

Softwaretechnik 1 - 0. Tutorium

Tutorium 18

Felix Bachmann | 24.04.2018

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



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

- E-Post-Adresse: *Name* *felix.bachmann@ewetel.net*
- Infostudent im 6. Semester
- zweites Tutorium
 - SS17: SWT1

- Name
- Programmiersprachen?
- Erfahrung mit Git/Maven o.Ä.?
- Von dem Tutorium erwarte ich...

- Wann?: ab heute 14-tägig
- Wo?: Raum -119
- Was?:

- Wann?: ab heute 14-tägig
- Wo?: Raum -119
- Was?:
 - Feedback letztes Blatt
 - Wiederholung des VL-Stoffs
 - “Rechnen“ von Aufgaben (Altklausuren)
 - Tipps nächstes Blatt

- Wann?: ab heute 14-tägig
- Wo?: Raum -119
- Was?:
 - Feedback letztes Blatt
 - Wiederholung des VL-Stoffs
 - “Rechnen“ von Aufgaben (Altklausuren)
 - Tipps nächstes Blatt
- Folien gibt's im Ilias und auf www.github.com/malluce/swt1-tut

- Wann?: ab heute 14-tägig
- Wo?: Raum -119
- Was?:
 - Feedback letztes Blatt
 - Wiederholung des VL-Stoffs
 - “Rechnen“ von Aufgaben (Altklausuren)
 - Tipps nächstes Blatt
- Folien gibt's im Ilias und auf www.github.com/malluce/swt1-tut
- Fragen stellen !!

cool

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

cool

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

!cool

- laut sein
- stören
- andere ablenken

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls
- 14-tägige Übungsblätter

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls
- 14-tägige Übungsblätter
- ab 50% der Punkte habt ihr sicher bestanden

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls
- 14-tägige Übungsblätter
- ab 50% der Punkte habt ihr sicher bestanden
- Besprechung der Musterlösung

- Bestehen des Scheins Voraussetzung zum Bestehen des Moduls
- 14-tägige Übungsblätter
- ab 50% der Punkte habt ihr sicher bestanden
- Besprechung der Musterlösung
- Abgaben
 - Theorieaufgaben im 3.Stock \implies Holzkasten
 - Programmieraufgaben auf <http://lez.ipd.kit.edu>

- Theorieaufgaben
 - handschriftlich
 - leserlich
 - Deckblatt (von Vorlage)

- Theorieaufgaben
 - handschriftlich
 - leserlich
 - Deckblatt (von Vorlage)
- Programmieraufgaben (Verstoß = Punktabzug)
 - Git
 - JavaDoc
 - CheckStyle
 - Stil (sinnvolle Namen, Kommentare etc.)

- Theorieaufgaben
 - handschriftlich
 - leserlich
 - Deckblatt (von Vorlage)
- Programmieraufgaben (Verstoß = Punktabzug)
 - Git
 - JavaDoc
 - CheckStyle
 - Stil (sinnvolle Namen, Kommentare etc.)
- keine Abgabe per Mail

- Theorieaufgaben
 - handschriftlich
 - leserlich
 - Deckblatt (von Vorlage)
- Programmieraufgaben (Verstoß = Punktabzug)
 - Git
 - JavaDoc
 - CheckStyle
 - Stil (sinnvolle Namen, Kommentare etc.)
- keine Abgabe per Mail
- harte Deadlines

- Theorieaufgaben
 - handschriftlich
 - leserlich
 - Deckblatt (von Vorlage)
- Programmieraufgaben (Verstoß = Punktabzug)
 - Git
 - JavaDoc
 - CheckStyle
 - Stil (sinnvolle Namen, Kommentare etc.)
- keine Abgabe per Mail
- harte Deadlines
- Plagiate teuer

- Theorieaufgaben
 - handschriftlich
 - leserlich
 - Deckblatt (von Vorlage)
- Programmieraufgaben (Verstoß = Punktabzug)
 - Git
 - JavaDoc
 - CheckStyle
 - Stil (sinnvolle Namen, Kommentare etc.)
- keine Abgabe per Mail
- harte Deadlines
- Plagiate teuer
- keine Punkte geschenkt \implies früh anfangen

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 \Rightarrow SWT1 \Rightarrow PSE

- Programmieren \implies SWT1 \implies PSE
- den Hacker strukturieren

- Programmieren \implies SWT1 \implies PSE
- den Hacker strukturieren
- Tools (Versionsverwaltung, Build-Management) erlernen

Was ihr bisher getan haben solltet..

