hochschule mannheim

Fakultät für Informatik

Programmierung 2 (PR2)

03 – Testen und Teamarbeit

Prof. Dr. Frank Dopatka

Hochschule Mannheim University of Applied Sciences

hochschule mannheim

Kommentare, JavaDoc und guter Code

Arten von Kommentaren

```
// Dies ist ein Kommentar in einer Zeile

/* dies ist ein Kommentar
   über mehrere Zeilen (Blockkommentar) */

/** Hier kommt JavaDoc */

/** Hier kommt JavaDoc,
 * gerne auch Mehrzeilig */

/* Das geht nicht, da
 /* Mehrzeilige Kommentare nicht geschachtelt werden können
 */
*/
```


Arten von Kommentaren

- Innerhalb von Methoden immer nur Zeilenkommentare verwenden:
  - So können Methoden einfach durch einen Blockkommentar auskommentiert werden!

```
void methodeMitKommentaren() {
  // Erzeuge zwei Strings mit sinnvollem Inhalt
  String a = "Hallo";
  String b = "Welt";
  // Vertausche die Variablen
  swap(a, b);
  // Gebe das Ergebnis aus
  System.out.println(a + " " + b);
}
```

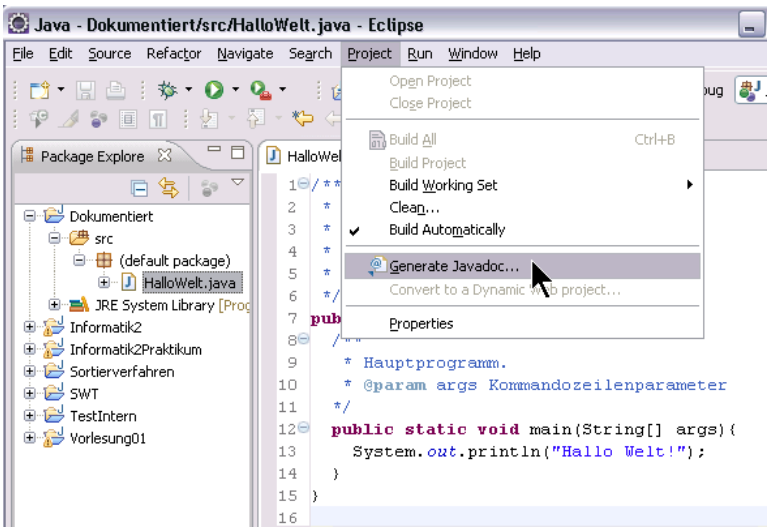
- JavaDoc ist ein Software-Dokumentationswerkzeug, das aus Java-Quelltextkommentaren automatisch HTML-Dateien erstellt.
- Dabei kommen Tags zum Einsatz, die dazu dienen z.B. Interfaces, Klassen, Methoden und Eigenschaften näher zu beschreiben.
- Neben der Standardausgabe in HTML sind alternative Ausgaben durch spezielle Doclets möglich.

Schlüsselwort	Ausgabe
@author name	Beschreibt den Autor einer Klasse oder eines Interfaces.
@version version	Erzeugt einen Versionseintrag; max. einmal pro Klasse oder Interface.
@since version	Seit wann die Funktionalität einer Klasse, Interface, Instanzvariable, Methode existiert.
@param name beschr.	Parameterbeschreibung einer Methode.
@return beschr.	Beschreibung des Returnwerts einer Methode.
@exception classname beschr. @throws classname beschr.	Beschreibung einer Exception, die von der Methode geworfen werden kann.
@deprecated beschreibung	Beschreibt eine veraltete Methode, die nicht mehr verwendet werden sollte.

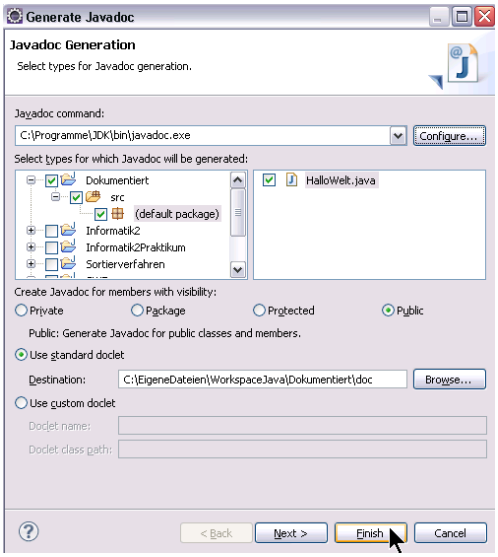
Ein JavaDoc-Beispiel

```
/**
 * Ein Hallo-Welt Programm in Java.
 * Dies ist ein JavaDoc-Kommentar.
 * @author Frank Dopatka
 * @version 1.0
 */
public class HalloWelt{
  /**
   * Hauptprogramm.
   * @param args Kommandozeilenparameter
  */
  public static void main(String[] args){
    System.out.println("Hallo Welt!");
  }
}
```

