Einführung in Software Testen

180.764 Software-Qualitätssicherung

Markus Zoffi
peso@inso.tuwien.ac.at

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

Inhalt

Statische Qualitätssicherung

Qualitätsmetriken
Sourcecode Qualitätskriterien
Conventional Commits
Testcoverage
Statische Analyse / Fehlermuster

Dynamische Qualitätssicherung

Unit Tests
JUnit
Test-Driven-Development

Klassifikation Qualitätssicherung

Organisatorische Methoden

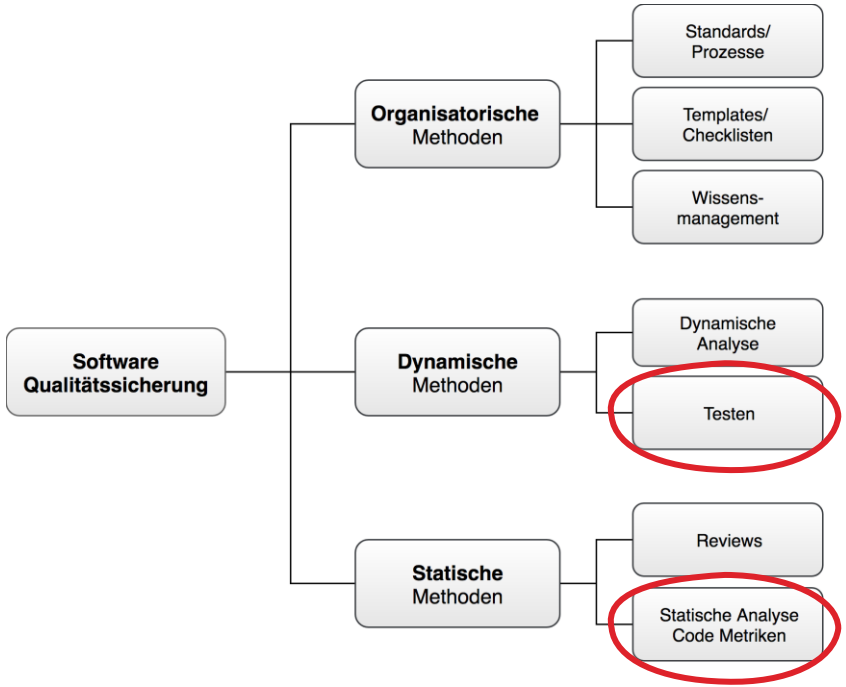
- Schaffen Infrastruktur und Rahmenbedingungen
- Definieren und steuern den Qualitätsprozess

Dynamische Methoden

- Software muss ausführbar sein
- Verhalten zur Laufzeit

Statische Methoden

- Software wird nicht ausgeführt
- Fokus auf die interne Struktur



Qualitätsmetriken

Qualitätsmetriken

- Metriken schaffen ein gemeinsames Verständnis von Qualitätsaspekten
- Metriken quantifizieren verschiedene Aspekte der Qualität
- Werden von Tools berechnet, die in die Entwicklungs-/Erstellungsumgebung integriert sind

- Aber:
  - Metriken müssen mit Sorgfalt / im richtigen Kontext interpretiert werden

Qualitätsmetriken

- Metriken dienen unterschiedlichen Zwecken:
  - Verstehen des Status Quo eines Projekts
  - Verfolgen von Trends im Laufe der Zeit
  - Vergleich von Status und Trends zwischen ähnlichen Projekten/Portfolios
  - Unterstützung von Managemententscheidungen
  - Ermöglichung korrigierender Maßnahmen

Qualitätsmetriken

- Beispiele:
  - Lines of Code (LOC)
  - Number of Statemens (NOS)
  - Zyklomatische Konplexität
  - Kognitive Komplexität
  - Testabdeckung (Coverage)
  - Anzahl gefundener Fehler
  - Technische Schuld
  - …

→ Weiterführende Vertiefung in dieses Thema in der LVA „183.652 Software Quality Management“

Statische Qualitätssicherung

Statische Analysen

- Statische Analysen werden durch Tools unterstützt

- Tools zur Überprüfung von Sourcecode Konventionen
  - Checkstyle
- Tools zur Überprüfung der Testabdeckung
  - JaCoCo
  - Clover
  - Cobertura
- Tools zum Finden von Fehlermustern
  - FindBugs/SpotBugs
  - SonarLint
  - SonarQube
- Frameworks für die Analyse der Systemarchitektur
  - ArchUnit

