 - Fast: schnell ablaufen (sonst werden sie seltener ausgeführt).
 - Independent: keine Abhängigkeiten zwischen Tests.
 - Repeatable: wiederholt in jeder Umgebung ausführbar.
 - Self-Validating: durch Assertions (keine manuellen Überprüfungen).
 - Timely: zeitnah geschrieben (kurz vor oder nach Produktionscode).



Refactoring

Refactoring

- Definition nach Fowler:

- *"Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."*
- *"Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior."*

Refactoring - Ziele

- Verbesserung des Software-Designs.
- Verbesserung der Lesbarkeit und Verständlichkeit.
- Codestruktur vereinfachen, sodass Bugs leichter gefunden werden.
- Entwicklungsgeschwindigkeit erhöhen.

Refactoring - Vorgehensweise

- Entwicklungsprozess wird unterteilt in:
 - Funktionalität erweitern (inklusive Testen!)
 - Refactoring
- Beim Refactoring:
 - Sicherstellen, dass der Code, welcher bearbeitet wird, getestet ist.
 - Code schrittweise verändern, damit eventuell durch Refactoring eingeführte Fehler schnell entdeckt werden können.

Refactoring - Arten

- Gelegenheitsbezogenes Refactoring
 - Als Vorbereitung um leichter neue Features umsetzen zu können.
 - Code leichter verständlich machen.
 - Ähnlichen Code zusammenführen.
 - Vereinfachen von komplexer Logik.
 - Beim Bug-Fixen.
- Geplantes Refactoring
 - Während Code Reviews.
 - Austauschen einer Komponente (beispielsweise verwendeten Bibliothek).

Refactoring - Richtlinien

- Quellcode immer in besserem Zustand hinterlassen als er vorgefunden wurde.
- Restrukturierung von Quellcode ist nicht gleich Refactoring.
 - Restrukturierung bezeichnet Reorganisation bzw. aufräumen des Codes.
 - Refactoring ist ein Spezialfall der Restrukturierung.
- Lesbarkeit und Erweiterbarkeit stehen im Fokus, somit kann Refactoring den Code schneller machen oder verlangsamen.
- Auf vermeintliche Performanceverbesserung zugunsten der Lesbarkeit oder Erweiterbarkeit sollte zu diesem Zeitpunkt verzichtet werden.

"premature optimization is the root of all evil (or at least most of it) in programming"

The Art of Computer Programming, Volume 1: Fundamental Algorithms – Donald Ervin Knuth



Bad Smells in Code

If it stinks, change it.

Bad Smells in Code (1)


- **Mysterious Name**
 - Kryptische Namen für Methoden und Variablen verringern die Lesbarkeit.
- **Duplicated Code**
 - (Fast) identischer Code in unterschiedlichen Methoden.
- **Long Methods**
 - Je länger eine Methode ist, desto schwerer zu verstehen ist sie.
 - Sobald ein Codeblock kommentiert werden muss, um ihn zu verstehen
→ Methode auslagern
 - Meist nicht mehr als 10 Zeilen für eine Methode.
- **Lazy Element**
 - Methoden und Klassen, die keinen Mehrwert bieten.
 - Beispielsweise eine Klasse mit nur einer einfachen Methode.
- **Primitive Obsession**
 - Primitive bzw. unspezifische Typen werden anstelle spezifischer Objekte verwendet.
 - Beispielsweise Telefonnummern als String.

Bad Smells in Code (2)

- Divergent Change
 - Viele unzusammenhängende Methoden müssen verändert werden, wenn Änderungen an der Klasse vorgenommen werden.
- Shotgun Surgery
 - Änderungen führen zu vielen kleinen Änderungen in anderen Klassen.
- Feature Envy
 - Eine Methode verwendet mehr Daten eines anderen Objekts als eigene Daten.
- Data Clumps
 - Verschiedene Variablen werden immer in Kombination verwendet.
- Long Parameter List
 - Unübersichtlich, da eine Vielzahl an Parametern verwendet wird.
- Repeated Switches
 - Mehrfaches auftreten der gleichen switch-Statements oder Kette von if-Anweisungen widersprechen den objektorientierten Prinzipien.