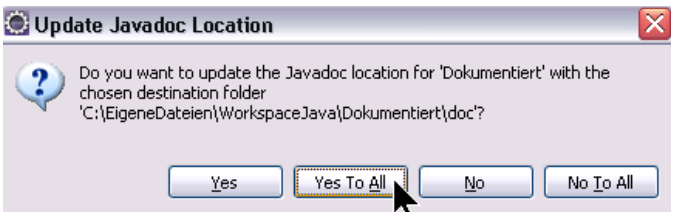

JavaDoc mit Eclipse erzeugen

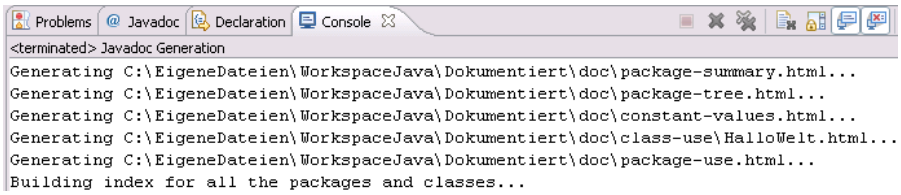


JavaDoc mit Eclipse erzeugen



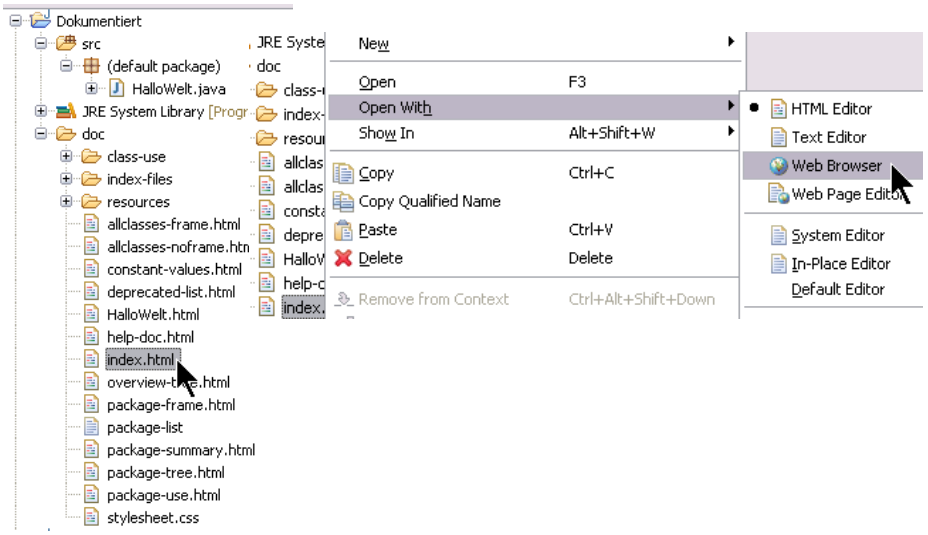
JavaDoc mit Eclipse erzeugen





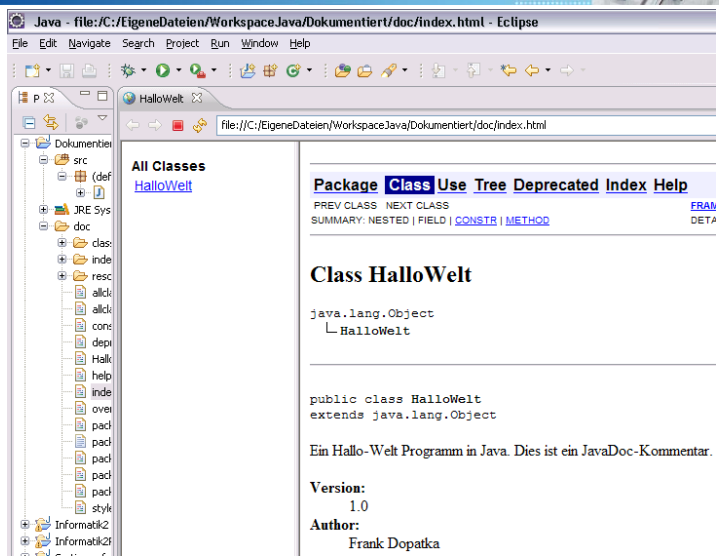
hochschule mannheim

JavaDoc mit Eclipse erzeugen



hochschule mannheim

JavaDoc mit Eclipse erzeugen



Constructor Summary

`HalloWelt()`

Method Summary

`static void`	`main(java.lang.String[] args)` Hauptprogramm.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

HalloWelt

```
public HalloWelt()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Hauptprogramm.

Parameters:

`args` - Kommandozeilenparameter

Java-Dokumentation der JavaDoc: In Eclipse integrierbar

http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html

https://www.tutorials.de/threads/javadoc-unter-eclipse-anzeigen-lassen.238377/

Java SE Development Kit 8 Documentation

Java SE Development Kit 8u131 Documentation

You must accept the Java SE Development Kit 8 Documentation License Agreement to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Documentation	89.1 MB	jdk-8u131-docs-all.zip

Java-Dokumentation der Klasse String

https://docs.oracle.com/javase/8/docs/api/

java.lang

Class String

java.lang.Object
    java.lang.String

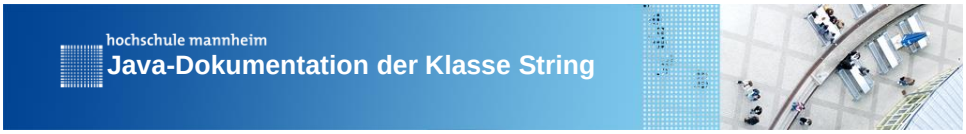
All Implemented Interfaces:

`Serializable, CharSequence, Comparable<String>`

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

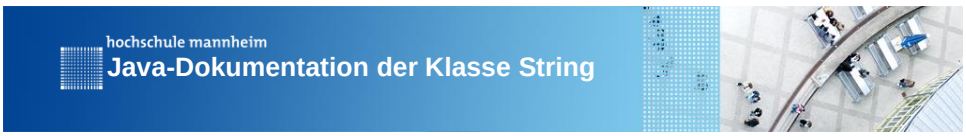
Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:



https://docs.oracle.com/javase/8/docs/api/

Constructors

Constructor and Description
`String()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.



https://docs.oracle.com/javase/8/docs/api/

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods** | **Deprecated Methods**

Modifier and Type	Method and Description
`char`	`charAt(int index)` Returns the `char` value at the specified index.
`int`	`codePointAt(int index)` Returns the character (Unicode code point) at the specified index.
`int`	`codePointBefore(int index)` Returns the character (Unicode code point) before the specified index.
`int`	`codePointCount(int beginIndex, int endIndex)` Returns the number of Unicode code points in the specified text range of this `String`.

hochschule mannheim
Java-Dokumentation der Klasse String

https://docs.oracle.com/javase/8/docs/api/

charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the char value.

Returns:

the char value at the specified index of this string. The first char value is at index 0.

hochschule mannheim
Motivation für guten Code

- Nicht der Compiler ist das Publikum, sondern Menschen, die den Quellcode lesen müssen!
- Probleme mit schlechtem Programmierstil:
  - Die Teamarbeit wird durch den inkonsistenten Stil erschwert.
  - Programme sind fehleranfälliger.
  - Änderungen am Programm werden erschwert, schlechte Wartbarkeit.
  - Lesen des Quellcode ist eine Qual.
  - Punktabzüge bei den Pflichtübungen.
- Einige Zitate:
  - Talk is cheap, show me the code! [Linus Torwalds]
  - Don‘t comment bad code – rewrite it! [Brian W. Kernighan]

hochschule mannheim

Einige Grundregeln für guten Code

- Einrückung pro Ebene: 2 Tabs!
- Zeilenlänge maximal 80 Zeichen!
- Klassennamen beginnen mit Großbuchstaben!
- Methodennamen beginnen mit Kleinbuchstaben!
- Variablennamen beginnen mit Kleinbuchstaben!
- Konstanten werden groß geschrieben!
- Pro Zeile genau eine Deklaration bzw. genau ein Statement!

hochschule mannheim

Ein Beispiel für guten Code

```java
/**
* Diese Klasse dient als Beispiel für den Coding-Standard und die
* Namenskonventionen bei Java-Programmen.
* @author Thomas Smits
*/
public class CodingStandard {
  /**
  * Konstante, die dem Rest der Welt etwas mitteilen soll.
  */
  public static final int KONSTANTE_MIT_TOLLEM_WERT = 3;
  private int erstesFeld;
  private double zweitesFeld;
```

Ein Beispiel für guten Code