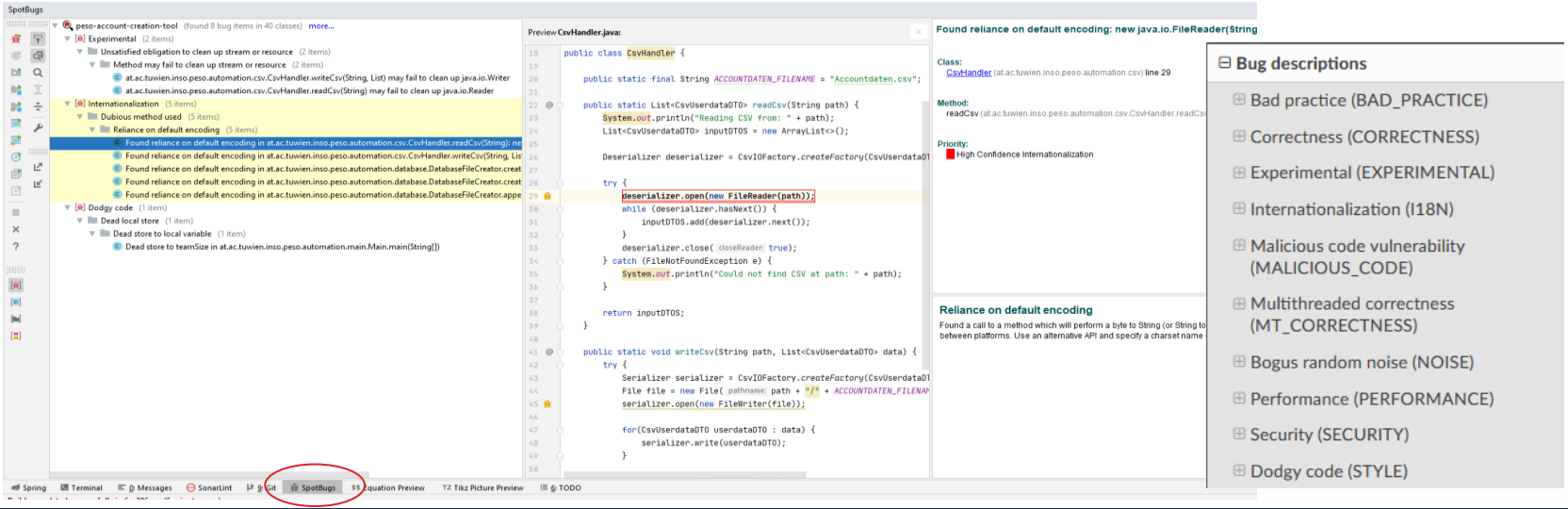
Fehlermuster

- Statische Code Analyse ermöglicht die Identifikation häufiger Fehlermuster im Code:
  - Variablen mit undefiniertem Wert
  - Komplexe Konstrukte
  - Toter Code
  - Potenzielle Endlosschleifen
  - Security Schwachstellen
  - Unused Code
  - …

- Vorteile:
  - Verbesserung der Code Qualität (z.B. Wartbarkeit)
  - Auffinden von Fehlern
  - Vermeidung/Reduktion zukünftiger Fehler

Fehlermuster – SpotBugs

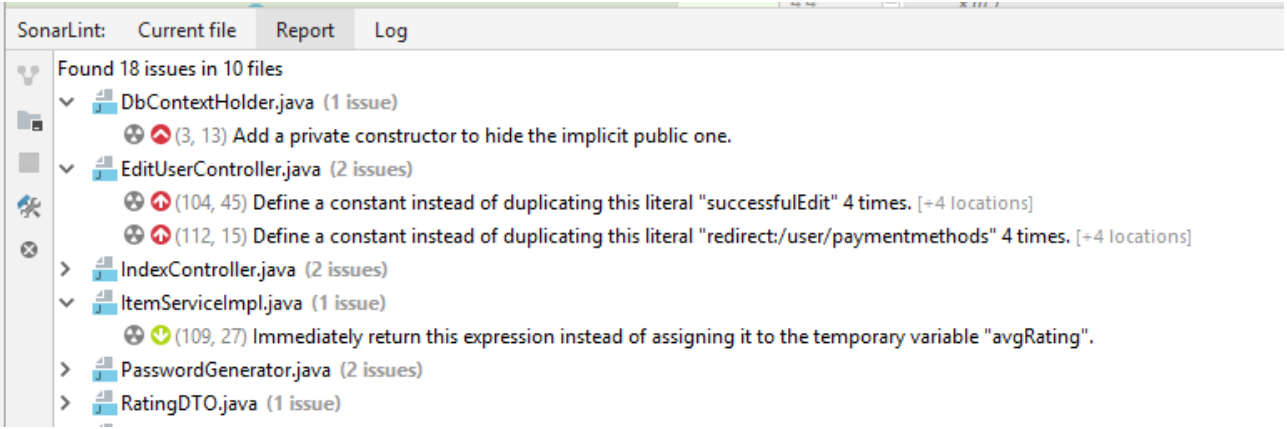
- Nachfolger Tool zu FindBugs (seit 2015 nicht mehr aktiv)
- Identifiziert potenzielle Problemstellen und häufige Fehlermuster in Java Code
  - Mehr als 400 Patterns
  - Verfügbar als Standalone Tool, als Maven/Gradle Plugin sowie als Plugin für IDEs, z.B. IntelliJ/eclipse



Fehlermuster – SonarLint

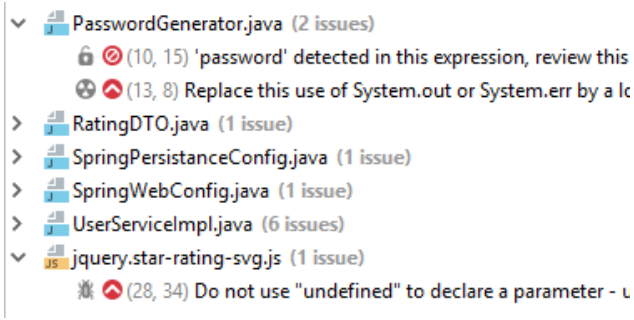
sonarlint

- SonarLint ist Teil des Sonar Ecosystems
  - Bekanntestes Produkt ist SonarQube
  - Open-Source Plattform für Continuous Code Quality
  - Aggregiert verschiedene bestehende Code Analyse Tools
- SonarLint ist die Light-Variante, die auch lokal in der Entwicklungsumgebung eingesetzt werden kann
  - Als Plugin für IntelliJ IDEA und eclipse



Fehlermuster – SonarLint

- SonarLint bietet eine große Anzahl an Regeln für eine Vielzahl an Programmier- und Scriptsprachen
  - https://rules.sonarsource.com/
- Regeln werden klassifiziert in:
  - Bug: Potenzielle Fehler im Code
  - Vulnerability: Sicherheitslücken im Code
  - Code Smell: Unschöner/unwartbarer Code



→ Weiterführende Vertiefung in dieses Thema in der LVA „183.652 Software Quality Management“

