Einführung in Software Testen II

Christina Zoffi

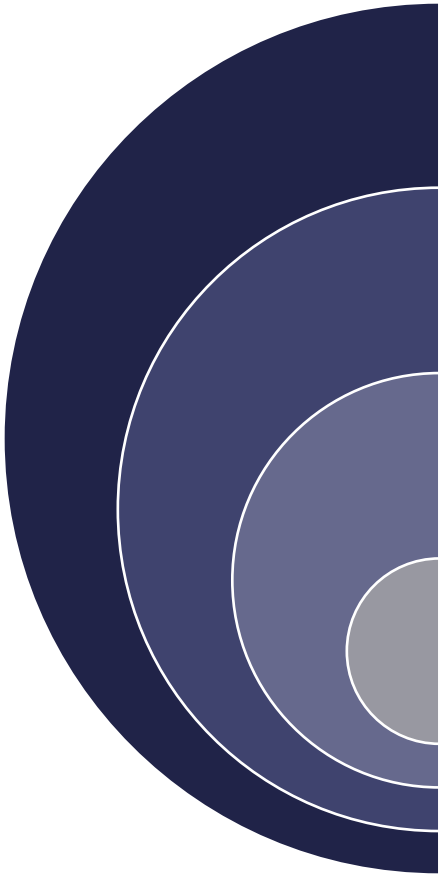
180.764 Software Qualitätssicherung

Research Group for Industrial Software (INSO)
https://www.inso.tuwien.ac.at

Inhalt

- Test Doubles
- Continuous Integration
- Refactoring

Wiederholung – Teststufen



Akzeptanztests

Systemtests

Integrationstests

Komponententests

Wiederholung – Komponententests

- Ziel ist das Testen der Funktionalität einer einzelnen Komponente
- Testet eine Komponente isoliert vom Rest des Systems

- Problem: Komponenten existieren meist nicht isoliert
  - Interagieren mit anderen Komponenten
  - Rufen externe Schnittstellen auf
  - Verwenden nicht deterministische Werte (z.B. Systemzeit)
  - Interagieren mit einer Datenbank
  - …

→ Wie können diese Abhängigkeiten ausgeblendet werden?

Test Doubles

Test Doubles

- Generischer Begriff für den Austausch einer Komponente des Systems durch eine alternative, meist simplifizierte Implementierung für Testzwecke

Isolation

- Isoliertes Testen einzelner Komponenten
- Bessere Steuerung der zu testenden Komponente
- Reduktion von Abhängigkeiten zu anderen System(teil)en

Effizienz

- Reduktion von Komplexität
- Reduktion der Ausführungszeit
- Reduktion von Kosten/Service-Gebühren

Flexibilität

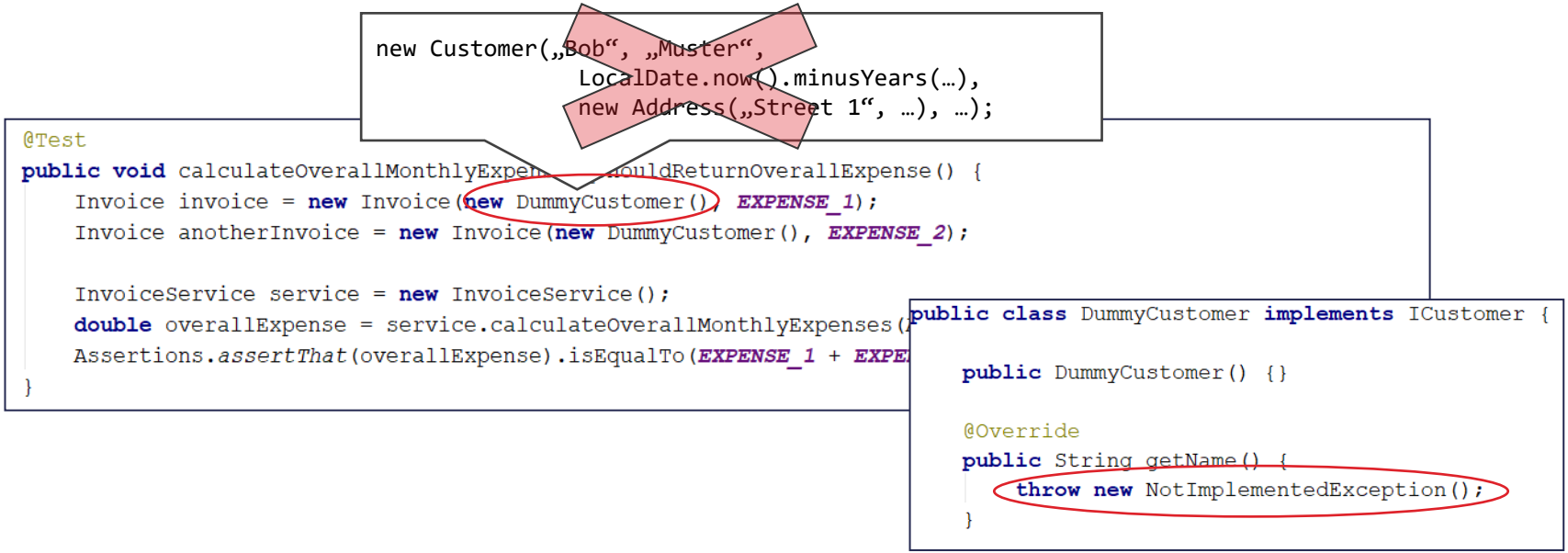
- Leichteres Testen von Randfällen und Fehlerfällen
- Kontrolle nicht deterministischer Werte (z.B. Datum/Zeit)