```
  /**
   * Methode, die etwas tut - oder nicht ;-).
   *
   * @param parameter Eingabewert für die Methode
   * @return gibt immer 42 zurück
   */
  public int methodeDieWasTut(int parameter) {
    if (parameter > 3) {
      erstesFeld = 12;
    }
    else {
      zweitesFeld = 13;
    }
    return 42;
  }
}
```


Weitere Regeln für guten Code

- Minimale Sichtbarkeit von Methoden und Feldern:
  Nur **`public`** wenn wirklich nötig!
- Verbergen Sie die Implementierungsdetails: Information Hiding
- Dokumentieren Sie alle öffentlichen Klassen und Methoden mit JavaDoc.
- Verwenden Sie keine Quellcode-Fragmente mehrfach:
  Kapseln Sie diese statt dessen in Methoden.
- Lesen Sie keine Daten von der Konsole ein.
- Erzeugen Sie keine leeren **`catch`**-Blöcke.
- Testen Sie Ihre Klassen mit JUnit-Tests.

Weitere Regeln für guten Code

- Programmieren Sie mit so wenig Seiteneffekten wie möglich.
- Denken Sie genau nach, was Sie in Feldern und was Sie in lokalen Variablen speichern.
- Fangen Sie mögliche Fehler korrekt ab.
- Denken Sie vor allem an die Randfälle und vertrauen Sie niemals den Eingabedaten.
- Halten Sie die Daten und die Methoden zusammen.
- Vermeiden Sie komplexen und komplizierten Code.
- Kommentieren Sie keinen Code aus:
  Verwenden Sie statt dessen ein Versionsverwaltungssystem, um alte Stände vorzuhalten.
- Lassen Sie keine `// TODO-Kommentare` im Quellcode stehen

Professionell Testen

hochschule mannheim

Was sind Quellcode-Tests?

- Ziel ist es, die Funktionalität eines Programms an den Anforderungen zu messen & Softwarefehler zu ermitteln.
- Dynamische Tests haben das Aufdecken von Fehlern zum Ziel.
  - Tschernobyl war ein dynamischer Test zur Laufzeit:
    http://www.spiegel.de/wissenschaft/technik/tschernobyl-der-super-gau-im-protokoll-a-1089220.html
  - Ein dynamischer Test findet meist nur einen Effekt und nicht die eigentliche Ursache!
  - Ein erfolgloser dynamischer Test ist niemals ein Beweis für ein korrektes Programm!
- Statische Tests haben den Beweis der Abwesenheit von Fehlern zum Ziel.

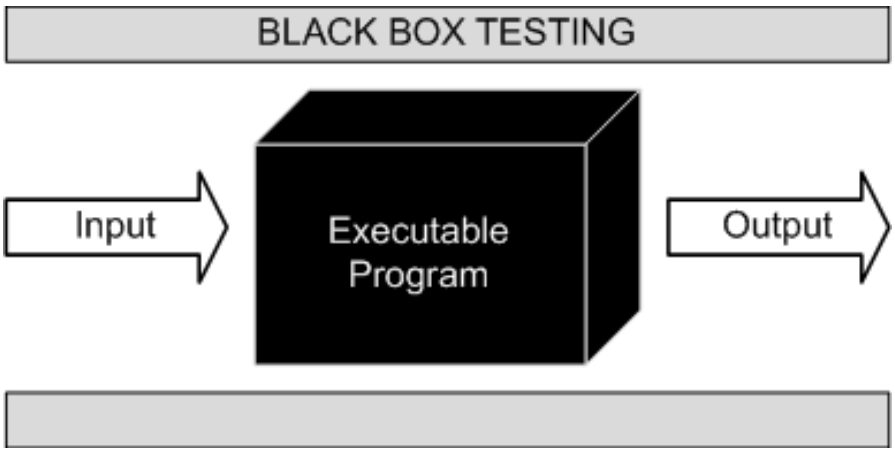
hochschule mannheim

Was sind Black-Box Tests?

- Black-Box Tests bezeichnen eine Methode des Software-Tests, bei der die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden.
- Die Tests beschränken sich auf funktionsorientiertes Testen, d. h. für die Ermittlung der Testfälle werden nur die Anforderungen, aber nicht die Implementierung des Testobjekts herangezogen.
- Die innere Beschaffenheit des Programms wird nicht betrachtet.
  Statt dessen wird das Programm als Black Box behandelt.
- Nur nach außen sichtbares Verhalten fließt in den Test ein.
- Black-Box Tests sollten nicht von den Entwicklern des Programms selbst geschrieben werden, da diese (unterbewußt) dazu neigen, um Fehler „herumzutesten“.

hochschule mannheim

Was sind Black-Box Tests?



hochschule mannheim

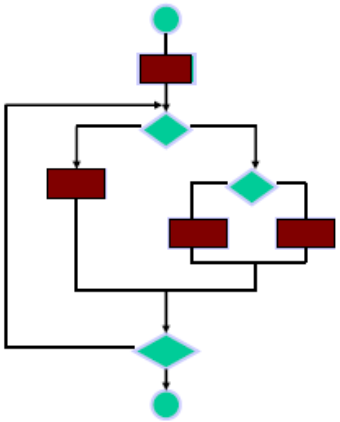
Black-Box Testen: Vorgehensweise & Ziele

- Ziel ist es, die Übereinstimmung eines Softwaresystems mit seiner Spezifikation zu überprüfen.
- Ausgehend von formalen oder informalen Spezifikationen werden Testfälle erarbeitet, die bei erfolgreicher Durchführung Indizien dafür liefern, dass der geforderte Funktionsumfang eingehalten wird.
- Auch ist ein erfolgreicher Black-Box-Test keine Garantie für die Fehlerfreiheit der Software, da in frühen Phasen des Software-Entwurfs erstellte Spezifikationen spätere Implementationsentscheidungen nicht abdecken.

hochschule mannheim

Was sind White-Box Tests?

- Statische White-Box Tests werden mit Kenntnissen über die innere Funktionsweise des zu testenden Systems entwickelt
- Anweisungsüberdeckungstests sind die am einfachsten anwendbaren steuerungsfluss-orientierten Testmethoden.
- Damit kann „toter Code“ gefunden werden. Dies sind Anweisungen, die niemals durchlaufen werden.
- White-Box Tests leiten Testfälle nicht aus der Spezifikation des Programms her, sondern aus der Implementation selbst



hochschule mannheim

Was ist testgetriebene Entwicklung?

- Testgetriebene Entwicklung (auch: test-driven development TDD) ist eine Methode, bei der der Programmierer Software-Tests konsequent vor den zu testenden Komponenten erstellt.
- Die dazu erstellten Unit-Tests sind „Grey-Box-Tests“.
- Bei der testgetriebenen Entwicklung ist zu unterscheiden zwischen dem
  - Testen im Kleinen (Unit-Tests) und dem
  - Testen im Großen (Systemtests, Akzeptanztests).
- Testgetriebene Entwicklung ist eine Methode, die häufig bei der agilen Entwicklung eingesetzt wird.

Was sind Grey-Box Tests?

- Gemeinsamkeit mit White-Box Tests:
  - Von den gleichen Entwicklern wie das zu testende System geschrieben