Fehlermuster – ArchUnit

- Architekturvorgaben in Unit Tests
- Einhaltung von Architekturvorgaben
  - Abhängigkeiten (inkl. Zyklen)
  - Namenskonventionen
  - Vererbung / Annotationen
  - ...

```java
@Test
public void testServicesInApiPackages_shouldBeInterfaces() {
    JavaClasses classes = new ClassFileImporter().importPackages("at.ac.tuwien.inso.bugstore.service.api");
    ArchRule rule = classes()
            .that().resideOutsideOfPackages().and().haveSimpleNameEndingWith( suffix: "Service")
            .should().beInterfaces();

    rule.check(classes);
}

@Test
public void testServicesInImplPackage_shouldBeSuffixed() {
    JavaClasses classes = new ClassFileImporter().importPackages("at.ac.tuwien.inso.bugstore.service.impl");
    ArchRule rule = classes()
            .should().haveSimpleNameEndingWith( suffix: "ServiceImpl");

    rule.check(classes);
}
```

Sourcecode Qualitätskriterien

Coding Conventions

- Code Guidelines
  - Einrückung (space/tab)
  - Max. Zeilenlänge
  - Import Reihenfolge
  - Methoden / Feld Reihenfolge in Klassen
    - Basierend auf Modifiern
    - Basierend auf Sichtbarkeit/Veränderbarkeit
  - System.out Vermeidung
  - Logging
  - Namenskonventionen
  - Code Formatierung

Coding Conventions

- Einheitlicher Stil innerhalb eines Projekts
- Definieren Konventionen/Guidelines für eine spezifische Programmiersprache

```
private void foo2()
{
    if(condition1)
    {
        doSomethingAboutIt();
    }
    else
    {
        doSomethingAboutIt();
    }
}
```

```
private void foo() {
    if(condition1) {
        doSomethingAboutIt();
    } else {
        doSomethingAboutIt();
    }
}
```

```
private void foo3() {
    if(condition1)
        doSomethingAboutIt();
    else
        doSomethingAboutIt();
}
```

Coding Conventions

- Variierender Stil erschwert Lesbarkeit und Verständlichkeit des Codes

```
private void foo4() {
    try {
        if ((condition1 && condition2)
            || (condition3 && condition4)
            || !(condition5 && condition6)) {
            doSomethingAboutIt();
        }

        if ((condition1 && condition2) ||
            (condition3 && condition4) ||
            !(condition5 && condition6)) {
            doSomethingAboutIt();
        }
        else
            doSomethingAboutIt();
    }catch (Exception e)
    {
        doSomethingAboutIt();
    }
```

Coding Conventions

- Eigene Regeln definieren
- Wie kontrolliert man die Einhaltung im Projekt / im Team?
  - Checkstyle (http://checkstyle.sourceforge.net/)
- Checkstyle
  - Entwickler Tool zur Spezifikation und Überprüfung von Coding Guidelines für Java Code
  - Forciert einheitlichen Code Stil im Projekt
  - Regeln werden über XML Config Datei spezifiziert
  - Vordefinierte Coding Guidelines:
    - Sun Code Conventions
    - Google Java Style
- Anpassung bestehender Regeln je nach Projekt

Coding Conventions – Checkstyle

- Standard Checks beinhalten:
  - JavaDoc (Existenz, Struktur)
  - Naming Conventions (Länge, Format, Stil, Zeichen, …)
  - Imports (*-Imports, Statische Imports, …)
  - Block Checks (Klammersetzung, …)
  - Modifier (Reihenfolge, Redundanz)
  - Size (Anzahl Zeilen, Länge Zeilen, Anzahl Parameter, …)

```
<module name="JavadocMethod">
    <property name="scope" value="public"/>
    <property name="allowMissingParamTags" value="true"/>
    <property name="allowMissingThrowsTags" value="true"/>
    <property name="allowMissingReturnTag" value="true"/>
    <property name="allowThr
</module>
```

```
<module name="LineLength">
    <property name="max" value="100"/>
    <property name="ignorePattern" value="^package.*|^import.* />
</module>
```

