



# REFACTORING

Maurice Müller

2023-02-22

# **Einleitung**

# Was ist Refactoring?

- Überarbeitung/Verbesserung von existierendem (und funktionierendem) Code
- die Funktionalität wird *nicht* geändert
- Ziel: **bessere Codequalität**
  - erhöhte Lesbarkeit
  - erhöhte Flexibilität / Modularisierung
  - klarere Strukturen / Einführung von Strukturen
  - Vereinheitlichung

# Warum?

- Verbesserung der Struktur
  - beim Schreiben des Codes liegt der Fokus auf der Funktionalität
  - beim Refactoring liegt der Fokus auf der Struktur
- Code wird besser lesbar und wartbar
  - der zweite Entwurf ist praktisch immer besser als der erste

- Finden von Fehlern
  - beim erneuten Durchgehen werden Randfälle etc. häufiger bedacht
  - sauberer Code erleichtert Fehlersuche
- Neue Funktionalität kann schneller implementiert werden
  - zu Projektbeginn wird die Entwicklungsgeschwindigkeit wenig von der Code-Qualität beeinflusst
  - ab einer bestimmten Größe ist Qualität der entscheidende Faktor

- neuer Code reduziert meistens die Qualität
  - durch Refactoring kann die Qualität wieder angehoben werden

**Refactoring ist ein kontinuierlicher Prozess**

# Code in der Praxis

- "zielorientierte" Programmierung
  - rudimentäre inkonsistente Strukturen
  - nachträglich eingefügte Erweiterungen
    - z.B. Randfälle
  - kryptische Namen
- keine / kaum Testabdeckung

# Definition

## Refactoring

eine Änderung an der internen Struktur einer Software, um sie leichter verständlich und änderbarer zu machen ohne das Verhalten nach außen zu ändern

## Refactorisieren

eine Abfolge von Refactorings auf eine Software anwenden ohne das Verhalten nach außen zu ändern

aus dem Englischen aus dem Buch "Refactoring (Second Edition)" von Martin Fowler (S. 45)

# Vereinfachter Entwicklungsprozess

- **make it**
  - einfach(ste) Funktionalität
  - erste lauffähige Version
- **make it run**
  - Randfälle finden und beheben
  - Testen
- **make it better**
  - Refactoring

# "make it"

Aufgabe: alle Log-Files, die älter als 5 Tage sind, in einem Verzeichnisbaum finden und löschen

```
class CleanUpProcess {  
    void cleanUp(String path) {  
        File[] files = new File(path).listFiles().listFiles();  
        for (File file : files) {  
            if (file.isDirectory()) {  
                cleanUp(file.getPath());  
                continue;  
            }  
            if (file.lastModified() < (System.currentTimeMillis() - 43200000L)) {  
                file.delete();  
            }  
        }  
    }  
}
```

# "make it run"

Bugs: NullPointerException, falls path nicht existent

```
class CleanUpProcess {  
    void cleanUp(String path) {  
        File dir = new File(path);  
        if(!dir.exists() || !dir.isDirectory()) {  
            return;  
        }  
        for (File file : dir.listFiles()) {  
            if (file.isDirectory()) {  
                cleanUp(file.getPath());  
                continue;  
            }  
            if (file.lastModified() < (System.currentTimeMillis() - 43200000L)) {  
                file.delete();  
            }  
        }  
    }  
}
```

# "make it better"

Was kann verbessert werden?

```
class CleanUpProcess {  
    void cleanUp(String path) {  
        File dir = new File(path);  
        if(!dir.exists() || !dir.isDirectory()) {  
            return;  
        }  
        for (File file : dir.listFiles()) {  
            if (file.isDirectory()) {  
                cleanUp(file.getPath());  
                continue;  
            }  
            if (file.lastModified() < (System.currentTimeMillis() - 43200000L)) {  
                file.delete();  
            }  
        }  
    }  
}
```

- Konstante 5 Tage auslagern
- File#isDirectory prüft schon implizit auf Existenz → !dir.exists() kann weg
- anstatt String ein höherwertiges Objekt als Parameter (z.B. File)

```
class CleanUpProcess {

    final static long FIVE_DAYS = 432000000L;

    void cleanUp(File path) {
        if(!path.isDirectory()) {
            return;
        }
        for (File file : path.listFiles()) {
            if (file.isDirectory()) {
                cleanUp(file);
                continue;
            }
            if (file.lastModified() < (System.currentTimeMillis() - FIVE_DAYS)) {
                file.delete();
            }
        }
    }
}
```

- sprechende if-condition
- @Nullable / @Nonnull Annotationen
- Error/Info-Log

```

class CleanUpProcess {

    final static long FIVE_DAYS = 432000000L;

    void cleanUp(@Nonnull File path) {
        if(!path.isDirectory()) {
            LOGGER.error("You passed a non directory path: " + path, new RuntimeException());
            return;
        }
        for (File file : path.listFiles()) {
            if (file.isDirectory()) {
                cleanUp(file);
                continue;
            }
            if (isOlderThan5Days(file)) {
                file.delete();
            }
        }
    }

    private boolean isOlderThan5Days(@Nonnull File file) {
        return file.lastModified() < (System.currentTimeMillis() - FIVE_DAYS);
    }
}

```

Verbessere den Code durch Refactorings so lange, bis er genau sagt, was er sagen soll / was er macht.

```
import java.io.File;
import java.time.temporal.ChronoUnit;

class CleanUpProcess {

    void cleanUp(@Nonnull File path) {
        streamFilesIn(path)
            .filter(olderThan(5, ChronoUnit.DAYS))
            .forEach(File::delete);
    }
}
```

# Wann?

Einfache Merkregel: **Three strikes and you refactor**

- beim ersten Programmieren einer Funktionalität neu entwickeln
- beim zweiten Programmieren einer *ähnlichen* Funktionalität → Copy&Paste mit Anpassung
- beim dritten Programmieren einer *ähnlichen* Funktionalität → refactorisieren

- bei einer Code-Review
  - man steckt sowieso schon tief drin
- vor dem Hinzufügen neuer Funktionalität
  - Code so umschreiben, dass die Implementierung leichter fällt
- beim Beseitigen eines Fehlers
  - Testabdeckung muss sowieso erhöht werden
  - Softwarefehler unterliegen dem *Lokalitätsprinzip*