Installation von:

- Eclipse (incl. CheckStyle und EclEmma)

Was ihr bisher getan haben solltet..

Installation von:

- Eclipse (incl. CheckStyle und EcJemma)

Überblick über:

- Maven
- Git

Was ihr bisher getan haben solltet..

Installation von:

- Eclipse (incl. CheckStyle und EcJemma)

Überblick über:

- Maven
- Git

Anmelden in der LEZ!

Was ihr bisher getan haben solltet..

Installation von:

- Eclipse (incl. CheckStyle und EcLEmma)

Überblick über:

- Maven
- Git

Anmelden in der LEZ!

Meine Empfehlung

- Installiert Git manuell!

Was ihr bisher getan haben solltet..

Installation von:

- Eclipse (incl. CheckStyle und EcLEmma)

Überblick über:

- Maven
- Git

Anmelden in der LEZ!

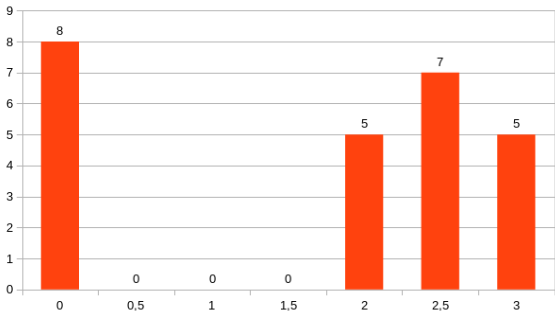
Meine Empfehlung

- Installiert Git manuell!

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

- Java 9 nicht mehr unterstützt
- \implies Java 10 verwenden

- Java 9 nicht mehr unterstützt
- \implies Java 10 verwenden
- LEZ forciert Verwendung von Java 9
- man kann aber auch mit Java 10 für Java 9 kompilieren



Kriterien für Punkte

je 1P.:

- Import + Abgabe (pom.xml muss stimmen)
- CheckStyle (+ **sinnvolles** JavaDoc)
- Implementierung (EditMe + EditMeTest)

Achtet zukünftig besonders auf:

- **sinnvolles** JavaDoc (siehe nächste Folie)
- alte Kommentare (TODOs...) entfernen
- nicht “throws Exception“ angewöhnen
- tut etwas in tearDown() \implies Objekte nullen...

```
package tuts.swt1;

/**
 * This class demonstrates how to use JavaDoc.
 * @author Felix Bachmann
 * @version 1.0
 */
public class JavaDocDemonstration {

    /**
     * Returns a random number in a specific range.
     * @param start the start of the range
     * @param end the end of the range
     * @return a random number in between start and end
     * @throws IllegalArgumentException is thrown if start is 1337
     */
    public int getRandomNumber(int start, int end) throws IllegalArgumentException {
        int random = 0;
        if (start == 1337) {
            throw new IllegalArgumentException("sorry, no leet numbers");
        }
        // very nice calculation
        return random;
    }
}
```

- 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)

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

- @BeforeClass (wird als erstes einmal ausgeführt)
- @Before (wird vor jeder Test-Methode 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)

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)

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:

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

Zu testende Methode in der Klasse Hallo

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

Zu testende Methode in der Klasse Hallo

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

Wie sieht Testmethode aus?

Zu testende Methode in der Klasse Hallo

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

Wie sieht Testmethode aus?

Testmethode in der Klasse HalloTest

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

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

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

B (Ignore = Methode nicht ausführen, AfterClass = nach Ausführung einer Tests einmal annotierte Methode)

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

B (Ignore = Methode nicht ausführen, AfterClass = nach Ausführung einer Tests einmal annotierte Methode)

Wahr oder falsch?

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

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

B (Ignore = Methode nicht ausführen, AfterClass = nach Ausführung einer Tests einmal annotierte Methode)

Wahr oder falsch?

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

Falsch, "zufällig"

Wahr oder falsch?

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

Wahr oder falsch?

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

Falsch (deprecated, `org.junit.Assert` benutzen!)