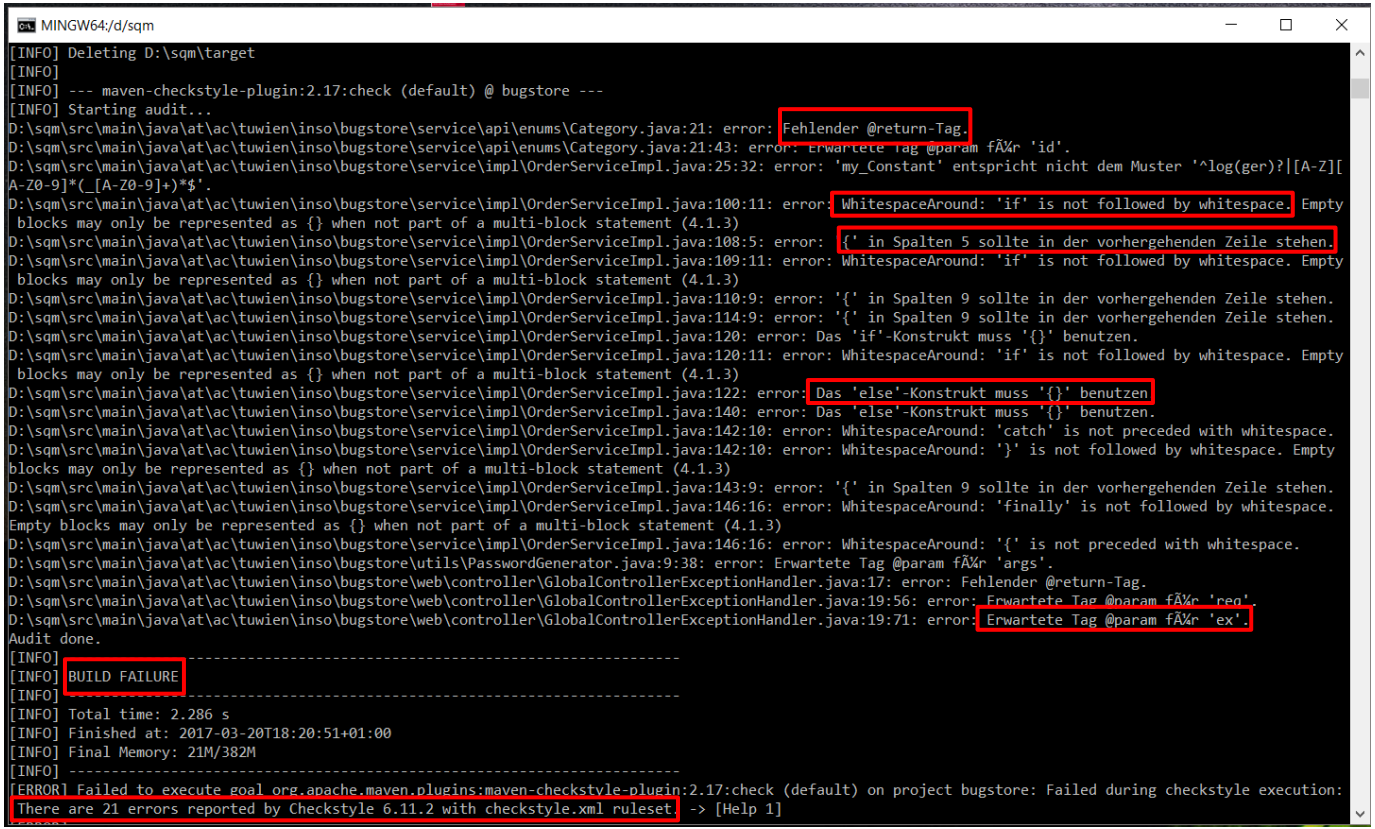
Coding Conventions – Checkstyle

- Überprüfung – Konfiguration mittels Maven

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>${maven-checkstyle-plugin.version}</version>
    <executions>
        <execution>
            <phase>process-sources</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <configLocation>checkstyle.xml</configLocation>
        <failsOnError>true</failsOnError>
        <consoleOutput>true</consoleOutput>
    </configuration>
</plugin>
```

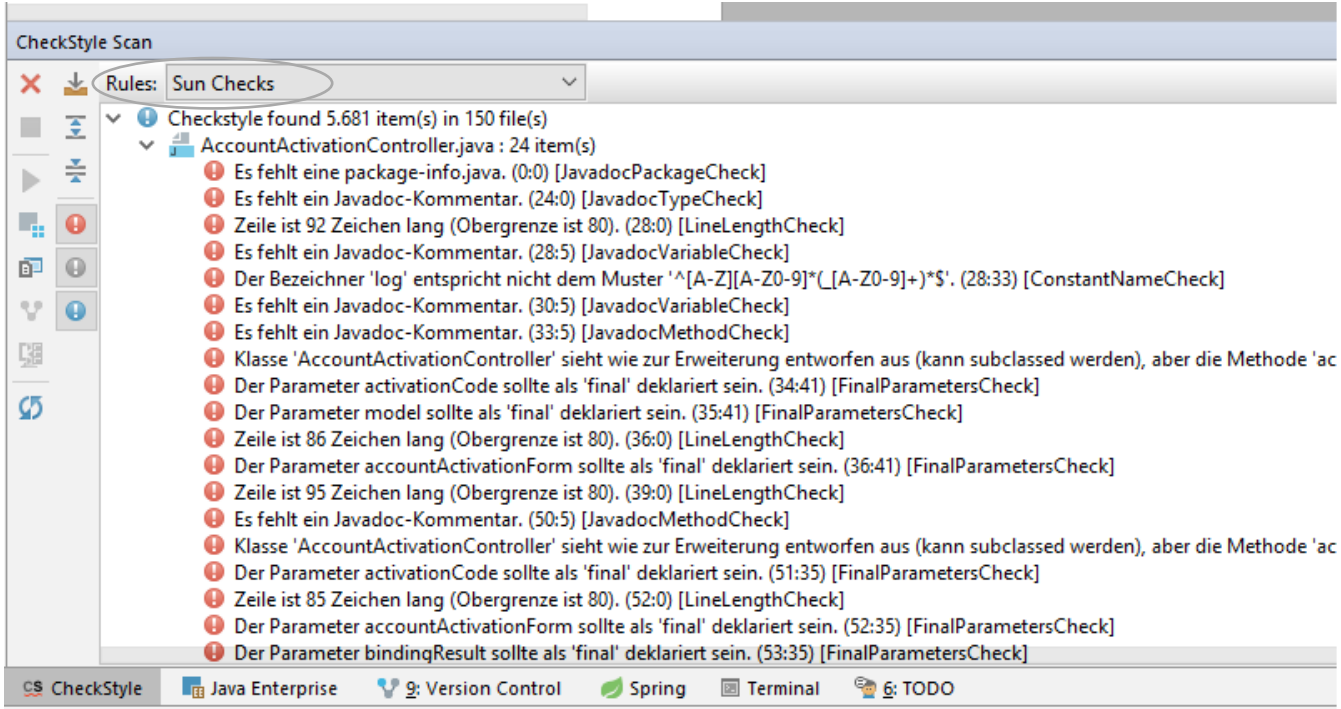
Coding Conventions – Checkstyle

- Integriert in den Build Prozess – nach „mvn clean install“



Coding Conventions – Checkstyle

- Überprüfung – Eingliederung mittels Plugin in IDE

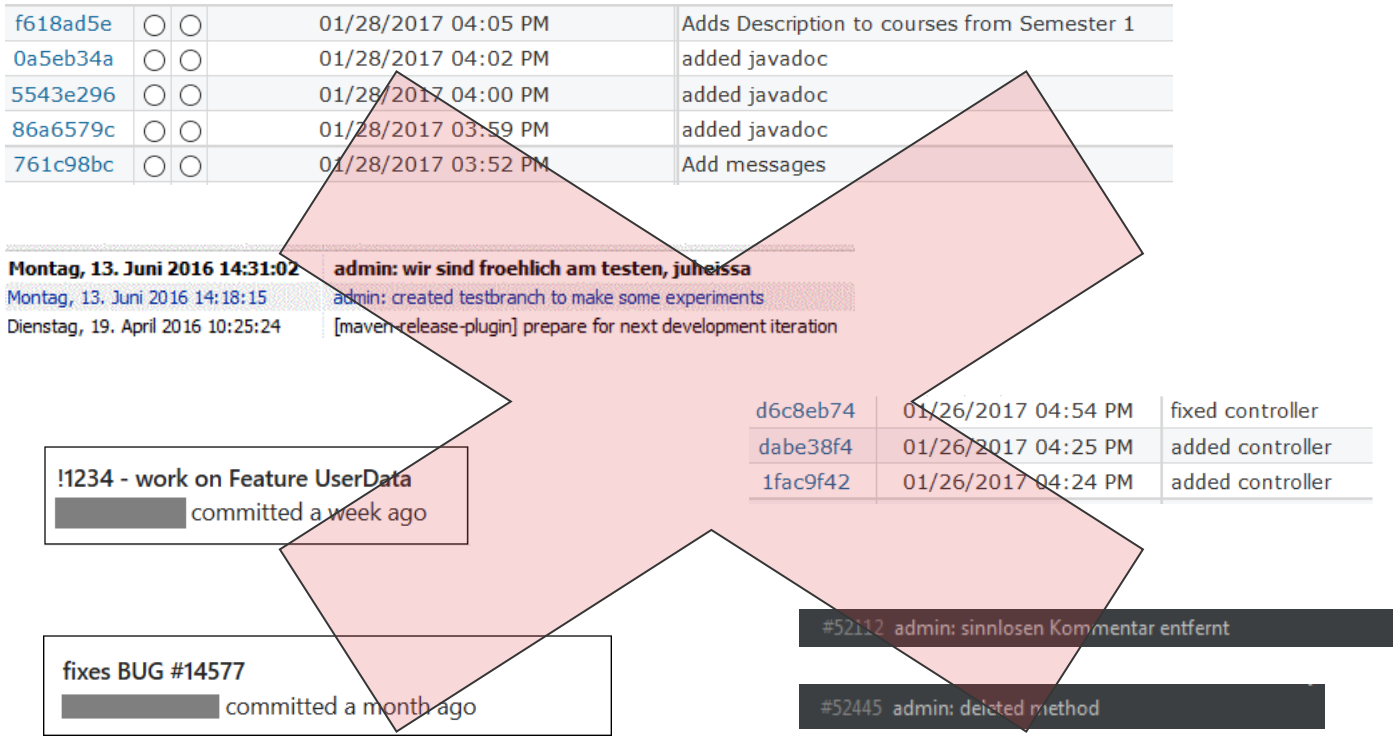


Commit Conventions

- Commit Messages
  - Dienen zumindest drei wichtigen Aspekten:
    - Beschleunigen den Review-Prozess
    - Unterstützen das Erstellen von Release Notes
    - Erleichtern zukünftige Wartung

  - Schreiben Sie ihre Commit Messages mit einem Zweck!
  - Gute Commit Messages:
    - Warum wurde diese Änderung gemacht?
    - Was wurde geändert?
    - Enthalten nur eine logische Änderung pro Commit
    - Idempotent (in sich geschlossen)
      - Zwingen Sie niemanden dazu externe Tools (z.B. Bugtracking Tools, …) zu benutzen, um einen Commit zu verstehen

Commit Conventions



Commit Conventions

- Commit Message – Best Practice

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