Test Doubles – Typen

- Es gibt unterschiedliche Ansätze, eine solche simplifizierte Implementierung zu erstellen:
  - Dummy Objekt
  - Fake Objekt
  - Spy
  - Stub
  - Mock

Test Doubles – Typen

- **Dummy Objekt**
  - „Leerer“ Platzhalter ohne Funktionalität
  - Wird in die Komponente gereicht, jedoch nicht zur Verwendung intentioniert



Test Doubles – Typen

- **Fake Objekt**
  - Ausführbare Implementierung
  - Dient jedoch **nicht** zur Steuerung/Überprüfung des Testobjekts

  - Einsatzzwecke:
    - Reale Implementierung zu langsam
    - Reale Implementierung nicht (bzw. nicht in Testumgebung) verfügbar

  - Beispiele:
    - Simulation einer Datenquelle
    - In-Memory Datenbank

Test Doubles – Typen

- **Spy**
  - Kein Ersetzen einer Komponente durch eine einfachere Variante
  - Erweiterung einer Komponente um „Überwachungsfunktionen“
    - Interaktionen zw. Testobjekt und abhängiger Komponente werden überwacht
    - Aufrufe werden jedoch an die reale Komponente weiter delegiert
    - Vorteil: Verhalten der realen Komponente wird nicht beeinflusst (→ realitätsnäheres Testsetup)
    - Nachteil: Komplexität/Abhängigkeit bleibt bestehen
  - Ermöglicht z.B.
    - Prüfung aufgerufener Methoden
    - Prüfung der übergebenen Aufrufparameter
    - Reihenfolge des Aufrufs
    - Anzahl der Aufrufe

Test Doubles – Typen

- **Stub**
  - Ausführbare Implementierung
  - Liefert vordefinierte Werte/Exceptions an den Aufrufer
    - Responder-Stub (Gutfall, Lieferung valider Werte)
    - Saboteur-Stub (Fehlerfälle, Werfen von Fehlern/Exceptions)
  - Ermöglicht indirekt die Steuerung der zu testenden Komponente
    - Steuerung, welcher Pfad durchlaufen wird
    - Erlaubt dadurch das Testen von Pfaden in der Komponente, die von außen nicht beeinflusst werden könnten

Test Doubles – Typen

- Stub Beispiel

```java
public class CountryService {

    private ICreditRatingService creditRatingService;

    public void setCreditRatingService(ICreditRatingService creditRatingService) {
        this.creditRatingService = creditRatingService;
    }

    public boolean isHighInvestmentRisk(int investmentSum, Country country) {

        Rating rating = creditRatingService.retrieveRatingForCountry(country);

        if(rating == Rating.AAA) {
            return false;
        }
        else if(investmentSum <= 150000 && rating.isRatingHigherThan(Rating.BBB)) {
            return false;
        }

        return true;
    }
}
```

- Externe Abhängigkeit
- Keine Kontrolle über Output
- Eventuell langsam
- Nachfolgender Kontrollfluss schwer steuerbar

```java
/**
 * Schnittstelle zu einem Webservice,
 * welches das Bonitätsrating für ein Land abfragt.
 */
public interface ICreditRatingService {

    Rating retrieveRatingForCountry(Country country);
}
```