Bad Smells in Code (3)

- Speculative Generality
 - Code, der sicherheitshalber für die Zukunft erzeugt wurde, derzeit aber nicht verwendet wird und möglicherweise auch niemals zur Anwendung kommen wird.
- Temporary Field
 - Objektvariablen, die nicht immer für den Objektzustand relevant sind.
- Middle Man
 - Klasse führt nur eine Handlung aus, indem an eine andere Klasse delegiert wird.
- Large Class
 - Zu viel Funktionalität in einer Klasse.
- Refused Bequest
 - Subklasse benötigt nur Teile der Superklasse bzw. dürfte nur Teile der Funktionalität der Superklasse bereitstellen.
- Comments
 - Kommentare werden verwendet, um Bad Smells zu übertünchen.



Refactoring-Strategien

Change Function Declaration

Der Name einer Methode sollte stets ihren Zweck bzw. ihre Aufgabe widerspiegeln.

→ Umbenennen (Lesbarkeit)

```
public BigDecimal getinvcdtlmt() {...}
```



```
public BigDecimal getInvoiceableCreditLimit() {...}
```


Separate Query from Modifier

Methode hat einen Seiteneffekt und gibt gleichzeitig einen Wert zurück.

→ Trennen von Abfrage- und Änderungsoperation
(Wiederverwendbarkeit, Wartbarkeit, Modularität, Testbarkeit)

```
public BigDecimal getTotalOutstandingAndSendBill() {  
    BigDecimal result = invoices  
        .stream()  
        .map(Invoice::getAmount)  
        .reduce(BigDecimal.ZERO, BigDecimal::add);  
    sendBill();  
    return result;  
}
```



```
public BigDecimal getTotalOutstanding() {  
    return invoices.stream()  
        .map(Invoice::getAmount)  
        .reduce(BigDecimal.ZERO, BigDecimal::add);  
}  
  
public void sendBill() {  
    emailGateway.send(formatBill(invoices));  
}
```

Preserve Whole Object

Mehrere Werte werden aus einem Objekt extrahiert, um sie als Parameter zu übergeben

→ gesamtes Objekt übergeben (kürzere Signatur; auch bei Erweiterung bereits alle Daten übergeben)

```
int low = dayTemperature.getLow();  
int high = dayTemperature.getHigh();  
boolean withinRange = forecast.withinRange(low, high);
```



```
boolean withinRange = forecast.withinRange(dayTemperature);
```

Replace Parameter with Query

Resultat eines Methodenaufrufes wird an Methode übergeben

→ Abfrage in die aufgerufene Methode schieben (kürzere Signatur)

```
int basePrice = quantity * itemPrice;  
double discountLevel = getDiscountLevel();  
double finalPrice = discountedPrice(basePrice, discountLevel);
```



```
int basePrice = quantity * itemPrice;  
double finalPrice = discountedPrice(basePrice);
```

Encapsulate Variable

Direkter Zugriff auf Objektvariable möglich.

→ Feld private setzen und nach Bedarf Getter bzw. Setter bereitstellen (Datenkapselung)

```
public String name;
```



```
private String name;  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}
```

Extract Function

Kommentar unterteilt die Methode

→ Zusammengehörige Codeblöcke in Methode auslagern.

(Wiederverwendbarkeit, Wartbarkeit, Modularität, Testbarkeit)

```
public void printOwing() {  
    printBanner();  
  
    // print details  
    System.out.println("name: " + name);  
    System.out.println("amount " + getOutstanding());  
}
```



```
public void printOwing() {  
    printBanner();  
    printDetails();  
}  
public void printDetails() {  
    System.out.println("name: " + name);  
    System.out.println("amount " + getOutstanding());  
}
```

Extract Variable

Ausdruck, der sehr komplex ist.

→ Vereinfachen durch Einführen sprechender Variablen (Lesbarkeit).

```
return order.quantity * order.itemPrice -  
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +  
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
```



```
final double basePrice = order.quantity * order.itemPrice;  
final double quantityDiscount =  
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;  
final double shipping = Math.min(order.quantity * order.itemPrice * 0.1, 100);  
return basePrice - quantityDiscount + shipping;
```

Inline Variable

Lokale Variable, die keinen Mehrwert bietet.