  - Diese haben eine „Ahnung“ über mögliche kritischen Pfade in dem noch zu erstellenden Quellcode.
- Gemeinsamkeit mit Black-Box Tests:
  - Anfänglich die Unkenntnis über die Interna des zu testenden Systems, weil der Grey-Box Test vor dem zu testenden System geschrieben wird (Test-First).

- Grey-Box Tests versuchen, die Vorteile von Black-Box & White-Box Tests zu verbinden.
- Dadurch werden Komponenten mit dem geringen organisatorischen Aufwand der White-Box Tests geprüft, ohne „um Fehler herum“ zu testen

Was sind Unit-Tests?

- Ein Programm ist in einzelne Teile mit klar definierten Klassen unterteilt.
- Der Unit-Test ist der Software-Test dieser einzelnen Programmteile, die zu einem späteren Zeitpunkt zusammengefügt werden.
  - Das Gesamtsystem wäre dann in einem Integrationstest zu testen.
- Ziel des Unit-Tests ist es, frühzeitig Programmfehler in den Methoden einzelner Klassen zu finden.
- Die Unit-Tests werden automatisiert durchgeführt.
- Es besteht die Möglichkeit, nach jeder Programmänderung durch einen weiteren Ablauf aller Unit-Tests nach Programmfehlern zu suchen.
- Die Automatisierung setzt voraus, dass die Unit-Tests vor Programm-änderungen 100%ig durchlaufen, also Entwickler nur dann ihren Sourcecode in eine Versionsverwaltung wie GIT einchecken, wenn alle Unit-Tests fehlerfrei durchlaufen.

Vorgehensweise bei der testgetriebenen Entwicklung

- Die Tests und die getesteten Units werden gemeinsam entwickelt.
- Die Programmierung erfolgt in kleinen und wiederholten Mikro-Iterationen iterativ-inkrementell.
- Die Mikro-Iterationen werden so lange wiederholt, bis die gewünschte Funktionalität erreicht ist und dem Entwickler keine sinnvollen weiteren Tests mehr einfallen, die vielleicht Fehler aufdecken könnten.
- Die so behandelte programmtechnische Einheit (Unit) wird dann als „vorerst fertig“ angesehen.
- Die gemeinsam mit ihr geschaffenen Tests bleiben erhalten, um auch zukünftige Umsetzungen daraufhin testen zu können, ob das erwünschte Verhalten fortbesteht.
- Die konsequente Befolgung dieser Vorgehensweise läuft auf evolutionären Entwurf hinaus, weil die ständige Änderung die Weiterentwicklung des Systems bestimmt.

hochschule mannheim

Vorgehensweise bei der testgetriebenen Entwicklung

- Eine Iteration, die nur wenige Minuten dauern sollte, hat drei Hauptteile:
  1. Schreibe einen Test für das erwünschte fehlerfreie Verhalten. Dieser Test wird vom bestehenden Quellcode erst einmal nicht erfüllt oder es gibt diesen Quellcode noch gar nicht.
     - Das „erwünschte fehlerfreie Verhalten“ wird im Test spezifiziert!
     - Der Test ist die Spezifikation!
  2. Ändere/schreibe den Programmcode mit möglichst wenig Aufwand, bis nach dem anschließend angestoßenen Testdurchlauf alle Tests fehlerfrei ablaufen.
  3. Räume dann im Code auf (Refactoring): Entferne Code-Duplizierung, abstrahiere wo nötig, richte den Code nach den Konventionen aus etc.
     - Natürlich wieder mit abschließendem Testen.
     - Ziel des Aufräumens ist es, den Code schlicht und verständlich zu halten.

hochschule mannheim

Beispiel-Testfälle einer Seminarverwaltung

Test:
Seminar anlegen

Vorbedingung:
---
prüfbare Nachbedingung Erfolg:
Seminar wurde mit seinen Daten in der Seminarverwaltung aufgenommen und eine neue Seminar-ID wurde vergeben.
Nachbedingung erwarteter Fehlschlag:
Seminar mit diesem Titel existiert bereits.

Test:
Seminar buchen

Vorbedingung:
Seminar existiert bereits und ist zum gewählten Termin nicht ausgebucht
prüfbare Nachbedingung Erfolg:
Kunde wurde als TN in die Liste der TN zum gewählten Termin aufgenommen
Nachbedingung erwarteter Fehlschlag:
Meldung „Nicht möglich: Kunde hat noch >3 offene Rechnungen!“

Test:
Anmeldung auf ein ausgebuchtes Seminar

Vorbedingung:
Seminar ist ausgebucht
prüfbare Nachbedingung Erfolg:
Kann nicht erfolgreich sein; Fehler erwartet.
Nachbedingung erwarteter Fehlschlag:
Meldung „Seminar bereits ausgebucht“; auf nächsten freien Termin verweisen

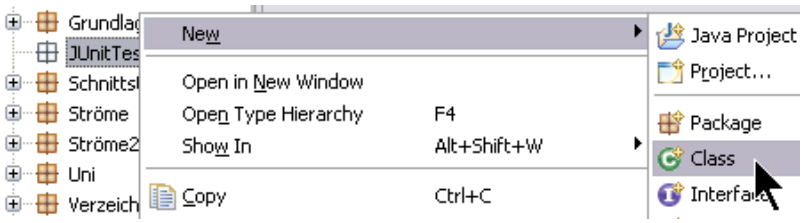
hochschule mannheim

Was ist JUnit?

- JUnit ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests geeignet ist.
- Es ist in Eclipse EE bereits integriert.
- Ein Test kennt nur zwei Ergebnisse:
  - Entweder der Test gelingt (dann ist er „grün“) oder
  - er misslingt (dann ist er „rot“).
- Misslingen kann als Ursache einen Fehler (Error) oder ein falsches Ergebnis (Failure) haben, die beide per Exception signalisiert werden.
- Während Failures erwartet sind, treten Errors unerwartet auf.
- Failures werden mittels einer speziellen Exception „`AssertionFailedError` signalisiert.
- Alle übrigen Exceptions werden als Error interpretiert.

hochschule mannheim

JUnit in Eclipse: Zu testende Klasse erstellen

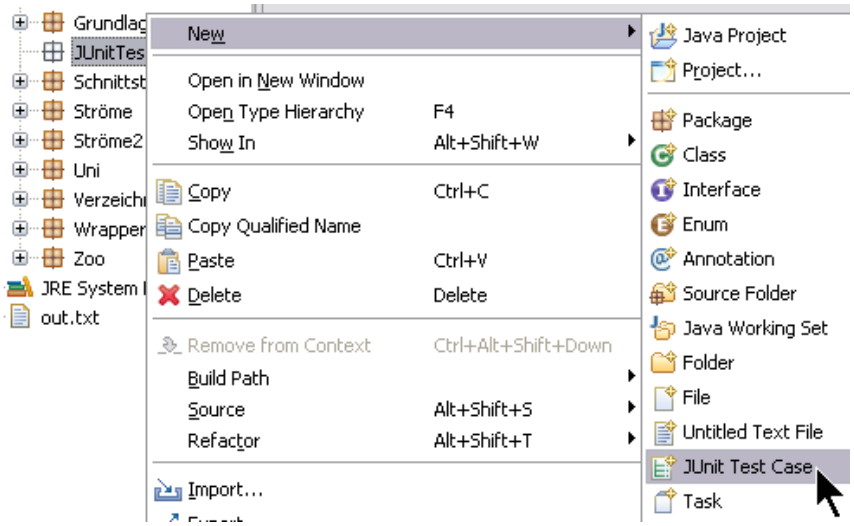