Test Doubles – Typen

- Stub Beispiel

```java
public class CreditRatingStub implements ICreditRatingService {

    private Rating rating;

    public void initCreditRatingStub(Rating rating) {
        this.rating = rating;
    }

    @Override
    public Rating retrieveRatingForCountry(Country country) {
        return rating;
    }
}
```

```java
@BeforeEach
public void init() {
    creditRatingStub = new CreditRatingStub();
    countryService = new CountryService();
    countryService.setCreditRatingService(creditRatingStub);
}

@Test
public void isHighInvestmentRisk_shouldReturnFalse() {
    //Configure Stub
   creditRatingStub.initCreditRatingStub(Rating.AAA);

    boolean isHighRisk = countryService.isHighInvestmentRisk(HIGH_SUM, Country.AT);
    Assertions.assertThat(isHighRisk).isFalse();
```

Test Doubles – Typen

- Stub Beispiel

```
public class CountryService {

    private ICreditRatingService creditRatingService;

    public void setCreditRatingService(ICreditRatingService creditRatingService) {
        this.creditRatingService = creditRatingService;
    }

    public boolean isHighInvestmentRisk(int investmentSum, Country country) {

        Rating rating = creditRatingService.retrieveRatingForCountry(country)

        if(rating == Rating.AAA) {
            return false;
        }
        else if(investmentSum <= 150000 && rating.isRatingHigherThan(Rating.BBB)) {
            return false;
        }

        return true;
    }
}
```

[creditRatingStub]

[Rating.AAA]

Test Doubles – Typen

- **Mock**
  - Ausführbare Implementierung
  - Liefert vordefinierte Werte/Exceptions an den Aufrufer
  - Im Gegensatz zum Stub werden
    - die Interaktionen des Testobjekts mit dem Mock überwacht
    - die an den Mock übergebenen Parameter überprüft
    - Erwartungen an das Verhalten des Testobjekts geprüft
  - Ein Mock ist somit selbst Teil des Tests
  - Ein Mock erkennt, wenn unerwartete Werte eingehen
    - Direkte Verwendung von Assertions
    - Auslösen von Exceptions

Test Doubles – Typen

- Mock Beispiel – Variante 1

```
public class CreditRatingMock implements ICreditRatingService {

    private Rating rating;

    public void initCreditRatingMock(Rating rating) {
        this.rating = rating;
    }

    @Override
    public Rating retrieveRatingForCountry(Country country) {

        if(country == null) {
            throw new IllegalArgumentException("Country must not be null");
        }
        else if(country.getIso3CountryCode().isEmpty() || country.getIso3CountryCode().length() != 3) {
            throw new IllegalArgumentException("ISO 3 Country Code must consist of 3 characters");
        }

        return rating;
    }
```

Test Doubles – Typen

- Mock Beispiel – Variante 2

```
public class CreditRatingMock implements ICreditRatingService {

    private Rating rating;

    private boolean mockCalled = false;

    public void initCreditRatingMock(Rating rating) {
        this.rating = rating;
        this.mockCalled = false;
    }

    @Override
    public Rating retrieveRatingForCountry(Country country) {
        Assertions.assertThat(country).isNotNull();
        Assertions.assertThat(country.getIso3CountryCode()).isNotBlank();
        Assertions.assertThat(country.getIso3CountryCode().length()).isEqualTo(3);

        mockCalled = true;

        return rating;
    }

    public boolean wasMockCalled() {
        return mockCalled;
    }
```

Mocking Frameworks

- Mocking Frameworks vereinfachen die Verwendung von Mock Objekten für Tests

- Unterstützen:
  - Erstellung von Mock Objekten
  - Konfiguration des gewünschten Verhaltens
  - Überprüfung des gewünschten Resultats

- Beispiele für Java Mocking Frameworks
  - Mockito
  - JMockit
  - EasyMock
  - PowerMock

Mocking Frameworks – Mockito

- Erzeugen des Mocks
  - `mock(Class<T> clazz)`
  - `@Mock`

- 3 Varianten

```
private ICreditRatingService creditRatingMock = Mockito.mock(CreditRatingServiceImpl.class);
```

```
@Mock
private ICreditRatingService creditRatingMock;

@BeforeEach
public void init() {
    MockitoAnnotations.openMocks( testClass: this);
}
```

Bei Verwendung von Annotations: Muss einmalig für die Initialisierung aller Mocks aufgerufen werden.

```
@ExtendWith(MockitoExtension.class)
class MockitoSampleTest {

    @Mock
    private ICreditRatingService creditRatingMock;
```

Alternativ kann auch eine JUnit5 Extension verwendet werden, diese übernimmt die Initialisierung