→ löschen (Lesbarkeit)

```
double basePrice = order.getBasePrice();  
return basePrice > 1000;
```



```
return order.getBasePrice() > 1000;
```

Replace Temp with Query

Das Verwenden einer temporären Variable zum Halten einer Berechnung

→ Auslagen in eine Methode (Wiederverwendbarkeit, Modularität, Testbarkeit)

```
public double getPrice() {  
    double basePrice = quantity * item.getPrice();  
    double discountFactor = 0.98;  
    if (basePrice > 1000) {  
        discountFactor -= 0.03;  
    }  
    return basePrice * discountFactor;  
}
```



```
public double basePrice() {  
    return quantity * item.getPrice();  
}  
public double getPrice() {  
    double discountFactor = 0.98;  
    if (basePrice() > 1000) {  
        return discountFactor -= 0.03;  
    }  
    return basePrice() * discountFactor;  
}
```


Beispiel (1)

```
public double getPrice() {  
    double basePrice = quantity * item.getPrice();  
    double discountFactor = 0.98;  
    if (basePrice > 1000) {  
        discountFactor -= 0.03;  
    }  
    return basePrice * discountFactor;  
}
```



Replace Temp with Query

```
public double getBasePrice() {  
    return quantity * item.getPrice();  
}  
  
public double getPrice() {  
    double discountFactor = 0.98;  
    if (getBasePrice() > 1000) {  
        discountFactor -= 0.03;  
    }  
    return getBasePrice() * discountFactor;  
}
```

Beispiel (2)

```
public double getBasePrice() {  
    return quantity * item.getPrice();  
}  
public double getPrice() {  
    double discountFactor = 0.98;  
    if (getBasePrice() > 1000) {  
        discountFactor -= 0.03;  
    }  
    return getBasePrice() * discountFactor;  
}
```



Replace Temp with Query

```
public double getBasePrice() {  
    return quantity * item.getPrice();  
}  
public double getDiscountFactor() {  
    double discountFactor = 0.98;  
    if (getBasePrice() > 1000) {  
        discountFactor -= 0.03;  
    }  
    return discountFactor;  
}  
public double getPrice() {  
    return getBasePrice() * getDiscountFactor();  
}
```

Beispiel (3)

```
public double getBasePrice() {  
    return quantity * item.getPrice();  
}  
public double getDiscountFactor() {  
    double discountFactor = 0.98;  
    if (getBasePrice() > 1000) {  
        discountFactor -= 0.03;  
    }  
    return discountFactor;  
}  
public double getPrice() {  
    return getBasePrice() * getDiscountFactor();  
}
```



```
public double getBasePrice() {  
    return quantity * item.getPrice();  
}  
public double getDiscountFactor() {  
    return getBasePrice() > 1000 ? 0.95 : 0.98;  
}  
public double getPrice() {  
    return getBasePrice() * getDiscountFactor();  
}
```

Split Variable

Eine lokale Variable hat mehrere Aufgaben.

→ Eine lokale Variable pro Aufgabe einführen (Wartbarkeit, Lesbarkeit)

```
double temp = 2 * (height + width);  
System.out.println(temp);  
temp = height * width;  
System.out.println(temp);
```



```
final double perimeter = 2 * (height + width);  
System.out.println(perimeter);  
final double area = height * width;  
System.out.println(area);
```

Extract Class

Trennung der Aufgaben einer Klasse nicht klar; Klasse führt verschiedene Aufgaben aus.

