

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() - 432000000L)) {  
                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 → refaktorisieren

- 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 (Symptombehebung)
 - 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)

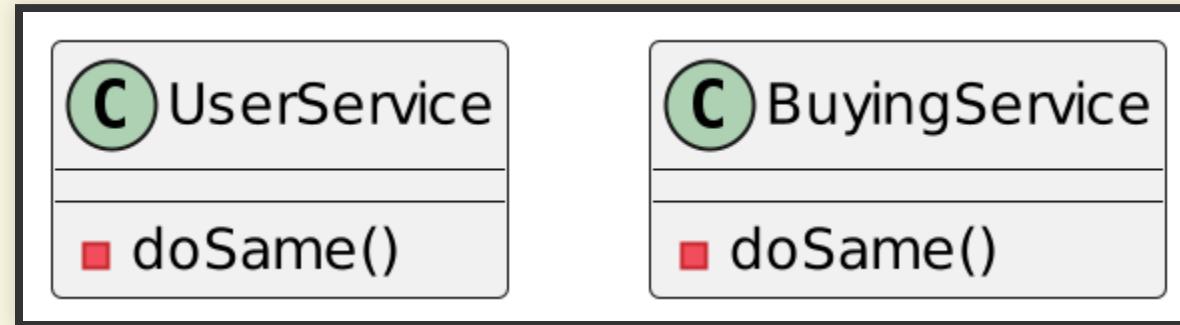
CODE SMELLS

- 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

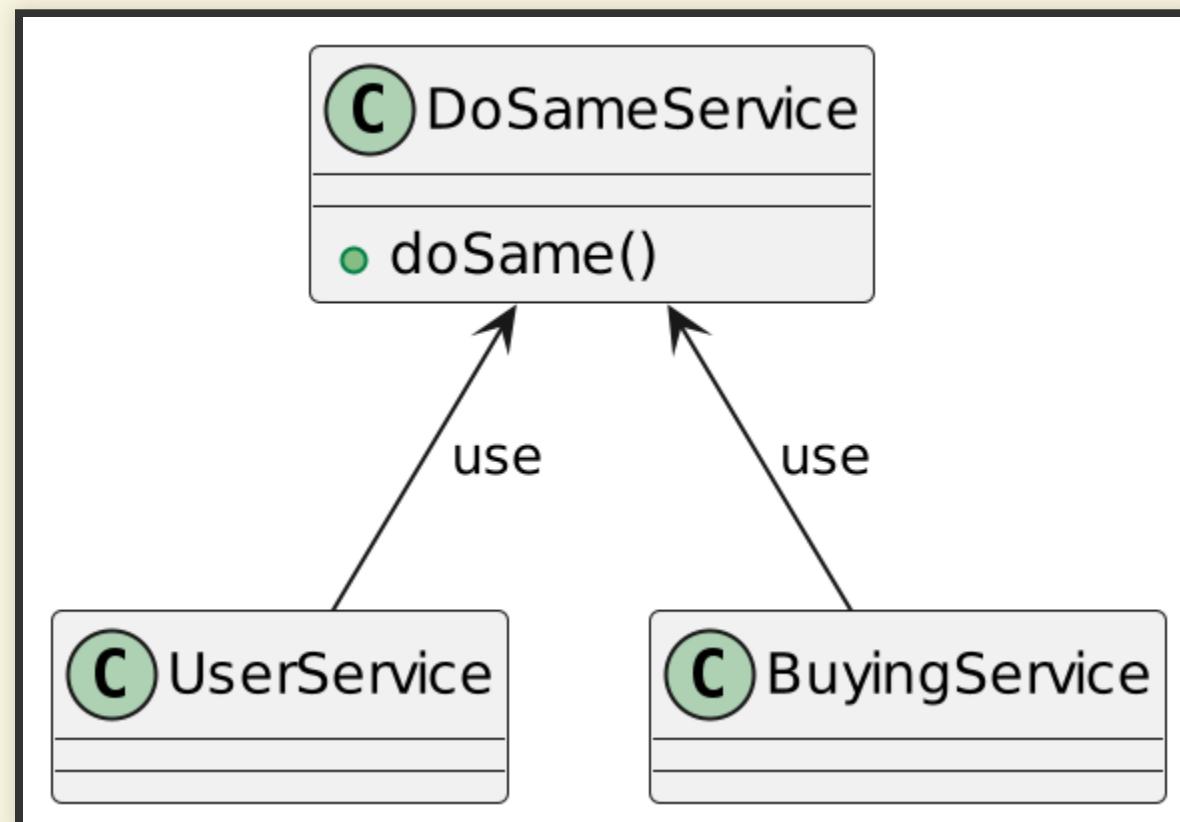
CODE SMELL: DUPLICATED CODE

- 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

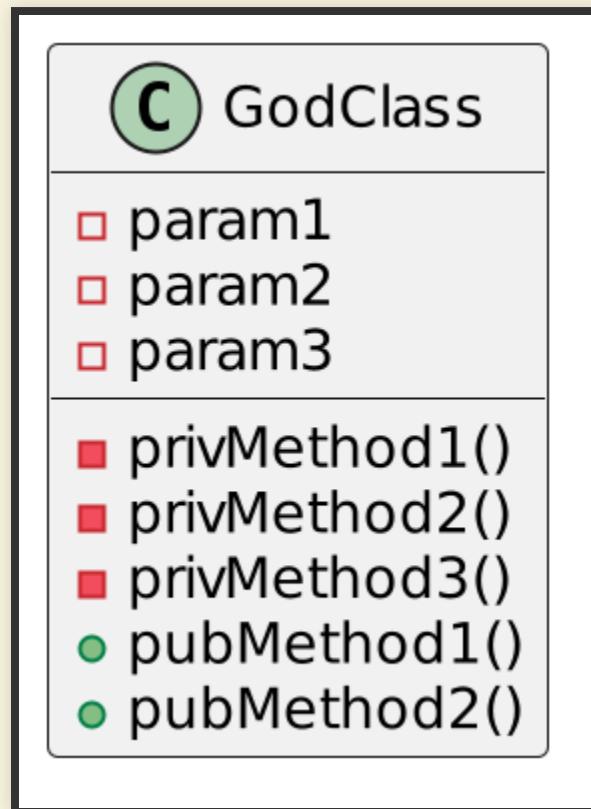
CODE SMELL: LONG PARAMETER LIST

- 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)`

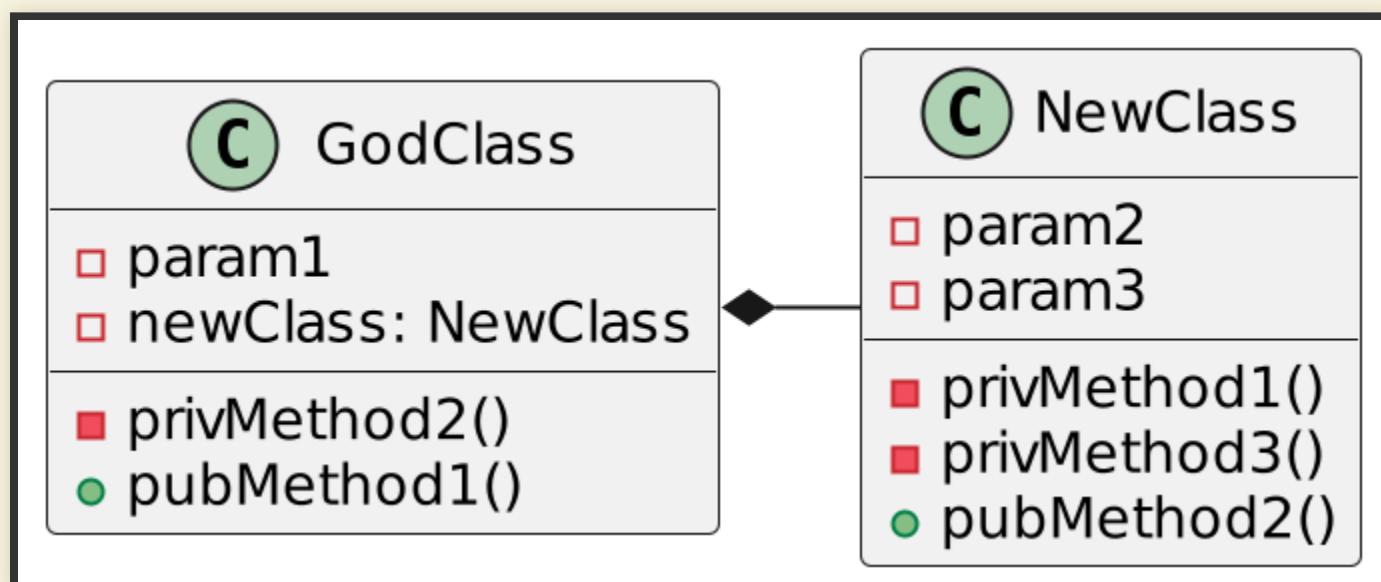
CODE SMELL: LARGE CLASS

- 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-Chirurgie
- 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)

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
- Lösung: eine elegante Lösung ist Polymorphie

nicht zu verwechseln mit Switch-Expression

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`)

CODE SMELL: MESSAGE CHAINS

- Aufrufketten (z.B.