# EXKURS: Lokalitätsprinzip

**Ein Fehler kommt selten allein**

- Wo ein Bug ist, da sind noch andere
- Warum treten Bugs häufiger 'im Rudel' auf?
  - komplexe Bereiche enthalten mehr Bugs
  - Fehler werden teilweise durch 'Gegenfehler' kompensiert (Symptonbehebung)
  - zusammenhängender Code wurde meistens vom gleichen Entwickler geschrieben
  - zusammenhängender Code wurde meistens zeitnah geschrieben
    - an einem schlechten Tag entstehen mehr Fehler

# Warum funktioniert Refactoring?

- frühere, jetzt unpassende Entscheidungen werden korrigiert
- neues Wissen wurde gesammelt, wie man es besser macht
- Applikationen sind schwieriger zu lesen als zu schreiben
  - schwer lesbarer Code ist schlecht änderbar
  - komplexe Applikationen sind schlecht änderbar
- Code-Verbesserungen haben einen Langzeit-Effekt
  - mittlere Entwicklungsgeschwindigkeit, aber dafür dauerhaft
    - im Gegensatz zu: am Anfang sehr schnell, dann immer langsamer

# Wie sage ich das meinem Chef?

- Gar nicht!
  - im besten Fall versteht er den Nutzen sofort
  - im schlechtesten Fall sieht er nur den Konflikt zwischen Termin/Budget und qualitätsbewusster Arbeitsweise
  - → Entwickler müssen selbst verantwortungsbewusst sein
- gehört zur professionellen Arbeitsweise
  - Entwickler sollen möglichst schnell fehlerfreien Code produzieren → Betonung auf *fehlerfrei* und nicht *möglichst schnell*
  - wie Code entwickelt wird, muss/sollte das Management nichts angehen
  - Refactoring erhöht langfristig die Entwicklungsgeschwindigkeit und hilft den Zeitplan einzuhalten

# Nachteile / Fallstricke beim Refactoring

- zentrale Designänderungen
  - Stillstand des gesamten Projektteams und/oder hoher Merge-Aufwand
  - erhöhte Gefahr von neuen Fehlern trotz Testabsicherung
  - bei zentralen Stellen lohnt sich schon zu Beginn mehr Zeit in das Design zu investieren
- Öffentliche Schnittstellen
  - Abhilfe: alte und neue Schnittstelle parallel anbieten
- Datenbank-Schema
  - zusätzlich Migration bereits vorhandener Daten
  - Milderung: explizite Zugriffsschicht auf DB

# Nachteile / Fallstricke beim Refactoring (fortgeführt)

- Zeitverbrauch
  - auch schnelles Refactoring benötigt Zeit
  - Kompensation: danach schnellere Entwicklung
- neue Fehler
  - auch mit Tests können bei Änderungen Fehler gemacht werden
  - Kompensation: Positive Effekte der Code-Review und erschweren von Fehlern bei neuen Implementierungen
- Performance
  - manche Refactorings wirken sich negativ auf die Performance aus

# EXKURS: Performance-Optimierung

*Premature optimization is the root of all evil.*

– Donald Knuth

- 90/10-Effekt (ähnlich Pareto-Prinzip)
  - 90% der Zeit wird in 10% des Codes verbraucht
- 3 Regeln für Optimierungen
  - Don't (Mach es nicht)
  - Not yet (Mach es später)
  - Measure (Analyse durch Messen)

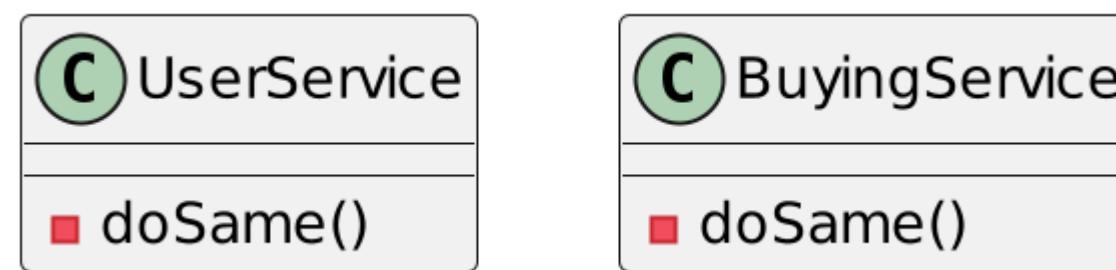
# Code Smells

- **If it stinks, change it.**
  - "Code Smells" deuten auf verbesserungswürdige Stellen im Code hin
- Einstiegsstelle für Refactorings
  - konkrete Missstände mit konkreten Lösungen
- Schlecht messbar
  - teilweise Unterstützung durch Tools oder Algorithmen
  - häufig Erfahrungswerte

