UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Software Entwicklung & Programmierung

Test-Driven Development (TDD)

PALUNO
The Ruhr Institute for Software Technology

Disclaimer

Bilder und Texte der Veranstaltungsfolien und -unterlagen sowie das gesprochene Wort innerhalb der Veranstaltung und Lehr-Lern-Videos dienen allein dem Selbst- bzw. Gruppenstudium. Jede weiterführende Nutzung ist den Teilnehmenden der Moodle-Kurse untersagt, z.B. Verbreitung an andere Studierende, in sozialen Netzwerken, dem Internet!

Darüber hinaus ist ein studentischer Mitschnitt von Webkonferenzen im Rahmen der Lehre nicht erlaubt.

Zielsetzung

Am Ende dieser Präsentation könnt Ihr:

- Die **Motivation** für das Test-Driven Development erläutern
- Die **Kernkonzepte** und **Vorgehensweise** des **Test-Driven Development** erläutern
- Eine erste eigene **Implementierung** anhand des Test-Driven Development Konzeptes selbstständig durchführen

Agenda

1. **Motivation**
2. Grundlagen
3. Werkzeuge
4. Beispiel TDD

Motivation

- In der Softwareentwicklung können unterschiedliche Probleme auftreten:
  - Hohe Wartungskosten
  - Die gelieferte Software entspricht nicht dem, was der Kunde erwartet
  - Das Softwareprodukt kann nicht rechtzeitig geliefert werden, da die Testphase zu lange dauert
  - Im Entwicklungsprozess spät angesiedelte Entwicklung und Durchführung automatisierter Tests tragen kaum zum Endprodukt bei - Wertbeitrag der Tests zweifelhaft
  - Das Test-Team und das Entwicklungs-Team sind unabhängig voneinander und arbeiten asynchron

- Durch diese und weitere Probleme hat sich in der **agilen Softwareentwicklung** die testgetriebene Entwicklung als eines der wichtigsten Tools etabliert

Agenda

1. Motivation
2. **Grundlagen**
3. Werkzeuge
4. Beispiel TDD

Grundlagen (1)

- Beim **Test-Driven Development (TDD)** sind die Erstellung von **Code** und **(Modul-) Tests** eng verzahnt
  - Im Gegensatz zum traditionellen Vorgehen in der Software-Entwicklung mit der Unterteilung und klaren Abfolge von Implementierung und anschließendem Testen
  - Fokus liegt hier auf den Modultests
  - Definition von Systemtestfällen (s. Foliensatz „Systemtests“) und Entwicklung bleibt weiterhin weitgehend unabhängig

Grundlagen (2)

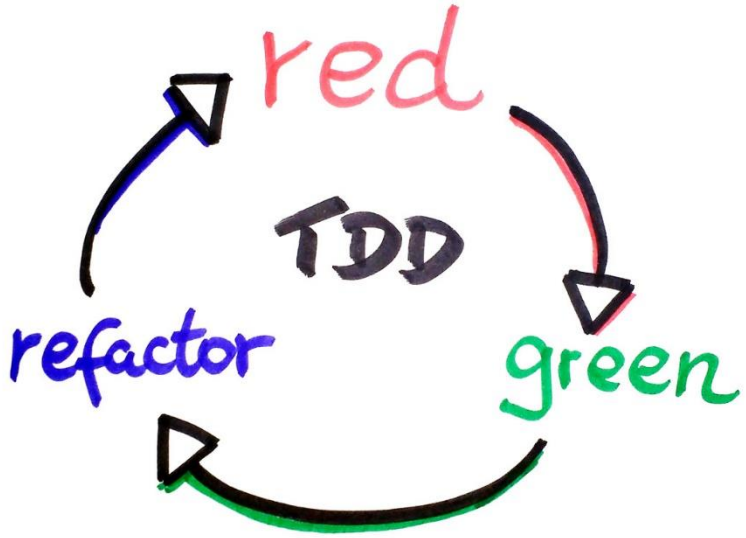
- Die Tests werden **vor der eigentlichen Implementierung** geschrieben. Dadurch erfolgt eine **Inversion des traditionellen Ansatzes**, bei dem das Testen nach dem Erstellen des Codes stattfindet

