

Softwaretechnik 1 - 0. Tutorium

Tutorium 03

Felix Bachmann | 27. April 2017

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



- 1 Organisatorisches
- 2 Vorbereitungsblatt
- 3 JUnit4
- 4 Maven
- 5 Git
- 6 Live-Demo
- 7 Tipps

- Felix Bachmann
- Infostudent im 4. Semester
- erstes Tutorium
- E-Post-Adresse: felix.bachmann@ewetel.net

- Name
- Studiengang und Semester
- erlernte Programmiersprachen, Lieblingsprogrammiersprache
- Erfahrung mit Git/Maven oder ähnlichen Tools?
- Von dem Tutorium erwarte ich...

cool

- **mitdenken**
- **Fragen stellen**
- **Fragen beantworten**
- essen & trinken
- gehen
- schlafen

!cool

- laut sein
- stören
- andere ablenken

cool

- **mitdenken**
- **Fragen stellen**
- **Fragen beantworten**
- essen & trinken
- gehen
- schlafen

!cool

- laut sein
- stören
- andere ablenken

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls
- neue Übungsblätter ungefähr alle 2 Wochen \implies 1+6 Blätter
- ab 50% der Punkte habt ihr sicher bestanden
- Besprechung der Musterlösung
- Abgaben
 - Theorieaufgaben (handschriftlich und leserlich!) + Deckblatt im 3.Stock
 - Programmieraufgaben auf <http://lez.ipd.kit.edu>
 - Plagiate können teuer werden
 - Deadlines sind hart!
 - keine Abgabe per Mail!

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls
- neue Übungsblätter ungefähr alle 2 Wochen \implies 1+6 Blätter
- ab 50% der Punkte habt ihr sicher bestanden
- Besprechung der Musterlösung
- Abgaben
 - Theorieaufgaben (handschriftlich und leserlich!) + Deckblatt im 3.Stock
 - Programmieraufgaben auf <http://lez.ipd.kit.edu>
 - Plagiate können teuer werden
 - Deadlines sind hart!
 - keine Abgabe per Mail!

- Wann?: ab dem 15.05 14-tägig
- Wo?: Raum -107
- Was?:
 - Wiederholung des VL-Stoffs
 - “Rechnen“ von Aufgaben (Altklausuren)
 - ggf. Tipps für die Übungsblätter
- Folien gibt's im Ilias und auf www.github.com/malluce/swt1-tut
- Fragen stellen !!

erst im Forum, auf Google oder Stackoverflow nachschauen, dann

- neuen Forum-Thread anlegen
- falls nicht öffentlich postbar: Mail an mich oder swt1@ipd.kit.edu (nur im Notfall)

- Programmieren \implies SWT1 \implies PSE
- den Hacker strukturieren
- den Umgang mit wichtigen Tools (insb. Build-Management-Tools, Versionsverwaltung) erlernen

Was ihr bisher getan haben solltet..

Installation von:

- Eclipse (incl. CheckStyle und EcJemma)

Überblick über:

- Maven
- Git

Tut euch den Gefallen

- Installiert Git manuell!

Probleme mit der Installation? \implies kommt nach dem Tut nach vorne

- Unittest-Tool für Java-Klassen
- über die pom.xml mit scope “test” einbinden
- Nur öffentliche Methoden testen
- Konventionen:
 - Für Klasse Hallo Testklasse HalloTest schreiben
 - Methode hallo(Object o) wird z.B. durch die Methode testHalloWithNull() getestet

Methoden können mit Annotationen (@XYZ) versehen werden
Aufbau:

- @BeforeClass (wird als erstes einmal ausgeführt)
- @Before (wird vor jeder Test-Methode einmal ausgeführt)
- @Test (vergleichen erwartetes und reales Ergebnis, schlagen ggf. fehl, Ausführung in beliebiger Reihenfolge)
- @After (wird nach jeder Test-Methode einmal ausgeführt)
- @AfterClass (wird am ende einmal ausgeführt)