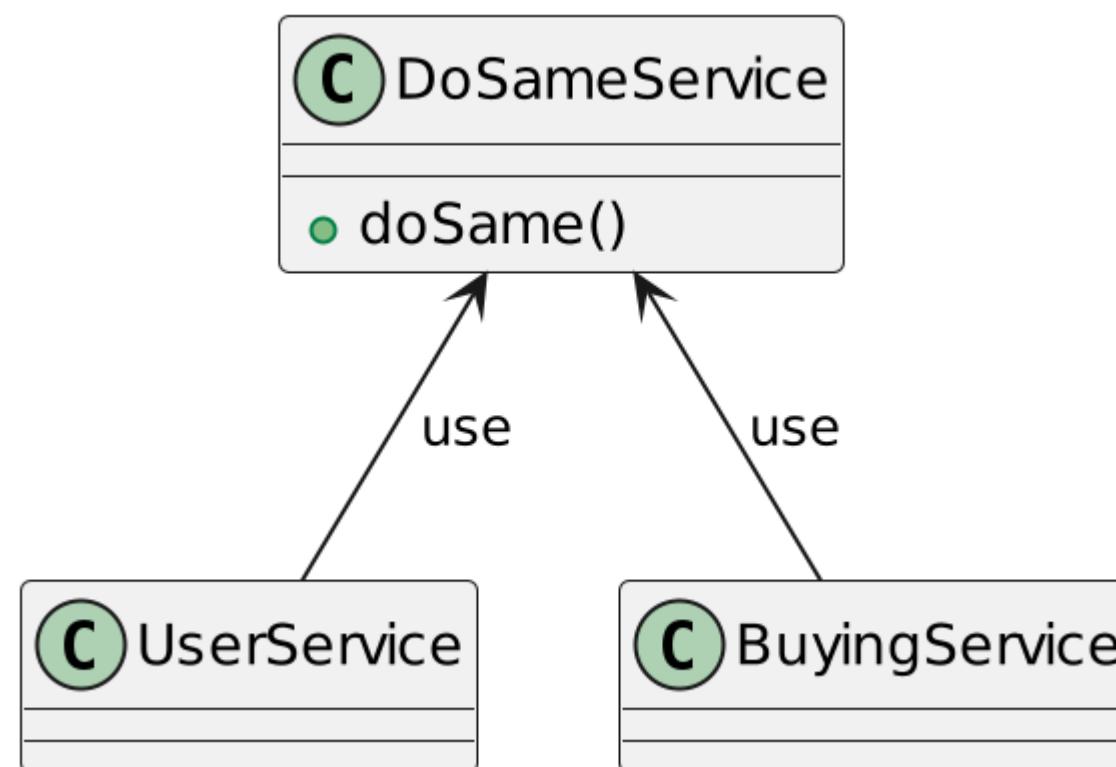
# Code Smell: Duplicated Code

- doppelt vorhandener Code
  - entweder durch Copy&Paste oder durch unbewusste doppelte Implementierung
- Problem
  - Änderung an einer Stelle führt nicht automatisch zur Änderung an der anderen Stelle → doppelter Pflegeaufwand und Divergieren der Logik
- Lösung
  - doppelte Stellen auslagern und neuen Aufruf referenzieren

# Ausgangslage



# Lösung



# Code Smell: Long Method

- lange Methoden
- kürzere Methoden "leben" länger
- lange Methoden sind schwierig zu verstehen
- Lösung
  - Methode aufspalten
  - gute Benennung der neuen Methode(n) führt zu besserer Lesbarkeit
  - beim Aufspalten an Kommentare orientieren
    - oder immer wenn man das Bedürfnis hat, einen Kommentar zu schreiben → in neue aussagekräftige Methode auslagern
  - Konditionalstrukturen und Schleifen sind ebenfalls gute Stellen

# Code Smell: Long Method

Große Anhaltspunkte für *zu lang*:

- > 10 Zeilen
- > Bildschirmseite
- nicht auf einen Blick verständlich

**Es gibt hier keine feste Metrik.**

# Long Method: Beispiel

```
public class User {  
  
    private LocalDate birthday;  
  
    public boolean allowedToBuy(Country country, Beverage type) {  
        LocalDate now = LocalDate.now();  
        long age = ChronoUnit.YEARS.between(birthday, now);  
        int minimumAge = switch (country) {  
            case USA_CALIFORNIA → 21;  
            case GERMANY → switch (type) {  
                case BEER → 16;  
                case WINE → 16;  
                case WHISKEY → 18;  
                case VODKA → 18;  
                case RUM → 18;  
            };  
            case ITALY → 18;  
        };  
        LocalTime timeNow = LocalTime.now();  
        boolean allowedToBuyNow = switch (country) {  
            case USA_CALIFORNIA → timeNow.isBefore(LocalTime.of(2, 0)) || timeNow.isAfter(LocalTime.of(23, 0));  
            case GERMANY → timeNow.isAfter(LocalTime.of(5, 0)) && timeNow.isBefore(LocalTime.of(23, 0));  
            case ITALY → timeNow.isAfter(LocalTime.of(6, 0));  
        };  
        return age ≥ minimumAge && allowedToBuyNow;  
    }  
}
```

# Long Method: Beispiel

*in kleinere Methoden zerlegt*

```
public class User {

    private LocalDate birthday;

    public boolean allowedToBuy(Country country, Beverage type) {
        long age = getAge();
        int minimumAge = getMinimumAge(country, type);
        boolean allowedToBuyNow = isAllowedToBuyNow(country);
        return age ≥ minimumAge && allowedToBuyNow;
    }

    private boolean isAllowedToBuyNow(Country country) {
        LocalTime timeNow = LocalTime.now();
        return switch (country) {
            case USA_CALIFORNIA → timeNow.isBefore(LocalTime.of(2, 0)) || timeNow.isAfter(LocalTime.of(3, 0));
            case GERMANY → timeNow.isAfter(LocalTime.of(5, 0)) && timeNow.isBefore(LocalTime.of(6, 0));
            case ITALY → timeNow.isAfter(LocalTime.of(6, 0));
        };
    }

    private int getMinimumAge(Country country, Beverage type) {
        return switch (country) {
            case USA_CALIFORNIA → 21;
            case GERMANY → switch (type) {
                case BEER → 16;
                case WINE → 16;
                case WHISKEY → 18;
            }
        };
    }
}
```

# Long Method: Beispiel

*Methoden ausgelagert (Information Expert)*

```
public class User {  
  
    private LocalDate birthday;  
  
    public boolean allowedToBuy(Country country, Beverage type) {  
        long age = getAge();  
        int minimumAge = country.getMinimumAgeFor(type);  
        boolean allowedToBuyNow = country.isAllowedToBuyNowAlcohol();  
        return age ≥ minimumAge && allowedToBuyNow;  
    }  
  
    private long getAge() {  
        LocalDate now = LocalDate.now();  
        return ChronoUnit.YEARS.between(birthday, now);  
    }  
}
```

# Long Method: Beispiel

## Optische Anpassung

```
public class User {  
  
    private LocalDate birthday;  
  
    public boolean allowedToBuy(Country country, Beverage type) {  
        return getAge() ≥ country.getMinimumAgeFor(type) &&  
               country.isAllowedToBuyNowAlcohol();  
    }  
  
    private long getAge() {  
        LocalDate now = LocalDate.now();  
        return ChronoUnit.YEARS.between(birthday, now);  
    }  
}
```