- Der Fokus liegt nicht auf dem Erstellen von Tests, sondern es geht vielmehr darum, sich während der Implementierung durch seine selbst erstellten Tests „**treiben**“ zu lassen
  - Aufweichung des traditionellen SE-Prinzips „Unabhängigkeit der Qualitätsprüfung“ (vgl. s. Foliensatz „Modultests“)

Grundsätzliche Vorgehensweise

1. Definition des Verhaltens, das zurzeit noch nicht vom Code erfüllt wird

2. Erstellen und ausführen des Tests, um sicher zu stellen, dass dieser **nicht** funktioniert („***Test runs red***“)



3. Erweiterung des Codes, sodass der Test nun funktioniert („***Test runs green***“)

4. Beliebig oft verschönern, perfektionieren und optimieren des Codes und jederzeit verifizieren, dass die implementierte Methode die Funktionalität weiterhin erfüllt (**Refaktorisieren**)

Das Prinzip „F.I.R.S.T. Class“ (1)

- Bei der Erstellung der Tests sollte das Prinzip „**F.I.R.S.T. Class**“ befolgt werden
- **Fast:** Die Testfälle sollten schnell auszuführen sein. Das Ausführen aller Tests sollte nicht länger als ein paar Sekunden für eine kleine Applikation dauern.
- **Isolated/Independent:** Die Tests können in jeder Reihenfolge oder sogar gleichzeitig laufen.
- **Repeatable:** Das Ergebnis der Tests sollte immer das gleiche sein, unabhängig davon, wie oft diese wiederholt und auf welchem System diese durchgeführt werden.

Das Prinzip „F.I.R.S.T. Class“ (2)

- **Self-Validating**: Die Tests geben an, ob sie bestanden oder fehlgeschlagen sind. Dies ist eine built-in Funktionalität des xUnit Frameworks. Fehlgeschlagene Tests sollten den Ort der Fehlerquelle genau angeben.

- **Timely**: Zuerst die Tests erstellen, um anschließend testbaren Code zu erhalten

Agenda

1. Motivation
2. Grundlagen
3. **Werkzeuge**
4. Beispiel TDD

Werkzeuge

- **JUnit** Test-Framework für die automatisierte Erstellung von Tests in Java
- **Hamcrest** Java Assertion Frameworks
- **AssertJ** Java Assertion Frameworks
- Weitere Frameworks sind z.B. TestNG (Automatisierung von Unit Tests und vergleichbar mit JUnit). In dieser Veranstaltung liegt der **Fokus auf JUnit!**

Werkzeuge - JUnit

- **JUnit** ist ein Framework zur Erstellung und Ausführung von Tests in Java
- Jeder Test stellt eine eigene Methode dar und jede Methode deckt ein spezielles Szenario ab, das definiert, wie sich der Code zu verhalten hat
- Die Grundkonzepte und eine Einführung in die Implementierung von JUnit findet Ihr in dem Foliensatz „**Modultests**“.

Werkzeuge - Hamcrest

- **Hamcrest** wird bereits im Kern von JUnit unterstützt
- Das komplette Hamcrest-Framework bietet zudem eine nützliche Erweiterung für JUnit
- Vereinfachung des „*assert*“-Befehls durch die Verwendung von sogenannten „**Matchers**“
  - Matchers ermöglichen eine individuelle, selbst definierte Vergleichs-Operation durchzuführen

Werkzeuge - Hamcrest

Beispiel „*Matcher*“

- **Implementierung**

```
public static Matcher<Person> hasFirstName(String firstName) {
  return new FeatureMatcher<Person, String>(equalTo(firstName),
     "a person with first name", "first name") {
    @Override
    protected String featureValueOf(Person actual) {
       return actual.getFirstName();
    }
   };
}
```

- **Aufruf**

```
assertThat(bob, hasFirstName("Alice"));
```

- **Ausgabe**

```
Expected: a person with first name "Alice"
but: first name was "Bob"
```

Werkzeuge

Vergleich JUnit und Hamcrest

JUnit

assert

```
List<String> friendsOfJoe =
  Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
Assert.assertTrue( friendships.getFriendsList("Joe")
    .containsAll(friendsOfJoe));
```