- Beispiel:

```
fix: prevent racing of requests

Introduce a request id and a reference to latest request.
Dismiss incoming responses other than from latest request.

Reviewed-by: Z
Refs: #123
```

Commit Conventions

- Conventional Commits
  - Basierend auf den Angular Konventionen
  - IntelliJ Plugin
- Vorteile
  - Changelogs
  - Nachvollziehbarkeit von Änderungen
  - Erhöht Wartbarkeit
  - Vereinfacht die Zusammenarbeit/Beteiligung

- https://www.conventionalcommits.org

Unit Tests

Testframework

- Komponententests werden idR automatisiert ausgeführt
- Implementierung wird mittels Unit Test Frameworks umgesetzt

- xUnit Frameworks
  - Überbegriff für Unit-Test-Frameworks in verschiedenen Programmiersprachen mit ähnlichem Design
  - JUnit – Java
  - NUnit - .Net
  - CPPUnit – C++
  - …

Testframework – Benennung

- Benennungsschema für Testmethoden
  - Einheitliches Benennungsschema erhöht
    - Lesbarkeit
    - Nachvollziehbarkeit

- Zusätzliche Informationen, falls notwendig
  - Spezielle Inputs
  - Spezielle Zustände
  - …

- `(test)<UseCase/Method>_should<ExpectedOutcome>`

Testframework – Benennung

- Benennungsschema Beispiele:
  - (test)<UseCase/Method>_should<ExpectedOutcome>
  - ```
    @Test
    public void testAdd_shouldContainAddedWord()
    ```
  - ```
    @Test
    public void testAdd_moreThanMax_shouldReplaceFirstWord()
    ```
  - ```
    @Test
    public void
    testRemove_Null_shouldThrowNullPointerException()
    ```
    ```