# Code Smell: Long Parameter List

- viele Parameter in der Methodensignatur
- Problem
  - schwieriger zu lesen/verstehen
  - Beispiel: myMethod(name, age, city, true)
- Lösung
  - ursprüngliches Objekt reingeben (z.B. user)
  - Parameter-Objekt erstellen
    - weiterer Vorteil: bei Durchreichen des Parameter-Objekts ist eine Änderung der Parameter leichter durchzuführen
  - Parameter mit Funktion ersetzen
    - z.B. myMethod(this :: extractStuff)

# Long Parameter List: Beispiel

```
public class EmployeeManagement {  
  
    public void update(long id, String firstName, String lastName, int age, double salary,  
                      String department, boolean isFullTime,  
                      String email, short yearsInCompany, int[] performanceScores,  
                      double[] monthlySalaries, String[] previousPositions,  
                      boolean[] completedTrainingModules, long[] trainingSessionIds,  
                      float[] yearlyBonuses, char[] employeeGrade,  
                      byte[] departmentCodes, short[] teamSizes,  
                      List<String> certifications, Map<Integer, String> emergencyContacts) {  
        updateBaseData(firstName, lastName, age, department, isFullTime, emergencyContacts, ye  
        updateSalaryData(monthlySalaries, salary, yearlyBonuses);  
        // ...  
    }  
}
```

# Long Parameter List: Beispiel

*Aufteilung in mehrere 'Parameter-Objekte'*

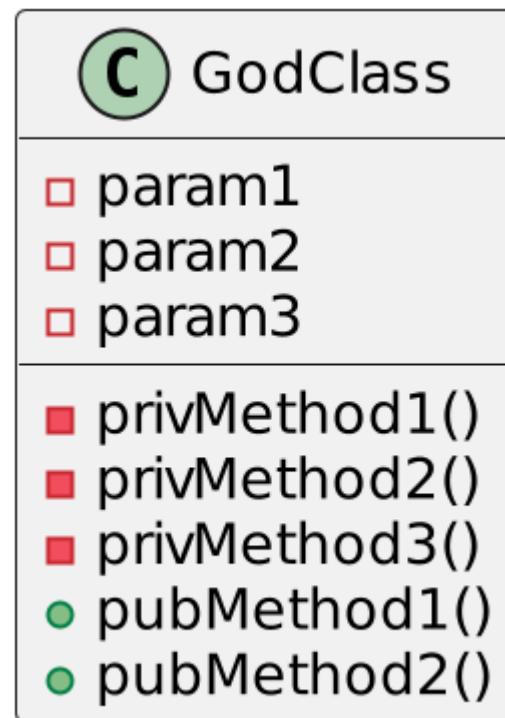
```
public class EmployeeManagement {  
  
    public void update(long id, BaseData baseData, SalarayData salaryData,  
                      TrainingData trainingData, ScoreData scoreData) {  
        updateBaseData(baseData);  
        updateSalarayData(salaryData);  
        // ...  
    }  
}
```

- deutlich übersichtlicher
- Hinzufügen von weiteren Parametern innerhalb der Parameter-Objekte werden automatisch weitergegeben

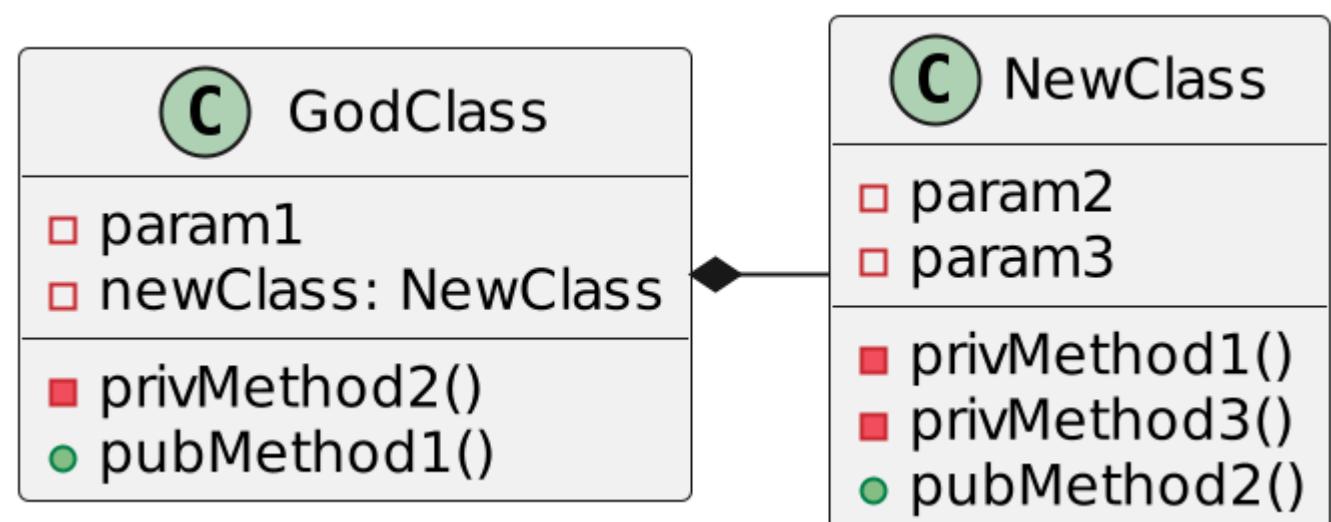
# Code Smell: Large Class

- große Klassen → Worst-Case: God-Class
- häufige Symptome
  - zu viele Instanzvariablen
  - zu viele Methoden
  - stark erhöhte Zeilenanzahl
- Verletzung von OO-Prinzipien
  - z.B. SRP, Low Coupling, High Cohesion, DIP
- Lösung
  - Teilfunktionalität in eigene Klassen auslagern

# Ausgangslage



# Lösung



# Code Smell: Shotgun Surgery

- deutsch: Schrotflinten-Operation
- um eine Änderung zu machen, müssen viele unterschiedliche Klassen / Code-Stellen angepasst werden
- kann sich mit Duplicated Code überschneiden
- Problem
  - es ist leicht, eine oder mehrere der Stellen zu übersehen
- Lösung
  - Code umstrukturieren (z.B. verteilte Logik in eine eigene Klasse extrahieren)