```
public class Mathe {

}
```

ACHTUNG:
Die zu testende Klasse ist nur ein Beispiel zu JUnit.
Ihre erstellten Methoden sind keine guten Beispiele für Objektorientierung!

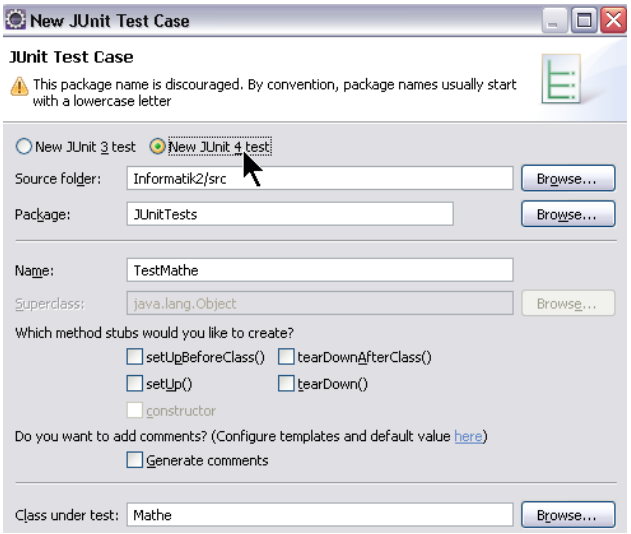
hochschule mannheim

JUnit in Eclipse: Test-Klasse erstellen



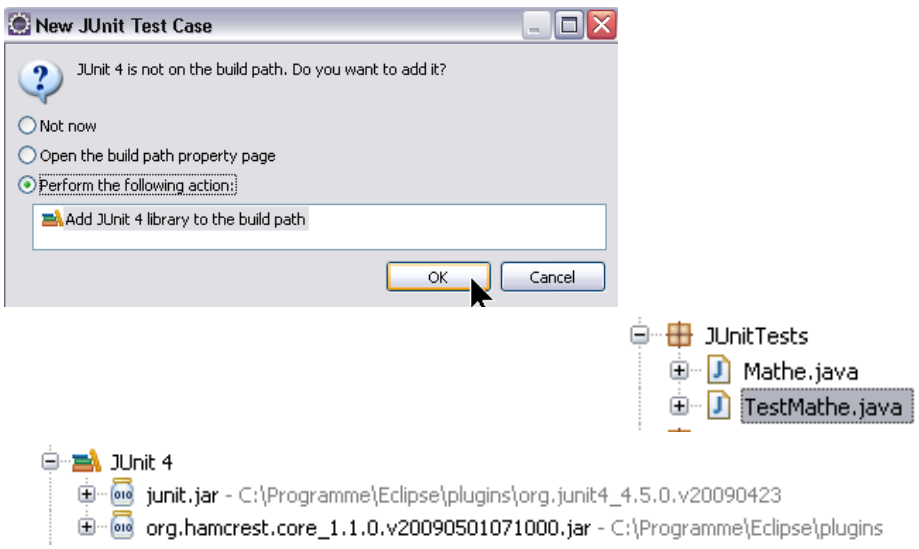
hochschule mannheim

JUnit in Eclipse: Test-Klasse erstellen



hochschule mannheim

JUnit in Eclipse: Test-Klasse erstellen



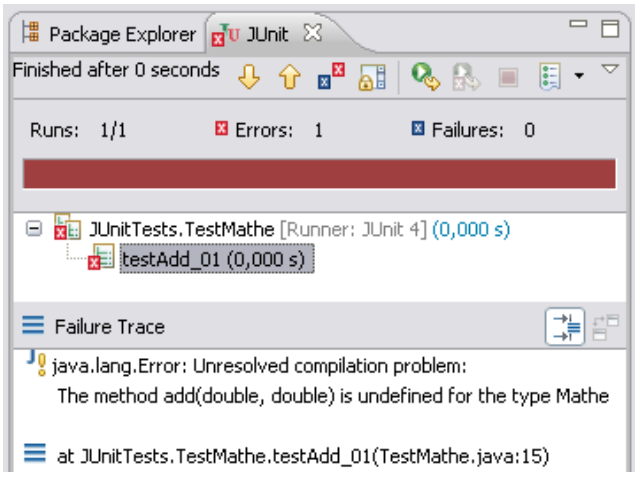
hochschule mannheim

JUnit in Eclipse: 1. Test schreiben

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class TestMathe {
  protected Mathe m;

  @Before
  public void vorMethode() {
   System.out.println("@Before");
   m=new Mathe();
  }

 @Test
  public void testAdd_01() {
    double erg=m.add(3.0,5.3);
    assertTrue(erg==8.3);
 }
}
```


hochschule mannheim

JUnit in Eclipse: 1b. Test ausführen & Fehlschlag



JUnit in Eclipse:
2. Code schreiben für erfolgreichen Test

```
@Test
 public void testAdd_01() {
   double erg=m.add(3.0,5.3);
   assertTrue(er
 }
}
```

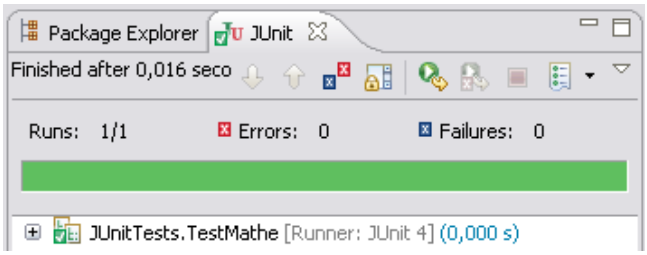
The method add(double, double) is undefined for the type Mathe

2 quick fixes available:

- Create method 'add(double, double)' in type 'Mathe'
- Add cast to 'm'

```
public class Mathe {
  public double add(double x, double y) {
    return x+y;
  }
}
```


JUnit in Eclipse:
2b+3. Erneut testen & Refactoring



3. Refactoring

der Klasse **`Mathe.java`** ist noch nicht notwendig.

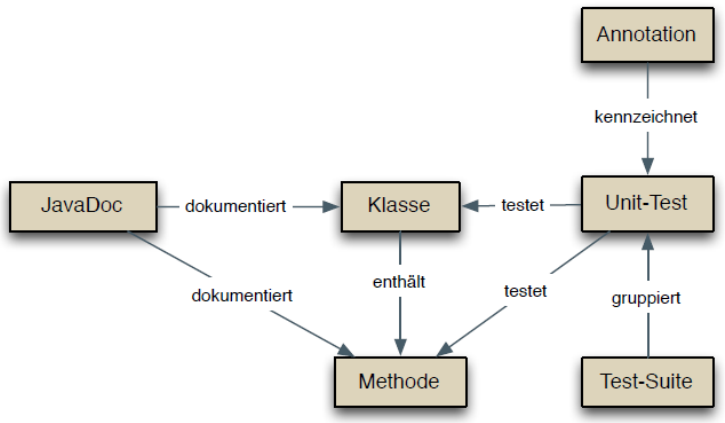
Annotationen für die Testumgebung

- **`@BeforeClass`** wird einmalig ausgeführt bevor der erste Test startet.
- **`@Before`** wird vor jedem Test ausgeführt.
- **`@Ignore`** wird eine Zeile vor **`@Test`** gesetzt, wenn der folgende Test ignoriert werden soll.
- **`@Test`** ist einer der Tests.
- **`@Test(expected=Exception.class)`** erwartet eine Exception als Ergebnis des korrekten Tests; auch beliebige Unterklassen von Exception sind möglich. Auch fehlerhafte Eingaben sollten Sie testen!
- **`@Test(timeout=100)`** lässt den Test scheitern, wenn seine Ausführung länger als 100ms dauert; dies ist sinnvoll zum Testen von Reaktions-Zeiten, z.B. in Verbindung mit einer Datenbank im Backend.
- **`@After`** wird nach jedem Test ausgeführt.
- **`@AfterClass`** wird einmalig ausgeführt nachdem der letzte Test beendet ist.

Verschiedene Arten von Assertions

- **`assertEquals`** stellt sicher, dass zwei Objekte/Daten **`equals`** sind.
- **`assertArrayEquals`** stellt sicher, dass zwei Datenfelder **`equals`** sind.
- **`assertFalse`** stellt sicher, dass ein boolscher Wert **`false`** ergibt.
- **`assertTrue`** stellt sicher, dass ein boolscher Wert **`true`** ergibt.
- **`assertNull`** stellt sicher, dass eine Objekt-Referenz **`null`** ist.
- **`assertNotNull`** stellt sicher, dass eine Objekt-Referenz nicht **`null`** ist.
- **`assertSame`** stellt sicher, dass zwei Objekt-Referenzen identisch sind (==).
- **`assertNotSame`** stellt sicher, dass zwei Objekt-Referenzen nicht identisch sind (!=).

Begriffsübersicht



Tools zur Teamarbeit

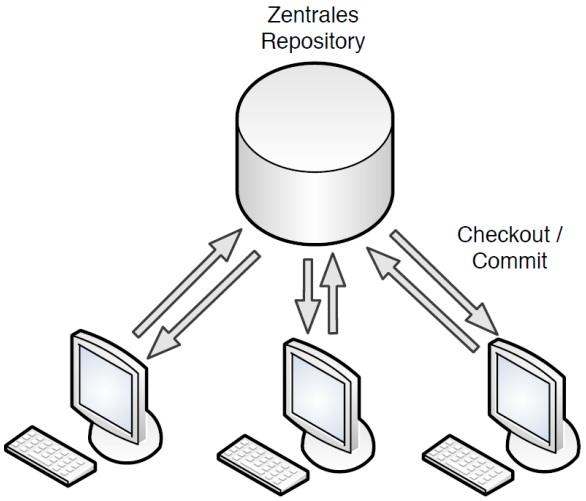
Das Problem in den Übungsgruppen...