`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

REFACTORINGS

kleine Auswahl aus über 60 Refactorings

REFACTORING: EXTRACT METHOD

Problem Zusammenhängendes Stück Code kann ausgelagert werden

Lösung Auslagern in eigene Methode mit einem aussagekräftigen
Namen

Effekte

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

Hilft gegen: Duplicated Code, Long Method, Code Comments

EXTRACT METHOD: BEISPIEL

```
void printMovies(FilmStudio studio) {  
    printLogo(studio);  
    // print all titles with price  
    for (Movie movie : getMovies(studio)) {  
        printLine(movie.title() + " " + movie.price());  
    }  
    printFooter(studio);  
}
```

- Code-Kommentar ist ein 'Smell' → Hinweis auf Extraktionsmöglichkeit
- Abstraktionsebenen vermischt

```

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

private void printTitlesWithPrice(FilmStudio studio) {
    for (Movie movie : getMovies(studio)) {
        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
- Lokale Variablen machen Probleme
 - 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: 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 getMeasurmentSensorDescription() {  
    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

RENAME METHOD: BEISPIEL

Vorher

```
Euro basePrice = new Euro(quantity * itemPrice.getValue());  
if (basePrice.getValue() > 1000.0d) {  
    return basePrice.multiplyWith(0.95);  
}  
return basePrice.multiplyWith(0.98);
```

Nachher

```
if (basePrice().getValue() > 1000.0d) {  
    return basePrice().multiplyWith(0.95);  
}  
return basePrice().multiplyWith(0.98);  
  
Euro basePrice() {  
    return new Euro(quantity * itemPrice.getValue());  
}
```

Methodenaufruf statt Zwischenspeicher, immer korrekter Wert

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();  
    }  
}
```

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);  
    }  
}
```

- 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
 - verschiedene Verarbeitung des Fehler-Rückgabewerts möglich
- ## REPLACE ERROR CODE WITH EXCEPTION: BEISPIEL

- der Fehler-Rückgabewert ~~kommt~~ aus dem Wertebereich des

```
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 Unterkasse verwendet nur einen Teil der Oberkasse