# Shotgun Surgery: Beispiel 1

```
@UrlBinding("/home")
class HomePage {
    // ...
}

@UrlBinding("/user/settings")
class UserSettingsPage {
    void onGetRequest(User user) {
        if(user == null) {
            redirect("/home");
            return;
        }
        // ...
    }
}

class GlobalExceptionHandler {
    void onException(Exception e) {
        // do exception handling
        redirect("/home");
    }
}
```

- Anforderung: Änderung der URL der HomePage

# Shotgun Surgery: Beispiel 1

```
@UrlBinding("/home")
class HomePage {
    static final String URL = "/home";
// ...
}

@UrlBinding("/user/settings")
class UserSettingsPage {
    void onGetRequest(User user) {
        if(user == null) {
            redirect(HomePage.URL);
            return;
        }
        // ...
    }
}

class GlobalExceptionHandler {
    void onException(Exception e) {
        // do exception handling
        redirect(HomePage.URL);
    }
}
```

- Anforderung: Änderung der URL der HomePage

# Shotgun Surgery: Beispiel 2

```
public class Order {  
    public void processOrder() {  
        // ...  
        log("Order processed");  
    }  
  
    private void log(String message) {  
        System.out.println("Order: " + message);  
    }  
}  
  
public class Payment {  
    public void processPayment() {  
        // ...  
        log("Payment processed");  
    }  
  
    private void log(String message) {  
        System.out.println("Payment: " + message);  
    }  
}  
  
public class CustomerService {  
    public void handleInquiry() {  
        // ...  
        log("Inquiry handled");  
    }  
  
    private void log(String message) {
```

# Shotgun Surgery: Beispiel 2

```
public class Logger {  
    public static void log(String message) {  
        System.out.println(message);  
    }  
}  
  
public class Order {  
    public void processOrder() {  
        // ...  
        Logger.log("Order: Order processed");  
    }  
}
```

# Code Smell: Code Comments

- (inline) Code-Kommentare
- häufig Deodorant für andere Smells
  - z.B. Erläuterung eines Code-Blocks innerhalb einer Methode → kann auf Long Method hindeuten und als Lösung Extract Method
- nach einem Refactoring sind Kommentare häufig überflüssig
- es gibt auch sinnvolle Kommentare (z.B. Dokumentation von Annahmen/Unsicherheiten, etc.)

# Code Smell: Switch Statement

- Problem
  - Komplexität: Switch Statements wachsen in der Regel nur in der Größe und werden damit komplexer und unübersichtlicher
  - Duplikation: häufig gleiche oder ähnliche Switch Statements
  - fehleranfällige Syntax
    - break vs. fall-through, Vergessen von Fällen
- ähnlich sind größere if/else-Kaskaden
- Lösungen
  - Polymorphie
  - in Methoden extrahieren

nicht zu verwechseln mit Switch-Expression

# Switch Statement: Beispiel

```
switch (playerInput) {  
    case 1:  
        System.out.println("You chose to explore the forest.");  
        System.out.println("Walking through the forest, you find a mysterious cave.");  
        System.out.println("Inside the cave, you find a treasure chest.");  
        System.out.println("You open the chest and find a golden sword.");  
        System.out.println("As you take the sword, you hear something behind you...");  
    case 2:  
        System.out.println("You chose to sail the seas.");  
        System.out.println("You board a ship and set sail.");  
        System.out.println("A storm hits and your ship is thrown off course.");  
        System.out.println("You end up on an uncharted island.");  
        System.out.println("They offer you a map to hidden treasures.");  
        break;  
    case 3:  
        System.out.println("You chose to visit the marketplace.");  
        System.out.println("You buy some exotic spices and a mysterious artifact.");  
        System.out.println("A merchant tells you about a secret guild of adventurers.");  
        System.out.println("You decide to seek out this guild for your next adventure.");  
        break;  
    default:  
        System.out.println("You chose to rest at the inn.");  
        System.out.println("You enjoy a peaceful night at the inn.");  
        System.out.println("Your health and energy are fully restored.");  
        System.out.println("The next morning, you feel ready for a new adventure.");  
        break;  
}
```

# Switch Statement: Beispiel

- Auslagern in eigene Methoden
- Code wird leichter zu lesen
- *fall-through* ist leichter zu erkennen

```
switch (playerInput) {  
    case 1:  
        exploreForest();  
    case 2:  
        sailToSea();  
        break;  
    case 3:  
        visitMarketPlace();  
        break;  
    default:  
        restAtInn();  
        break;  
}
```

# Code Smell: Middle Man

- Mittelsobjekt
- Datenkapselung und Geheimnisprinzip resultieren häufig in Delegation
- Problem: zu viel Delegation führt zu aufgeblähtem Code
- Lösungen
  - Mittelsobjekt entfernen und direkt mit dem eigentlichen Objekt reden
  - Methoden einbetten (`inline method`)

# Middle Man: Beispiel

```
class PlayerCharacter {  
    GameWorld gameWorld;  
    void openChest(Chest chest) {  
        gameWorld.interactWithChest(chest);  
    }  
}  
  
class GameWorld {  
    void interactWithChest(Chest chest) {  
        chest.open();  
    }  
}  
  
class Chest {  
    List<Item> items;  
    void open() {  
        // ...  
    }  
}
```

# Middle Man: Beispiel

- *Middle Man* kann eingespart werden
- Was wäre eine weitere Code-Verbesserung?

```
class PlayerCharacter {  
    void openChest(Chest chest) {  
        chest.open();  
    }  
}  
  
class Chest {  
    List<Item> items;  
    void open() {  
        // ...  
    }  
}
```

# Middle Man: Beispiel

- Open-Closed-Principle (OCP)

```
class PlayerCharacter {  
    void open(Openable thing) {  
        thing.open();  
    }  
}  
  
interface Openable {  
    void open() {  
        // ...  
    }  
}
```