→ Aufteilen (eindeutige Zuständigkeit, Wiederverwendbarkeit)

```
class Person {  
    public String getOfficeAreaCode() { return areaCode; }  
    public String getOfficeNumber() { return officeNumber; }  
    ...  
}
```



```
class Person {  
    public String getOfficeAreaCode() { return telephoneNumber.getAreaCode(); }  
    public String getOfficeNumber() { return telephoneNumber.getNumber(); }  
    ...  
}
```

```
class TelephoneNumber {  
    public String getAreaCode() { return areaCode; }  
    public String getNumber() { return number; }  
    ...  
}
```

Decompose Conditional

Komplizierte if-Statements

→ Vereinfachen durch das Einführen von Methoden (Lesbarkeit, Wartbarkeit)

```
if (!date.isBefore(plan.getSummerStart() && !date.isAfter(plan.getSummerEnd())) {  
    charge = quantity * plan.getSummerRate();  
} else {  
    charge = quantity * plan.getRegularCharge() + plan.getRegularServiceCharge();  
}
```



```
if (isSummer()) {  
    charge = getSummerCharge();  
} else {  
    charge = getRegularCharge();  
}
```



```
charge = isSummer() ? getSummerCharge() : getRegularCharge();
```

Replace Conditional with Polymorphism (1)

Typ-abhängige Berechnungen in Methoden.

→ Polymorphie ausnutzen: Subklassen mit spezifischem Verhalten einführen (objektorientiert, typsicher, abstrahiert)

```
public class RegularPolygon {
    private final double sideLength;
    private final RegularPolygonType regularPolygonType;

    public RegularPolygon(double sideLength, RegularPolygonType regularPolygonType) {
        this.sideLength = sideLength;
        this.regularPolygonType = regularPolygonType;
    }

    public double getArea() {
        switch (regularPolygonType) {
            case REGULAR_TRIANGLE:
                return Math.sqrt(3) * sideLength * sideLength / 4;
            case SQUARE:
                return sideLength * sideLength;
            case REGULAR_PENTAGON:
                return Math.sqrt(5 * (5 + 2 * Math.sqrt(5))) * sideLength * sideLength / 4;
        }
        throw new IllegalStateException("Unknown regular polygon type!");
    }
}
```

Replace Conditional with Polymorphism (2)

```
public abstract class RegularPolygon {  
    private final double sideLength;  
  
    public RegularPolygon(double sideLength) {  
        this.sideLength = sideLength;  
    }  
  
    public abstract double getArea();  
  
    public double getSideLength() {  
        return sideLength;  
    }  
}
```

```
public class Square extends RegularPolygon {  
    public Square(double sideLength) {  
        super(sideLength);  
    }  
  
    @Override  
    public double getArea() {  
        return getSideLength() * getSideLength();  
    }  
}
```


Substitute Algorithm

Verboser und unflexibler Algorithmus.

→ Verwendung eines anderen Algorithmus (Lesbarkeit, Erweiterbarkeit)

```
public String foundPerson(List<String> people){  
    for (String person : people) {  
        if (person.equals("Don")) {  
            return "Don";  
        }  
        if (person.equals("John")) {  
            return "John";  
        }  
    }  
    return "";  
}
```



```
public String foundPerson(List<String> people) {  
    final Set<String> candidates = Set.of("Don", "John");  
    return people.stream()  
        .filter(candidates::contains)  
        .findFirst()  
        .orElse("");  
}
```

Pull Up Method

Zwei Subklassen implementieren das gleiche Verhalten.

→ Methoden in die Superklasse schieben (DRY)

```
class Employee {...}

class Salesperson extends Employee {
    String getName() {...}
}

class Engineer extends Employee {
    String getName() {...}
}
```



```
class Employee {
    String getName() {...}
}

class Salesperson extends Employee {...}

class Engineer extends Employee {...}
```

Pull Up Constructor Body

Subklassen haben (nahezu) idente Konstruktoren.

→ nach oben schieben (DRY)

```
public class Student extends Person {  
    public Student(String name, String discipline) {  
        this.name = name;  
        this.discipline = discipline;  
    }  
    ...  
}
```



```
public class Student extends Person {  
    public Student(String name, String discipline) {  
        super(name);  
        this.discipline = discipline;  
    }  
    ...  
}
```

Quellen

- Martin Fowler: **Refactoring: Improving the Design of Existing Code**, Addison-Wesley Professional, second edition, 2018
- Robert C. Martin: **Clean Code: A Handbook of Agile Software Craftsmanship**, Pearson, 2008