Hamcrest

assert

```
assertThat(
  friendships.getFriendsList("Joe"),
  containsInAnyOrder("Audrey", "Peter", "Michael", "Britney", "Paul")
);
```

Werkzeuge - AssertJ

- **AssertJ** funktioniert ähnlich wie Hamcrest
- Der Hauptunterschied besteht darin, dass mit AssertJ die Assertions **aneinandergereiht** werden können

JUnit:

```
Assert.assertEquals(5, friendships.getFriendsList("Joe").size());
List<String> friendsOfJoe =
   Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
Assert.assertTrue(  friendships.getFriendsList("Joe")
   .containsAll (friendsOfJoe)
);
```

AssertJ:

```
assertThat(friendships.getFriendsList("Joe"))
  .hasSize(5)
  .containsOnly("Audrey", "Peter", "Michael", "Britney", "Paul");
```

Werkzeuge

Zusammenfassung Test-Frameworks

- **Individuelle Anpassung** von Test Cases
- **Zeitersparnis** bei einmaliger Nutzung nicht sehr hoch, aber je größer das Projekt, desto mehr Zeitersparnis
- **Qualität** der Tests **steigt** und **Fehleranfälligkeit sinkt**

Dokumentation Hamcrest: http://hamcrest.org/JavaHamcrest/index

Dokumentation AssertJ: https://assertj.github.io/doc/

Agenda

1. Motivation
2. Grundlagen
3. Werkzeuge
4. **Beispiel TDD**

Beispiel TDD

1. Definiere das gewünschte Verhalten der Methode

Es soll eine Methode implementiert werden, die wie das Spiel „*FizzBuzz*“ (https://de.wikipedia.org/wiki/Fizz_buzz) funktioniert.

```
Die Methode erhält eine Zahl als Eingabeparameter.
Rückgabewert der Methode soll die Zahl sein, außer:
- die Zahl ist durch 3 teilbar, dann gib "Fizz" aus.
- die Zahl ist durch 5 teilbar, dann gib "Buzz" aus.
- die Zahl ist durch 3 UND 5 teilbar, dann gib "FizzBuzz" aus.
```

Beispiel TDD

– die Zahl ist durch 3 teilbar, dann gib "Fizz" aus.

2. Erstelle aufbauend auf einer Anforderung einen **Test**

- Eingabeparameter: „*3*“
- Soll-Ausgabe: „*Fizz*“
- Durchführen des Tests mit dem Ergebnis „***Test runs red***“

```
30      @Test
31      void FizzBuzzTestWith3() {
32          assertThat(Main.FizzBuzz( number: 3), is(equalTo( operand: "Fizz")));
33      }
34
35
36
37  }
38
```

Run: TDDTest.FizzBuzzTestWith3 | Rerun Failed Tests

Tests failed: 1 of 1 test – 19 ms

Test Results 19 ms
TDDTest 19 ms
FizzBuzzTestWith3() 19 ms

```
Expected: is "Fizz"
     but: was "3"
java.lang.AssertionError:
Expected: is "Fizz"
     but: was "3"
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:6)
    at TDDTest.FizzBuzzTestWith3(TDDTest.java:32) <31 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <46 internal calls>
```

Beispiel TDD

- die Zahl ist durch 3 teilbar, dann gib "Fizz" aus.

3. Methode anpassen, sodass das **gewünschte Verhalten** erzielt wird

(Zeile 9 wurde hinzugefügt)

```
7     public class Main {
8         public static String FizzBuzz(Integer number){
9             if(number%3==0) return "Fizz";
10            return String.valueOf(number);
11        }
12    }
13
```

Beispiel TDD

- die Zahl ist durch 3 teilbar, dann gib "Fizz" aus.

Erneutes Durchführen des Tests nach dem Anpassen der Methode

```
33      @Test
34      void FizzBuzzTestWith3() {
35          assertThat(Main.FizzBuzz( number: 3), is(equalTo( operand: "Fizz")));
36      }
37
```

Run: FizzBuzz TDD

Tests passed: 3 of 3 tests – 16 ms

Test Results	16 ms
TDDTest	16 ms
FizzBuzzTestWith1()	16 ms
FizzBuzzTestWith2()	
FizzBuzzTestWith3()	

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...