# Code Smell: Method Chains

- Aufrufketten
  - `order.getBuyer().getBankAccount().getBank().getName()`
- Problem
  - Aufrufer ist stark an die Struktur der Kette gekoppelt
    - Änderungen in der Kette bedeutet Änderung beim Aufrufer
  - fehleranfällig (v.a. NullPointerExceptions)
- Lösungen
  - falls möglich, Funktionen/Methoden extrahieren oder verschieben
  - Delegation (z.B. `order.getBuyerBankName()` → delegiert intern den Aufruf weiter)
    - Achtung vor dem Middle Man

# Code Smell: Method Chains



# Refactorings

kleine Auswahl aus über 60 Refactorings

# Refactoring: Extract Method

**Problem** eine Methode ist (zu) groß und unverständlich

**Lösung** Auslagern in einzelne Methode mit einem aussagekräftigen Namen

## Effekte

- feingranularer Code (SRP, High Cohesion)
- bildet Abstraktionsstufen aus
  - "höhere" Methoden sind näher an Problemdomäne und natürlicher Sprache
- Wiederverwendung ist wahrscheinlicher

**Hilft gegen:** Duplicated Code, Long Method, Code Comments

# Extract Method: Beispiel

```
void printMovies(FilmStudio studio) {
    Logo logo = logoRepository.get(studio);
    Image logoImg = new Image(1200, 900, logo);
    printImage(logoImg);

    // print all titles with price (and not on index)
    for (Movie movie : getMovies(studio)) {
        if(movie.isOnIndex()) {
            continue;
        }
        printLine(movie.title() + " " + movie.price());
    }

    print(studio.name());
    print(studio.address().firstLine());
    print(studio.address().secondLine());
    if(studio.address().thirdLine() != null) {
        print(studio.address().thirdLine());
    }
    if(studio.phone() != null) {
        print(studio.phone());
    }
}
```

```

void printMovies(FilmStudio studio) {
    printLogo(studio);
    printTitlesNotOnIndexWithPrice(studio);
    printFooter(studio);
}

void printLogo(FilmStudio studio) {
    Logo logo = logoRepository.get(studio);
    Image logoImg = new Image(1200, 900, logo);
    printImage(logoImg);
}

void printTitlesNotOnIndexWithPrice(FilmStudio studio) {
    for (Movie movie : getMovies(studio)) {
        if(movie.isOnIndex()) {
            continue;
        }
        printLine(movie.title() + " " + movie.price());
    }
}

```

- Code-Kommentar braucht man nicht mehr
- Abstraktionsebene etabliert
- Methode verkürzt

# Extract Methods: Tipps

- Erkennen der semantisch zusammenhängenden Code-Blöcke
  - Orientierung an Landmarken → *for*-Schleifen, *if*-Statements, Code-Kommentare, Leerzeilen
- Lokale Variablen können Extraktion erschweren
  - Methoden können nur einen Rückgabewert haben
  - Rückgabe-Typ der extrahierten Methode ist eventuell ein neuer, komplexer Typ
  - Möglichst auf In-/Out-Parameter verzichten
- Iteratives Vorgehen
  - Nach dem Extrahieren der kleinsten Code-Blöcke bilden sich wieder Extraktionspunkte

# Refactoring: Extract Class

**Problem** eine Klasse ist (zu) groß und unverständlich

**Lösung** Auslagern in einzelne Klassen

## Effekte

- feingranularer Code (SRP, High Cohesion)
- bildet Abstraktionsstufen aus
  - "höhere" Methoden sind näher an Problemdomäne und natürlicher Sprache
- Wiederverwendung ist wahrscheinlicher