Testframework – Benennung

- Alternative:
  - `@DisplayName(<Beschreibung>)`
  - DisplayName kann Leerzeichen, Sonderzeichen, … enthalten und ersetzt im Testreport/IDE den Test Namen
  - ```
    @Test
    @DisplayName(„Error: Remove Nullvalue“)
    public void
    testRemove_Null_shouldThrowNullPointerException()
    ```
    ```

JUnit

- JUnit 5
- Lifecycle Annotations
  - `@BeforeAll`
  - `@AfterAll`
    - auf statische Methoden
    - Ausführung einmalig vor/nach allen Tests dieser Klasse
  - `@BeforeEach`
  - `@AfterEach`
    - Ausführung vor/nach jedem einzelnen Test
  - `@Test`
    - Spezifiziert einzelne Tests

```
class StandardTest {

    @BeforeAll
    static void initAll() {
        //...
    }

    @AfterAll
    static void tearDownAll() {
        //...
    }

    @BeforeEach
    void init() {
        //...
    }

    @AfterEach
    void tearDown() {
        //...
    }

    @Test
    void testAdd_ShouldDoSomething() {
        //...
    }

}
```

JUnit – Struktur eines Tests

- Aufbau eines (Unit) Tests
  - Given / When / Then
  - Arrange / Act / Assert

- Unterteilen des Tests in klare Abschnitte für
  - Vorbedingung / Aufbau
  - Ausführung
  - Validierung

- Vorteil:
  - Erhöhte Lesbarkeit / Nachvollziehbarkeit
- Nachteil:
  - Nicht immer klar anwendbar

JUnit – Assertions

- **Zusicherungen** zur Überprüfung, ob das zu testende Objekt den **korrekten Zustand** besitzt
- Statische Methoden der Klasse org.junit.jupiter.api.Assertions;
- Können für erhöhte Lesbarkeit mittels statischem Import importiert werden

```
import static org.junit.jupiter.api.Assertions.assertEquals;

assertEquals(1, person.getId(), “Person ID not as expected");
```

```
org.opentest4j.AssertionFailedError: Person ID not as expected ==>
Expected :1
Actual   :0
```

JUnit – Assertions

- AssertJ
  - Zusätzliches Assertion-Framework mit Fluent API
  - Verbessert Lesbarkeit und Nachvollziehbarkeit

  Beispiele:

```
assertEquals(customer.getId(), person.getId());

assertThat(person.getId()).isEqual(customer.getId());
```

JUnit – Assertions

JUnit

- assertEquals
- assertNotEquals
- assertTrue
- assertFalse
- assertNull
- assertNotNull
- …

AssertJ

- assertThat(x)
  - .isEqualTo
  - .isNotEqualTo
  - .isTrue
  - .isFalse
  - .isNull
  - .isNotNull
- …

JUnit – Assertions

- Grouped Assertions / Soft Assertions

- Gruppierungen von zusammenhängenden Assertions
- Fehlermeldungen werden gesammelt ausgewertet

```
assertAll(
    () -> assertTrue(. . .),
    () -> assertEquals(. . .)
)
```

JUnit – Testen von Exceptions

- Tests sollen nicht nur positive Fälle sondern auch Fehlerfälle abdecken
- JUnit Assertion
  - `assertThrows(Class<T> expectedType, Executable executable)`
- Beispiel

```
private int calculate(int a, int b) throws  CustomException {
    if(a < 0 || b < 0) {
        throw new CustomException();
    }
    return a + b;
}


@Test
void calculateNegativeInput_shouldThrowException() {
    assertThrows(CustomException.class, () -> calculate( a: -1 , b: 1));
}
```

JUnit – Parametrisierte Tests

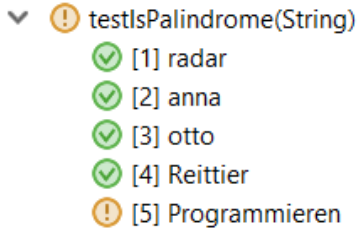
- Tests werden mehrmals mit unterschiedlichen Testdaten ausgeführt
- Benötigt eigene Dependency
  - `junit-jupiter-params`
- Benutzt die Annotation `@ParameterizedTest`
- Unterschiedliche Quelle der Testdaten
  - `@ValueSource(ints = { 1, 2, 3 })`
  - `@EnumSource(TimeUnit.class)`
  - `@MethodSource("stringProvider")`
  - `@CsvFileSource(resources = "/data.csv")`
  - …

JUnit – Parametrisierte Tests

- Beispiel Palindrom Überprüfung
  - Zeichenkette, die vorwärts wie rückwärts gelesen ident ist

```
@ParameterizedTest
@MethodSource("palindromeData")
void isPalindrome_shouldReturnTrue(String string) {
    assertThat(isPalindrome(string)).isTrue();
}

private static Stream<String> palindromeData() {
    return Stream.of("radar", "anna", "otto", "Reittier", "Programmieren");
}
```



JUnit - TempDir

- Temporäres Verzeichnis

```java
@Test  new *
void testWriteCsv_noData_shouldCrateEmptyFile (@TempDir Path tempDir) {
    List<Entry> entries = Collections.emptyList();

    String filepath = tempDir.toString() + "/output.csv";
    new CsvWriter().writeCsv(filepath, entries);

    List<Entry> lines = readFile(filepath);
    assertThat(lines).isEmpty();
}
```

JUnit – Extensions

- Erweiterungspunkte für Unit Tests
  - `org.junit.jupiter.api.extension`
    - `TestWatcher`
    - `BeforeTestExecutionCallback`
    - `AfterTestExecutionCallback`
    - `ParameterResolver`
    - …
- Annotation der Tests mit
  - `@ExtensWith(…)`
- Ermöglicht eine zentrale Behandlung von Erweiterungen
  - Z.B. Logging der Zeit pro Testfall
  - Tasks basierend auf Testergebnissen
  - Tasks die vor/nach einem Test ausgeführt werden sollen
  - …

JUnit – Extensions

- Beispiel: Precondition