Mocking Frameworks – Mockito

- Spezifikation des Verhaltens
  - `when(T methodCall)`
    - Welche Methode soll gemockt werden?
  - `thenReturn(T value)`
    - Welchen Response soll der Mock antworten?
  - `thenThrow(Throwable t)`
    - Welche Exception soll der Mock werfen?

```
Mockito.when(creditRatingMock.retrieveRatingForCountry(Country.AT))
        .thenReturn(Rating.AAA);
```

```
Mockito.when(creditRatingMock.retrieveRatingForCountry(null))
        .thenThrow(IllegalArgumentException.class);
```

Mocking Frameworks – Mockito

- Verwendung von Platzhaltern in „when“ (Argument Matcher)
  - Zur Überprüfung der eingehenden Argumente
  - Achtung vor Verwendung von zu allgemeinen Matchern (Aufweichung der Prüfung!)
  - `any(…)`
  - `anyInt(…)`
  - `argThat(somestring -> somestring.length() > 10)`
  - …

Country.AT
Country.FR
Country.DE

```
Mockito.when(creditRatingMock.retrieveRatingForCountry(Mockito.any(Country.class)))
        .thenReturn(Rating.BBB);
```

  - Built-In Matcher siehe auch hier: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/ArgumentMatchers.html

Mocking Frameworks – Mockito

- Überprüfung der Aufrufe am Mock
  - Zur Überprüfung, ob bzw. wie oft Funktionen aufgerufen wurden
  - `verify(…)`
  - `never() / times() / atLeastOnce() …`

```
Mockito.verify(creditRatingMock).retrieveRatingForCountry(Country.AT);

Mockito.verify(creditRatingMock, Mockito.times(1)).retrieveRatingForCountry(Country.AT);

Mockito.verify(creditRatingMock, Mockito.atLeastOnce()).retrieveRatingForCountry(Country.AT);
```

  - Dokumentation siehe auch:
    https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#4

Mocking Frameworks – Mockito

- Beispiel

```
private CountryService countryService = new CountryService();
private ICreditRatingService creditRatingMock;

@BeforeEach
public void init() {
    //Create Mock Instance
    creditRatingMock = Mockito.mock(CreditRatingServiceImpl.class);
    countryService.setCreditRatingService(creditRatingMock);
}

@Test
public void isHighInvestmentRisk_shouldReturnFalse() {
    //Specify Behaviour on Mock
    Mockito.when(creditRatingMock.retrieveRatingForCountry(Country.AT))
            .thenReturn(Rating.AAA);

    boolean isHighRisk = countryService.isHighInvestmentRisk(HIGH_SUM, Country.AT);
    Assertions.assertThat(isHighRisk).isFalse();

    Mockito.verify(creditRatingMock).retrieveRatingForCountry(Country.AT);


}
```

Continuous Integration (CI)

Definition

- „*An automated software development procedure that merges, integrates and tests all changes as soon as they are committed.*“ (ISTQB Glossar)

- Ursprünglich im Kontext von Extreme Programming (XP) etabliert, mittlerweile Standard in agilen Projekten
- Entwicklungsansatz, bei dem Code Änderungen laufend integriert, getestet und gebaut werden
- Jede Integration wird durch einen automatisierten Buildprozess begleitet

Ausgestaltung

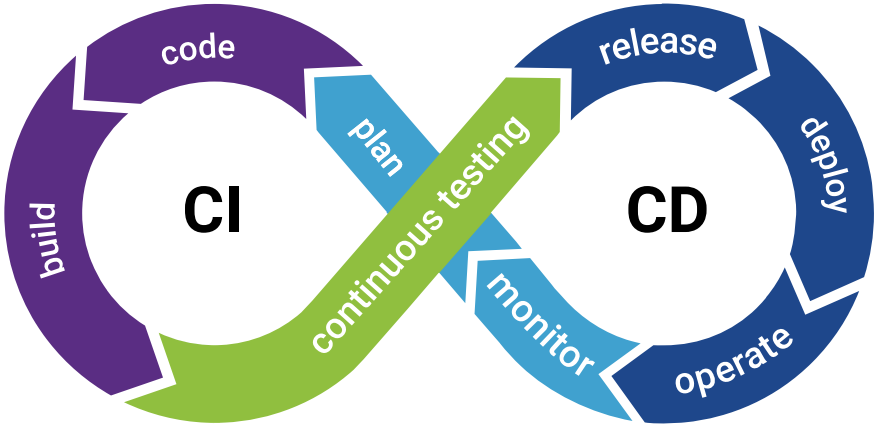
- Grundprinzipien
  - Häufige Commits („Commit Daily – Commit Often“)
  - Häufige Builds („Build Early – Build Often“)
  - Stabile Builds („Keep it Green“)
- Ausgestaltung abhängig von
  - Projektkontext
  - Techn. Setup/Technologien
  - Verfügbares Versionsverwaltungs- und CI-Tool
  - Gewünschter Umfang, z.B.
    - Kompilieren / Bauen des Projekts
    - Ausführung automatisierter Tests (z.B. Unit-/Integration-/Systemtests)
    - Statische Analysen (z.B. Sonar, Checkstyle)
    - Security Analysen (z.B. Schwachstellenscan)