**Hilft gegen:** Duplicated Code, Long Class

# Extract Class: Beispiel

```
class LandingPage {

    public void printBannerAd() {
        FilmStudio studio = getRandomStudio();
        printMovies(studio);
    }

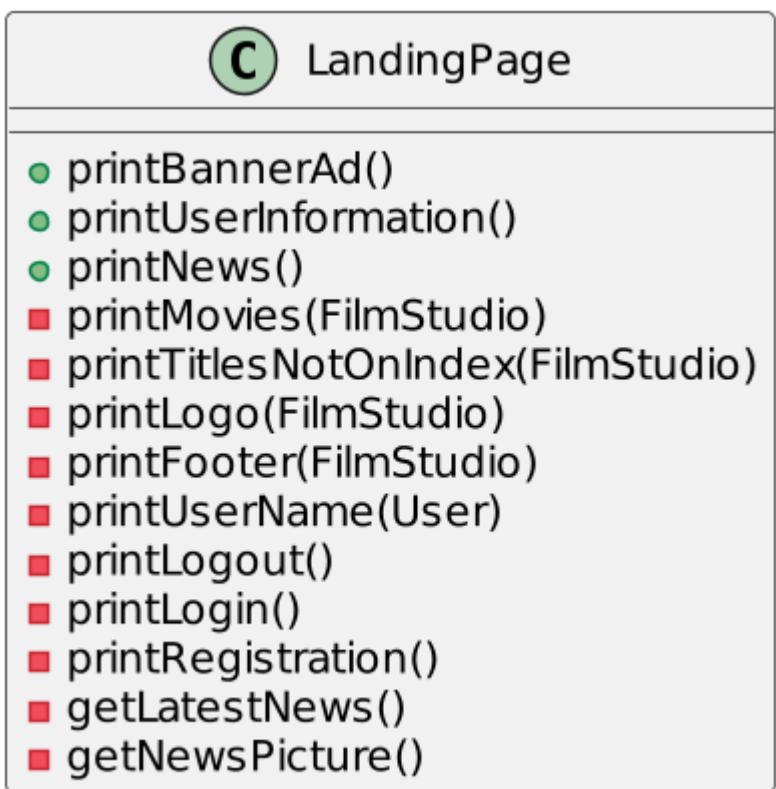
    void printMovies(FilmStudio studio) {/* ... */}
    void printLogo(FilmStudio studio) {/* ... */}
    void printTitlesNotOnIndexWithPrice(FilmStudio studio) { /* ... */ }
    void printFooter(FilmStudio studio) { /* ... */ }

    public void printUserInformation() {
        if(loggedIn()) {
            printUserName(getUser());
            printLogout();
        } else {/* ... */}
    }

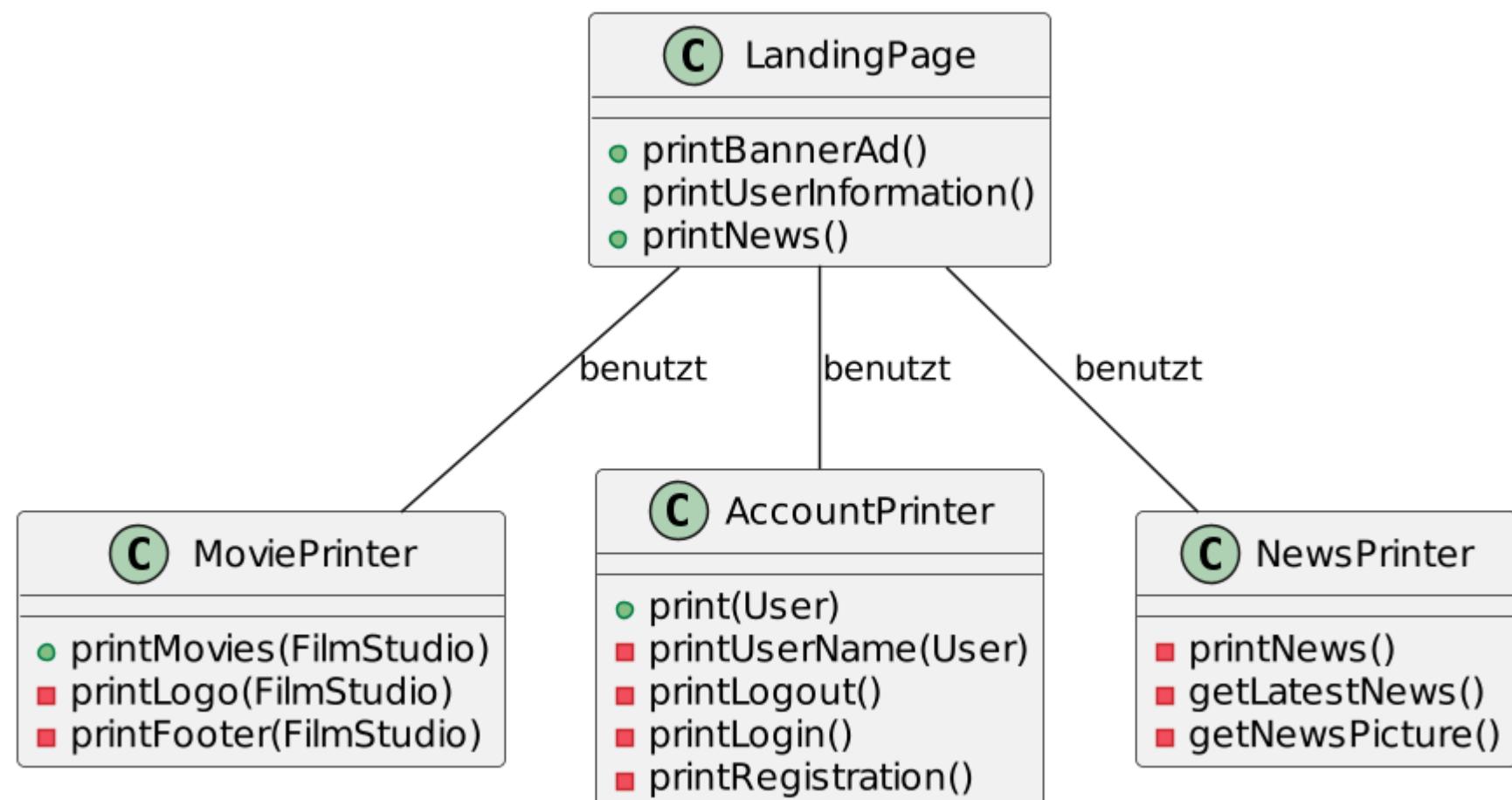
    void printUserName(User user) {/*...*/}
    void printLogout() {/* ... */}

    // ...
}
```

- Klassenstruktur kann ebenfalls Hinweise geben



- Klassenstruktur kann ebenfalls Hinweise geben



# Extract Class: Tipps

- Erkennen der fachlich zusammenhängenden Methoden
  - Orientierung an Landmarken → ähnliche Namen, ähnliche Parameter, ähnliche Rückgabewerte
- Iteratives Vorgehen
  - nach dem Extrahieren können sich wieder Extraktionspunkte bilden

# Refactoring: Rename Method

**Problem** Methodename ist kryptisch, nicht sprechend oder nicht passend

**Lösung** Methodennamen ändern → moderne IDEs unterstützen das Refactoring

## Effekte

- erhöhte Lesbarkeit
  - leichtes Lesen ist wichtiger als leichtes Schreiben → **Code wird häufiger gelesen als geschrieben**
- Selbstdokumentation des Codes

**Hilft gegen:** Code Comments

# Rename Method: Beispiel

## Vorher

```
Money getLmtCC() {  
    return account.getCreditCard().getLimit();  
}  
  
String getMsmtSensDescr() {  
    return measurement.getSensor().getDescription();  
}
```

## Nachher

```
Money getCreditCardLimit() {  
    return account.getCreditCard().getLimit();  
}  
  
String getMeasurementSensorDescription() {  
    return measurement.getSensor().getDescription();  
}
```

# Rename Method: Tipps

- Methodennamen sind quasi Code-Kommentare
  - Name sollte die Intention möglichst genau wiedergeben
- kleine Verbesserungsmöglichkeiten sind auch gut
  - z.B. `List<MovieTitle> getTitles(FilmStudio studio)` →  
`List<MovieTitle> getTitlesOf(FilmStudio studio)`
  - → führt zu `getTitlesOf(warnerBros)`

# Refactoring: Replace Temp With Query

## Problem

eine (temporäre) Variable wird benutzt, um das Ergebnis einer Berechnung zu speichern

## Lösung

Berechnung in Methode auslagern → Methode aufrufen, anstatt Variable zu lesen

## Effekte

- Seitenfreiheit der Berechnung wird geklärt
  - Schreibzugriffe auf lokale Variablen werden sichtbar
- Extract Method kann leichter angewandt werden