- Build-Management-Tool (Automatisierung von möglichst vielen Schritten)

- Build-Management-Tool (Automatisierung von möglichst vielen Schritten)
- Maven ist in jeder Eclipse-Installation integriert
⇒ keine manuelle Installation nötig

- 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)

- 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

- 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

- 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

- 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

- 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
- Goals = Maven-Befehle (z.B. mvn package)
 - Ausführung eines Goals führt ggf. vorandgehende Goals aus
 - <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

Verzeichnisstruktur:

- src
 - main
 - java
 - resources

Verzeichnisstruktur:

- src
 - main
 - java
 - resources
 - test
 - java
 - resources

Verzeichnisstruktur:

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

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?

- 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?

- 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?

- 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

```
mvn compile
```

 Quelltexte \Rightarrow .class-Dateien

Wichtige Befehle

<code>mvn compile</code>	Quelltexte \implies .class-Dateien
<code>mvn test</code>	Test-Quelldateien \implies .class-Dateien, führt Tests aus und zeigt Ergebnisse an

Wichtige Befehle

<code>mvn compile</code>	Quelltexte \implies .class-Dateien
<code>mvn test</code>	Test-Quelldateien \implies .class-Dateien, führt Tests aus und zeigt Ergebnisse an
<code>mvn package</code>	verpackt Projekt in eine Datei (.war/.jar/.zip)

Wichtige Befehle

<code>mvn compile</code>	Quelltexte \implies .class-Dateien
<code>mvn test</code>	Test-Quelldateien \implies .class-Dateien, führt Tests aus und zeigt Ergebnisse an
<code>mvn package</code>	verpackt Projekt in eine Datei (.war/.jar/.zip)
<code>mvn install</code>	installiert Projekt lokal

Wichtige Befehle

<code>mvn compile</code>	Quelltexte \implies .class-Dateien
<code>mvn test</code>	Test-Quelldateien \implies .class-Dateien, führt Tests aus und zeigt Ergebnisse an
<code>mvn package</code>	verpackt Projekt in eine Datei (.war/.jar/.zip)
<code>mvn install</code>	installiert Projekt lokal
<code>mvn deploy</code>	liefert Projekt (remote) aus

wasserfallartige Ausführung! (Tafel)

Lösungsansätze:

- Rechtsklick auf Projekt \Rightarrow Maven \Rightarrow Update Maven Project
 \Rightarrow Haken bei “Force Update...”
 - Synchronisiert pom.xml mit Projekt, aktualisiert Abhängigkeiten

Lösungsansätze:

- Rechtsklick auf Projekt \Rightarrow Maven \Rightarrow Update Maven Project
 \Rightarrow Haken bei “Force Update...”
 - Synchronisiert pom.xml mit Projekt, aktualisiert Abhängigkeiten
- mvn clean
 - vielleicht war der target-Ordner verschmutzt

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)

A, B, C oder D?

Welcher Maven-Befehl kompiliert die Testklassen?

- A: mvn compile
- B: mvn package
- C: mvn test
- D: mvn test-compile

A, B, C oder D?

Welcher Maven-Befehl kompiliert die Testklassen?

- A: mvn compile
- B: mvn package
- C: mvn test
- D: mvn test-compile

B,C,D. A kompiliert nur src-Quelltexte

Wahr oder falsch?

Damit Maven funktioniert, muss die komplette pom.xml manuell ausgefüllt werden.

A, B, C oder D?

Welcher Maven-Befehl kompiliert die Testklassen?

- A: mvn compile
- B: mvn package
- C: mvn test
- D: mvn test-compile














B,C,D. A kompiliert nur src-Quelltexte

Wahr oder falsch?

Damit Maven funktioniert, muss die komplette pom.xml manuell ausgefüllt werden.

Falsch, default-Werte!