- Wie kann ich ...
  - mit mehreren Leuten an einem Projekt entwickeln?
  - die Historie der Änderungen der Quellen behalten?
  - in der Zeit zurückreisen und alte Stände wieder herstellen?
  - herausfinden, welcher Entwickler ein Problem ausgelöst hat?
  - verschiedene Versionen der Software entwickeln?
  - meine Code ablegen und konsistent per Backup sichern?
  - Änderungen aus verschiedenen Quellen zusammenführen?
- Welche Probleme lösen Filesharing-Dienste wie Dropbox, Google Drive und OneDrive? Und welche Probleme lösen sie nicht?

Versionsverwaltungssysteme (VCS)

- Versionsverwaltungssysteme (Version Control Systems) (VCS) verwalten Source Code.
- Eigenschaften:
  - Historie aller Änderungen
  - Vergleich (diff) verschiedener Stände
  - Abzweigen (branching) und Zusammenführen (merging)
  - Koordinierung des gemeinsamen Zugriffs auf Source Code
- Man unterscheidet
  - lokale Versionsverwaltungssysteme
  - zentrale Versionsverwaltungssysteme und
  - verteilte Versionsverwaltungssysteme.

Zentrale Versionsverwaltungssysteme



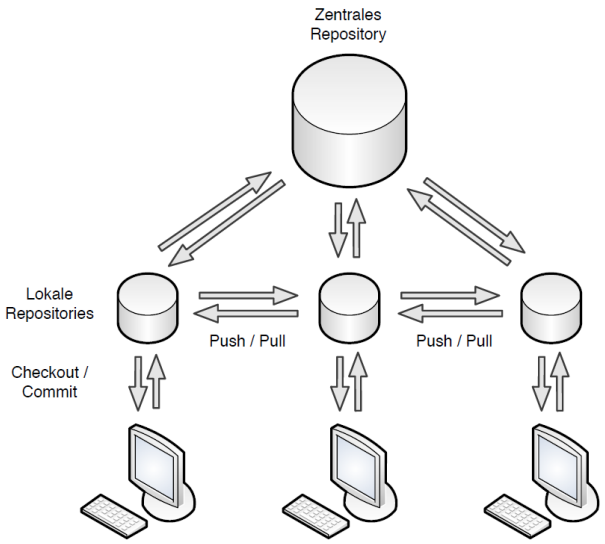
Zentrale Versionsverwaltungssysteme

- Open Source
  - CVS - Urvater, inzwischen veraltet
  - Subversion - das populärste zentrale System
- Closed Source
  - Visual SourceSafe - Veraltet, viele Probleme
  - Team Foundation Server - Nachfolger von Visual Source Safe von Microsoft
  - Perforce - Kostenlos für Open Source oder max. zwei Benutzer

Subversion

- Für alle Betriebssysteme verfügbar
  http://subversion.apache.org/
- Gute Eclipse-Integration
- Gute Windows-Tools wie TortoiseSVN
- Ein zentrales Repository wird benötigt: / muss aufgesetzt werden
  - Selbst betreiben
    z. B. mit http://www.visualsvn.com/server
  - Einen existierenden Hoster verwenden
    z. B. http://sourceforge.net
  - Der Zugriff auf einen gemeinsamen Fileshare ist nicht optimal!

Verteilte Versionsverwaltungssysteme



Verteilte Versionsverwaltungssysteme

- Open Source
  - GIT - Linux Kernel Community
  - Mercurial - Python Community
  - Bazaar - Ubuntu als Hauptnutzer
- Closed Source
  - Bitkeeper
  - ClearCase

Was genau ist GIT?

- Git (engl. Blödmann) ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die ursprünglich für die Quellcode-Verwaltung des Linux-Kernels entwickelt wurde.
- Eine Versionsverwaltung ist ein System, das zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird.
- Alle Versionen werden in einem Archiv mit Zeitstempel und Benutzerkennung gesichert und können später wiederhergestellt werden.

hochschule mannheim

Was ist ein Repository (Repo)?

- Ein Repository ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten.
- Bei den verwalteten Objekten kann es sich beispielsweise um
  - Programme (Software-Repository),
  - Publikationen (Dokumentenserver) oder
  - Datenmodelle (Metadaten-Repository)

  handeln.
- Häufig beinhaltet ein Repository auch Funktionen zur Versionsverwaltung der verwalteten Objekte.

hochschule mannheim

Was ist die Idee einer verteilten Versionsverwaltung?

- Die verteilte Versionsverwaltung verwendet kein zentrales Repository mehr.
- Jeder, der an dem verwalteten Projekt arbeitet, hat sein eigenes Repository und kann dieses mit jedem beliebigen anderen Repository abgleichen.
- Die Versionsgeschichte ist dadurch genauso verteilt.
- Änderungen können lokal verfolgt werden, ohne eine Verbindung zu einem Server aufbauen zu müssen.
- Im Gegensatz zur zentralen Versionsverwaltung kommt es hier nicht zu einem Konflikt, wenn mehrere Benutzer dieselbe Version einer Datei ändern.

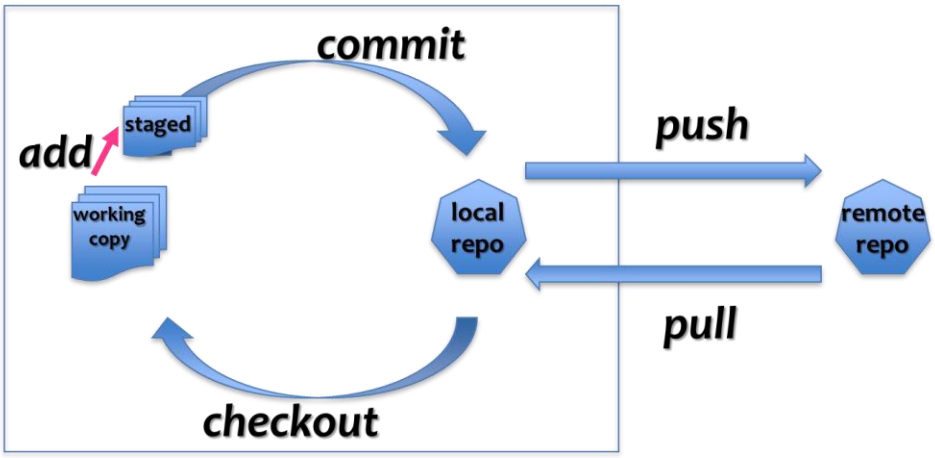
hochschule mannheim

Was ist die Idee einer verteilten Versionsverwaltung?

- Die sich widersprechenden Versionen existieren zunächst parallel und können weiter geändert werden. Sie können später in eine neue Version zusammengeführt werden.
- Systembedingt bieten verteilte Versionsverwaltungen keine Locks.
- Obwohl konzeptionell nicht unbedingt notwendig, existiert in verteilten Versionsverwaltungsszenarien üblicherweise ein offizielles Repository.
- Das offizielle Repository wird von neuen Projektbeteiligten zu Beginn ihrer Arbeit geklont, d.h. auf das lokale System kopiert.