Process finished with exit code 0
```

- *„**Test runs green**“* und erzielt das gewünschte Verhalten

→ Wiederholen aller Schritte, bis **alle Anforderungen** erfüllt werden!

Beispiel TDD

Resultat

Methode **erfüllt alle Anforderungen** und Ausnahmefälle wurden ebenfalls durch Tests abgedeckt

```java
public class Main {
    public static String FizzBuzz(Integer number){
        if(number%15==0) return "FizzBuzz";
        else if(number%3==0) return "Fizz";
        else if (number%5==0) return "Buzz";
        return String.valueOf(number);
    }
}
```

```java
@Test
void FizzBuzzTestWith6() {
    assertThat(Main.FizzBuzz( number: 6), is(equalTo( operand: "Fizz")));
}

@Test
void FizzBuzzTestWith10() {
    assertThat(Main.FizzBuzz( number: 10), is(equalTo( operand: "Buzz")));
}

@Test
void FizzBuzzTestWith15() {
    assertThat(Main.FizzBuzz( number: 15), is(equalTo( operand: "FizzBuzz")));
}

@Test
void FizzBuzzTestWith45() {
    assertThat(Main.FizzBuzz( number: 45), is(equalTo( operand: "FizzBuzz")));
}
}
```

Run: FizzBuzz TDD

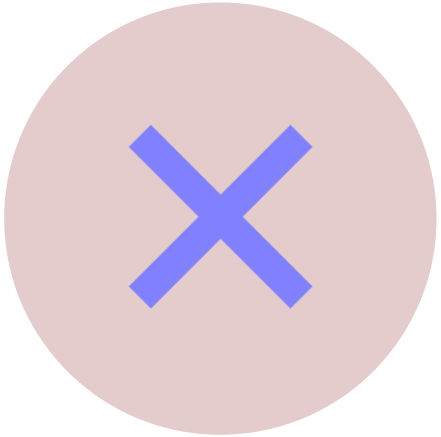
Tests passed: 8 of 8 tests – 32 ms

Test Results 32 ms
TDDTest 32 ms
FizzBuzzTestWith1() 16 ms
FizzBuzzTestWith2()
FizzBuzzTestWith3()
FizzBuzzTestWith5() 16 ms
FizzBuzzTestWith6()
FizzBuzzTestWith10()
FizzBuzzTestWith15()
FizzBuzzTestWith45()

"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...

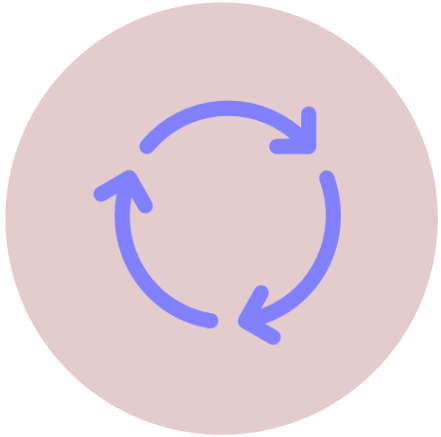
Process finished with exit code 0

Zusammenfassung
Überblick Vorteile TDD



Kein ungetesteter Code

Aufwändige Fehlersuche erübrigt sich, nur getestete Methoden werden zugelassen



Keine/wenig Redundanzen durch regelmäßige Refaktorisierung

Quellen

- [Beck 2009] Kent Beck: *Test Driven Development, By Example.* Addison-Wesley signature series, 2009
- [Freeman et al. 2010] Steve Freeman,Nat Pryce: *Growing object oriented Software, guided by tests.* Addison-Wesley signature series, 2010
- [Garcia et al. 2018] Viktor Farcic, Alex Garcia: *Test-Driven Java Development*. 2nd edition, 2018
- https://www.informatik-aktuell.de/entwicklung/methoden/tdd-erfahrungen-bei-der-einfuehrung.html
- https://www.sigs-datacom.de/uploads/tx_dmjournals/philipp_JS_06_15_gRfN.pdf

Verwendete Grafiken

- https://www.informatik-aktuell.de/fileadmin/templates/wr/pics/Artikel/02_Entwicklung/Methoden/TDD-abb-Fichtner.jpg

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Vielen Dank für Eure Aufmerksamkeit

PALUNO
The Ruhr Institute for Software Technology