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

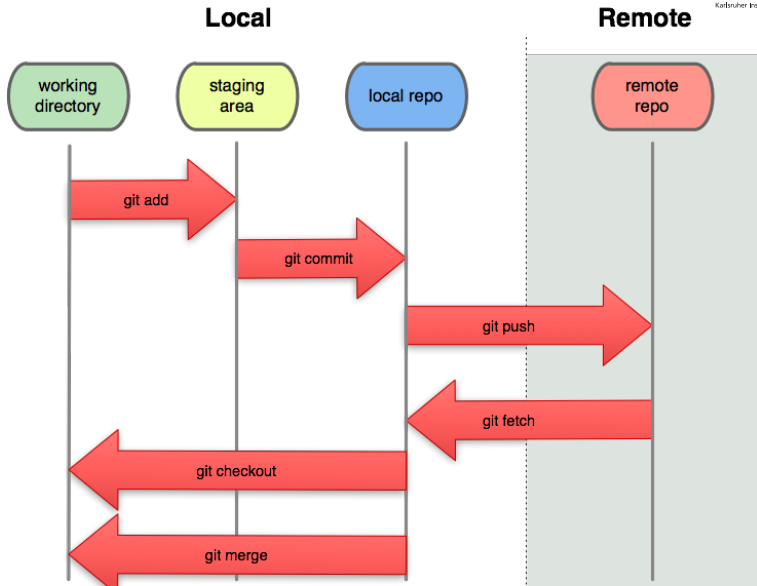
Hacks

- 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.
<code>git branch hallo</code>	Erzeugt einen neuen Branch namens hallo.
<code>git merge hallo</code>	Merged den Branch hallo in den aktuellen.

Git - Workflow



- 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

- sinnvolle Commit-Nachrichten
- Dateien ggf. sinnvoll zu Commits zusammenfassen (mit git add)
- Übung: pro Teilaufgabe o.Ä. ein commit

Richtig oder falsch?

Mit `git commit "message"` wird ein neuer Commit erzeugt, dessen Commit-Nachricht `message` ist.

Richtig oder falsch?

Mit `git commit "message"` wird ein neuer Commit erzeugt, dessen Commit-Nachricht `message` ist.

Falsch, es fehlt die Option `-m`

Richtig oder falsch?

Git ist im Gegensatz zu SVN ein zentrales Versionsverwaltungssystem.

Richtig oder falsch?

Mit `git commit "message"` wird ein neuer Commit erzeugt, dessen Commit-Nachricht `message` ist.

Falsch, es fehlt die Option `-m`

Richtig oder falsch?

Git ist im Gegensatz zu SVN ein zentrales Versionsverwaltungssystem.

Falsch, andersrum.

Richtig oder falsch?

`git log` zeigt eine Liste aller bisher getätigten Commits an und zeigt dabei Informationen wie Datum, Zeit, Hashcode und Commitnachricht der jeweiligen Commits an.

Richtig oder falsch?

Mit `git commit "message"` wird ein neuer Commit erzeugt, dessen Commit-Nachricht `message` ist.

Falsch, es fehlt die Option `-m`

Richtig oder falsch?

Git ist im Gegensatz zu SVN ein zentrales Versionsverwaltungssystem.

Falsch, andersrum.

Richtig oder falsch?

`git log` zeigt eine Liste aller bisher getätigten Commits an und zeigt dabei Informationen wie Datum, Zeit, Hashcode und Commitnachricht der jeweiligen Commits an.

Richtig.

- Eclipse von z.B. Neon zu Oxygen upgraden (für Java 9/10 Support)
 - https://wiki.eclipse.org/FAQ_How_do_I_upgrade_Eclipse_IDE%3F
- Hilfe zu Mockito
 - <https://www.javacodegeeks.com/2012/05/mocks-and-stubs-understanding-test.html>

Aufgabe 1: Altsoftware vorbereiten

- löchriges Kochrezept für Umgang mit Maven, Git, Checkstyle
- bei Fehlern Google + Forum benutzen
- an Maven-Ordnerstruktur erinnern

Aufgabe 1: Altsoftware vorbereiten

- löchriges Kochrezept für Umgang mit Maven, Git, Checkstyle
- bei Fehlern Google + Forum benutzen
- an Maven-Ordnerstruktur erinnern

Aufgabe 2: Modultests

- Aufgaben zum Testen mit JUnit4
- Ordner sollen erstellt werden, wenn sie nicht existieren
- Tests ohne Erwartung sind keine Tests (Asserts oder @expect benutzen)

Aufgabe 3: Testüberdeckung

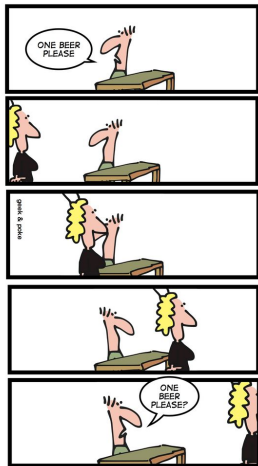
- Testüberdeckung mit EcJemma überprüfen
- Mockito klingt komplizierter als es ist (siehe Link, Abschnitt Mock Objekt)

Abgabe

- in der LEZ bis zum 02.05, 12:00
- falls ihr ein Feedback wollt, werft das Deckblatt ein

Bis dann! (dann:=08.05.18)

SIMPLY EXPLAINED



.gitignore

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