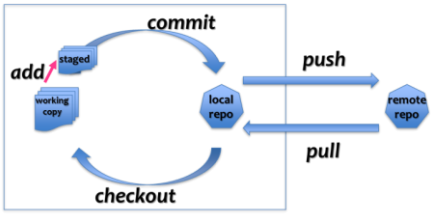
hochschule mannheim

Das Prinzip von GIT



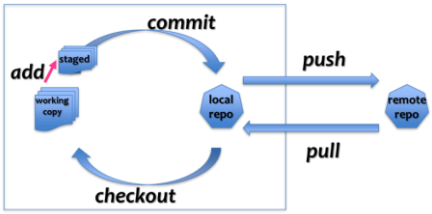
hochschule mannheim
Das Prinzip von GIT

- Als working copy werden die Dateien und Ordner bezeichnet, die auf meinem lokalen Rechner aktuell im Workspace sichtbar sind.
- Diese Dateien programmiert man in Eclipse.
- Im lokalen Repository verwaltet und speichert GIT alle „alten Versionen“ der Dateien auf dem eigenen Rechner.
- Wenn man eine Veränderung an einer Datei vorgenommen hat, kann man sie mit dem Befehl **`add`** in den Zustand **`staged`** versetzten.
- Das macht Eclipse automatisch beim Speichern.



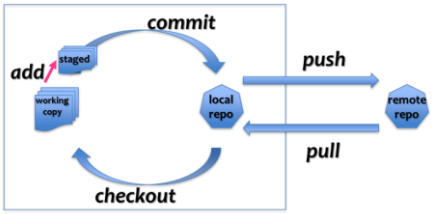
hochschule mannheim
Das Prinzip von GIT

- Mit dem Befehl **`commit`** werden die gestageden Dateien im lokalen Repository abgespeichert, und erhalten dort einen eindeutigen Hashwert und eine Meldung, die man vorher eingeben muss und die die Neuerungen dokumentiert.
- Mit dem Befehl **`push`** kann ich die commits aus meinem lokalen Repository in ein anderes, remote Repository übertragen.
- Darauf haben dann alle Team-Mitglieder Zugriff.

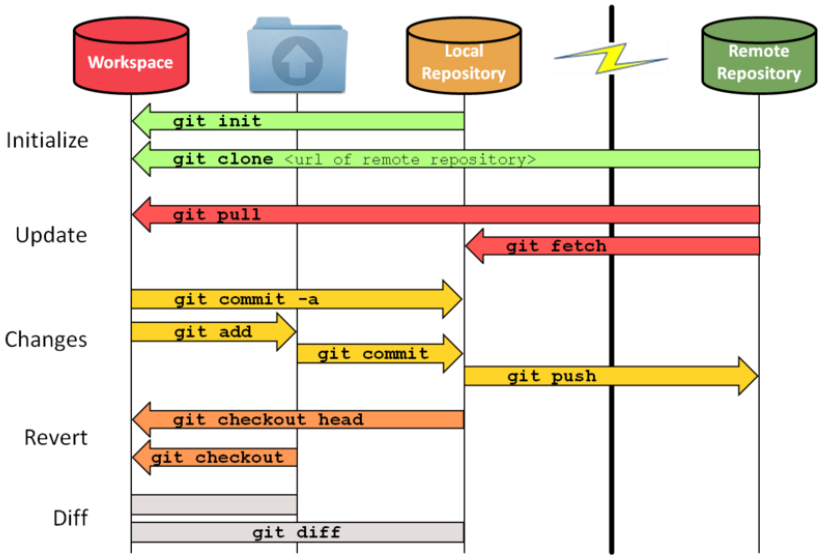


Das Prinzip von GIT

- Um Änderungen der anderen Teilnehmer auf seinen Rechner zu übertragen in die lokale Repository, muss man einen **`pull`** Befehl absetzen.
- Mit **`checkout`** werden dann die Dateien in den Workspace zur weiteren Bearbeitung geladen. In Eclipse wird bei dem pull Befehl der neue Inhalt des lokalen Repository automatisch in den Workspace kopiert.



Das Prinzip von GIT



hochschule mannheim
GIT bei den Übungen

- Die Verfügbarkeit des GIT-Servers und des Netzes kann bei einer Abgabe NICHT garantiert werden!
- Bei einer Abnahme müssen Sie stets die aktuelle Version auf Ihrem PC im lokalen Repository verfügbar haben, die mit der Remote Repository abgeglichen ist!
- Vermeiden Sie also last-minute Tätigkeiten!

hochschule mannheim
Grundregeln für verteilte Software-Entwicklung

- Sprechen Sie sich ab, wer welche Klassen erstellt, bearbeitet bzw. dafür zuständig ist!
- Eine Person = Eine Klasse
  Niemand editiert in den Klassen von anderen Teilnehmern!
- Wenn nur ein Teilnehmer der Gruppe bei GIT „irgendwo klickt, um irgendwie seinen Sourcecode einzupflegen“, so ist die Wahrscheinlichkeit sehr hoch, dass alle Teilnehmer massive Probleme erhalten!
- Geschieht der vorherige Punkt einige Stunden vor einer Deadline, so sorgt das in der Regel für große „Spannungen“ innerhalb einer Übungsgruppe!
- Verwendet ein Teammitglied eine andere Version von Java, Eclipse oder gar ein anderes Betriebssystem, so wird eine Zusammenarbeit erschwert!

Der Konfliktfall

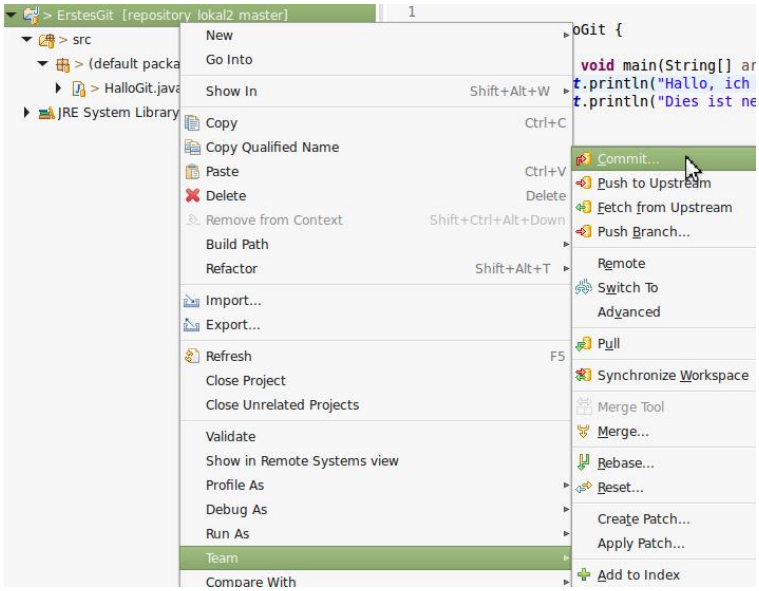
*HalloGit.java

```
1
2  public class HalloGit {
3
4   public static void main(String[] args) {
5     System.out.println("Hallo, ich bin ganz <Aenderung von Teilnehmer 1> neuer Code...");
6     System.out.println("Dies ist neuer Code vom 2. Teilnehmer der Praktikumsgruppe...");
7   }
8
9  }
10
11
```