Methoden können mit Annotationen (@XYZ) versehen werden
Aufbau:

- @BeforeClass (wird als erstes einmal ausgeführt)
- @Before (wird vor jeder Test-Methode einmal ausgeführt)
- @Test (vergleichen erwartetes und reales Ergebnis, schlagen ggf. fehl, Ausführung in beliebiger Reihenfolge)
- @After (wird nach jeder Test-Methode einmal ausgeführt)
- @AfterClass (wird am ende einmal ausgeführt)

Methoden können mit Annotationen (@XYZ) versehen werden
Aufbau:

- @BeforeClass (wird als erstes einmal ausgeführt)
- @Before (wird vor jeder Test-Methode einmal ausgeführt)
- @Test (vergleichen erwartetes und reales Ergebnis, schlagen ggf. fehl, Ausführung in beliebiger Reihenfolge)
- @After (wird nach jeder Test-Methode einmal ausgeführt)
- @AfterClass (wird am ende einmal ausgeführt)

Methoden können mit Annotationen (@XYZ) versehen werden
Aufbau:

- @BeforeClass (wird als erstes einmal ausgeführt)
- @Before (wird vor jeder Test-Methode einmal ausgeführt)
- @Test (vergleichen erwartetes und reales Ergebnis, schlagen ggf. fehl, Ausführung in beliebiger Reihenfolge)
- @After (wird nach jeder Test-Methode einmal ausgeführt)
- @AfterClass (wird am ende einmal ausgeführt)

Methoden können mit Annotationen (@XYZ) versehen werden
Aufbau:

- @BeforeClass (wird als erstes einmal ausgeführt)
- @Before (wird vor jeder Test-Methode einmal ausgeführt)
- @Test (vergleichen erwartetes und reales Ergebnis, schlagen ggf. fehl, Ausführung in beliebiger Reihenfolge)
- @After (wird nach jeder Test-Methode einmal ausgeführt)
- @AfterClass (wird am ende einmal ausgeführt)

- org.junit.Assert bietet diverse Methoden, um Ergebnis mit Erwartung abzugleichen
- zu jeder Methode kann als erstes Argument ein String mitgegeben werden (wird bei Fehlschlag angezeigt)

Beispiele:

- `assertArrayEquals(int[] expected, int[] actual)`
- `assertNotNull(Object obj)`
- `assertSame(Object expected, Object actual)`

Zu testende Methode in der Klasse Hallo

```
public static int add(int a, int b) {  
    return a + b;  
}
```

Testmethode in der Klasse HalloTest

```
@Test  
public void testAdd() {  
    Assert.assertEquals(7, Hallo.add(5, 2));  
}
```

(mehr Beispiele später)

A, B oder C?

Welche Annotation führt dazu, dass die annotierte Methode nach jeder mit `@Test` versehenen Methode einmal ausgeführt wird?

- A: `@Ignore`
- B: `@After`
- C: `@AfterClass`

Wahr oder falsch?

Die mit `@Test` versehenen Methoden werden in der Reihenfolge ausgeführt, in der sie im Quellcode stehen.

Wahr oder falsch?

Um Ergebnisse von Methodenaufrufen mit dem erwarteten Ergebnis abzugleichen, benutzt man Methoden aus `junit.framework.Assert`.

A, B oder C?

Welche Annotation führt dazu, dass die annotierte Methode nach jeder mit `@Test` versehenen Methode einmal ausgeführt wird?

- A: `@Ignore`
- B: `@After`
- C: `@AfterClass`

Wahr oder falsch?

Die mit `@Test` versehenen Methoden werden in der Reihenfolge ausgeführt, in der sie im Quellcode stehen.

Wahr oder falsch?

Um Ergebnisse von Methodenaufrufen mit dem erwarteten Ergebnis abzugleichen, benutzt man Methoden aus `junit.framework.Assert`.

A, B oder C?