Vorteile

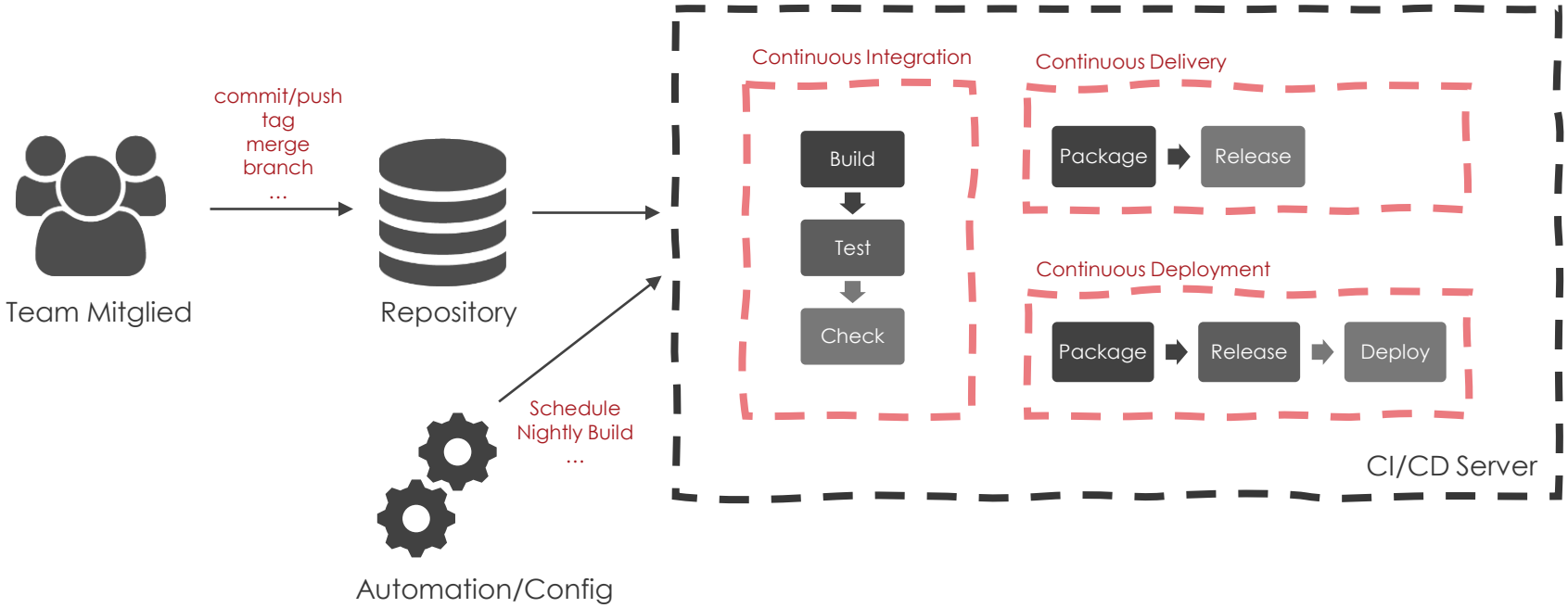
- Schnelles und häufiges Feedback
- Reduktion manueller Tätigkeiten
- Frühzeitige Identifikation von Fehlern
- Bessere Ausgangssituation für Refactorings
- Vermeidung einer späten „Integrationshölle“
- Vermeidung einer „works on my maschine“ Problematik
- Mehr Transparenz über Status und Qualitätsniveau
- Einblick in Projekt-Historie

Abgrenzung CI/CDE/CD

- CI fokussiert auf den automatisierten Build der Software bis an den Punkt, wo ein integrierter, lauffähiger SW-Stand vorliegt
- Darauf aufbauende Ansätze
  - Continuous Delivery (CDE)
    - Automatisierung des Packaging-/Release-Prozesses
  - Continuous Deployment (CD)
    - Automatisierung des Deployment-Prozesses
    - Automatisierte Bereitstellung neuer SW-Versionen auf Produktivumgebungen