Hilft gegen: Long Method

# Replace Temp With Query: Beispiel

```
public class GameScoreCalculator {  
    private int baseScore;  
    private int level;  
    private int timeLeft;  
  
    public GameScoreCalculator(int baseScore, int level, int timeLeft) {  
        this.baseScore = baseScore;  
        this.level = level;  
        this.timeLeft = timeLeft;  
    }  
  
    public int calculateFinalScore() {  
        int finalScore = baseScore;  
        if (level > 5) {  
            finalScore += level * 150;  
        } else {  
            finalScore += level * 100;  
        }  
        if (timeLeft > 60) {  
            finalScore += timeLeft * 5;  
        } else {  
            finalScore += timeLeft * 3;  
        }  
        return finalScore;  
    }  
}
```

```
public class GameScoreCalculator {  
    // ...  
    public int calculateFinalScore() {  
        return baseScore + getLevelBonus() + getTimeBonus();  
    }  
  
    private int getLevelBonus() {  
        if (level > 5) {  
            return level * 150;  
        }  
        return level * 100;  
    }  
  
    private int getTimeBonus() {  
        if (timeLeft > 60) {  
            return timeLeft * 5;  
        }  
        return timeLeft * 3;  
    }  
}
```

# Refactoring: Replace Conditional With Polymorphism

**Problem** unübersichtliche Konditionalstrukturen in Abhängigkeit eines Parameters

**Lösung** Logik auslagern in Unterklassen und originale Methode abstrakt definieren

## Effekte

- Vermeidung einer Wiederholung der Konditionalstruktur
- leicht änderbar und erweiterbar (OCP)

**Hilft gegen:** Switch-Statements

# Replace Conditional With Polymorphism: Beispiel

## Vorher

```
public class CarOrder {

    private Car car = /* ... */;

    public double calculatePrice() {
        switch (this.car.type()) {
            case BMW:
                return 77777.99;
            case MERCEDES:
                return 99999.99;
        }
        return 0.0;
    }

    public Country getBuildCountry() {
        switch (this.car.type()) {
            case BMW:
                return Country.Bavaria;
            case MERCEDES:
                return Country.BW;
        }
        return null;
    }
}
```

## Nachher

```
public class CarOrder {  
  
    private Car car = /* ... */;  
  
    public double calculatePrice() {  
        return car.price();  
    }  
  
    public Country getBuildCountry() {  
        return car.buildCountry();  
    }  
}
```

*Middle Man kann auch gleich entfernt werden*

# Replace Conditional With Polymorphism: Tipp

- zusätzlich zur Polymorphie eine Lookup-Datenstruktur verwenden
  - in Java: Map bzw. HashMap
  - ermöglicht ein dezentrales und dynamisches Befüllen
  - bisheriger Switch-Parameter wird Lookup-Schlüssel

```
public class ExtraPrices {  
  
    private final Map<Extra, Price> prices = new HashMap<>();  
  
    public Price getPrice(Extra extra) {  
        return prices.get(extra);  
    }  
  
    public void addPriceFor(Extra extra, Price price) {  
        prices.put(extra, price);  
    }  
}
```

# Replace Conditional With Polymorphism: Tipp (fortgesetzt)

## Verwenden von switch-Expressions anstatt switch-Statements

```
public class CarOrder {  
  
    private Car car = /* ... */;  
  
    public double calculatePrice() {  
        return switch (this.car.type()) {  
            case BMW:  
                return 77777.99;  
            case MERCEDES:  
                return 99999.99;  
        };  
    }  
}
```

- der Compiler kann auf fehlende Typen hinweisen → es werden alle Stelle gefunden

# Refactoring: Replace Error Code With Exception

## Problem

ein oder mehrere spezielle Rückgabewerte werden im Fehlerfall zurückgegeben

## Lösung

Exception werfen

## Effekte

- trennt Fehlerfall vom Normalfall
- Rückgabewerte müssen nicht nach Fehlern geprüft werden
- versehentliche Verarbeitung des Fehler-Rückgabewerts nicht möglich
  - der Fehler-Rückgabewert muss aus dem Wertebereich des Rückgabetyps stammen → Exceptions nicht

# Replace Error Code With Exception: Beispiel

## Vorher

```
public class CarOrder {  
  
    private Car car = /* ... */;  
  
    public double calculatePrice() {  
        switch (this.car.type()) {  
            case BMW:  
                return 77777.99;  
            case MERCEDES:  
                return 99999.99;  
        }  
        return 0.0;  
    }  
}
```

Im nicht-abgefangenem Fehlerfall wird das Auto für 0,00€ verkauft.

## Nachher

```
public class CarOrder {

    private Car car = /* ... */;

    public double calculatePrice() {
        switch (this.car.type()) {
            case BMW:
                return 77777.99;
            case MERCEDES:
                return 99999.99;
        }
        throw new RuntimeException("Unhandled car type: " + car.type());
    }
}
```

# Refactoring: Replace Inheritance With Delegation

## Problem

eine Unterklasse verwendet nur einen Teil der Oberklasse

## Lösung

die Oberklasse wird zur Instanzvariablen (Delegation)

## Effekte

- klarere Schnittstelle
- keine "unbenutzbaren" Methoden
  - und damit keine Verletzung von LSP mehr
- losere Kopplung

# Replace Inheritance With Delegation: Beispiel

## Vorher

```
public class MyStack<T> extends ArrayList<T> {

    public MyStack() {
        super();
    }

    public void push(T element) {
        add(0, element);
    }

    public T pop() {
        return remove(0);
    }
}
```

- List/ArrayList bietet zu viele Methoden, die in diesem Kontext kontraproduktiv sind
    - z.B. remove(int)
- für Stack reicht: push, pop, getSize und isEmpty

# Nachher

```
public class MyStackWithDelegation<T> {  
    private final ArrayList<T> elements;  
  
    public MyStackWithDelegation() {  
        super();  
        this.elements = new ArrayList<T>();  
    }  
  
    public void push(T element) {  
        this.elements.add(0, element);  
    }  
  
    public T pop() {  
        return this.elements.remove(0);  
    }  
  
    public int getSize() {  
        return this.elements.size();  
    }  
  
    public boolean isEmpty() {  
        return this.elements.isEmpty();  
    }  
}
```

# Weiterführende Bücher