HalloGit.java

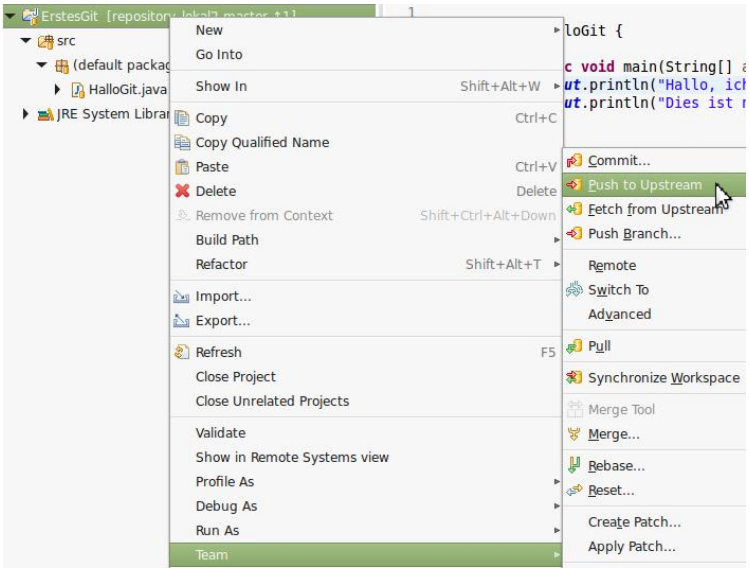
```
1
2  public class HalloGit {
3
4      public static void main(String[] args) {
5          System.out.println("Hallo, ich bin <Änderung von Teilnehmer 2> ganz neuer Code...");
6          System.out.println("Dies ist neuer Code vom 2. Teilnehmer der Praktikumsgruppe...");
7      }
8
9  }
```


Der Konfliktfall



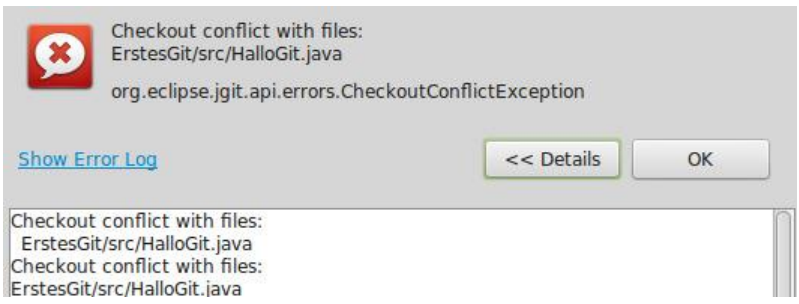
hochschule mannheim

Der Konfliktfall



hochschule mannheim

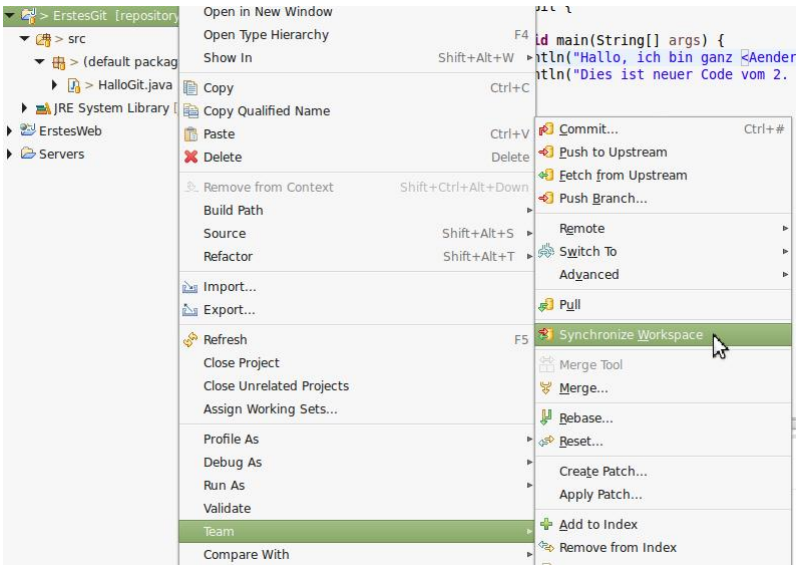
Der Konfliktfall: Mangelnde Teamfähigkeit...



...bedeutet bei GIT immer mehr Arbeit und mehr Risiken!

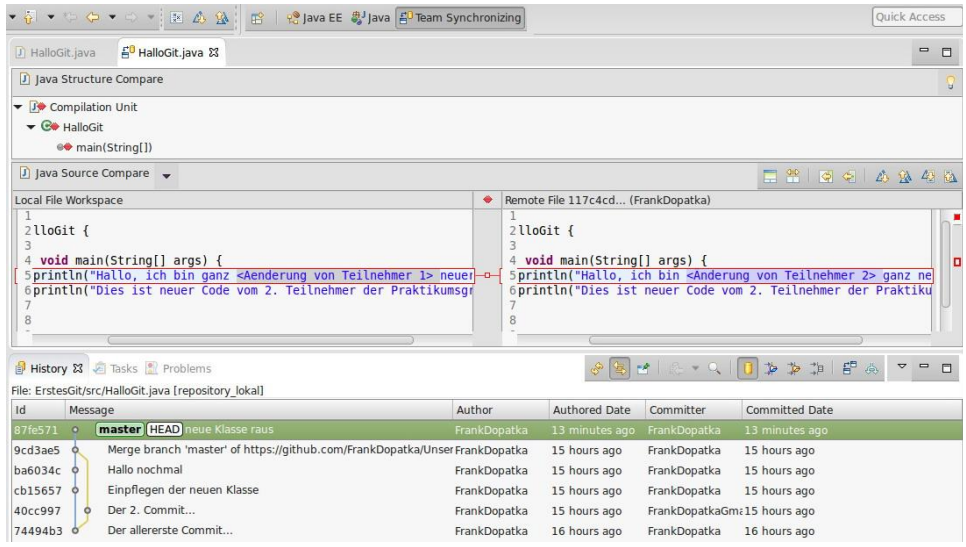
hochschule mannheim

Der Konfliktfall: Mangelnde Teamfähigkeit...



hochschule mannheim

Der Konfliktfall: Mangelnde Teamfähigkeit...



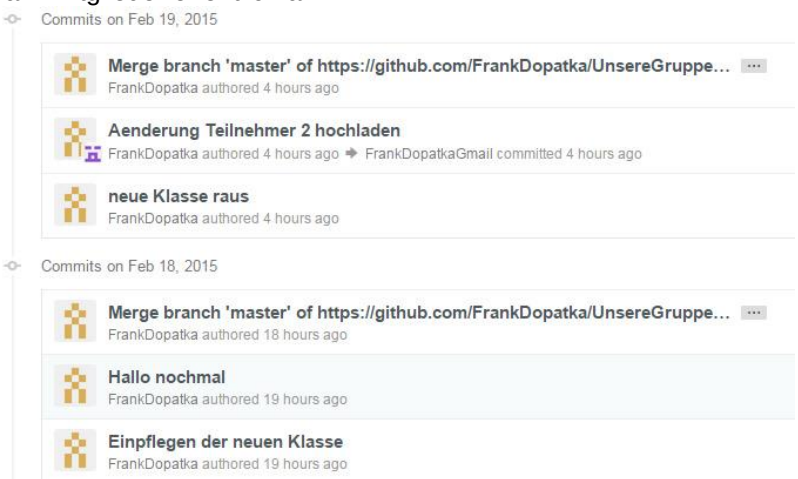
... macht ein Merging nötig!

hochschule mannheim
Der Konfliktfall: Mangelnde Teamfähigkeit...

... macht ein Merging nötig!

hochschule mannheim
Aktivitäten der Teilnehmer

- Die Repository zeigt die Historie und die Aktivitäten der Team-Mitglieder öffentlich an...



Aktivitäten der Teilnehmer

- ...und mangelnde Aktivität zieht das Interesse des Betreuers bei der Abnahme an und fördert das Nachfragen ;-)