Workflow CI/CDE/CD



Relevanz für DevOps

- Traditionell strikte Trennung zwischen Entwicklung und Betrieb von Software
  - Entwicklung zuständig für Bereitstellung des Produkts
  - Betrieb zuständig für Deployment, Installation, Infrastruktur und Support
- Problematischer Gap
  - Fehlende Berücksichtigung betrieblicher Aspekte während Entwicklung
  - Fehlendes Testing betrieblicher Aspekte
  - Späte Identifikation von Blockern
  - Fehlende Transparenz + fehlendes Know-How

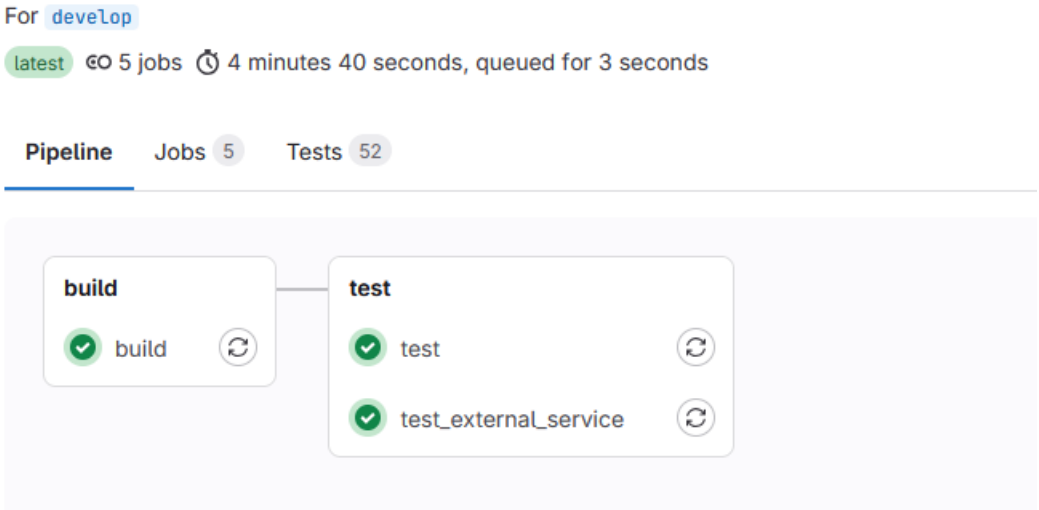
- DevOps liefert Praktiken und Tools, die die teamübergreifende Kooperation fördern und die Durchlaufzeit bis zur Bereitstellung der Software reduzieren soll
  - → CI/CD ist ein wesentlicher Baustein

Beispiele CI-Server

- Build-Automatisierung kann auf zwei Arten erfolgen
  - Separater CI-Server
  - Integration in Versionsverwaltungs-Plattform
- Beispiele
  - Jenkins
  - Travis CI
  - Circle CI
  - GitLab CI/CD
  - GitHub Actions

CI-Pipeline Beispiel

- Einfache Pipeline mit GitLab CI (analog zur SQS VU UE)



Refactoring

Refactoring

- Systematischer Ansatz zur Steigerung der Code Qualität
- Verbesserung der internen Struktur des Codes hinsichtlich
  - Lesbarkeit
  - Verständlichkeit
  - Komplexität
  - Architektur/Design
- Keine Änderung des sichtbaren Verhaltens
  - Keine neue Funktionalität
  - Keine Behebung von Fehlern
- Keine Änderungen, die nicht das Ziel haben, die Struktur des Codes zu verbessern
  - z.B. Performanceverbesserungen

Technische Schuld

- Metapher-Begriff, um nicht-techn. Stakeholdern die Notwendigkeit von Refactoring zu erklären
  - Ursprünglich geprägt von Ward Cunningham
  - Schlechte Code Qualität = sich in Schuld begeben
  - Schulden müssen (zeitnah) zurückgezahlt werden
- Problematik – 2 Aspekte
  - Code „verwahrlost“ über die Zeit
  - Entstehung schlechter Qualität wird als bewusster Kompromiss in Kauf genommen, um kurzfristig schneller liefern zu können (→ 2. VO-Einheit, Teufelsquadrat)
- Anhäufung von Qualitätsmängeln führt langfristig zu
  - Verlust an Produktivität
  - Verlust an Wartbarkeit
  - Höheren Fehlerraten
  - Explosion von Fehlerkosten

Refactoring – Motivation

- Verbesserung des Designs
  - Erhaltung von klarer Struktur, z.B. Kapselung
  - Aufarbeitung technischer Schuld
- Erhöhung der Verständlichkeit
  - Erleichtert die Einarbeitung neuer Entwickler
  - Reduziert zukünftigen Aufwand bei Anpassung/Wartung
- Auffinden von Fehlern
  - Mehr Verständnis über den Code ermöglicht Fehler zu identifizieren
  - Reflexion über getroffene Annahmen