```java
public class TestDataPrecondition implements BeforeEachCallback, AfterEachCallback {

    4 usages
    private static final Logger LOG = LoggerFactory.getLogger(TestDataPrecondition.class);

    @Override
    public void beforeEach(ExtensionContext extensionContext) {
        String testName = extensionContext.getRequiredTestMethod().getName();
        LOG.info("PRECONDITION STARTED");
        LOG.info("INSERTING DATA FOR: " + testName);
    }

    @Override
    public void afterEach(ExtensionContext extensionContext) {
        String testName = extensionContext.getRequiredTestMethod().getName();
        LOG.info("POSTCONDITION STARTED");
        LOG.info("DELETING DATA FOR: " + testName);
    }
}
```

```java
@Test
@ExtendWith(TestDataPrecondition.class)
void testLoad_shouldReturnResults() {
    //setup already done
}
```

JUnit – Extensions

- Beispiel: Parameter Resolver

```java
public class ParameterResolverDemo implements ParameterResolver {

    @Override
    public boolean supportsParameter(ParameterContext parameterContext, ExtensionContext extensionContext) throws ParameterResolutionException {
        return parameterContext.getParameter().getType() == PersonDTO.class;
    }

    @Override
    public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext) throws ParameterResolutionException {
        String testName = extensionContext.getRequiredTestMethod().getName();
        return new PersonDTO(
            testName,
            testName,
            email: testName + "@mail.com"
        );
    }
}
```

```java
@Test
@ExtendWith(ParameterResolverDemo.class)
void testSaveCustomer(PersonDTO person) {
    Assertions.assertEquals( expected: "testSaveCustomer@mail.com", person.getEmail());
}
```

JUnit – Conditional Execution

- Bedingungen für Testausführung
- Beispiel:
  - `@Disabled`
  - `@Enabled/Disabled OnOS(WINDOWS)`
  - `@Enabled/Disabled IfEnvironmentVariable (named = "LC_TIME", matches = ".*UTF-8.")`
  - `@Enabled/Disabled ifSystemProperty (named = "file.separator", matches = "[/]")`

JUnit – Conditional Execution

- Bedingungen für Testausführung
- Beispiel:
  - Eigene Bedingungen

```
public class CustomExecutionCondition implements ExecutionCondition {

    @Override  no usages  Markus Zoffi *
    public ConditionEvaluationResult evaluateExecutionCondition(ExtensionContext context) {
        return Service.loadConfiguration() == null
            ? ConditionEvaluationResult.disabled( reason: "Keine Konfiguration hinterlegt")
            : ConditionEvaluationResult.enabled( reason: "Konfiguration vorhanden");
    }
}
```

```
@Test  new *
@ExtendWith(CustomExecutionCondition.class)
void testKonfiguration(){
```

JUnit – Best Practices

- Annotation Wrapping
- Erstellen von Meta-Annotationen um häufig genutzte Annotationen zusammenzufassen
- Erhöht die Verständlichkeit und Wartbarkeit

JUnit – Best Practices

- Annotation Wrapping

```
@Target({ElementType.TYPE, ElementType.METHOD})  1 usage
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(CustomExecutionCondition.class)
public @interface DisabledIfNoKonfigurationIsPresent {
}
```

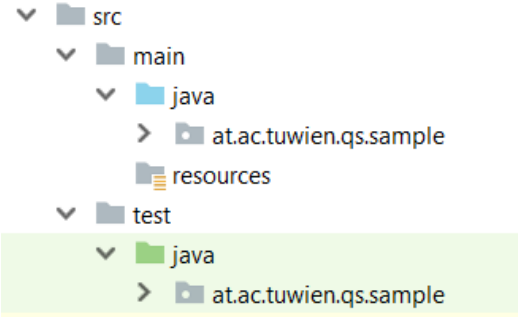
```
@Test  new *
@DisabledIfNoKonfigurationIsPresent
void testKonfiguration(){
```

```
@Target({ElementType.METHOD, ElementType.TYPE})  2 usages  new *
@Retention(RetentionPolicy.RUNTIME)
@Test
@Tag("external")
@ExtendWith(TimingExtension.class)
@ExtendWith({ExternalServiceExtension.class, AnotherExtension.class})
@ExtendWith(ExternalServiceParameterResolver.class)
public @interface ExternalServiceTest {
```