Welche Annotation führt dazu, dass die annotierte Methode nach jeder mit `@Test` versehenen Methode einmal ausgeführt wird?

- A: `@Ignore`
- B: `@After`
- C: `@AfterClass`

Wahr oder falsch?

Die mit `@Test` versehenen Methoden werden in der Reihenfolge ausgeführt, in der sie im Quellcode stehen.

Wahr oder falsch?

Um Ergebnisse von Methodenaufrufen mit dem erwarteten Ergebnis abzugleichen, benutzt man Methoden aus `junit.framework.Assert`.

- Maven ist Jiddisch und heißt “Sammler des Wissens“
- Build-Management-Tool (Automatisierung von möglichst vielen Schritten)
- Maven ist in jeder Eclipse-Installation integriert
⇒ keine manuelle Installation nötig
- Aufgaben von Maven
 - Strukturierung (durch vorgegebene Verzeichnisstruktur)
 - Kompilieren
 - Testen
 - Verwalten von Abhängigkeiten
 - Verpacken

Verzeichnisstruktur:

- src
 - main
 - java
 - resources
 - test
 - java
 - resources
- target
 - classes
 - test-classes
 - *.jar / *.war / *.zip ...
 - ...
- pom.xml

- pom steht für “Project Object Model“
- konfiguriert euer Maven Projekt im XML-Format (gefüllt durch default-Werte)
 - Wo sucht Maven Tests?
 - Wohin speichert Maven Build-Dateien?
 - In welches Format soll das Projekt verpackt werden?
 - ...
- Eclipse-Plugin bietet GUI














Wichtige Befehle

<code>mvn compile</code>	kompiliert Quelltexte zu .class-Dateien
<code>mvn test</code>	kompiliert Test-Quelldateien zu .class-Dateien, führt Tests aus und zeigt Ergebnisse an
<code>mvn package</code>	verpackt euer Projekt in eine Datei (.war/.jar/.zip)
<code>mvn clear</code>	leert euren target-Ordner

Lösungsansätze:

- Rechtsklick auf Projekt \implies Maven \implies Update Maven Project
 \implies Haken bei "Force Update..."
 - Synchronisiert pom.xml mit Projekt, aktualisiert Abhängigkeiten
- mvn clean
 - vielleicht war der target-Ordner verschmutzt
- C:/Users/MeinName/.m2/ löschen und mvn compile (oder mvn package) ausführen
 - löscht alle Dependencies und lädt sie neu runter (ab und zu lädt man leider korrupte Dateien runter oder Dateien fehlen)

Warum Versionsverwaltung?

 final1-09-03(changed split-method)	01.07.2016 17:47	Dateiordner
 final1-12-02	01.07.2016 17:47	Dateiordner
 final1-13-02	01.07.2016 17:47	Dateiordner
 final1-14-02	01.07.2016 17:47	Dateiordner
 final1-15-02	01.07.2016 17:47	Dateiordner
 final1-16-02	01.07.2016 17:47	Dateiordner
 final1-17-02	01.07.2016 17:47	Dateiordner
 final1-20-02(1)	01.07.2016 17:47	Dateiordner
 final1-20-02(2)	01.07.2016 17:47	Dateiordner
 final1-25-02(passed public tests)	01.07.2016 17:47	Dateiordner
 final1-26-02(all commands implemented)	01.07.2016 17:47	Dateiordner
 final1-27-02(version 1.0 - works so far)	01.07.2016 17:47	Dateiordner
 final1-29-02(version 1.1 - finished)	01.07.2016 17:47	Dateiordner

So nicht!



- git ist Englisch, bedeutet Schwachkopf, Penner oder Nudelauge (?)
- dezentrales Versionsverwaltungssystem
- wichtig! (universell)

Nötig?