Refactoring – Vorgehensweise

1. Identifikation

   Auffinden von Stellen im Code, die verbessert werden können, manuell oder durch den Einsatz von Tools

2. Testabdeckung

   Sicherstellung, dass ausreichend Tests für den von Refactoring betroffenen Code vorhanden sind. Bei Bedarf Ergänzung von zusätzlichen Tests, *bevor* mit dem Refactoring begonnen wird

3. Durchführung

   Schrittweise Verbesserung des Codes, regelmäßiges Ausführen der Tests zur Überprüfung, dass die Funktionalität nicht verändert wird

   Moderne IDEs unterstützen bei Refactorings!

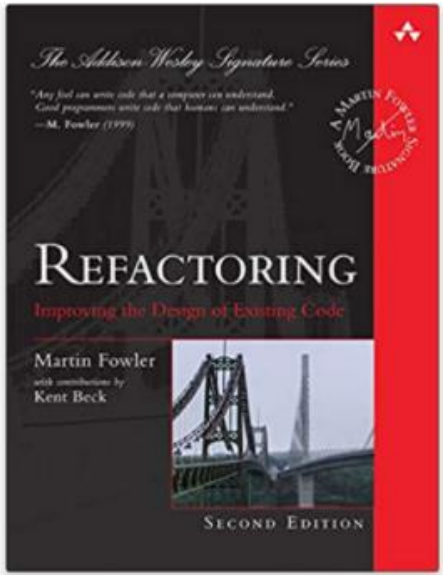
Refactoring – Bad Smells

- Indikatoren für die Notwendigkeit von Refactoring
- Unterteilung in Gruppen, z.B.

Bloaters	Code, der über die Zeit hinweg sukzessive gewachsen ist z.B. Large Class, Long Method, Long Parameter List
Object-Orientation Abusers	Falsche oder unvollständige Nutzung objektorientierter Paradigmen z.B. Temporary Field
Couplers	Zu starke Abhängigkeiten zwischen Klassen z.B. Feature Envy
Change Preventers	Änderungen an einer Stelle haben Auswirkungen auf viele andere Stellen z.B. Shotgun-Surgery
Despensables	Überflüssiger Code, negative Auswirkungen auf Lesbarkeit und Wartbarkeit z.B. Duplicate Code, Dead Code

Refactoring – Patterns

- Beschreiben allgemeine Lösungsvorschläge für wiederkehrende Probleme
- Unterteilung in Gruppen, z.B:
  - Composing Methods
  - Moving Features Between Objects
  - Organizing Data
  - Simplifying Conditional Expressions
  - Simplifying Method Calls
  - Dealing With Generalization



Als Online Ressource der TU:
https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_safari_books_v2_9783958459434

Refactoring – Patterns

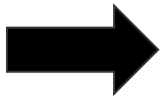
Gruppe	Fokus	Beispiele
Composing Methods	Struktur/Schnitt von Methoden	• Extract Method • Extract Variable • Replace Temp with Query
Moving Features between Objects	Verschieben von Funktionalität zwischen Klassen	• Move Method • Move Field • Extract Class
Organizing Data	Verwaltung/Kapselung von Daten	• Encapsulate Field • Replace Magic Number
Simplifying Conditional Expressions	Vereinfachen von logischen Ausdrücken	• Decompose Conditional
Simplifying Method Calls	Vereinfachen von Methodenaufrufen	• Rename Method • Add Parameter
Dealing with Generalization	Erzeugen von Vererbungshierarchien Verschieben von Funktionalität zwischen Vererbungsstrukturen	• Pull Up Field • Pull Up Method • Extract Superclass

Refactoring – Beispiele

- Encapsulate Field

Indikator	Problem	Lösung
Feld besitzt public Modifier	• Fehlende Kapselung • Direkte Manipulation von außen	• Feld wird private • Klasse stellt Accessors bereit

```
public class Person {
  public String name;
}
```



```
public class Person {
  private String name;

  public void setName(String name) {
     this.name = name;
  }

  public String getName() {
     return name;
  }
}
```