```
@ExternalServiceTest  new *
void testExternalService_should_CallExternalService() {
```

JUnit – Best Practices

- Testfälle sind isoliert
  - Jeder Testfall testet nur einen Aspekt bzw. Zustand
  - Keine Abhängigkeiten zwischen Testfällen
  - Jeder Testfall bereitet seine Daten vor und räumt sie auf
- Einhaltung von Namenskonventionen
- Klare Fehlermeldungen bei Assertions
- Tests sind Code – auf Lesbarkeit achten
  - Einhaltung von Code Guidelines / Refactorings etc.
- Klassen und Testklassen liegen im selben Package

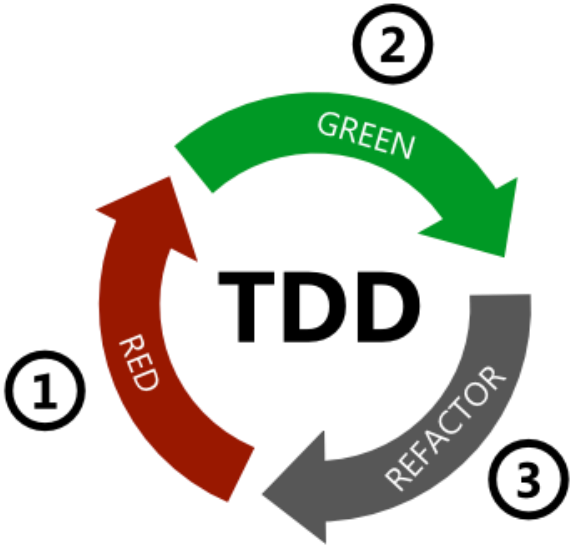


JUnit – Bad Practices

- Test testet nicht das gewünschte Verhalten
  - Test erfolgreich, unabhängig vom Ergebnis
- Test testet mehrere Zustände gleichzeitig
  - Mehrere Testfälle in einer Testmethode
- Zufällige Logik
  - Testdaten aus Zufallsdaten anstatt Äquivalenzklassen
  - Testdaten hardcoded im Test statt in sprechenden Variablen
- Überprüfung der Vorbedingungen
  - z.B. Überprüfung der in @BeforeEach ausgeführten Aktionen
- Logik in Tests (Abfragen, Schleifen, try/catch)

Test-Driven-Development (TDD)

- RED
  - Implementierung und Ausführung der Testfälle
    - Tests müssen fehlschlagen
- GREEN
  - Implementierung der Funktionalität
    - Testfall erfolgreich
- REFACTOR
  - Optimierung der Implementierung
    - Keine Änderung des Verhaltens
    - Testfälle dürfen nicht fehlschlagen



TDD Beispiel

- Testen eines Wörterbuchs
  - Wörterbuch
    - Enthält Wörter ohne Duplikate
    - Umfasst folgende Operationen:
      - `add` – Hinzufügen von Wörtern
      - `first` – Liefert das erste Element
      - `last` – Liefert das letzte Element
      - `get` – Abrufen eines Elements
      - `size` – Liefert die Anzahl der enthaltenen Wörter

TDD Beispiel

- 0. Interface/Klasse vorbereiten
  - Erstellen der Klasse
    - mit Methoden Signaturen
    - syntaktisch korrekter Code

```
public class Dictionary {

    private List<String> words;

    public Dictionary() {
        words = new ArrayList<>();
    }

    public void add(String word){
    }

    public String get(Integer position){
        return null;
    }

    public String first(){
        return null;
    }

    public String last(){
        return null;
    }

    public Integer size(){
        return null;
    }
}
```

TDD Beispiel

- 1. Testfall schreiben
  - Auswahl der Anforderung
  - Spezifikation der Testfälle
  - Test muss fehlschlagen

```java
public class DictionaryTest {

    public static final String WORT1 = "Software Testen";
    private Dictionary dictionary;

    @BeforeEach
    void init(){
        dictionary = new Dictionary();
    }

    @AfterEach
    public void tearDown(){
        dictionary = null;
    }

    @Test
    public void add_shouldAddWord(){
        dictionary.add(WORT1);
        assertThat(dictionary.get(0)).isEqualTo(WORT1);
    }

    @Test
    public void add_sameWord_shouldThrowIAE(){
        dictionary.add(WORT1);
        assertThrows(IllegalArgumentException.class,
                () -> dictionary.add(WORT1));
    }
}
```

TDD Beispiel

- 2. Implementierung
  - Implementierung der Methoden
    - `add`
    - `get`
  - Testfall muss erfolgreich durchlaufen

```
public class Dictionary {

    private List<String> words;

    public Dictionary() {
        words = new ArrayList<>();
    }

    public void add(String word){
        if(words.contains(word)){
            throw new IllegalArgumentException
                    ("Wort ist bereits enthalten");
        }
        words.add(word);
    }

    public String get(Integer position){
        return words.get(position);
    }

    public String first(){
        return null;
    }

    public String last(){
        return null;
    }

    public Integer size(){
        return null;
    }
}
```

TDD Beispiel

- 3. Refactoring
  - Refactoring der Implementierung
    - Tests müssen weiterhin erfolgreich durchlaufen
  - Wiederhole TDD Zyklus mit nächster Anforderung
  - Commit in SCM

Testabdeckung

- Coverage Reports

```
11      public class Dictionary {
12
13          private List<String> words;
14
15          public Dictionary() {
16              words = new ArrayList<>();
17          }
18
19          public void add(String word){
20              if(words.contains(word)){
21                  throw new IllegalArgumentException
22                          ("Wort ist bereits enthalten");
23              }
24              words.add(word);
25          }
26
27          public String g
28              return word
29          }
30
31          public String f
32              return word
33          }
34
35          public String l
36              return word
37          }
38
39          public Integer
40              return word
41          }
42      }
43
```

Coverage

TestAllPackages

Element	Coverage	Covered Lines	Missed Lines	Total Lines
commons-collections	80.7 %	11092	2646	13738
src	80.7 %	11092	2646	13738
org.apache.commons.collections	77.1 %	3991	1188	5179
org.apache.commons.collections.bag	66.9 %	234	116	350
org.apache.commons.collections.bidimap	91.2 %	964	93	1057
AbstractBidiMapDecorator.java	85.7 %	6	1	7
AbstractBidiMapDecorator	85.7 %	6	1	7
AbstractBidiMapDecorator(BidiMap)	100.0 %	2	0	2
getBidiMap()	100.0 %	1	0	1
getKey(Object)	100.0 %	1	0	1
inverseBidiMap()	0.0 %	0	1	1

Referenzen

- Junit 5 UserGuide [https://junit.org/junit5/docs/current/user-guide]
- AssertJ Guide [https://joel-costigliola.github.io/assertj]
- ArchUnit Guide [https://www.archunit.org/]
- Thomas Grechenig, Mario Bernhart, Roland Breiteneder, Karin Kappel: „Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten“, Pearson Studium, 2009