Lösung die Oberkasse 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);
    }
}
```

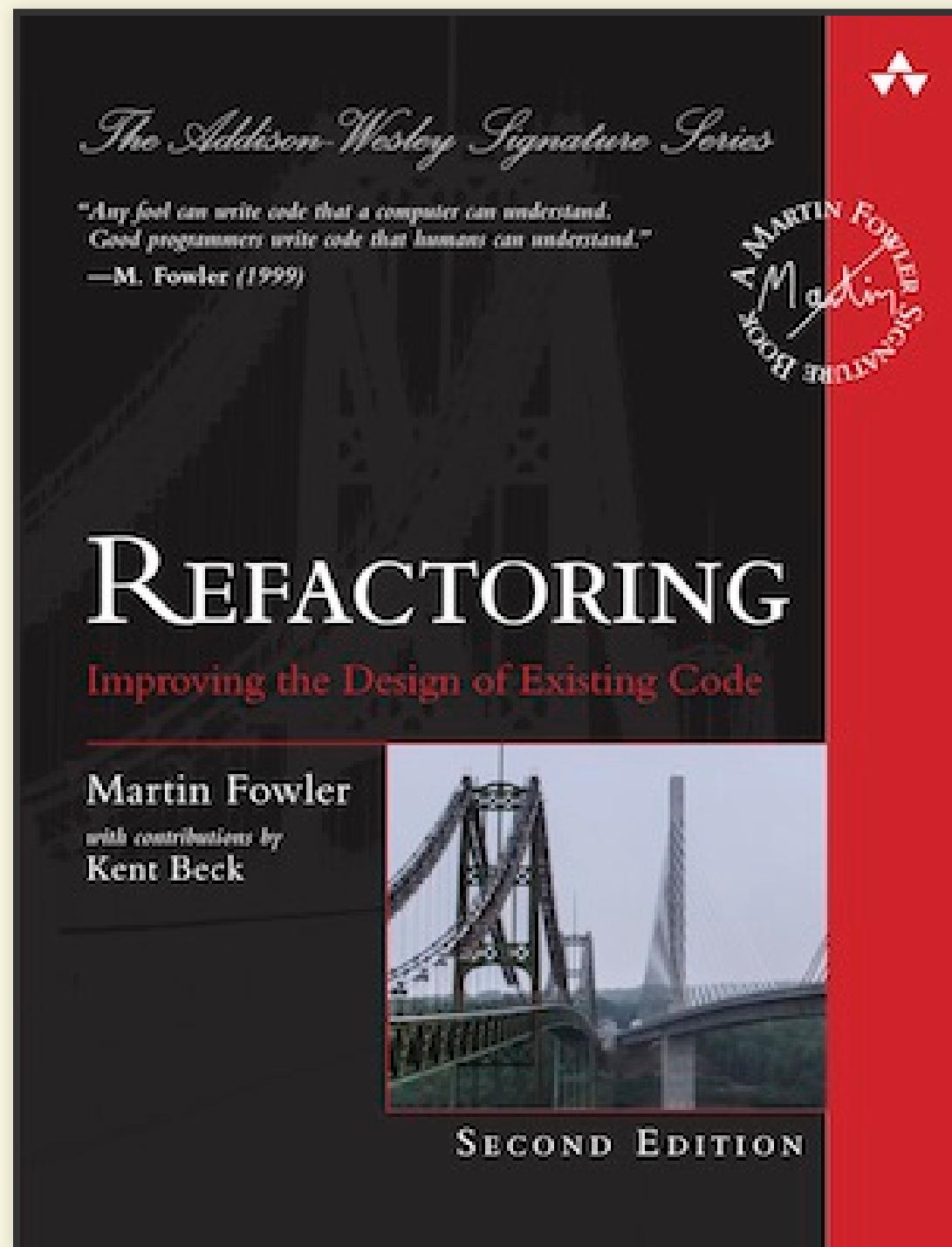
- List/ArrayList bietet zu viele Methoden, die in diesem Kontext kontraproduktiv sind
 - z.B. remove(int)
- für Stack reicht: push, pop, getSize und isEmpty

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();  
    }  
}
```

- Schnittstelle ist jetzt genau passend
- keine "verbotenen" Methoden mehr
- Umbenennung möglich

WEITERFÜHRENDE BÜCHER



EOF