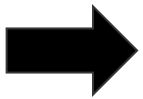
Refactoring – Beispiele

- Replace Temp With Query

Indikator	Problem	Lösung
Temporäre Variable speichert das Ergebnis einer Berechnung, wird jedoch öfter benötigt	Temporäre Variablen sind nur in eingeschränktem Kontext verfügbar → Code Duplication	Verschieben des Ausdrucks in eine eigene Methode

```
public double calculatePrice(Order order) {
   double taxFee = order.getPrice() * 0.2;
   if(order.getPrice() > 10.000) {...}
   ...
}

public void printBill(Order order) {
  double taxFee = order.getPrice() * 0.2;
  print(order.price);
  print(order.taxFee);
  print(order.price + order.taxFee);
}
```



```
public double calculatePrice(Order order) {
   double taxFee = calculateTaxFee(order.getPrice());
   if(order.getPrice() > 10.000) {...}
   ...
}

public void printBill(Order order) {
  double taxFee = calculateTaxFee(order.getPrice());
  print(order.price);
  print(order.taxFee);
  print(order.price + order.taxFee);
}


private double calculateTaxFee(double price) {
   return price * 0.2;
}
```

Refactoring – Beispiele

- Extract Method

Indikator	Problem	Lösung
• Lange Methode • Kommentare erforderlich um Methode zu verstehen	• Fehlende Verständlichkeit • Komplexe Logik • Keine Wiederverwendbarkeit	• Auslagern in eigene Methode(n) • Code an alter Stelle wird durch Methodenaufruf ersetzt

```
public void processData (String
pathIn) {

  //read data
  File inputFile = ...
  List<Person> dataList = ...

  //sort data
  for(Person p: dataList ) {...}
}
```

```
public void processData(String pathIn) {
  List<Person> dataList = readData(pathIn);
  List<Person> sortedList = sortData(dataList);
}

private List<Person> readData(String pathIn)
{...}

private List<Person> sortData(List data) {...}
```

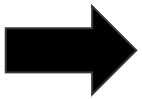
Refactoring – Beispiele

- Move Method

Indikator	Problem	Lösung
• Eine Methode wird von einer anderen Klasse häufiger verwendet als von der Klasse selbst • Eine Methode verwendet mehrere Funktionen einer anderen Klasse	• Hohe Kopplung zwischen Klassen (Abhängigkeit!) • Klasse hat zu viele Verantwortlichkeiten	• Verschieben der Methode

```
public class Product {

  public double calculatePrice(Order o) {
     if(o.isInternationalOrder()) {...}
     else if(o.type == OrderType.A) {...}
     ...
  }
}
```



```
public class Order {
    private boolean internationalOrder;
    private OrderType type;

    public double calculatePrice() {
        if(internationalOrder) {...}
        else if(type == OrderType.A) {...}
        ...
    }
}
```

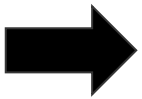
Refactoring – Beispiele

- Extract Class

Indikator	Problem	Lösung
• Eine Klasse hat mehr als eine Verantwortlichkeit • Umfangreiche Klasse mit vielen Daten • Subset an Daten ändert sich immer gemeinsam	• Verständlichkeit des Codes sinkt • Wiederverwendbarkeit eingeschränkt • Vererbung schwer möglich	• Herauslösen von Funktionalität in separate Klasse • Herstellen einer Relation zwischen den Klassen, sofern erforderlich

```
public class Person {
  private String name;
  ...
  private String street;
  private String zipCode;
  private String city;
  private Country country;

  public String printAddress() { ...}
}
```



```
public class Person {
  private String name;
  ...
  private Address address;
}

public class Address {
  private String street;
  private String zipCode;
  private String city;
  private Country country;

   public String printAddress() {...}
}
```

Fragen?

Referenzen

- Mockito Webseite [http://site.mockito.org/], 06.04.2022
- Mockito Documentation [https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html], 06.04.2022
- SpotBugs Webseite [https://spotbugs.github.io/], 06.04.2022
- SonarLint Webseite [https://www.sonarlint.org/], 06.04.2022
- Thomas Grechenig, Mario Bernhart, Roland Breiteneder, Karin Kappel: „Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten“, Pearson Studium, 2009
- Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2020
- Gerard Meszaros; xUnit Test Patterns: Refactoring Test Code; Addison-Wesley Professional, 2007
- Source Making Webseite [https://sourcemaking.com/refactoring], 23.10.2024
- Brent Laster, Continuous Integration vs. Continuous Delivery vs. Continuous Deployment, 2nd Edition, O‘Reilly, 2020
- Blackduck Webseite [https://www.blackduck.com/glossary/what-is-cicd.html], 24.10.2024
- Sandro Pedrazzini, Exploiting the Advantages of Continuous Integration in Software Engineering Learning Projects, 9th Koli Calling International Conference on Computing Education Research, 2009
- Mojtaba Shahin et al., Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, IEEE Access, 2017