Wichtige Befehle - Navigation

<code>cd test</code>	Wechselt in das Verzeichnis test.
<code>dir</code> bzw. <code>ls</code>	Zeigt Inhalt des aktuellen Ordners an.
<code>.</code>	= aktuelles Verzeichnis
<code>..</code>	= übergeordnetes Verzeichnis

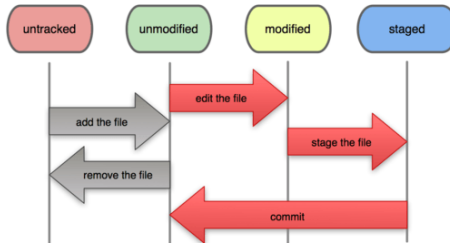
Hacks

- Mit den Pfeiltasten können bereits eingegebene Befehle durchgescrollt werden.
- Tabulator = Autovervollständigung

Wichtige Befehle

<code>git init</code>	Initialisiert ein leeres Git-Repo.
<code>git log</code>	Zeigt alle vergangenen Commits.
<code>git status</code>	Zeigt den Status der Dateien im Repo.
<code>git checkout</code>	Lässt HEAD zwischen Commits springen.
<code>git add</code>	Fügt Datei(en) zur Staging Area hinzu.
<code>git commit -m "message"</code>	Erzeugt einen Commit.

File Status Lifecycle



- Datei, die Namen von Pfaden/ Dateien enthält, die von git ignoriert werden sollen (z.B IDE-spezifisches)
- Beispiele:
 - target/
 - *.java
 - dis.like
- # dient als Kommentar-Zeichen

Live-Demo

Aufgabe 1: Altsoftware vorbereiten

- löchriges Kochrezept für Umgang mit Maven, Git, Checkstyle - da müsst ihr durch
- Google ist euer Freund (meistens)

Aufgabe 2: Modultests

- Aufgaben zum Testen mit JUnit4
- Ordner sollen erstellt werden, wenn sie nicht existieren
- Asserts benutzen !

Aufgabe 3: Testüberdeckung

- Mockito klingt komplizierter als es ist (schaut mal auf <https://www.javacodegeeks.com/2012/05/mocks-and-stubs-understanding-test.html>)

Aufgabe 1: Altsoftware vorbereiten

- löchriges Kochrezept für Umgang mit Maven, Git, Checkstyle - da müsst ihr durch
- Google ist euer Freund (meistens)

Aufgabe 2: Modultests

- Aufgaben zum Testen mit JUnit4
- Ordner sollen erstellt werden, wenn sie nicht existieren
- Asserts benutzen !

Aufgabe 3: Testüberdeckung

- Mockito klingt komplizierter als es ist (schaut mal auf <https://www.javacodegeeks.com/2012/05/mocks-and-stubs-understanding-test.html>)

Aufgabe 1: Altsoftware vorbereiten

- löchriges Kochrezept für Umgang mit Maven, Git, Checkstyle - da müsst ihr durch
- Google ist euer Freund (meistens)

Aufgabe 2: Modultests

- Aufgaben zum Testen mit JUnit4
- Ordner sollen erstellt werden, wenn sie nicht existieren
- Asserts benutzen !

Aufgabe 3: Testüberdeckung

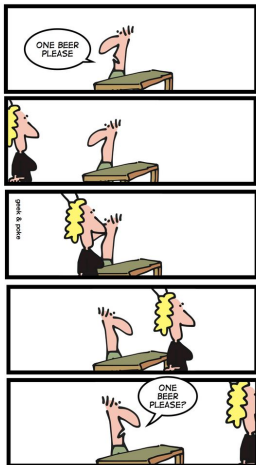
- Mockito klingt komplizierter als es ist (schaut mal auf <https://www.javacodegeeks.com/2012/05/mocks-and-stubs-understanding-test.html>)

Abgabe

- in der LEZ bis zum 10.05, 12:00
- falls ihr ein Feedback wollt, werft das Deckblatt ein

Bis dann! (dann=15.05.17)

SIMPLY EXPLAINED



.gitignore

geek-and-poke.com/geekandpoke/2012/11/7/simply-explained.html